Technische Universität München
Fakultät für Informatik
M. Bauer, D. Golias, S. Riesner,
Dr. M. Schneider

WS 2002/2003
Lösungsvorschläge zu Blatt 3
7. November 2002

Übungen zu Einführung in die Informatik I

Aufgabe 9 Arbeiten mit Modulen

Zur besseren Übersicht und um gleichnamige Operationen für verschiedenen Datentypen zu unterscheiden verwendet OCaml sagenannte *Module*. Ein Besipiel haben wir auf dem letzten Übungsblatt bereits kennengelernt: `Random.float` ruft die Funktion *float* aus dem Modul *Random* auf.

In dieser Aufgabe verwenden wir Funktionen aus dem Modul *List* zur Operation auf Listen von Datentypen.

a) Definieren und belegen Sie in OCaml die beiden Zeichensequenzen `vorname` und `nachname` jeweils mit Ihrem Vor- bzw. Nachnamen. Verknüpfen Sie anschließend diese beiden Zeichensequenzen zu einer einzigen Zeichensequenz `name` unter Verwendung des Konkatenationsoperator `@` und einem Komma sowie einem Leerzeichen zwischen den beiden Namensteilen.

```
# let nachname = ['l'; 'e'; 'g'; 'r'; 'a'; 'n'; 'd']
  and vorname = ['w'; 'i'; 'l'; 'l'; 'i'; 'a'; 'm'];;
val nachname : char list = ['l'; 'e'; 'g'; 'r'; 'a'; 'n'; 'd']
val vorname : char list = ['w'; 'i'; 'l'; 'l'; 'i'; 'a'; 'm']


# let name = nachname @ [','] @ [' '] @ vorname;;
val name : char list =
  ['l'; 'e'; 'g'; 'r'; 'a'; 'n'; 'd'; ','; ' ';
   'w'; 'i'; 'l'; 'l'; 'i'; 'a'; 'm']
```

b) Verwenden Sie die Funktion *length* aus dem Modul *List* um die Länge des ganzen Namens festzustellen.

```
# List.length name;;
- : int = 16
```

c) Revertieren Sie Ihren (oben definierten) Namen mit Hilfe von OCaml durch Zerlegung und unter ausschließlicher Verwendung der Funktionen *hd* und *tl* aus dem Modul *List* und des Operators `::` ('Prepend').
Hinweis: Zur Vereinfachung der Schreibweise bietet es sich an zuerst zwei Funktionen `head` und `tail` zu definieren, welche genau die entsprechenden Funktionen aus dem Modul *List* aufrufen.

```
# let head = List.hd;;
val head : 'a list -> 'a = <fun>
# let tail = List.tl;;
val tail : 'a list -> 'a list = <fun>

# head(tail(tail(tail(tail(tail(tail(name))))))) ::
head(tail(tail(tail(tail(tail(name)))))) ::
head(tail(tail(tail(tail(name))))) ::
head(tail(tail(tail(name)))) ::
head(tail(tail(name))) ::
head(tail(name)) ::
[head(name)]
;;
- : char list = ['d'; 'n'; 'a'; 'r'; 'g'; 'e'; 'l']
```

d) **(H)** Definieren Sie die folgende *Cast*-Funktion `list_of_string` die mit Hilfe von Funktionen aus den Modulen *String* und *Stream* aus einem String eine Liste von Einzelzeichen macht.

```
let list_of_string s =
  Stream.npeek (String.length s) (Stream.of_string s);;
```

Testen Sie diese Funktion mit ein paar Eingabestrings.

```
# list_of_string("evitez les courrants d'air");;
- : char list =
['e'; 'v'; 'i'; 't'; 'e'; 'z'; ' '; 'l'; 'e'; 's'; ' ';
 'c'; 'o'; 'u'; 'r'; 'r'; 'a'; 'n'; 't'; 's'; ' '; 'd'; '\''; 'a'; 'i'; 'r']
```

e) **(H)** Definieren Sie nun eine rekursive Funktion *revert*, die eine beliebige Liste umkehrt. Testen Sie diese Funktion für Ihren oben definierten Namen sowie für den String `reliefpfeiler` (unter Verwendung der Cast-Funktion aus Teilaufgabe c).

