**Ludwig-Maximilians-Universität München** WS 2005/2006
**Institut für Informatik** Blatt 8
Prof. Dr. F. Kröger, Dr. M. Hölzl, F. Hacklinger

# Übungen zu Informatik I (Lösungsvorschlag)

**Aufgabe 8-1** **Quantoren für Listen** (keine Abgabe)

a) Zunächst definieren wir die gesuchte Funktion durch Rekursion wie folgt:

```
fun exists _ nil = false
  | exists f l   = f(hd l) orelse exists f (tl l);
```

Man kann den Aufruf von *hd* und *tl* auch umgehen, indem man Pattern-Matching für $l$ verwendet:

```
fun exists _ nil = false
  | exists f (l::lt)   = f l orelse exists f lt;
```

Die Funktion *exists* läßt sich ohne explizite Rekursion definieren, wenn man die Linksfaltung einer Liste verwendet: Die Funktion *foldl* ist definiert als:

```
fun foldl f b nil = b
  | foldl f b (x::xs) = foldl f (f(x,b)) xs;;
```

Damit kann man die Funktion *exists* folgendermaßen implementieren:

```
fun exists f l = foldl (fn (a, b) => b orelse f(a)) false l;
```

Wenn man die Funktion *foldl* partiell anwendet kann man auf das explizites Argument für die Liste verzichten:

```
fun exists f = foldl (fn (a, b) => b orelse f(a)) false;
```

Falls bei einem Aufruf $exists(f)(l)$ die Funktion $f$ für jedes Element der Liste $l$ terminiert ohne eine Ausnahme auszulösen, liefert auch die folgende Definition von *exists* den korrekten Wert:

```
fun exists f = foldr (fn (a, b) => b orelse f(a)) false;
```

Für das folgende Beispiel stimmt das Verhalten der mit Hilfe der Rechtsfaltung definierten Version von *exists* aber nicht mit dem der vorhergehenden Varianten überein:

```
exists (fn x => (100 div x) = 2) [50, 0];
```

b) Wie eben, geben wir zunächst eine rekursive Definition der Funktion *all*:

```
fun all _ nil = true
  | all f l   = f(hd l) andalso all (f) (tl(l));
```

Auch in diesem Fall kann man die gesuchte Funktion in kompakterer Form definieren, wenn man die Faltung verwendet:

```
    (* mit Hilfe der Faltung *)

    fun all f = foldl (fn (a, b) => b andalso f(a)) (true);
```

