

# Abschnitt 1alg

17. Juni 2000

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme des folgenden sind vorbehalten:

*Studenten dürfen für ihre persönlichen Lernzwecke dieses Dokument ausdrucken und auf ihren Rechnern Kopien der entsprechenden Datei vorhalten.*

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken und Dateikopien; insbesondere darf dieses Skriptum nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Datei auf irgendwelchen „Skriptenservern“ angelegt werden. Gegen die Anlage von Verweisen („hypertext links“) auf diese Datei ist selbstverständlich nichts einzuwenden.

Copyright © Herbert Klaeren, 1999.

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>9</b>
1.1	Was ist Informatik? . . . . .	9
1.2	Geschichte der Programmierung . . . . .	17
<b>2</b>	<b>Induktive Definitionen</b>	<b>21</b>
2.1	Natürliche Zahlen . . . . .	21
2.2	Wortmengen . . . . .	28
2.3	Syntaktische Beschreibungsmittel . . . . .	32
2.4	Terme . . . . .	40
2.5	Aufgaben . . . . .	47
<b>3</b>	<b>Algorithmen und Programme</b>	<b>49</b>
3.1	Algorithmus . . . . .	49
3.2	Programme . . . . .	54
3.2.1	Ausdrücke, Namen . . . . .	55
3.2.2	Fallunterscheidungen . . . . .	59
3.2.3	Rekursive Definitionen . . . . .	60
3.3	Aufgaben . . . . .	62
<b>4</b>	<b>Abstrakte Datentypen</b>	<b>93</b>
4.1	Einführung . . . . .	93
4.2	Boolesche Werte . . . . .	95
4.3	Zähler . . . . .	98
4.4	Listen . . . . .	101
4.5	Bäume . . . . .	108
4.6	Aufgaben . . . . .	116
<b>5</b>	<b>Logische Kalküle</b>	<b>117</b>
5.1	Ein Kalkül für die Aussagenlogik . . . . .	118
5.2	Ein Kalkül für die Prädikatenlogik . . . . .	122
5.3	Der Reduktionskalkül $RC_1$ . . . . .	125
5.4	Der $\lambda$ -Kalkül . . . . .	127
<b>A</b>	<b>Mathematische Grundlagen</b>	<b>119</b>
A.1	Mengen, Relationen, Abbildungen . . . . .	119

A.2	Formale Logik . . . . .	124
	A.2.1 Aussagenlogik . . . . .	124
	A.2.2 Prädikatenlogik . . . . .	126
A.3	Halbordnungen . . . . .	128
A.4	Aufgaben . . . . .	129

## Kapitel 3

# Algorithmen und Programme

Im Einführungskapitel haben wir festgestellt, daß Informationsverarbeitungsprozesse außer einem Computer auch noch *Daten* und ein *Programm* brauchen. Im vergangenen Kapitel haben wir uns mit der Beschreibung von Daten beschäftigt, wobei diese Thematik keinesfalls bereits erschöpft ist. Jetzt wenden wir uns zunächst jedoch den Programmen zu.

Grundlage für ein Programm ist immer ein *Verfahren*, welches präzise und bis in alle Einzelheiten beschreibt, wie sich das Problem mechanisch lösen läßt. Da der Computer mit unerwarteten Situationen nicht umgehen kann, muß jede eventuell bei der Lösung auftretende Schwierigkeit im Voraus bedacht sein und entsprechend behandelt werden. Ein solches Verfahren nennt man üblicherweise einen *Algorithmus*.

### 3.1 Algorithmus

Der Begriff „Algorithmus“ ist wesentlich älter als die Computer. Bereits Euklids „Elemente“ (3. Jhdt. v.Chr.) ist eine Sammlung von Algorithmen. Das Wort „Algorithmus“ kommt trotzdem nicht von den Griechen, sondern ist von dem Namen des Mathematikers MOHAMMED IBN MUSA ABU DJAFAR AL KHOWARIZMI (ca. 783–850, auch al Khwarizmi, al Choresmi u.a.) aus Choresmien im heutigen Usbekistan abgeleitet, der um 800 in Bagdad in dem von dem Kalifen HARUN AL RASCHID gegründeten „Haus der Weisheit“ zusammen mit anderen Wissenschaftlern Übersetzungen der griechischen mathematischen und medizinischen Schriften ins Arabische anfertigte und auf dieser Basis selbst weiter forschte. Er schrieb ein weit verbreitetes Buch mit dem arabischen Titel „*Kitab al muhtasar fi hisab al gebr we al muqabala*“ („Kurzgefaßtes Lehrbuch für die Berechnung durch Vergleich und Reduktion“), das bereits Lösungen von Gleichungen mit mehreren Unbekannten behandelte, und hat damit außer dem „Algorithmus“ auch das Wort „Algebra“ geprägt. In der lateinischen Übersetzung dieses Buchs, das

durch die Kreuzfahrer nach Europa kam, begann das Buch mit „*Dixit algorismi:*“ („So sprach al Khowarizmi:“), woraus sich die Bezeichnung Algorismus (später auch Algoritmus, Algorithmus) für eine Rechenvorschrift ableitete.

