Proseminar „Computer Science Unplugged“
Algorithmen zur Textkompression

Harald Zauner

12. November 2004

Zusammenfassung

Trotz der rasch zunehmenden Speicherkapazität heutiger Speichermedien besteht nach wie vor das Bedürfnis, Daten so effizient wie möglich zu speichern. Durch Kodierung kann der Speicherplatzbedarf von Daten auf Kosten einer leicht erhöhten Zugriffszeit deutlich reduziert werden.

Im Folgenden wird nach einem spielerischen Einstieg der Huffman- und LZW-Algorithmus vorgestellt werden.

1 Einführung

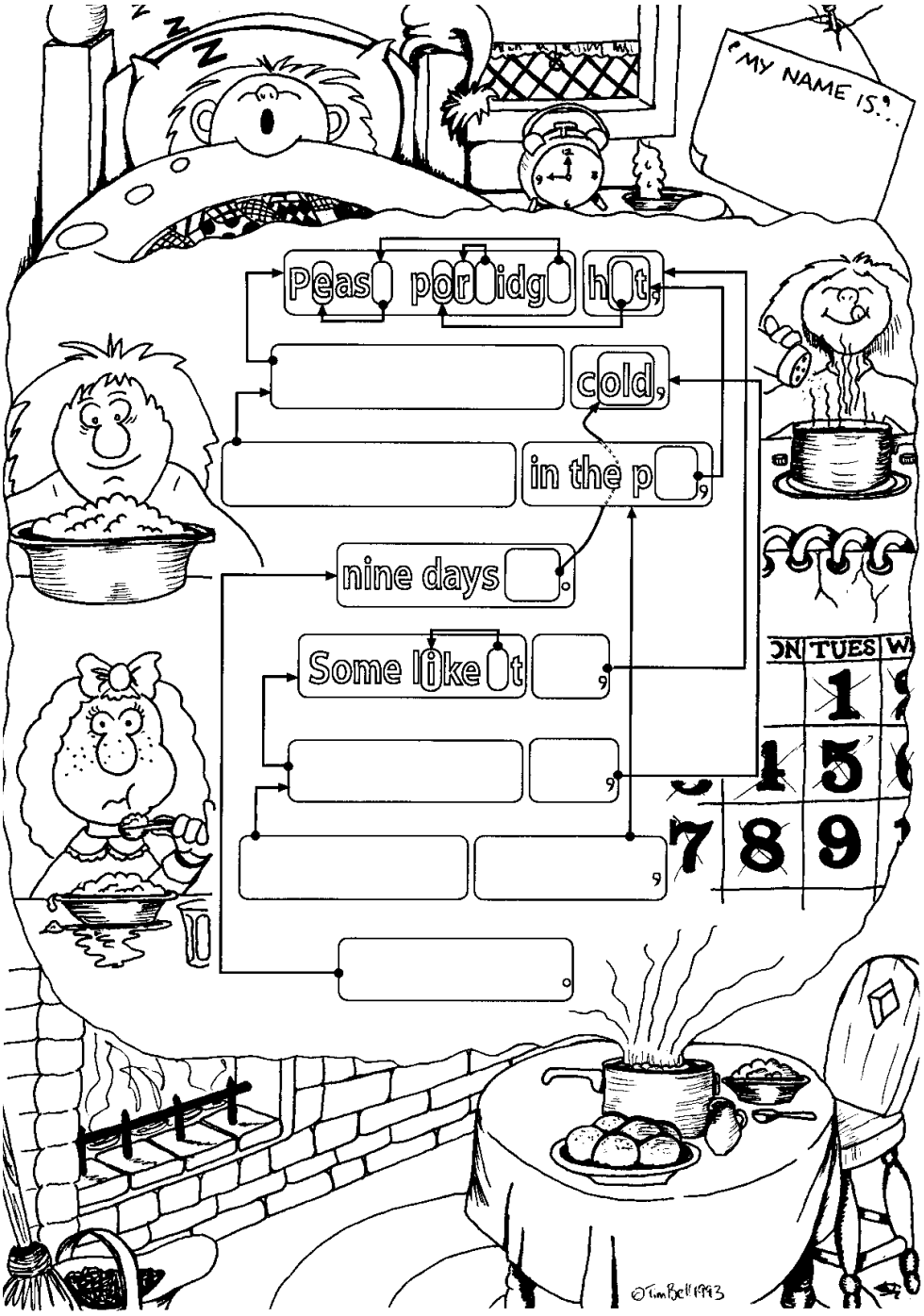
Die Speicherkapazität von Computern wächst mit erstaunlicher Geschwindigkeit, innerhalb der letzten 25 Jahre etwa um den Faktor eine Million.
Doch mit wachsender Kapazität ergeben sich immer wieder neue Möglichkeiten. So möchte man z.B. auf einem Computer, der ein Buch abspeichern kann, lieber eine ganze Bibliothek unterbringen. Oder anstatt eines Bildes lieber ein ganzes Fotoalbum oder einen Film.
Jeder, der einen Computer besitzt, wird bereits festgestellt haben, dass nach einer Variation von Parkinson's Gesetz („Work expands to fill the time available for its completion“) sich die Daten soweit ausbreiten bis der verfügbare Speicherplatz aufgebraucht ist.

