4 Vom Algorithmus zum Programm

4.1 Menschen und Maschinen

Unsere Absicht ist es, Maschinen statt Menschen mit der Ausführung von Algorithmen (speziell unseres Verteilungsalgorithmus) zu beauftragen. Überlegen wir zunächst, was eine "ideale" Maschine vom Menschen unterscheidet:

- Ideale Maschinen sind sehr schnell und vor allem unermüdlich.
- Ideale Maschinen arbeiten absolut zuverlässig. Eine Handlung wird beim millionsten Male noch genauso exakt durchgeführt wie beim ersten Male.
- Ideale Maschinen sind unbedingt gehorsam. Sie tun genau das, was man ihnen befiehlt; sie tun nichts, was man ihnen nicht befiehlt.

Die letzte Eigenschaft bedeutet insbesondere, daß sie jede Handlung ohne irgendein Verständnis ausführen. Maschinen befolgen Algorithmen, egal, ob sie sinnvoll oder absurd sind. Um Algorithmen von Maschinen ausführen lassen zu können, müssen wir also *jedes* Detail *vollständig* und *ganz exakt* zu beschreiben. Insbesondere muß im Algorithmus *jede* mögliche Situation, die bei der Ausführung auftreten kann, berücksichtigt werden, indem festgelegt wird, wie jeweils weiter zu verfahren ist.

Ist unser Verteilungsalgorithmus nun so präzise formuliert, wie es eine Maschine verlangt? Obwohl wir in Abschnitt 2 bereits versucht hatten, exakt zu formulieren, lassen Präzision und Vollständigkeit des Algorithmus noch erheblich zu wünschen übrig:

a) Wie soll eine Maschine wissen, was in Anweisung 1. und 2. eine "gut sichtbare Stelle" ist?

b) Was bedeutet die Anweisung 3.1 "Leere den Briefkasten", wenn keine Zettel abgegeben wurden, der Briefkasten also schon leer ist?

c) Was bedeutet die Anweisung 3.2 "Schreibe auf jeden Zettel eine Zahl (die Codenummer)...", wenn der Briefkasten leer war, und daher in 3.2 gar keine Zettel da sind, auf die man etwas schreiben kann?

d) Wie bestimmt man in 3.2.1 "Die ersten Ziffern sind das Ergebnis der Rechnung..." die ersten Ziffern, wenn auf einem Zettel versehentlich keine Kinderzahl eingetragen wurde?

e) Wie sortiert man einen nicht vorhandenen Zettelstapel in Anweisung 3.3 (s. auch c))?

f) Angenommen, es gibt mehr Eintrittskarten als wartende Besucher. Nachdem die abgegebenen Zettel in die richtige Reihenfolge gebracht worden sind, tritt in Anweisung 5. beim Ausrufen der Ordnungsnummern irgendwann der Fall ein, daß zwar "Noch Eintrittskarten vorhanden sind", der Stapel mit den Zetteln aber leer ist. Welches ist dann in Anweisung 5.1 "Rufe die Ordnungsnummer des obersten Zettels aus" der oberste Zettel?

g) Gleiche Objekte haben wir häufig verschieden, verschiedene Objekte häufig gleich bezeichnet, z.B. "Stapel", "Zettelstapel", "Stapel von Zetteln", "dieser Stapel", "Stapel, der aus den Zetteln besteht, die ...". Die Unterscheidung der Objekte ergab sich nur aus dem Zusammenhang.

Neben diesen Präzisionsdefiziten stehen auch noch technische Probleme einer vollständigen Automatisierung des Verteilungsvorgangs entgegen, z.B. können wir von einer Maschine kaum verlangen, daß sie einen Briefkasten aufhängt oder die Eintrittskarten verkauft. Solange wir den Algorithmus von einem menschlichen Prozessor ausführen ließen, brauchten wir uns um solche Feinheiten nicht zu kümmern. Die Personen, die das Verteilungsverfahren jeweils abzuwickeln hatten, waren mit genügend viel Intelligenz ausgestattet, um die fehlende Information zu ergänzen und die erwähnten problematischen Situationen und Anweisungen vernünftig zu bewältigen. Bei einer Maschine kann man dieses Verständnis nicht voraussetzen. (Unsere Überlegungen zeigen hier insbesondere, daß die oft verwendeten Begriffe "Elektronengehirn" oder "Denkmaschine" die Fähigkeiten von Computern aus heutiger Sicht unzutreffend beschreiben.)

Was folgt nun aus unseren Überlegungen?

1. Wir können nicht erwarten, daß eine Maschine *alle* Tätigkeiten beim Verteilungsverfahren übernimmt, die vorher der Mensch erledigt hat,

und

2. ein Automatisierungsversuch scheitert auf jeden Fall an der unpräzisen Darstellung unseres Verteilungsalgorithmus.

Wenn schon nicht der Gesamtalgorithmus maschinell durchgeführt werden kann, dann vielleicht Teile davon. Die Anweisungen 1., 2., 3.1, 4. und 5. sind kaum automatisierbar, da hierfür überwiegend menschliche, manuelle Fähigkeiten gefordert sind. Die übrigen Anweisungen 3.2, 3.3 und 3.4 bzw. das Mischen beschreiben Handlungen (Vergleichen, Subtrahieren, Sortieren) an Zahlen und Zahlenfolgen (Stapel), die man von einem Computer durchaus erwarten darf.

4.2 Spezifikation des Problems

Zunächst müssen wir genau festlegen, was der Computer leisten soll. Solch eine Beschreibung nennt man **Spezifikation**. Formale Methoden zur Spezifikation lernen wir später kennen (s. Kapitel 8). Hier soll eine umgangssprachliche Darstellung genügen.

Nehmen wir an, es steht ein Kleincomputer C mit einer Tastatur und einem Bildschirm zur Verfügung. C könnten wir zum Beispiel folgendermaßen in das Verteilungsverfahren einbinden: Jedesmal, wenn M den Briefkasten geleert hat, drückt er die Taste "E" (Eingabe) und tippt dann die auf den Zetteln eingetragenen bzw. angekreuzten Daten nacheinander, getrennt durch Druck auf die Eingabetaste ↵ und die Leertaste, ein. C

codiert die Eingaben und bildet nach den Anweisungen 3.2.1 bis 3.2.3 die Codenummer. Nachdem alle Zettel einer Leerung eingetippt sind, drückt M die Taste "F" (Eingabe fertig), worauf C die Zahlen sortiert und mit den früher eingegebenen Werten zusammenmischt. Nachdem M den Briefkasten zum letzten Mal geleert und alle Zettel eingetippt hat, drückt M fortlaufend die Taste "A" (Abrufen), worauf C jedes Mal eine Zahl in der vorgeschriebenen Reihenfolge auf dem Bildschirm zeigt. M ruft dann die letzten drei Ziffern dieser Zahl, also die Ordnungsnummer, aus.
Diese umgangssprachlich beschriebenen Aufgaben von C wollen wir im folgenden so formulieren, daß C sie auch "versteht".

