

# Kapitel 2

## Rekursion, Induktion, Iteration

Die bisher vorgestellten Scheme-Prozeduren generieren einfach strukturierte Berechnungsprozesse. Diese Prozesse sind dabei immer recht geradlinig gewesen, wie Straßen von A nach B: eine ausreichende Anzahl von Schritten führt immer zum Ziel. Gelegentlich gibt es Umleitungen verursacht durch Conditionals.

Die Probleme, die mit solchen Prozeduren zu lösen sind, haben nur begrenzten Appeal. Viel interessanter sind Probleme variabler Größe, wo bei immer größeren Eingaben auch immer mehr Berechnungen durchzuführen sind. Die Bearbeitung solcher Probleme erfordert die Wiederholung von Rechenschritten. In der Programmierung heißt der Mechanismus dafür *Rekursion*. Sie ist das Thema dieses Kapitels.

### 2.1 Rekursion

Die gemeinhin übliche Definition für die Fakultät ist die folgende:

$$n! := n \cdot (n - 1) \cdot \dots \cdot 1$$

Diese Definition ist zu unpräzise für eine Formulierung als Computerprogramm. Eine induktive Definition ist besser:

$$n! := \begin{cases} 1 & \text{für } n = 1 \\ n \cdot (n - 1)! & \text{sonst} \end{cases}$$

Das entsprechende Programm ist nun einfach hinzuschreiben:

```
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* n (factorial (- n 1))))))
```

Wie funktioniert es? Mit dem Substitutionsmodell läßt sich der Berechnungsprozeß einer Anwendung von `factorial` darstellen:

```
(factorial 4)
=> ((lambda (n) ...) 4)
=> (if (= 4 1) 1 (* 4 (factorial (- 4 1))))
=> (if #f 1 (* 4 (factorial (- 4 1))))
=> (* 4 (factorial (- 4 1)))
=> (* 4 ((lambda (n) ...) 3))
=> (* 4 (if (= 3 1) 1 (* 3 (factorial (- 3 1)))))
```

```

=> (* 4 (if #f 1 (* 3 (factorial (- 3 1)))))
=> (* 4 (* 3 (factorial (- 3 1))))
...
=> (* 4 (* 3 (* 2 (factorial 1))))
=> (* 4 (* 3 (* 2 (if (= 1 1) 1 (* 1 (factorial (- 1 1)))))))
=> (* 4 (* 3 (* 2 1)))
=> (* 4 (* 3 2))
=> (* 4 6)
=> 24

```

`Factorial` ist eine Prozedur, die *sich selbst aufruft*. Die Technik, daß die Anwendung einer Prozedur schließlich zu einer weiteren Anwendung der Prozedur führt, heißt *Rekursion*. Rekursion gehört zu den wichtigsten Programmier-techniken, weil sie erlaubt, Wiederholung auszudrücken.

Zunächst erscheint dieser Selbstbezug paradox: Wenn eine Prozedur sich selbst anwendet, müßte auf den ersten Blick ein Knoten entstehen, der nie zum Schluß kommt. Warum funktioniert es trotzdem?

Einige Beobachtungen:

- `Factorial` ruft sich nie mit derselben Zahl  $n$  auf, mit der es selbst aufgerufen wurde, sondern mit  $n - 1$ .
- Die natürlichen Zahlen sind so strukturiert, daß die Kette  $n, n - 1, n - 2 \dots$  irgendwann bei 0 abbrechen muß.
- `Factorial` ruft sich bei  $n = 0$  *nicht* selbst auf.

Aus diesen Gründen kommt der Berechnungsprozeß, der von `(factorial  $n$ )` erzeugt wird, immer zum Schluß. Aus theoretischer Sicht funktioniert Rekursion deshalb, weil `(factorial  $n$ )`, nicht wirklich eine voll ausgewachsene Version von `(factorial  $n$ )` benötigt, um ihre Berechnung auszuführen. Eine „etwas kleinere“ Fassung, die nur die Fakultäten bis  $n - 1$  berechnet, reicht völlig aus.

