

Das Huffman- Kodierverfahren

Zusammenfassung

Der Huffman-Algorithmus kodiert die Zeichen eines Textes entsprechend ihrer Häufigkeit und hat damit Ähnlichkeit mit dem Morsecode. Zentraler Bestandteil des Algorithmus ist der Huffman-Kodierbaum, dessen Hierarchie die Häufigkeit der Zeichen widerspiegelt.

Die folgende Darstellung zeigt das Verfahren der Huffman-Kodierung zunächst sprachunabhängig an einem Beispiel auf. Dabei werden die einzelnen Schritte ausführlich aufgezeigt und grafisch veranschaulicht.

Im zweiten Teil wird eine Implementierung mit der funktionalen Sprache Haskell vorgestellt. Dabei wird der entsprechende Huffman-Baum mit Hilfe eines algebraischen Datentyps HTree realisiert. Die Entwicklung der einzelnen Funktionen erfolgt sowohl nach der bottom-up-Methode als auch im top-down-Verfahren. Zum Schluss an weiteren Beispielen erläutert.

Motivation

Der zunehmende Datenaustausch erfordert einen möglichst gut komprimierten Code. Neben mehreren bekannten Verfahren für Bilder und Videos (JPEG, MPEG) sind das ZIP-, RAR-, ARJ- und TGZ-Format weit verbreitet.

Einige Verfahren kodieren Folgen sich wiederholender Zeichen (sog. "runs"). Tritt ein run auf, dann wird dieser durch den entsprechenden Code mit Angabe der Wiederholungsrate ersetzt. (Laufängenkodierung - RLE).

Das Huffman Verfahren orientiert sich an der Idee des Morsealphabet: häufig vorkommende Zeichen sollten einen kurzen Code, seltene Zeichen können einen längeren Code erhalten. Zur Ermittlung der Zeichenhäufigkeit muss die Datei einmal gelesen werden.

Die meisten Programme verwenden einen festen Code für die Zeichen. Lange Zeit war der 8-Bit ASCII-Code üblich, heute verwendet man meist den Unicode (16-Bit).

Zeichen	Ascii	Unicode
'T'	0101 0100	0000 0000 0101 0100
'e'	0110 0101	0000 0000 0110 0101
'x'	0111 1000	0000 0000 0111 1000
't'	0111 0100	0000 0000 0111 0100

Das Wort "Text" wird demnach durch folgende Binärfolge dargestellt:

```
Ascii:  01010100 01100101 01111000 01110100
```

```
Unicode: 0000000001010100 0000000001100101
          0000000001111000 0000000001110100
```

Im Ascii-Code sind hierfür $4 \cdot 8 = 32$ Bit erforderlich, der Unicode benötigt sogar 64 Bit. Eine Buch mit ca. 300 Seiten á 45 Zeilen á 60 Zeichen benötigt demnach im ASCII-Code ca. $300 \cdot 45 \cdot 60 = 810\,000$ Byte. Dies entspricht etwa 790 KB. Im Unicode wäre die Datei doppelt so groß und würde nicht mehr auf eine Diskette passen.

Das Rückcodieren geht bei beiden Codes aufgrund ihrer festen Länge sehr einfach.

Bei einer variablen Codelänge muss darauf geachtet werden, dass der Text wieder eindeutig decodiert werden kann. Der folgende Code 1 wäre nicht sinnvoll, da die Zeichenfolge 111100 sowohl "Text" als auch "Teett" bedeuten könnte. Beim zweiten Code käme dieses Wort überhaupt nicht vor. man erkennt, dass jedes mögliche Wort wieder eindeutig zurückcodiert werden kann.

Zeichen	Code 1
't'	0
'e'	1
'x'	10
'T'	11

ungünstiger
Code

Zeichen	Code 2
't'	0
'e'	10
'x'	110
'T'	111

günstiger
Code

Das Problem lässt sich allgemein lösen, indem man nur Codes verwendet, bei dem kein Code ein Anfangsteil eines anderen Codes darstellt (Code 2). Einen solchen Code nennt man *Präfix-Code*.

Wenn man es schafft, die durchschnittliche Codelänge zu halbieren, schrumpft das oben erwähnte Buch auf knapp 400 KB zusammen.

