RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2

Prof. Dr. Jürgen Giesl
Darius Dlugosz, Thomas v. d. Maßen, Antje Nowack

Übung *Informatik I - Programmierung* - Blatt 14

(Lösungsvorschlag)

Aufgabe 1

(a) Die Funktion `h` erwartet zwei Argumente, die an die Funktion `g` übergeben werden. `g` ist vom Typ `Int -> Int -> Int`, weshalb die Argumente vom Typ `Int` sein müssen. Das Ergebnis von `(g x y)` vom Typ `Int`, wird an die Funktion `f` übergeben, die ein `Int`-Wert zurückliefert. Das ist auch das Ergebnis der Funktion `h`. Somit ist `Int -> Int -> Int` der Typ der Funktion `h`.

(b) (i) falsch, da
`comp f g = \x -> f (g x)` und `Int -> Int` der Typ von `(g x)` ist, während `f` einen Wert vom Typ `Int` erwartet. Damit ist `comp f g` typinkorrekt. (`h` hat den Typ `Int -> Int -> Int`)

(ii) richtig, da
`comp f (g x) = \z -> f ((g x) z)`[1] und dies `h x` entspricht. (Es gilt `((g x) z) = (g x z)`.)

(iii) [2] falsch (folgt aus (i)), da
`(comp f g) x y = (\z -> f (g z)) x y`[3] und `Int -> Int` der Typ von `(g z)` ist, während `f` einen Wert vom Typ `Int` erwartet. Damit ist `(comp f g)` typinkorrekt. (Die Terme in (i) wurden lediglich um die Parameter `x` und `y` erweitert.)

(iv) [4] richtig (folgt aus (ii)), da die Terme in (ii) lediglich um den Parameter `y` erweitert worden sind.

(v) falsch, da in `comp f (g x y)` der Typ von `(g x y)` `Int` ist, während `comp` ein Argument mit einem Funktionstyp erwartet.
[5]

[1] Die Variable `x` in der Definition von `comp` wurde in `z` umbenannt.
[2] !!! hier sollte man Folgefehler berücksichtigen, wenn jemand (i) falsch gelöst hat und hier mit (i) argumentiert
[3] Die Variable `x` in der Definition von `comp` wurde in `z` umbenannt.
[4] !!! hier sollte man Folgefehler berücksichtigen, wenn jemand (ii) falsch gelöst hat und hier mit (ii) argumentiert
[5] !!! Es gibt nicht nur diese Argumente für die Typinkorrektheit, z.B. kann man für `comp f (g x y) = lambda z -> f ((g x y) z)` in (v) sagen, dass `((g x y) z)` Typinkorrekt ist, da ein `Int`-Wert nicht auf einen `Int`-Wert angewendet werden kann.

Aufgabe 2

(a) Die letzte Ziffer einer Zahl erhält man durch Rechnung mit modulo 10. Sie wird dann durch Ganzzahldivision mit 10 "abgeschnitten". Zur Berechnung der Quersumme von negativen Zahlen, negieren wir die Zahl.

```
intqsum :: Int -> Int
intqsum 0 = 0
intqsum x
 | x < 0     = intqsum (-x)
 | otherwise = (mod x 10) + (intqsum (div x 10))
```

(b) Wir wandeln die Liste der Zeichen (String) in eine Liste von (entsprechenden) Zahlen um, und zählen anschliessend diese zusammen.

```
strqsum :: String -> Int
strqsum s = listsum (map ord s)
  where
  listsum :: [Int] -> Int   -- summiert die Zahlen einer Liste
  listsum []     = 0
  listsum (x:xs) = x + listsum xs
```

(c)
```
bmap :: (a -> b) -> Baum a -> Baum b
bmap f (Knoten x []) = Knoten (f x) []
bmap f (Knoten x b)  = Knoten (f x) (map (bmap f) b)
```

Aufgabe 3

(a) Eine Datenbank ist entweder leer oder sie enthält einen Eintrag, dem weitere Einträge folgen. Ein Eintrag besteht aus einem Wert vom Typ a und der zugehörigen Reihe mit Werten vom Typ b. Die Reihe wird mit Hilfe einer Liste realisiert.

Man erhält insgesamt folgende Datenstrukturdeklaration.

```
data Hash a b = Empty | Entry a [b] (Hash a b)
```

(b)
```
insert :: a -> b -> Hash a b -> Hash a b
{- insert:
 - Fuegt einen neuen Eintrag, der aus einem Element e vom Typ a und dem
 - zugehoerigen Wert v vom Typ b besteht, in einer Hash-DB ein.
 - Existiert bereits ein Eintrag e mit dem Wert v, dann bleibt die Hash-DB
 - unveraendert. Ist ein Eintrag mit dem Element e vorhanden ohne Wert v,
```

```
 - dann wird die Liste der zu e zugeordneten Werte um v erweitert.
 - Ansonsten wird die Hash-DB um die "Spalte" e mit dem Wert v erweitert.
 -}
insert e v Empty = Entry e [v] Empty
insert e v (Entry x l h)
 | e == x    = Entry x (insert2list v l) h
 | otherwise = Entry x l (insert e v h)
   where
   insert2list v [] = [v]
   insert2list v (x:xs)
     | v == x    = (x:xs)
     | otherwise = x : (insert2list v xs)
```

