6 Funktionale Modellierung

Bei der Beschreibung komplexer Systeme stößt man meist auf zwei konkurrierende Anforderungen: Einerseits will man die Beschreibung übersichtlich, intuitiv und möglichst auf den ersten Blick verständlich gestalten, andererseits soll sie möglichst aussagekräftig und detailliert sein. Dieses Dilemma kann man lösen, indem man die Beschreibung in zwei Stufen aufteilt.

1. Schritt („Black Box"-Sicht)
Zunächst gliedert man das betrachtete System in Teilsysteme (Komponenten) und beschreibt die Interaktion bzw. Kommunikation *zwischen* diesen Komponenten:

- Welche Information empfängt eine Komponente?
- Welche Information gibt sie an andere Teilsysteme weiter?

Nicht beschrieben wird die *innere* Struktur der Komponenten. Beantwortet werden also zunächst nur die beiden obigen Fragen. Oft ergibt sich die Aufteilung in Komponenten bereits aus der offensichtlichen Struktur des jeweiligen Systems. So drängt sich in größeren Firmen beispielsweise die Aufteilung in Abteilungen auf (siehe Abbildung 6.1).

Da es in dieser Sichtweise um die *Funktion* der Komponenten im Gesamtsystem geht, bezeichnet man solche Modelle als *funktionale Modelle*. Als Beschreibungstechnik verwendet man dabei meist Datenflussdiagramme (siehe Abschnitt 6.1).

2. Schritt („Glass Box"-Sicht)
Erst im zweiten Schritt untersucht man für jede der Komponenten ihre innere Struktur: Wie arbeiten die Komponenten intern? In einer Firma würde man hierbei etwa die Organisation der einzelnen Abteilungen beschreiben, z.B. durch Zustandsmodelle, Algorithmen oder Aktionsstrukturen.

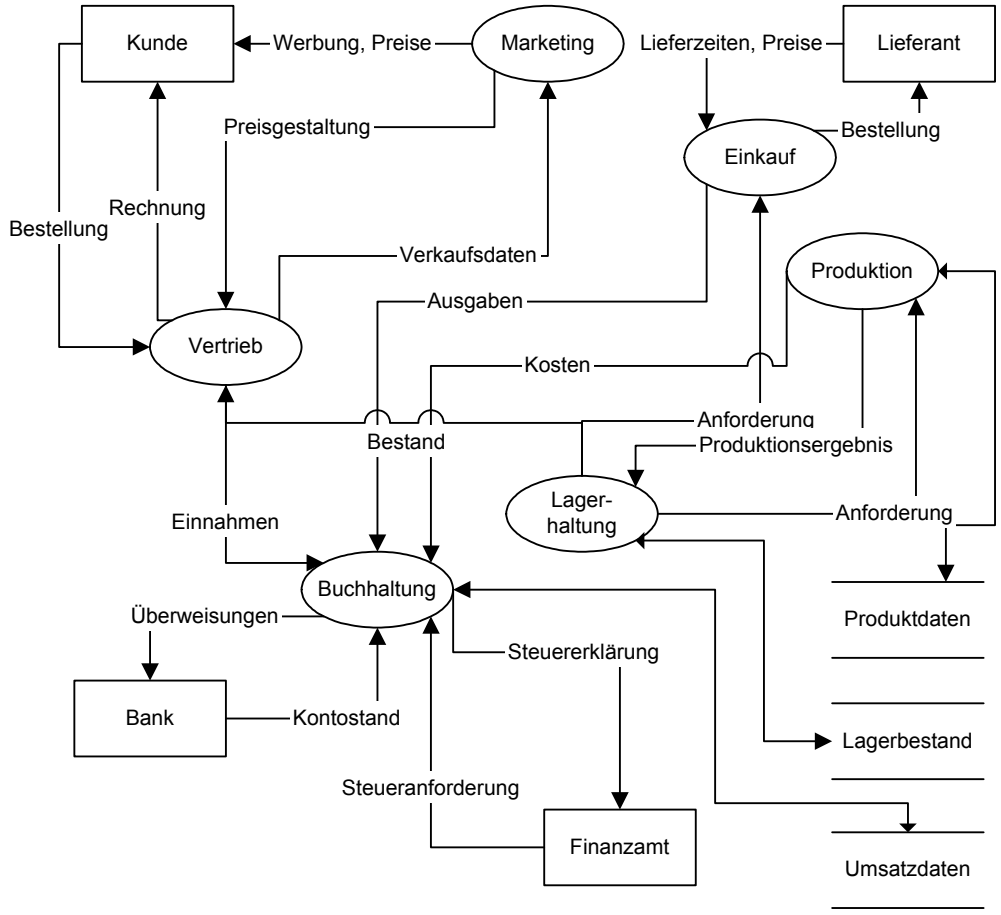

Abb. 6.1 Datenflussdiagramm eines Betriebs

6.1 Datenflussdiagramme und Programme

Am Beispiel eines verarbeitenden Betriebs wollen wir uns die Bestandteile eines funktionalen Modells (in Form eines Datenflussdiagramms) klar machen (siehe Abbildung 6.1). Zunächst haben wir (elliptisch gezeichnet) *datenverarbeitende Prozesse* vor uns: Sie nehmen Daten über Leitungen entgegen, verarbeiten diese und geben Ausgabedaten auf anderen Kanälen weiter. Die Verbindungslinien zwischen den einzelnen Prozessen symbolisieren *Datenflüsse*: Leitungen, über die in einer oder auch in beiden Richtungen Daten transportiert werden. Die Rechtecke symbolisieren *Datenquellen bzw. -senken*: An diesen Stellen kommuniziert das System mit der „Außenwelt“, d.h. es empfängt Informationen von außen oder

gibt solche nach außen ab. *Speicherkomponenten* werden mit Ober- und Unterstrich (z.B. „Produktdaten“) dargestellt.