Rekursion ist also die Ausnutzung der Struktur induktiv definierter Mengen in der Programmierung: die induktive Definition der natürlichen Zahlen postuliert folgende Struktur:

- 0 (1) ist eine natürliche Zahl.
- Ist  $n$  eine natürliche Zahl, so ist  $n + 1$  auch eine.
- Nichts sonst ist eine natürliche Zahl.

Damit fallen die natürlichen Zahlen in zwei Klassen:

- 0 (1)
- Zahlen  $n$ , die einen Vorgänger  $m = n - 1$  haben mit  $n = m + 1$ .

Die `factorial`-Prozedur folgt genau dieser Struktur.

An `factorial` läßt sich ein generelles Prinzip für die Lösung von Problemen über den natürlichen Zahlen erkennen, welche die induktive Struktur ausnutzen. Es besteht aus zwei Mantras:

**Selbstähnlichkeit** Führe das Problem für  $n$  auf das gleiche Problem für  $n - 1$  zurück.

**Rekursionsbasis** Behandle den Basisfall (in der Regel 0 oder 1) separat und ohne Rekursion.

Bereits die natürlichen Zahlen tauchen sehr häufig in realen Programmen auf, und damit auch rekursive Berechnungen über ihnen. Außerdem gibt es noch ein ganzes Arsenal weiterer Datenstrukturen, die auf induktive Definitionen zurückzuführen sind — auch bei ihrer Verarbeitung sind fast immer rekursive Prozesse am Werk. Es lohnt sich also, den Umgang mit Rekursion ausgiebig zu trainieren.

Da `factorial` der induktiven Definition folgt, ist ihre Korrektheit einleuchtend. Wie ist es mit komplexeren Beispielen? Die folgende Prozedur quadriert natürliche Zahlen:

```
(define square
  (lambda (n)
    (if (= n 0)
        0
        (+ (square (- n 1))
           (- (+ n n) 1))))))
```

Diese Version ist hinreichend obskur, daß ein Beweis notwendig ist, um von ihrer Korrektheit auszugehen. Der Beweis funktioniert (natürlich) mit Induktion und dreht den Rekursionsprozess um:

Induktionsanfang:

`(square 0)`  $\equiv \dots \equiv 0 = 0^2 \quad \checkmark$

$n \mapsto n + 1$ :

```
(square (n + 1))
 $\implies$  (if (= (n + 1) 0) ...)
 $\implies$  (+ (square (- (n + 1) 1)) (- (+ (n + 1) (n + 1)) 1))
 $\implies$  (+ (square n) (- (2n + 2) 1))
 $\implies$  (+ n2 (2n + 1))
 $\implies$  n2 + 2n + 1 = (n + 1)2
```

Wichtig beim Lesen ist dabei, daß die nicht in äquidistanter Schrift angegebenen Ausdrücke wie  $(n + 1)$  für Zahlen stehen, nicht für Scheme-Ausdrücke.

Der Induktionsbeweis zeigt, daß es nicht nötig ist, den gesamten Berechnungsprozeß einer rekursiven Prozedur Schritt für Schritt nachzuvollziehen. In der Tat ist es für das effektive Training am Umgang mit Rekursion selten hilfreich, rekursiven Prozessen stur hinterherzulaufen: Das menschliche Hirn ist für diese Aufgabe einfach nicht gebaut. Da es oft trotzdem versucht ist, genau dies zu tun, und sich dabei häufig im Morast der rekursiven Aufrufe verirrt, empfiehlt sich das Studium eines weiteren Mantras:

**Just say no** Versuche nicht, rekursiv über einen rekursiven Prozeß nachzudenken.

## 2.2 Iteration und Invarianten

Das Fakultätsproblem ist ein schönes Beispiel für Rekursion, weil es so einfach ist. Zur Erinnerung — die Scheme-Prozedur zu seiner Lösung sah folgendermaßen aus:

```
(define factorial
  (lambda (n)
    (if (= n 1)
        1
        (* n (factorial (- n 1))))))
```

Die Strategie von `factorial` ist, für die Berechnung der Fakultät einer Zahl  $n$  erst ein Teilproblem zu lösen, das fast so groß ist, wie das ursprüngliche: die Fakultät von  $n - 1$ . Die eigentlichen Multiplikationen, die zur Berechnung des Ergebnisses führen, werden dabei bis ganz zum Schluß aufgestaut.