Um dieser Entwicklung entgegenzuwirken, bedient man sich der *Kompression*. Dabei wird die Repräsentation der Daten derart geändert, dass weniger Speicherplatz verbraucht wird. Der Prozeß des Komprimierens und Dekomprimierens wird dabei im Hintergrund ausgeführt, ohne dass der Benutzer davon Kenntnis haben muss.
Einerseits braucht der Vorgang des Komprimierens bzw. Dekomprimierens selbst natürlich eine gewisse Zeit, was sich in der leichten Verlangsamung der Schreib- bzw. Lesevorgänge bemerkbar machen könnte.
Andererseits kann es aber auch zu einer Beschleunigung kommen, da weniger Daten von der Festplatte gelesen werden müssen und dieser Effekt die nötige Zeit für das Komprimieren kompensiert.

Im Laufe der Zeit wurden viele Kompressionsverfahren entwickelt. Dabei zeigt sich aber, dass das Verweisen auf bereits früher aufgetretene Textfragmente eine relativ gebräuchliche Methode ist, welche z.B. beim LZW-Algorithmus angewandt wird.

Ein alternativer Ansatz besteht darin, dass man öfter auftretende Zeichen mit einer kürzeren Codierung versieht als weniger oft auftretende Zeichen. Auf dieser Grundidee basieren beispielsweise der Morse-Code und auch der Huffman-Algorithmus.

Um eine anschauliche Vorstellung des LZW-Algorithmus zu bekommen, soll nun das Arbeitsblatt aus [1] bearbeitet werden.



Instructions: *Fill in each blank box by following the arrow attached to the box and copying the letters in the box that it points to.*

Abbildung 1: Spielerischer Einstieg aus [1]

2 Der LZW-Algorithmus

Der LZW-Algorithmus ist eine Weiterentwicklung des LZ78-Algorithmus, welcher 1984 von Terry Welch veröffentlicht wurde. Der LZ77- bzw. LZ78-Algorithmus geht auf Abraham Lempel und Jacob Ziv zurück.
LZW findet beispielsweise Verwendung im Grafikformat GIF, in diversen UNIX-Tools (*compress*, *gzip*), im Postscript Level 2 Standard (optional), usw.

2.1 Arbeitsweise

Der LZW-Algorithmus unterscheidet sich von seinen Vorgängern dadurch, dass das initiale Codebuch alle Zeichen des zu codierenden Alphabets enthält. Auf diese Weise entfällt das explizite Abspeichern von Zeigern auf den Codebucheintrag und auf das nachfolgende Zeichen.

Die **Komprimierung** geschieht dabei wie folgt:

Zu Beginn der Kompression sei w der leere String.

Schritt 1: Lese das nächste Symbol x aus dem Eingabestring ein und gehe zu Schritt 2.
Falls der Eingabestring bereits komplett eingelesen ist, gib w aus.

