

Kapitel 3

Paare und Listen

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, denn 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. (Streng genommen schon, aber das Resultat ist 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 der ersten Maschine, auf einer der Vorläufer von Scheme lief.)

Cons, car und cdr erfüllen eine Gleichung. Sei p ein Paar:

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

Für die Lösung des Hanoi-Puzzles wird ein Paar $(s . d)$ für einen Zug vom Ursprungspfad s zum Zielpfad 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 Klammer auf, so kann das Scheme-System bei der Ausgabe den Punkt und das Klammernpaare 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)
```

(Dieses Beispiel zeigt auch, daß `#f` ein Literal für den Wert gleicher Repräsentation ist.)

Jetzt funktioniert der Zugriff gleichmäßig:

```
> (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 leere Liste 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 nichtleeren Fall erfolgt dann 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 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`.

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)
        '()

```

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)))
```

