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

**SS 2006**
**Übungsblatt 5**
**9. Juni 2006**

# Übungen zu Einführung in die Informatik I

## Aufgabe 17 Sortierte Bäume in OCaml

In dieser Aufgabe wenden wir uns sortierten binären Bäumen zu. Dazu verwenden wir eine Sortierfunktion `f: 'a -> 'a -> int`, die zwei Elemente $a$ und $b$ von beliebigen Typ vergleicht und $-1$ für $a < b$, 0 für $a = b$ und 1 für $a > b$ zurückgibt. Ein Baum ist bezüglich `f` sortiert, wenn für jeden Knoten gilt: *alle Elemente im linken Teilbaum* $\leq$ *Element des Knotens* $<$ *alle Elemente im rechten Teilbaum*

a) Verwenden Sie die Datenstruktur aus Aufgabe 18. In dem Baum sollen zweidimensionale Koordinaten $(x,y)$ gespeichert werden. Schreiben Sie eine Sortierfunktion `comp_xy` die die Koordinaten lexikographisch vergleicht.

b) Schreiben Sie eine Funktion `in_order`, die eine Liste der Elemente des Baumes in *InOrder*-Reihenfolge (Linker Teilbaum - Knoten - Rechter Teilbaum) zurückgibt.

c) Schreiben Sie die Funktion `sort_tree f btree`. Diese Funktion sortiert den Baum `btree` bezüglich der Funktion `f`. Implementieren Sie dazu einen Algorithmus der mittels sortierten Einfügen rekursiv einen sortierten Baum erzeugt (Insertsort). (Hinweis: In dem sortierten Baum soll jedes Element nur einmal vorhanden sein, d.h. es werden Elemente nur eingefügt, solange sie noch nicht im Baum enthalten sind). Verwenden Sie für die Implementierung nur funktionale Konzepte und testen Sie Ihre Funktionen anhand eines geeigneten Beispielbaumes. Verwenden Sie zum Testen Koordinatenwerte $< (1000, 1000)$.

d) Optional: Schreiben Sie eine Funktion `pre_order` und `post_order`, die eine Liste der Elemente des Baumes in *PreOrder* bzw. *PostOrder*-Reihenfolge zurückgibt.

e) Schreiben Sie eine Funktion `map_tree`, die eine Funktion `f` auf jedes Element des Baumes anwendet. Die Struktur des Baumes bleibt dabei erhalten. Linearisieren Sie die Elemente des Baumes durch Anwendung von `map_tree`. (Hinweis: Speichern Sie hierfür in den Knoten den Wert $x \cdot 1000 + y$).

## Aufgabe 18 Verifikation funktionaler Programme

Gegeben sei folgende OCaml-Funktion zur Umkehrung einer Liste:

```
let rec rev list = match list with
        | [] -> []
        | el :: rest -> app (rev rest) [el]
```

sowie die Funktion `app`:

```
let rec app x y =
  match x with
      [] -> y
    | x::xs -> x :: app xs y
```

Ziel dieser Aufgabe ist es zu zeigen, dass die Funktion `rev` involutorisch (oder selbst-invers ist), d.h. es gilt:

$$\texttt{rev}\ (\texttt{rev}\ \texttt{list}) = \texttt{list} \tag{1}$$

a) Um die Gültigkeit des Prädikates (1) zu zeigen, muss zunächst Folgendes gezeigt werden:

$$\texttt{rev}\ (\texttt{app}\ \texttt{xs}\ \texttt{ys}) = \texttt{app}\ (\texttt{rev}\ \texttt{ys})\ (\texttt{rev}\ \texttt{xs}) \tag{2}$$

b) Zeigen Sie nun die Gültigkeit des Prädikates (1)!

## Aufgabe 19 Existenz- und Allquantor

In der Prädikatenlogik treten Ausdrücke mit dem sogenannten Existenzquantor $\exists$ und dem so genannten Allquantor $\forall$ auf, wie etwa $\forall\, x \in \mathbb{N} : f(x)$ oder $\exists\, x,y \in \mathbb{N} : g(x,y)$; hierbei handelt es sich bei $f$ und $g$ um boolesche Funktionen, d.h ihr Ergebnis ist ein Wahrheitwert.

Auf Sequenzen seien nun in analoger Weise die Funktionen `existiert` und `fuer_alle` in folgender Weise definiert: Sei `f` eine einstellige, boolesche Funktion über $\mathbb{Z}$ und `l` eine Liste über $\mathbb{Z}$. Dann gilt:

$$existiert(f,l) = \begin{cases} true & \text{wenn die Liste ein Element x enthält mit f(x) = true} \\ false & \text{sonst} \end{cases}$$

und

$$fuer\_alle(f,l) = \begin{cases} true & \text{wenn für alle Elemente der Liste gilt: f(x) = true} \\ false & \text{sonst} \end{cases}$$

Modellieren Sie die Funktionen `existiert` und `fuer_alle` durch eine geeignete rekursive Funktion in OCaml. Geben Sie dabei auch die Signatur der Funktionen an!

## Aufgabe 20 (H) Konvertierung zwischen römischen und arabischen Zahlen (5+5 Punkte)

Die römischen Zahlen, mit ihrem Ursprung im antiken römischen Reich, eignen sich zur Darstellung positiver ganzer Zahlen. Mit Hilfe von Symbolen (I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000) können in einem Additionssystem Zahlen nach folgenden Regeln gebildet werden:

- Begonnen wird links mit dem Symbol der größten Zahl. Dabei werden die Symbole I, X, C, M höchsten drei mal hintereinander geschrieben. Die Symbole V, L, D kommen nur einzeln vor. Die größte darstellbare Zahl ist somit 3999.
- Symbole einer kleineren Zahl, die vor dem einer größeren stehen, werden von diesem subtrahiert, wobei allerdings nur höchstens ein kleineres Symbol einem größeren vorangestellt werden darf.
- Einer (I) können nur von Fünfern (V) und Zehnern (X) abgezogen werden.
- Zehner (X) können nur von Fünfzigern (L) und Hundertern (C) abgezogen werden.
- Hunderter (C) können nur von Fünfhundertern (D) und Tausendern (M) abgezogen werden.

Eine Zahl wird durch die Umsetzung der einzelnen Ziffern der Zahl in entsprechende Symbole der römischen Zahlen gebildet. Die endültige Zahl ergibt sich dann aus der Addition und Subtraktion der einzelnen Symbole.

**Aufgabenstellung:** Implementieren Sie Funktionen zur Konvertierung aus dem Arabischen ins Römische Zahlensystem und umgekehrt.