Das Huffman-Verfahren

Das Huffman Verfahren ist sehr verbreitet und wird von vielen Komprimierprogrammen benutzt (z.B. PKZIP/PKUNZIP oder ARJ) oder in Verbindung mit anderen Verfahren (mit LZ77 bei LHA).

Zur Verdeutlichung wird nur ein kurzer Text mit wenigen Zeichen kodiert (keine Großbuchstaben).

```
FISCHERS_FRITZ_FISCHT_FRISCHE_FISCHE
```

1. Zuerst wird die Häufigkeit aller Zeichen ermittelt.

F	I	S	_	C	H	E	R	T	Z
5	5	5	4	4	4	3	3	2	1

2. Nun wird der sogenannte Huffman-Kodierbaum aufgebaut. Dabei wird jedes Zeichen als Blatt eines Binärbaumes gespeichert. Die Struktur des Baumes ergibt sich aus der Häufigkeit der einzelnen Zeichen.

- (a) Zunächst werden alle Zeichen in ein-blättrige Baume umgewandelt. Dabei werden die einzelnen Elemente entsprechend ihrer Häufigkeit (aufsteigend) sortiert.

```
[(Blatt 'Z' 1), (Blatt 'T' 2), (Blatt 'R' 3),
 (Blatt 'E' 3), (Blatt '_' 4), (Blatt 'H' 4),
 (Blatt 'C',4), (Blatt 'S' 5), (Blatt 'I' 5),
 (Blatt 'F' 5)]
```

- (b) Nun werden die Baume mit der geringsten Häufigkeit verschmolzen (Links der Baum mit der größeren Häufigkeit). Die Summe der einzelnen Häufigkeit ergibt dabei die Gesamthäufigkeit. Der neue Baum wird dann wieder in die Liste entsprechend seiner Häufigkeit eingefügt.

```
3 • ; (r,3) ; (e,3) ; (_,4) ; (h,4) ; (c,4) ; (s,5) ; (i,5) ; (f,5)
  / \
 T   Z
```

- (c) Das Verfahren wird nun solange wiederholt, bis die Liste nur noch einen Baum, den Huffman-Kodierbaum, enthält.

```
(e,3) ; (_,4) ; (h,4) ; (c,4) ; (s,5) ; (i,5) ; (f,5) ;
                                     6 •
                                    / \
                                   3 • R
                                   / \
                                  T   Z
```

```
(h,4) ; (c,4) ; (s,5) ; (i,5) ; (f,5) ;
                                     6 • ; 7 •
                                    / \ ; / \
                                   3 • R ; _ E
                                   / \
                                  T   Z
```

```
(s,5) ; (i,5) ; (f,5) ;
                                     6 • ; 7 • ; 8 •
                                    / \ ; / \ ; H C
                                   3 • R ; _ E
                                   / \
                                  T   Z
```

```
(f,5) ;
                                     6 • ; 7 • ; 8 • ; 10 •
                                    / \ ; / \ ; / \ ; S I
                                   3 • R ; _ E
                                   / \
                                  T   Z
```

```
7 • ; 8 • ; 10 • ;
 / \ ; H C ; S I ;
 _ E
                                     11 •
                                    / \
                                   6 • F
                                   / \
                                  3 • R
                                  / \
                                 T   Z
```


Zeichencodes gemäß Huffman-Baum

I	S	F	E	H	C	_	R	Z	T
100	101	110	000	001	011	010	1110	1110	1111

4. Der Beispieltext kann nun nach dieser Tabelle kodiert werden.

```
110 100 101 011 001 000 1110 101 010 110 1110 100 1111 1110
F   I   S   C   H   E   R   S   _   F   R   I   T   Z
```

```
010 110 100 101 011 001 1111 010 110 1110 100 101 011 001 000
_   F   I   S   C   H   T   _   F   R   I   S   C   H   E
```

```
010 110 100 101 011 001 000
_   F   I   S   C   H   E
```

Tatsächlich sind die Bits natürlich nicht getrennt sondern bilden einen Strom von 0 und 1:

```
110100101011001000111010101011011101001111111001011010010
101100111110101101110100101011001000010110100101011001000
```

5. Zum Dekodieren benötigt man eine Funktion, die die Bitfolge anhand des Kodierbaumes zurückübersetzt. Dabei wandert man in dem Baum bis zu einem Blatt, liest das entsprechende Zeichen aus und setzt diesen Vorgang wieder bei der Wurzel beginnend fort, bis die Bitfolge leer ist. Bleiben dabei am Ende einige Bits übrig oder kommt man auf eine Folge von Bits, die zu keinem Blatt führen, so ist der Code ungültig.