Schritt 2:

Fall 1: wx befindet sich noch nicht im Codebuch
Ausgabe der Codierung von w
Füge dem Codebuch einen neuen Eintrag für wx hinzu
Ersetze w durch x und gehe zu Schritt 1

Fall 2: wx ist bereits im Codebuch enthalten
Ersetze w durch wx und gehe zu Schritt 1

Die **Dekomprimierung** funktioniert wie folgt:

Zu Beginn der Ausführung sei w das erste aus dem Eingabestring eingelesene Symbol x, dessen Decodierung mit Hilfe des initialen Codebuchs erfolgt und anschließend ausgegeben wird.

Schritt 1: Lese das nächste Symbol x aus dem Eingabestring ein und gehe zu Schritt 2.
Falls der Eingabestring bereits vollständig eingelesen ist, ist man fertig.

Schritt 2: Decodiere x mit Hilfe des Codebuchs, und gib die Dekodierung $C(x)$ aus.
Weiter mit Schritt 3.

Schritt 3: Füge dem Codebuch einen neuen Eintrag hinzu, welcher aus der Konkatenation von w und dem ersten Zeichen von $C(x)$ besteht. Gehe zu Schritt 4.

Schritt 4: Setze $w = C(x)$ und springe zu Schritt 1.

Anmerkung: Bei dieser Version des Dekodierungsalgorithmus wird davon ausgegangen, dass das eingelese Symbol x bei Schritt 2 immer in der Codetabelle gefunden wird. Jedoch kommt es bei manchen Wörtern auch vor, dass der Eintrag dafür noch nicht vorhanden ist.
In diesem Fall wird bei Schritt 2 $C(x)$ auf w gesetzt, die Konkatenation von $C(x)$ und dem ersten Zeichen von $C(x)$ ausgegeben und die Ausführung wie gehabt fortgesetzt.

2.2 Beispiel

Wir komprimieren das Wort „abrakadabra“, das initiale Codebuch C enthalte alle ASCII-Zeichen, also insbesondere $C(\text{a}) = 97$, $C(\text{b}) = 98$, $C(\text{d}) = 100$, $C(\text{k}) = 107$ und $C(\text{r}) = 114$.

Schritt	w_{alt}	x	$wx \in C$	w_{neu}	Ausgabe
1	“”	a	ja	a	
2	a	b	nein $\Rightarrow C(\text{ab}) = 256$	b	97
3	b	r	nein $\Rightarrow C(\text{br}) = 257$	r	98
4	r	a	nein $\Rightarrow C(\text{ra}) = 258$	a	114
5	a	k	nein $\Rightarrow C(\text{ak}) = 259$	k	97
6	k	a	nein $\Rightarrow C(\text{ka}) = 260$	a	107
7	a	d	nein $\Rightarrow C(\text{ad}) = 261$	d	97
8	d	a	nein $\Rightarrow C(\text{da}) = 262$	a	100
9	a	b	ja	ab	
10	ab	r	nein $\Rightarrow C(\text{abr}) = 263$	r	256
11	r	a	ja	ra	258

Nun führen wird die Dekomprimierung von “97 98 114 97 107 97 100 256 258“ durch:

Wir verwenden dabei wieder das initiale Codebuch mit allen ASCII-Zeichen, also insbesondere $C(97) = \text{a}$, $C(98) = \text{b}$, $C(100) = \text{d}$, $C(107) = \text{k}$ und $C(114) = \text{r}$.

Schritt	w_{alt}	x	Codebuch	$w_{neu} = C(x) =$ Ausgabe
1	*undef*	97	kein neuer Eintrag	a
2	a	98	$C(256) = \text{ab}$	b
3	b	114	$C(257) = \text{br}$	r
4	r	97	$C(258) = \text{ra}$	a
5	a	107	$C(259) = \text{ak}$	k
6	k	97	$C(260) = \text{ka}$	a
7	a	100	$C(261) = \text{ad}$	d
8	d	256	$C(262) = \text{da}$	ab
9	ab	258	$C(263) = \text{abr}$	ra