4.3 Maschinenorientierte Präzisierung des Verteilungsalgorithmus

4.3.1 Präzisierung der Objekte

Welche elementaren Fähigkeiten muß C besitzen und wie präzisiert man den Verteilungsalgorithmus?
C operiert mit Zahlen, Zahlenfolgen (Zettelstapel) und Texten. *Zahlen* sind durch ihre ziffernmäßige Darstellung bereits exakt repräsentiert:

771036, 971007, 920126.

Eine mögliche Form zur Modellierung eines Zettelstapels ist eine *Zahlenfolge*, z.B.:

[771036, 971007, 920126].

Diese Zeichenfolge modelliert einen Zettelstapel von drei Zetteln, die von M gemäß Anweisung 3.2 des Algorithmus mit den Codenummern 771036, 971007 und 920126 versehen worden sind. Der Zettel mit der Nummer 771036 liegt oben, der mit 920126 beschriftete Zettel unten. Eine *leere* Zahlenfolge, d.h. einen Zettelstapel ohne Elemente, stellen wir durch das Symbol

[ ]

dar.
Texte schließen wir in Anführungszeichen ein:

"Kommen Sie von auswärts?".

4.3.2 Präzisierung der Elementaranweisungen

Es reicht nicht, daß C mit Zahlen, Zahlenfolgen und Texten hantieren kann. C muß sich diese Objekte auch dauerhaft (zumindest so lange, bis das Verteilungsverfahren abgeschlossen ist) merken (*speichern*) können, und man muß sie auch wieder abrufen können, z.B. um sie anzuschauen. Außerdem muß C, wenn er z.B. mit mehreren Zahlen operiert, diese unterscheiden können. Die einfachste Lösung besteht darin, den einzelnen Objekten unterschiedliche *Namen* zu geben. Unser Algorithmus verwendet zwei Zahlenfolgen (Zettelstapel). Diejenige, welche immer aus den korrekt sortierten Zahlen

besteht, nennen wir fortan "fertig". Mit "leerung" bezeichnen wir die Folge, welche die Zahlen *der* Zettel enthält, die zu einer Briefkastenleerung gehören.
Durch das Zeichen "←" weisen wir C an, sich eine Zahl, Zahlenfolge oder einen Text unter einem Namen zu merken:

Beispiele: fertig ← [771036, 971007, 920126]
leerung ← [ ]
z ← 1
frage ← "Kommen Sie von auswärts?".

Anweisungen dieser Art bezeichnet man als **Zuweisungen**.
Um gespeicherte Daten wieder sichtbar zu machen (*auszugeben*), definieren wir die Anweisung zeige(...). Nach

zeige(fertig)

schreibt C die Zahlen der Zahlenfolge fertig untereinander auf den Bildschirm.
Das nächste Sprachelement weist den Computer an, Zahlen, Zahlenfolgen oder Texte von der Tastatur entgegenzunehmen. Wir verwenden lies(...).

lies(leerung)

fordert den Computer auf, eine Zahlenfolge einzulesen und unter dem Namen leerung zu speichern. C erwartet, daß bei Zahlen und Texten der Druck auf die Eingabetaste ↵ die Eingabe abschließt. Bei der Eingabe einer Zahlenfolge ist nach jeder Zahl und zum Abschluß der Eingabe ein weiteres Mal die Wagenrücklauftaste zu drücken.

Beispiele:

1) Drückt man, wenn der Computer die Anweisung lies(leerung) ausführt, die Tasten

so steht anschließend unter dem Namen leerung die Zahlenfolge
[27, 3, 461].

2) Nach Ausführen der Anweisung lies(z) und Drücken der Tasten

steht anschließend unter dem Namen z die Zahl
1426.

3) Texte gibt man auf die Anweisung lies(frage) so ein:

P r o b e t e x t ↵

4.3.3 Präzisierung der Objektbeschreibung

Doch halt! Woher weiß C eigentlich beim Einlesen in Beispiel 1) [4.3.2], daß leerung eine Zahlenfolge und nicht eine einzelne Zahl oder einen Text bezeichnet? Davon hängt aber ab, ob die Ausführung der Anweisung lies(...) nach erstmaligem Drücken der ↵-Taste beendet ist, oder ob noch auf die Eingabe weiterer Zahlen gewartet werden muß. Was also noch fehlt, ist eine Festlegung analog zur Objektbeschreibung, welche Namen verwendet werden und welchen Typ die Objekte haben, die mit diesen Namen bezeichnet werden. Wir kennen nur die Typen "Zahl", "Zahlenfolge" und "Text" und schreiben daher zu Beginn:

```
def fertig, leerung: Zahlenfolge;
def z: Zahl;
def frage: Text .
```

Wir *vereinbaren* durch diese Zeichenfolge, daß fertig und leerung in Zukunft jeweils eine Zahlenfolge bezeichnen dürfen und z für eine Zahl stehen darf. frage bezeichnet einen Text. Eine solche Festlegung bezeichnet man als **Vereinbarung** oder **Deklaration**. Statt von Namen spricht man meist von **Bezeichnern**.

Definition A:
Ein **Bezeichner** ist eine Zeichenfolge, die zur eindeutigen Identifizierung eines Objekts dient.

Definition B:
Die **Zuweisung** ist eine Anweisung, durch die ein Bezeichner einen (neuen) Wert erhält.

Definition C:
Eine **Deklaration/Vereinbarung** ist die Festlegung, welche Bedeutung ein Bezeichner im Algorithmus besitzen soll, d.h., welche Typen von Objekten mit dem Bezeichner benannt werden dürfen.

4.3.4 Präzisierung von Operationen auf Objekten

Wir können für unseren hypothetischen Computer C nun mittlerweile Bezeichner deklarieren, Zahlen, Zahlenfolgen und Texte von der Tastatur einlesen und über den Bildschirm wieder ausgeben und Objekte mit Bezeichnern benennen. Uns fehlt aber noch

jede Möglichkeit, um Zahlen und Zahlenfolgen zu verändern, zu sortieren oder zu mischen. Diesen Mangel beseitigen wir im folgenden. Bei Zahlen setzen wir die arithmetischen Operationen +, -, *, / (plus, minus, mal, geteilt) voraus.

Beispiel: Addition zweier Zahlen a und b:

```
def a,b,c: Zahl;
a ← 3;
b ← 6;
c ← a + b;
zeige(c) .
```

Nach Verarbeitung dieser Anweisungsfolge schreibt C eine Neun auf den Bildschirm.

Für eine Zahlenfolge f bezeichne

sort(f)

die Version von f, die die Zahlen von f in aufsteigender Reihenfolge enthält. Die Anweisung

3.3 Sortiere die Zettel aufsteigend nach diesen Zahlen zu einem Stapel, die kleinste Zahl nach oben

unseres Algorithmus schreiben wir in dieser Notation dann konsequenterweise als

leerung ← sort(leerung) .