Was haben nun Programme im Sinne von Kapitel 4 und 5 mit solchen funktionalen Modellen zu tun? Zunächst kann man ein *laufendes* Programm als Ganzes als *datenverarbeitenden Prozess* betrachten und damit als Ellipse in einem Datenflussdiagramm darstellen. Dann spiegelt dieses Diagramm die Kommunikation dieses Programms (über seine Ein- bzw. Ausgaben) mit anderen laufenden Programmen bzw. Prozessen (ebenfalls als Ellipsen dargestellt), mit den Benutzern (Rechtecke) oder mit Speicherkomponenten (mit Ober- und Unterstrich) wieder. So könnte in Abbildung 6.1 beispielsweise der Vertrieb automatisiert werden. Das entsprechende Programm würde dann Bestellungen, Preise und Bestandsdaten entgegennehmen und Einnahmen, Verkaufsdaten sowie Rechnungen ausgeben.

Bisher haben wir Programme betrachtet, die aus „einem Stück“ bestanden: Zu bestimmten Eingaben wurde durch das Programm jeweils eine bestimmte Ausgabe erzeugt (E-V-A-Prinzip, siehe Abschnitt 5.5). Dies erlaubt es, ein Programm als *eine Funktion* zu betrachten, die in Abhängigkeit von gewissen Eingabedaten aufgrund eines Algorithmus Ausgabedaten erzeugt, z.B.:

```
mittelwert(3, 5) = 4
```

In dieser *funktionalen Sichtweise* interessiert man sich vor allem für die (oft durch mathematische Ausdrücke beschriebene) *Zuordnung* zwischen Ein- und Ausgabedaten und weniger für den *Algorithmus*, der die Ausgabedaten aus den Eingabedaten berechnet („Black-Box“-Sicht, siehe oben):

```
Sortiere(„Theo“, „Anna“, „Katharina“, „Emil“) =
(„Anna“, „Emil“, „Katharina“, „Theo“).
```

In der funktionalen Sicht interessieren wir uns nur für die Tatsache, dass diese Funktion eine Liste von Zeichenketten entgegennimmt und dieselben Zeichenketten in (aufsteigend) sortierter Reihenfolge ausgibt, aber *nicht* für den speziellen *Algorithmus,* der diese Sortierung erzeugt (z.B. Sortieren durch Einfügen, Bubblesort, Quicksort etc.).

In einem Datenflussdiagramm kann man also (wie oben beschrieben) ein Programm (als Funktion) durch einen informationsverarbeitenden Prozess (Ellipse) symbolisieren, wie z.B. unser Sortierprogramm in Abbildung 6.2.

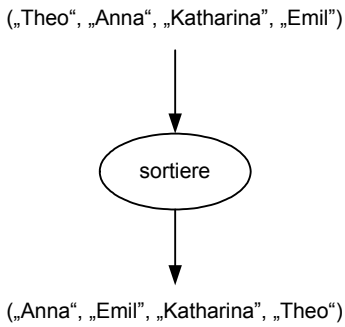

Abb. 6.2 Ein Programm als datenverarbeitender Prozess

6.2 Aufteilung von Programmen in Unterprogramme

Neben der Kommunikation mit der Außenwelt kann ein funktionales Modell aber auch die *innere Struktur* eines Programms darstellen. Dies ist vor allem bei der Aufteilung eines Programms in relativ selbständige Teile hilfreich. Diese Teile nennt man **Unterprogramme**.

Beispiel: Ein Programm zur Bruchrechnung könnte z.B. in folgende Unterprogramme aufgeteilt werden:

- *Eingabe* von zwei Brüchen (jeweils Zähler und Nenner),
- *Ausgabe* eines Bruches,
- *Kürzen* eines Bruches,
- *Erweitern* eines Bruches mit einer bestimmten ganzen Zahl,
- *Kehrwertbildung* eines Bruches,
- *Invertierung* des Vorzeichens eines Bruches,
- *Addition* zweier Brüche,
- *Multiplikation* zweier Brüche.

Die Nutzung dieser Unterprogramme zu weiteren Berechnungen kann wiederum durch Datenflussdiagramme dargestellt werden. Abbildung 6.3 zeigt ein funktionales Modell für die Division zweier Brüche unter Benutzung einiger dieser Unterprogramme.

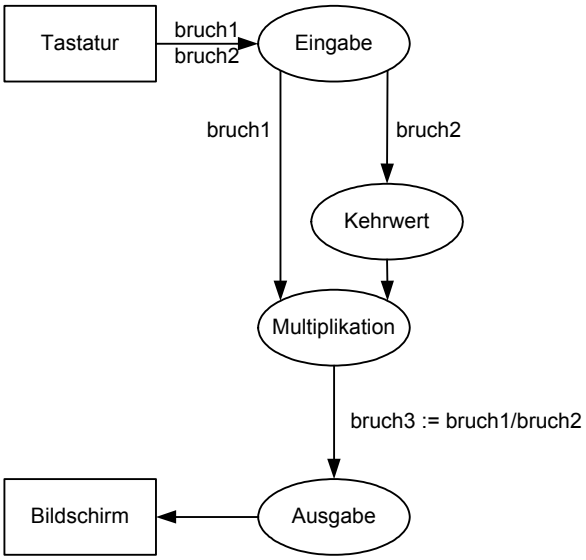

Abb. 6.3 Funktionales Modell für die Division zweier Brüche

Eine solche Aufteilung in Unterprogramme kann aus verschiedenen Gründen nützlich sein:

- *Arbeitsteilung* (Teamarbeit): Die Unterprogramme können gleichzeitig von je einer Arbeitsgruppe erstellt oder verändert werden.
- *Wiederverwendung* von Code: Ein Unterprogramm kann von mehreren Stellen des Hauptprogramms aus aufgerufen werden. So kann man seinen Programmtext mehrfach nutzen, ohne ihn mehrfach schreiben zu müssen, was vor allem Probleme bei der nachträglichen Veränderung dieses Textes vermeidet.
- *Abstraktion* von der konkreten Aufgabenstellung: Durch *Parametrisierung* (siehe Abschnitt 6.6) kann ein Unterprogramm auf eine ganze Klasse von Aufgabenstellungen angewandt werden.
- Die geschickte Aufteilung in Unterprogramme (mit aussagekräftigen Bezeichnern) kann die *Übersichtlichkeit* eines Programmtextes enorm steigern, da so der Hauptalgorithmus sehr knapp dargestellt werden kann.

