Technische Universität München
Fakultät für Informatik
M. Bauer, D. Golias, S. Riesner,
Dr. M. Schneider

WS 2002/2003
Lösungsvorschläge zu Blatt 5
14. November 2002

Übungen zu Einführung in die Informatik I

Aufgabe 17 Zählen von Zeichen-Häufigkeiten (Lösungsvorschlag)

a) In der Funktion *add_char_to_charcntlist* wird in der Liste `charcntlist` rekursiv nach einer Häufigkeitszählung für das Zeichen c gesucht. Im Erfolgsfall wird der cnt des Zeichens erhöht, andernfalls das Zeichen neu (mit Zählung 1) in die Liste aufgenommen.

```
# let rec add_char_to_charcntlist c charcntlist
      = match charcntlist with
        | []                        -> [(c,1)]
        | (a,cnt)::tail when a = c  -> [(c,cnt+1)]@tail
        | (a,cnt)::tail     -> [(a,cnt  )]
                                   @
                                   (add_char_to_charcntlist c tail)
;;
val add_char_to_charcntlist : 'a -> ('a * int) list -> ('a * int) list
                            = <fun>
```

Definiert wird eine (eingebettete) Funktion *generate_charcntlist_rec*, welche eine Liste aus Einzelzeichen als Eingabe erwartet, und diese rekursiv Zeichen für Zeichen via *add_char_to_charcntlist* der ebenfalls mitgegebenen `charcntlist` einordnet.

```
# let rec generate_charcntlist_rec c charcntlist = match c with
        | []       -> charcntlist
        | hd::tail -> generate_charcntlist_rec
                        tail (add_char_to_charcntlist hd charcntlist)
;;
val generate_charcntlist_rec : 'a list -> ('a * int) list ->
                                 ('a * int) list = <fun>
```

Diese Funktion wird von der *generate_charcntlist* mit dem via *list_of_string zerstückelten* Eingabestring und einer (initial) leeren Ergebnisliste aufgerufen.

```
# let generate_charcntlist str
        = generate_charcntlist_rec (list_of_string str) []
;;
val generate_charcntlist : string -> (char * int) list = <fun>

# generate_charcntlist "abbaccabba" ;;
- : (char * int) list = [('a', 4); ('b', 4); ('c', 2)]
```

Aufgabe 18 Generierung von Huffmanbäumen (Lösungsvorschlag)

a) In der dargestellten Definition eines `hufftree` wird (entgegen z.B. der Zentralübung) die Summe der von einem Knoten ausgehenden Zeichen-Häufigkeiten nicht im Knoten gespeichert. Ineffizienterweise muß diese deshalb, wann immer nötig, von einer Funktion (*treesum*) bestimmt werden:

```
# let rec treesum ht = match ht with
        | Atom(a,cnt)  -> cnt
        | Node(l,r)  -> (treesum l) + (treesum r)
;;
val treesum : hufftree -> int = <fun>
```

b) Es wird eine neue Liste erstellt, indem rekursiv durch die bestehende gegangen wird, und die Paare in `hufftree`-Atome *umgewandelt* werden.

```
# let rec hufftreelist_of_charcntlist charcntlist
        = match charcntlist with
        | []  -> []
        | (a,cnt)::tail -> [Atom(a,cnt)]
                           @
                           (hufftreelist_of_charcntlist tail)
;;
val hufftreelist_of_charcntlist : (char * int) list ->
                                  hufftree list = <fun>
```

c) Liegen die Elemente der Häufigkeit (kleinstes Element zuerst vor), so besteht die Lösung naheliegenderweise darin, die beiden ersten Elemente der Liste zusammenzufassen und den übrigen *tail* daran zu konkatenieren:

```
# let combine_leading lst = match lst with
        | []                    ->      []
        | hd::[]                ->      [hd]
        | hd1::hd2::tail        ->      Node(hd1,hd2)::tail
;;
val combine_leading : hufftree list -> hufftree list = <fun>
```

Für das *combine_min* muß daher zunächst die Eingabe-Liste sortiert werden:

```
# let combine_min lst = combine_leading (sort_hufftreelist lst);;
val combine_min : hufftree list -> hufftree list = <fun>
```

d) Der Huffman-Baum ist dann erstellt, wenn nach einer rekursiven Verkürzung einer Huffman-Atom-Liste via *combine_min* nur mehr ein Element in der Liste (nämlich der Code-Baum) verbleibt:

```
# let rec generate_hufftree htl = match htl with
        | []     -> failwith "Empty List"
        | hd::[] -> hd
        | _      -> generate_hufftree (combine_min htl)
;;
val generate_hufftree : hufftree list -> hufftree = <fun>
```