Dies bedeutet, die Zahlenfolge mit dem Bezeichner leerung wird sortiert. Die Ergebnisfolge erhält anschließend wieder den Namen leerung. Die ursprüngliche unsortierte Folge ist verschwunden. Sei weiter

misch(f,g)

die Folge, die durch Mischen der beiden Zahlenfolgen f und g entsteht. Anweisung

3.4 Mische diesen Stapel und den Stapel, der aus den Zetteln besteht, die du bei früheren Briefkastenleerungen geholt hast, zusammen. Verwende den Algorithmus für Mischen (siehe unten)

des Algorithmus lautet dann

fertig ← misch(fertig,leerung) .

In Anweisung

5.1 Rufe die Ordnungsnummer des obersten Zettels aus

des Algorithmus soll M nacheinander die Ordnungsnummern der endgültigen Folge ausrufen. Dazu ist es notwendig, daß C die einzelnen Zahlen der Folge fertig auf Tastendruck der Reihe nach am Bildschirm anzeigt. Um die erste Zahl einer Zahlenfolge abzuspalten, verwenden wir die Sprachelemente erstes(...) und rest(...). Für eine Zahlenfolge f, die nicht leer ist, steht dann

erstes(f)

für die erste Zahl der Folge f.

Beispiel: erstes([771036, 971007, 920126]) liefert 771036.

Mit

rest(f)

bezeichnen wir die Folge f ohne das erste Folgenelement.

Beispiel: rest([771036, 971007, 920126]) ist die Folge [971007, 920126].
Speziell ist rest([771036]) die leere Folge [ ].

Die Anweisungen

5.1 Rufe die Ordnungsnummer des obersten Zettels aus;
5.2 Lege den Zettel beiseite

des Verteilungsalgorithmus schreiben sich in dieser Notation dann als

zeige(erstes(fertig));
fertig ← rest(fertig) .

Die erste Zahl der Folge wird am Bildschirm angezeigt. M hat nun die Möglichkeit, die Ordnungsnummer (d.h. die letzten drei Ziffern der Zahl) auszurufen und die Karten zu verkaufen. C löscht anschließend das erste Folgenglied von fertig mit rest. Beim nächsten Mal liefert zeige(erstes(fertig)) dann das zweite Element der ursprünglichen Liste fertig.

An mehreren Stellen im Algorithmus muß M zwei Zettelstapel verbinden. Präzisierung: Zwei Folgen f und g werden mit dem Zeichen • aneinandergehängt (*konkateniert*).

Beispiel: Für die Folgen [3, 27, 5] und [8, 1], ist das Ergebnis von

[3, 27, 5] • [8, 1]

die Folge

[3, 27, 5, 8, 1].

Speziell ist für eine Folge f sowohl das Ergebnis von

f • [ ]

als auch von

[ ] • f

wieder die Folge f.

4.3.5 Präzisierung der Konstruktoren und Bedingungen

Wir verwenden für Bedingungen bei Zahlen, Zahlenfolgen und Texten den Test auf Gleichheit "=" und auf Ungleichheit "≠" und bei Zahlen zusätzlich die Vergleichsoperationen "<", ">", "≤", "≥".

Beispiele für Bedingungen sind:

a = 1, 3 = 4, [1,2] ≠ [2,3], f = [ ], "ja" ≠ antwort, erstes(f) ≤ 3 .

Mehrere Bedingungen können wir durch <u>und</u>, <u>oder</u> und <u>nicht</u> miteinander verknüpfen und erhalten neue kompliziertere Bedingungen. Seien B und B' zwei Bedingungen. Die Bedingung

B <u>und</u> B'

ist dann und nur dann wahr, wenn sowohl B als auch B' wahr ist. Die Bedingung

B <u>oder</u> B'

ist dann und nur dann wahr, wenn entweder B oder B' oder beide wahr sind. Die Bedingung

<u>nicht</u> B

ist dann und nur dann wahr, wenn B nicht wahr sind.

Beispiele:

1) a = 1 <u>und</u> erstes(f) ≤ 3.
   Diese Bedingung ist wahr, wenn sowohl der Bezeichner a für die Zahl 1 steht, als auch das erste Folgenelement von f kleiner oder gleich der Zahl 3 ist.
2) [1,2] ≠ f <u>oder</u> "ja" ≠ antwort.
   Diese Bedingung ist erfüllt, wenn entweder die Folge mit dem Bezeichner f verschieden von der Folge [1,2] ist oder der Bezeichner antwort nicht für den Text "ja" steht oder beide Fälle zutreffen.
3) (a=2 <u>und</u> b=3) <u>oder</u> f=[1,2].
   Diese Bedingung ist erfüllt, wenn sowohl a für die Zahl 1 steht, als auch b für die Zahl 3 steht, oder wenn f gleich der Folge [1,2] ist, falls eine der ersten beiden Bedingungen nicht zutrifft.
   Wie man an diesem Beispiel erkennt, kann man einfache Bedingungen mittels <u>und</u> und <u>oder</u> und Klammerbildung zu beliebig komplizierten Bedingungen kombinieren. Bei der Auswertung geht man wie bei arithmetischen Ausdrücken vor, indem man zunächst die innersten Klammern auswertet und berücksichtigt, daß <u>nicht</u> stärker bindet als <u>und</u> und <u>und</u> stärker bindet als <u>oder</u>.

Nun zu den Konstruktoren: Was die *Konkatenation* von Anweisungen betrifft, so bleiben wir bei der bisherigen Schreibweise: Was oben steht, wird auch zuerst ausgeführt, und Anweisungen werden durch den Operator ";" (Semikolon) verbunden.

Den *Alternativkonstruktor* "Wenn ... dann ... sonst ..." notieren wir durch

<u>wenn</u> *Bedingung* <u>dann</u>
...
<u>sonst</u>
...
<u>ende</u>.

Den *Iterationskonstruktor* "Solange ... tue folgendes ..." schreiben wir folgendermaßen:

```
solange Bedingung tue
...
ende.
```

Die Wörter tue, sonst, wenn, def usw. haben eine besondere Bedeutung. Es sind *Schlüsselwörter*. Um sie von Bezeichnern zu unterscheiden, werden sie fortan immer unterstrichen.