Meist folgt man bei der Erstellung solcher aufgeteilter Programme dem Schema der *schrittweisen Verfeinerung*: Man unterteilt die Aufgabenstellung zunächst in einige wenige grobe Teilaufgaben. In den folgenden Schritten wird diese Aufteilung weiter verfeinert, bis man ein System von Teilaufgaben erhält, die jeweils durch einen (Teil-) Algorithmus lösbar sind. Danach können diese Teilalgorithmen als Unterprogramme implementiert und schließlich die Unterprogramme zu einem Gesamtsystem kombiniert werden. Bezogen auf unser Datenflussdiagramm aus Abbildung 6.1 würde das z.B. der Aufteilung des Prozesses "Produktion" bei einer Automobilfabrik in die Teilprozesse "Produktion Fahrgestell", "Produktion Karosserie" und "Produktion Antrieb" und damit einer Verfeinerung des Modells entsprechen.

Leider kann man ein Programm durch *ungeschickte Aufteilung* und bzw. oder schlechte Umsetzung dieser Aufteilung auch sehr unübersichtlich gestalten. Insbesondere sollte man darauf achten, dass für jedes Unterprogramm möglichst klar wird, welche Variablen es verändert, um unerwünschte Nebeneffekte zu vermeiden. Davon wird weiter unten noch ausführlich die Rede sein.

Die Bezeichnung von Unterprogrammen als *Funktionen* bzw. *Prozeduren* ist in der Literatur leider nicht ganz einheitlich. Wir werden in diesem Buch Unterprogramme zunächst als *Prozeduren* auffassen und daraus Kriterien entwickeln, die eine Bezeichnung als *Funktion* rechtfertigen.

In objektorientierten Sprachen werden Variablen und Unterprogramme (Prozeduren oder Funktionen) zu *Objekten* zusammengefasst. Die Struktur dieser Objekte wird in *Klassenbeschreibungen* festgehalten. Dabei bezeichnet man die Funktionen eines Objektes bzw. einer Klasse meist als *Methoden*. Mehr darüber erfahren Sie in der Literatur über „Objektorientierte Modellierung“.

6.3 Deklaration und Aufruf von Prozeduren

Am Beispiel eines Programms zum Zahlenraten soll nun die Verwendung von Prozeduren in *PPS* erklärt werden. Die Regeln für Deklarationen von Unterprogrammen finden Sie im Anhang zur Syntax von *PPS*.

```
program prozedurtest:
var nat eingabe, ratezahl := 17;
var bool erraten := false;
// Deklaration der Prozedur „treffer“
procedure treffer:
begin
output(„* WIR GRATULIEREN! *“);
output(„**********************“);
output(„Sie haben die Zahl erraten!“)
endproc;
// Deklaration der Prozedur „daneben“
procedure daneben:
begin
output(„#             Leider daneben!         #“);
output(„############################“);
output(„Bitte versuchen Sie es noch mal!“)
endproc;
begin
while erraten = false do
      input(eingabe);
      // Aufruf der beiden Prozeduren über ihren Namen:
```

```
      // „erraten", „daneben"
      if eingabe = ratezahl then treffer; erraten := true
      else daneben
      endif
endwhile
end.
```

In einer Prozedurdeklaration werden alle wesentlichen Eigenschaften der Prozedur, insbesondere ihr *Algorithmus*, vereinbart. Danach steht die Prozedur als *neue Anweisung* zum Aufruf zur Verfügung. Die Deklaration alleine löst allerdings nur die Organisation von Speicherplatz und einiger anderer Verwaltungsmaßnahmen aus. Der eigentliche Start einer Prozedur (im Sinne des Ablaufs ihres Algorithmus) wird durch den *Namen* der Prozedur (innerhalb des Hauptprogramms oder eines anderen Unterprogramms) ausgelöst.

6.4 Globale und lokale Variable

In Unterprogrammen kann man spezielle Variablen für interne Zwecke deklarieren. Diese Deklaration ist dann nur *innerhalb* dieses Unterprogramms gültig. Solche Variablen heißen deshalb *lokale Variablen*.

```
procedure quadratzahlen:
var nat i;
begin
for i := 1 to 100 do output(i*i) endfor
endproc
```

Falls man Variablen nur für Berechnungen *innerhalb* eines Unterprogramms benötigt, sollten diese auch nur *lokal* deklariert werden (so wie oben die Variable *i*). Das hat (verglichen mit einer Deklaration im Hauptprogramm *außerhalb* des Unterprogramms) zwei Vorteile:

1. Der Speicherplatz für diese Variablen wird nach der Beendigung des Unterprogramms wieder freigegeben.
2. Diese Variablen sind außerhalb des Unterprogramms nicht sichtbar (siehe Abschnitt 6.5) und können daher dort auch nicht verändert werden.

