Technische Universität München

Fakultät für Informatik

# Huffman-Codierung

**Proseminar im Rahmen der Vorlesung „Perlen der Informatik“**

**Thema:** Huffman-Codierung

**von:** Martin Klenk

**gehalten am:** 10. Juli 2008

**Betreuer:** Herr Prof. T. Nipkow

# 1 Einführung

Es gibt viele Möglicheiten, eine bestimmte Datennachricht zu repräsentieren. Bei der verlustfreien Kompression wird eine Nachricht mit dem Ziel dargestellt, durch die Entfernung von Redundanz die Länge der gesamten Nachricht ohne Informationsverlust zu minimieren. Hierfür gibt es verschiedene Möglichkeiten. Eine davon ist die Huffman-Codierung. Zur Veranschaulichung soll hier zur

Tabelle 1: Tabelle der Häufigkeiten des Beispielfiles

| | P | e | r | l | n | 2 |
|---|---|---|---|---|---|---|
| Häufigkeit | 2250 | 650 | 600 | 800 | 450 | 250 |
| Codewort | 0 | 101 | 100 | 111 | 1101 | 1100 |

Einführung ein kleines Beispiel dienen: Angenommen ein zu kodierendes Datenfile besitzt 5000 Zeichen, jedes Zeichen benötigt zunächst 8 Bit. Die gesamte Nachricht hat somit eine Länge von 40000 Bit. Wir stellen fest, dass in der Nachricht lediglich die Zeichen $\{P, e, r, l, n, 2\}$ vorkommen. Nun bestimmen wir die Häufigkeit der einzelnen Buchstaben. Die Häufigkeiten sowie Beispielcodes sind in der Tabelle 1 dargestellt. Dieser Code benötigt lediglich

$$2250 * 1 + 650 * 3 + 600 * 3 + 800 * 3 + 450 * 4 + 250 * 4 = 11200 \text{ Bit}$$

womit die Nachricht nur noch etwa ein Viertel des ursprünglichen Platzes belegt. Doch wie kann man einen solchen geeigneten Code erstellen?

# 2 Die Grundlage - Präfix-Codes

Damit die kodierte Nachricht wieder dekodierbar ist, hat man die Möglichkeit, jedem Zeichen ein Codewort von konstanter Länge zuzuweisen. Im Beispiel würde man 6 verschiedene Codewörter benötigen, jedes Codewort hätte eine Länge von 3 Bit. Damit würden jedoch statt den obigen 11200 Zeichen 15000 Zeichen benötigt, um die Nachricht kodiert zu speichern. Auch würde hierbei nicht ausgenutzt werden, dass manche Zeichen häufiger vorkommen, als andere. Eine andere Möglichkeit ist die Verwendung eines Präfix-Codes. Beim genaueren Betrachten der Codeworte des Beispiels fällt auf, dass kein Codewort ein anderes Codewort als Präfix enthält. Die Nachricht kann durch Aneinanderhängen der entsprechenden Codewörter codiert werden. Anschließend kann mit Hilfe der bekannten Übersetzungstabelle die ursprüngliche Nachricht wieder hergestellt werden. Das einzige Problem, welches sich beim Dekodieren stellt, ist eine geeignete Form für die Darstellung der Codewörter zu finden. Man stelle sich einen Binärbaum vor, in dem der linke Zweig eines Knotens mit einer Null, der rechte Zweig mit einer Eins belegt ist. In einem solchen Baum stellt jedes Codewort eines Präfix-Codes einen Pfad zu einem einzigen Blatt dar. Dies wird

