Algorithmieren und Programmieren

Sommersemester 2023

Lehrstuhl Programmiersprachen und Compilerbau
Prof. Dr. rer. nat. habil. Petra Hofstedt
Ilja Becker
Andreas Eberle, Jan Robert Meyer, Oliver Pflug

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Übungsblatt 7

Abgabedatum: 11.06.2023

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 des ***Glasgow Haskell Compiler (GHC)*** (`https://www.haskell.org/`) empfohlen. Weitere Informationen finden Sie im Software-Reiter auf Moodle.
- Als **Tutoriumsaufgabe** markierte (Teil-)aufgaben werden in den Übungen ausführlicher besprochen. Die schriftlichen 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-Dateien, 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 (`https://wiki.haskell.org/Tutorials`) sowie die Suchmaschine Hoogle (`http://hoogle.haskell.org/`) für die Haskell-API, die Ihnen mit zunehmender Haskell-Erfahrung wertvolle Dienste leisten wird.

Sie können maximal **(7 Punkte (+1 Zusatzpunkt))** mit diesem Übungsblatt erreichen.

Aufgabe 1 (abstrakter Datentyp für Mengen) 4 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 3 von Aufgabenblatt 4. 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. Der Datentyp `Menge` soll dabei ein eigener Datentyp sein und kein Typalias. Legen Sie eine geeignete Exportliste an. Es soll außerdem möglich sein, Mengen auf Gleichheit zu testen und automatisch 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 3 des vierten Aufgabenblatts 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 (Kartendeck-Modul) 3 (+1) Punkte

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

Ziel dieser Aufgabe ist es, ein Modul zur Repräsentation eines Kartenspiels mit Französischem Blatt zu implementieren[1]. Das Französische Blatt besteht Karten die sich zusammensetzen aus Farben und Werten. Die Farben sind:

- Herz ♡ (engl.: Heart Suit)
- Karo ♢ (engl.: Diamond Suit)
- Kreuz ♣ (engl.: Club Suit)
- Pik ♠ (engl.: Spade Suit)

Die Werte sind gegeben durch:

- Bube *B* (engl.: Jack)
- Dame *D* (engl.: Queen)
- König *K* (engl.: King)
- Ass *A* (engl.: Ace)
- Zahl $\in 2, \ldots, 10$

Das Kartesische Produkt von Farben und Werten ergibt 52 Karten, die das volle Französische Blatt darstellen:

$$\{A\heartsuit, K\heartsuit, D\heartsuit, B\heartsuit, 10\heartsuit, \ldots, 2\heartsuit, A\diamond, \ldots, 2\diamond, A\clubsuit, \ldots, 2\clubsuit, A\spadesuit, \ldots, 2\spadesuit\}$$

Implementieren Sie zunächst ein Modul `Karten`. Der abstrakte Datentyp soll folgende Operationen bereit stellen:

- `Karte`
  Der Datentyp der eine Spielkarte darstellt und alle Spielkarten aus dem oben beschriebenen Französischen Blatt darstellen kann
- `getFarbe :: Karte →??`
  getFarbe gibt die Farbe der Karte zurück. Der Typ der Farbe ist davon abhängig, wie Sie die Information der Kartenfarbe implementieren
- `getWert :: Karte →??`
  getWert gibt den Wert der Karte zurück. Der Typ des Werts ist davon abhängig, wie Sie die Information des Kartenwertes implementieren
- **`instance Eq`** `Karte`
  das Modul soll die Möglichkeit bereitstellen, Karten auf Gleichheit zu vergleichen

[1] Siehe auch Wikipedia: `https://de.wikipedia.org/w/index.php?title=Spielkarte&oldid=200005556#Franz%C3%B6sisches_Blatt`

- `instance Show Karte`
  das Modul soll die Möglichkeit bereitstellen, Karten auf Basis der Typklasse Show auszugeben
- `Kartenstapel`
  geeigneter Typ zur Darstellung eines Stapels von Karten (kann auch ein Type Synonym[2] sein)
- `neuerKartenstapel :: Kartenstapel`
  Die Funktion neuerKartenstapel soll ein komplettes Französisches Blatt (52 Karten, ohne Joker) zurückgeben
- `geben :: Int →Int →Kartenstapel →[Kartenstapel]`
  `geben` bekommt eine Anzahl an Stapeln st_{count}, eine Anzahl an Karten n und einen Kartenstapel T übergeben. Vom übergebenen Kartenstapel T sollen dann Karten gegeben werden, und zwar einzeln und der Reihe nach. Das heißt Stapel 1 St_1 bekommt die erste Karte von Stapel T, der zweite Stapel St_2 die zweite, bis zur st_{count}-en Karte auf Stapel $St_{st_{count}}$. Dies wiederholt sich solange, bis alle Stapel n Karten enthalten. Sollte der Kartenstapel vorher zu Neige gehen, soll eine Fehlermeldung ausgegeben werden.