Zur Zeit von Adam RIESE[Rie91] galten Rechenaufgaben wie etwa Verdoppeln, Halbieren, Multiplizieren von Dezimalzahlen als so schwierig, daß hierfür ausdrücklich Algorithmen formuliert und gelehrt wurden. Hierzu ein Beispiel in der Formulierung von Riese:

**Dupliren** *Lert wie du eine zahl zwifeltigen solt/ thu yhm also/ schreib die zal vor dich/mach eine linihen darunder/ heb an zu forderst/ duplir die erste figur kömet eine zal die du mit einer figur schreiben magst/ so setz sie unden/ wo mit zweien schreib die erst/ die ander behalt im sin/ darnach duplir die ander und gib darzu das du behalten hast/ und schreib abermals die erst figur wo zwo vorhanden/ und duplir fort bis zur letzten/ die schreib gantz aus/ als folgende exempeln ausweisen.*

$$\begin{array}{r} 41232 \quad 98765 \quad 68704 \\ \hline 82464 \quad 197530 \quad 137408 \end{array}$$

Natürlich lernen auch heute noch Grundschüler die Algorithmen zur Multiplikation und Division von Dezimalzahlen, auch wenn sie nicht ausdrücklich Algorithmen genannt werden.

Ein weiteres Beispiel soll die Grundeigenschaften eines Algorithmus demonstrieren. Es ist dies der *Euklidische Algorithmus* zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen, der bei Euklid einfach so formuliert war:

*Wenn man wechselweise immer die kleinere von der größeren abzieht, wird eine Zahl übrigbleiben, welche die vorangehende teilt.*

und der in moderner sprachlicher Formulierung etwa so lautet:

*Euklidischer Algorithmus*

**Aufgabe:** Seien  $a$  und  $b$  natürliche Zahlen,  $a, b \neq 0$ . Bestimme den größten gemeinsamen Teiler  $g \stackrel{\text{def}}{=} \text{ggT}(a, b)$  von  $a$  und  $b$ .

**Verfahren:** Unterscheide die folgenden drei Fälle:

- $a = b$ : Setze  $g = a$ .
- $a > b$ : Berechne  $g = \text{ggT}(a - b, b)$ .
- $a < b$ : Berechne  $g = \text{ggT}(a, b - a)$ .

Gegenüber Euklids Formulierung ist dies deutlich genauer, weist aber das Problem auf, daß es sich um eine *rekursive* Formulierung handelt, bei der das gleiche Problem auf einer inneren Stufe wieder auftaucht. Im Abschnitt über induktive Definitionen wurde gezeigt, daß solche rekursiven Definitionen durchaus sinnvoll sein können. Dabei handelte es sich allerdings um eine spezielle Form der Rekursion, nämlich die strukturelle Rekursion entsprechend der Struktur der induktiv erzeugten Daten. Im obigen Beispiel ist eine Fallunterscheidung nötig, um zu zeigen, daß die Rekursion endet: Wenn  $a, b > 0$  sind, so wird im zweiten Fall  $a - b < a$  und im dritten Fall  $b - a < b$  sein. So lange  $a \neq b$  gilt, wird außerdem  $a \neq 0 \neq b$  sein, d.h. bei den rekursiven Aufrufen von  $\text{ggT}$  wird immer mindestens ein Argument kleiner, und zwar so lange, bis beide Argumente gleich sind. Im schlimmsten Fall ist dies für  $a = 1 = b$  der Fall; dann sind die beiden Zahlen teilerfremd.

Die Mathematik hat sich lange Zeit nicht intensiv mit dem Algorithmusbegriff auseinandergesetzt, zumal die allgemeine Auffassung herrschte, daß es zu jedem mathematisch präzise formulierten Problem auch einen Algorithmus zu seiner Lösung gab. Resultate wie etwa der Beweis der Unmöglichkeit der Quadratur des Kreises mit Zirkel und Lineal widersprechen dieser Grundaffassung nicht, da hier lediglich gezeigt wird, daß eine bestimmte Aufgabe mit bestimmten eingeschränkten Hilfsmitteln algorithmisch nicht gelöst werden kann. Die prinzipielle Lösbarkeit durch Algorithmen mit stärkeren Hilfsmitteln bleibt jedoch unberührt.

