

# Kapitel 3

## Paare und Listen

Die bisherigen Scheme-Programme beschäftigen sich allesamt mit Berechnungen auf recht einfachen Datenwerten: Zahlen und Booleans. Für die anspruchsvolle Programmierung ist allerdings die Verwendung von zusammengesetzten Datenwerten angezeigt, sogenannten *Datenstrukturen*. Die wichtigste Datenstruktur in Scheme (neben der Prozedur) ist das *Paar*. Mit Hilfe von Paaren lassen sich dann *Listen* erzeugen. Beide zusammen sind das Thema dieses Kapitels.

Listen sind mit großem Abstand die nützlichsten Datenstrukturen. Dadurch, daß sie in Scheme fest eingebaut sind, ist es möglich, sehr viele nützliche Prozeduren auf ihnen zu definieren, die in sehr vielen Programmieraufgaben Anwendung finden. Dieses Kapitel beschreibt zwei solche Aufgaben.

### 3.1 Die Türme von Hanoi

Die Türme von Hanoi sind ein klassisches Puzzle, meist als Spielzeug angesehen. Das Puzzle besteht aus einer Platte mit drei Pfählen. Auf einem der Pfähle sind runde Scheiben mit Loch in Pyramidenform aufgetürmt: stets kleinere auf größeren Scheiben. Die Aufgabe des Puzzles ist es, den Turm auf einen der anderen Pfähle zu bewegen, allerdings unter folgenden Einschränkungen:

- Es darf nur eine Scheibe auf einmal bewegt werden.
- Es darf nie eine größere auf eine kleinere Scheibe gelegt werden.

Eine mögliche Strategie zur Lösung des Puzzles ist die folgende:

- Falls der Turm die Höhe  $n$  hat, bewege den Turm der Höhe  $n - 1$  zunächst auf den dritten Pfahl. (Wie das zu machen ist, wird auf morgen verschoben.)
- Bewege die untere, größte Scheibe auf den Zielpfahl.
- Bewege den Turm der Höhe  $n - 1$  auf den Zielpfahl. (Wie das zu machen ist, wird auf morgen verschoben.)

Einen Tag später hat sich das Problem darauf reduziert, das Hanoi-Puzzle für  $n - 1$  Scheiben zu lösen. Das läßt sich wiederum auf das Puzzle für  $n - 2$  Scheiben lösen etc., bis schließlich das Problem nur noch für 0 Scheiben zu lösen ist und damit trivial geworden ist.

## 3.2 Paare

Die Strategie für die Lösung der Hanoi-Türme ist einfach. Um sie in ein Programm zu verwandeln, fehlt uns eine Möglichkeit, die Lösung nach außen zu kommunizieren: Zahlen und Booleans reichen nun einmal nicht aus, um Zugfolgen im Hanoi-Puzzle auszudrücken. (Zumindest wäre das Resultat für normale Menschen dann schwer zu interpretieren.) Ein Programm, das die Türme von Hanoi löst, muß also mit Werten hantieren, die eine Folge von Zügen im Puzzle repräsentieren können.

Das einfachere Teilproblem ist die Repräsentation einzelner Züge. Ein Wert, der für einen Zug steht, sollte in der Lage sein, zwei Zahlen zu enthalten: beispielsweise die Nummer des Pfahls, von dem eine Scheibe genommen wird und die Nummer des Pfahls, auf den sie gesetzt wird — Ursprungs- und Zielpfahl sozusagen.

Zum Glück sind in Scheme spezielle Werte eingebaut, die zwei Werte enthalten können: die sogenannten *Paare*:

```
> (cons 23 42)
(23 . 42)
```

`cons` ist eine eingebaute Prozedur, deren Name für „construct“ steht. Es macht aus seinen beiden Parametern ein Paar. Die externe Repräsentation eines solchen Paares besteht aus:

- einer Klammer auf,
- der Repräsentation der ersten Komponente des Paares,
- einem Punkt,
- der Repräsentation der zweiten Komponente
- und einer Klammer zu.

Paare werden erst dadurch wirklich nützlich, daß sich die beiden Komponenten auch wieder extrahieren lassen:

```
> (define p (cons 23 42))
> (car p)
23
> (cdr p)
42
```

`car` und `cdr` sind ebenfalls eingebaute Prozeduren und extrahieren jeweils die erste und die zweite Komponente. (`Car` und `cdr` (gesprochen „kudder“) waren die Namen von Anweisungen auf einer Maschine, auf der ein Vorläufer von Scheme lief.)