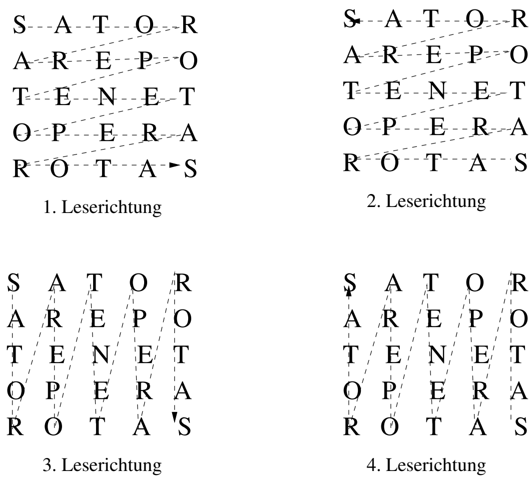
```
let rec revert l = match l with
  | []    -> []
  | a::l  -> revert l @ [a];;

# revert name;;
- : char list =
['m'; 'a'; 'i'; 'l'; 'l'; 'i'; 'w'; ' '; ',';
 'd'; 'n'; 'a'; 'r'; 'g'; 'e'; 'l']

# revert (list_of_string("reliefpfeiler"));;
- : char list =
['r'; 'e'; 'l'; 'i'; 'e'; 'f'; 'p'; 'f'; 'e'; 'i'; 'l'; 'e'; 'r']
```

Aufgabe 10 Operationen auf Zeichensequenzen (Lösungsvorschlag)

a) Bei genauerer Betrachtung des Palindrom erkennt man, dass sich bei folgenden Leserichtungen jeweils dasselbe Wort ergibt:



1. Leserichtung

2. Leserichtung

3. Leserichtung

4. Leserichtung

Es müßen also für die Leserichtungen 2 - 4 geeignete Funktionen angegeben werden.

Wir zeigen das am Beispiel der Richtung 2: Dazu wird zunächst folgende Sequenz von Sequenzen, die das Palindrom enthält, definiert:

```
let palindrom = [['S';'A';'T';'O';'R'];
                 ['A';'R';'E';'P';'O'];
                 ['T';'E';'N';'E';'T'];
                 ['O';'P';'E';'R';'A'];
                 ['R';'O';'T';'A';'S'];];;
```

Anschließend definieren wir eine Funktion `revert`, die eine Sequenz revertiert:

```
let rec revert ls = match ls with
        | []      -> []
        | x :: l  -> revert l @ ([x]);;
```

und schließlich bauen wir nach demselben Muster eine Funktion zur Revertierung des Palindroms:

```
let rec revertPal p = match p with
        |      [] -> []
        | x :: l  -> (revertPal l) @ [(revert x)];;
```

Um die 3. Leserichtung zu implementieren benötigen wir eine Funktion, die in der obigen Skizze, die Spalten mit den Zeilen vertauscht. Dies leisten folgende Funktionen:

```
#use "C:/Programme/Objective Caml/lib/list.ml";;

let rec elements3 (n, palin) = match (n, palin) with
    |(n, []) -> []
    |(n, x :: tail) -> (nth x n) :: elements3 (n, tail);;
```

```
let rec direction3 (rev, n, palin) = match (rev, n, palin) with
   |(rev, n, palin) when n = (length palin) -> rev
   |(rev, n, palin) -> direction3 (rev @ [(elements3 (n, palin))], n+1, palin);;

let  revertPalin3 palin = direction3 ([], 0, palin)
```

Die vierte Leserichtung geht beinahe analog zu Leserichtung 3:

```
let rec elements4 (n, palin) = match (n, palin) with
   |(n, []) -> []
   |(n, x :: tail) ->  elements4 (n, tail) @ [(nth x n)];;

let rec direction4 (rev, n, palin) = match (rev, n, palin) with
   |(rev, n, palin) when n < 0 -> rev
   |(rev, n, palin) -> direction4 (rev @ [(elements4 (n, palin))], n-1, palin);;

let  revertPalin4 palin = direction4 ([], (length palin)-1, palin)
```

b) Eine Funktion, die den „Existenzquantor“ für Sequenzen modelliert lautet:

```
# let rec existiert a = match a with
  | (f, []) -> false
  | (f, h::tail) when f h -> true
  | (f, h::tail) -> existiert (f, tail);;
val existiert : ('a -> bool) * 'a list -> bool = <fun>
```

Für den Allquantor ergibt sich analog:

```
# let rec fuer_alle a = match a with
  | (f, []) -> true
  | (f, h::tail) when not(f h) -> false
  | (f, h::tail)-> fuer_alle (f, tail);;
val fuer_alle : ('a -> bool) * 'a list -> bool = <fun>
```

Aufgabe 11 Einbettung

In dieser Aufgabe soll eine Funktion *max* implementiert werden, die aus einer gegebenen Liste das Maximum findet. Dazu wird die Liste der Reihe nach durchsucht und das jeweils größte gefundene Element zwischengespeichert.