Beispiele:

1) Um das Ergebnis der Rechnung 1+2+3+...+10 zu berechnen, können wir nun schreiben:

```
def i,s: Zahl;
i ← 1;
s ← 0;
solange i ≤ 10 tue
    s ← s + i;
    i ← i + 1
ende;
zeige(s) .
```

2) Die Anweisung

5. Solange noch Eintrittskarten vorhanden sind, tue folgendes:
   5.1 Rufe die Ordnungsnummer des obersten Zettels aus;
   5.2 Lege den Zettel beiseite;
   5.3 Verkaufe an die Gruppe mit dieser Ordnungsnummer die gewünschte Zahl an Eintrittskarten

unseres Algorithmus notieren wir unter Berücksichtigung von Punkt f) ("mehr Eintrittskarten als Besucher") der Mängelliste aus 4.1 folgendermaßen:

```
solange taste = "A" und fertig ≠ [ ] tue
    zeige(erstes(fertig));
    fertig ← rest(fertig);
    lies(taste)
ende .
```

Dabei sei taste ein Bezeichner, der folgendermaßen definiert ist:

```
def taste: Text.
```

Die erste Bedingung der solange-Anweisung besagt dann, daß, immer wenn M die Taste "A" drückt und die Folge fertig nicht leer ist, das erste Element von fertig auf dem Bildschirm angezeigt wird.

3) Anweisung

3.2.2 Wenn auf die Frage "Kommen Sie von auswärts?" "ja" angekreuzt ist, dann ist die nächste Ziffer eine Null, sonst ist die nächste Ziffer eine Eins

des Algorithmus lautet in der neuen Schreibweise nun:

```
wenn antwort = "ja" dann
    ziffer ← 0
sonst
```

```
        ziffer ← 1
    ende  .
```

4.3.6 Vollständiger Verteilungsalgorithmus in präziser Form

Wir haben nun alle Hilfsmittel bereitgestellt, um den automatisierbaren Teil des Algorithmus in einer so präzisen Form zu formulieren, das er von einem Computer ausgeführt werden kann:

```
def ziffer, z, kinderzahl, ordnungsnr: Zahl;
def fertig, leerung: Zahlenfolge;
def antwort, taste: Text;
fertig ← [ ];
lies(taste);
solange taste = "E" tue
    leerung ← [ ];
    solange taste ≠ "F" tue
        lies(kinderzahl);
        lies(antwort);
        wenn antwort = "ja" dann
            ziffer ← 0
        sonst
            ziffer ← 1
        ende;
        lies(ordnungsnr);
        z ← (100 - kinderzahl) * 10000 + ziffer * 1000 + ordnungsnr;
        leerung ← leerung • [ z ];
        lies(taste)
    ende;
    leerung ← sort(leerung);
    fertig ← misch(fertig,leerung);
    lies(taste)
ende;
solange taste ="A" und fertig ≠ [ ] tue
    zeige(erstes(fertig));
    fertig ← rest(fertig);
    lies(taste)
ende  .
```

Wir sehen, daß unter den Anweisungen innerhalb einer Schleife wieder eine Schleife auftreten kann. Man sagt, Schleifen können **geschachtelt** werden, und spricht von *inneren* und *äußeren* Schleifen. Hier ist also

```
solange taste ≠ "F" tue ... ende
```

innere Schleife zur äußeren Schleife

```
solange taste = "E" tue ... ende .
```

Die innere Schleife beschreibt eine Leerung des Briefkastens, die äußere Schleife den Vorgang aller Leerungen und zwischenzeitlichen Sortierungen.

4.3.7 Abstraktionskonstruktor

Da wir nicht annehmen können, daß ein Computer klüger ist als ein Mensch und sofort weiß, was mit der Anweisung misch gemeint ist, müssen wir diese genau so erläutern, wie wir das für M hatten tun müssen. Wir wollen dies getrennt von dem übrigen Algorithmus tun. Eine solche getrennte Erläuterung bezeichnet man als **Prozedur** oder **Funktion**. Eine Prozedur oder Funktion wird selbst als Deklaration aufgefaßt und im Deklarationsteil eines Algorithmus aufgeführt.

Definition D:
Die Zusammenfassung von Deklarationen und Anweisungen zu einer Einheit bezeichnet man als **Prozedur**. Liefert eine Prozedur einen Wert, so nennt man sie **Funktion** oder **Funktionsprozedur**. Jede beliebige Prozedur oder Funktion kann als elementare Anweisung oder elementarer Ausdruck (durch *Aufruf*) in jedem anderen Algorithmus verwendet werden.

Setzen wir nun den Algorithmus für "Mischen" in eine Funktion um. Zu Anfang steht dort der Satz "Zwei Zettelstapel S1 und S2 mischst du ... zu S3 ...". Wir wissen bereits, daß Zettelstapel in unserer Präzisierung Zahlenfolgen sind und daß alle Objekte mit Bezeichnern versehen und deklariert sein müssen. Daher:

```
def s1,s2,s3: Zahlenfolge.
```

Anweisung

M1. Solange auf beiden Stapeln S1 und S2 noch Zettel liegen, tue folgendes: ...

bekommt die Form

```
solange s1 ≠ [ ] und s2 ≠ [ ] tue
    ...
ende .
```

Anweisung

M1.1 Nimm von den beiden obersten Zetteln von S1 und S2 den mit der kleineren Codenummer weg

kann mit Hilfe des wenn-dann-Konstruktors umgesetzt werden:

```
wenn erstes(s1) < erstes(s2) dann
    s3 ← s3 • [ erstes(s1) ];
    s1 ← rest(s1)
sonst
    s3 ← s3 • [ erstes(s2) ];
```

```
        s2 ← rest(s2)
    ende .
```

Die Anweisung

M2. Wenn einer der Stapel S1 oder S2 leer ist, dann hänge den anderen komplett hinten an den Stapel S3 an

vereinfacht sich zu

```
s3 ← s3 • s1 • s2 .
```

Man mache sich klar, daß diese Anweisung mit M2 übereinstimmt, obwohl sie keine bedingte Anweisung verwendet; denn eine der Zahlenfolgen s1 oder s2 muß an dieser Stelle ja die leere Folge sein.

Insgesamt lautet der Algorithmus für das Mischen zweier Zahlenfolgen in präziser Form also:

```
def s1, s2 s3: Zahlenfolge;
solange s1 ≠ [ ] und s2 ≠ [ ] tue
    wenn erstes(s1) < erstes(s2) dann
        s3 ← s3 • [ erstes(s1) ];
        s1 ← rest(s1)
    sonst
        s3 ← s3 • [ erstes(s2) ];
        s2 ← rest(s2)
    ende
ende;
s3 ← s3 • s1 • s2
```

Bei dieser Formulierung sind aber noch eine Reihe von Dingen nicht berücksichtigt:

a) Der vorgelegte Algorithmus soll eine *Funktion* für den im Programm verwendeten *Aufruf* misch sein. Diese Beziehung zwischen dem Algorithmus und seinem Aufruf tritt aber noch nirgends auf. Wir wollen daher im folgenden Funktionen und Prozeduren optisch abheben und dem Computer kenntlich machen, indem wir zu Beginn das *Schlüsselwort* prozedur oder funktion gefolgt von ihrem Namen (also misch) stellen und die Prozedur mit prozende oder funkende beenden. Also:

```
funktion misch
    ...
funkende
```

b) Im Algorithmus wird misch auf die beiden Zahlenfolgen fertig und leerung angewendet. In der Funktion sind hierfür aber die Bezeichner s1 und s2 vorgesehen. Es wäre sogar vorstellbar, daß misch in einem anderen Algorithmus auf weitere Paare von Zahlenfolgen angewendet wird, die andere Bezeichner haben. Dies ist hier jedoch nicht vorgesehen. misch ist also eine Funktion mit zwei **Parametern**, nämlich den zwei zu mischenden Zahlenfolgen. s1 und s2 sind nur Platzhalter (sog. **formale Pa-**

rameter) für zwei Zahlenfolgen. Für welche konkreten Zahlenfolgen (sog. **aktuelle Parameter**) s1 und s2 im Einzelfall stehen sollen, stellt sich erst bei Verwendung einer Prozedur oder Funktion heraus.

