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 5

Abgabedatum: 15.05.2019

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 **(8 Punkte)** mit diesem Übungsblatt erreichen.

Aufgabe 1 (Mengen als Listen) 1,5 Punkte

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

Mengen können in Haskell durch Listen repräsentiert werden. Allerdings unterscheiden sich Mengen und Listen in zwei Punkten:

- Die Elemente einer Liste besitzen eine Reihenfolge, die Elemente von Mengen nicht.
- Derselbe Wert darf in einer Liste mehrfach vorkommen, in einer Menge nicht.

Zur Darstellung von Mengen verwenden wir daher nur solche Listen, deren Elemente streng monoton fallen. Die Reihenfolge der Elemente ist damit fest gelegt und eine Dopplung von Werten ist nicht möglich.

Es sollen nun Mengenoperationen auf Basis von Listen implementiert werden. Geben Sie für folgende Funktionen deren allgemeinste Typen an. Implementieren Sie anschließend die Funktionen. Sie können davon ausgehen, dass als Argumente übergebene Mengendarstellungen immer die oben genannte Eigenschaft erfüllen. Sie müssen aber sicher stellen, dass Sie keine ungültigen Mengendarstellungen als Resultate generieren.

1. **(Tutoriumsaufgabe)** Die Funktion `istElement` entscheidet, ob ein Wert in einer Menge enthalten ist und implementiert damit die Relation $\in$. Es ergibt sich z.B.

   `istElement 4 [7,5,3,1]` $\leadsto$ False

   `istElement 5 [7,5,3,1]` $\leadsto$ True

2. Die Funktion `istTeilmenge` entscheidet, ob die erste übergebene Menge Teilmenge der zweiten übergebenen Menge ist und implementiert damit die Relation $\subseteq$. Es ergibt sich z.B.

   `istTeilmenge [3,2,1] [5,4,2,1]` $\leadsto$ False

   `istTeilmenge [3,2,1] [5,4,3,2,1]` $\leadsto$ True

3. Die Funktion `istEchteTeilmenge` entscheidet, ob die erste übergebene Menge eine *echte* Teilmenge der zweiten übergebenen Menge ist und implementiert damit die Relation $\subset$. Es ergibt sich z.B.

   `istEchteTeilmenge [3,2,1] [3,2,1]` $\leadsto$ False

   `istEchteTeilmenge [3,2,1] [4,3,2,1]` $\leadsto$ True

   `istEchteTeilmenge [3,2,1] [5,4,2,1]` $\leadsto$ False

4. Die Funktion `vereinigung` bestimmt die Vereinigungsmenge zweier Mengen und implementiert damit die Funktion $\cup$. Es ergibt sich z.B.

   `vereinigung [4,3,1] [3,2,1]` $\leadsto$ `[4,3,2,1]`

5. Die Funktion `schnitt` bestimmt die Schnittmenge zweier Mengen und implementiert damit die Funktion $\cap$. Es ergibt sich z.B.

   `schnitt [5,4,2,1] [3,2,1]` $\leadsto$ `[2,1]`

Aufgabe 2 (Listcomprehensions) 3 Punkte

Für jeweils 5 gelöste Aufgaben gibt es einen Punkt. Beachten Sie, dass zu den meisten Aufgaben eine Lösung mit und eine ohne Listcomprehensions erwartet wird.

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

In der Mathematik werden Mengen häufig mit Hilfe anderer Mengen beschrieben. So beschreibt die Menge $\{x^2 | x \in \{1, ..., 5\}\}$ die Menge der Quadrate der Elemente aus der Menge $\{1, ..., 5\}$.

In Haskell gibt es eine ähnliche Notation mit deren Hilfe Listen aus anderen Listen erstellt werden können. So beschreibt `[x*x | x ← [1..5]]` die Liste der Quadrate der Elemente aus der Liste `[1..5]`. Dabei ist `[1..5]` eine Kurzschreibweise für die Liste $[1, 2, 3, 4, 5]$.