Implementieren Sie hierfür:

1. Legen Sie ein Modul `Karten` für den abstrakten Datentyp an. Fügen Sie in dieses Modul die Deklaration des Datentyps `Karte` sowie die Typsignaturen der Funktionen ein. Legen Sie eine geeignete Exportliste an - dabei sollen nur die Funktionen und Typen exportiert werden, die für den Nutzer zur Nutzung nötig sind.
2. Implementieren Sie die oben gegebenen Typen und Funktionen und ggf. nötige Hilfsfunktionen.

Im Folgenden wollen wir das Modul, dass wir eben implementiert haben, nutzen.

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

3. Importieren Sie ihr Modul `Karten` in ihrer Datei `SimpleGames.hs`. Schreiben Sie eine Funktion

   `testCardCreation :: Karte`

   die testweise eine Spielkarte der Farbe Herz mit dem Wert Ass anlegt. Schreiben Sie außerdem eine Funktion

   `testStapel :: Kartenstapel`

   die lediglich die Funktion `neuerKartenstapel` aus dem Modul `Karten` aufruft und das Ergebnis ausgibt.
4. Schreiben Sie eine Funktion

   `countCards :: Kartenstapel →Int`

   die die Werte der Karten folgendermaßen zusammenzählt[3]:

   - Karten mit Zahlenwert sollen einfach entsprechend ihren Zahlenwerts aufaddiert werden
   - Karten mit König-, Dame- oder Bube-Wert sollen 10 Punkte wert sein
   - Karten mit Ass-Wert sollen 11 Wert sein, es sei denn die Summe des restlichen Stapels + 11 ergibt einen Wert höher als 21 - dann sei eine Karte mit Ass-Wert 1 Punkt wert

[2] Siehe auch: `https://wiki.haskell.org/Type_synonym`
[3] Die Regeln zur Aufsummierung orientieren sich am Spiel Black Jack, siehe auch: `https://de.wikipedia.org/wiki/Black_Jack`

`countCards` $[A\heartsuit, K\clubsuit, D\spadesuit] \leadsto^* 21$

`countCards` $[A\heartsuit, D\spadesuit] \leadsto^* 21$

`countCards` $[A\heartsuit, A\spadesuit, 10\heartsuit] \leadsto^* 12$

5. Schreiben Sie für ihre Funktion `countCards` eine Testfunktion

   `testCountCards1 :: Bool`

   die überprüft, ob ein Aufruf von countCards mit einem Kartenstapel, der die folgenden Karten beinhaltet:

   $$\{A\heartsuit, K\clubsuit, D\spadesuit\}$$

   auch das Ergebnis 21 liefert. Schreiben Sie in selber Manier weitere Testfunktionen

   `testCountCardsN :: Bool`

   für wenigstens die folgenden Kartenstapel:

   - $\{A\heartsuit, K\clubsuit\}$
   - $\{A\heartsuit, A\clubsuit, D\spadesuit\}$
   - $\{10\heartsuit, 10\clubsuit, 8\spadesuit\}$

6. **(Zusatzaufgabe)** Schreiben Sie eine Funktion

   `containsFullHouse :: Kartenstapel →Bool`

   die für einen Kartenstapel berechnet, ob sich darin ein sogenanntes Full House finden lässt. Ein Full House ist im Poker eine Kombination von 5 Karten, die aus 3 Karten eines Wertes und 2 Karten eines weiteren Wertes besteht[4]. Die Funktion `containsFullHouse` soll aber auf einen beliebig großen Kartenstapel anwendbar sein.

   `containsFullHouse` $[A\heartsuit, A\clubsuit, 8\heartsuit, 8\diamondsuit, 8\spadesuit] \leadsto^*$ True

7. **(Zusatzaufgabe)** Schreiben Sie für ihre Funktion `countainsFullHouse` eine Testfunktion

   `testFullHouse1 :: Bool`

   die überprüft, ob ein Aufruf von containsFullHouse mit einem Kartenstapel, der die folgenden Karten beinhaltet:

   $$\{A\heartsuit, A\clubsuit, 3\clubsuit, 3\spadesuit, 3\heartsuit\}$$

   auch das Ergebnis codeStyleTrue liefert. Schreiben Sie in selber Manier weitere Testfunktionen

   `testFullHouseN :: Bool`

   für wenigstens die folgenden Kartenstapel:

   - $\{A\heartsuit, A\clubsuit, 3\clubsuit, 3\spadesuit, 3\heartsuit\} \leadsto$ `*True`
   - $\{A\heartsuit, A\clubsuit, 3\clubsuit, 3\spadesuit, 4\heartsuit\} \leadsto$ `*False`
   - $\{A\spadesuit, A\heartsuit, A\clubsuit, 3\clubsuit, 3\spadesuit, 3\heartsuit\} \leadsto$ `*True`

[4] Siehe auch Wikipedia: `https://de.wikipedia.org/wiki/Hand_(Poker)#Full_House`

Achten Sie darauf, dass ihre Testfunktion immer dann `True` zurückliefert, wenn das Ergebnis von `containsFullHouse` korrekt ist. Ihre Testfunktion soll nicht einfach das Ergebnis von `containsFullHouse` spiegeln.