Definition E:
Als **Parameter** bezeichnet man variable oder austauschbare Objekte in einer Prozedur oder Funktion. Die in einer Prozedur oder Funktion benutzten und im Kopf der Prozedur anzugebenden Parameter heißen **formale Parameter**; sie sind Platzhalter für später einzusetzende Objekte. Die im Aufruf einer Prozedur oder Funktion stehenden Argumente nennt man **aktuelle Parameter**.

Diese Beziehung zwischen aktuellen und formalen Parametern müssen wir C mitteilen. Im einzelnen müssen wir festlegen, welche Objekte als Parameter überhaupt zugelassen sein sollen und welche formalen mit welchen aktuellen Parametern assoziiert werden. Diese *Parameterspezifikation* setzen wir in Klammern unmittelbar hinter den Namen der Funktion. s1 und s2 müssen wir in der Funktion misch dann natürlich aus dem Deklarationsteil streichen. Die so modifizierte Funktion lautet nun:

```
funktion misch(s1,s2: Zahlenfolge);
def s3: Zahlenfolge;
      ... (wie oben)
funkende
```

Die erste Zeile legt fest, daß als aktuelle Parameter zwei Zahlenfolgen vorgesehen sind und daß der erste aktuelle Parameter in einem Aufruf von misch mit s1 und der zweite mit s2 identifiziert wird.

c) Wir haben immer gesagt "misch(fertig,leerung) steht für die Zahlenfolge, die durch Mischen der Zahlenfolgen fertig und leerung entsteht". Dies ist bisher noch nicht berücksichtigt, denn das Ergebnis des Mischens von s1 und s2 ist in der Funktion mit s3 bezeichnet. Zwischen den Bezeichnern misch und s3 besteht im Moment noch keine Beziehung. Um misch und s3 zu assoziieren, verwenden wir am Ende der Funktion die Anweisung

```
ergebnis s3 .
```

Diese Anweisung besagt, daß die Zahlenfolge s3 nach Abarbeiten der Funktion misch das Ergebnis ist. Die Tatsache, daß misch eine Funktion ist, die als Ergebnis einen Wert vom Typ Zahlenfolge liefert, halten wir in der ersten Zeile der Funktion durch das nachgestellte Wort Zahlenfolge fest.

Die Funktion lautet daher in ihrer endgültigen Form

```
funktion misch (s1, s2: Zahlenfolge) Zahlenfolge;
def s3: Zahlenfolge;
```

```
s3 ←[ ];
solange s1 ≠ [ ] und s2 ≠ [ ] tue
      wenn erstes(s1) < erstes(s2) dann
            s3 ← s3 • [ erstes(s1) ];
            s1 ← rest(s1)
      sonst
            s3 ← s3 • [ erstes(s2) ];
            s2 ← rest(s2)
      ende
ende;
s3 ← s3 • s1 • s2;
ergebnis s3
funkende
```

Bearbeitung eines Aufrufs.

Was passiert nun genau, wenn C bei der Ausführung auf die Anweisung

fertig ← misch(fertig,leerung)

trifft? C unterbricht die Ausführung dieser Anweisung so lange, bis die Ergebnisse des Aufrufs vorliegen. Dazu sucht C in seinem Speicher nach einer Funktion mit dem Bezeichner misch. Findet er sie, dann vergleicht er die Typen der aktuellen Parameter mit der Parameterspezifikation der formalen Parameter. fertig und leerung haben den Typ Zahlenfolge, die beiden formalen Parameter s1 und s2 haben ebenfalls den Typ Zahlenfolge. C kann bei diesem Test also keine Ungereimtheiten finden. C wird daher die aktuellen Parameter mit den zugehörigen formalen Parametern identifizieren; genauer: Überall in der Funktion wird der formale Parameter s1 durch den Wert von fertig und der formale Parameter s2 durch den Wert von leerung ersetzt. Anschließend beginnt C, die Anweisungen der Funktion auszuführen. Nach Abarbeitung der Funktion ist dann unter dem Bezeichner s3 die Ergebnisfolge des Mischens verfügbar. Die letzte Anweisung der Funktion ergebnis s3 weist C dann an, an die Stelle des Aufrufs misch(fertig,leerung) die Ergebnisfolge s3 zu setzen, was zur Ausführung der Anweisung fertig←s3 führt. Die Funktion ist damit beendet, und die Abarbeitung kann an der unterbrochenen Stelle fortgesetzt werden.

Das *Prozedur- und Funktionskonzept* ist eines der mächtigsten Konzepte der Algorithmisierung: Hierdurch kann jede als Algorithmus formulierte Vorschrift zu einer Elementaranweisung in einem anderen Algorithmus werden.

Darüber hinaus dienen Prozeduren und Funktionen der Zerlegung und *Strukturierung* umfangreicher Algorithmen: Man zerlegt die Lösung eines großen Problems in etwa 5 bis 10 Teilschritte, die man als Prozedur- oder Funktionsaufrufe formuliert. Anschließend

zerlegt man jeden Teilschritt (also jede Prozedur/Funktion) wiederum in Teilschritte, die ebenfalls als Prozeduren/Funktionen formuliert werden. Dieses *schrittweise Verfeinerung* einer Problemlösung treibt man so lange, bis man überschaubare kleine Einheiten erreicht hat.

Auch Prozeduren und Funktionen können ineinander geschachtelt werden.

Definition F:
Eine Prozedur ist in einer anderen **geschachtelt**, wenn sie in ihr deklariert oder aufgerufen wird. Im ersten Fall spricht man von **statischer**, im zweiten Fall von **dynamischer** Schachtelung bzw. **Abstützen** einer Prozedur auf eine andere.