Erst die zu Beginn dieses Jahrhunderts begonnene Axiomatisierung der Mathematik führte im Bereich der mathematischen Logik zu einer genaueren Beschäftigung mit dem Algorithmusbegriff. Diese Forschungen begannen in den Dreißiger Jahren, also vor der Computerzeit. Es ergab sich die überraschende Erkenntnis, daß sich bestimmte Probleme *prinzipiell* nicht algorithmisch lösen lassen, wie etwa das von D. HILBERT aufgeworfene sogenannte Entscheidungsproblem der Prädikatenlogik oder das von A. THUE behandelte Wortproblem der Gruppentheorie.

Aussagen über die algorithmische Unlösbarkeit setzen selbstverständ-

lich eine allgemein anerkannte mathematisch präzise Formulierung des Algorithmusbegriffs voraus. Eine erste solche Präzisierung enthält implizit der *Unvollständigkeitssatz der Arithmetik* von K. GÖDEL (1931), der als erster Unlösbarkeitsbeweis seinerzeit großes Aufsehen erregte. Die Aussage dieses Satzes ist, daß es für jedes formale System zur Beschreibung der Arithmetik (auf natürlichen Zahlen) wahre Aussagen über Zahlen gibt, die sich innerhalb des Systems nicht beweisen lassen.

In der Informatik beschäftigen wir uns natürlich hauptsächlich mit Problemen, die sich algorithmisch lösen lassen, weil das der Bereich ist, in dem der Computer eingesetzt werden kann. Es sei jedoch bereits hier als Warnung vermerkt, daß es viele Probleme gibt, die im Prinzip algorithmisch lösbar sind, aber von der praktischen Seite als unlösbar betrachtet werden müssen, weil eine Berechnung nach den Algorithmen länger dauerte, als irgendjemand zu warten bereit oder in der Lage ist.

Es ist viel geistige Energie darauf verwendet worden, zwischen *Algorithmus* und *Programm* feinsinnige Unterscheidungen zu treffen. Meist ist ein Algorithmus etwas allgemeineres, das noch nicht in einer bestimmten Programmiersprache abgefaßt ist. Wir behaupten, daß sich diese Unterscheidung im Prinzip nicht lohnt. Zwar gibt es Möglichkeiten, Algorithmen außerhalb jeder Programmiersprache und ohne jeden Bezug zum Computer zu formulieren, aber alle Programme sind spezielle Erscheinungsformen von Algorithmen, weil die Programmiersprachen letztlich nur als formale Werkzeuge zur Notation von Algorithmen erfunden worden sind. In diesem Buch ist „Algorithmus“ lediglich ein Oberbegriff zu „Programm“.

Als Grundprämisse wollen wir davon ausgehen, daß unser Problem durch die Berechnung einer *Funktion* im mathematischen Sinne gelöst werden kann; das heißt, daß der Algorithmus aus einem oder mehreren *Parametern* einen eindeutig bestimmten Wert abzuleiten hat. (In Wirklichkeit berechnen nicht alle Programme berechnen Funktionen im mathematischen Sinne, aber alle enthalten sie Teilaufgaben, die sich durch den Funktionsbegriff beschreiben lassen.)

**Definition 3.1 (Algorithmus)** Ein Algorithmus ist eine Menge von Regeln für ein Verfahren, um aus gewissen *Eingabegrößen* bestimmte *Ausgabegrößen* herzuleiten, wobei die folgenden Bedingungen erfüllt sein müssen:

**Finitheit der Beschreibung:** Das vollständige Verfahren muß in einem endlichen Text beschrieben sein. Die elementaren Bestandteile der Be-

schreibung heißen *Schritte*.

**Effektivität:** Jeder einzelne Schritt des Verfahrens muß tatsächlich ausführbar sein.

**Terminiertheit:** Das Verfahren kommt in endlich vielen Schritten zu einem Ende.

**Determiniertheit:** Der Ablauf des Verfahrens ist zu jedem Punkt fest vorgeschrieben.

Zu dieser Definition sind einige Anmerkungen angebracht:

1. Es erscheint zuerst banal, zu verlangen, daß das Verfahren durch einen endlichen Text beschrieben sein muß, da niemand unendliche Texte aufschreiben kann. Manchmal kommen unendliche Texte aber „in Verkleidung“ vor, wie zum Beispiel in

$$\text{Berechne } 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \dots$$

Derartige Vorschriften gehören nicht in einen Algorithmus. Offensichtlich läßt sich die Forderung nach Finitheit nur erfüllen, wenn die Objekte, mit denen der Algorithmus operiert, eine endliche Darstellung besitzen. So gibt es strenggenommen keinen Algorithmus für die Addition reeller Zahlen in Dezimalbruch-Darstellung. Da wir in der Praxis sowieso stets mit *endlichen Approximationen* solcher unendlichen Objekte arbeiten, können wir trotzdem wieder sinnvoll von Algorithmen sprechen.