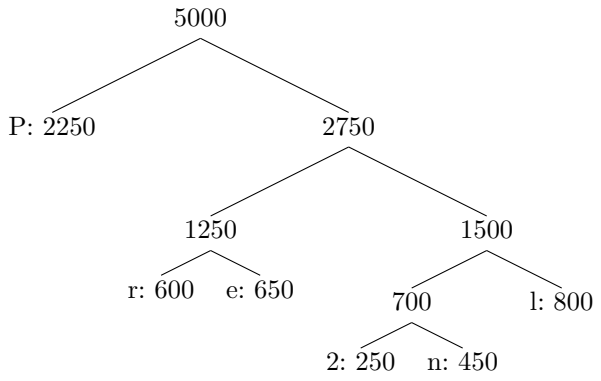


Abbildung 1: Beispielbaum

in Abbildung 1 noch einmal verdeutlicht. Eine optimale Kodierung wird immer durch einen vollständigen Binärbaum dargestellt. Daraus folgt: Wird aus einem Alphabet $\Sigma$ ein Baum erstellt, dann besitzt der Baum zum optimalen Präfixcode $|\Sigma|$ Blätter und $|\Sigma| - 1$ innere Knoten. Die Anzahl der Bits, welche zum Enkodieren einer Nachricht benötigt werden, lässt sich damit auf einfache Art und Weise berechnen. Gegeben sei ein Baum $C$, welcher zu einem Präfix-Code gehört. Es sei für $\sigma \in \Sigma$ $h(\sigma)$ die Häufigkeit eines Zeichens und $t(\sigma)$ die Tiefe des Blattes im Baum und somit die Länge des Codeworts. Dann ist

$$K(C) = \sum_{\sigma \in \Sigma} h(\sigma) * t(\sigma)$$

die Anzahl der Bits, welche für die Codierung einer Nachricht benötigt werden. Dies stellt die *Kosten* des Baumes dar.

# 3 Konstruktion einer Huffman-Codierung - Der Algorithmus

Der Amerikaner David A. Huffman hat 1952 einen Greedy-Algorithmus entwickelt, der einen Präfix-Baum zu einer Nachricht aufbaut.
Der als Huffman-Codierung bekannte Algorithmus erzeugt einen Baum zu einem optimalen Präfix-Code, wie wir später beweisen werden. Der Pseudocode in Algorithm 1 soll die Arbeitsweise des Algorithmus verdeutlichen. Die Funktion Huffman ist die Hauptfunktion des Algorithmus. Sie überführt unseren String, repräsentiert durch eine Liste von einzelnen Charaters in einen Baum. Hierzu wird zunächst die Funktion CreateTupleList und anschließend die Funktion CreateTree angewendet.

Betrachten wir CreateTupleList: CreateTupleList überführt eine Liste von einzelnen Zeichen in eine Liste von Tupeln der Form (Char, Int). Hierbei bilden jeweils ein einzelnes Zeichen zusammen mit der Häufigkeit im ursprünglichen

**Algorithm 1** Huffman(M)

```
Huffman : [Char] → Tree
Huffman = CreateTree ∘ CreateTupleList

CreateTupleList : [Char] → [(Char, Int)]
CreateTupleList = Sort ∘ MergeLetters

CreateTree : [(Char, Int)] → Tree
CreateTree = BuildTree ∘ ConvertToLeaves

ConvertToLeaves : [(Char, Int)] → [Tree]
ConvertToLeaves = map (uncurry Leaf)

BuildTree : [Tree] → Tree
BuildTree (x1:x2:xs) = BuildTree (Insert (Merge x1 x2) xs))
BuildTree [x] = x

Merge: Tree → Tree → Tree
Merge x1 x2 = Node ((Value x1) + (Value x2)) x1 x2

Insert: Tree → [Tree] → [Tree]
Insert x (first:ls)
            | (Value x) > (Value first) = first:(Insert x ls)
            | otherwise = x:first:ls

data Tree = Leaf Char Int |
            Node Int Tree Tree
Value: Tree → Int
Value (Leaf _ k) = k
Value (Node k _ _) = k
```

String ein Tupel. Anschließend werden diese Tupel anhand ihrer Häufigkeit aufsteigend geordnet. Das erste Element in der Liste ist jenes mit der kleinsten Häufigkeit. Im Fall, dass zwei Elemente die gleiche Häufigkeit besitzen, wird das Element mit dem kleineren Zeichenwert als erstes Element genommen. Das Zählen und Ordnen der Zeichen wird durch eine Komposition der beiden Funktionen MergeLetters, welche die (Char, Int)-Tupel erstellt, und Order, welche die Tupel aufsteigend ordnet.

