

## Kapitel 4

# Programmieren mit Prozeduren

Das Hanoi-Puzzle bietet sich als Basis für die Demonstration weiterer Programmier-techniken an. Die bisherige Lösung ist noch in einigen Aspekten verbesserungsfähig:

```
(define hanoi
  (lambda (n source-peg dest-peg)
    (if (= n 0)
        '()
        (append
         (hanoi (- n 1)
                source-peg
                (other-peg source-peg dest-peg))
         (cons (cons source-peg dest-peg)
               (hanoi (- n 1)
                      (other-peg source-peg dest-peg)
                      dest-peg))))))
```

### 4.1 Lokale Variablen

Es fällt auf, daß `(other-peg source-peg dest-peg)` zweimal in der Lösung vorkommt. Das heißt, die Berechnung, die `other-peg` durchführt (so vernachlässigbar klein sie auch sein mag), passiert zweimal. Wie läßt sich dies verhindern?

Die offensichtliche Lösung ist, den Wert von `(other-peg source-peg dest-peg)` an eine Variable zu binden, und dann den Namen statt der Prozeduranwendung zu verwenden:

```
(define hanoi
  (lambda (n source-peg dest-peg)
    (if (= n 0)
        '()
        ((lambda (temp-peg)
           (append
            (hanoi (- n 1)
                   source-peg
                   temp-peg)
            (cons (cons source-peg dest-peg)
                  (hanoi (- n 1)
                         temp-peg
                         dest-peg))))
           (other-peg source-peg dest-peg))))))
```

Die neue Version ist leider nicht besonders übersichtlich: Die Bindung für `temp-peg` ist weit entfernt von dem Ausdruck, dessen Wert an `temp-peg` gebunden wird. Aus diesem Grund gibt es in Scheme eine weitere Sorte Ausdruck, den `let`-Ausdruck:

```
(define hanoi
  (lambda (n source-peg dest-peg)
    (if (= n 0)
        '()
        (let ((temp-peg (other-peg source-peg dest-peg)))
          (append
            (hanoi (- n 1)
                  source-peg
                  temp-peg)
            (cons (cons source-peg dest-peg)
                  (hanoi (- n 1)
                        temp-peg
                        dest-peg))))))))
```

`Let` hat folgende allgemeine Form:

```
(expression) ::= (let expression)
(let expression) ::= (let ((binding spec)* (body))
(binding spec) ::= ((variable) (expression))
```

Dabei ist `let` strenggenommen überflüssig in der Sprache, da es durch die entsprechende Anwendung einer Abstraktion ersetzt werden kann:

```
(let ((v1 e1) ... (vn en)) b)
⇒
(lambda (v1 ... vn) b) e1 ... en)
```

`Let` ist damit eine sogenannte *abgeleitete Form*, auch genannt *syntaktischer Zucker*.

## 4.2 Mehr Hanoi

Eine weitere Ineffizienz fällt an `hanoi` auf: Der erste rekursive Aufruf von `hanoi` unterscheidet sich vom zweiten lediglich durch die Nummern von Quell- und Zielpfahl. Hier der erste:

```
(hanoi (- n 1)
       source-peg
       temp-peg)
```

... und hier der zweite:

```
(hanoi (- n 1)
       temp-peg
       dest-peg)
```

Nun wird aber beide Male der Turm auf dieselbe Art und Weise bewegt. Die zweite Zugfolge läßt sich also aus der ersten durch bloßes Umnúmerieren der Pfähle gewinnen. Wie ließe sich das umsetzen? Die Anwendung auf Wunschdenken könnte etwa zu folgender neuer `hanoi`-Prozedur führen:

```
(define hanoi
  (lambda (n source-peg dest-peg)
    (if (= n 0)
        '()
        (let ((temp-peg (other-peg source-peg dest-peg)))
          (let ((moves-1 (hanoi (- n 1) source-peg temp-peg)))
            (let ((moves-2 (renumber-moves moves-1
                                   source-peg temp-peg
                                   temp-peg dest-peg
                                   dest-peg source-peg)))
              (append moves-1
                      (cons (cons source-peg dest-peg)
                            moves-2))))))))))
```

Es fällt auch hier eine Unschönheit auf: die verschachtelten `let`-Ausdrücke. Es ist leider nicht möglich, ein einzelnes `let` mit mehreren Bindungen zu benutzen, da sich die zweite und dritte Bindung jeweils auf die vorhergehende beziehen. Zum Glück gibt es syntaktischen Zucker für geschachtelte `lets`, genannt `let*`:

```
(define hanoi
  (lambda (n source-peg dest-peg)
    (if (= n 0)
        '()
        (let* ((temp-peg (other-peg source-peg dest-peg))
              (moves-1 (hanoi (- n 1) source-peg temp-peg))
              (moves-2 (renumber-moves moves-1
                                   source-peg temp-peg
                                   temp-peg dest-peg
                                   dest-peg source-peg)))
          (append moves-1
                  (cons (cons source-peg dest-peg)
                        moves-2))))))
```

Wunschdenken diktiert jetzt die Realisierung von `renumber-moves`. Eine Sache wird schnell klar: das Mitschleppen von allen sechs Parameters für die drei Umnumerierungen wird schnell umständlich:

```
(define renumber-move
  (lambda (move
          source-peg-1 dest-peg-1
          source-peg-2 dest-peg-2
          source-peg-3 dest-peg-3)
    (cons (renumber-peg (car move)
                       source-peg-1 dest-peg-1
                       source-peg-2 dest-peg-2
                       source-peg-3 dest-peg-3)
          (renumber-peg (cdr move)
                       source-peg-1 dest-peg-1
                       source-peg-2 dest-peg-2
                       source-peg-3 dest-peg-3))))

(define renumber-peg
  (lambda (peg
          source-peg-1 dest-peg-1
```



```

                                (cons temp-peg dest-peg)
                                (cons dest-peg source-peg))))))
      (append moves-1
                (cons (cons source-peg dest-peg)
                      moves-2))))))

(define renumber-move
  (lambda (move renumber-alist)
    (cons (renumber-peg (car move) renumber-alist)
          (renumber-peg (cdr move) renumber-alist))))

(define renumber-peg
  (lambda (peg renumber-alist)
    (let ((stuff (associate peg renumber-alist)))
      (if stuff
          (cdr stuff)
          #f))))

(define renumber-moves
  (lambda (moves renumber-alist)
    (if (null? moves)
        '()
        (cons (renumber-move (car moves) renumber-alist)
              (renumber-moves (cdr moves) renumber-alist)))))

```

Aus der letzten Version von `renumber-move` läßt sich ein häufig vorkommendes Rekursionsmuster erkennen: Auf alle Elemente einer Liste (alle Züge einer Zugfolge) wird dieselbe Operation angewendet. Dieses Rekursionsmuster läßt sich in einer Prozedur festhalten:

```

(define map
  (lambda (f l)
    (if (null? l)
        '()
        (cons (f (car l))
              (map f (cdr l))))))

```

Map ist (mit leicht erweiterter Funktionalität) unter ebendiesem Namen schon eingebaut. Mit der Hilfe von `map` läßt sich `renumber-moves` noch etwas knapper schreiben:

```

(define renumber-moves
  (lambda (moves renumber-alist)
    (map (lambda (move)
          (renumber-move move renumber-alist))
         moves)))

```