Technische Universität München
Fakultät für Informatik
Prof. Dr. A. Knoll

SS 2006
Übungsblatt 2
19. Mai 2006

Übungen zu Einführung in die Informatik II

Aufgabe 5 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 6 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 7 Kontonummern (Lösungsvorschlag)

Eine mögliche Implementierung könnte folgendermaßen aussehen (beachten Sie die Kommentare zu den einzelnen Programmteilen:

```
import java.util.Vector;
import java.lang.Math;
import java.text.DecimalFormat;

class Konto
{
  public static void main(String args[]) {

    // Vorgabe der Menge an Kontonnummern die erzeugt werden soll,
    // falls vorhanden wird der Wert durch die Benutzereingabe gesetzt.

    int num = 15;
    if (args.length >= 1) num = new Integer(args[0]).intValue();


    // Vorgabe der Menge an Kontonnummern die erzeugt werden soll,
    // falls vorhanden wird der Wert durch die Benutzereingabe gesetzt.

    int start = 1000;
    if (args.length >= 2) start = new Integer(args[1]).intValue();

    System.out.println("Die ersten " + num + " Kontonummern mit Startzahl " + start);


    // Als naechstes wird die Loesungsmenge der Kontonummern als Vektor angelegt
    // und "start" als erstes Element in die Menge eingefuegt.

    Vector v = new Vector();
    v.addElement(new Integer(start));

    int zaehler = 0;


    // In der folgenden while-Schleife wird "zaehler" sukzessive inkrementiert
    // und sein Hammingabstand zu jeder Zahl die bereits in der Loesungsmenge
    // enthalten ist ueberprueft. Entspricht der Abstand jeweils der geforderten
    // Bedingung, so wird "zaehler" als neue Kontonummer in die Menge
    // aufgenommen. Dies wird solange wiederholt, bis die erforderliche Anzahl
    // (== num) erreicht ist.

    while (v.size() < num) {
      boolean fitsIn = true;
      for (int i = 0; i < v.size(); i++) {
        int ktoNr = ((Integer)v.elementAt(i)).intValue();
        fitsIn &= (hamDist(zaehler, ktoNr) >= 3);
      }
      if (fitsIn) v.addElement(new Integer(zaehler));
      zaehler++;
    }
```

```
    // Zum Schluss wird die Ausgabe so formatiert, dass zu kurze Zahlen mit
    // fuehrenden Nullen aufgefuellt werden

    DecimalFormat df = new DecimalFormat();
    df.setGroupingSize(100);
    df.setMinimumIntegerDigits((int)Math.floor(Math.log((double)zaehler) / Math.log(10)) + 1);
    for (int i = 0; i < v.size(); i++) {
      System.out.println(df.format(((Integer)v.elementAt(i)).intValue()));
    }
  }


  // Die Funktion hamDist brechnet den Hammingabstand und gibt diesen zurueck
  // Sie arbeitet nach folgendem Prinzip: Zunaechst wird die Laenge der
  // groessten Zahl berechnet (max. Stellen). Hinweis: zur Vermeidung von
  // Rundungsfehlern wird die Laenge mit
  // floor(log10(max(a,b) + 1)) + 1 berechnet anstatt
  // floor(log10(max(a,b))) + 1
  // Danach werden einander entsprechende Stellen in der Zahl selektiert und
  // miteinander verglichen. Unterscheidet sich die Stelle, so wird der Abstand
  // jeweils inkrementiert.

  private static int hamDist(int a, int b) {
    int res = 0;
    long stellen = stellen = (int)Math.floor(Math.log((double)(Math.max(a,b) + 1)) / Math.log(10
    int base = 10;
    for (int i = 0; i < stellen; i++) {
      int cpA, cpB;
      if (i == 0) {
        cpA = a % base;
        cpB = b % base;
      }
      else {
        cpA = (a - a % base) % (10 * base) / base;
        cpB = (b - b % base) % (10 * base) / base;
        base *= 10;
      }
      if (cpA != cpB) res++;
    }
    return res;
  }

}
```