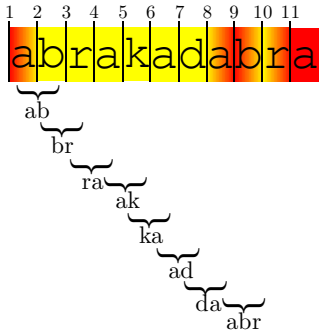


Abbildung 2: Veranschaulichung der LZW-Komprimierung

2.3 Java-Implementierung

LZWEncode.java

```
import java.util.Map;
import java.util.TreeMap;

public class LZWEncode {

    private String input;
    private Map codetable;
    private StringBuffer result;

    public LZWEncode(String input) {
        result = new StringBuffer();
        codetable = new TreeMap();
        this.input = input;
        initCodetable();
        encode();
        System.out.println(result.toString());
    }

    private void initCodetable() {
        for (int i = 0; i < 256; i++) {
            codetable.put(Character.toString((char) i),
                    new Integer(codetable.size()));
        }
    }

    private void encode() {
        String w = "";
        for (int i = 0; i < input.length(); i++) {
            char x = input.charAt(i);
            String wx = w + x;
            if (codetable.containsKey(wx)) {
                w = wx;
            } else {
                codetable.put(wx, new Integer(codetable.size()));
                result.append(codetable.get(w));
                w = Character.toString(x);
            }
        }
        result.append(codetable.get(w));
    }

    public static void main(String[] args) {
        new LZWEncode("abrakadabra");
    }
}
```

LZWDecode.java

```
import java.util.Map;
import java.util.TreeMap;

public class LZWDecode {

    private Map codetable;
    private int[] input;
    private StringBuffer result;

    public LZWDecode(int[] input) {
        this.input = input;
        codetable = new TreeMap();
        result = new StringBuffer();
        initCodetable();
        decode();
        System.out.println(result);
    }

    private void initCodetable() {
        for (int i = 0; i < 256; i++) {
            codetable.put(new Integer(codetable.size()),
                    Character.toString((char) i));
        }
    }

    private void decode() {
        String w = (String) codetable.get(new Integer(input[0]));
        result.append(w);
        for (int i = 1; i < input.length; i++) {
            String Cx = (String) codetable.get(new Integer(input[i]));
            if (Cx != null) {
                result.append(Cx);
            } else {
                Cx = w;
                result.append(Cx + Cx.charAt(0));
            }
            codetable.put(new Integer(codetable.size()), w
                    + Character.toString(Cx.charAt(0)));
            w = Cx;
        }
    }
    public static void main(String[] args) {
        new LZWDecode(new int[] { 97, 98, 114, 97, 107, 97, 100,
                256, 258 });
    }
}
```

3 Der Huffman-Algorithmus

David Huffman entwickelte im Jahre 1952 ein heute noch sehr beliebtes Verfahren zur verlustfreien Kompression von Daten, welches z.B. im UNIX-Tool *compact* und in Faxgeräten als Teil des CCITT-Standards Anwendung findet. Weiterhin wird es oft in Kombination mit LZW und anderen Kompressionsverfahren z.B. beim Grafikformat JPEG eingesetzt.

Ähnlich wie beim Morse-Code codiert man Zeichen, welche häufiger im Text auftreten mit einem kürzerzen Code als weniger oft auftretende Zeichen, vgl. untenstehende Tabelle.

Symbol	A	B	C	D	E	F	G	H	I	J	K	L	M
Code	.-	-...	-.-.	-..	.	..-.	- -.	....	..	.- - -	-.-	.-..	- -
Symbol	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Code	-.	- - -	.- -.	- -.-	.-.	...	-	..-	...-	.- -	-..-	-.- -	- -..

An den Buchstaben A und E sieht man diesen Effekt sehr deutlich.

3.1 Arbeitsweise

Mit Hilfe des Huffman-Algorithmus ist es möglich, einen bezüglich der Codewortlänge optimalen Code zu erzeugen.

Sei $s_1, s_2, \ldots, s_n$ das Alphabet, welches von dem zu kodierenden Textstück benutzt wird. p_i bezeichne die Wahrscheinlichkeit, dass s_i an einer zufälligen Position auftaucht. Jedes Symbol s_i wird durch einen binären Code l_i ersetzt, so dass sich die durchschnittliche Länge eines codierten Symbols ergibt zu

$$L = \sum_{i=1}^{n} p_i l_i$$

Um ein Minimum von L zu erreichen, darf man annehmen, dass die s_i schon nach ihrer Wahrscheinlichkeit sortiert sind, und dann gilt: $p_1 \geq p_2 \geq \ldots \geq p_n$. L wird genau dann minimal, wenn man nun $l_1 \leq l_2 \leq \ldots \leq l_n$ wählt, was durch den Huffman-Algorithmus gewährleistet ist.

