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 4

Abgabedatum: 14.05.2023

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

Aufgabe 1 (Arithmetische Ausdrücke) 1 Punkt

Wir betrachten einen rekursiven Datentyp Ausdruck, dessen Werte arithmetische Ausdrücke darstellen. Als Operationen sind Addition, Subtraktion, Multiplikation und (ganzzahlige) Division zugelassen, atomare Einheiten können Konstanten und Variablen sein. Die Deklaration von Ausdruck ist wie folgt:

```
data Ausdruck = Konstante Integer
    | Variable String
    | Summe Ausdruck Ausdruck
    | Differenz Ausdruck Ausdruck
    | Produkt Ausdruck Ausdruck
    | Quotient Ausdruck Ausdruck
   deriving (Show) -- ermoeglicht Ausgabe der Werte in GHCi
```

Der Ausdruck $x + (3 - x) * y$ wird beispielsweise durch

```
Summe (Variable "x") (Produkt (Differenz (Konstante 3) (Variable "x")) (Variable "y"))
```

repräsentiert.

1. Geben Sie die Repräsentationen folgender arithmetischer Ausdrücke an:
   a) $12 + z$
   b) $x * z + 7$
   c) $(8 - x) * y$
   d) $(z - j)/5$
2. Geben Sie die arithmetischen Ausdrücke an, die folgendermaßen repräsentiert werden:
   a) `Summe (Konstante 3) (Variable "x")`
   b) `Summe (Summe (Variable "x") (Konstante 3)) (Quotient (Konstante 4) (Variable "y"))`
   c) `Differenz (Produkt (Variable "k") (Konstante 8)) (Konstante 1)`
3. **Verwenden Sie für die Abgabe den Dateinamen:** `Arithmetik.hs`

   Programmieren Sie eine Haskell-Funktion `ausdruckNachString::Ausdruck →String` , welche arithmetische Ausdrücke in Zeichenfolgen konvertiert. Die ausgegebenen Zeichenfolgen sollen die arithmetischen Ausdrücke in Haskell-Syntax darstellen. Es ist z.B. folgendes möglich:

   `ausdruckNachString (Summe (Konstante 10) (Variable "a"))` ⇝ `"10 +a"`

   Zum Aneinanderhängen von Zeichenfolgen kann der Operator ++ mit

   `(++) :: String →String →String`

   verwendet werden. Ganzzahlen können durch Anwenden der Funktion **show** in ihre Zifferndarstellung konvertiert werden. Es ergibt sich beispielsweise:

   show `325` ⇝ `"325"`

   Beachten Sie, dass Punktrechnung vor Strichrechnung gilt. (Solange es nicht ausartet, sind „zuviele“ Klammern erlaubt, solange der Ausdruch richtig ist.)
4. Programmieren Sie eine Haskell-Funktion `belegungVonVariable::String →[(String, Integer)] → Integer`, welche zu einem Variablennamen die entsprechende Belegung der Variable aus einer Liste von Variablenbelegungen ermittelt.

`belegungVonVariable "x"[("x", 3), ("y", 4), ("z", 1)]` ⇝ `3`

`belegungVonVariable "y"[("x", 3), ("y", 4), ("z", 1)]` ⇝ `4`

`belegungVonVariable "a"[("x", 3), ("y", 4), ("z", 1)]` ⇝ **`error`** `"Nicht definiert"`

5. Programmieren Sie eine Haskell-Funktion

   `auswerten :: Ausdruck →[(`**`String`**`,` **`Integer`**`)] →Ausdruck`

   Die Funktion soll den übergebenen `Ausdruck` unter der gegebenen Belegung so weit wie möglich auswerten. Beachten Sie, dass die **`div`** Funktion für das teilen von **`Integer`**-Werten genutzt werden kann.

Aufgabe 2 (Rekursion über Listen) 3 Punkte

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

Bei dieser Aufgabe gibt es für jeweils 6 richtig implementierte Funktionen einen Punkt.

In den folgenden Teilaufgaben sollen nicht die entsprechenden vordefinierten Bibliotheksfunktionen verwendet werden. Die selber programmierten Funktionen dürfen für die nachfolgenden Funktionen verwendet werden. Programmieren Sie die folgenden Funktionen:

1. `istLeer :: [a] →`**`Bool`**

   Überprüft, ob die übergebene Liste leer ist. `istLeer` gibt **`True`** zurück, wenn die Liste leer ist, sonst **`False`**.

2. `snoc :: [a] →a →[a]`

   Hängt ein Element an das Ende der Liste an.

