

Übungen zu Informatik I (Lösungsvorschlag)

Aufgabe 8-1

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

Statt die Funktion *foldl* partiell anzuwenden kann man auch ein explizites Argument für die Liste angeben:

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

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

Fahrzeugvermietung

(keine Abgabe)

Untenstehender SML-Code enthält die Lösung zu allen Teilaufgaben:

```
(* Wir benutzen Existenz- und Allquantor sowie filter, muessen sie  
deshalb definieren. 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)) = 46.0
  | preis (Auto(_, _, B)) = 52.0
  | preis (Auto(_, _, C)) = 64.0
  | preis (Auto(_, _, D)) = 87.0
  | preis (Motorrad(_, _)) = 53.0;
```

```
(* g *)
val tageseinnahmen = (foldr (op +) 0.0) o (map preis) o (filter ist_vermietet);
```

Aufgabe 8-3 Kartenspiel mit Datentypen (2+3+5+2 Punkte)

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 Enumerationstyp *)
```

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

(* Fuer die Teilaufgabe paar und doppel paar 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. *)

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 doppel paar = 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;
```