Eine Variable heißt also (von einem bestimmten Unterprogramm aus gesehen) *lokal*, wenn sie nur innerhalb dieses Unterprogramms deklariert ist. Sie heißt global, wenn sie auch *außerhalb* des Unterprogramms gültig (d.h. deklariert) ist:

```
program mitarbeiter:
var nat anzahl;
var [1:100] array string name;

procedure einstellen:
```

```
var string einname;
begin
input(einname);
if anzahl < 100
// Achtung, schlechter Stil:
// Veränderung globaler Variablen!
then anzahl := anzahl + 1; name[anzahl] := einname
else output(„Maximale Mitarbeiterzahl erreicht“)
endif
endproc;

begin
…
// Hier werden evtl. die globalen Variablen
// anzahl und name[anzahl] von einstellen verändert:
einstellen;
…
end.
```

In diesem Programm sind von der Prozedur *einstellen* aus gesehen

- die Variable *einname* lokal,
- die Variablen *anzahl* und *name* dagegen global.

Falls in einem Unterprogramm eine lokale Variable mit dem *gleichen Bezeichner* wie eine globale Variable deklariert wird, so kann über diesen Bezeichner innerhalb dieses Unterprogramms auch nur auf die *lokale* Variable zugegriffen werden. Die globale Variable wird dann von der lokalen „*verschattet*“:

```
program verschattung:
var nat i;
…
procedure quadratzahlen:
var nat i;
begin
for i := 1 to 100 do output(i*i) endfor
endproc;
…
```

Innerhalb der Prozedur *quadratzahlen* ist die globale Variable *i* nicht sichtbar, da sie von der gleichnamigen lokalen verschattet wird.

6.5 Bindung und Gültigkeit

Je nach dem Ort ihrer Deklaration kann man auf die Variablen eines Programms an manchen Stellen zugreifen, an anderen nicht. Der Bereich, innerhalb dessen eine Deklaration überhaupt bekannt ist, heißt *Bindungsbereich* der Variablen. Der Bezeichner der Variablen ist in diesem Bereich an diese Deklaration *gebunden* und kann nicht beliebig anderweitig verwendet werden.

Da es die Möglichkeit der Verschattung einer globalen Variablen durch eine gleichnamige lokale gibt (siehe oben), kann es vorkommen, dass man auf eine Variable an manchen Stellen ihres Bindungsbereichs nicht zugreifen kann. Man unterscheidet daher den *Gültigkeitsbereich* (Sichtbarkeitsbereich) einer Variablen (innerhalb dessen man auf sie zugreifen kann) von ihrem *Bindungsbereich*.

Beispiel: Im obigen Programm *mitarbeiter* sind Bindungs- und Gültigkeitsbereiche identisch (siehe Tabelle 6.1):

Tab. 6.1 Bindungs- und Gültigkeitsbereich im Programm mitarbeiter

Variable	*Bindungsbereich*	*Gültigkeitsbereich*
anzahl	Hauptprogramm	Hauptprogramm
name	Hauptprogramm	Hauptprogramm
einname	Prozedur *einstellen*	Prozedur *einstellen*

Beispiel: Im Programm *verschattung* unterscheiden sich dagegen Bindungs- und Gültigkeitsbereich der globalen Variablen *i*:

Tab. 6.2 Bindungs- und Gültigkeitsbereich im Programm Verschattung

Variable	*Bindungsbereich*	*Gültigkeitsbereich*
globale Variable *i*	Hauptprogramm	Hauptprogramm mit Ausnahme der Prozedur *quadratzahlen*
lokale Variable *i*	Prozedur *quadratzahlen*	Prozedur *quadratzahlen*

Wegen der Möglichkeit der Schachtelung von Prozeduren (man kann innerhalb einer Prozedur andere Prozeduren deklarieren, die dann nur *lokal* innerhalb dieser Prozedur bekannt sind), ist es oft nicht ganz einfach, den Gültigkeitsbereich einer Variablen festzustellen.

Die folgende Tabelle zeigt ein Beispiel für Bindungs- und Gültigkeitsbereiche einer Reihe von Variablen.

B = Bindungsbereich **G = Gültigkeitsbereich**	hvar[1]		avar1		ivar		avar2		hvar[2]	
program hauptprogramm:										
var nat hvar;	**B**	**G**								
procedure aussen1:	**B**	**G**								
var nat avar1;	**B**	**G**	**B**	**G**						
//Lokale Prodezur innerhalb von aussen1:	**B**	**G**	**B**	**G**						
procedure innen:	**B**	**G**	**B**	**G**						
var nat ivar;	**B**	**G**	**B**	**G**	**B**	**G**				
begin	**B**	**G**	**B**	**G**	**B**	**G**				
…	**B**	**G**	**B**	**G**	**B**	**G**				
endproc;	**B**	**G**	**B**	**G**	**B**	**G**				
begin	**B**	**G**	**B**	**G**						
…	**B**	**G**	**B**	**G**						
endproc;	**B**	**G**	**B**	**G**						
	B	**G**								
procedure aussen2:	**B**	**G**								
var nat avar2, hvar;	**B**	[3]					**B**	**G**	**B**	**G**
begin	**B**	[3]					**B**	**G**	**B**	**G**
…	**B**	[3]					**B**	**G**	**B**	**G**
endproc;	**B**	[3]					**B**	**G**	**B**	**G**
begin	**B**	**G**								
…	**B**	**G**								
end.	**B**	**G**								

[1] Globale Variable des Hauptprogramms

[2] Lokale Variable des Unterprogramms *aussen2* (mit gleichem Namen wie die globale Variable *hvar*)

[3] Hier wird die globale Variable *hvar* durch die gleichnamige lokale verschattet

Über die Möglichkeit, Variablen (oder auch Sorten und Unterprogramme) in Unterprogrammen zu deklarieren, bieten manche Programmiersprachen auch die Möglichkeit, Deklarationen und Anweisungen mit Hilfe von speziellen Klammerelementen (z.B. *begin* bzw. *end* oder Paare geschweifter Klammern) zu *Blöcken* zusammenzufassen, z.B.:

```
begin
var nat x, y;
y := x * x;
output(y)
end
```

Diese Blöcke können (ebenso wie Unterprogramme) wiederum geschachtelt und/oder mit Unterprogrammen kombiniert werden. Die obigen Aussagen für die Gültigkeit bzw. Bindung

von Variablen gelten dann entsprechend auch für solche Blöcke. Wir werden in diesem Modul aber auf deren Verwendung verzichten.

