Algorithmieren und Programmieren

Sommersemester 2019

Prof. Dr. rer. nat. habil. Petra Hofstedt
Sven Löffler M. Sc.
Sonja Breuß, Deborah Buckenauer, Johannes Kuhn, Julius Schöning, Carlo Bückert

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Übungsblatt 7

Abgabedatum: 05.06.2019

Hinweise

- Beachten Sie, dass die Tutoren unkommentierte Programme nicht als Lösung akzeptieren, selbst wenn die Programme richtig funktionieren. Zu einer richtigen Lösung gehören immer sinnvolle Kommentare, deren Umfang der Komplexität des Programms angemessen ist.
- **Halten Sie sich an die in den Übungsblättern vorgegebenen Namen von Funktionen und Dateien, Funktionstypen (Typsignaturen), Reihenfolge der Parameter und verwenden Sie - sofern vorhanden - die Vorgaben!**
- Auf den Übungsblättern finden Sie einige Haskell-Quelltextfragmente. Diese sind der besseren Lesbarkeit wegen unter Nutzung einiger mathematischer Sonderzeichen wiedergegeben.
- Für diese Veranstaltung wird die Verwendung der ***Haskell-Plattform*** (`https://www.haskell.org/platform/`) empfohlen.
- Als **Tutoriumsaufgabe** markierte (Teil-)aufgaben werden in den Übungen ausführlicher besprochen. Die schriftliche bzw. elektronischen Lösungen müssen jedoch trotzdem mit abgegeben werden. Bitte schauen Sie Sich diese Aufgaben im Vorfeld der Übung an und bereiten Sie Sich darauf vor.
- Die Abgabe Ihrer Lösungen erfolgt vor Ablauf der Abgabefrist digital über die Moodle-Plattform an Ihren Tutor. Erstellen Sie dazu ein PDF-Dokument, das die Lösungen Ihrer schriftlichen Aufgaben enthält. Laden Sie dieses PDF-Dokument und die erzeugten Hakell-Datein, mit den in den Aufgaben vorgegebenen Namen, bei Moodle hoch.

Informationsquellen

- Sie finden unter `http://haskell.org/` sehr viele Informationen über die Programmiersprache Haskell. Von besonderem Interesse sind dabei sicherlich die Übersicht über zahlreiche online verfügbare Haskell-Tutorials (`http://haskell.org/haskellwiki/Tutorials`) sowie die Suchmaschine Hoogle (`http://haskell.org/hoogle/`) für die Haskell-API, die Ihnen mit zunehmender Haskell-Erfahrung wertvolle Dienste leisten wird.

Sie können maximal **(9 (+1) Punkte)** mit diesem Übungsblatt erreichen.

Aufgabe 1 (abstrakter Datentyp für Mengen) 3 Punkte

Verwenden Sie für die Abgabe den Dateinamen: `Menge.hs`

Implementieren Sie einen abstrakten Datentyp `Menge`. Verwenden Sie intern die Darstellung von Mengen als absteigend sortierte Listen aus Aufgabe 1 von Aufgabenserie 5. Der abstrakte Datentyp soll folgende Operationen bereit stellen:

- `leer :: Menge el`
  Gibt die leere Menge zurück
- `einfuegen :: (Ord el) ⇒el →Menge el →Menge el`
  Einfügen eines Elements in eine Menge
- `loeschen::(Ord el) ⇒el →Menge el →Menge el`
  Löschen eines Werts aus einer Menge Die Funktion `loeschen` soll die ursprüngliche Menge zurück gegeben, wenn der angegebene Wert nicht Element dieser Menge ist.
- `vereinigung :: (Ord el) ⇒Menge el →Menge el →Menge el`
  Vereinigung zweier Mengen
- `schnitt :: (Ord el) ⇒Menge el →Menge el →Menge el`
  Schnitt zweier Mengen
- `differenz :: (Ord el) ⇒Menge el →Menge el →Menge el`
  Differenz zweier Mengen
- `istLeer :: Menge el →Bool`
  Test, ob eine Menge leer ist
- `istElement :: (Ord el) ⇒el →Menge el →Bool`
  Test, ob ein Wert Element einer Menge ist
- `istTeilmenge :: (Ord el) ⇒Menge el →Menge el →Bool`
  Test, ob eine Menge Teilmenge einer zweiten Menge ist
- `istEchteTeilmenge :: (Ord el) ⇒Menge el →Menge el →Bool`
  Test, ob eine Menge echte Teilmenge einer zweiten Menge ist
- `minimalesElement :: Menge el →el`
  Gibt das minimale Element einer Menge zurück. Abbruch mit einer Fehlermeldung wenn die Menge leer ist.
- `maximalesElement :: Menge el →el`
  Gibt das maximale Element einer Menge zurück. Abbruch mit einer Fehlermeldung wenn die Menge leer ist.

1. **(Tutoriumsaufgabe)** Legen Sie ein Modul `Menge` für den abstrakten Datentyp an. Fügen Sie in dieses Modul die Deklaration des Datentyps `Menge` sowie die Typsignaturen der Mengenoperationen ein. Legen Sie eine geeignete Exportliste an. Es soll außerdem möglich sein, Mengen auf Gleichheit zu testen und in der üblichen Notation $\{x_1, \ldots, x_n\}$ auszugeben. Sorgen Sie für entsprechende Typklasseninstanziierungen.
2. Implementieren Sie die Mengenoperationen. Sie können dazu Ihre Lösung von Aufgabe 1 der Aufgabenserie 5 verwenden.

3. Ermitteln Sie die Aufwandsklassen (O(n)) für Ihre Implementierungen der Mengenoperationen und stellen Sie diesen die Aufwandsklassen der entsprechenden Operationen aus dem Modul `Data.Set` in einer Tabelle gegenüber. Letztere finden Sie in der API-Dokumentation von Data.Set.

Aufgabe 2 (Funktionen höherer Ordnung programmieren) 3 (+0,5) Punkte

In dieser Aufgabe sollen einige häufig verwendete Funktionen höherer Ordnung selber implementiert werden.

Für jeweils 5 richtig gelöste Aufgaben erhalten Sie einen Punkt.

Verwenden Sie für die Abgabe den Dateinamen: `Hof.hs`

Mit der folgenden Codezeile zu Begin Ihres Haskell-Programmes können Sie die vorgefertigten Methoden verbergen, so dass jeweils nur die von Ihnen programmierten aufgerufen werden können.

```
import Prelude hiding (map, any, all, iterate, filter, takeWhile, dropWhile, zipWith, curry,
uncurry, (.), foldr, foldl, foldl1, foldr1, flip)
```

1. **(Tutoriumsaufgabe)** Programmieren Sie eine Funktion `map :: (a →b) →[a] →[b]`, welche eine Funktion und eine Liste bekommt und diese Funktion auf jedes Element in der Liste anwendet.

   ```
   map (2*) [1,2,3,4,5] ~>* [2,4,6,8,10]
   map isLower "Hallo Du" ~>* [False,True,True,True,True,False,False,True]
   ```

2. **(Tutoriumsaufgabe)** Programmieren Sie eine Funktion `any :: (a →Bool) →[a] →Bool`, welche eine Prädikatsfunktion und eine Liste bekommt und `True` ausgibt, wenn ein Element in der Liste existiert, welches das Prädikat erfüllt.

   ```
   any (==2) [1,2,3,4,5] ~>* True
   any (==6) [1,2,3,4,5] ~>* False
   ```

3. Programmieren Sie eine Funktion `all :: (a →Bool) →[a] →Bool`, welche eine Prädikatsfunktion und eine Liste bekommt und `True` ausgibt, wenn alle Elemente der Liste das Prädikat erfüllen.

