

Kapitel 2

Rekursion, Induktion, Iteration

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:

$$1! = 1$$

Für $n > 1$:

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

Das entsprechende Programm ist nun einfach hinzuschreiben:

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

Wie funktioniert es?

```
(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.

Da `factorial` der induktiven Definition direkt folgt, ist deren 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) ≡ ... ≡ 0 = 02 ✓`

$n \mapsto n + 1$:

```

(square n + 1)
=> (if (= (n + 1) 0) ...)
=> (+ (square (- (n + 1) 1)) (- (+ (n + 1) (n + 1)) 1))
=> (+ (square n) (- (2n + 2) 1))
=> (+ n2 (2n + 1))
=> 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.)

2.1 Iteration

Das Fakultätsproblem ist ein schönes Beispiel für Rekursion, weil es so einfach ist. Um die Fakultät als rekursive Prozedur zu formulieren, haben wir eine andere Definition als die landläufige benutzt. Bei genauem Hinsehen tut deshalb die Prozedur `factorial` von oben nicht wirklich das, was die ursprüngliche Definition vorsah: Es multipliziert „von hinten“, was nur deshalb nicht ins Gewicht fällt, weil die Multiplikation assoziativ ist.

Ein weiteres Manko kommt hinzu: Die Auswertung von Hand von `(factorial 4)` hat zunächst eine immer größer werdende Kette von Multiplikationen erzeugt, die erst zum Schluß abgerollt werden können. Bei größeren Zahlen entsteht dabei viel Schreibarbeit. Bei vielen rekursiven Prozeduren ist diese Schreibarbeit notwendig, aber bei der Fakultät ist offensichtlich, daß es auch ohne geht:

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

Entscheidend ist dabei die Buchhaltung über ein Zwischenergebnis. 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*, und die iterativen Berechnungsprozesse, die mit Zahlen rechnen, entsprechen gerade den primitiv-rekursiven arithmetischen Funktionen. (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.