6.6 Parameter

Wie wir in Kapitel 2 erfahren haben, beschreibt ein Algorithmus in der Regel eine Lösung für eine ganze *Klasse* von Aufgaben. Ein Algorithmus zur Berechnung der Quadratwurzel einer Zahl kann beispielsweise für alle positiven Zahlen angewandt werden. Implementiert man einen Algorithmus in einer Prozedur, so will man diese natürlich auch auf alle Aufgabenstellungen dieser Klasse anwenden können.

Um diese Flexibilität in der Anwendung zu erreichen, verwendet man *Parameter*: Das Unterprogramm erhält für jeden zu übergebenden Wert einen *formalen Parameter* im Kopf der Deklaration. Diese formalen Parameter dienen als *Platzhalter* für konkrete Werte, die der Prozedur bei ihrem Aufruf übergeben werden, hier am Beispiel der Berechnung der Potenz x^y mit ganzen Zahlen x, y und y>0:

```
procedure x_hoch_y (nat px, py):
var nat ergebnis, i;
begin
ergebnis := 1;
for i := 1 to py do
      ergebnis := ergebnis * px
endfor
output(ergebnis)
endproc
```

Ein Aufruf könnte dann z.B. lauten:

```
x_hoch_y(2,3)
```

Das Ergebnis wäre die Ausgabe der Zahl 8.

In vielen Sprachen (z.B. in unserer Sprache *PPS* oder in *Java*) werden bei der Deklaration von Unterprogrammen die *Sorten* der formalen Parameter festgelegt. Dann sollten beim Aufruf des Unterprogramms auch nur Werte (bzw. im folgenden Abschnitt 6.7 Speicheradressen) der jeweils festgelegten *Sorte* an die Parameter übergeben werden (z.B. natürliche Zahlen an Parameter der Sorte **nat**). Die Reaktion auf Verletzungen dieser Regel hängt von den beiden Sorten und der jeweiligen Programmiersprache ab. So ist z.B. die Übergabe von ganzzahligen Werten (**nat**) an Parameter vom Typ **float** meist unproblematisch, die Umkehrung dagegen meist nicht. Wir gehen in diesem Skript grundsätzlich davon aus, dass diese Bedingung eingehalten wird.

6.7 Ergebnisübergabe

Das obige Unterprogramm *x_hoch_y* kann zwar das Ergebnis der Berechnung von x^y direkt (z.B. am Bildschirm) ausgeben, es jedoch nicht dem Hauptprogramm für weitere Berechnungen zur Verfügung stellen, wie es z.B. zur Aufsummierung aller 3er-Potenzen von 3^1 bis 3^{10} notwendig wäre.

Wie kann man die Ergebnisse eines Unterprogramms an das aufrufende Programm übermitteln? Dafür gibt es im Wesentlichen drei Möglichkeiten:

1. Schreibzugriff auf *globale Variable*,
2. Nutzung formaler *Ausgangs*parameter,
3. Implementierung des Unterprogramms als *Funktion*.

Diese drei Möglichkeiten sollen nun eingehender betrachtet werden.

6.7.1 Schreibzugriff auf globale Variable

Globale Variablen sollten nur in Notfällen (wie z.B. dem Fall, dass eine relativ zum Speicherangebot sehr große Mengen einzelner Daten übergeben werden soll) zur Rückgabe der Ergebnisse von Unterprogrammen verwendet werden. Da diese Vorgehensweise zu sehr unübersichtlichen Abläufen (lokal „unsichtbare“ Veränderung von Variablen) führt, sollte sie ansonsten jedoch tunlichst vermieden werden (siehe auch das obige Programm *mitarbeiter*).

```
program globale_rückgabe:
var nat global;

procedure x_hoch_y (nat px, py):
var nat ergebnis, i;
begin
ergebnis := 1;
for i := 1 to py do
      ergebnis := ergebnis * px
endfor
// ACHTUNG: Veränderung globaler Variablen:
global := ergebnis
endproc;

// Beginn des Hauptprogramms
begin
global := 1;
// Folgender Aufruf verändert global,
// was an dieser Stelle nicht erkenntlich ist:
x_hoch_y(2, 3);
output(global)
end.
```

Das Programm würde in diesem Fall die Ausgabe 8 liefern. Wenn man sich vor Augen führt, dass große kommerzielle Programme nicht selten Hunderttausende bis Millionen Programmzeilen umfassen, kann man sich vorstellen, wie problematisch eine lokal nicht erkennbare (weil im Aufruf einer Prozedur versteckte) Veränderung einer globalen Variablen sein kann. Solche nicht direkt (d.h. an der Aufrufstelle) sichtbaren Nebenwirkungen eines Unterprogramms nennt man auch *Seiteneffekte*.

Trotz aller möglichen Probleme ist man in bestimmten Fällen zu Schreibzugriffen auf globale Variable gezwungen. Ein Beispiel hierfür wäre die Zählung der laufenden Prozesse einer bestimmten Funktion.

