Technische Universität München
Fakultät für Informatik
D. Gmach, M. Rickert,
Dr. M. Schneider

WS 2005/2006
Lösungsvorschläge zu Blatt 8
16. Dezember 2005

Übungen zu Einführung in die Informatik I

Aufgabe 32 Funktionen höherer Ordnung: fold_left (Lösungsvorschlag)

a)
```
# let rec fold_left f a = function
          [] -> a
          | head::tail -> fold_left f (f a head) tail;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (+) 0 [2;4;6;8];;
- : int = 20
```

b)
```
# let list_sum = fold_left (+) 0;;
val list_sum : int list -> int = <fun>
# list_sum [2;4;6;8];;
- : int = 20
```

c) Der Unterschied zwischen `fold_right` und `fold_left` wird durch die Auswertung der Funktionen deutlich. `fold_left` wendet die Funktion `f` von links auf die Elemente der Liste an und `fold_right` wendet `f` zuerst auf des letzte Element an.

- Auswertung von `list_sum [2;4;6;8]`:

```
list_sum [2;4;6;8] = fold_left (+) 0 [2;4;6;8]
                   = fold_left (+) (2) [4;6;8]
                   = fold_left (+) (6) [6;8]
                   = fold_left (+) (12) [8]
                   = fold_left (+) (20) []
                   = 20
```

- Auswertung von `fold_right sum [2;4;6;8] 0`:

```
fold_right sum [2;4;6;8] 0
  = sum 2 (fold_right sum [4;6;8] 0)
  = sum 2 (sum 4 (fold_right sum [6;8] 0))
  = sum 2 (sum 4 (sum 6 (fold_right sum [8] 0)))
  = sum 2 (sum 4 (sum 6 (sum 8 (fold_right sum [] 0))))
  = sum 2 (sum 4 (sum 6 (sum 8 0)))
  = sum 2 (sum 4 (sum 6 8))
  = sum 2 (sum 4 14)
  = sum 2 18
  = 20
```

Zusätzlich ist `fold_left` im Gegensatz zu `fold_right` Endrekursiv, d. h., der definierende Teil der Funktion ist kein zusammengesetzter Ausdruck (z. B.: 1 + f(n-1)), sondern besteht nur aus einem rekursiven Aufruf. Dadurch wird für die Auswertung von `fold_left` nur konstant Speicherplatz ($O(1)$) benötigt.

d)
```
# let for_all f =
      fold_left (fun x y -> x && f y) true;;
val for_all : ('a -> bool) -> 'a list -> bool = <fun>

# for_all (fun x -> if (x<10) then true else false) [3;5;2;7;9];;
- : bool = true
# for_all (fun x -> if (x<10) then true else false) [3;5;2;11;9];;
- : bool = false

# let exists f =
      fold_left (fun x y -> x || f y) false;;
val exists : ('a -> bool) -> 'a list -> bool = <fun>

# exists (fun x -> if (x>10) then true else false) [3;5;2;7;9];;
- : bool = false
# exists (fun x -> if (x>10) then true else false) [3;5;2;11;9];;
- : bool = true
```

Aufgabe 33 Von OCaml zu Java (Lösungsvorschlag)

a)
- Alles in Java wird innerhalb einer Klasse geschrieben.
- Quellcodedateien müssen den gleichen Namen tragen wie die darin enthaltene Klasse und der Endung `.java` (`Sum.java` im Beispiel).
- Der Compiler generiert für jede Klasse einer Quellcodedatei eine zugehörige Datei mit der Endung `.class` (in der Klasse enthaltene Klassen wird der Name der Oberklasse und ein $-Zeichen vorangestellt).
- Jedes ausführbare Programm muß über eine `main()`-Methode verfügen.
- Typen müssen explizit angegeben werden (`int`, `float`)
- Das Schlüsselwort `void` bedeutet daß die `main()`-Methode keinen Wert zurückgibt (s. `unit` bzw. `()` in OCaml).
- Der `main()`-Methode muß eine Liste von Strings übergeben werden, meistens als `args` bezeichnet, der Name ist jedoch freigestellt.
- Das Schlüsselwort `static` steht für eine Klassenmethode (diese ist mit der Klasse selbst, nicht einer Objektinstanz verknüpft). Die `main()`-Methode muß statisch definiert sein.
- Das Schlüsselwort `public` bezeichnet eine Methode oder Klasse die von anderen Klasse aus aufgerufen werden kann.
- Die hier verwendete Druckfunktion ist teil der *Java Standard Library*. Die Klasse `System` verfügt über ein public-Feld namens `out`, das eine Methode `println()` für die Ausgabe auf dem Bildschirm zur Verfügung stellt (standard out).
- Kompilieren über den Aufruf von `javac` und der Hauptklasse mitsamt Dateiendung (`javac Sum.java` im Beispiel).
- Ausführen über den Aufruf von `java` und der Hauptklasse (`java Sum` im Beispiel).