`cons`, `car` und `cdr` erfüllen eine Gleichung. Sei  $p$  ein Paar. Dann gilt folgendes:

$$(\text{cons } (\text{car } p) (\text{cdr } p)) \equiv p$$

Für die Lösung des Hanoi-Puzzles wird hier ein Paar  $(s . d)$  für einen Zug vom Ursprungspfahl  $s$  zum Zielpfahl  $d$  stehen.

## 3.3 Listen

Einzelne Züge sind ein Schritt in die richtige Richtung, aber noch nicht ganz ausreichend: Das Hanoi-Puzzle benötigt ganze Zugfolgen, also Datenstrukturen mit beliebig (endlich) vielen Komponenten. Mit Paaren lassen sich Datenstrukturen mit mehr als zwei Komponenten bauen:

```
> (cons (cons 1 2) 3)
((1 . 2) . 3)
```

Dies ist eine umständliche Methode, Folgen zu repräsentieren. Es ist abzusehen, daß es häufig notwendig sein wird, das erste Element einer Folge zu extrahieren. In diesem Fall erfordert dies zwei Anwendungen von `car`:

```
> (car (car (cons (cons 1 2) 3)))
1
```

Es ginge auch andersherum:

```
(cons 1 (cons 2 3))
(1 2 . 3)
```

*Hoppla!* Nach den bisherigen Erkenntnissen sollte die Ausgabe anders aussehen:

```
(1 . (2 . 3))
```

Das Scheme-System hat ungefragt einen Punkt und ein Klammernpaar unterschlagen! Dies ist eine Konvention in Scheme, um „Klammerwüsten“ bei der Repräsentation von Paaren zu vermeiden: Wenn ein Punkt gefolgt wäre von einer offenen Klammer, so kann das Scheme-System bei der Ausgabe den Punkt und das Klammernpaar weglassen. Diese Regel heißt *Punkt-Klammer-Zap-Regel*.

Jedenfalls lassen sich jetzt die ersten beiden Komponenten der Folge einfach extrahieren:

```
> (car (cons 1 (cons 2 3)))
1
> (car (cdr (cons 1 (cons 2 3))))
2
```

Für Element  $n$  der Folge wird also `cdr`  $n - 1$ -mal angewendet, dann einmal `car`. Dieses Prinzip funktioniert leider nicht für das dritte Element:

```
> (cdr (cdr (cons 1 (cons 2 3))))
3
```

Es wäre also sinnvoll, in den `cdr` des letzten Paares einen anderen Wert zu stopfen, z.B.

```
> (cons 1 (cons 2 (cons 3 #f)))
(1 2 3 . #f)
```

Jetzt funktioniert der Zugriff auf regelmäßige Art und Weise:

```
> (define s (cons 1 (cons 2 (cons 3 #f))))
> (car s)
1
> (car (cdr s))
2
> (car (cdr (cdr s)))
3
```

Es gibt jetzt nur noch ein kleines ästhetisches Problem: das „. #f“ am Ende ist häßlich. Zum Glück gibt es in Scheme einen Wert, der als externe Repräsentation `()` hat. Das zugehörige Literal ist `'()`:

```
> (cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

Die Punkt-Klammer-Zap-Regel sorgt nun dafür, daß die Ausgabe gut aussieht. In der Tat heißt ein solcher Wert *Liste*, und `()` heißt die *leere Liste*. Dementsprechend bilden die Listen eine induktiv definierte Menge:

- Die leere Liste ist eine Liste.
- Falls  $l$  eine Liste und  $v$  ein beliebiger Wert ist, so ist das Paar mit  $v$  als `car` und  $l$  als `cdr` ebenfalls eine Liste.
- Nichts sonst ist eine Liste.

Listen lassen sich also in zwei Klassen aufteilen:

- die leeren Listen (wovon es nur eine gibt) und die
- nicht-leeren Listen, die in ihren `car` (das erste Element) und den `cdr` (die restliche Liste ohne das erste Element) zerfallen.

```
> (define l (cons 1 (cons 2 (cons 3 '()))))
> (car l)
1
> (cdr l)
(2 3)
> (cdr (cdr l))
(3)
> (cdr (cdr (cdr l)))
()
```

Für realistische Programme ist notwendig, zwischen den beiden Sorten Liste zu unterscheiden. Es gibt dazu zwei eingebaute Prozeduren:

- `(null? v)` erkennt die leere Liste: Es liefert `#t`, falls  $v$  die leere Liste ist, `#f` sonst.
- `(pair? v)` erkennt Paare: Es liefert `#t`, falls  $v$  ein Paar ist, `#f` sonst. (Unter den Listen erkennt also `pair?` gerade die nicht-leeren.)