Kurzübersicht über das Vorgehen:

- Erstelle eine Liste der vorkommenden Buchstaben mit ihrer Häufigkeit (aufsteigend nach den Häufigkeiten sortiert).
- Wandle diese Liste in eine Liste von Bäumen (Blätter) um.
- Verschmelze jeweils die zwei Bäume mit der geringsten Häufigkeit zu einem Gesamtbaum, bis ein einzelner Baum entstanden ist.
- Die Pfade in diesem Kodierbaum ergeben den Code .
- Eine Codefunktion ermittelt den Pfad eines Zeichens im Kodierbaum.
- Eine Dekodierfunktion wandelt eine Bitfolge mit dem Kodebaum in den Text zurück.

Implementierung

In diesem Kapitel werden schrittweise Funktionen zum kodieren und dekodieren von Texten nach dem Huffman-Kodieralgorithmus entworfen.

Zeichenliste erstellen

Zunächst muss der Text eingelesen und eine Zeichentabelle erstellt werden. Für jedes Zeichen wird dabei die Häufigkeit mitabgespeichert. Diese Liste muss nach der Häufigkeit der Zeichen aufsteigend sortiert sein.

Im dargestellten Beispiel muss man folgende Liste erhalten:

```
> makeZeichenliste "fischers_fritz_fischt_frische_fische"
[(z,1),(t,2),(r,3),(e,3),(_,4),(h,4),(c,4),(s,5),(i,5),(f,5)]
```

Für ein einzelnes Zeichen mit seiner Häufigkeit wird sinnvollerweise ein Datentyp Zeichen vereinbart.

```
type Zeichen = (Char,Int)
```

Die Entwicklung der Funktion `makeZeichenliste` wird nun im *top-down-Verfahren* entwickelt. Dabei geht man von der gewünschten Zielfunktion aus und reduziert das Problem durch Einführung neuer Funktionen.

Zunächst wird also die Zielfunktion definiert:

```
-- -----
-- (1) Erstellen einer Liste der vorkommenden Zeichen
--      sortiert nach der Häufigkeit (aufsteigend)
-- -----
makeZeichenliste :: String -> [Zeichen]
makeZeichenliste xs = qsortlist (mklist xs [])
```

Die Funktionen `qsortlist` und `mklist` müssen nun im nächsten Schritt erstellt werden.

Die Funktion `qsortlist` sortiert eine Liste von Elementen der Form `(Char, Int)` entsprechend der 2. Komponente aufsteigend. Als Sortieralgorithmus wird Quicksort verwendet.

```
qsortlist :: [Zeichen] -> [Zeichen]
qsortlist [] = []
qsortlist ((x,n):xs) = qsortlist kleiner
                      ++ [(x,n)]
                      ++ qsortlist groesser
  where kleiner = [(y,m) | (y,m) <- xs, m < n]
        groesser = [(y,m) | (y,m) <- xs, m >= n]
```

Die zweite Funktion `mklist` erstellt aus dem String die Liste der Zeichen. Dabei braucht auf die Sortierung keine Rücksicht genommen werden. Diese Funktion stützt sich auf eine Hilfsfunktion `add2list`, die ein Element in eine Liste einfügt. `mklist` selbst benutzt die Akkumulatortechnik. Die aufzubauende Liste wird als 2. Parameter mitübergeben und Schritt für Schritt vervollständigt. Der initiale Aufruf muss hier mit einer leeren Liste erfolgen (siehe `makeZeichenliste`).

```
mklist :: String -> [Zeichen] -> [Zeichen]
mklist [] liste = liste
mklist (x:xs) liste = mklist xs (add2list x liste)
```

Die Funktion `add2list` fügt schließlich ein Zeichen in eine übergebene Liste ein. Dabei können zwei Fälle auftreten: Wenn das Zeichen bereits enthalten ist, dann wird die Häufigkeit um eins erhöht. Andernfalls wird das neue Zeichen mit der Häufigkeit eins am Ende eingefügt.

```
add2list :: Char -> [Zeichen] -> [Zeichen]
add2list c [] = [(c,1)]
add2list c ((l,n):ls)
  | c==l      = (l,n+1):ls
  | otherwise = (l,n):add2list c ls
```

Huffman-Kodierbaum erstellen

Im zweiten Schritt muss nun aus der Liste der vorkommenden Zeichen ein Kodierbaum erstellt werden. Hierzu muss zunächst eine geeignete Datenstruktur bereitgestellt werden. Dabei müssen folgende Fälle für einen Huffman-Baum berücksichtigt werden:

- Er besitzt Blätter, in denen einzelne Zeichen abgespeichert sind.
- Er besitzt innere Knoten, in denen die Gesamthäufigkeit aller Blätter der beiden Unteräume abgespeichert sind.

