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

SS 2006
Übungsblatt 7
23. Juni 2006

Übungen zu Einführung in die Informatik I

Aufgabe 26 Funktionen höherer Ordnung: fold_left

In Ocaml gibt es neben der in der Vorlesung vorgestellten Funktion `fold_right` noch die Funktion `fold_left`:

$$\texttt{fold\_left}\ f\ x\ [x_1;\ \ldots\ ;x_n] = f\ (\ \ldots\ (f\ (f\ x\ x_1)\ x_2)\ \ldots\ )\ x_n$$

a) Implementieren Sie die Funktion `fold_left` in OCaml.

b) Implementieren Sie basierend auf `fold_left` eine Funktion `list_sum`, welche die Werte einer Integer-Liste addiert.

c) Machen Sie sich den Unterschied zwischen `fold_left` und `fold_right` deutlich. Führen sie hierfür die Auswertung der Funktion `list_sum [2;4;6;8]` Schritt für Schritt durch. Wie sieht im Gegensatz dazu Auswertung für `fold_right sum [2;4;6;8] 0` aus?

d) Implementieren Sie die den Existenz- und Allquantor aus Aufgabe 26 basierend auf `fold_left`.

Aufgabe 27 Von OCaml zu Java

Auf den bisherigen Übungsblättern wurden rekursive Funktionen in OCaml implementiert, so z.B. die Berechnung der Zahlensumme:

$$\text{sum}(n) = \sum_{i=1}^{i=n} i$$

Ziel dieser Aufgabe ist es, bereits erlernte Verfahren in eine weitere Programmiersprache, in diesem Fall Java, zu übertragen. Ein geeignete rekursive OCaml-Funktion zur Lösung eben genannter Aufgabe könnte wie folgt aussehen:

```
(* Rekursive Berechnung der Summe *)
let rec sum n = match n with
    1 -> 1
  | _ -> n + sum (n - 1);;

(* Aufruf für den Wert 4 und Ausgabe mit Zeilenumbruch *)
print_endline (string_of_int (sum 4));;
```

Eine Umsetzung dieser Funktion in ein entsprechendes Java-Programm lautet:

```
public class Sum {
    /* Rekursive Berechnung der Summe */
    public static int sum(int n) {
        if (n == 1) {
```

```
            return 1;
        } else {
            return n + sum(n - 1);
        }
    }

    public static void main(String[] args) {
        // Aufruf für den Wert 4 und Ausgabe mit Zeilenumbruch
        System.out.println(sum(4));
    }
}
```

a) Vergleichen Sie beide Implementierungen miteinander und arbeiten Sie die für Java wichtigen Konzepte heraus.

b) Üben Sie Funktionen in OCaml und Java anhand eines Beispiels `fullName`, das aus den Parametern `firstName` und `lastName` den vollständigen Namen zurückliefert. Für die Darstellung von Zeichenketten kann der Java-Typ `String` verwendet werden.

c) Wiederholen Sie die Implementierung einer geeigneten Funktion zur Berechnung der Fibonacci-Zahlen in OCaml.

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \qquad \text{fib}(0) = 0, \quad \text{fib}(1) = 1$$

d) Implementieren Sie nun Ihre OCaml-Funktion geeignet in Java.

e) Testen Sie beide Lösungen mit geeigneten Eingaben.

Aufgabe 28 Aufrufstrategien bei Ocaml

a) Gegeben sei die Ocaml-Funktion **let** zero x = 0. D.h. es handelt sich um eine Funktion, die unabhängig vom Parameter die Null zurückgibt. Was passiert, wenn diese Funktion mit zero(fib (200)) aufgerufen wird ? (Testen Sie dies direkt am Rechner!)

b) Die Strategie nach der Ocaml Funktionsaufrufe auswertet, nennt man „Eager Evaluation“. Was könnte sich dahinter verbergen und inwiefern ist diese Strategie problematisch?

c) Betrachten Sie die Funktion höherer Ordnung map (f, ls)! Weshalb kann bei Argumenten, die Funktionen sind, die Aufrufstrategie „Eager Evaluation“ nicht angewandt werden?

d) In Ocaml ist das leere Tupel () ein spezieller Datentyp, der nur aus einem einzigen Element namens `unit` besteht. Der Datentyp des leeren Tupels läßt sich dazu verwenden, um die Auswertung eines Ausdrucks zu verzögern und damit eine „Lazy Evaluation“ Strategie umzusetzen. Testen Sie folgende Funktionsdeklarationen:

```
let f x = match x with
 |() -> fib (20);;
```