**Aufgabe 8-2** **Kartenspiel mit Datentypen** (keine Abgabe)

Untenstehender SML-Code enthält die Lösung zu allen Teilaufgaben:

```
val all = List.all;
val exists = List.exists;
val filter = List.filter;

(* a *)
(* die Farbe der Karten ist ein Aufzählungstyp *)

datatype farbe =
  Herz | Karo | Kreuz | Pik;

(* Der Wert der Karte ist entweder eine Zahl oder ein Bild. *)

datatype wert =
  n2 | n3 | n4 | n5 | n6 | n7 | n8 | n9 | n10 | Bube | Dame | Koenig | Ass;

(* Eine Karte ist nun ein Paar (Wert, Bild).  Durch diese Definition
   des Datentyps koennen wir _alle_ Karten darstellen und nichts, was
   _keine_ Karte ist *)

datatype poker_karte = Poker_Karte of wert * farbe;

(* Obwohl es an dieser Stelle noch nicht noetig ist, definieren wir
   gleich die Projektionsfunktionen, die zu einer Karte Farbe
   bzw. Wert bestimmen. *)

fun wertvon    (Poker_Karte(k, c)) = k;
fun farbevon  (Poker_Karte(k, c)) = c;

(* Spaeter brauchen wir die verschiedenen Werte und die Farben, die
   eine Karte haben kann. *)

val farben = [Herz, Karo, Kreuz, Pik];
val werte = [n2, n3, n4, n5, n6, n7, n8, n9, n10, Bube, Dame, Koenig, Ass];

(* b *)
(* Um die Funktionen paar und drilling definieren zu koennen, verwenden
   wir eine Hilfsfunktion, die abzaehlt, wie oft ein Wert in einer
   Liste von Karten vorkommt. *)

fun nwerte ([], k) = 0
  | nwerte (x::xs, k) = if wertvon(x) = k
then 1 + nwerte(xs, k)
else nwerte(xs, k);
```

```
(* Diese Funktion lässt sich auf auch knapper aufschreiben.  Die
   folgenden Varianten zeigen, wie man schrittweise zur Formulierung
   nwerte4 kommt. *)

(* elemente_mit_wert ist eine Hilfsfunktion, die aus einer Liste eine
   Elemente, deren Wert ungleich k ist entfernt.  Genauer gesagt: die
   eine neue Liste zurückliefert, die nur die Elemente der
   ursprünglichen Liste enthält, deren Wert gleich k ist. *)

fun elemente_mit_wert k [] = []
  | elemente_mit_wert k (x::xs)
      = if wertvon(x) = k
then x :: (elemente_mit_wert k xs)
else elemente_mit_wert k xs;

fun nwerte1 (l, k)
  = let val ws = elemente_mit_wert k l
    in length ws
    end;

(* Die Definition der Hilfsfunktion elemente_mit_wert ist relativ
   umständlich.  Eleganter lässt sich das unter Verwendung der
   Funktion filter implementieren. *)

fun nwerte2 (l, k)
 = let val ws = filter (fn x => wertvon(x) = k) l
   in length ws
   end;

(* Eigentlich ist es nicht nötig, die gefilterte Liste in einer
   Variablen zu speichern, wir können den Wert auch direkt an
   length übergeben. *)

fun nwerte3 (l, k)
 = length (filter (fn x => wertvon(x) = k) l);

(* Statt den Wert einzusetzen kann man auch die Funktion length mit
   dem Filter komponieren. *)

fun nwerte4 (l, k)
 = (length o (filter (fn x => wertvon(x) = k))) l;


(* Fuer die Teilaufgabe paar und doppelpaar definieren wir allgemeiner
   eine Funktion, die entscheidet, ob es einen Wert gibt, der n-mal
   vorkommt.  Wir ueberpruefen also, ob es einen Wert w aus werte
   gibt, der mindestens n-mal in der Liste der Karten enthalten
   ist. *)
```

```
fun tupel n l = exists (fn x => nwerte(l, x) >= n) werte;

(* Die oben definierte Funktion tupel hat Funktionstyp int -> ...,
   wobei das erste Argument die Mindestanzahl der Vorkommen ist.  Wir
   koennen tupel also partiell instantiieren: *)

val paar = tupel 2;
val drilling = tupel 3;


(* c *)

(* Um entscheiden zu koennen, ob es zwei _verschiedene_ Werte gibt,
   definieren wir zunaechst eine Funktion, die alle Paare von
   verschiedenen Werten berechnet.  Die Grundidee hinter der
   Implementierung ist folgende:

   Für die leere und einelementige Liste xs ist comb(xs) = [].  Wenn
   eine Liste die Form x::xs mit xs != [] hat dann erhält man
   comb(x::xs) indem man alle Paare der Form (x,y) bzw. (y,x) mit y
   aus xs bildet und diese Paare mit comb(xs) konkateniert. *)

fun pairlist x [] = []
  | pairlist x (y::ys) = (x,y)::(y,x)::(pairlist x ys);

fun comb [] = []
  | comb (x::xs) = (pairlist x xs)  @ comb(xs);

(* Variante von comb ohne Verwendung einer Hilfsfunktion. *)

fun comb' (nil) = nil
 |  comb' (x::xs) = (foldr (fn(y, z) => (x, y)::(y, x)::z) nil xs) @ comb'(xs);

(* Allgemein suchen wir Kombinationen von Werten, wobei der erste Wert
   mindestens n-mal und der zweite mindestens m-mal vorkommt. *)

fun paare (n, m) l =
  exists
    (fn x => (nwerte(l, #1 x) >=n andalso nwerte(l, #2 x) >= m)) (comb werte);

(* Da die Funktion paare einen Funktionstyp hat, koennen wir sie wie
   oben teilweise instantiieren: *)

val doppelpaar = paare(2, 2);
val fullhouse = paare(2, 3);


(* d *)
(* Bei einer Liste von Karten handelt es sich um einen Flush, wenn es
   eine Farbe f gibt, so dass fuer alle Karten k der Liste gilt: Die
```

```
   Farbe von k ist f. Diese Idee setzen wir mit den Funktionen
   all und exists um.  *)

fun flush l =
  exists (fn c => (all (fn k => farbevon(k) = c) l)) farben;

(* Alternativ kann man auch ueberpruefen, ob die Farbe der Karten der
   Restliste gleich der Farbe der ersten Karte ist: *)

fun flush' (nil) = true
 |  flush' (x::xs) = all(fn z => farbevon(z)  = farbevon(x)) xs;
```