Realisierung:

```
data HTree = Blatt Char Int | Knoten HTree Int HTree
  deriving (Eq, Show)
```

Die Entwicklung des Kodierbaumes soll nun nach der *bottom-up-Methode* vorgestellt werden. Dabei ist das Ziel eine Funktion, die aus einer Liste von Zeichen einen geeigneten Huffman-Baum erzeugt.

```
-- Ziel:
makeHuffmanntree :: [Zeichen] -> HTree
```

1. Schritt:

Zunächst muss die Liste der Zeichen in eine Liste von Blättern umgewandelt werden. Dies geht mit Hilfe der map-Funktion ganz einfach, wenn es eine Funktion gibt, die ein Zeichen in ein Blatt umwandelt

```
-- aus einem Zeichen ein Blatt machen
makeBlatt      :: Zeichen -> HTree
makeBlatt (c,n) = Blatt c n

-- Zeichenliste in Baumliste umwandeln
list2tree :: [Zeichen] -> [HTree]
list2tree xs = map makeBlatt xs
```

2. Schritt:

Nun müssen die ersten beiden Elemente der Baumliste zu einem gemeinsamen Baum verschmolzen werden. Dabei bildet das Element mit der größeren Häufigkeit den linken Unterbaum. Die Summe der beiden Einzelhäufigkeiten wird im neuentstandenen Knoten abgespeichert.

Hierfür sind eine Reihe von Unterfunktionen nötig:

- Ermitteln der Häufigkeit eines Baumes
- Verschmelzen zweier Bäume
- Einfügen eines Baumes in eine sortierte Liste

Die Bestimmung der Häufigkeit eines Baumes hängt davon ab, ob es sich um ein Blatt oder einen Knoten handelt.

```
gewicht :: HTree -> Int
gewicht (Blatt c n)      = n
gewicht (Knoten li n re) = n
```

Bei der Verschmelzung muss die Häufigkeit der beiden Teilbäume berücksichtigt werden.

```
-- links den Baum mit dem größeren Gewicht
verschmelzen :: HTree -> HTree -> HTree
verschmelzen b1 b2
  | n1 >= n2    = Knoten b1 (n1+n2) b2
  | otherwise  = Knoten b2 (n1+n2) b1
  where n1 = gewicht b1
        n2 = gewicht b2
```

Ein verschmolzener Baum muss nun an die richtige Stelle wieder eingefügt werden. Da die Liste stets sortiert ist, gelingt dies durch einfache Rekursion.

```

-- einfuegen eines Baums in eine Liste von Bäumen
-- entsprechend dem Baumgewicht (aufsteigend)
instTreeInList :: HTree -> [HTree] -> [HTree]
instTreeInList b [] = [b]
instTreeInList b (x:xs)
  | gewicht b <= gewicht x = b:x:xs
  | otherwise               = x:instTreeInList b xs

```

3. Schritt:

Nun müssen solange die ersten beiden Bäume der Liste verschmolzen werden, bis die Liste nur noch ein Element, den Huffman-Kodierbaum, enthält.

```

-- schrittweise verschmelzen der Bäume
mkHTree :: [HTree] -> [HTree]
mkHTree [b] = [b] -- Stoppfall
mkHTree (b1:b2:bs) = mkHTree (instTreeInList b bs)
                    where b=verschmelzen b1 b2

```

4. Schritt:

Damit kann die gewünschte Zielfunktion `makeHuffmantree` erstellt werden. Sie ruft die beschriebenen Funktionen nun nur noch in einer geeigneten Weise auf.

```

-- -----
-- (2) Erstellen des Huffman-Kodierbaumes
-- -----

makeHuffmantree :: [Zeichen] -> HTree
makeHuffmantree xs = head (mkHTree (list2tree xs))

```

Nachricht kodieren

Endlich können Nachrichten nun kodiert werden. Die Entwicklung einer geeigneten Funktion `kodieren` erfolgt wieder per *top-down-Verfahren*.

