TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
Lehrstuhl für Sprachen und Beschreibungsstrukturen
Einführung in die Informatik 2
Prof. Dr. Helmut Seidl, Ralf Vogler, Stefan Schulze Frielinghaus

WS 2015/16
Übungsblatt 7

Aufgabe 7.1 [21 Punkte] **OCaml-Hausaufgabe: Listen-Funktionen**

Zur Abgabe melden Sie sich mit Ihrer TUM-Kennung auf `https://vmnipkow3.in.tum.de` an und laden ihre `ha3.ml`-Datei (nicht umbenennen!) hoch. Die Ergebnisse der Tests sind sichtbar sobald die Abgabe abgearbeitet wurde. Die endgültige Bewertung erfolgt aber erst nach der Frist.

In der Datei gibt es zwei Module mit Funktionen die zu implementieren sind:

a) `MyList`: mehr Funktionen auf Listen (beachten Sie die Signatur in `ha3.mli`),

b) `MySet`: durch Listen simuliertes Set.

Lösungsvorschlag 7.1

Siehe Moodle.

Aufgabe 7.2 OCaml-Tutoraufgaben

a) Wie oft wird die Funktion `fib` für ein $n > 1$ aufgerufen? Bestimmen Sie die Laufzeitklasse.

```
let rec fib = function
  | 0 | 1 -> 1
  | n -> fib (n-1) + fib (n-2)
```

Überprüfen Sie Ihre Vermutung indem Sie in jedem Aufruf das aktuelle n ausgeben. Wie viele unnötige Aufrufe ergeben sich für ein $n > 1$ wenn man nur am Ergebnis und nicht an Seiteneffekten interessiert ist?

b) Modifizieren Sie die Funktion so, dass die Anzahl an Aufrufen in $\mathcal{O}(n)$ liegt.

c) Überlegen Sie sich welche Signatur die folgenden Funktionen im allgemeinsten Fall haben und implementieren Sie diese.

```
(* return Some element at index i or None if out of bounds.
   e.g. at 1 [1;2;3] = Some 2, at 1 [] = None *)
let at = ??
(* concatenate a list of lists. e.g. flatten [[1];[2;3]] = [1;2;3] *)
let flatten = ??
(* a list containing n elements x.
   e.g. make 3 1 = [1;1;1], make 2 'x' = ['x';'x'] *)
let make = ??
(* range 1 3 = [1;2;3], range 1 (-1) = [1;0;-1] *)
let range = ??
```

Lösungsvorschlag 7.2

a) Eine einfach zu zeigende obere Schranke für die Laufzeit ist $\mathcal{O}(2^n)$.

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

Die Anzahl unnötiger Aufrufe der exponentiellen Version ist also in $\mathcal{O}(2^{n-1} - n)$.

b) Wir berechnen zwei Ergebnisse pro Aufruf:

```
let rec fib_pair = function
  | 0 -> 1,1
  | n -> let x,y = fib_pair (n-1) in y,x+y
```

Wobei uns nur das erste Element interessiert:

```
let fib = fst % fib_pair
```

c) Signaturen:

```
val at : int -> 'a list -> 'a option
val flatten : 'a list list -> 'a list
val make : int -> 'a -> 'a list
val range : int -> int -> int list
```

Implementierung:

```
(* return Some element at index i or None if out of bounds.
   e.g. at 1 [1;2;3] = Some 2, at 1 [] = None *)
let at i =
  let rec f j = function
    | [] -> None
    | x::xs -> if j=i then Some x else f (j+1) xs
  in f 0
(* concatenate a list of lists. e.g. flatten [[1];[2;3]] = [1;2;3] *)
let rec flatten = function
  | [] -> []
  | x::xs -> x @ flatten xs
(* a list containing n elements x.
   e.g. make 3 1 = [1;1;1], make 2 'x' = ['x';'x'] *)
let rec make n x = if n <= 0 then [] else x :: make (n-1) x
(* range 1 3 = [1;2;3], range 1 (-1) = [1;0;-1] *)
let rec range a b =
  let s x = if a<=b then x+1 else x-1 in
  if a=b then [a]
  else a :: range (s a) b
```