PHILIPPS-UNIVERSITÄT MARBURG
Fachbereich Mathematik und Informatik
Prof. Dr. R. Loogen

D-35032 Marburg
Hans Meerwein Straße
Lahnberge

5. November 2003

Übungen zur „Praktischen Informatik III“, WS 2003/04

Nr. 3, Besprechung bzw. Abgabe: 12. und 13. November in den Übungen

---

A. Mündliche Aufgaben

11. Ähnlichkeit von Worten

    Schreiben Sie eine Funktion `similar`, die prüft, ob zwei Worte (Typ `String`) *ähnlich* sind. Betrachten Sie als „Ähnlichkeit“ folgende unterschiedlichen Definitionen:

    (a) Zwei Worte heißen ähnlich, wenn sie sich nur in einem Buchstaben unterscheiden.

    (b) Zwei Worte heißen ähnlich, wenn sie durch Auslassen bzw. Hinzufügen genau eines Buchstabens ineinander übergehen.

    (c) Zwei Worte heißen ähnlich, wenn sie durch Vertauschung zweier benachbarter Buchstaben ineinander übergehen.

    Wie ändert sich die Funktion, wenn man alle drei Möglichkeiten als Definition von Ähnlichkeit akzeptiert?

12. Mischsortieren

    Implementieren Sie eine Funktion `mergeSort :: Ord a => [a] -> [a]`

    die eine Liste von Elementen durch Mischen sortiert. Beim Mischsortieren wird die zu sortierende Folge in zwei Teilfolgen zerlegt, die rekursiv mit demselben Verfahren sortiert werden. Die zwei sortierten Teilfolgen werden dann zu einer sortierten Gesamtfolge gemischt.

    Der Kontext `Ord a` bedeutet, dass die Typvariable `a` nur durch solche Typen ersetzt werden darf, auf deren Elementen eine Ordnungsrelation mit den üblichen Vergleichsoperationen (`<`, `<=`, etc.) definiert ist. Der Kontext `Ord a` beinhaltet implizit den Kontext `Eq a`.

13. Mengen

    (a) Eine Menge kann in Haskell durch eine Liste repräsentiert werden. Implementieren Sie die folgenden Operationen auf Mengen ohne Verwendung von Listenabstraktionen:

    ```
    schnitt         :: Eq a => [a] -> [a] -> [a]
    vereinigung     :: [a] -> [a] -> [a]
    differenz       :: Eq a => [a] -> [a] -> [a]
    potenzmenge     :: [a] -> [[a]]
    keine_duplikate :: Eq a => [a] -> [a]
    ```

    Nützlich sind außerdem die folgenden Testfunktionen, die ebenfalls implementiert werden sollen:

```
disjunkt  :: Eq a => [a] -> [a] -> Bool
teilmenge :: Eq a => [a] -> [a] -> Bool
gleich    :: Eq a => [a] -> [a] -> Bool
```

(b) Bei welchen Funktionen können Listenabstraktionen sinnvoll eingesetzt werden? Geben Sie für diese Funktionen eine solche alternative Implementierung an.

B. Hausaufgaben

Hinweise: Die Lösungen sollten grundsätzlich schriftlich, Programme zusätzlich auf Diskette oder per E-Mail an Ihren Tutor abgegeben werden. Die Abgabe ist in Gruppen bis zu zwei Personen erlaubt.

14. Wortsuche — 4 Punkte

Schreiben Sie eine Funktion `isIn :: String -> String -> Bool`, die zu einem Wort und einem Text (beide vom Typ `String`) prüft, ob das Wort in dem Text vorkommt.

15. Zentrieren von Zeichenketten — 4 Punkte

Eine Zeichenkette wird in Haskell als eine Liste von Zeichen ausgedrückt:

`type String = [Char]`.

(a) Schreiben Sie eine Funktion

```
wiederholeZeichen :: Char -> Int -> String
```

so dass der Aufruf `wiederholeZeichen 't' 5` die Zeichenkette `"ttttt"` erzeugt.

(b) Implementieren Sie eine Funktion

```
zentriere :: Int -> String -> String
```

die als Argumente eine Zahl, die die Breite einer Ausgabezeile beschreibt, und eine in dieser Zeile zu zentrierende Zeichenkette bekommt. Als Ergebnis erhält man die Zeile der gewünschten Länge, in der die Zeichenkette durch Einfügen von Leerzeichen in die Mitte gerückt wurde. Verwenden Sie die Funktion `wiederholeZeichen` zum Erzeugen der benötigten Leerzeichen am Anfang und am Ende der Zeile. Sie können davon ausgehen, daß die Zeichenkette immer kürzer als die Zeilenlänge ist.

Der Aufruf `zentriere 10 "Hallo!"` soll die Zeichenkette `"  Hallo!  "` erzeugen.

16. Listenpermutationen — 4 Punkte

Definieren Sie unter Verwendung von Listenabstraktionen

(a) eine Funktion `removeList :: [a] -> [(a,[a])]`, die zu einer Liste alle Möglichkeiten bestimmt, ein Element zu entfernen. Als Ergebnis soll eine Liste aller möglichen Paare (entferntes Element, Restliste) zurückgegeben werden.

Beispiel: `removeList [1,2,3] =>* [(1,[2,3]), (2,[1,3]), (3,[1,2])]`

(b) eine Funktion `perms :: [a] -> [[a]]`, die zu einer Liste alle möglichen Permutationen der Listenelemente bestimmt.

Beispiel:

`perms [1,2,3] =>* [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`