Das Schachteln von Prozeduren tritt vorwiegend bei der schrittweisen Verfeinerung von Problemlösungen und bei der Rekursion (dazu später) auf.
Beispiel: Der Deklarationsteil eines durch schrittweise Verfeinerung entstandenen Algorithmus hat prinzipiell folgenden Aufbau:

```
prozedur p1(...) ...
    funktion p11(...) ...
        prozedur p111(...) ...
            ...
        prozende;
        funktion p112(...) ...
            ...
        funkende;
        ...
    funkende;
    prozedur p12(...) ...
        funktion p121(...) ...
            ...
        funkende;
        prozedur p122(...) ...
            ...
        prozende;
        ...
    prozende
prozende;
prozedur p2(...) ...
    prozedur p21(...) ...
        prozedur p211(...) ...
            ...
        prozende;
        prozedur p212(...) ...
            ...
        prozende;
        ...
    prozende
    prozedur p22(...) ...
```

```
            funktion p221(...) ...
                ...
            funkende;
            funktion p222(...) ...
                ...
            funkende;
            ...
        prozende
    prozende
    ...
```

Dabei hat man die Problemlösung zunächst in die Teilschritte (Prozeduren/Funktionen) p1, p2, ... zerlegt. Anschließend hat man p1, p2, ... jeweils wiederum in die Teilschritte p11, p12, ..., p21, p22, ... unterteilt. Bei der nächsten Zerlegung wird dann z.B. p22 in die Schritte p221, p222, ... aufgeteilt. Es sind also immer diejenigen Prozeduren/Funktionen in einer anderen Prozedur geschachtelt, die durch schrittweise Verfeinerung aus ihr entstanden sind. Zum Beispiel sind p221, p222, ... in p22 geschachtelt, p22 ist ihrerseits in p2 geschachtelt. Das Endergebnis ist eine sog. *hierarchische Modularisierung* des Ausgangsproblems. Graphisch veranschaulicht man sich die Struktur des obigen Programms durch einen *Baum* (Abb. 1).

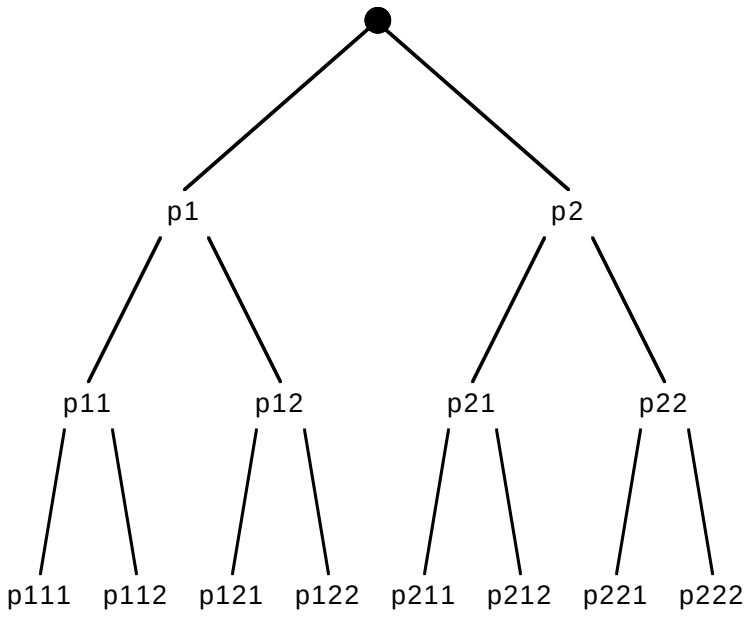

Abb. 1: Baum

4.4 Programm und Programmiersprache

Unser Verteilungsproblem ist nun vollständig gelöst. Das Ergebnis ist eine präzise Beschreibung der wesentlichen (automatisierbaren) Elemente des Algorithmus in einer Form, die man C übermitteln kann und die C ausführen kann. Darstellungen von Algo-

rithmen, die so präzise sind, daß sie von einem Computer ausgeführt werden können, bezeichnet man als **Programme**. Dazu benötigt man einen Formalismus, dessen Darstellungselemente die erforderliche Präzision besitzen. Formalismen, die diese Forderung erfüllen, nennt man **Programmiersprachen**. Unsere sehr einfache Programmiersprache soll den Namen **PRO** tragen. Sie ist erfunden, orientiert sich aber an den Sprachelementen real vorkommender Programmiersprachen, z.B. PASCAL, MODULA-2, ADA usw. Ein in PRO geschriebenes Programm kann daher leicht in konkrete Sprachen übertragen werden.

Definition G:
Eine Sprache zur Formulierung von Algorithmen, die von einem Computer ausgeführt werden können, bezeichnet man als **Programmiersprache**.
Die Formulierung eines Algorithmus in einer Programmiersprache nennt man **Programm**.

Während Algorithmen relativ allgemein beschrieben werden können und an keine besonderen formellen Vorschriften gebunden sind, sind Programme im exakt definierten Formalismus einer Programmiersprache verfaßt. Ein und derselbe Algorithmus kann in verschiedenen Programmiersprachen formuliert werden; ein Algorithmus bildet daher eine *Abstraktion aller Programme*, die ihn formal beschreiben.

Es gibt zur Zeit über 1000 verschiedene Programmiersprachen, von denen aber nur etwa 20 in nennenswertem Umfang praktisch verwendet werden. Diese Programmiersprachen kann man in drei große Klassen einteilen, die sich darin unterscheiden, welches Konzept oder Denkschema zur Beschreibung von Abläufen den Sprachen zugrundeliegt. Weitgehend fixe Größe aller Programmiersprachen sind jedoch die elementaren Datentypen und die Datentypkonstruktoren; sie kommen in allen Sprachen in etwa gleicher Weise vor.
Bei den *imperativen Programmiersprachen* besteht ein Programm aus einer Folge von Anweisungen (Befehlen, daher imperativ von lat. imperare=befehlen) an den Computer, eine Handlung durchzuführen. Die elementarste Anweisung ist die Zuweisung. Wesentliches weiteres Merkmal ist das Variablenkonzept: Daten werden eingelesen und in Variablen abgelegt, die gelesen oder beschrieben werden können. Imperative Sprachen orientieren sich weitgehend an der Maschine, denn in ihnen spiegelt sich deutlich die Architektur heutiger Computer wider. Beispiele für imperative Sprachen sind offenbar PRO sowie, ADA, ALGOL, BASIC, C, EIFFEL, FORTRAN, MODULA, OBERON, PASCAL.