Alternative Version, die nicht berücksichtigt, ob ein der Wert `v` bereits zu `e` existiert, und `v` zu der Liste der zugeordneten Werten immer hinzufügt. Welche Version man implementiert, war in der Aufgabenstellung freigestellt.

```
insert2 :: (Eq a, Eq b) => a -> b -> Hash a b -> Hash a b
insert2 e v Empty = Entry e [v] Empty
insert2 e v (Entry x l h)
 | e == x    = Entry x (v:l) h
 | otherwise = Entry x l (insert2 e v h)
```

Hinweis: Um diese Funktionen in Hugs eingeben zu können erweitert man, die Typdeklaration der Funktion um "`... :: (Eq a, Eq b) => ...`" , z.B. `insert :: (Eq a, Eq b) => a -> b -> Hash a b -> Hash a b`, was bedeutet, dass für a und b nur Typen erlaubt sind, die der Klasse `Eq` (Vergleichbar) angehören (wie z.B. `Int`, `Char`, `String`). Insbesondere sind dann die Vergleichsoperatoren `(==)`, `(/=)` vordefiniert. In Analogie zur Java ist `Eq` wie das Interface Vergleichbar mit den Methoden `(==)` und `(/=)`.

(c)
```
hmap :: (b -> c) -> Hash a b -> Hash a c
{- hmap:
 - Wendet auf die Werte, die einem Schluessel zugeordnet sind,
 - eine Funktion an.
 -}
hmap f Empty = Empty
hmap f (Entry x l h) = Entry x (map f l) (hmap f h)
```

Aufgabe 4

(a)
```
auf(8,5).
auf(5,1).
auf(1,tisch).
```

```
auf(6,7).
auf(7,tisch).
auf(2,4).
auf(4,9).
auf(9,3).
auf(3, tisch).
```

(b)
```
ueber(X ,Y) :- auf(X, Y).
ueber(X, Y) :- auf(X, A), ueber(A, Y).
```

(c) Hinter "`?-`" wird die geforderte Anfrage gestellt. Die Zeilen darunter geben die Lösungen an.

(Steht Block 8 über Block 1?)
```
?- ueber(8,1).
Yes.
```

(Steht Block 2 über Block 7?)
```
?- ueber(2,7).
No.
```

(Was steht über Block 3?)
```
?- ueber(X,3).
X = 9 ;
X = 2 ;
X = 4 ;
No.
```
(Keine weitere Lösungen.)

(Was steht unter Block 4?)
```
?- ueber(4,X).
X = 9 ;
X = 2 ;
X = tisch ;
No.
```
(Keine weitere Lösungen.)

Aufgabe 5

(a)
```
liebt(peter, susi).    % Peter liebt Susi.
liebt(hans, susi).     % Hans liebt Susi.
liebt(hans, sabine).   % Hans liebt Sabine.
liebt(sabine, peter).  % Sabine liebt Peter.
liebt(susi, peter).    % Susi liebt Peter.
liebt(susi, felix).    % Susi liebt Felix.
liebt(X,X).            % Jeder liebt sich selbst.

hasst(sabine, hans).   % Sabine hasst Hans.
```

```
hasst(susi, sabine).   % Susi hasst Sabine.

pechvogel(X) :- liebt(X, A), hasst(A, X). % Jemand ist ein Pechvogel,
                                          % wenn seine Liebe mit Hass
                                          % vergolten wird.
```

(b) Hinter "`?-`" wird die geforderte Anfrage gestellt. Die Zeilen darunter geben die Lösungen an.

(Liebt Peter Susi?)
```
?- liebt(peter, susi).
Yes.
```

(Liebt Susi Felix?)
```
?- liebt(susi, felix).
Yes.
```

(Wenn liebt Sabine?)
```
?- liebt(sabine, X).
X = peter ;
X = sabine ;
No.
```
(Keine weitere Lösungen.)

(Wer liebt Sabine?)
```
?- liebt(X, sabine).
X = hans ;
X = sabine ;
No.
```
(Keine weitere Lösungen.)

(Wer liebt jemanden, der ihn auch liebt?)
```
?- liebt(X, Y), liebt(Y, X).
X = peter
Y = susi ;
X = susi
Y = peter ;
No.
```
(Keine weitere Lösungen. Damit lieben sich Susi und Peter gegenseitig.)

(Wer liebt einen Pechvogel?)
```
?- liebt(X, Y), pechvogel(Y).
X = hans
Y = hans ;
No.
```
(Keine weitere Lösungen. Damit liebt Hans einen Pechvogel, da er sich selbst liebt.)

(c)
```
befreundet(X, Y) :- liebt(X, Y).
befreundet(X, Y) :- liebt(Y, X).
befreundet(X, Y) :- befreundet(A, X), befreundet(A, Y).
```