1. **(Tutoriumsaufgabe)** Schreiben Sie eine Haskell-Funktion

   `kreuzprodukt::[a]→[b] →[(a,b)]`,

   welche die Elemente aus zwei Listen so kombiniert, dass alle möglichen Kombinationen `(a,b)` mit `a` aus der ersten und `b` aus der zweiten Liste entstehen. Verwenden Sie dabei Listcomprehensions.

2. **(Tutoriumsaufgabe)** Schreiben Sie eine Haskell-Funktion

   `geordnetePaare::[Integer] →[(Integer, Integer)]`,

   welche mögliche Paare (x,y) aus den Elementen der Liste bildet, wobei $x \leq y$ gilt. Verwenden Sie dabei Listcomprehensions. Sie können davon ausgehen, dass die Elemente in der Eingabeliste aufsteigend sortiert sind. Verwenden Sie dabei Listcomprehensions.

   `geordnetePaare [1,2,3]` ⇝ `[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]`

3. **(Tutoriumsaufgabe)** Schreiben Sie eine Funktion

   `zusammenhaengen :: [[a]] →[a]`,

   welche eine Liste von Listen zu einer zusammenhängenden Liste umwandelt. Verwenden Sie dazu Listcomprehensions.

   `zusammenhaengen [[1,2], [3,4], [5,6,7,8]]` ⇝ `[1,2,3,4,5,6,7,8]`

   Schreiben Sie eine Variante `zusammenhaengen2` ohne dabei Listcomprehensions zu verwenden.

4. Schreiben Sie eine Haskell-Funktion

   `zweite :: [(a, b)] →[b]`,

   welche aus einer Liste von Tupeln, jeweils das zweite Element übernimmt. Verwenden Sie dazu Listcomprehensions.

   `zweite [(1,2),(2,7),(3,2)]` ⇝ `[2,7,2]`

   Schreiben Sie eine weitere Version `zweite2` ohne dabei Listcomprehensions zu verwenden.

5. Schreiben Sie eine Haskell-Funktion

   `durchFuenfTeilbar :: [Integer] →[Integer]`,

   welche alle Elemente, die durch fünf teilbar sind, aus einer Liste filtert. Verwenden Sie dabei Listcomprehensions.

   `durchFuenfTeilbar [1,2,3,5,12,15,20]` ⇝ `[5,15,20]`

Schreiben Sie eine weitere Variante `durchFuenfTeilbar2` ohne dabei Listcomprehensions zu verwenden.

6. **(Tutoriumsaufgabe)** Schreiben Sie eine Haskell-Funktion

   `teiler::Integer →[Integer]`,

   welche zu einer gegebenen Zahl alle Teiler berechnet. Nutzen Sie dazu Listcomprehensions.

   `teiler 24` ⇝ `[1,2,3,4,6,8,12,24]`

   Schreiben Sie eine Variante `teiler2` ohne dabei Listcomprehensions zu verwenden.

7. Schreiben Sie unter Nutzung der Funktion `teiler` eine Funktion `istPrim :: Integer →Bool`, welche für eine natürliche Zahl prüft, ob diese eine Primzahl ist.

   `istPrim 15` ⇝ False `istPrim 15` ⇝ True

8. Schreiben Sie eine Funktion `primzahlen :: Integer →[Integer]`, welche zu einer gegebenen Zahl `n` alle Primzahlen bis `n` als Liste zurück gibt. Nutzen Sie dazu Listcomprehensions.

   `primzahlen 40` ⇝ `[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]`

   Schreiben Sie eine Variante `primzahlen2` ohne dabei Listcomprehensions zu verwenden.

9. **(Tutoriumsaufgabe)**Schreiben Sie eine Funktion `paare :: [a] →[(a, a)]`, welche alle Paare aufeinander folgender Elemente der Liste bildet.