Der Ausdruck fib (20) wird vorerst nicht ausgewertet, sondern nur die Signatur angegeben: `val f : unit -> int = <fun>`. Was ist zu tun, um die Auswertung des Ausdrucks fib (20) zu erzwingen?

e) Bei einer Funktion höherer Ordnung läßt sich die Signatur einer Funktion, die als Argument übergeben wird, detailliert angeben. Beispielsweise könnte man die Funktion map auch durch

```
let rec map (f : (int -> int)) ls = match ls with
 |[] -> []
 |x :: rest -> (f x) :: (map f rest);;
```

festlegen. (Testen Sie diese Funktion!) Inwiefern bedeutet dies eine Einschränkung gegenüber der bekannten map-Funktion? Führen Sie diese Einschränkung auch bei der Funktion filter durch!

f) Definieren Sie die Funktion zero als Funktion höherer Ordnung, die die Argumente nur bei Bedarf auswertet, also letztendlich eine „Lazy"-Strategie umsetzt und rufen sie die Funktion mit fib (200) gemäß der „Lazy"-Strategie auf.

Aufgabe 29 Türme von Hanoi in Java

In Aufgabe 6 des 2. Übungsblattes wurde bereits eine Lösung für das Knobelspiel *Türme von Hanoi* in OCaml programmiert. Benutzen Sie die Lösung dieser Aufgabe um eine ähnliche Implementierung in Java zu entwickeln.

a) Definieren Sie eine statische Hilfsfunktion `stringOfTower`, welche die Position des Turmes darstellt.

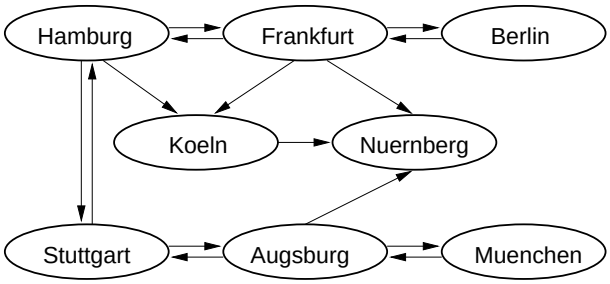
b) Definieren Sie eine statische Hilfsfunktion `stringOfMove`, die einen Zug darstellt.

c) Definieren Sie die statische Funktion `hanoi`, die den Turm von Hanoi verschiebt.

d) Testen Sie Ihre Implementierungen geeignet.

Aufgabe 30 (H) Breitensuche in gerichteten Graphen (10 Punkte)

Auf Blatt 6 haben wir Graphen betrachtet mit Kanten (u,v). In vielen Problemen ist es sinnvoll, sich die Kanten gerichtet vorzustellen. Ein Straßensystem mit Einbahnstraßen oder in eine Richtung gesperrte Autobahnen sind prägnante Beispiele. Ein gerichteter Graph $\vec{G}(E,K)$ besteht demnach aus einer endlichen Menge von Knoten E und einer endlichen Menge von Kanten $K = \{(u,v) | u,v \in E \wedge u \neq v \wedge \texttt{Es gibt eine gerichtete Verbindung zwischen}\ u\ \texttt{und}\ v\}$. In $k = (u,v)$ heißt $u = k^-$ der Anfangsknoten und $v = k^+$ der Endknoten von k. Betrachten wir nun wieder unsere leicht modifizierte Straßenkarte:



Im Gegensatz zur Tiefensuche, wo der *erste* unbesuchte Nachbar immer weiter expandiert wurde, werden bei der Breitensuche zunächst *alle* Nachbarn eines Knotens nacheinander untersucht. Informell bleibt der auf Blatt 6 vorgestellte Algorithmus auch für die Breitensuche unverändert, nur dass die *LIFO*-Datenstruktur (`Stack`) durch eine *FIFO*-Datenstruktur (`Queue`) ersetzt wird.

Aufgabenstellung:

a) Implementieren Sie die Zugriffsoperationen der $FIFO$-Datenstruktur `Queue` unter Zuhilfenahme von Listen.

b) Übernehmen Sie die Hilfsfunktionen der Tiefensuche und modifizieren Sie sie derart, dass nun gerichtete Kanten berücksichtigt werden können.

c) Implementieren Sie eine generische Suchfunktion, der die Zugriffsfunktionen der entsprechenden Hilfs-Datenstruktur als Parameter übergeben werden können.

d) Das Ergebnis der Suche soll nun kein Wahrheitswert sein, sondern eine Liste aller besuchten Knoten auf dem Weg von der Quelle zum Ziel.

e) Testen Sie Ihr Programm indem Sie überprüfen, ob man von Berlin nach München kommt, jeweils mit Tiefen- und Breitensuche.