

# Kapitel 6

## Binäre Bäume

Eine besonders praktische Datenstruktur ist der *Binärbaum*. Auf ihnen lassen sich eine Reihe weiterer praktischer Datenstrukturen aufbauen. In diesem Kapitel dienen Suchbäume und Bäume für Huffman-Codierung als Beispiele.

### 6.1 letrec

Zuerst allerdings ist es Zeit für ein weiteres Sprachelement: Für Rekursion war es bisher häufig notwendig, eine separate Hilfsprozedur zu schreiben, welche die rekursiven Aufrufe vornimmt. Ein typisches Beispiel ist die endrekursive Fassung von `factorial`:

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

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

Dies ist auf Dauer etwas umständlich, da jedesmal ein neuer Name verbraucht wird. Außerdem müssen ggf. zusätzliche Daten an die Hilfsprozedur übergeben werden, welche bei jedem rekursiven Aufruf unverändert durchgereicht werden.

Die offensichtliche Abhilfe ist, die Hilfsprozedur an eine lokale Variable zu binden. Hierzu steht uns `let` zur Verfügung. Leider erlaubt `let` keine Rekursion, da die Namen auf der linken Seite in den Ausdrücken auf der rechten Seite nicht sichtbar sind. Ebenso ist es bei `let*` (schließlich auch nur syntaktischer Zucker für `let`), bei dem zusätzlich zur `let`-Situation auch noch die darüberliegenden Bindungen sichtbar werden. Darum gibt es noch ein drittes Mitglied der `let`-Familie, mit Syntax identisch zu der der beiden anderen. `Letrec` zeichnet sich dadurch aus, daß alle Variablen auf den linken Seite in den Ausdrücken auf den rechten Seiten sichtbar sind.

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

```

      result
      (factorial-1 (- n 1) (* n result))))))
(factorial-1 n 1)))

```

Letrec ist zwar im Scheme-Standard als syntaktischer Zucker ausgewiesen; allerdings wurden bisher noch nicht alle Sprachelemente eingeführt, die für die Übersetzung notwendig sind.

## 6.2 Binäre Bäume

Binäre Bäume treten in verschiedenen Varianten auf. Die hier betrachteten binären Bäume (meist kurz „Bäume“ genannt) gehören zu einer der folgenden Klassen:

- ein *leerer Baum*,
- ein (innerer) *Knoten*, der sich innen im Baum befindet, und
  - eine *Markierung*,
  - einen *linken Teilbaum* und
  - einen *rechten Teilbaum*

besitzt.

In vielen Realisierungen von Bäumen gibt es keine leeren Bäume aber dafür *Blätter*, Knoten mit leerem linken und rechten Teilbaum. Ein Beispiel dafür sind die Huffman-Bäume im zweiten Teil des Kapitels. In der obigen Definition lassen sich Blätter als Knoten mit zwei leeren Teilbäumen ausdrücken.

Bei der Repräsentation für Bäume werden Typen verwendet und die Abstraktionen aus dem vorigen Kapitel benutzt. Es gibt separate Typen für leere Bäume und Knoten. Die leeren Bäume zuerst:

```

(define empty-tree-type (make-type 'empty-tree))

(define make-empty-tree
  (let ((construct (typed-value-maker empty-tree-type)))
    (lambda ()
      (construct 'dont-care))))

```

Das `dont-care` kommuniziert dabei keine Information, aber aufgrund der Definition von `typed-value-maker` muß irgendetwas dort stehen. Das Prädikat ist als nächstes dran:

```

(define empty-tree? (typed-value-predicate empty-tree-type))

```

Letztlich macht es wenig Sinn, zwischen verschiedenen leeren Bäumen zu unterscheiden. Es reicht einer:

```

(define empty-tree (make-empty-tree))

```

Knoten sind repräsentiert aus einem Paar, dessen `car` die Markierung und dessen `cdr` ein Paar aus linkem und rechtem Teilbaum ist:

```

(define node-type (make-type 'node))

(define make-node
  (let ((construct (typed-value-maker node-type)))
    (lambda (label left-branch right-branch)
      (construct (cons label
                       (cons left-branch
                             right-branch))))))

