194.026 Funktionale Programmierung WS 23

— Test 3 —

Freitag, 07.06.2024, 10:00–12:00 Uhr

Nachname	Vorname	Matrikel-nummer	Hörsaal	Platz-nummer	Unterschrift
			FH HS1		

Aufgabe	1	2	3	4	5	Summe Punkte
Punkte	21	22	18	18	21	100
Erreichte Punkte						

...schreiben Sie bitte auf jedes Blatt, das Sie abgeben, Ihren Namen und Ihre Matrikelnummer!

Aufgabe 1 (2*0.5+14+6 = 21 Punkte)

Jedes Element in einer Liste hat entweder ein oder zwei direkte Nachbarelemente.

Ein Element einer Liste heißt *Senke* genau dann, wenn es echt kleiner als alle seine direkten Nachbarelemente ist.

1. Gegeben sind die Listen `lst1` und `lst2` ganzer Zahlen:

   (a) `lst1 = [8,6,12,4,4,17,3,4,5,4]`

   (b) `lst2 = [17,20,25,0,5,-12,-10,-11,-12]`

   Markieren Sie (durch Einkreisen) alle Senken in `lst1` und `lst2`.

2. Schreiben Sie so typallgemein wie möglich eine Haskell-Rechenvorschrift **`senke`**, die angewendet auf eine Liste `lst` von Elementen die aufsteigend geordnete Liste der Positionen liefert, an denen sich in `lst` eine Senke befindet. Dabei gilt: das erste Element einer Liste steht an Position 0, das zweite Element an Position 1 usw. Geben Sie auch die Typsignatur von **`senke`** an.

```
type Nat0     = Int
type Position = Nat0

senke :: ... -> [Position]
```

3. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Implementierung von **`senke`** vorgeht und warum sie so typallgemein wie möglich implementiert ist.

Aufgabe 2 (12+5+2+3 = 22 Punkte)

Gegeben ist folgender Typ für Listen:

```
data Liste a = L                    -- L für 'Leer'
             | a 'V' (Liste a)      -- V für 'Verknüpfe'
```

1. Schreiben Sie eine Funktion **`verflechte`**:

```
verflechte :: (Liste a) -> (Liste a) -> (Liste a)
```

   die die Werte zweier Listen miteinander verflechtet: angewendet auf zwei Listen, liefert **`verflechte`** eine Liste als Resultat, die die Elemente der beiden Argumentlisten der Reihe nach abwechselnd enthält: die Elemente der ersten Liste stehen in der Resultatliste an den geraden Positionen, die der zweiten an den ungeraden Positionen. Das erste Element einer Liste steht dabei an Position 0, das zweite Element an Position 1 usw.

   Sind die Argumentlisten ungleich lang, bildet der nichtverflochtene Rest der längeren Argumentliste das Endstück der Resultatliste.

2. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Implementierung von **`verflechte`** vorgeht.

3. Warum ist die Schreibweise `a 'V' (Liste a)` möglich?

4. Geben Sie zwei Werte

   (a) `lst1 = ... :: Liste Int`

   (b) `lst2 = ... :: Liste Int`

   mit 2 bzw. 3 Elementen an und den Wert des Aufrufs:

   (c) `verflechte lst1 lst2 ->> ...`

Aufgabe 3 ((3*2+3)+(3*2+3) = 18 Punkte)

1. Gegeben ist:

```
f :: Int -> Int
f n
 | n > 0 = n + f (n-1)
 | True  = n
```

   Womit muss `f` jeweils aufgerufen werden, damit folgende Ausgaben beobachtet werden:

   (a) `f ... ->> 15`

   (b) `f ... ->> 5050`

   (c) `f ... ->> -5050`

   Geben Sie jeweils ein passendes Argument an. Erklären Sie zusätzlich, was `f` leistet, welche Aufgabe es erfüllt.

2. Gegeben sind:

```
g :: [a] -> [a]
g []     = []
g (x:xs) = h (g xs) x

h :: [a] -> a -> [a]
h [] y     = [y]
h (x:xs) y = x : (h xs y)
```

   Womit muss `g` jeweils aufgerufen werden, damit folgende Ausgaben beobachtet werden:

   (a) `g ... ->> [1]`

   (b) `g ... ->> [3,3,3]`

   (c) `g ... ->> 'die Liste der Ziffern Ihrer Matrikelnummer'`

   Geben Sie jeweils ein passendes Argument an. Erklären Sie zusätzlich, was `g` und `h` leisten, welche Aufgabe sie erfüllen.

Aufgabe 4 (6+6+6 = 18 Punkte)

Richtig oder falsch? Begründen Sie Ihre Antwort jeweils.

1. Die Funktion `f`:

```
f :: Ord a => [a] -> a -> [a]
f [] _ = []
f (x:xs) y
 | x < y = x : f xs
 | True  = f xs
```

   ist direkt *ad hoc* polymorph.

2. Die Deklaration `g`:

```
g :: (Ord a, Num a) => [a] -> a -> a
g [] _      = error "Argumentfehler"
g (x:[]) y = if (x > y) || (x == y) then x else y
g (x:xs) y = if (x > y) || (x == y) then x + g xs y else y + g xs y
```

   genügt dem *Hauptleitsatz funktionaler Programmierung.*

3. Anonyme λ-Abstraktionen sind stets nichtrekursiv.

Aufgabe 5 (14+7 = 21 Punkte)

Gegeben sind die Typen `Rat` und `Rat'` zur Modellierung rationaler Zahlen:

```
type Nat0 = Int
type Nat1 = Int

type Zaehler = Nat0
type Nenner  = Nat1

data ZN     = Z      -- Z für Zähler
            | N      -- N für Nenner
newtype Rat = R (ZN -> Int)

data Vorzeichen = P     -- P für plus
                | M     -- M für minus

data Rat' = R' { vorzeichen :: Vorzeichen,
                 zaehler    :: Zaehler,
                 nenner     :: Nenner
               }
```

1. Implementieren Sie die Funktion `durch` zur Division zweier rationaler Zahlen:

   ```
   data Either a b = Left a | Right b
   data Select     = Links  | Rechts

   durch :: Rat -> Rat' -> Select -> Either Rat Rat'
   ```

   Der Dividend von `durch` liegt in `Rat`-Darstellung vor, der Divisor in `Rat'`-Darstellung. Über den Wert des `Select`-Arguments wird gesteuert, ob das Ergebnis der Division in `Rat`- oder in `Rat'`-Darstellung als `Left`- bzw. `Right`-Wert des Typs `Either Rat Rat'` geliefert wird. Das Ergebnis der Division muss nicht gekürzt sein. Ist eines der Argumente von `durch` nicht wohldefiniert, soll eine transparente Fehlerbehandlung erfolgen. Ergänzen Sie, wo nötig, `deriving`-Klauseln.

2. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Implementierung von `durch` vorgeht und warum Ihre Fehlerbehandlung transparent ist.