Bei *funktionalen Programmiersprachen* ist ein Programm eine Zusammenstellung von Funktionen, die ggf. aufeinander aufbauen und sich schließlich auf elementare Funktionen abstützen. Der Start eines Programms erfolgt durch Aufruf einer Funktion ggf. mit Parametern. Es gibt (zumindest bei rein funktionalen Sprachen) kein zu imperativen Programmiersprachen vergleichbares Variablenkonzept. Wesentlicher Konstruktor von Funktionen ist die Rekursion, die die Iteration ersetzt. Beispiele für funktionale Programmiersprachen sind ML als zentrale Sprache dieser Vorlesung sowie GOFER, HASKELL, HOPE, LISP, MIRANDA.
Bei *prädikativen Programmiersprachen* wird Programmieren als Beweisen von Behauptungen in einem System von Tatsachen und Schlußfolgerungen aufgefaßt. Der Programmierer gibt sein Wissen über einen Sachverhalt in Form von Fakten und Regeln (=Regeln, wie man aus bekannten Fakten neue Fakten gewinnt) vor, und die Aufgabe des Computers besteht darin, eine aufgestellte Behauptung zu beweisen oder zu widerlegen. Auch hier unterscheidet sich der Variablenbegriff vom imperativen und funktionalen Stil: Variablen sind hier Unbekannte. Der prädikative Stil ist der abstrakteste der drei Stile, das zugrundeliegende Modell unterscheidet sich am weitesten von der realen Architektur heutiger Rechner. Beispiele für prädikative Sprachen sind CLP, GÖDEL, PROLOG.
In weiteren Verlauf der Vorlesung werden wir die Unterschiede der drei Sprachstile genauer herausarbeiten.

4.4.1 Syntax und Semantik

Programmiersprachen besitzen immer eine genau festgelegte und eindeutige Syntax und Semantik.
Die **Syntax** bestimmt die sprachliche Notierung von Programmen. Sie muß präzise definiert sein, damit ein Benutzer der Sprache jederzeit feststellen kann, welche aller möglichen Aneinanderreihungen von Buchstaben, Ziffern und sonstigen Zeichen korrekte Programme sind und welche nicht. Natürliche Sprachen wie Deutsch, Englisch usw. besitzen eine mehr oder wenige feste Syntax. Diese wird durch "grammatikalische Regeln" definiert. Ähnlich geht man auch bei der Definition von Programmiersprachen vor.

Die **Semantik** legt die Bedeutung syntaktisch korrekter Programme fest. Auch sie muß präzise definiert sein, damit ein Benutzer der Sprache stets nachvollziehen kann, was Programme auf dem Computer bewirken. Die Semantik (Bedeutung) eines Programms präzisiert man z.B. durch Definition
- der Art und Weise, wie ein Rechner es ausführt (*operationale Semantik*),

- der Eigenschaften, die die (Ein-/Ausgabe-)Variablen vorher und nachher erfüllen (*axiomatische Semantik*),
- der Funktion, die das Programm berechnet (*denotationale Semantik*).

Dabei geht man induktiv vor, indem man die operationale/axiomatische/denotationale Semantik aller Sprachelemente der Programmiersprache festlegt und die Semantik eines Programms aus der Semantik der verwendeten Sprachelemente zusammensetzt.
Natürliche Sprachen besitzen *keine* genau bestimmte Semantik: Viele Wörter oder Sätze haben mehrere Bedeutungen (Man erinnere sich an die Situation, als M die mehrdeutige Anweisung "Mischen" falsch verstand) oder sind nur durch Einbeziehung von Hintergrundinformationen verständlich. Daher sind natürliche Sprachen ungeeignet, um als Programmiersprache zu dienen, wovon wir uns eingangs von Kapitel 4 selbst überzeugt haben.

Beispiele:

1) Die Zeichenfolge (das Programm) P

```
def s: Zahl;
lies(s);
s ← 0;
zeige(s) .
```

ist syntaktisch korrekt. Die Semantik lautet umgangssprachlich: "Lies eine beliebige Zahl ein und gib die Zahl 0 aus". Die denotationale Semantik erhält man durch Ermittlung der Funktion f_P, die das Programm berechnet. Es gilt:

f_P: $Z \rightarrow Z$ mit

$f_P(s)=0$ für alle $s \in Z$.

Offenbar besitzt obiges Programm die gleiche Semantik wie das folgende einzeilige Programm:

```
zeige(0) .
```

Unterschiedliche Programme können also die gleiche Semantik besitzen. Man sagt in diesem Falle: Die Programme sind **semantisch äquivalent**.

2) Die Zeichenfolge (das Programm) P'

```
def i,s: Zahl;
lies(n);
i ← 1;
s ← 0;
solange i ≤ n tue
    s ← s + i;
    i ← i + 1
zeige(s)
```

ist syntaktisch *nicht* korrekt: Der Bezeichner n vom Typ Zahl ist nicht deklariert, und der Konstruktor solange ist nicht durch das Wort ende abgeschlossen. Man kann P' daher keine Semantik zuordnen. Die folgende abgeänderte Version P" des obigen Programms ist syntaktisch korrekt:

```
def i, n, s :Zahl;
lies(n);
i ← 1;
s ← 0;
solange i ≤ n tue
      s ← s + i;
      i ← i + 1
ende;
zeige(s)  .
```

Seine Semantik lautet: "Falls der eingegebene Wert n kleiner oder gleich 0 ist, so gib die Zahl 0 aus. Falls n größer als 0 ist, so gib die Summe der ersten n Zahlen 1+2+3+…+n aus". Wiederum definiert man die denotationale Semantik des Programms, indem man die berechnete Funktion $f_{P''}$ angibt. Es gilt:

$f_{P''}: Z \rightarrow Z$ mit

$$f_{P''}(n) = \begin{cases} \sum_{i=1}^{n} i, & \text{falls } n \geq 1, \\ 0, & \text{sonst.} \end{cases}$$

4.4.2 Korrektheit und Komplexität

Bei der Formulierung des Algorithmus für einen menschlichen Verteiler waren wir in einer komfortablen Situation. Wir brauchten uns um die korrekte Arbeitsweise des Algorithmus nicht im Detail zu kümmern. Ein paar Tests genügten, denn wir konnten darauf vertrauen, daß der menschliche Verteiler Ungereimtheiten kraft seiner inneren Einsicht auflöst und schon irgendwie das Gewünschte leistet.

Auch die Effizienz des Algorithmus haben wir durch ein paar Probeläufe abgeschätzt und für hinreichend befunden. Notfalls hätte sich der Verteiler halt ein bißchen beeilen müssen, um noch vor Vorstellungsbeginn fertig zu werden.

Nun wo wir den Computer in das Verfahren eingebunden haben, können wir uns diese Ungenauigkeiten nicht mehr leisten. Wir müssen vielmehr peinlich genau beweisen, daß das Programm die Spezifikation erfüllt, also stets und in jeder Hinsicht das Gewünschte leistet, und die Zeitvorgaben einhält. Wir beweisen diese beiden Eigenschaften hier nur für die Funktion misch, um einen Geschmack von der Vorgehensweis zu vermitteln.

Zunächst zur Korrektheit. Die (*funktionale*) Spezifikation von misch lautet halbformal (formale Darstellungen folgen später):

Eingabe: $s_1, s_2 \in IN_0^*$

Ausgabe: $s_3 \in IN_0^*$

Funktionaler Zusammenhang: Für $s_1=[x_1,...,x_n]$ mit $x_1 \le x_2 \le ... \le x_n$ und $s_2=[y_1,...,y_m]$ mit $y_1 \le y_2 \le ... \le y_m$ ist $s_3=[z_1,...,z_{n+m}]$ mit $z_1 \le z_2 \le ... \le z_{n+m}$ und s_3 ist eine Permutation von $s_1 \bullet s_2$.