2. *Effektivität* (prinzipielle Machbarkeit) darf nicht mit *Effizienz* (wirtschaftlich vernünftige Machbarkeit) verwechselt werden. Ein Beispiel für einen nicht *effektiven* Rechenschritt wäre etwa:

*Falls die Dezimalbruchentwicklung von  $x$  nur endlich oft die Ziffer 3 enthält, ist das Ergebnis 5.*

Die hier angegebene Bedingung läßt sich z.B. für  $x = \pi$  nicht überprüfen. Ein nicht *effizienter* Rechenschritt demgegenüber wäre:

*Berechne alle möglichen Fortsetzungen des gegebenen Schachspiels jeweils bis zum Spielende und suche danach den besten Folgezug aus.*

Dieser Rechenschritt ist effektiv durchführbar, weil sich in einem Spiel keine Stellung wiederholen darf und es nur endliche viele davon gibt. Allerdings ist die Zahl der möglichen Fortsetzungen normalerweise so groß, daß wir sie weder alle (zum notwendigen Vergleich) speichern noch innerhalb eines Menschenlebens berechnen können.

3. Manchmal wird die Forderung nach Terminiertheit fallengelassen, um eine größere Klasse von Problemen behandeln zu können. Bei diesen Algorithmen kann die Berechnung für manche Eingaben abbrechen, für andere ewig weiterlaufen.
4. Auch von der Forderung nach Determiniertheit wird manchmal abgesehen, was bedeutet, daß an manchen Stellen des Verfahrens eine Wahlmöglichkeit besteht, die nicht durch feste Regeln abgefangen wird.

Die Theoretische Informatik beschäftigt sich unter anderem mit der Klasse aller algorithmisch lösbaren Probleme und deren Komplexität sowie mit der algorithmischen Unlösbarkeit.

Hier interessieren wir uns mehr dafür, wie und nach welchen Regeln wir von einem Problem zu einer algorithmischen Lösung kommen. Wir werden feststellen, daß es sogenannte *Paradigmata* („Algorithmen-Grundmuster“) gibt, die in ganzen Klassen von Problemen zu einer Lösung führen.

## 3.2 Programme

Zur Notation von Programmen ist zunächst einmal eine bestimmte *Sprache* vonnöten. Hier wird es diese Sprache in zwei Erscheinungsformen geben, eine von der mathematischen Notation beeinflusste „kalligraphische“ Variante (jeweils in der linken Hälfte der Seite) und eine für die Maschine lesbare Variante (in der rechten Hälfte der Seite). Eine solche Aufspaltung ist erstmals bei der Programmiersprache ALGOL 60 vorgenommen worden. Die eigentliche Sprache ist in beiden Erscheinungsformen die gleiche.

Die von uns verwendete Sprache ist *Scheme* [KCR98]; als Programmiersystem hierzu empfehlen wir *DrScheme*. Das Programmiersystem basiert auf einem *interaktiven Paradigma*: Die Verwendung eines Rechners entspricht einem „Frage-Antwort-Spiel“, ähnlich wie bei einem Taschenrechner. Der Rechner wartet auf eine Eingabe in Form eines Terms; wenn diese Eingabe vorliegt, wertet der Rechner den Term aus und gibt seinen Wert als Antwort aus, anschließend wartet er wieder auf eine Eingabe. Auf Englisch heißt dieses Vorgehen des Rechners „*read-eval-print loop*“, kurz: REPL. Einen einfachen Ablauf der REPL notieren wir als

```
a
>>> b
```

notiert, was bedeutet, daß der Rechner auf die „Frage“ a mit dem Wert b antwortet.

### 3.2.1 Ausdrücke, Namen

Die einfachste Leistung der REPL ist die Auswertung von Ausdrücken, die in Scheme in einer vollständig geklammerten Präfixnotation aufgeschrieben werden müssen und in der Fachterminologie *Formen* heißen.

<pre>23 + 42 &gt;&gt;&gt; 65 (7 + 4)(9 - 3) &gt;&gt;&gt; 66</pre>		<pre>(+ 23 42) &gt;&gt;&gt; 65 (* (+ 7 4) (- 9 3)) &gt;&gt;&gt; 66</pre>
---	--	--

Die Zeichenfolgen „23“, „42“ etc. heißen in der Fachterminologie *Literale*; sie bezeichnen ohne jeden Umweg bestimmte (konstante) Werte.