6.7.2 Ausgangsparameter

Eine zweite (in der Regel günstigere) Möglichkeit zur Rückgabe der Ergebnisse eines Unterprogramms bieten *Ausgangsparameter* (bzw. *Ergebnisparameter*). Das sind spezielle Parameter, die in den meisten Programmiersprachen auch als solche gekennzeichnet werden müssen, in *PPS* (wie in *Pascal* und *Modula*) durch das vorangestellte Schlüsselwort **var**:

```
program ausgangsparameter:
var nat globalx := 2, globaly := 3, globalz;

procedure x_hoch_y (nat px, py, var nat pz):
var nat ergebnis, i;
begin
ergebnis := 1;
for i := 1 to py do
      ergebnis := ergebnis * px
endfor
// Zuweisung an Ausgangsparameter pz,
// kenntlich durch „var“ in der Parameterliste
pz := ergebnis
endproc;

// Beginn des Hauptprogramms
begin
// Folgender Aufruf weist das Ergebnis
// der Variablen „globalz“ zu
x_hoch_y(globalx, globaly, globalz);
output(globalz)
end.
```

Auch hier würde das Programm wieder die Ausgabe 8 liefern. Im Gegensatz zur obigen Rückgabe des Ergebnisses mittels globaler Variable ist hier wenigstens sichtbar, dass die Variable *globalz* zumindest *beteiligt* ist (wenn auch die Rolle als Ausgangsvariable im *Aufruf* nicht sichtbar ist). Über die Kennzeichnung der Ausgangsparameter durch **var** im Kopf (der

ersten Zeile) der Prozedurdeklaration könnte man außerdem automatisch eine Liste der Unterprogramme mit Ausgangsparametern erstellen und so im Programmcode lokalisieren, wo ein Schreibzugriff möglich ist (was bei Schreibzugriffen auf globale Variablen nicht unbedingt machbar ist).

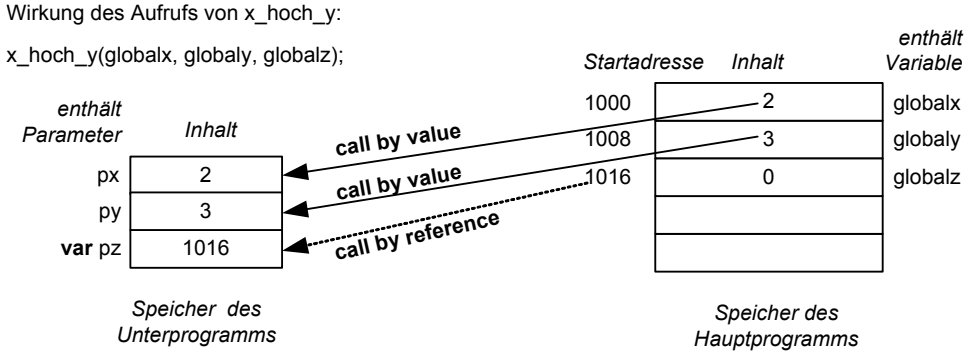

Abb. 6.4 Vergleich der Übergabemechanismen für Parameter

Während es bei *Eingangsparametern* genügt, wenn nur der aktuelle *Wert* für den Parameter an die Prozedur übergeben wird („call by value"), muss bei Ausgangsparametern die *Anfangsadresse* des Speicherbereichs einer globalen Variablen übergeben werden, damit dort ein Schreibzugriff stattfinden kann („call by reference"), siehe auch Abbildung 6.4.

6.7.3 Funktionskonzept

Das dritte (und eleganteste) Übergabekonzept ist die Implementierung des Unterprogramms als *Funktion*: Es wird als (neu definierte) Funktion implementiert und aufgerufen, die (genau!) einen Rückgabewert zurückliefert. Diese Rückgabe wird *innerhalb der Deklaration* durch die **return**-Anweisung festgelegt.

Für die Deklaration solcher Funktionen (in diesem Sinne) gilt (im Vergleich zu Prozeduren) eine etwas veränderte Syntax:

1. Im Kopf der Deklaration wird anstatt **procedure** das Schlüsselwort **function** verwendet.
2. Nach der Liste der Parameter wird (durch einen Doppelpunkt abgetrennt) die *Sorte* des *Rückgabewertes* der Funktion festgelegt.

Falls möglich, sollte dieser Variante immer der Vorzug vor den anderen beiden gegeben werden, da nur hier die zuweisende Wirkung auf die Variable (z.B. *globalz*) im Aufruf offensichtlich ist:

```
program funktionskonzept:
var nat globalx := 2, globaly := 3, globalz;

function x_hoch_y (nat px, py): nat
var nat ergebnis, i;
begin
ergebnis := 1;
for i := 1 to py do
      ergebnis := ergebnis * px
endfor
// Rückgabe des Ergebnisses als Wert der Funktion
return ergebnis
endfct;

// Beginn des Hauptprogramms
begin
// Folgender Aufruf weist das Ergebnis der Funktion
// der Variablen „globalz" zu
globalz := x_hoch_y(globalx, globaly);
output(globalz)
end.
```

Mit Hilfe von Funktionen können viele Aufgaben sehr kompakt formuliert werden, z.B. die oben erwähnte Aufsummierung aller 3er-Potenzen zwischen 3^1 und 3^{10}:

```
…
var nat k, summe := 0;
for k := 1 to 10 do
      summe := summe + x_hoch_y(3, k)
endfor
output(summe);
…
```

Dieses Konzept liegt sehr nahe am *funktionalen Programmierstil*, der im Kapitel 7 ausgiebig besprochen wird. Dort werden wir Funktionen allerdings grundsätzlich als *Ausdrücke* und nicht mehr als *Folgen von Anweisungen* behandeln.

6.8 Module

In der Regel wird der Programmcode in modernen Programmiersprachen in mehrere Module aufgeteilt. Das hat zahlreiche Vorteile, u.a.:

- Übersichtlichkeit: Überlange Programmtexte lassen sich schwer lesen. Auch können so korrespondierende Codestellen aus mehreren Modulen in mehreren Fenstern verglichen und geändert werden, ohne dauernd im Programmtext blättern zu müssen.
- Arbeitsteilung: Die Arbeit kann nach Modulen auf mehrere Personen, Teams, Abteilungen, Firmen aufgeteilt werden.
- Kapselung: Durch selektive Zuteilung von Schreibrechten kann man genau festlegen, wer welchen Modul ändern darf.