Nachdem die sortierte Liste der Tupel erfolgreich erstellt wurde, wird eben diese Liste an die Funktion CreateTree weitergeleitet, welche selbst wiederum eine Komposition von zwei Funktionen ist. Die erste Funktion, ConvertToLeaves, ist eine einfach Konvertierungsfunktion. Als Eingabe nimmt diese Funktion eine Liste von (Char, Int)-Tupeln welche in ein Blatt unseres Baumes überführt werden. Ein Blatt besitzt zwei Attribute, zum einen das Zeichen, was von diesem Blatt repräsentiert werden soll, zum anderen die Anzahl des entsprechenden Zeichens. Die Liste der Blätter ist nach den gleichen Kriterien geordnet wie die Liste der Tupel.

Mit dieser Blätterliste wird nun die Funktion BuildTree aufgerufen, welche eine Liste von Trees (Bäumen) in einen einzigen Tree überführt. Ein Tree ist hierbei die „Basisklasse“ von Leaf (Blatt) und Node (Knoten). Der Tree, der dabei zurückgegeben wird ist die Wurzel unseres Huffman-Baumes. Die Funktion BuildTree verwendet die Hilfsfunktion Merge welche als Argument zwei Parameter vom Typ Tree erhält. Diese werden kombiniert zu einer neuen Node, wobei der erste Parameter den linken Teilbaum der Node, der zweite Parameter den rechten Teilbaum der Node bilden. Wie ein Leaf erhält auch jede Node einen Wert. Dieser entspricht der Summe der beiden Werte der Teilbäume. BuildTree nimmt nun jeweils die ersten beiden Elemente aus der Liste. Dies sind die beiden Elemente mit der kleinsten Häufigkeit, was vorher durch die Sortierung sichergestellt wurde. Diese beiden Elemente werden mit der Funktion Merge zusammengesetzt und ein neuer Knoten wird erzeugt. Anschließend wird dieser neue Knoten mit der ursprünglichen Liste verkettet und die Sortierung wird in der Liste wieder hergestellt. Dies übernimmt die Funktion Sort. Zuletzt wird BuildTree rekursiv aufgerufen. Wenn beim rekursiven Aufruf nur noch ein einziges Element in der Liste ist, dann wird dieses eine Element zurückgeliefert. Dieser Fall tritt ein, wenn alle anderen Elemente gemerged wurden. Das heißt, dass dieses Element die Wurzel des Baumes bildet. Eben diese Wurzel wird schlussendlich auch von der Funktion Huffman zurückgeliefert und der Aufrufer der Funktion kann aus dem Baum die entsprechenden Codewörter ableiten.

# 4 Korrektheit des Huffman-Algorithmus

Es bleibt zu beweisen, dass es sich beim Huffman-Baum um einen Baum handelt, welcher einen optimalen Präfix-Code erzeugt. Da es sich beim Huffman-Algorithmus um einen Greedy-Algorithmus handelt, muss einerseits gezeigt werden, dass das Problem die Greedy-Eingenschaft besitzt und zweitens, dass sich eine optimale Lösung aus optimalen Lösungen von Teilproblemen zusammen-

setzt.

## 4.1 Greedy-Eigenschaft

### Lemma 1

Sei $\Sigma$ ein Alphabet, in dem jedes Zeichen $\sigma \in \Sigma$ die Häufigkeit $h[\sigma]$ besitzt. Seien $x$ und $y$ zwei Zeichen aus $\Sigma$ mit den niedrigen Häufigkeiten. Dann existiert ein optimaler Präfix-Code für $\Sigma$, in dem die Codewörter für $x$ und $y$ die gleiche Länge haben und sich nur im letzten Bit unterscheiden.