Sinnvoll werden Ausdrücke erst, wenn wir *Namen* für Ausdrücke einführen können und diese Namen dann stellvertretend für die Werte der entsprechenden Ausdrücke verwendet werden können:

<pre><math>\pi</math> <math>\stackrel{\text{def}}{=} 3.1415926</math> r <math>\stackrel{\text{def}}{=} 4</math> <math>2\pi r</math> &gt;&gt;&gt; 25.132741</pre>		<pre>(define pi 3.1415926) (define radius 4) (* 2 pi radius) &gt;&gt;&gt; 25.132741</pre>
--	--	---

Mathematiker verwenden für Konstanten und Variable gerne einzelne lateinische oder griechische Buchstaben, hauptsächlich deshalb, weil sie Multiplikationszeichen einsparen möchten: Eine Variable  $bla$  könnte dann mit dem Produkt  $b \cdot l \cdot a$  verwechselt werden. Unter Programmierern sind einzelne Buchstaben als Konstanten- oder Variablenbezeichner regelrecht verpönt; sie wählen lieber aussagekräftige Bezeichner wie z.B.  $radius$  im obigen Beispiel. Wir werden uns hier auch in der mathematischen Notation meistens den „Luxus“ aussagekräftiger Bezeichner leisten.

Ein Vorteil der *Scheme*-Syntax ist im übrigen, daß sich die Multiplikation (wie auch andere sonst binäre Operationen) auch auf mehr als zwei Argumente anwenden läßt.

Auf *Definitionen* antwortet die REPL nicht mit der Ausgabe eines Werts. Definitionen haben nämlich auch keinen eigenen Wert, sondern sie verändern den internen Zustand der REPL in der Weise, daß an den neuen Namen nunmehr der dahinter folgende Wert *gebunden* wird.

Das Vergeben von Namen ist eine erste schwache Form der bereits angesprochenen *Abstraktion*: Nachdem der Programmierer einmal einen Namen für, beispielsweise,  $\pi$  vergeben hat, kann er anschließend den Wert vergessen und statt dessen die abstrakte Bezeichnung  $\pi$  verwenden.

In der Programmiersprache *Scheme* sind fast alle Kombinationen von Zeichen gültige Bezeichner; die *Scheme*-Konventionen setzen sich damit von vielen anderen Programmiersprachen ab, in denen im wesentlichen nur Folgen aus Buchstaben und Ziffern als Bezeichner verwendet werden können.

Offensichtlich wird das Abstraktionsprinzip erst dann voll wirksam, wenn auch *Funktionen* durch Ausdrücke bezeichnet werden können. Der Logiker A. CHURCH hat dafür die sogenannten  $\lambda$ -Ausdrücke eingeführt; wir werden uns in einem späteren Abschnitt (5.4) noch damit beschäftigen.

$C \stackrel{\text{def}}{=} \lambda r. 2\pi r$  $A \stackrel{\text{def}}{=} \lambda r. \pi r^2$  $C(4)$ $\ggg 25.132741$ $A(4)$ $\ggg 50.26548160$	<pre>(define circumference   (lambda (radius)     (* 2 pi radius))) (define area   (lambda (radius)     (* pi radius radius))) (circumference 4) &gt;&gt;&gt; 25.132741 (area 4) &gt;&gt;&gt; 50.26548160</pre>
---	---

Die Formel  $\lambda r. 2\pi r$ , die auch  $\lambda$ -Abstraktion genannt wird, bedeutet dabei: „dies ist eine Funktion, die ein Argument erwartet; wenn dieses Argument  $r$  ist, dann ist der Wert der Funktion  $2\pi r$ “. Der Ausdruck  $2\pi r$  heißt dabei auch der *Rumpf* der Abstraktion.  $r$  heißt *Parameter* der Abstraktion. Der Punkt ist in der mathematischen Notation nötig, um mehrere mögliche Argumente vom Rumpf der Abstraktion abzutrennen (s. nachfolgendes Beispiel); in *Scheme* erledigen Klammern diese Aufgabe:

$P \stackrel{\text{def}}{=} \lambda mn. \frac{m-2n}{2}$  $P(14,5)$ $\ggg 2$	<pre>(define parking-lot-cars   (lambda (number-of-wheels           number-of-vehicles)     (/ (- number-of-wheels         (* 2 number-of-vehicles))        2))) (parking-lot-cars 14 5) &gt;&gt;&gt; 2</pre>
--	---

Programmiersprachen, welche die vollen Möglichkeiten der  $\lambda$ -Abstraktion abbilden, nennt man *funktionale Programmiersprachen*, unter anderem deshalb, weil hier Funktionen in vollem Umfang als *Werte* behandelt werden können. Mit anderen Worten: auch Ausdrücke wie

$\lambda r. 2\pi r$  | `(lambda (r) (* pi r r))`

haben einen Wert; wir nennen diesen Wert eine *Prozedur*. Ausdrücke der Form

$$A(B, C) \quad | \quad (A \ B \ C)$$

nennen wir *Applikationen* (Anwendungen), und zwar deshalb, weil wir uns hier das erste Element ( $A$ ) als *Operator* vorstellen, der auf die *Operanden* ( $B$  und  $C$ ) angewendet werden soll.