3. `laenge :: [a] →`**`Int`**

   Gibt die Länge der übergebenen Liste zurück.

4. `erstesElement :: [a] →a`

   Gibt das erste Element der übergebenen Liste zurück. Ist die Liste leer, soll eine Fehlermeldung ausgegeben werden.

5. `rest :: [a] →[a]`

   Gibt die Restliste nach Entfernen des ersten Elements zurück. Ist die Eingabeliste leer, soll eine Fehlermeldung ausgegeben werden.

6. `letztesElement :: [a] →a`

   Gibt das letzte Element der übergebenen Liste zurück. Ist die Liste leer, soll eine Fehlermeldung ausgegeben werden.

7. `anfang :: [a] →[a]`

   Gibt die Anfangsliste nach Entfernen des letzten Elements zurück. Ist die Eingabeliste leer, soll eine Fehlermeldung ausgegeben werden.

8. `nimm :: `**`Int`**` →[a] →[a]`

`nimm n list` gibt die ersten `n` Elemente der Liste zurück. Sind weniger als `n` Elemente vorhanden, werden so viele Elemente wie möglich zurückgegeben.

9. `verwerfe :: Int →[a] →[a]`

   `verwerfe n list` lässt die ersten `n` Elemente der Liste weg. Sind weniger als `n` Elemente vorhanden, verbleibt eine leere Liste.

10. `produkt :: (Num a) ⇒[a] →a`

    Bildet das Produkt aller in der Liste enthaltenen Elemente.

11. `verdopple :: (Num a) ⇒[a] →[a]`

    Verdoppelt die Werte der Liste.

    ```
    verdopple [1,2,3] ~>* [2,4,6]
    ```

12. `verkette :: [a] →[a] →[a]`

    Verkettet die zwei übergebenen Listen.

13. `rueckwaerts :: [a] →[a]`

    Gibt die Eingabeliste in umgekehrter Reihenfolge zurück.

14. `und :: [Bool] →Bool`

    Gibt **`True`** zurück, wenn alle Elemente in der Liste **`True`** sind, sonst **`False`**.

15. `nicht :: [Bool] →[Bool]`

    Gibt eine Liste zurück, in der jeder bool'sche Wert negiert wurde.

16. `aufteilen :: Int →[a] →([a],[a])`

    Teilt die Liste an der durch die ganze Zahl angegebenen Position auf.

17. `verzahne :: [a] →[b] →[(a,b)]`

    Verzahnt zwei Listen so, dass eine Liste aus Tupeln entsteht. Ist eine Liste länger als die andere, soll der Rest der längeren Liste verworfen werden.

    ```
    verzahne [1,2,3,4] [True, False, True] ~>* [(1,True), (2, False), (3, True)]
    ```

18. `aktualisiere :: [a] →Int →a →[a]`

    `aktualisiere list n e` ersetzt das n-te Element durch das Element `e`.

    ```
    aktualisiere [3,5,4,6] 2 7 ~>* [3,5,7,6]
    ```

Aufgabe 3 (Konvertierung in Binärzahlen und zurück) 2 Punkte

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

1. **(Tutoriumsaufgabe)** Geben Sie einen geeigneten Datentyp *Bit* zur Repräsentation von Bits an.
2. **(Tutoriumsaufgabe)** Geben Sie einen geeigneten Datentyp *Bits* zur Repräsentation von natürlichen Zahlen in der Binärdarstellung an. Dabei steht das höchstwertige Bit ganz vorne.
3. Programmieren Sie eine Haskellfunktion `integer2Bits :: Integer →Bits`, welche die in einem Integer gespeicherte natürliche Zahl in die entsprechende Binärdarstellung umwandelt.

4. Programmieren Sie die Haskellfunktionen `bits2String::Bits →String` und `string2Bits::String →Bits`, welche eine Liste von Bits in einen Binärstring bzw. zurück umwandeln.
5. Programmieren Sie eine Haskellfunktion `integer2binString :: Integer →String`, welche eine natürliche Zahl in einen Binärstring umwandelt.
6. Programmieren Sie eine Haskellfunktion `bits2Integer :: Bits →Integer`, welche eine Liste von Bits in die entsprechende natürliche Zahl als Integer konvertiert.
7. Programmieren Sie eine Haskellfunktion `bitString2Integer :: String →Integer`, welche eine als Binärstring angegebene Zahl in die entsprechende Zahl als Integer konvertiert.

Aufgabe 4 (Mengen als Listen) 2 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]`