```

Ein Knoten hat drei Bestandteile, die allesamt in dem getypten Wert eingepackt werden müssen. Bei `make-node` wird ein Paar aus der Markierung und einem Paar aus linkem und rechtem Zweig erzeugt. Dieses Paar wird dann an den Konstruktor von `node-type` — hier `construct` genannt — übergeben. Entsprechend werden neben dem Prädikat entsprechende Selektoren für Markierung, linken und rechten Teilbaum benötigt, die alle auf dem „Basis-Selektor“ des Typs `node-type` aufbauen:

```
(define node? (typed-value-predicate node-type))

(define node-data (typed-value-selector node-type))

(define node-label
  (lambda (node)
    (car (node-data node))))

(define node-left-branch
  (lambda (node)
    (car (cdr (node-data node)))))

(define node-right-branch
  (lambda (node)
    (cdr (cdr (node-data node)))))
```

Basierend auf diesen Prozeduren läßt sich eine gelegentlich nützliche abgeleitete Prozedur definieren:

```
(define tree?
  (lambda (value)
    (or (empty-tree? value)
        (node? value))))
```

## 6.3 Suchbäume

Eine typische Anwendung für Bäume sind *Suchbäume*, eine Repräsentation für Mengen. Suchbäume arbeiten effizienter als Listen, wenn sie Mengen repräsentieren: Eine Liste muß Element für Element abgearbeitet werden, um festzustellen, ob sich ein Element darin befindet. Bei Suchbäumen muß auf den Elementen eine totale Ordnung definiert sein. Suchbäume sind dann Bäume mit folgender Eigenschaft:

- Alle Markierungen des linken Teilbaums eines Knotens haben eine kleinere Markierung als die Markierung des Knotens.
- Alle Markierungen des rechten Teilbaums eines Knotens haben eine größere Markierung als die Markierung des Knotens.

Also läßt sich durch Ansicht der Markierung eines Knotens feststellen (wenn diese nicht sowieso schon die gesuchte ist), in welchem Teilbaum des Knotens eine gesuchte Markierung stecken muß.

Für Suchbäume wird eine ein neuer Typ definiert:

```
(define search-tree-type (make-type 'search-tree))
```

Suchbäume bestehen — neben dem Baum selbst — noch aus der Ordnung und einer Gleichheitsrelation (die natürlich zueinander passen müssen):

```
(define make-search-tree
  (let ((construct (typed-value-maker search-tree-type))
        (lambda (= < tree)
          (construct (cons (cons = <) tree))))))
```

Dabei soll `make-search-tree` nur „intern“ verwendet werden. Dieser Konstruktor ist in gewisser Weise gefährlich, da er es erlaubt, Werte vom Typ mit Namen `search-tree` zu erzeugen, welche die Suchbaumeigenschaft nicht erfüllen. Stattdessen fangen alle Suchbäume beim leeren Suchbaum an:

```
(define make-empty-search-tree
  (lambda (= <)
    (make-search-tree = < empty-tree)))
```

Prädikat und Selektoren gehen streng schematisch:

```
(define search-tree-data (typed-value-selector search-tree-type))
```

```
(define search-tree-<
  (lambda (search-tree)
    (cdr (car (search-tree-data search-tree)))))
```

```
(define search-tree-=
  (lambda (search-tree)
    (car (car (search-tree-data search-tree)))))
```

```
(define search-tree-tree
  (lambda (search-tree)
    (cdr (search-tree-data search-tree))))
```

Zunächst zum Elementtest. Dazu werden die Komponenten des Suchbaums extrahiert; dann wird eine Hilfsprozedur aufgerufen, die nicht jedesmal alles auspacken muß:

```
(define search-tree-member?
  (lambda (element search-tree)
    (let ((= (search-tree-= search-tree))
          (< (search-tree-< search-tree)))
      (letrec
        ((member?
          (lambda (tree)
            (cond
              ((empty-tree? tree)
               #f)
              ((= (node-label tree) element)
               #t)
              ((< element (node-label tree))
               (member? (node-left-branch tree)))
              (else
               (member? (node-right-branch tree)))))))
        (member? (search-tree-tree search-tree))))))
```