### Beweis

Hierbei ist die Idee des Beweises, einen Baum $T$ zu einem optimalen Präfix-Code so zu modifizieren, dass er einen anderen optimalen Präfix-Code darstellt und die beiden Zeichen $x$ und $y$ als Bruderblätter mit maximaler Tiefe vorkommen. Wenn dies gelingt, dann unterscheiden sich deren Codewörter nur in der letzten Stelle. Seien $a$ und $b$ zwei weitere Zeichen aus $\Sigma$, die in $T$ maximale Tiefe haben. Wieterhin nehmen wir o.B.d.A an, dass $h[a] \leq h[b]$ und $h[x] \leq h[y]$ gelten. Da die beien Zeichen $x$ und $y$ so gewählt wurden, das die Häufigkeit minimal ist, gilt außerdem $h[x] \leq h[a]$ und $h[y] \leq h[b]$. Es wird nun ein Baum $T'$ erzeugt, indem die beiden Blätter $a$ und $x$ vertauscht werden. Dieser wird anschließend zum Baum $T''$ durch vertauschen von $b$ und $y$ modifiziert. Nun berechnen wir die Differenz der Kosten von $T$ zu $T'$

$$
\begin{aligned}
K(T) - K(T') &= \textstyle\sum_{\sigma\in\Sigma} h[\sigma]t_T[\sigma] - \sum_{\sigma\in\Sigma} h[\sigma]t_{T'}[\sigma] \\
&= (h[x]t_T[x] + h[a]t_T[a]) - (h[x]t_{T'}[x] + h[a]t_{T'}[a]) \\
&= h[x]t_T[x] + h[a]t_T[a] - h[x]t_T[a] + h[a]t_T[x] \\
&= \underbrace{(h[a] - h[x])}_{\geq 0} * \underbrace{(t_T[a] - t_T[x])}_{\geq 0} \\
&\geq 0
\end{aligned}
$$

Analog kann dies für die Differenz von $T'$ zu $T''$ gezeigt werden. Da sich beim Vertauschen der Blätter die Kosten des Baumes nicht erhöht haben und $T$ ein optimaler Baum ist, ist auch $T''$ ein optimaler Baum. Es gilt: $K(T) = K(T'')$. $T''$ ist also ein optimaler Baum in dem $x$ und $y$ als Bruderblätter maximaler Tiefe auftauchen. Daraus folgt Lemma 1. Lemma 1 zeigt außerdem, dass beim Bau eines optimalen Baumes durch Verschmelzung mit einer Greedy-Entscheidung begonnnen werden kann und die beiden Knoten mit der geringsten Häufigkeit zuerst ausgewählt werden. Dies stellt eine optimale Lösung für eben dieses Teilproblem dar, da ein Baum genau dann optimal ist, wenn seine Kosten minimal sind. Eben diese Funktionsweise setzt der Huffman-Algorithmus um. Er wählt jeweils aus allen möglichen Verschmelzungen jene, die die geringsten Kosten verursacht. Nun ist noch zu zeigen, dass sich die Gesamtstruktur aus optimalen Lösungen der Teilstrukturen zusammensetzt.

## 4.2 Eigenschaft der optimalen Teilstruktur

**Lemma 2**

Sei $\Sigma$ ein Alphabet, $\sigma \in \Sigma$ habe die Häufigkeit $h[\sigma]$ und $x$ und $y$ seien zwei Zeichen mit minimaler Häufigkeit in der Nachricht. Es wird nun ein neues Alphabet $\Sigma'$ gebildet, für das gilt: $\Sigma' = \Sigma \backslash \{x, y\} \cup z$ wobei $h[z] = h[x] + h[y]$ und $z \notin \Sigma$. Sei $T'$ ein Baum, der einen optimalen Präfix-Code für $\Sigma'$ darstellt. Dann stellt $T$ einen optimalen Präfix-Code für $\Sigma$ dar, wenn $T$ aus $T'$ entsteht, indem das Blatt $z$ durch einen inneren Knoten ersetzt wird, der $x$ und $y$ als Kindknoten besitzt.