Das ganze ließe sich aber genauso gut von der anderen Seite aufziehen: Alle Multiplikationen werden sofort ausgerechnet und dabei wird ein Zwischenergebnis mitgeführt. Hier ist eine Beispielrechnung für die Fakultät von 4:

$$\begin{aligned} 1 \cdot 4 &= 4 \\ 4 \cdot 3 &= 12 \\ 12 \cdot 2 &= 24 \end{aligned}$$

Die Buchhaltung über ein Zwischenergebnis ist dabei entscheidend. Das ganze läßt sich auch als Programm schreiben:

```
(define factorial
  (lambda (n)
    (factorial-1 n 1)))

(define factorial-1
  (lambda (n result)
    (if (= n 1)
        result
        (factorial-1 (- n 1) (* n result)))))
```

Diese neue Prozedur erzeugt einen anderen Berechnungsprozeß (etwas abgekürzt geschrieben):

```
(factorial 4)
=> (factorial-1 4 1)
=> (if (= 4 1) 1 (factorial-1 (- 4 1) (* 4 1)))
=> (factorial-1 3 4)
=> (if (= 3 1) 4 (factorial-1 (- 3 1) (* 3 4)))
=> (factorial-1 2 12)
=> (if (= 2 1) 12 (factorial-1 (- 2 1) (* 2 12)))
=> (factorial-1 1 24)
=> (if (= 1 1) 24 (factorial-1 (- 1 1) (* 1 24)))
=> 24
```

Während die Prozedur `factorial-1` genau wie das ursprüngliche `factorial` rekursiv ist, erzeugt sie eine besondere Sorte Berechnungsprozeß, die keine Multiplikationen mehr bis zur letzten Sekunde aufstaut, sondern während der rekursiven Aufrufe alle notwendigen Berechnungen schon durchführt. Fachbegrifflich gesehen erzeugt der rekursive Aufruf von `factorial` keinen neuen *Kontext* und ist somit *endrekursiv*.

Diese Sorte Berechnungsprozeß heißt nicht mehr rekursiv, sondern *iterativ*. (In anderen Programmiersprachen müssen aus technischen Gründen spezielle Konstrukte, sogenannte *Schleifen*, verwendet werden, um iterative Prozesse zu erzwingen.)

Eine effektive Möglichkeit, um über die Korrektheit von rekursiven Prozeduren nachzudenken, die iterative Prozesse erzeugen, ist über sogenannte *Invarianten*: Bei allen rekursiven Aufrufen von `factorial-1` bleibt `n!.result` konstant. Daraus folgt direkt die Korrektheit von `factorial`.

Beweis:

Es steht  $n$  für `n` und  $r$  für `result`. Dann ist zu beweisen:  $(n - 1)! \cdot (n \cdot r) = n! \cdot r$ .

$$\begin{aligned} (n - 1)! \cdot (n \cdot r) &= ((n - 1)! \cdot n) \cdot r \\ &= (n \cdot (n - 1)!) \cdot r \\ &= n! \cdot r \end{aligned}$$

Invarianten sind eine sehr nützliche Methode, um Beweise über iterative Prozesse zu führen.

Das gleiche Spiel läßt sich auch am Beispiel von `square` demonstrieren. Zunächst eine iterative Variante:

```
(define square
  (lambda (n)
    (square-1 n 0)))

(define square-1
  (lambda (n result)
    (if (= n 0)
        result
        (square-1 (- n 1)
                  (+ result
                    (- (+ n n) 1)))))))
```

Die Schleifeninvariante von `square-1` ist dabei  $n^2 + r$  wenn  $n$  der Wert von `n` sowie  $r$  der Wert von `result` ist. Der Beweis ist eine einfache Fingerübung.