An dieser Stelle wollen wir erstmalig zu verstehen versuchen, wie die REPL funktioniert. Dazu eignet sich an dieser Stelle noch das sogenannte *Substitutionsmodell*, das wir induktiv wie folgt beschreiben:

**Definition 3.2 (Auswertung einer Form)** Gegeben sei eine Form  $x$ ; zu ermitteln ist der Wert von  $x$ . Dieser Wert wird wie folgt bestimmt:

1.  $x$  ist Literal: Der Wert von  $x$  ist der Wert des Literals.
2.  $x$  ist Name: Wenn an  $x$  durch eine Definition ein Wert gebunden wurde, dann ist der Wert von  $x$  der an  $x$  gebundene Wert. Anderenfalls liegt ein Fehler vor; es gibt dann keinen Wert von  $x$ .
3.  $x = (r \ n_1 \ \dots \ n_m)$  ist Applikation: Ermittle (rekursiv) die Werte von  $r$  und  $n_1, \dots, n_m$ . Wenn der Wert von  $r$  eine Prozedur  $P$  mit Parametern  $p_1, \dots, p_m$  ist, dann ersetze („substituiere“) im Rumpf von  $P$  den Namen  $p_1$  durch den Wert von  $n_1$  usf. bis zur Ersetzung von  $p_m$  durch den Wert von  $n_m$  und ermittle anschließend den Wert des solchermaßen modifizierten Rumpfs von  $P$ ; dieser ist der Wert von  $x$ . Ist der Wert von  $r$  keine Prozedur oder hat sie eine unpassende Anzahl von Parametern, so liegt ein Fehler vor; es gibt dann keinen Wert von  $x$ .

Bei der letzten Regel kann es scheinbar zu Mehrdeutigkeiten kommen; wie soll etwa

$$\lambda x. (\lambda x. x + 1)(x + 2)13 \quad | \quad \begin{array}{l} ((\text{lambda } (x) \\ (\text{lambda } (x) (+ x 1)) (+ x 2))) \\ 13) \end{array}$$

ausgewertet werden? Das Problem hierbei ist offensichtlich die Zuordnung der Vorkommen von  $x$  in dem Rumpfen zu den entsprechenden  $\lambda$ -Abstraktionen. Hier gilt das Prinzip der *lexikalischen Bindung*: Ein Name gehört immer zu der Abstraktion, die ihm, von innen nach außen gesucht, am nächsten liegt. Der obige Ausdruck ist somit gleichbedeutend zu dem Ausdruck

$$(\lambda x. (\lambda y. y + 1)(x + 2))13 \quad \Bigg| \quad \begin{array}{l} ((\text{lambda } (x) \\ \quad ((\text{lambda } (y) (+ y 1)) (+ x 2))) \\ \quad 13) \end{array}$$

bei dem es keine Mehrdeutigkeiten gibt.

### 3.2.2 Fallunterscheidungen

Mit Ausdrücken der bisher studierten Form lassen sich keine sehr interessanten Programme schreiben; der Ablauf ist jeweils vollständig determiniert und für alle Auswertungen, abgesehen von möglicherweise unterschiedlichen Datenelementen, gleich. Das mindeste, was wir für die Konstruktion nicht-trivialer Programme benötigen, sind *Fallunterscheidungen*, wie im folgenden Beispiel:

$$|x| \stackrel{\text{def}}{=} \lambda x. \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{sonst} \end{cases} \quad \Bigg| \quad \begin{array}{l} (\text{define abs} \\ \quad (\text{lambda } (x) \\ \quad \quad (\text{if } (>= x 0) \\ \quad \quad \quad x \\ \quad \quad \quad (- x)))) \end{array}$$

In der mathematischen Tradition verwendet man für Fallunterscheidungen gerne geschweifte Klammern, in *Scheme* ist dafür die *if*-Kombination (*conditional*) vorgesehen. Für den Augenblick wollen wir uns auf Fallunterscheidungen der oben aufgeführten Form beschränken, bei der ein Zweig durch einen „Test“, d.h. durch einen booleschen Ausdruck („ $x \geq 0$ “), und der andere Zweig nur mit „sonst“ dekoriert ist, d.h. also alle anderen Fälle abdeckt. Der durch den Test bewehrte Zweig nennt sich *Konsequente*, der andere *Alternative*.