Die Prozedur `search-tree-insert` fügt ein neues Element in den Suchbaum ein und erhält dabei die Suchbaumeigenschaft.

```
(define search-tree-insert
  (lambda (element search-tree)
```

```

(let ((= (search-tree= search-tree))
      (< (search-tree-< search-tree)))
  (letrec
    ((insert
      (lambda (tree)
        (cond
          ((empty-tree? tree)
           (make-node element empty-tree empty-tree))
          ((= element (node-label tree))
           tree)
          (<< element (node-label tree))
           (make-node (node-label tree)
                      (insert (node-left-branch tree))
                      (node-right-branch tree)))
          (else
           (make-node (node-label tree)
                      (node-left-branch tree)
                      (insert (node-right-branch tree)))))))
      (make-search-tree
       = <
       (insert (search-tree-tree search-tree))))))

```

## 6.4 Huffman-Bäume

Eine andere der vielen Anwendungen von Binärbäumen liegt in der Codierungstheorie: *Huffman-Bäume* dienen dazu, Daten als Folge von Bits (also von Nullen und Einsen) zu repräsentieren. Für Textdaten gibt es z.B. eine ganze Reihe von Standards, in denen jeweils ein Buchstabe durch eine feste Folge von Bits dargestellt wird. Beispiele hierfür sind ASCII (7 Bits, für den angelsächsischen Sprachraum) und ISO Latin 1 (8 Bits, für den westeuropäischen Sprachraum). Diese Codierungen werden auch *Codierungen mit fester Länge* genannt.

Die Codierungen fester Länge haben den Nachteil, daß Symbole, die so gut wie gar nicht in realen Texten vorkommen, trotzdem den gleichen Platz in der Codierung einnehmen wie solche, die ständig vorkommen. Deswegen ist es gelegentlich vorteilhaft, Codierungen mit *variabler Länge* zu verwenden, in denen häufig verwendete Symbole weniger Bits beanspruchen als selten verwendete Symbole. (Im Morse-Alphabet ist „E“ ein einzelner Punkt.)

Codierungen variabler Länge sind schwieriger zu konstruieren als solche mit fester Länge: Angenommen, „E“ ist codiert als 0, und „S“ ist codiert als 00. Steht nun die Bitfolge 000 für „EEE“, „SE“ oder „ES“? Eine Lösung für das Problem ist, einen speziellen Separator-Code zu verwenden, um die Grenzen zwischen Symbolen zu markieren. (Eine Pause im Morse-Alphabet.) Eine andere Lösung ist es, die Symbole so zu codieren, daß niemals der Code für ein Symbol der Anfang (oder *Präfix*) des Codes eines anderen Zeichens ist. Solche Codierungen heißen *Präfix-Codierungen*. Der Code für „E“ und „S“ oben erfüllt diese Eigenschaft nicht, da 0 ein Präfix von 00 ist.

Effektive Präfix-Codierungen benutzen kurze Codes für häufig vorkommende und lange für selten vorkommende Symbole. Eine Methode für die Konstruktion solcher Codierungen ist die *Huffman-Methode*. Eine Huffman-Codierung kann als ein binärer Baum mit Knoten und Blättern — ein sogenannte *Huffman-Baum* — repräsentiert werden. Die Blätter eines Huffman-Baums mit den Symbolen markiert sind. Jeder Knoten ist mit der Menge der Symbole der Blätter markiert, die unter ihm liegen. Zusätzlich wird jedes Blatt und jeder Knoten mit einem *Gewicht* mar-

kiert: Bei Blättern steht das Gewicht für die relative Häufigkeit des Symbols, bei einem Knoten für die Summe der Gewichte unter ihm. Die Gewichte werden weder für die Codierung noch für die Decodierung benötigt, sind aber bei der Konstruktion eines Huffman-Baums nützlich.