1. Schritt:

Gesucht ist eine Kodierfunktion, die einen String in einen Binärcode umwandelt. Da für das Dekodieren aber der hierbei entwickelte Huffman-Baum benötigt wird, muss die Kodierfunktion auch diesen zurückliefern.

```

-- -----
-- (3) Kodieren einer Nachricht
-- -----

```

```

kodieren :: String -> (String,HTree)
kodieren xs = (kodieren' xs htree,htree)
  where htree = makeHuffmantree (makeZeichenliste xs)

```

2. Schritt:

kodieren stützt sich auf die Hilfsfunktion `kodieren'`, die einen Text mit Hilfe eines Baumes kodiert. Dabei wird der Eingabestring Zeichen für Zeichen kodiert (Rekursion). Die Kodierung eines einzelnen Zeichens erfolgt mit der Funktion `code`

```

kodieren' :: String -> HTree -> String
kodieren' [] b = []
kodieren' (x:xs) b = (code b x)++kodieren' xs b

```

3. Schritt:

Den Code eines Zeichens kann man durch Suchen im Kodierbaum finden. Diese Methode ist jedoch sehr Zeitaufwendig, da der Kodierbaum über die Funktion `member` mehrfach durchlaufen wird.

```

code :: HTree -> Char -> String
code (Blatt c n) x
  | c==x      = []
  | otherwise  = error "Zeichen nicht in Baum enthalten"
code (Knoten li n re) x
  | member x li  = '1':code li x
  | otherwise    = '0':code re x
  where member a (Blatt c n)      = (a==c)
        member a (Knoten li c re) = (member a li) || (member a re)

```

Nachrichten dekodieren

Zum dekodieren einer Nachricht wird natürlich der benutzte Kodierbaum benötigt. Dieser muss also bei einer Datenkomprimierung / Datenübertragung mitgespeichert werden.

Das Dekodieren erfolgt nach folgendem Schema: Aus dem Code werden solange Bits gelesen, bis man im zugehörigen Kodierbaum auf ein Blatt trifft. Das so ermittelte Zeichen wird als Klartext zurückgegeben. Anschließend wird mit dem Rest der kodierten Nachricht nach demselben Muster weiterverfahren.

Die Funktionen `dekodieren` benutzt die Hilfsfunktion `dekodieren'`, die als zusätzlichen Parameter eine Kopie des Kodierbaumes erhält. Diese wird benötigt, da beim rekursiven Durchlaufen des Baumes der Originalbaum verloren geht. Sobald ein Zeichen gefunden wurde, muss aber die erneute Suche wieder mit dem Originalbaum beginnen.

Bleiben dabei am Ende einige Bits übrig oder kommt man auf eine Folge von Bits, die zu keinem Blatt führen, so ist der Code ungültig.

```

-----
-- (4) Dekodieren einer Nachricht
--     Es wird der zugehörige HTree benötigt
-----

dekodieren :: String -> HTree -> String
dekodieren s b = dekodieren' s b b

dekodieren' :: String -> HTree -> HTree -> String
dekodieren' [] b (Blatt c n) = [c]
dekodieren' [] b (Knoten b1 n b2) = error "Code ungültig"
dekodieren' xs b (Blatt c n) = c:dekodieren' xs b b
dekodieren' (x:xs) b (Knoten li n re)
  | x=='1'      = dekodieren' xs b li
  | otherwise   = dekodieren' xs b re

```

Effizienz

Der Beispieltext "FISCHERS FRITZ FISCHT FRISCHE FISCHE" besitzt 36 Zeichen (inklusive Leerzeichen). Bei der Speicherung/Übertragung im 8-Bit-ASCII-Code werden dafür $8 * 36 = 288$ Bit benötigt.

Der Huffman-kodierte Text hat eine Länge von 114 Bit. Dies entspricht einer Datenkompression von ca. 60 %.

Eine noch bessere Reduzierung würde man erhalten, wenn ganze Silben im Baum abgespeichert würden. Dies würde jedoch den Aufwand erheblich steigern.

Beispiele

Die vorgestellten Funktionen können nun für beliebige Eingaben angewandt werden.