Aufgabe 19 Huffman- Encodierung von Texten (Lösungsvorschlag)

a) Zunächst wird eine rekursive Funktion definiert, welche in einem Huffman-Baum nach einem bestimmten Zeichen sucht, und den Weg *mitschreibt*. Für den Treffer wird der Weg als Ergebnis geliefert, für alle anderen Fälle eine leere Liste. Geht man davon aus, dass im Huffman-Baum nur ein Atom für ein Zeichen existiert, so liefert die Konkatenation aller Weg-Ergebnisse den gesuchten Weg für dieses Zeichen:

```
# let rec huffman_encode_rec zeichen baum ergebnis = match baum with
        | Atom(a, cnt) when a = zeichen
                    -> ergebnis
        | Node(l,r) -> (huffman_encode_rec zeichen l (ergebnis@['O']))
                       @
                       (huffman_encode_rec zeichen r (ergebnis@['L']))
        | _         -> []

;;
val huffman_encode_rec : char -> hufftree -> char list ->
                         char list = <fun>
```

Diese Funktion wird nun für alle Zeichen des Eingabestrings aufgerufen. Hierzu wird wiederum eine rekursive, eingebettete, Funktion geschrieben, welche auf einer Liste von Zeichen arbeitet, und die Zeichen-Encodierungsfunktion für die einzelnen Zeichen zusammen mit jeweils initial leeren Weg-Liste aufruft. Durch Konkatenation ALLER Ergebnisse ergibt sich die gesuchte Codierung der Eingabe-Liste.

```
# let rec huffman_encode_charlist zeichenkette baum
        = match zeichenkette with
        | [] -> []
        | hd::tail -> (huffman_encode_rec hd baum [])
                      @
                      (huffman_encode_charlist tail baum)
;;
val huffman_encode_charlist : char list -> hufftree ->
                              char list = <fun>
```

Und nun Aufruf der Funktion nach *Zerstückelung* des Eingabe-Strings:

```
# let huffman_encode eingabestring baum
        = huffman_encode_charlist (list_of_string eingabestring) baum
;;
val huffman_encode : string -> hufftree -> char list = <fun>
```

b) In Anlehnung an die *huffman_encode_rec* geschieht die Erstellung der

Codetabelle: Wiederum wird durch den vollständigen Baum gelaufen, nur das diesmal für jedes Atom die Wegliste zusammen mit dem Zeichen (in der gesuchten Form) zurückgegeben wird:

```
# let rec generate_codetable_rec baum ergebnis = match baum with
        | Atom(a, cnt) -> [a, ergebnis]
        | Node(l,r)    ->
                          (generate_codetable_rec l (ergebnis@['O']))
                          @
                          (generate_codetable_rec r (ergebnis@['L']))
;;
val generate_codetable_rec : hufftree ->
       char list -> (char * char list) list = <fun>

# let generate_codetable baum = generate_codetable_rec baum [];;
val generate_codetable : hufftree -> (char * char list) list = <fun>

Beispiel:

# let ht = Node(Node(Atom('a',1), Atom('b',2)),
                Node(Atom('c',3), Atom('d',4))
               )
;;

# generate_codetable ht;;
- : (char * char list) list =
[('a', ['O'; 'O']); ('b', ['O'; 'L']); ('c', ['L'; 'O']);
 ('d', ['L'; 'L'])
]
```

c) Die Suche nach dem richtigen Code für ein Zeichen vereinfacht sich deshalb zu einer Suche in der Codetabelle:

```
# let rec huffman_decode_char zeichen codetable = match codetable with
        | []             -> failwith("char not in codetable")
        | hd::tail when (fst hd) = zeichen
                         -> snd hd
        | hd::tail       -> huffman_decode_char zeichen tail
;;
```

Für die Codierung einer Zeichen-Liste ergibt sich damit:

```
# let rec huffman_encode_charlist zeichenkette codetable
     =  match zeichenkette with
        | []       -> []
        | hd::tail -> (huffman_encode_char hd codetable)
                      @
                      (huffman_encode_charlist tail codetable)
;;
```

Damit dann die Codierung eines Strings mit einem Huffman-Baum:

```
# let huffman_encode str baum
    = huffman_encode_charlist
            (list_of_string str) (generate_codetable baum)
;;
```