Solche Module bestehen in der Regel aus einer *Reihe von Deklarationen* für Sorten, Variablen, Prozeduren und Funktionen. Besonders in der Objektorientierten Programmierung ist die Verwendung solcher Module weit verbreitet. Hier enthalten sie meist Klassendefinitionen. Mehr darüber erfahren Sie in der Literatur über „Objektorientierte Modellierung und Programmierung“.

6.9 Unterprogramme in *Python*

Zunächst ist festzustellen, dass *Python* in der Deklaration keine Unterschiede zwischen Funktionen und Prozeduren macht. Wir werden also alle *Python*-Unterprogramme als Funktionen bezeichnen. Funktionen müssen in einem Script deklariert werden, wie z.B. die folgende „prozedurartige“ Funktion (ohne Rückgabewert) *quadratzahlen* in *quadratzahlen.py*:

```
# quadratzahlen.py

def quadratzahlen():
    for i in range(1,101): print i*i
```

In der Interpreterumgebung wird die Funktion dann folgendermaßen aufgerufen:

```
>>> quadratzahlen()
1
4
9
..
9801
10000
```

Dabei ist zu beachten, dass nach dem Bezeichner der Funktion sowohl in der Deklaration als auch im Aufruf ein rundes Klammerpaar folgen muss (auch wenn keine Parameter benutzt werden). Im Folgenden machen wir den Aufruf der jeweiligen Funktionen nur noch durch das Prompt des *Python*-Interpreters (>>>) deutlich.

Wie aufgrund des großzügigen Umgangs von *Python* mit Typen nicht anders zu erwarten, muss bei der Deklaration von Funktionen der Typ der Parameter nicht angegeben werden.

Die Rückgabe des Ergebnisses sollte, wie in Abschnitt 6.7 ausgeführt, wenn irgend möglich über *return* vollzogen werden:

```
def x_hoch_y(px,py):
    if py == 0:return 1
    else:
        ergebnis = 1
        for i in range(1,py+1): ergebnis = ergebnis*px
        return ergebnis

>>> x_hoch_y(3,4)
81
```

Mit einer Ausnahme (siehe unten) sind in *Python* alle Parameter Eingangsparameter (call-by-value-Übergabe):

```
def partest(px):
    px = 99
    return px

>>> x = 5
>>> partest(x)
99
>>> x
5
```

Der Wert der Variablen *x* wurde durch den Aufruf *partest(x)* also nicht verändert, obwohl innerhalb von *partest* ein Schreibzugriff auf den Parameter *px* stattfand. In *Python* gibt es kein Schlüsselwort (wie **var** in *PPS* oder *Pascal*), das die Kennzeichnung eines Parameters als Ausgangsparameter ermöglicht. Dennoch erlaubt die einzige Ausnahme von diesem beinahe durchgehenden *call-by-value* Konzept die Verwendung von Ausgangsparametern:

```
def partest(px):
    px[0] = 99
    return px

>>> partest(liste)
[99, 2, 3]
>>> liste
[99, 2, 3]
```

Komponenten von Listen werden also über *call-by-reference* (ganze Listen dagegen ebenfalls nur über *call-by-value*) übergeben und können so als Ausgangsparameter genutzt werden (indem man die zu verändernde Variable als *Komponente einer Liste* anlegt):

```
def x_hoch_y(px,py, erg):
    if py == 0:ergebnis = 1
```

```
    else:
        ergebnis = 1
        for i in range(1,py+1): ergebnis = ergebnis*px
    erg[0] = ergebnis

>>> erg = [0]
>>> x_hoch_y(3,4,erg)
>>> erg
[81]
```

In *Python* ist grundsätzlich kein Zugriff aus einer Funktion heraus auf globale Variablen möglich, es sei denn, dass diese Variablen ausdrücklich als *global* gekennzeichnet werden:

```
def writeglobal(px):
    global gx
    gx = px
    return "fertig!"

>>> gx = 5
>>> writeglobal(99)
'fertig!'
>>> gx
99
```

6.10 Module in *Python*

Python verfügt über eine Unzahl von Modulen für beinahe jeden Zweck: von der Netzwerkprogrammierung bis zur Mathematik. Diese Module stellen Bibliotheken für Kostanten, Datenstrukturen und Funktionen dar, aus denen man sich bei Bedarf bedienen kann. Nähere Informationen finden Sie in der *Python*-Dokumentation *Globale Module Index* (von *Idle* aus über das *Help*-Menü zugänglich).

Am Beispiel des Moduls *random*, der für die Erzeugung von Zufallszahlen zuständig ist, wollen wir den Mechanismus der Einbindung kurz aufzeigen:

```
# wuerfeln.py
import random
zahl = random.randint(1,6)
print zahl
ratezahl = input("Bitte raten Sie die gewuerfelte Zahl (1-6):
")
if ratezahl == zahl: print "Erraten!"
else: print "Leider daneben!"

>>>
```

```
Wir wuerfeln!
Bitte raten Sie die gewuerfelte Zahl (1-6): 1
Erraten!
```

Der Modul *random* wird also durch die Anweisung *import.random* zugänglich gemacht. Danach kann auf seine Komponenten zugegriffen werden, z.B. mittels *random.randint(a,b)* auf die Funktion *randint*, die eine ganze Zufallszahl zwischen (jeweils inklusive) *a* und *b* liefert.