   ```
   paare [1, 2, 3, 4, 5, 6, 6, 4, 3] ⇝
   [(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 6), (6, 4), (4, 3)]
   ```

10. Schreiben Sie eine Funktion `sortiert :: Ord a ⇒[a] →Bool`, welche für eine Liste überprüft, ob diese sortiert ist. Verwenden Sie dabei Listcomprehensions.

    `sortiert [1, 2, 3, 4, 5, 6, 6, 4, 3]` ⇝ False `sortiert [1, 2, 3, 4, 5, 6, 6]` ⇝ True

    Schreiben Sie eine Variante `sortiert2` ohne Listcomprehensions zu verwenden.

11. Schreiben Sie eine Funktion `anzKleinBuch :: String →Int`, welche die Anzahl der Kleinbuchstaben in der übergebenen Zeichenkette zählt. Verwenden Sie dazu Listcomprehensions.

    Beachten Sie, dass Sie die Code-Zeile:

    `import Data.Char`

    an erster Stelle in Ihrer Haskell-Datei einfügen müssen, damit Sie auf die Funktion `isLower` zugreifen können.

    `anzKleinBuch "Haskell"` ⇝ `6`

    Schreiben Sie eine Variante `anzKleinBuch2` ohne dabei Listcomprehensions zu verwenden.

12. Schreiben Sie eine Haskell-Funktion

    `anzahl::Char →String →Int`,

    welche die Anzahl der Vorkommen eines Zeichens in einer Zeichenkette berechnet. Verwenden Sie dazu Listcomprehensions.

    `anzahl 's' "Mississippi"` ⇝ `4`

    Schreiben Sie eine Variante `anzahl2` ohne dabei Listcomprehensions zu verwenden.

13. Schreiben Sie eine Funktion `wiederhole :: Integer →a →[a]`, welche ein Element so oft wie durch eine natürliche Zahl angegeben in einer Liste ausgibt. Verwenden Sie dazu Listcomprehensions.

    `wiederhole 3 True` $\rightsquigarrow$ `[True, True, True]`

    Schreiben Sie eine Variante `wiederhole2` ohne dabei Listcomprehensions zu verwenden.

14. **(Tutoriumsaufgabe)** Eine Zahl `n` ist perfekt, wenn die Summe ihrer *echten* Teilen gleich der Zahl `n` ist. Schreiben Sie eine Funktion `istPerfekt :: Integer →Bool`, welche bestimmt, ob eine übergebene Zahl perfekt ist.

    `istPerfekt 6` $\rightsquigarrow 1 + 2 + 3 = 6 \rightsquigarrow$ `True`

    `istPerfekt 7` $\rightsquigarrow 1 \neq 7 \rightsquigarrow$ `False`

    Schreiben Sie eine Funktion `perfekteZahlen :: Integer →[Integer]`, welche alle perfekten Zahlen kleiner oder gleich einer gegebenen Zahl berechnet. Verwenden Sie dabei Listcomprehensions.

    `perfekteZahlen 500` $\rightsquigarrow$ `[6, 28, 496]`

    Schreiben Sie eine Variante `perfekteZahlen2` ohne dabei Listcomprehensions zu verwenden.

15. Schreiben Sie eine Haskell-Funktion

    `positionen :: Eq a ⇒a →[a] →[Int]`,

    welche zu einem gegebenen Element und einer Liste angibt, an welchen Positionen in der Liste sich das Element befindet, ohne Listcomprehensions zu verwenden.

    `positionen 3 [1, 2, 3, 4, 5, 6, 6, 4, 3]` $\rightsquigarrow$ `[2, 8]`

    Zusatz: (+0,5) Schreiben Sie eine Variante `positionen2`, verwenden Sie Listcomprehensions.

Aufgabe 3 (Funktionen als Listen) 1,5 Punkte

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

Wir stellen Funktionen als Listen von Paaren dar. Die Liste zu einer Funktion f enthält für jedes x, für das $f(x)$ definiert ist, das Paar $(x, f(x))$. Die Paare sind innerhalb der Liste aufsteigend nach den Funktionsargumenten sortiert. Kein Paar kommt mehrfach vor.