   ```
   all (even) [1,2,3,4,5] ~>* False
   all (even) [2,4,6,8,10] ~>* True
   ```

4. Programmieren Sie eine Funktion `flip :: (a →b →c) →b →a →c`, welche eine Funktion bekommt sowie die Argumente für die Funktion in umgekehrter Reihenfolge. Die Funktion soll auf ihre Argumente angewendet werden.

   ```
   flip (:) [1,2,3] 0 ~>* [0,1,2,3]
   flip foldl 0 (+) [1,2,3,4,5] ~>* 15
   ```

5. **(Tutoriumsaufgabe)** Programmieren Sie eine Funktion `iterate :: (a →a) →a →[a]`, welche eine Funktion `f` und ein initiales Argument `x` erhält und daraus die unendliche Liste `[x, f x, f (f x),...]` erzeugt.

   ```
   iterate (2*) 1 ~>* [1,2,4,8,16,32,64,128,256,512,...]
   iterate (^2) 2 ~>* [2,4,16,256,65536,...]
   ```

6. Programmieren Sie eine Funktion `takeWhile :: (a →Bool) →[a] →[a]`, welche solange Elemente aus der Liste nimmt, wie das Prädikat erfüllt ist.

```
takeWhile even [2,4,6,7,8,10] ~>* [2,4,6]
takeWhile (<10) [2,4,6,8,10,12] ~>* [2,4,6,8]
```

7. Programmieren Sie eine Funktion **dropWhile** `:: (a →Bool) →[a] →[a]`, welche solange Element aus der Liste verwirft, wie das Prädikat erfüllt ist.

   ```
   dropWhile even [2,4,6,7,8,10] ~>* [7,8,10]
   dropWhile (<10) [2,4,6,8,10,12] ~>* [10,12]
   ```

8. Programmieren Sie eine Funktion **filter** `:: (a →Bool) →[a] →[a]`, welche aus einer Liste alle Elemente entfernt, die das Prädikat nicht erfüllen.

   ```
   filter even [1,2,3,4,5,6,7,8,9,10] ~>* [2,4,6,8,10]
   filter odd [1,2,3,4,5,6,7,8,9,10] ~>* [1,3,5,7,9]
   ```

9. Programmieren Sie eine Funktion **partition** `:: (a →Bool) →[a] →([a], [a])`, welche eine Liste in zwei Listen aufteilt. Die erste Liste soll alle Elemente enthalten, die das Prädikat erfüllen. Die zweite Liste soll alle Elemente enthalten, die das Prädikat nicht erfüllen.

   ```
   partition even [1,2,3,4,5,6,7,8,9,10] ~>* ([2,4,6,8,10],[1,3,5,7,9])
   ```

10. Programmieren Sie eine Funktion **zipWith** `:: (a →b →c) →[a] →[b] →[c]`, welche zwei Listen mit Hilfe der übergebenen Funktion verknüpft. Sind die beiden Eingabelisten unterschiedlich lang, so soll die resultierende Liste so lang sein, wie die kürzere der beiden Eingaben.

    ```
    zipWith (+) [1,2,3,4] [6,7,8,9,10] ~>* [7,9,11,13]
    zipWith (,) [1,2,3,4,5] [6,7,8,9,10] ~>* [(1,6),(2,7),(3,8),(4,9),(5,10)]
    ```

11. Mit dem Operator `.` können zwei Funktion komponiert werden. Die Resultatfunktion entspricht der Hintereinanderausführung der beiden Funktionen.

    Programmieren Sie den Kompositionsoperator

    `(.) :: (b →c) →(a →b) →(a →c),`

    welcher der Funktionskomposition entspricht.

12. In Haskell verwenden fast alle Funktion die Curry-Notation. Das heißt statt `f(a,b)` schreibt man `f a b` um die Funktion `f` auf die beiden Argumente `a` und `b` anzuwenden.