6.11 Aufgaben

Aufgabe 6.1: Erstellen Sie jeweils ein Datenflussdiagramm für die durch folgende Nutzungsfälle beschriebenen Systeme:

a) Flugbuchungssystem: Ein Kunde bucht von einem Reisbüro aus einen Flug einer bestimmten Fluggesellschaft und bezahlt mit Kreditkarte.

b) Kraftfahrzeug: Ein ABS-System steuert das Bremsverhalten bei einer Vollbremsung.

c) Zwei Personen telefonieren mit ihrem Mobiltelefon über eine Landes- (Provider-) grenze hinweg.

d) Eine Frau bestellt sich bei einem Versandhaus per Telefon ein Kleid. Die Rechnung wird von ihrem Konto abgebucht.

e) Ein LKW-Mautsystem bucht die Mautgebühren nach Feststellung der Fahrtstrecken über GPS vom Konto der Fuhrunternehmen ab.

f) Vereinfachte Lohnsteuerberechnung: Vom Bruttolohn werden die bezahlte Kirchensteuer, die Vorsorgepauschale von 1000 € sowie die Werbungskosten abgezogen. Der Rest r wird mit einem festen Steuersatz von 25% besteuert.

g) Notendurchschnitt Ihrer Schulleistungen in der 10. JGSt.: Die Durchschnittsnote ergibt sich aus dem schriftlichen und dem mündlichen Durchschnitt im Verhältnis 2:1. Der schriftliche Durchschnitt ergibt sich aus 4 Schulaufgaben, der mündliche aus 2 mündlichen Noten und 2 Extemporalien zu gleichen Teilen.

h) Stellen Sie den folgenden Term als Datenflussdiagramm dar: C = Quadratwurzel(Summe(Quadrat(a), Quadrat(b))). Es handelt sich um eine Umformung des Satzes von Pythagoras: $a^2 + b^2 = c^2$.

Aufgabe 6.2: Die Firma CallCar betreibt ein innovatives, automatisiertes System zur Vermietung von Automobilen. An 5 Flughäfen (München, Köln/Bonn, Berlin, Hamburg und Frankfurt) kann man Automobile ausleihen und sie auch wieder abgeben. Nach der Anmeldung als Kunde (unter Einsendung einer amtlich beglaubigten Führerscheinkopie) kann man Fahrzeu-

ge bis spätestens einen Tag vor der Abholung (unter Angabe von Name, Vorname, Adresse, Kreditkartendaten, Fahrzeugklasse, Tag, Uhrzeit und Ort von Abholung und Rückgabe) über das Internet reservieren. Falls das gewünschte Fahrzeug (oder ggf. nach Rückfrage eine Ersatzklasse) verfügbar ist, erhält man eine Reservierungsnummer. Mit dieser und der bei der Reservierung angegebenen Kreditkarte erhält man an einem Automaten in der Flughafenhalle den Fahrzeugschlüssel sowie die Parkplatznummern für Abholung und Abgabe und kann danach in der Parkgarage das Fahrzeug abholen. Die Rückgabe erfolgt, indem man das Fahrzeug auf dem vorgesehenen Parkplatz abstellt und die Schlüssel in einen Briefkasten wirft. Falls das Fahrzeug nicht mit vollem Tank oder beschädigt zurückgegeben wird, belastet CallCar die Kreditkarte entsprechend. Die Fahrzeuge enthalten Sender, die den aktuellen Aufenthaltsort (über ein *Global Positioning System*) und Ausleihstatus alle 5 Minuten an CallCar melden.

a) Erstellen Sie ein Datenflussdiagramm, das die für die obigen Vorgänge notwendigen Komponenten und Datenflüsse des Systems beschreibt. Sie können sich bei den Flughäfen und Fahrzeugen dabei jeweils auf *einen* Repräsentanten beschränken.

b) Beschreiben Sie den Ausleihvorgang durch ein Zustandsdiagramm.

Aufgabe 6.3: Programmieren Sie in *PPS* und/oder *Python* ein Programm zur Bruchrechnung unter Verwendung der folgenden Funktionen (Unterprogramme):

- Eingabe eines Bruches, Eingabe zweier Brüche (jeweils Zähler und Nenner),
- Ausgabe eines Bruches,
- Kürzen eines Bruches,
- Erweitern eines Bruches mit einer bestimmten ganzen Zahl,
- Kehrwertbildung eines Bruches,
- Addition bzw. Subtraktion zweier Brüche,
- Multiplikation zweier Brüche,
- Division zweier Brüche.

Sie können dabei auf den Euklidischen Algorithmus zur Berechnung des ggT zweier Zahlen (siehe Aufgabe 5.5) sowie auf die Beziehung

$$ggT(a, b) = ab/kgV(a, b)$$

zurückgreifen.

Legen Sie das Hauptprogramm in Form eines Menüs an, das dem Benutzer die Wahl aus den o.g. Funktionen lässt.

Aufgabe 6.4: Gegeben ist folgendes Programm (mit Zeilennummern) in *PPS*, das eine Potenz x^y berechnet:

```
1     program Potenz:
2     var nat x, y;
3     procedure berechnen (nat x, y):
4           procedure x_hoch_y (nat x, y):
```

```
5           var nat erg, i;
6           begin
7           erg := 1;
8           for i := 1 to y do
9                 erg := erg * x
10          endfor
11          output(erg);
12          endproc
13    begin
14    x_hoch_y(x, y);
15    endproc
16    begin
17    output("Geben Sie bitte die erste Zahl ein:");
18    input(x);
19    output("Geben Sie bitte die zweite Zahl ein:");
20    input(y);
21    berechnen(x, y);
22    end.
```

Geben Sie für jedes Auftreten einer Variablen die Zeile ihrer Deklaration an! Geben Sie dann (analog zur Tabelle in Abschnitt 6.5) für *jede* Variable ihren Bindungsbereich und ihren Gültigkeitsbereich an.