**Beweis**

Zunächst wird gezeigt, das die Kosten $K(T)$ als Funktion der Kosten $K(T')$ dargestellt werden können. Für jedes $\sigma \in \Sigma \backslash \{x, y\}$ gilt $t_T[\sigma] = t_{T'}[\sigma]$ und daher $h[\sigma]t_T[\sigma] = h[\sigma]t_{T'}[\sigma]$.
Da $t_T[x] = t_T[y]t_T[z] + 1$:

$$\begin{aligned} h[x]t_T[x] + h[y]t_T[y] &= (h[x] + h[y]) * (t_{T'}[z] + 1) \\ &= h[z]t_{T'}[z] + (h[x] + h[y]) \end{aligned}$$

woraus folgt

$$K(T) = K(T') + h[x] + h[y]$$

oder

$$K(T') = K(T) - h[x] - h[y]$$

Der Beweis wird nun als indirekter Beweis geführt. Wir nehmen an, dass T keinen optimalen Präfix-Code für $\Sigma$ darstellt. Dann existiert ein Baum $T''$, sodass $K(T'') < K(T)$ gilt. Nach Lemma 1 darf o.B.d.A angenommen werden, dass $x$ und $y$ als Brüder in $T''$ enthalten sind. Es wird nun ein Baum $T'''$ erstellt, in dem der gemeinsame Vater von $x$ und $y$ durch ein neues Blatt $z$ ersetzt wird, für das wiederrum gilt, dass $h[z] = h[x] + h[y]$. Dann gilt:

$$\begin{aligned} K(T''') &= K(T'') - h[x] - h[y] \\ &< K(T) - h[x] - h[y] \\ &= K(T') \end{aligned}$$

Da $T'$ jedoch als Baum definiert wurde, der einen optimalen Präfix-Code darstellt, ist dies ein Widerspruch zur Annahme, dass $T$ keinen optimalen Präfix-Code darstellt. Somit muss $T$ einen optimalen Präfix-Code darstellen.

**Lemma 3**

Der Huffman-Algorithmus erzeugt einen optimalen Präfix-Code.

**Beweis**

Wir haben in Lemma 1 bewiesen, dass sich die Teilprobleme durch die Greedy-Strategie optimal lösen lassen und in Lemma 2, dass sich eine gesamte Lösung aus optimalen Lösungen zusammensetzt. Dies ist eben die Vorgehensweise, nach der der Huffman-Algorithmus arbeitet. Er löst ein Teilproblem und substituiert dann zwei Knoten $x$ und $y$ durch einen neuen Knoten. Daher folgt Lemma 3 unmittelbar aus Lemma 1 und Lemma 2.

# 5 Implementierung

Im Folgenden findet sich eine mögliche Implementierung des Huffman-Algorithmus in der funktionalen Programmiersprache Haskell.

## 5.1 Design

Der Code ist in die folgenden Module unterteilt:

- Types: In diesem Modul sind die Datentypen festgelegt.
- CodeCreation: Dieses Modul liefert das Obermodul zur Erstellung von einer Tabelle von Codewörtern sowie einem Baum zu der Tabelle. Dieses Modul besitzt wiederum mehrere Untermodule:
  - MessageAnalyzation: Die Messages, die kodiert werden sollen, müssen zunächst analysiert werden. Dieses Modul bringt die entsprechende Funktionalität.
  - TreeCreation: Hier werden Funktionen zur Erstellung der Bäume implementiert.
  - CodetableCreation: Hier wird aus einem Baum eine Tabelle von Codewörtern mit den entsprechenden Zeichen erstellt.
- Coding: Hier findet das eigentlich Kodieren und Dekodieren einer Nachricht statt.