    Schreiben Sie eine Funktion **uncurry** `:: (a →b →c) →(a,b) →c`, welche eine Funktion in der Curry-Notation (also f a b) bekommt und daraus eine Funktion der Form `f (a,b)` macht.

    ```
    uncurry (+) (1,2) ~>* 3
    uncurry (:) (1,[2,3,4,5]) ~>* [1,2,3,4,5]
    ```

13. Programmieren Sie eine Funktion **curry** `:: ((a,b) →c) →a →b →c`, welche die Umkehrfunktion von **uncurry** darstellt.

    ```
    (curry ∘ uncurry) (+) 1 2 ~>* 3
    ```

14. Programmieren Sie eine Funktion

    foldl `:: (acc →el →acc) →acc →[el] →acc,`

    welche die Liste mit der übergebenen Funktion von vorne bzw. von links faltet.

```
foldl (+) 0 [1,2,3,4,5] ~>* 15
foldl (*) 1 [1,2,3,4,5] ~>* 120
foldl (-) 10 [1] ~>* 9
foldl (-) 10 [1,2] ~>* 7
foldl (-) 10 [1,2,3] ~>* 4
```

15. Programmieren Sie eine Funktion

    `foldr :: (el →acc →acc) →acc →[el] →acc,`

    welche die Liste mit der übergebenen Funktion von hinten bzw. von rechts faltet.

```
foldr (+) 0 [1,2,3,4,5] ~>* 15
foldr (*) 1 [1,2,3,4,5] ~>* 120
foldr (-) 10 [1] ~>* -9
foldr (-) 10 [1,2] ~>* 9
foldr (-) 10 [1,2,3] ~>* -8
```

16. **(Zusatz)** Programmieren Sie die Funktionen

    `foldr1 :: (a →a →a) →[a] →a` und

    `foldl1 :: (a →a →a) →[a] →a,`

    welche sich wie die Funktionen **`foldr`** und **`foldl`** verhalten, jedoch das erste Listenelement als initialen Akkumulator verwenden. Die beiden Funktionen dürfen deshalb nicht auf leere Listen angewendet werden.

```
foldr1 (+) [1,2,3,4,5] ~>* 15
foldl1 (*) [1,2,3,4,5] ~>* 120
```

17. **(Zusatz)** Programmieren Sie eine Haskell-Funktion

    `alleAnwenden :: [arg →wert] →arg →[wert]` die jede Funktion einer Funktionsliste auf ein einzelnes Argument anwendet und die Ergebnisse in einer Liste sammelt. Es ergibt sich zum Beispiel `alleAnwenden [(*3),(+1),(div 4)] 3` ⇝ `[9,4,1]`

18. **(Zusatz)** Programmieren Sie eine Haskell-Funktion

    `hintereinanderAnwenden :: [wert →wert] →wert →wert`

    welche ausgehend von einem gegebenen Argument alle Funktionen einer Funktionsliste hintereinander anwendet und das schließlich entstehende Resultat zurück gibt. Es ergibt sich zum Beispiel

    `hintereinanderAnwenden [(*3),(+1),(-) 4] 3` ⇝ `-6`

    da `(*3) 3` ⇝ = 9, `(+1) 9` ⇝ = 10 und schließlich `((-) 4) 10` ⇝ = -6 ist.

Aufgabe 3 (Faltungen) 3 (+0,5) Punkte

Es gibt 1 bzw. 2 bzw. 3 Punkte für 5 bzw. 10 bzw. 14 richtig bearbeitete Teilafugaben.

In dieser Aufgabe sollen unter Verwendung der Funktionen **`foldr`** und **`foldl`** andere Funktionen implementiert werden.