Beschreibung des Algorithmus:

Gegeben sei eine initiale Liste bzw. ein Array von Huffman-Blättern mit mindestens 2 Elementen (sonst fertig). Ein Huffman-Blatt ist ein Paar, welches aus einem Symbol s_i und der dazugehörigen Häufigkeit p_i oder der absoluten Symbolanzahl (schneller!) besteht.

Schritt 1: Sortiere die Liste von Huffman-Bäumen bzgl. der Häufigkeiten bzw. der absoluten Symbolanzahl. Für die Implementierung geschieht dies mit aufsteigender, für den nachfolgenden Beweis mit absteigender Häufigkeit bzw. Symbolanzahl.

Schritt 2: Ersetze die beiden Huffman-Bäume mit dem niedrigsten Wert durch einen neuen Knoten, welcher jene als linken und rechten Teilbaum erhält. Die Wertigkeit des neuen Knotens ergibt sich aus der Summe der Häufigkeiten der Teilbäume.

Schritt 3: Wiederhole Schritt 1 und 2 solange bis die Liste nur noch aus einem einzigen Baum besteht.

Aus dem fertigen Huffman-Baum erhält man nun die Codierung jedes einzelnen Symbols, indem man den Pfad von der Wurzel bis zu dem entsprechenden Blatt betrachtet. Für jede Verzweigung nach links verlängert sich die Codierung um eine „0“, für jede Verzweigung nach rechts um eine „1“.

3.2 Beweis der Optimalität der Huffman-Codierung

Zu zeigen: L ist minimal

Beweis: Beweis durch vollständige Induktion über die Anzahl n der Symbole. Im Folgenden werde mit $c(s_i)$ die Huffmancodierung des Symbols s_i bezeichnet, und es gilt $l_i = |c(s_i)|$.

Induktionsanfang
Sei $n = 2$. Für die Codierung $c(s_1)$ erhält man entweder "0" oder "1" und somit $c(s_2) = 1 - c(s_1)$. $\Rightarrow L = \sum_{i=1}^{2} p_i l_i = p_1 l_1 + p_2 l_2 = p_1|c(s_1)| + p_2|c(s_2)| = p_1 + p_2 = 1$. Dieser Wert ist optimal, denn 2 Symbole können nicht besser als durch Codelänge 1 codiert werden.

Induktionsschritt
Induktionsannahme: Die Huffmancodierung sei bereits optimal für alle Textstücke mit $n-1$ Symbolen ($n > 2$).

Bezeichne S^* das Alphabet welches sich durch Reduktion gemäß dem Huffmanalgorithmus ergibt, c^* die entsprechende Codierungsfunktion und s^* das „Symbol", welches die beiden Symbole mit der kleinsten rel. Häufigkeit s_{n-1} und s_n ersetzt. Weiterhin bezeichne p^* die rel. Häufigkeit von s^*, also $p^* = p_{n-1} + p_n$. Für das reduzierte Alphabet S^* gilt $|S^*| = n-1$, wodurch die Induktionsannahme erfüllt wird. Die durchschnittliche Codelänge von S^* sei L^* mit $L^* = \sum_{i=1}^{n-2} p_i \cdot |c(s_i)| + p^* \cdot |c^*(s^*)|$.

