universität innsbruck

Funktionale Programmierung	WS 2025/2026	LVA 703025
Übungsblatt 9, 10 Punkte	Abgabefrist: Mittwoch, 10. Dezember 2025, 6 Uhr	

- Kreuzen Sie gelöste Aufgaben im OLAT Kurs des Proseminars an.
- Lösen Sie die Programmieraufgaben: Und laden Sie die Dateien `DList_09.hs`, `OList_09.hs`, `Tests_09.hs` einzeln in OLAT hoch.

Aufgabe 1 *Sichtbarkeit von Variablen und Funktionen* **3 P.**

In den folgenden Übungen geht es um die Sichtbarkeit von Variablen und Funktionen.

1. Analysieren Sie die Sichtbarkeit von `radius` in den drei Funktionen `operationA`, `operationB` und `operationC` aus dem Haskell-Programm unten. Geben Sie auch an, auf welches `radius` (global oder lokal) sich die jeweilige Funktion bezieht und begründen Sie ihre Antworten. (1 Punkt)

```haskell
radius :: Double
radius = 10  -- global radius

computeVolume :: Double -> Double
computeVolume rad = (4/3)*pi*rad^3

operationA :: Double -> Double
operationA radius = computeVolume radius

operationB :: Double
operationB  = computeVolume radius

operationC :: Double -> Double
operationC = computeVolume
```

2. Analysieren Sie die Implementierung von `reverseList`. Funktioniert die Funktion wie erwartet? Führen Sie eine Variablenumbenennung entsprechend der Folien von Woche 9 durch. (1 Punkt)

```haskell
reverseList :: [a] -> [a]
reverseList xs =
    let reverseListAux xs ys = case xs of
            (x:xs) -> reverseListAux xs (x:ys)
            _ -> ys
    in reverseListAux xs []
```

3. Wir betrachten das folgende Programm und die Funktion `squareRootTwo`, welche die Wurzel von 2 basierend auf einer anfänglichen Schätzung über n Iterationen approximiert. Funktionieren `squareRootTwoA` und `squareRootTwoB` wie erwartet? Begründen Sie Ihre Antworten. (1 Punkt)

```haskell
squareRootTwo :: Double -> Integer -> Double
squareRootTwo guess n
    | n == 0 = guess
    | otherwise = squareRootTwo ((guess + 2/guess) / 2) (n-1)

squareRootTwoA :: Double -> Integer -> Double
squareRootTwoA guess n
    | n == 0 = guess
    | otherwise = squareRootTwoA ((guess + 2/guess) / 2) (n-1) where n=n

squareRootTwoB :: Double -> Integer -> Double
squareRootTwoB guess n
    | n == 0 = guess
    | otherwise = let n = n-1 in squareRootTwoB ((guess + 2/guess) / 2) n
```

Aufgabe 2 *Mengen* **7 P.**

In der Programmierung muss man oft *Mengen* von Elementen speichern, d. h. mathematische Mengen, in denen $\{1,3\} = \{1,1,3\}$ gilt, da keine Duplikate gezählt werden. Es gibt mehrere Möglichkeiten, Mengen zu implementieren, und in dieser Übung betrachten wir zwei Varianten.

1. Eine mögliche Implementierung stellt Mengen durch *Duplikat-freie Listen* dar, d. h. Listen, in denen kein Element mehr als einmal vorkommt. Beispielsweise sind sowohl `[1,3]` als auch `[3,1]` gültige Darstellungen der Menge $\{1,3\}$, `[1,1,3]` hingegen nicht.

   Betrachten Sie den folgenden Haskell-Code in der Datei `DList_09.hs`, der Mengen als Duplikat-freie Listen implementiert. Hier entsprechen die Operationen `empty`, `member`, `insert` und `remove` der leeren Menge, der Mitgliedschaftsprüfung, dem Einfügen eines einzelnen Elements und dem Entfernen eines einzelnen Elements. Für Letzteres ist zu beachten, dass $A \setminus \{a\} = A$ gilt, wenn $a \notin A$.

   ```haskell
   data DList a = DList [a]

   empty = DList []

   member x (DList xs) = x `elem` xs

   insert x a@(DList xs)
     | member x a = a
     | otherwise = DList $ x : xs

   remove x a@(DList xs) = case span (/= x) xs of
     (_, []) -> a
     (first, _ : last) -> DList $ first ++ last
   ```

   Identifizieren Sie die Teile der Implementierung, die davon abhängen, dass die Listen Duplikat-frei sind, d. h. die bei Listen mit Duplikaten falsche Ergebnisse liefern können.

   Fügen Sie außerdem eine sinnvolle Moduldeklaration hinzu, sodass ein externer Benutzer auf alle Mengenoperationen zugreifen kann, aber keine Elemente vom Typ `DList a` erstellen kann, die gegen die Duplikat-Freiheit verstoßen. (1 Punkt)

2. Machen Sie `DList` zu einer Instanz von `Show`.

   Beispiel:

   ```haskell
   show $ foldr insert empty [1..5] = "{1, 2, 3, 4, 5}"
   show $ foldr insert empty "abcba" = "{'c', 'b', 'a'}"
   show $ foldr insert empty [] = "{}"
   ```

   Ihre Implementierung muss auf einer Variante von `fold` basieren, d. h. explizite Rekursion ist nicht zulässig. (1 Punkt)

3. Machen Sie `DList` zu einer Instanz von `Eq`.

   Beispiel:

```
foldr insert empty ([1..5] ++ [2..4]) == foldr insert empty [5,4..1]
```

Versuchen Sie, die Duplikat-Freiheit in Ihrer Implementierung auszunutzen. (1 Punkt)

4. Stellen Sie eine alternative Implementierung von Mengen bereit, die auf *geordneten Listen* basiert, d. h. Listen $[x_1, \ldots, x_n]$ mit der Eigenschaft $x_i < x_{i+1}$ für alle $1 \leq i < n$. Kopieren Sie dazu die aktuelle Implementierung in eine neue Datei `OList_09.hs` für geordnete Listen, ersetzen Sie den Modulnamen `DList_09` durch `OList_09`, benennen Sie den Typ `DList` in `OList` um und passen Sie die Implementierung so an, dass sie nun die neue Invariante der Ordnung berücksichtigt. (2 Punkte)

5. Schreiben Sie Ihre eigenen Tests. Ändern Sie `Tests_09.hs` so, dass es Tests enthält, mit denen überprüft wird, ob die Eigenschaft

   ```
   foldr insert empty (xs ++ xs) == foldr insert empty (reverse xs)
   ```

   für beliebige Listen `xs` und *für beide Set-Implementierungen* erfüllt ist. Neben der oben genannten Eigenschaft sollten Sie auch einen Test für eine weitere Eigenschaft hinzufügen, die Sie selbst definieren. Die Eigenschaft sollte eine Beziehung zwischen `remove` und `insert` ausdrücken und kann sich auch auf einige andere Funktionen beziehen.

   Beachten Sie, dass eine Testfunktion beliebige Argumente vom Typ `Int`, `[Int]` und `Bool` haben kann, aber sie darf weder `DList` noch `OList` als Argumenttyp verwenden. Außerdem muss der Rückgabetyp `Bool` sein. (2 Punkte)