```
> (null? '())
#t

> (null? (cons 1 2))
#f

> (pair? '())
#f

> (pair? (cons 1 2))
#t
```

Hier ist eine kleine Prozedur auf Listen zur Übung:

```
(define list-length
  (lambda (list)
    (if (null? list)
        0
        (+ 1 (list-length (cdr list))))))
```

Fast jede Prozedur, die auf Listen operiert, hat die gleiche Form: Ein Conditional, das zwischen leeren und nichtleeren Listen unterscheidet. Im Basisfall erfolgt kein rekursiver Aufruf. Im nichtleeren Fall erfolgt ein rekursiver Aufruf auf dem `cdr` der Liste.

Hier ist noch eine weitere praktische Operation auf Listen: `cons` erlaubt es, an eine Liste vorn noch etwas heranzuhängen:

```
> (define l (cons 1 '()))
> (cons 23 l)
(23 1)
```

Wie lassen sich zwei Listen aneinanderhängen?

```
(define concatenate
  (lambda (list-1 list-2)
    (if (null? list-1)
        list-2
        (cons (car list-1)
              (concatenate (cdr list-1) list-2)))))
```

```
> (concatenate (cons 1 (cons 2 (cons 3 '())))
              (cons 4 (cons 5 '())))
(1 2 3 4 5)
```

Tatsächlich sind `list-length` und `concatenate` bereits in Scheme eingebaute Prozeduren mit den Namen `length` und `append`.

Für das Experimentieren mit Listen ist die Prozedur `list` extrem praktisch: sie akzeptiert beliebig viele Argumente und macht aus ihnen eine Liste:

```
> (list 1 2 3 4 5)
(1 2 3 4 5)
> (list (cons 1 2) (cons 3 4))
((1 . 2) (3 . 4))
```

Eine weitere praktische Prozedur auf Listen ist `map`, die eine einzige Prozedur auf alle Elemente einer Liste anwendet und eine Liste der Rückgabewerte liefert:

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

Zum Beispiel lassen sich mit `map` einfach alle Elemente einer Liste skalieren:

```
(map (lambda (x) (* x 10)) (list 1 2 3 4 5 6))
```

## 3.4 Hanoi lösen

Zurück zu den Türmen von Hanoi. Zur Erinnerung noch einmal die anvisierte Strategie für die Lösung des Problems. Es geht darum, eine Turm der Höhe  $n$  von Pfahl  $s$  nach Pfahl  $d$  zu transferieren. Dabei sei  $t$  der dritte Pfahl, also der Pfahl, der weder  $s$  noch  $d$  ist.

- Falls der Turm die Höhe 0 hat, habe fertig.

- Bewege den Turm aus den oberen  $n - 1$  Scheiben  $s$  nach  $t$ .
- Bewege die untere Scheibe von  $s$  nach  $d$ .
- Bewege den Turm auf  $t$  nach  $d$ .

Das Programm wird eine Folge von Zügen durch eine Liste von Paaren repräsentieren:

```
((1 . 2) (2 . 3) (3 . 1))
```

heißt: bewege die obere Scheibe von Pfahl 1 auf Pfahl 2, dann bewege die obere Scheibe von Pfahl 2 auf Pfahl 3, dann bewege von 3 auf 1.

Die Prozedur `hanoi` hat drei Parameter:

- `n` ist die Höhe des Turms.
- `source-peg` ist die Nummer des Pfahls, auf dem der Turm bisher steht.
- `dest-peg` ist die Nummer des Pfahls, auf den der Turm übertragen werden soll.

Der Anfang ist einfach:

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

Für nichtleere Türme müssen drei Zugfolgen aneinandergehängt werden. Die erste Zugfolge muß die oberen  $n - 1$  Türme von `source-peg` zu dem Pfahl bewegen, der nicht `source-peg` und auch nicht `dest-peg` ist:

```
(append
 (hanoi (- n 1)
        source-peg
        ?))
```

Wie läßt sich die Nummer des dritten Pfahls bestimmen? In solchen Situationen, in denen die Lösung eines Teilproblems nicht sofort offensichtlich ist oder von der Arbeit ablenken würde, kommt eine wichtige Programmieretechnik ins Spiel: *Wunschdenken* (in vornehmeren Kreisen auch *Top-Down-Design* genannt). Der Programmierer tut einfach so, als ob das Problem schon durch eine Prozedur gelöst wäre:

```
(append
 (hanoi (- n 1)
        source-peg
        (other-peg source-peg dest-peg)))
```

Damit ist die erste Teilzugfolge fertig. Die zweite ist ein einzelner Zug von `source-peg` nach `dest-peg`:

```
(cons source-peg dest-peg)
```

Die letzte Teilzugfolge muß den  $n - 1$  Scheiben hohen Turm nach `dest-peg` übertragen:

```
(hanoi (- n 1)
        (other-peg source-peg dest-peg)
        dest-peg)
```

Alles zusammen sieht dann etwa so aus:

```
(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)))
```

Damit ist der schwierige Teil der Hanoi-Lösung fertig. Leider fehlt noch etwas, da `other-peg` bisher noch reines Wunschdenken ist. Die Aufgabe erscheint trivial, aber die naive Lösung ist erstaunlich häßlich:

```
(define other-peg
  (lambda (peg-1 peg-2)
    (if (= peg-1 1)
      (if (= peg-2 2)
        3
        2)
      (if (= peg-1 2)
        (if (= peg-2 1)
          3
          1)
        (if (= peg-2 1)
          2
          1))))))
```

`other-peg` läßt sich allerdings mit etwas elementarerer Algebra vereinfachen. Die Nummern der Pfähle sind 1, 2 und 3:

$$1 + 2 + 3 = 6$$

Zwei Pfähle lassen sich aus der Summe durch Subtraktion entfernen. Der restliche in der Summe verbleibende Pfahl könnte  $t$  genannt werden:

$$t = 1 + 2 + 3 - s - d = 6 - s - d$$

Eine alternative Lösung für `other-peg` folgt direkt:

```
(define other-peg
  (lambda (peg-1 peg-2)
    (- (- 6 peg-1) peg-2)))
```

### 3.5 Exkurs: Lokale Variablen und let

Für die weitere Programmierung wird es sich ein weiteres bisher unbekanntes Programmierkonstrukt als praktisch erweisen. Es heißt `let`. `Let` ist für das Anlegen *lokaler Variablen* zuständig. Lokale Variablen sind besonders dann praktisch, wenn der Wert eines Ausdrucks mehrfach verwendet. Als Beispiel mag folgende Formel dienen:

$$a(a + b)^2 + a(a + b) + a(a - b)^2 + b(a - b)$$

Das ganze ließe sich auch so schreiben:

$$\begin{aligned}x &= a + b \\y &= a - b \\ax^2 + ax + ay^2 + by\end{aligned}$$

In Scheme sieht ein entsprechender Ausdruck folgendermaßen aus:

```
(let ((x (+ a b))
      (y (- a b)))
  (+ (* a (square x))
     (* b x)
     (* a (square y))
     (* b y)))
```

Das dort benutzte `let` führt dabei zwei lokale Variablen `x` und `y` ein, die nur innerhalb der `let`-Form sichtbar sind. Dort haben sie die Werte von `(+ a b)` bzw. `(- a b)`.

Das `let`-Konstrukt wird durch folgenden Zusatz zur Scheme-Grammatik beschrieben:

```
(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*.

Bei der Benutzung von `let` ist zu beachten, daß die Ausdrücke, deren Werte an die lokalen Variablen gebunden werden, allesamt *außerhalb* des Einzugsbereichs des `let` ausgewertet werden. Der Wert der `let`-Form in folgendem Programm ist somit 46, nicht 65:

```
(define x 23)

(let ((x 42)
      (y (- x 19)))
  (+ x y))
```

### 3.6 Listen und die reale Welt

Hier ist ein typisches Problem (von leider schwindender Bedeutung), zu dessen Lösung die Kombination von Listen und Rekursion hervorragend taugt:

Die Aufgabe ist, den Inhalt einer Audio-CD auf ein Audio-Tape zu überspielen, das zwei gleichlange Seiten hat. Das Problem ist, daß eine CD in der Regel nicht auf eine einzelne Seite paßt, die Titel der CD also auf beide Seiten verteilt werden müssen. Es ist aber wünschenswert, daß die erste Seite so vollgepackt wie möglich ist, damit beim Umdrehen nicht gespult werden muß.