**Aufgabe 8-3** **Sortierte Liste** (2 Punkte)

Wir berücksichtigen zwei Spezialfälle: Die leere Liste sowie Listen mit einem Element sind immer sortiert.

```
fun ist_sortiert [] = true
  | ist_sortiert [x] = true
  | ist_sortiert (x :: y :: zs)
      = (x <= y) andalso ist_sortiert(y :: zs);

fun test_ist_sortiert ()
  = ist_sortiert []
      andalso ist_sortiert [1]
      andalso ist_sortiert [1, 2]
      andalso ist_sortiert [1, 2, 3]
      andalso ist_sortiert [1, 2, 3, 7, 23, 123]
      andalso ist_sortiert [~1000, ~20, 0, 1, 2, 3, 7, 23, 123]
      andalso not (ist_sortiert [2, 0])
      andalso not (ist_sortiert [1, 2, 0, 3]);
```

**Aufgabe 8-4** **Fahrzeugvermietung** (1+1+1+2+2+3 Punkte)

Untenstehender SML-Code enthält die Lösung zu allen Teilaufgaben:

```
(* Wir benutzen Existenz- und Allquantor sowie filter, muessen sie
   deshalb defninieren. Dies tun wir durch die gleichnamigen
   Funktionen aus der Standardbibliothek. *)

val all = List.all;
val exists = List.exists;
val filter = List.filter;

(* a *)
datatype kategorie = A | B | C | D;
datatype fahrzeug = Auto of int * bool * kategorie | Motorrad of int * bool;

(* und ein paar Elemente des Datentyps zum testen *)
```

```
val auto_1 = Auto(11, true, B);
val auto_2 = Auto(22, false, D);
val motorrad_1 = Motorrad(1, true);
val motorrad_2 = Motorrad(2, false);

val vs = [auto_1, auto_2, motorrad_1, motorrad_2];

(* b *)
fun ist_vermietet (Auto(_, vermietet, _)) = vermietet
  | ist_vermietet (Motorrad(_, vermietet)) = vermietet;

(* c *)
val ein_fahrzeug_vermietet  = exists ist_vermietet ;

val alle_fahrzeuge_vermietet  = all ist_vermietet ;

(* d *)
fun auto (Auto(_, _, _)) = true
  | auto _ = false;

val ein_auto = exists auto ;

val nur_autos  = all auto ;

(* e *)
fun upgrade_auto (Auto(nr, vermietet, B)) = Auto(nr, vermietet, C)
  | upgrade_auto f = f;

val upgrade = map upgrade_auto;


(* f *)
fun preis (Auto(_, _, A)) = 46
  | preis (Auto(_, _, B)) = 52
  | preis (Auto(_, _, C)) = 64
  | preis (Auto(_, _, D)) = 87
  | preis (Motorrad(_, _)) = 53;


(* g *)
fun tageseinnahmen [] = 0
  | tageseinnahmen (x :: xs) = if ist_vermietet(x)
       then preis(x) + tageseinnahmen(xs)
       else tageseinnahmen(xs);

fun vermietete_fahrzeuge [] = []
  | vermietete_fahrzeuge (f :: fs) = if ist_vermietet(f)
     then f :: (vermietete_fahrzeuge fs)
     else (vermietete_fahrzeuge fs);
```

```
fun tageseinnahmen1 fs
  = let val vs = vermietete_fahrzeuge fs
fun t [] = 0
  | t (x :: xs) = preis(x) + t(xs)
    in t vs
    end;

fun preisliste [] = []
  | preisliste (x :: xs) = preis(x) :: (preisliste xs);


fun tageseinnahmen2 fs
  = let val vs = vermietete_fahrzeuge fs
val ps = preisliste vs
fun t [] = 0
  | t (x :: xs) = x + t(xs)
    in t ps
    end;

fun tageseinnahmen3 fs
  = let val vs = filter ist_vermietet fs
val ps = map preis vs
fun t [] = 0
  | t (x :: xs) = x + t(xs)
    in t ps
    end;

fun tageseinnahmen4 fs
  = let fun t [] = 0
  | t (x :: xs) = x + t(xs)
    in t (map preis (filter ist_vermietet fs))
    end;

fun tageseinnahmen5 fs
 = let fun t [] = 0
  | t (x :: xs) = x + t(xs)
   in (t o (map preis) o (filter ist_vermietet)) fs
   end;

fun tageseinnahmen6 fs
  = ((foldl (op +) 0) o (map preis) o (filter ist_vermietet)) fs;

val tageseinnahmen7  = (foldl (op +) 0) o (map preis) o (filter ist_vermietet);
```