## 5.2 Types

Um den Huffman-Code zu implementieren werden vier neue Datentypen eingeführt um einerseits den Baum, andererseits die Codewörter zu repräsentieren. Ein Baum besteht aus Blättern und Knoten, ein einzelnes Bit wird durch $L$ oder $R$ repräsentiert, ein Codewort ist eine Menge von Bits. Hierbei kann das $L$ als 0 und das $R$ als 1 interpretiert werden. Damit sieht die Implementierung der Typen wie folgt aus:

```
module Types ( Tree(Leaf, Node),
               Bit(L,R),
               CodeWord,
               CodeTable ) where
```

```
data Tree = Leaf Char Int |
           Node Int Tree Tree deriving (Show)

data Bit = L | R deriving (Eq, Show)

type CodeWord = [Bit]

type CodeTable = [(Char, CodeWord)]
```

## 5.3 MessageAnalyzation

Das Modul MessageAnalyzation stellt die Funktion createTupleList bereit, welche eine Nachricht in eine Liste von Tupeln der Form (Char c, Int n) zerlegt. Hierbei enthält die Liste für jeden Buchstaben c aus der Originalnachricht die Häufigkeit n des Buchstabens. Des Weiteren sind die Tupel aufsteigend nach n geordnet. Diese Ordnung wird erreicht, indem zunächst die Häufigkeiten der einzelnen Zeichen berechnet werden und anschließend die Liste sortiert wird. Implementiert wird dies in einer generischen Merge Sort-Funktion.

```
module MessageAnalyzation ( createTupleList ) where

mergeSort :: ([a] -> [a] -> [a]) -> [a] -> [a]

mergeSort mergeFunc xs
     | length xs < 2 = xs
     | otherwise
         = mergeFunc (mergeSort mergeFunc first)
                (mergeSort mergeFunc second)
           where
            first = take half xs
            second = drop half xs
            half = (length xs) `div` 2

mergeCount :: [(Char, Int)] -> [(Char, Int)] -> [(Char, Int)]
mergeCount xs [] = xs
mergeCount [] ys = ys
mergeCount ((c1, i1):xs) ((c2,i2):ys)
     | (c1 == c2)     = (c1, i1 + i2) : mergeCount xs ys
     | (c1 < c2)       = (c1, i1) : mergeCount xs ((c2,i2):ys)
     | otherwise       = (c2, i2) : mergeCount ((c1, i1):xs) ys


mergeOrder :: [(Char, Int)] -> [(Char, Int)] -> [(Char, Int)]
mergeOrder xs [] = xs
mergeOrder [] ys = ys
mergeOrder ((c1, i1):xs) ((c2, i2):ys)
     | ((i1 < i2) || ((i1 == i2) && (c1 < c2)))
               = (c1, i1) : mergeOrder xs ((c2,i2):ys)
     | otherwise
```

```
              = (c2, i2) : mergeOrder ((c1,i1):xs) ys

createTupleList :: [Char] -> [(Char, Int)]

createTupleList [] = []
createTupleList xs = mergeSort mergeOrder
                     (mergeSort mergeCount (map start xs))
                  where start s = (s, 1)
```

## 5.4 TreeCreation

Innerhalb dieses Moduls findet der eigentliche Aufbau eines Baumes statt.

```
module TreeCreation where

import Types

convertToLeaves :: [(Char, Int)] -> [Tree]
convertToLeaves = map (uncurry Leaf)

createTree :: [(Char, Int)] -> Tree
createTree = buildTree . convertToLeaves

buildTree :: [Tree] -> Tree
buildTree [ singleNode ] = singleNode
buildTree xs = buildTree (performMerge xs)

performMerge :: [Tree] -> [Tree]
performMerge (n1:n2:nodes) = reInsertNode (combineNodes n1 n2) nodes


combineNodes :: Tree -> Tree -> Tree
combineNodes n1 n2 = Node ((nodeValue n1) + (nodeValue n2)) n1 n2

nodeValue :: Tree -> Int
nodeValue (Leaf _ n) = n
nodeValue (Node n _ _) = n

reInsertNode :: Tree -> [Tree] -> [Tree]
reInsertNode node [] = [node]
reInsertNode node (first:nodes)
| (nodeValue node <= nodeValue first) = node:first:nodes
| otherwise = first : reInsertNode node nodes
```