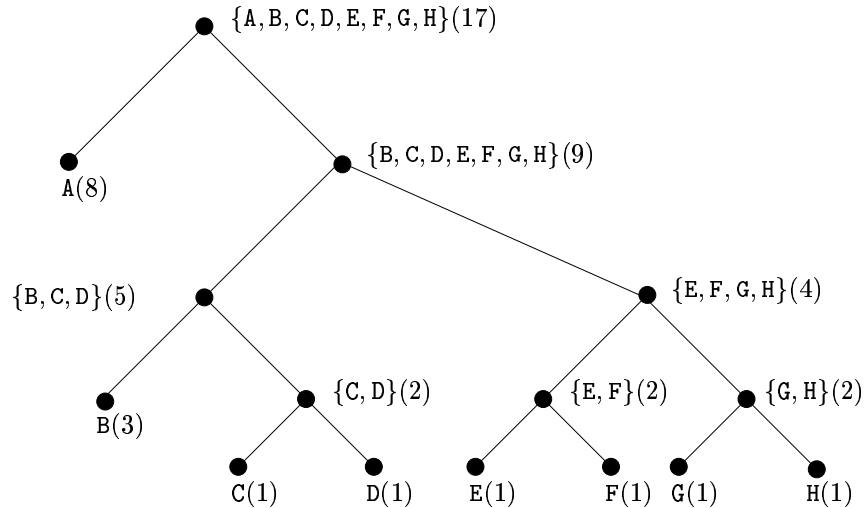


Abbildung 6.1: Ein Huffman-Baum

Abbildung 6.1 zeigt einen Huffman-Baum mit den Buchstaben A–H als Symbole, wobei A die relative Häufigkeit 8, B die relative Häufigkeit 3 und alle anderen Symbole die relative Häufigkeit 1 haben. Die Codierung eines Symbols ergibt sich aus dem Pfad von der Wurzel des Baums bis zu dem Blatt, welches das Symbol als Codierung hat: Bei jeder Abzweigung nach links wird eine 0 an die Codierung gehängt, bei jeder Abzweigung nach rechts eine 1. Für D zum Beispiel ist die Codierung 1011.

Die Decodierung funktioniert entsprechend: Die Bitfolge kennzeichnet einen Pfad innerhalb des Baums, dabei steht eine 0 für eine Abzweigung nach links, eine 1 für eine Abzweigung nach rechts. Ist ein Blatt erreicht, so ist das entsprechende Symbol erkannt und ein neues beginnt.

Bei der Repräsentation von Huffman-Bäumen sind Blätter mit einem Symbol und einem Gewicht markiert:

```

(define huffman-leaf-type (make-type 'huffman-leaf))

(define make-huffman-leaf
  (let ((construct (typed-value-maker huffman-leaf-type)))
    (lambda (symbol weight)
      (construct (cons symbol weight)))))

(define huffman-leaf? (typed-value-predicate huffman-leaf-type))

(define huffman-leaf-data (typed-value-selector huffman-leaf-type))

(define huffman-leaf-symbol
  (lambda (leaf)
    (car (huffman-leaf-data leaf))))

(define huffman-leaf-weight

```

```
(lambda (leaf)
  (cdr (huffman-leaf-data leaf)))
```

Ein Knoten soll mit der Menge der Symbole der Blätter unter ihm markiert werden. Für die Repräsentation der Menge reicht eine einfache Liste. Diese wird bei der Knotenkonstruktion gleich mit gebildet:

```
(define huffman-node-type (make-type 'huffman-node))

(define make-huffman-node
  (let ((construct (typed-value-maker huffman-node-type)))
    (lambda (left-branch right-branch)
      (let ((label (cons (append (huffman-symbols left-branch)
                                (huffman-symbols right-branch))
                        (+ (huffman-weight left-branch)
                          (huffman-weight right-branch)))))
        (construct (cons label (cons left-branch right-branch)))))))

(define huffman-node? (typed-value-predicate huffman-node-type))

(define huffman-node-data (typed-value-selector huffman-node-type))

(define huffman-node-label
  (lambda (huffman-node)
    (car (huffman-node-data huffman-node))))

(define huffman-node-left-branch
  (lambda (huffman-node)
    (car (cdr (huffman-node-data huffman-node)))))

(define huffman-node-right-branch
  (lambda (huffman-node)
    (cdr (cdr (huffman-node-data huffman-node)))))

(define huffman-node-symbols
  (lambda (node)
    (car (huffman-node-label node))))

(define huffman-node-weight
  (lambda (node)
    (cdr (huffman-node-label node))))
```

Die verwendete Hilfsprozedur `huffman-symbols` sorgt dafür, daß die einzelnen Symbole eines Blattes noch in eine Liste eingepackt werden, damit `append` sie als Parameter akzeptiert:

```
(define huffman-symbols
  (lambda (tree)
    (if (huffman-leaf? tree)
        (list (huffman-leaf-symbol tree))
        (huffman-node-symbols tree))))
```

Die Prozedur `huffman-weight` verallgemeinert über Blätter und Knoten, was das Gewicht betrifft:

```
(define huffman-weight
```

```
(lambda (tree)
  (if (huffman-leaf? tree)
      (huffman-leaf-weight tree)
      (huffman-node-weight tree))))
```