Für die Lösung muß erst einmal geregelt werden, wie Ein- und Ausgabe des Programms aussehen sollen. Die elementare Größe im Problem ist ein *CD-Titel*, der aus einer Track-Nummer und der Länge des Tracks besteht. Eine CD besteht dann aus einer Liste von CD-Titeln. Zum Beispiel wird der Batman-Soundtrack von Prince durch folgende Liste beschrieben:

```
(define batman
  (list (cons 1 248)
        (cons 2 248)
        (cons 3 303)
        (cons 4 192)
        (cons 5 292)
        (cons 6 264)
        (cons 7 256)
        (cons 8 375)
        (cons 9 374)))
```

Die Zeitangaben sind dabei in Sekunden.

Die Lösung des Problems soll eine Prozedur `side-a-titles` sein, welche eine Liste der CD-Titel zurückliefert, die auf Seite A des Tapes landen sollen. Neben der Liste der CD-Titel braucht `side-a-titles` auch noch die Länge einer Tape-Seite. Im Fall des Batman-Soundtracks, der auf ein 2×30-Minuten-Tape überspielt werden soll, passiert also folgendes:

```
> (side-a-titles batman 1800)
((3 . 303) (4 . 192) (5 . 292) (6 . 264) (8 . 375) (9 . 374))
```

Da es sich um ein klassisches rekursives Problem handelt, ist die Grundstruktur der Prozedur bereits klar:

```
(define side-a-titles
  (lambda (titles length)
    (if (null? titles)
        ?
        ... (side-a-titles (cdr titles) length) ... )))
```

Das erste Fragezeichen ist schnell geklärt: Falls keine Titel da sind, können sie auch nicht auf Seite A des Tapes geschnitten werden. Damit muß dort '()' stehen.

Der Rekursionsschritt ist allerdings nicht ganz so einfach zu klären. Wichtig ist dabei die Befolgung des dritten Mantras der Rekursion:

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

`titles` ist eine Liste, die natürlich in ein erstes Element — also einen ersten Titel — und einen Rest ohne diesen Titel zerfällt. Die entscheidende Frage an dieser Stelle ist also: Ist dieser erste Titel auf Seite A oder nicht? Der einfachste Fall ist, wenn der Titel gar nicht mehr auf die Seite paßt. Hier also eine Teillösung des Problems, welche die einfachen Fälle bereits berücksichtigt:

```
(define side-a-titles
  (lambda (titles length)
    (if (null? titles)
        '()
        (let ((first-title (car titles)))
          (if (> (cdr first-title) length)
              (side-a-titles (cdr titles) length)
              ... (side-a-titles (cdr titles) length) ... )))))
```

Nun ist immer noch die Wahl zu treffen, ob der erste Titel in `titles` auf Seite A geschnitten werden soll. Wenn ja, so verbleibt  $t_t - t_1$  Zeit auf der Seite, wenn  $t_t$  die Länge der Seite und  $t_1$  die Länge des Titels ist. Das Problem reduziert sich dann darauf, daß der Rest der Seite mit den restlichen Titeln vollgepackt werden muß. Ein rekursiver Aufruf erledigt diese Aufgabe; an das Ergebnis muß dann nur noch `first-title` angehängt werden, um eine mögliche Bespielung von Seite A zu konstruieren:

```
(let ((titles-1
      (cons first-title
            (side-a-titles (cdr titles)
                          (- length (cdr first-title))))))
```

Andererseits kann es ebensogut sein, daß die Verwendung des ersten Titels zu einer suboptimalen Bespielung von Seite A führt. Was passiert, wenn Seite A ohne `first-title` auskommt? Es kommen dann folgende Titel auf der Tapeseite zu liegen:

```
(let ((titles-2 (side-a-titles (cdr titles) length)))
```

Jetzt muß `side-a-titles` nur noch entscheiden, ob `titles-1` oder `titles-2` die bessere Lösung, also die Lösung mit der größeren Gesamtlänge ist. Die Anwendung von Wunschdenken delegiert die Aufgabe der Ermittlung der Länge an eine Hilfsprozedur `titles-length`:

```
(if (> (titles-length titles-1)
      (titles-length titles-2))
    titles-1
    titles-2))))))
```

`Titles-length` ist dann geradezu ein Klaps:

```
(define titles-length
  (lambda (titles)
    (if (null? titles)
        0
        (+ (cdr (car titles))
           (titles-length (cdr titles))))))
```

Das Ergebnis ist sehr erfreulich:

```
> (titles-length (side-a-titles batman 1800))
1800
```