Eine andere Möglichkeit ist die Typen in der folgenden Form zu definieren:

```
(* Wir benutzen Existenz- und Allquantor sowie filter, muessen sie
   deshalb defninieren. Dies tun wir durch die gleichnamigen
   Funktionen aus der Standardbibliothek. *)
```

```
val all = List.all;
val exists = List.exists;
val filter = List.filter;

(* a *)

(* In dieser Version der Lösung wird nur eine einzige Variante für den
   Typ Fahrzeug definiert, allerdings enthält diese Variante jetzt ein
   Feld für den Fahrzeugtyp.  Die nur für manche Fahrzeugtypen
   benötigten Informationen werden im Variantentyp "Fahrzeugtyp"
   codiert. *)

datatype kategorie = A | B | C | D;
datatype fahrzeugtyp = Auto of kategorie | Motorrad;
datatype fahrzeug = KFZ of fahrzeugtyp * int * bool

(* und ein paar Elemente des Datentyps zum testen *)
val auto_1 = KFZ(Auto(B), 11, true);
val auto_2 = KFZ(Auto(D), 22, false);
val motorrad_1 = KFZ(Motorrad, 1, true);
val motorrad_2 = KFZ(Motorrad, 2, false);

val vs = [auto_1, auto_2, motorrad_1, motorrad_2];

(* b *)
fun ist_vermietet (KFZ(Auto(_), _, vermietet)) = vermietet
  | ist_vermietet (KFZ(Motorrad, _, vermietet)) = vermietet;

(* c *)
val ein_fahrzeug_vermietet  = exists ist_vermietet ;

val alle_fahrzeuge_vermietet  = all ist_vermietet ;

(* d *)
fun auto (KFZ(Auto(_), _, _)) = true
  | auto _ = false;

val ein_auto = exists auto ;

val nur_autos  = all auto ;

(* e *)
fun upgrade_auto (KFZ(Auto(B), nr, vermietet)) = KFZ(Auto(C), nr, vermietet)
  | upgrade_auto f = f;

val upgrade = map upgrade_auto;


(* f *)
```

```
fun preis (KFZ(Auto(A), _, _)) = 46
  | preis (KFZ(Auto(B), _, _)) = 52
  | preis (KFZ(Auto(C), _, _)) = 64
  | preis (KFZ(Auto(D), _, _)) = 87
  | preis (KFZ(Motorrad, _, _)) = 53;


(* g *)
val tageseinnahmen  = (foldr (op +) 0) o (map preis) o (filter ist_vermietet);
```