Es folgt:

$$\begin{aligned}
L &= \sum_{i=1}^{n} p_i l_i = \sum_{i=1}^{n} p_i |c(s_i)| = \sum_{i=1}^{n-2} p_i |c(s_i)| + p_{n-1}|c(s_{n-1})| + p_n |c(s_n)| \\
&= \sum_{i=1}^{n-2} p_i |c(s_i)| + p_{n-1}|c^*(s^*) \circ \text{“0“}| + p_n |c^*(s^*) \circ \text{“1“}| \\
&= \sum_{i=1}^{n-2} p_i |c(s_i)| + (p_{n-1} + p_n) \cdot (|c^*(s^*)| + 1) \\
&= \sum_{i=1}^{n-2} p_i |c(s_i)| + p^* |c^*(s^*)| + p^* \\
&= L^* + p^*
\end{aligned}$$

Nach der Induktionsannahme ist L^* minimal. Da p^* durch den Huffman-Algorithmus ebenfalls minimal gewählt wird, ist auch L minimal.
Dies zeigt, dass die Huffman-Codierung optimal ist. □

3.3 Beispiel

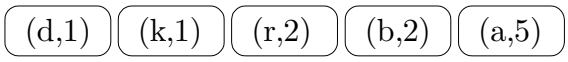


Abbildung 3: Ausgangszustand: Fünf Knoten, die bereits nach Anzahl sortiert sind

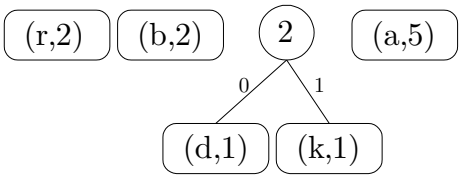


Abbildung 4: 1. Iteration

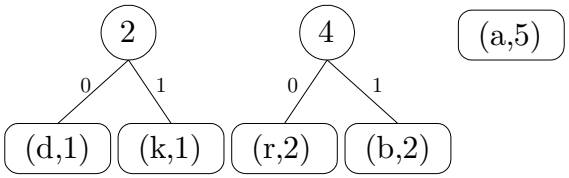


Abbildung 5: 2. Iteration

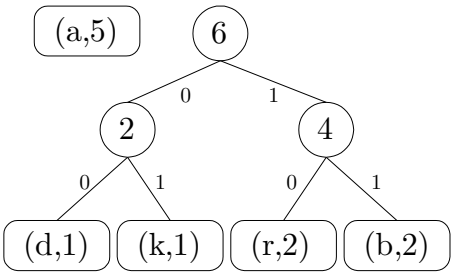


Abbildung 6: 3. Iteration

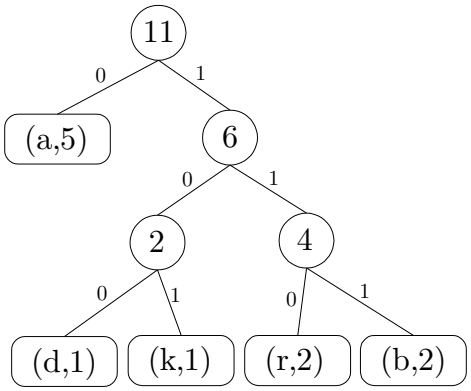


Abbildung 7: Huffman-Baum von „abrakadabra“ (4. Iteration)

Aus dem Huffman-Baum ergibt sich folgende Codetabelle:

a	b	d	k	r
0	111	100	101	110

Bei der Kompression von ASCII-Text wird in realen Anwendungen nicht für jeden Text ein neuer Huffman-Baum erzeugt, sondern mit einer fest vorgegebenen Codierungstabelle gearbeitet, welche sich aus den Häufigkeiten der einzelnen Zeichen ergibt.

Für die englische Sprache ergibt sich beispielsweise folgende Verteilung:

Symbol	A	B	C	D	E	F	G	H	I	J	K	L	M
Wahrscheinlichkeit	.065	.013	.022	.032	.104	.021	.015	.047	.058	.001	.005	.032	.032
Symbol	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Wahrscheinlichkeit	.058	.064	.015	.001	.049	.056	.081	.023	.008	.018	.001	.017	.001