- Definieren Sie eine (rekursive) eingebettete Funktion die das Maximum einer Liste und eines einzelnen Elements berechnet.

```
# let rec max_e a l = match l with
    []                  -> a
  | b::tail when b > a -> max_e b tail
  | b::tail             -> max_e a tail
  ;;
```

```
val max_e : 'a -> 'a list -> 'a = <fun>
# max_e 3 [1;2;6;2;];;
- : int = 6
# max_e 7 [1;2;6;2;];;
- : int = 7
```

- Betten Sie diese Funktion in eine neue Funktion *max* ein, um das Maximum einer Liste zu berechnen.

```
# let max l = match l with
    []   -> failwith "undefined"
  | a::t -> max_e a t;;
val max : 'a list -> 'a = <fun>
# max [1;2;6;2;];;
- : int = 6
```

Beachten Sie hier auch die Klammerung die durch die Auswertung von links her nötig ist:

```
# max list_of_string("was it a cat i saw?");;
This expression has type string -> char list but is here used with type
  ('a -> 'b) list
# max (list_of_string("was it a cat i saw?"));;
- : char = 'w'
```

Aufgabe 12 Binärbäume in OCaml (Lösungsvorschlag)

a) Entgegen der Vorlesung führen wir eine Baum-Definition ein, welche den Knoten nicht in der Mitte, sondern als erstes Element führt. Dies soll den Studenten zeigen, dass es mehrere unterschiedliche Realisations-Möglichkeiten gibt.

```
# type 'a btree = Empty | Node of ('a * 'a btree * 'a btree);;
type 'a btree = Empty | Node of ('a * 'a btree * 'a btree)
```

Die Deklaration von Typen mit `'a` sollte den Studenten aus der Vorlesung bekannt sein.

b) **(H)** Einhacken mit obiger Deklaration:

```
# let sample_tree1 =
  Node("Gaston",
  Node("Frodo", Node("Darth Vader",Node("Fred Clever",Empty,Empty),Empty),
  Node("Luke Skywalker",Node("Gandalf",Empty,Empty),Empty)),
  Node("Mr. Spock",Node("Jeff Smart",Empty,Empty),
  Node("Marvin",Empty,Node("Zaphod Beeblebrox",Empty,Empty))));;
val sample_tree1 : string btree =
  Node ("Gaston",
   Node ("Frodo",
    Node ("Darth Vader", Node ("Fred Clever", Empty, Empty), Empty),
    Node ("Luke Skywalker", Node ("Gandalf", Empty, Empty), Empty)),
   Node ("Mr. Spock", Node ("Jeff Smart", Empty, Empty),
    Node ("Marvin", Empty, Node ("Zaphod Beeblebrox", Empty, Empty))))
```

c) Funktion für Zahl der Knoten. Selbsterklärend.

```
# let rec count btree =
  match btree with
    Empty     -> 0
  | Node(_, left, right) -> 1 + count left + count right;;
val count : 'a btree -> int = <fun>
```

d) Suchen und Rückgabe des Niveaus:

```
# let rec niveau_recursive x level btree =
  match btree with
    Empty -> -1
  | Node(y, left, right)  when x = y -> level
  | Node(y, left, right)  when x < y -> niveau_recursive x (level+1) left
  | Node(y, left, right)             -> niveau_recursive x (level+1) right;;
val niveau_recursive : 'a -> int -> 'a btree -> int = <fun>
```

Und die Einbettungsfunktion:

```
# let niveau x btree =
  niveau_recursive x 0 btree;;
val niveau : 'a -> 'a btree -> int = <fun>
```

Die definierte rekursive Funktion `niveau_recursive` verwendet mit `level` den Abstand des Knotens von der Wurzel. Zum Einstieg setzt die Funktion `niveau` diesen auf 0. Ein leerer Baum wird mit -1 gezählt.

e) Ebenene des Baumes zählen

Hier sollen die Studenten darauf hingewiesen werden, dass eine ggf. naheliegende Lösung, nämlich u.U. die Niveau-Funktion zu verwenden, nicht unbedingt die geschickteste ist. Denn es geht viel, viel einfacher:

```
# let rec height btree =
  match btree with
    Empty -> 0
  | Node(_, left, right) -> 1 + max (height left) (height right);;
val height : 'a btree -> int = <fun>
```