Die Decodierung ist die einfachste Aufgabe beim Umgang mit Huffman-Bäumen. Deshalb kommt sie zuerst: Die Prozedur `huffman-decode` läuft den Huffman-Baum gemäß einer Bitfolge herunter, bis sie bei einem Blatt ankommt. Das Symbol am Blatt ist dann ein decodiertes Symbol:

```
(define huffman-decode
  (lambda (bits tree)
    (letrec
      ((decode-1
        (lambda (bits current-branch)
          (if (null? bits)
              '()
              (let ((next-branch
                    (choose-branch (car bits) current-branch)))
                (if (huffman-leaf? next-branch)
                    (cons (huffman-leaf-symbol next-branch)
                          (decode-1 (cdr bits) tree))
                    (decode-1 (cdr bits) next-branch)))))))
      (decode-1 bits tree))))

(define choose-branch
  (lambda (bit branch)
    (cond
      ((= bit 0) (huffman-node-left-branch branch))
      ((= bit 1) (huffman-node-right-branch branch))
      (else (error "bad bit")))))
```

Die Codierung findet Symbol für Symbol statt; die entstehenden Bitfolgen werden aneinandergehängt:

```
(define huffman-encode
  (lambda (message tree)
    (fold-right (lambda (symbol rest)
                  (append (huffman-encode-symbol symbol tree)
                          rest))
                '()
                message)))
```

Die Codierung muß für ein Symbol dasjenige Blatt finden, das mit ihm markiert ist. Da sich der Weg zum passenden Blatt nicht von der Wurzel aus sehen läßt, muß der Baum durchsucht werden. `huffman-encode-symbol` liefert `#f`, wenn das Symbol `symbol` nicht im Baum enthalten ist, und sonst eine Bitfolge, die das Symbol codiert:

```
(define huffman-encode-symbol
  (lambda (symbol tree)
    (if (huffman-leaf? tree)
        (if (equal? (huffman-leaf-symbol tree) symbol)
            '()
            #f)
        (let ((maybe-encoding
```

```

(huffman-encode-symbol symbol
  (huffman-node-left-branch tree)))
(if maybe-encoding
  (cons 0 maybe-encoding)
  (let ((maybe-encoding
        (huffman-encode-symbol symbol
          (huffman-node-right-branch tree))))
    (if maybe-encoding
      (cons 1 maybe-encoding)
      #f))))))

```

Es verbleibt das Problem, zu einer gegebenen Aufstellung der relativen Häufigkeiten der Symbole einer Sprache den entsprechenden Huffman-Baum zu konstruieren. Dabei sollten die Symbole mit der geringsten relativen Häufigkeit möglichst weit unten im Baum landen: „Weit unten“ heißt ja gerade „lange Codierung“.

Begonnen wird mit einer Menge der Blätter des Baums, konstruiert aus der Tabelle der Symbole und ihren relativen Häufigkeiten. Nun werden zwei Elemente mit geringstmöglichen Häufigkeiten aus der Menge entfernt, zu einem Binärbaum mit gerade diesen Elementen als linken und rechten Teilbaum verschmolzen, und der neu gewonnen Baum der Menge wieder hinzugefügt. Dieser Vorgang wird sukzessive wiederholt, bis die Menge einelementig ist; sie enthält dann gerade den Huffman-Baum für die Tabelle. Der entsprechende Prozess für den Beispielbaum sieht folgendermaßen aus:

Anfang	{(A, 8), (B, 3), (C, 1), (D, 1), (E, 1), (F, 1), (G, 1), (H, 1)}
Verschmelzen	{(A, 8), (B, 3), ({C, D}, 2), (E, 1), (F, 1), (G, 1), (H, 1)}
Verschmelzen	{(A, 8), (B, 3), ({C, D}, 2), ({E, F}, 2), (G, 1), (H, 1)}
Verschmelzen	{(A, 8), (B, 3), ({C, D}, 2), ({E, F}, 2), ({G, H}, 2), }
Verschmelzen	{(A, 8), (B, 3), ({C, D}, 2), ({E, F, G, H}, 4)}
Verschmelzen	{(A, 8), ({B, C, D}, 5), ({E, F, G, H}, 4)}
Verschmelzen	{(A, 8), ({B, C, D, E, F, G, H}, 9)}
Verschmelzen	{{A, B, C, D, E, F, G, H}, 17}