Behauptung: misch terminiert nach höchstens n+m Schleifendurchläufen und erfüllt die Spezifikation.

Beweis: Wir beweisen die Behauptung durch vollständige Induktion über n+m.

Induktionsanfang: Es ist zu zeigen, daß die Behauptung für n+m=0 gilt.

Beweis des Induktionsanfangs: Bei Eingabe von s_1=[ ] und s_2=[ ] werden folgende Anweisungen des Programms durchlaufen:

s3 ← [ ]; s3 ← s3•s1•s2; <u>ergebnis</u> s3

Am Schluß ist, wie gewünscht, s_3=[ ]. Folglich ist der Induktionsanfang bewiesen.

Induktionsvoraussetzung: Sei die Behauptung wahr bis zu einer Zahl n+m.

Induktionsbehauptung: Die Behauptung gilt dann auch für n+m+1.

Beweis der Induktionsbehauptung: Ist s_1=[ ] oder s_2=[ ], so folgt die Behauptung durch Nachvollziehen der Anweisungen des Programms:

s3 ← [ ]; s3 ← s3•s1•s2; <u>ergebnis</u> s3 .

Seien also $s_1 \neq [\ ] \neq s_2$. Sei o.B.d.A. erstes(s_1)<erstes(s_2). Dann ist erstes(s_1) das min$\{x_1,...,x_n,y_1,...,y_m\}$ und in der Schleife werden die Anweisungen

s3 ← s3•[erstes(s1)]; s1 ← rest(s1)

ausgeführt. Zu Beginn des zweiten Schleifendurchlaufs gilt also s_3=[erstes(s_1)] und $s_1=[x_2,...,x_n]$. Nun sind also nur noch zwei Folgen mit zusammen n+m Elementen zu mischen. Nach Induktionsvoraussetzung mischt die Funktion diese beiden Folgen in der behaupteten Weise mit höchstens n+m weiteren Schleifendurchläufen. Da jede Operation nur Elemente hinten an s_2 anfügt, bleibr das erste Element das Minimum aller Folgenelemente. Dann werden also die beiden ursprünglichen Folgen in korrekter Weise mit höchstens n+m+1 Schleifendurchläufen gemischt. €€

Das Programm ist also bezgl. der Spezifikation korrekt. Nun zur Komplexität: Wie effizient ist das Programm? Da jede einzelne Operation Zeit kostet, zählen wir, wieviele Elementaroperationen bei Eingabe zweier Folgen ausgeführt werden. Da diese Zahl von den Elementen der Folge abhängt, wir diese Elemente aber vorab nicht kennen, gehen wir von dem *schlimmsten Fall* (*worst-case*) aus. Auch später werden wir Algorith-

men fast immer danach beurteilen, wie hoch ihre Komplexität im schlimmsten Fall ist (*worst-case complexity*).
Wir zählen also die Elementaroperationen, die beim Mischen zweier Folgen s_1,s_2 mit n bzw. m Elementen im *Höchstfalle* benötigt werden. Diese Überlegung erscheint vernünftig, denn für das Verteilungsverfahren ist ebenfalls eine Maximalzeit vorgegeben. Kommen wir im Einzelfall mit weniger Zeit aus, stört das nicht.

Behauptung: Zum Mischen zweier Folgen mit zusammen n+m Elementen benötigt die Funktion misch höchstens 11(n+m)+4 elementare Operationen.
Beweis: Seien $s_1=[x_1,...,x_n]$ mit $x_1 \leq x_2 \leq ... \leq x_n$ und $s_2=[y_1,...,y_m]$ mit $y_1 \leq y_2 \leq ... \leq y_m$ die zu mischenden Folgen. Wir wissen bereits aus dem Korrektheitsbeweis, daß hierzu höchstens n+m Schleifendurchläufe erforderlich sind. Mit jedem Durchlauf werden 11 Elementaroperationen ausgeführt: In der Schleifenbedingung 3, in der bedingten Anweisung 8, darunter 3 in der Bedingung und 5 bei den Zuweisungen. Außerhalb der Schleife noch einmal 4 Operationen. Insgesamt also höchstens 11(n+m)+4 Operationen. ♦

Wie erwähnt ist 11(n+m)+4 die Komplexität des Mischens im schlimmsten Fall für zwei Folgen $s_1=[x_1,...,x_n]$ und $s_2=[y_1,...,y_m]$. Im Einzelfall kann es auch schneller gehen, z.B. für $s_1=[x_1,...,x_n]$ und $s_2=[y_1,...,y_m]$ mit $x_n \leq y_1$ (Wie schnell denn genau?).
Wir wollen uns mit dieser Komplexität zufrieden geben, denn offenbar ergibt sich beim Mischen größerer Folgen immer nur ein *linearer* zusätzlicher Aufwand. Das erscheint vertretbar. Schlimmer wäre ein *quadratischer* oder gar *exponentieller* zusätzlicher Aufwand.

4.5 Zusammenfassung

Unser Ziel war es, den Verteilungsalgorithmus aus Abschnitt 2 auf einem Computer ausführen zu lassen. Sehr schnell bekamen wir damit aber Probleme, denn der Algorithmus enthielt viele unpräzise, unvollständige Anweisungen, die alle beseitigt werden mußten. In einer intuitiven Vorgehensweise haben wir danach eine (abstrakte) Programmiersprache, genannt PRO, eingeführt. In PRO haben wir alle von einem Computer erwarteten elementaren Fähigkeiten präzise definiert: Man muß Objekte (Zahlen, Zahlenfolgen, Texte) deklarieren, benennen, einlesen, verändern und ausgeben können; man muß Sprachelemente zur Verfügung haben, um die elementaren Anweisungen und die Konstruktoren aus dem Verteilungsalgorithmus formulieren zu können.
Ein etwas komplizierteres Problem bestand darin, dem Computer mitzuteilen, daß er eine Anweisungsfolge als Prozedur oder Funktion auffassen soll. Die Präzisierung der für die korrekte Durchführung eines Aufrufs notwendigen Informationen war ebenfalls

umfangreich. Nach Klärung aller dieser Schwierigkeiten erhielten wir ein Programm, d.h., die Darstellung eines Algorithmus in der Programmiersprache PRO, die so präzise ist, daß sie von einem Computer ausgeführt werden kann.
Zum Abschluß haben wir uns kurz mit einer Charakterisierung von Programmiersprachen befaßt und Unterschiede zu natürlichen Sprachen aufgezeigt. Zentrale Begriffe waren hier die Syntax (Festlegung, welche Zeichenfolgen korrekte Programme sind) und die Semantik (Festlegung, was korrekte Programme auf dem Rechner bewirken).