Selbstverständlich ist ein Test für sich auch ein gültiger Ausdruck, den wir auswerten können:

$$\begin{array}{l} 3 \geq 1 \\ \gg \top \\ 1 \geq 3 \\ \gg \perp \end{array} \quad \Bigg| \quad \begin{array}{l} (>= 3 1) \\ \gg \#t \\ (>= 1 3) \\ \gg \#f \end{array}$$

Damit haben wir gleichzeitig gesehen, wie die Literale für die Wahrheitswerte in *Scheme* aussehen. In *Scheme* ist es selbstverständlich, daß *con-*

*ditionals* an jeder Stelle eines Ausdrucks auch eingeschachtelt vorkommen können. Da sich ähnliches in der mathematischen Notation mit den geschweiften Klammern nur schwer nachvollziehen läßt, gehen wir von dieser Notation ganz ab und verwenden statt dessen **if-then-else**:

$$|x| \stackrel{\text{def}}{=} \lambda x. (\text{if } x \geq 0 \text{ then } + \text{ else } -)x \quad \left| \begin{array}{l} (\text{define abs} \\ \quad (\text{lambda } (x) \\ \quad \quad (\text{if } (>= \ x \ 0) \\ \quad \quad \quad + \\ \quad \quad \quad -) \\ \quad \quad x)) \end{array} \right.$$

Die Vorschrift in Definition 3.2 zur Auswertung einer Form müssen wir jetzt ergänzen um

**Definition 3.3 (Auswertung von if-Klauseln)** Zu den Vorschriften von Definition 3.2 tritt folgende hinzu:

4.  $x = \text{if } t \text{ then } c \text{ else } a$ : Ermittle den Wert von  $t$ . Wenn der Wert  $\top$  ist, dann ist der Wert von  $x$  der Wert von  $c$ , sonst der Wert von  $a$ .

### 3.2.3 Rekursive Definitionen

Der Hauptwert der Fallunterscheidung liegt darin, daß es mit ihrer Hilfe möglich ist, rekursive Funktionsdefinitionen aufzuschreiben. Bei den rekursiven Funktionsdefinitionen in Kapitel 2 haben wir die Fallunterscheidungen auf den linken Seiten von Gleichungssystemen vorgenommen; die praktische Anwendung einer solchen Definition beruht auf einem Prozeß der Auswahl eines passenden Musters (*pattern matching*) und ist für die Maschine nicht so geradlinig zu befolgen wie Definitionen der folgenden Gestalt:

$$n! \stackrel{\text{def}}{=} \lambda n. \text{if } x \geq 1 \text{ then } 1 \quad \left| \begin{array}{l} (\text{define factorial} \\ \quad (\text{lambda } (n) \\ \quad \quad (\text{if } (= \ n \ 1) \\ \quad \quad \quad 1 \\ \quad \quad \quad (* \ n \\ \quad \quad \quad \quad (\text{factorial } (- \ n \ 1)))))) \end{array} \right.$$

An dieser Stelle wollen wir uns nochmals vergewärtigen, wie die Auswertung einer Form funktioniert. Die Zwischenschritte listen wir dabei in jeweils einer neuen Zeile auf; die REPL wird nur das letzte Element dieser Folge (hinter „ $\ggg$ “) ausgeben. In der *Scheme*-Version kürzen wir den Rumpf von `factorial` meist durch „...“ ab.

<pre>4! (<math>\lambda n</math>.if n = 1 then 1 else n(n - 1)!)4 if 4 = 1 then 1 else 4(4 - 1)!  4 · 3! 4 · (<math>\lambda n</math>. if n = 1 then 1       else n(n - 1)!)3) 4 · (if 3 = 1 then 1 else 3(3 - 1)!)  ... 4 · 3 · 2 · (if 1 = 1 then 1 else 1(1 - 1)!)  4 · 3 · 2 · 1 <math>\ggg</math> 24</pre>	<pre>(factorial 4) ((lambda (n) ... ) 4) (if (= 4 1) 1     (* 4       (factorial (- 4 1))     )) (* 4 (factorial 3)) (* 4 ((lambda (n) ... ) 3)) (* 4 (if (= 3 1) 1         (* 3           (factorial (- 3 1))         ))) ... (* 4 (* 3 (* 2 (if (= 1 1) 1                   (* 1                     (factorial (- 1 1))                   ))))) <math>\ggg</math>24</pre>
---	--

Wie bereits in Beispiel 2.6 erkennen wir auch hier, daß sich im Verlauf der Rechnung eine immer größere Kette unausgewerteter Ausdrücke ansammelt, die sich erst nach der Erledigung aller rekursiven Aufrufe auswerten lassen. Wir nennen solche unerledigten Aufgaben, die erst nach der Erledigung eines rekursiven Aufrufs bearbeitet werden können, den *Kontext* des rekursiven Aufrufs. Im Beispiel 2.6 haben wir gesehen, wie sich durch „Rechnen auf den Parametern“ das Ansammeln eines immer größeren Kontexts vermeiden läßt; ähnliches können wir unter Verwendung einer zweistelligen Hilfsfunktion auch bei der Fakultätsfunktion bewirken:

$n! \stackrel{\text{def}}{=} \lambda n. \text{fac2}(n, 1)$ $\text{fac2} \stackrel{\text{def}}{=} \lambda nr. \text{if } n = 1 \text{ then } r$ $\qquad \qquad \text{else fac2}(n - 1, n \cdot r)$	<pre> (define factorial   (lambda (n)     (fac2 (n 1)))) (define fac2   (lambda (n result)     (if (= n 1)         result         (fac2 (- n 1)               (* n result)))))) </pre>
---	--

Eine rekursive Definition der bei `fac2` vorliegenden Form heißt *endrekursiv* oder *iterativ*. Offensichtlich sind iterative Definitionen für eine Maschine leichter zu befolgen, da keine Kontexte entstehen und deshalb keine Verwaltung von Aufgaben erfolgen muß, die erst nach der Rückkehr von dem rekursiven Aufruf bearbeitet werden können.



Hier leider noch

### 3.3 Aufgaben

**Aufgabe 3.1** Beschreiben Sie den Algorithmus zur Multiplikation zweier Dezimalzahlen unter der Voraussetzung, daß zwei Dezimalzahlen  $a$  und  $b$  gegeben sind sowie eine Tabelle  $K(i, j)$  („kleines Einmaleins“) mit  $1 \leq i \leq 9$ ,  $1 \leq j \leq 9$ , wobei  $K(i, j) = i \cdot j$ . Begründen Sie anhand von Eigenschaften der Dezimalzahlen, warum dieser Algorithmus korrekt ist.

**Aufgabe 3.2** Für zwei Zeichenfolgen  $T = T_1 \dots T_t$  und  $M = M_1 \dots M_m$  soll entschieden werden, ob  $M$  innerhalb von  $T$  auftritt, und, falls ja, an welcher Stelle dies zum erstmaligen Fall ist. Geben Sie eine genaue Spezifikation und einen Algorithmus für dieses Problem an.

**Aufgabe 3.3** Entwerfen Sie einen Algorithmus „Lexikographische Ordnung“ für eine Liste von  $n$  Namen, die geordnet auszugeben sind. Bedienen Sie sich des Prinzips der schrittweisen Verfeinerung. Die Zerlegungsschritte

sind anzugeben und die Unteralgorithmen zu spezifizieren. Noch notwendige Festlegungen für die Spezifikation sind in angemessener Weise selbst zu treffen.

**Aufgabe 3.4** Spezifizieren Sie das folgende Problem und entwickeln Sie einen rekursiven Algorithmus zu seiner Berechnung:

Aus einer Folge von  $n$  Zahlen sei das Minimum und das Maximum zu ermitteln.

**Aufgabe 3.5** Bestimmen Sie für  $n = 7$  die Anzahl der notwendigen Vergleichsoperationen im Algorithmus aus der vorangehenden Aufgabe und geben Sie die Zerlegung der Eingabefolge in Teilfolgen an.

## Literaturverzeichnis

- [Bau68] BAUMANN, RICHARD: *Algol-Manual der Alcor-Gruppe*. R. Oldenbourg, München/Wien, 3. Auflage, 1968.
- [Bau89] BAUER, F. L.: *100 Jahre Peano-Zahlen*. Informatik-Spektrum, 12:340–341, 1989.
- [End72] ENDERTON, H. B.: *A Mathematical Introduction to Logic*. Academic Press, 1972.
- [GKP89] GRAHAM, R. L., D. E. KNUTH und O. PATASHNIK: *Concrete Mathematics*. Addison-Wesley, 1989.
- [Hal69] HALMOS, P.: *Naive Mengenlehre*. Vandenhoeck & Ruprecht, Göttingen, 1969.
- [KCR98] KELSEY, RICHARD, WILLIAM CLINGER und JONATHAN REES: *Revised<sup>5</sup> Report on the Algorithmic Language Scheme*. SIGPLAN Notices, 33(9):26–76, 1998.
- [Kla83] KLAEREN, HERBERT: *Algebraische Spezifikation — Eine Einführung*. Springer Verlag, Berlin-Heidelberg-New York, 1983.
- [Knu73] KNUTH, D. E.: *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973. Sorting and Searching.
- [Mes71] MESCHKOWSKI, H.: *Einführung in die moderne Mathematik*, Band 75/75a der Reihe *Hochschultaschenbücher*. BI, 1971.
- [MHR80] METROPOLIS, N., J. HOWLETT, and GIAN-CARLO ROTA (editors): *A History of Computing in the Twentieth Century*. Academic Press, 1980.
- [Rie91] RIESE, ADAM: *Rechnung auff der Linihen und Federn /Auff allerley handthirung gemacht / durch Adam Risen (Faksimile 2. Aufl. Erfurt 1532)*. Magistrat der Stadt Erfurt, 1991.
- [Wex81] WEXELBLAT, RICHARD L. (editor): *History of Programming Languages*, New York, 1981. Academic Press.