b)
```
let fullName firstName lastName = firstName ^ " " ^ lastName;;

fullName "Max" "Mustermann";;
```

```
public static String fullName(String firstName, String lastName) {
    return firstName + " " + lastName;
}

fullName("Max", "Mustermann");
```

c)
```
let rec fib n = match n with
    0 -> 0
  | 1 -> 1
  | _ -> fib(n - 1) + fib(n - 2);;

fib 0;;
fib 1;;
fib 2;;
fib 3;;
fib 4;;
fib 5;;
```

d)
```
public class Fib {
    public static int fib(int n) {
        if (n == 0) {
            return 0;
        } else if (n == 1) {
            return 1;
        } else {
            return fib(n - 1) + fib(n - 2);
        }
    }

    public static void main(String[] args) {
        System.out.println(fib(0));
        System.out.println(fib(1));
        System.out.println(fib(2));
        System.out.println(fib(3));
        System.out.println(fib(4));
        System.out.println(fib(5));
    }
}
```

e) Test mit Eingaben $0, 1, 2, 3, 4, 5$ (s. Quellcode) für OCaml:

```
val fib : int -> int = <fun>
- : int = 0
- : int = 1
- : int = 1
- : int = 2
- : int = 3
- : int = 5
```

und Java:

```
0
1
1
2
3
5
```

Aufgabe 34 Aufrufstrategien bei Ocaml (Lösungsvorschlag)

a) Ruft man die Funktion zero (fib(200)) auf, so kommt es höchstwahrscheinlich zu einem so genannten „stack overflow“,d.h. es müßen zu viele rekursive Aufrufe der Funktion

```
let rec fib n = match n with
 |0|1 -> 1
 |n -> fib(n-1) + fib(n-2);;
```

archiviert werden. Grund für die zahlreichen Aufrufe ist die nichtlinear rekursive Fibonacci-Funktion. Bei der Funktion zero ist allerdings der Wert des Argumentes unwichtig, es wird zur Berechnung des Funktionsresultats nicht gebraucht.

b) Man erkennt an dem obigen Beispiel, dass Ocaml eine Aufrufstrategie verfolgt, bei der zuerst die Argumente der Funktion und dann der Funktionsrumpf berechnet wird. Diese Strategie ist unabhängig davon, ob die Argumente in der Funktion später gebraucht werden. Man nennt eine solche deshalb „strenge Auswertung “ oder „Eager Evaluation“. Diese Strategie ist deshalb problematisch, da sie die Schnelligkeit der Berechnung negativ beeinflußen kann, da evtl. Zwischenergebnisse berechnet werden, die später überhaupt nicht gebraucht werden.

c) Funktionen höherer Ordnung, wie die bekannte Funktion map

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

haben Funktionen als Argumente. Betrachtet man die Implementierung der Funktion map so erkennt man, dass bei der Funktion f die „Eager“-Strategie nicht angewendet werden kann, denn die Auswertung der Funktion erfolgt erst im Funktionsrumpf. Folglich ist bei den funktionsartigen Argumenten von Funktionen höherer Ordnung eine „Eager“-Strategie nicht möglich. Darüber ergäbe sich folglich die Möglichkeit den „Eager“-Mechanismuszu umgehen.

d) Die Auswertung des Ausdrucks fib (20) kann durch f (), d.h. die Funktion f wird auf das leere Tupel angewandt, erzwungen werden.

e) Die Filterfunktion läßt sich sinnvoll wie folgt einschränken:

```
let rec filter (f: ('a -> bool)) ls = match ls with
 |[] -> []
 |x :: rest when (f x) -> x :: (filter f rest)
 |x :: rest -> (filter f rest);;
```

f) Definiert man die Funktion zero durch

```
let zero (f: unit -> 'a) = 0;;
```

so kann durch den Aufruf

```
zero (fun () -> fib (200));;
```

eine „Lazy-Evaluation“ erzwungen werden.

Aufgabe 35 Türme von Hanoi in Java (Lösungsvorschlag)

a)
```
public static String stringOfTower(int tower) {
    switch (tower) {
    case 0:
        return "A";
    case 1:
        return "B";
    case 2:
        return "C";
    default:
        return "";
    }
}
```

b)
```
public static String stringOfMove(int start, int goal) {
    return stringOfTower(start) + " -> " + stringOfTower(goal) + "\n";
}
```

c)
```
public static String hanoi(int n, int start, int goal) {
    if (n != 0) {
        return hanoi(n - 1, start, 3 - start - goal)
             + stringOfMove(start, goal)
             + hanoi(n - 1, 3 - start - goal, goal);
    } else {
        return "";
    }
}
```

d)
```
public static void main(String[] args) {
    System.out.print(hanoi(3, 0, 1));
}
```