3.4 Ocaml-Implementierung

Huffman.ml (teilweise in Anlehnung an [2])

```
(* Wandelt eine Liste von chars in einen String um *)
let string_of_list l = List.fold_left (^) ""
    (List.map (fun c -> String.make 1 c) l)

(* Wandelt einen String in eine Liste von einzelnen Symbolen (chars) um *)
let list_of_string s = Stream.npeek (String.length s) (Stream.of_string s)

(* Rek. Definition eines Huffman-Baums *)
type 'a hufftree = Atom of char * 'a | Node of 'a * 'a hufftree * 'a hufftree
```

```
(* Funktion, welche die Gesamtanzahl von Symbolen in einem
   Huffman-Baum zurueckgibt *)
let sum ht = match ht with
  Atom(a,cnt) -> cnt
| Node(cnt,l,r) -> cnt

(* Fuegt einen Huffman-Baum in eine Liste von Huffman-Baeumen ein *)
let rec insert elem lst = match lst with
  [] -> [elem]
| x::l -> if (sum elem) < (sum x) then elem::x::l else
  x::insert elem l

(* Sortiert eine Liste von Huffman-Baeumen *)
let rec sort_htlist htl = match htl with
  [] -> []
| x::l -> insert x (sort_htlist l)

(* Erzeugt aus einem String s eine Liste aus Paaren, die das entsprechende
   Symbol mit der zugehoerigen Haeufigkeit enthalten *)
let generate_charcntlist s =
  let rec add_to_charcntlist c charcntlist = match charcntlist with
    [] -> [(c,1)]
  | (x,cnt)::t when x = c -> (c,cnt+1)::t
  | (x,cnt)::t -> (x,cnt)::(add_to_charcntlist c t) in
  let rec generate_charcntlist_aux c charcntlist = match c with
    [] -> charcntlist
  | h::t -> generate_charcntlist_aux t (add_to_charcntlist h charcntlist)
  in generate_charcntlist_aux (list_of_string s) []

(* Wandelt die "Haeufigkeitsliste" in eine Liste aus Huffman-Baeumen um *)
let rec htlist_of_charcntlist charcntlist = match charcntlist with
  [] -> []
| (a,cnt)::t -> Atom(a,cnt)::(htlist_of_charcntlist t)

(* Erzeugt einen vollstaendigen Huffman-Baum aus einer Liste von
   Huffman-Baeumen *)
let rec generate_huffman_tree_of_htlist htlist =
  (* Hilfsfunktion, welche die ersten beiden Elemente der htlist
     zu einem neuen Huffman-Baum vereinigt *)
  let combine_min = match htlist with
    [] -> []
  | h::[] -> [h]
  | h1::h2::t -> insert (Node(sum h1 + sum h2,h1,h2)) t in
  (* Falls die Liste noch mindestens 2 Elemente enthaelt, erfolgt
     ein rek. Aufruf, ansonsten wird der vollstaendige Huffman-Baum
     zurueckgegeben *)
  if (List.length htlist > 1) then
    let htlist = combine_min in generate_huffman_tree_of_htlist htlist
  else List.hd htlist

(* Erzeugt eine Codierungstabelle als Assoziationsliste, indem fuer
   jedes Symbol der Pfad berechnet und akkumuliert wird *)
let generate_codetable ht =
  let rec generate_codetable_aux ht result = match ht with
    Atom(a,cnt) -> [(a,result)]
  | Node(cnt,l,r) -> (generate_codetable_aux l (result@['0'])) @
      (generate_codetable_aux r (result@['1']))
  in generate_codetable_aux ht []
```

```
(* Codiert eine Liste von Symbolen mit einem zugehoerigen Huffman-Baum *)
let huffman_encode_charlist charlist ht =
  let codetable = generate_codetable ht in
  List.flatten (List.map (fun x -> List.assoc x codetable) charlist)

(* Codiert einen String mit Hilfe eines Huffman-Baums *)
let huffman_encode s ht =
  huffman_encode_charlist (list_of_string s) ht

(* Erzeugt einen Huffmann-Baum aus einem String *)
let generate_huffman_tree s =
  generate_huffman_tree_of_htlist
    (sort_htlist (htlist_of_charcntlist (generate_charcntlist s)))

(* Gibt die Codierung des Huffman-Codierung des Strings zurueck *)
let huffman s = string_of_list (huffman_encode s (generate_huffman_tree s))
```

Literatur

[1] Tim Bell, Ian H. Witten, and Mike Fellows. *Computer Science Unplugged*. 1998.

[2] Prof. Dr. Alois Knoll. *Einführung in die Informatik I WS 2002/03*. `http://www6.in.tum.de/info1/`.