Verwenden Sie für die Abgabe den Dateinamen: `Faltungen.hs`

1. Programmieren Sie die Funktionen

   `andWithFoldr :: [Bool] →Bool` und `andWithFoldl :: [Bool] →Bool`,

   welche sich identisch zu der Funktion **`and`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

2. Programmieren Sie die Funktionen

   `orWithFoldr :: [Bool] →Bool` und `orWithFoldl :: [Bool] →Bool`,

   welche sich identisch zu der Funktion **`or`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

3. Programmieren Sie die Funktionen

   `sumWithFoldr :: (Num a) ⇒[a] →a` und `sumWithFoldl :: (Num a) ⇒[a] →a`,

   welche sich identisch zu der Funktion **`sum`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

4. Programmieren Sie die Funktionen

   `productWithFoldr :: (Num a) ⇒[a] →a` und `productWithFoldl :: (Num a) ⇒[a] →a`,

   welche sich identisch zu der Funktion **`product`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

5. Programmieren Sie die Funktionen

   `concatWithFoldr :: [[a]] →[a]` und `concatWithFoldl :: [[a]] →[a]`,

   welche sich identisch zu der Funktion **`concat`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

6. Programmieren Sie die Funktionen

   `reverseWithFoldr :: [a] →[a]` und `reverseWithFoldl :: [a] →[a]`,

   welche sich identisch zu der Funktion **`reverse`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

7. Programmieren Sie die Funktionen

   `mapWithFoldr :: (a →b) →[a] →[b]` und

   `mapWithFoldl :: (a →b) →[a] →[b]`,

   welche sich identisch zu der Funktion **`map`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

8. Programmieren Sie die zwei Funktionen

   `anyWithFoldr :: (a →Bool) →[a] →Bool` und

   `anyWithFoldl :: (a →Bool) →[a] →Bool`,

   welche sich identisch zu der Funktion **`any`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

9. Programmieren Sie die zwei Funktionen

`allWithFoldr :: (a →Bool) →[a] →Bool` und

`allWithFoldl :: (a →Bool) →[a] →Bool`,

welche sich identisch zu der Funktion **`all`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

10. Programmieren Sie die zwei Funktionen

    `filterWithFoldr :: (a →Bool) →[a] →[a]` und

    `filterWithFoldl :: (a →Bool) →[a] →[a]`,

    welche sich identisch zu der Funktion **`filter`** verhalten. Nutzen für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

11. Programmieren Sie die zwei Funktionen

    `maximumWithFoldr1 :: Ord a ⇒[a] →a` und

    `maximumWithFoldl1 :: Ord a ⇒[a] →a`,

    welche sich identisch zu der Funktion **`maximum`** verhalten. Beachten Sie, dass die Funktionen nur auf eine nicht leere Liste angewendet werden können. Nutzen für die Implementierung entsprechend die Funktionen **`foldr1`** bzw. **`foldl1`**.

12. Programmieren Sie analog zur vorherigen Teilaufgabe die beiden Funktionen

    `minimumWithFoldr1` und `minimumWithFoldl1`.

13. Programmieren Sie eine Funktion

    `appendWithFoldr :: [a] →[a] →[a]` und

    `appendWithFoldl :: [a] →[a] →[a]`,

    welche zwei Listen zusammenhängt, sich also wie der Operator `(++)` verhält. Nutzen Sie für die Implementierung entsprechend die Funktion **`foldr`** bzw. **`foldl`**.

14. Programmieren Sie die Funktionen

    `lengthWithFoldr :: [a] →Int` und `lengthWithFoldl :: [a] →Int`,

    welche sich identisch zur Funktion **`length`** verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.

15. **(Zusatz)** Programmieren Sie die Funktionen

    `hintereinanderAnwendenFoldl :: [wert →wert] →wert →wert` und

    `hintereinanderAnwendenFoldr :: [wert →wert] →wert →wert`

    welche sich identisch zur Funktion `hintereinanderAnwenden` verhalten. Nutzen Sie für die Implementierung entsprechend die Funktionen **`foldr`** bzw. **`foldl`**.