1. **(Tutoriumsaufgabe)**Programmieren Sie eine Haskell-Funktion `resultat`, welche zu einer als Liste dargestellten Funktion f und einem Wert x des Definitionsbereichs den entsprechenden Funktionswert $f(x)$ liefert. Ist $f(x)$ nicht definiert, soll `resultat` mit einer Fehlermeldung abbrechen. Es ergibt sich z.B.

    `resultat [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] 2` $\rightsquigarrow 4$

2. Häufig ist es nicht erwünscht, dass Funktionen mit Fehlermeldungen abbrechen. Stattdessen werden oft Werte des Datentyps `Maybe` als Funktionsresultate verwendet. `Maybe` ist im Prelude wie folgt definiert:

    `data Maybe wert =Nothing | Just wert`

Der Wert `Nothing` signalisiert einen Fehler, während ein Wert `Just x` die erfolgreiche Berechung des Wertes x darstellt. Schreiben Sie analog zur Funktion `resultat` eine Funktion `evtlResultat`, welche nicht mit Fehlermeldungen abbricht, sondern `Maybe`-Werte liefert. Es ergibt sich z.B.

`evtlResultat [(-2,4), (-1,1), (0,0), (1,1), (2,4)] 3` $\leadsto$ **`Nothing`**

`evtlResultat [(-2,4), (-1,1), (0,0), (1,1), (2,4)] 2` $\leadsto$ **`Just 4`**

3. Programmieren Sie eine Haskell-Funktion `urbilder`, die zu einer gegebenen Funktion f und einem gegebenen Wert y die Menge $\{x|f(x) = y\}$ berechnet. Diese Menge soll wie in 1. beschrieben dargestellt werden. Es ergibt sich z.B.

   `urbilder [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)] 4` $\leadsto$ `[-2, 2]`

4. Programmieren Sie eine Haskell-Funktion `echteArgumente`, welche zu einer gegebenen Funktion f die Menge aller x liefert, für die es einen Funktionswert $f(x)$ gibt. Verwenden Sie wieder die Mengendarstellung aus 1. Es ergibt sich z.B.

   `echteArgumente [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)]` $\leadsto$ `[2, 1, 0, -1, -2]`

5. Wir betrachten nun Funktionen, deren Definitions- und Wertebereich identisch sind. Ein Wert x ist Fixpunkt einer solchen Funktion f, wenn $f(x) = x$ gilt. Programmieren Sie eine Haskell-Funktion `fixpunkte`, die zu einer gegebenen Funktion die Menge aller Fixpunkte berechnet. Diese Menge soll wieder wie in 1. beschrieben dargestellt werden. Es ergibt sich z.B.

   `fixpunkte [(-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4)]` $\leadsto$ `[1, 0]`

6. Programmieren Sie eine Haskell-Funktion `funKomposition`, die aus zwei Funktionen f und g deren Komposition $f \circ g$ berechnet. Es ergibt sich z.B.

   `funKomposition [(1,0),(2,1)] [(-2, 4), (-1, 1), (0, 0), (1, 1)]` $\leadsto$ [(-1,0),(1,0)]

Aufgabe 4 (Merge Sort) 2 Punkte

1. **(Tutoriumsaufgabe)** Diskutieren Sie, wie das Merge-Sort-Verfahren funktioniert.

2. Implementieren Sie die aus der Vorlesung bekannten Sortierverfahren „Insertion Sort“ und „Merge Sort“. Ändern Sie „Merge Sort“ wie folgt ab:

   Es soll ein `Int`-Wert n übergeben werden. Wenn die Länge der eingegebenen Liste größer als n ist, dann soll „Merge Sort“ verwendet werden, ansonsten „Insertion Sort“. Dem zufolge muss bei der Eingabe `mergeSort [4,3,2,1] 2` zunächst „Merge sort“ angewendet werden und dann im rekursiven Aufruf „Insertion Sort“.

   `mergeSort :: Ord a ⇒[a] →Int →[a]`