Beispiel 1:

```

> kodieren "abrakadabra"
("01101001111011100110100",
 Knoten (Knoten (Knoten (Knoten (Blatt 'k' 1) 2 (Blatt 'd' 1))
 4 (Blatt 'b' 2)) 6 (Blatt 'r' 2)) 11 (Blatt 'a' 5))

> dekodieren "01101001111011100110100" _
      (snd (kodieren "abrakadabra"))
"abrakadabra"

```

Beispiel 2:

```
> kodieren "simsalabim"
("001111110001001110010111110",
 Knoten (Knoten (Knoten (Blatt 'i' 2) 4 (Blatt 'm' 2)) 6
 (Blatt 'a' 2)) 10 (Knoten (Knoten (Blatt 'l' 1) 2
 (Blatt 'b' 1)) 4 (Blatt 's' 2)))

> dekodieren "001111110001001110010111110" _
      (snd (kodieren "simsalabim"))
"simsalabim"
```

Anwendung : Kodieren/Dekodieren einer Textdatei

In der klassischen Haskell-Programmierung kommen interaktive Ein- und Ausgabe nicht vor; diese lassen sich aber durchaus realisieren. Die Grundidee, dass keine sequentiellen Fragmente benutzt werden, muss dabei aber aufgegeben werden. Schließlich soll zunächst eine Textdatei eingelesen, dann kodiert und schließlich wieder dekodiert werden.

Haskell bietet folgende Konstruktion zur sequentiellen Bearbeitung an:

```
codefile dateiname =
  do s<- readFile filename
     putStrLn s
     ...
```

Dieses "Programm" ist wie folgt zu lesen:

- Das Programm wird mit `codefile "dateiname"` aufgerufen.
- Zunächst wird dann die Datei eingelesen. Mit `s` (String) kann auf den Text anschließend zugegriffen werden.
- Der Text kann z.B. zeilenweise ausgegeben werden (`putStrLn`).
- Es ist darauf zu achten, dass die Befehle hinter `do` gleich weit eingerückt werden.

Sinnvollerweise sollte ein möglichst großer Teil des "Programms" in echte Funktionen ausgelagert werden. Z.B. kann die Statistik der Huffman-Codierung in eine (klassische) Haskellfunktion ausgelagert werden. Diese liefert einen String als Ergebnis zurück, der dann ausgegeben werden kann (`putStrLn (statistik s)`).

```
statistik s = "Statistik :\n"
  ++ "-----\n"
  ++ "Textlänge      =" ++ (show ls)      ++ " Zeichen\n"
  ++ "Textlänge      =" ++ (show (ls*8)) ++ " Bits\n"
```

```
++ "Codelänge      =" ++ (show lc)      ++ " Bits\n"
++ "Komprimierung : " ++ (show kompr)  ++ " %\n"
++ "-----\n"
  where (c,b) = kodieren s
        ls = length s
        lc = length c
        kompr = fromInt (ls*8-lc)/(fromInt (ls*8)) *100
```

Wendet man diese Funktion etwa auf einen Auszug von Shakespeares Julius Caesar an (*Act 2, Scene 1: Rome. BRUTUS's orchard*), so erhält man folgende Übersicht:

```
Tree> codefile "caesar21.txt"
Statistik :
-----
Textlänge      = 12111 Zeichen
Textlänge      = 96888 Bits
Codelänge      = 55321 Bits
Komprimierung  : 42.9021 %
-----
```

Eine Auswertung der Kurzgeschichte „Auf der Galerie“ von Franz Kafka ergibt:

```
Tree> codefile "kafka.txt"
Statistik :
-----
Textlänge      = 2121 Zeichen
Textlänge      = 16968 Bits
Codelänge      = 9570 Bits
Komprimierung  : 43.5997 %
-----
```

Literatur

- [1] Richard Bird, Philip Wadler : *Einführung in die funktionale Programmierung*
Hanser-Verlag München Wien 1992

Standardwerk zur Einführung, Programmiersprache *Miranda*

- [2] Simon Thompson : *MIRANDA - The Craft Of Functional Programming*
Addison-Wesley Co. , Inc. 1995

Hierzu gibt es auch eine neuere Ausgabe, das die Sprache Haskell benutzt.

- [3] Christoph Oehler, Raimond Reichert : *Applet und Lernaufgabe zu "Kompression"*, *ETH Zürich*

<http://www.educeth.ch/informatik/interaktiv/kompression/>