## 5.5 CodetableCreation

Dieses Modul erzeugt aus einem Baum eine Codetabelle für jeden Buchstaben. Eine Codetabelle ist hierbei eine Liste von Tupeln der Form (Char, Codewort).

```
module CodetableCreation where
```

```
import Types

createCodetable :: Tree -> CodeTable
createCodetable t = addCodePath [] t

addCodePath :: CodeWord -> Tree -> CodeTable
addCodePath prevCodePath (Leaf c v)
             = [(c, prevCodePath)]
addCodePath prevCodePath (Node v n1 n2)
             = (addCodePath (prevCodePath++[L]) n1)
                   ++ (addCodePath (prevCodePath++[R]) n2)
```

## 5.6 CodeCreation

Dieses Modul bietet die Funktionalität, einen Baum zu erzeugen, und ihn in eine Codetabelle umzuwandeln. Es verwendet hierfür die Analysefunktion createTupleList aus dem Modul MessageAnalyzation sowie die Baumerstellungsfunktion buildTree aus dem Modul TreeCreation. Die Codetabelle ist im Modul CodetableCreation implementiert

```
module CodeCreation where

import Types
import TreeCreation
import CodetableCreation
import MessageAnalyzation

createCode :: [Char] -> CodeTable
createCode = createCodetable . createHuffmanTree

createHuffmanTree :: [Char] -> Tree
createHuffmanTree = createTree . createTupleList
```

## 5.7 Coding

Zweck dieses Moduls ist das Kodieren und Dekodieren von Nachrichten. Die Funktion encodeMessage kodiert hierbei eine Nachricht anhand einer Codetabelle, die Funktion decodeMessage dekodiert eine Nachricht anhand eines Huffman-Baumes. lookupTable wandelt ein einzelnes Zeichen in ein Codewort anhand einer Codetabelle um.

```
module Coding ( encodeMessage, decodeMessage ) where

import Types ( Tree(Leaf, Node),
               Bit(L,R),
               CodeWord,
               CodeTable )
```

```
encodeMessage :: CodeTable -> [Char] -> CodeWord

encodeMessage tbl = concat . map ( lookupTable tbl )

lookupTable :: CodeTable -> Char -> CodeWord

lookupTable [] c = error "Character not in Codetable"
lookupTable ((ch,n):tb) c
      | ch == c        = n
      | otherwise      = lookupTable tb c



decodeMessage :: Tree -> CodeWord -> [Char]
decodeMessage tr
      = decodeByte tr
        where
         decodeByte (Node n t1 t2) (L:rest)
            = decodeByte t1 rest
         decodeByte (Node n t1 t2) (R:rest)
            = decodeByte t2 rest
         decodeByte (Leaf c n) rest
            = c : decodeByte tr rest
         decodeByte t [] = []
```

## 5.8 Main

Diese Module können nun von einer Main-Funktion geeignet verwendet und die Funktionen aufgerufen werden:

```
module Main where

import CodeCreation
import Coding

------ Here are some Examples

echo :: String -> IO()
echo = putStr . (++ "\n")

target = "Perlen2"
targetCode = createCode target
targetTree = createHuffmanTree target

main = do
 let i1 = "Beispiel 1 : Codetabelle für das Wort " ++ target
 let i2 = targetCode
 let i3 = "Beispiel 2 : Baum für das Wort " ++ target
 let i4 = targetTree
```

```
let i5 = "Beispiel 3 : Enkodieren des Wortes"
let i6 = encodeMessage targetCode target
let i7 = "Beispiel 4 : Dekodieren des kodierten Wortes"
let i8 = decodeMessage targetTree (encodeMessage targetCode target)
echo i1
echo (show i2)
echo i3
echo (show i4)
echo i5
echo (show i6)
echo i7
echo (show i8)
```

# Literatur


[1] Simon Thompson, *Haskell - The Craft of Functional Programming*, Addison-Wesley, Bonn, 1999

[2] Cormen, Leiserson, Rivest, Stein, *Algorithmen - Eine Einführung*, Oldenbourg, München, 2001

[3] Richard Bird, Philip Wadler, *Introduction to Functional Programming*, Prentice Hall, 1998