Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

27. Januar 2016

Aufgaben zur Klausur **Grundlagen der funktionalen Programmierung** im WS 2015/16 (BInf)

Zeit: 90 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Nutzen Sie die Rückseiten der Klausur zur Entwicklung der Lösungen und übertragen die fertigen Lösungen in das Aufgabenblatt.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 9 Seiten.

Aufgabe 1:

Gegeben seien folgende kleine Funktionen und Prädikate

```
toUpper :: Char → Char
toUpper c
  | isLower c =
    toEnum (fromEnum c − fromEnum 'a' + fromEnum 'A')
  | otherwise = c
isSpace :: Char → Bool
isSpace c =
  c == ' '        -- blank
  || c == '\n'    -- linefeed
  || c == '\t'    -- tabulator
isAlpha :: Char → Bool
isAlpha c = isUpper c || isLower c
isUpper :: Char → Bool
isUpper c = c ⩾ 'A' && c ⩽ 'Z'
isLower :: Char → Bool
isLower c = c ⩾ 'a' && c ⩽ 'z'
isCL :: String → Bool
isCL ('_' : c1 : _) = isLower c1
isCL cs = False
```

Entwickeln Sie die fogenden Funktionen mit Hilfe der hier vorgegebenen und der vordefinierten Funktionen. Nutzen Sie, wenn möglich, anstatt expliziter Rekursion vordefinierte Funktionen für das rekursive Verarbeiten von Listen.

Entwickeln Sie eine Funktion `wordToUpper`, die in einem String alle Kleinbuchstaben in Großbuchstaben transformiert.

```
wordToUpper :: String → String
```

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `wordToCapital`, die nur den Anfangsbuchstaben (erstes Zeichen) in einen Großbuchstaben transformiert.

$wordToCapital :: String \rightarrow String$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `wordsToUpper`, die in allen Wörter einer Liste alle Buchstaben in Großbuchstaben transformiert.

$wordsToUpper :: [String] \rightarrow [String]$

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `wordsToCapital`, die für alle Wörter in einer Liste den Anfangsbuchstaben zu einem Großbuchstaben macht.

$wordsToCapital :: [String] \rightarrow [String]$

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `toCamelCase`, die in einem String alle Zeichenkombinationen aus einem Unterstrich und einem Kleinbuchstaben (siehe `isCL`) durch den entsprechenden Großbuchstaben ersetzt:

```
toCamelCase "get_money" == "getMoney"
```

$toCamelCase :: String \rightarrow String$

.....

.....

.....

.....

.....

.....

.....

.....

Aufgabe 2:

Gegeben sei die folgende Datenstruktur für Bäume, die vier Arten von Knoten besitzen. In allen Knotenarten wird ein Wert eines Typs `a` gespeichert. Bei der ersten Ausprägung wird keine weitere Information gespeichert, bei der zweiten hängt zusätzlich links ein Teilbaum, bei der dritten rechts und bei der vierten auf beiden Seiten.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{TLeaf}\ a \\
&\quad \mid \mathit{TLeft}\ (\mathit{Tree}\ a)\ a \\
&\quad \mid \mathit{TRight}\ a\ (\mathit{Tree}\ a) \\
&\quad \mid \mathit{TBin}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)
\end{aligned}
$$

Mit der Funktion `flatten` können die Elemente aus dem Baum extrahiert und in einer Liste gesammelt werden. Die Elemente sind in der Liste in der gleichen Reihenfolge wie im Baum gespeichert.

$$
\begin{aligned}
&\mathit{flatten} :: \mathit{Tree}\ a \to [a] \\
&\mathit{flatten}\ (\mathit{TLeaf}\ x) = [x] \\
&\mathit{flatten}\ (\mathit{TLeft}\ \ l\ x) = \mathit{flatten}\ l +\!\!+ [x] \\
&\mathit{flatten}\ (\mathit{TRight}\ x\ r) = x : \mathit{flatten}\ r \\
&\mathit{flatten}\ (\mathit{TBin}\ l\ x\ r) = \mathit{flatten}\ l +\!\!+ [x] +\!\!+ \mathit{flatten}\ r \\
&\mathit{smartFlatten} :: \mathit{Tree}\ a \to [a] \\
&\mathit{smartFlatten}\ t = \mathit{consTree}\ t\ [\,]
\end{aligned}
$$

`flatten` ist ineffizient, da in dieser Funktion viel mit `(++)` gearbeitet wird. Die Laufzeit dieser Operation hängt linear von der Länge des linken Operanden ab. Die Funktion `smartFlatten` versucht dieses besser zu machen. Sie ruft eine Funktion `consTree` mit dem Baum und einer leeren Liste auf. Diese `consTree`-Funktion bestimmt also die Laufzeit von `smartFlatten`

Entwickeln Sie die Funktion `consTree` so, dass sie alle Elemente des Baumes (1. Argument) vor die Liste (2. Argument) packt. `smartFlatten` soll immer das gleiche Ergebnis liefern wie `flatten`, die Funktion soll aber in einer Rechenzeit proportional zur Anzahl der im Baum gespeicherten Werte laufen.

$consTree :: Tree\ a \to [a] \to [a]$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Läuft `smartFlatten` in einer Zeit proportional zur Länge der Resultat-Liste?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `leftmost`, die aus einem Baum das am weitesten links stehende Element selektiert, es soll also gelten `leftmost t = head (flatten t)`

$leftmost :: Tree\ a \to a$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Ist `leftmost` eine totale Funktion?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Aufgabe 3:

`isPrefix` sei ein Prädikat, das testet, ob eine Liste `xs` ein Anfangsstück einer zweiten Liste `ys` ist, das heisst, alle Elemente der ersten Liste stehen am Anfang der zweiten Liste. Es muss also gelten:

```
xs == take (length xs) ys
```

Entwickeln Sie das Prädikat so, das es dieses Gesetz erfüllt. Dieses Gesetz bildet eine präzise Spezifikation, ist als Implementierung aber aus Effizenzgründen nicht sinnvoll.

$isPrefix :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

`tails` ist eine Funktion, die zu einer Liste `xs` alle Endstücke dieser Liste berechnet. Es gilt `tails "abc" == ["abc","bc","c",""]`, Das Resultat ist also eine Liste deren Länge um 1 größer ist als die Länge der Argumentliste.

$tails :: [a] \rightarrow [[a]]$

. . . . . .

. . . . . .

. . . . . .

. . . . . .

`or` ist eine vordefinierte Funktion.

$$or :: [Bool] \to Bool$$
$$or\ xs = foldl\ (\ ||\ )\ False\ xs$$

Entwickeln Sie ein Prädikat `containsPrefix`, mit dem getestet wird, ob eine Liste `xs` Prefix eines Elementes einer Liste `yss` ist. Es gilt also:

```
containsPrefix "abc" ["xabc", "a", ""] == False
```

und

```
containsPrefix "a" ["xabc", "a", ""] == True.
```

Nutzen Sie dabei ausschließlich `isPrefix`, `map` und `or`.

$$containsPrefix :: Eq\ a \Rightarrow [a] \to [[a]] \to Bool$$

. . . . . .

. . . . . .

Entwickeln Sie eine Funktion `occurs` zum Testen, ob eine Zeichenreihe `xs` in einer anderen Zeichenreihe `ys` als Teilzeichenreihe vorkommt. Nutzen sie dafür ausschließlich Funktionen, die in dieser Aufgabe definiert sind.

$$occurs :: Eq\ a \Rightarrow [a] \to [a] \to Bool$$

. . . . . .

. . . . . .
