Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
L. Noschinski, L. Hupel, M. Eberl

WS 2014/15
09.01.2015
Abgabe: 16.01.2015

Einführung in die Informatik 2
12. Übung

Am 16.01.2015 findet von 13-15 Uhr eine Sprechstunde im Raum MI 01.09.014 statt. Sie können dort den Tutoren allgemeine Fragen zum Unterrichtsstoff, zu den Übungsaufgaben (auch vergangene Blätter) und zu Altklausuren stellen. Die Altklausuren finden Sie online auf unseren Lehrstuhlseiten, sowie gedruckt zum Abholen in der Sprechstunde.

Aufgabe G12.1 Huffman-Kodierung

In der Vorlesung wurde Huffmans Algorithmus zur Komprimierung von Daten vorgestellt.

Schreiben Sie ein Programm, das diesen Algorithmus verwendet, um Dateien zu komprimieren. Das heißt, implementieren Sie die folgenden Funktionen:

```
compress :: String -> FilePath -> IO ()
decompress :: FilePath -> IO (Maybe String)
```

`compress` xs $file$ soll den String xs komprimieren und das Ergebnis in der Datei $file$`.huff` ablegen. Der zur Dekomprimierung benötigte Baum soll in der Datei $file$`.code` landen. Umgekehrt soll `decompress` $file$ die beiden Dateien $file$`.huff` und $file$`.code` lesen und den dekomprimierten String zurückgeben.

Hinweis: Verwenden Sie das Modul `Data.ByteString` um die von Huffman ausgegebene Bitfolge platzsparend abzulegen. Hilfreich ist es dabei, Funktionen `toWord8` und `fromWord8` zu schreiben, die eine Liste von 8 Bits in ein `Word8` und zurück konvertieren.

Wir haben das Modul `Huffman` um Funktionen

```
serializeTree :: Tree -> ByteString
deserializeTree :: ByteString -> Tree
```

erweitert, die sie zum Ablegen des Kodierungsbaums verwenden können.

Aufgabe G12.2 Skewed

In der Vorlesung wurde mit Skew Heaps eine Datenstruktur vorgestellt, die eine Operation `extractMin` anbietet.

Für Ihre Anwendung benötigen Sie jetzt eine Datenstruktur, die stattdessen `extractMax` erlaubt. Wie können Sie das erreichen, *ohne* die Skew-Heap-Implementierung zu verändern oder zu kopieren?

Implementieren Sie Ihre Lösung.

Hinweis: Die Vergleichsoperationen `<=`, `<`, `>`, `>=` sind über die Typklasse `Ord` definiert.

Aufgabe H12.1 Morse I

Morse-Code ist ein Verfahren, mit dem Zeichen mithilfe eines konstanten Signals (z.B. Licht, Sinuston, Spannung) übertragen werden können. Das Signal ist hierbei immer abwechselnd für eine bestimmte Zeit angeschaltet und ausgeschaltet; die Länge der Signal-/Pausenabschnitte kodiert dabei das Symbol.

Folgende grundlegende Symbole gibt es im Morsecode

Dit: kurzes Signal, geschrieben als Punkt (`.`)

Dah: langes Signal (etwa 3 *Dits*), geschrieben als Strich (`-`)

Zeichen (wie Buchstaben oder Zahlen) werden als eine Abfolge mehrerer *Dits* und *Dahs* kodiert; die genaue Kodierung können Sie Abb. 1 entnehmen.

Gegeben Sind die Typdefinitionen

```
data Morse = Dit | Dah deriving (Eq, Ord, Show, Read)
type MorseLetter   = [Morse]
type MorseWord     = [MorseLetter]
type MorseSequence = [MorseWord]
```

Ein Morse-Buchstabe ist also eine Liste von *Dits* und *Dahs*, ein Morse-Wort eine Liste von Morse-Buchstaben und eine Morse-Sequenz eine Liste von Morse-Wörtern.

Ihre Aufgabe ist es, eine Funktion

```
morseToString :: MorseSequence -> Maybe String
```

zu schreiben, die eine Morse-Sequenz in den dazugehörigen String (in Großbuchstaben) umwandelt und umgekehrt. Falls die Eingabe keine gültige Morse-Sequenz darstellt (also Morse-Buchstaben, die nicht in Abb. 1 aufgeführt sind), soll die Funktion `Nothing` zurückgeben.

Beispiel:

```
morseToString [[[Dit,Dit,Dit],[Dah,Dah,Dah],[Dit,Dit,Dit]]]
    == Just "SOS"
morseToString [
    [[Dit,Dit]],
    [[Dit,Dah], [Dah,Dah]],
    [[Dah,Dah,Dit], [Dit,Dah,Dit], [Dah,Dah,Dah], [Dah,Dah,Dah], [Dah]]
  ] == Just "I AM GROOT"
morseToString [[[Dit,Dit,Dit],[Dah,Dah,Dah],[Dit,Dit,Dit,Dah,Dit]]]
    == Nothing
```

Es kann sinnvoll sein, zuerst eine Funktion

```
morseLetterToChar :: MorseLetter -> Maybe Char
```

zu schreiben, die einen einzelnen Morse-Buchstaben umwandelt, und mithilfe dieser Funktion dann eine Funktion

```
morseWordToString :: MorseWord -> Maybe String
```

A	.-	M	--	Y	-.--
B	-...	N	-.	Z	--..
C	-.-.	O	---	0	-----
D	-..	P	.--.	1	.----
E	.	Q	--.-	2	..---
F	..-.	R	.-.	3	...--
G	--.	S	...	4	....-
H	....	T	-	5	.....
I	..	U	..-	6	-....
J	.---	V	...-	7	--...
K	-.-	W	.--	8	---..
L	.-..	X	-..-	9	----.

Abbildung 1: Morse-Kodierung der 26 lateinischen Buchstaben sowie der Ziffern von 0 bis 9

die ein einzelnes Morse-Wort in ein Wort umwandelt, und mithilfe dieser Funktion dann die gesuchte Funktion `morseToString` zu implementieren. Diese zwei Hilfsfunktionen sind allerdings nicht erforderlich und werden nicht bewertet.

Die Funktionen `lookup` aus `Prelude` und `sequence` aus `Control.Monad` könnten nützlich sein. Letztere verhält sich, auf `Maybe` spezialisiert, folgendermaßen:

```
sequence :: [Maybe a] -> Maybe [a]
sequence [] == Just []
sequence [Just 1, Just 2, Just 3] == Just [1,2,3]
sequence [Just 1, Nothing, Just 3] == Nothing
```

Aufgabe H12.2 Morse II

Wie bereits zuvor erwähnt werden Morse-Zeichen normalerweise mithilfe eines konstanten Signals übertragen, also eines Signals, das entweder an oder aus ist. *Dits* und *Dahs* werden dadurch kodiert, dass das Signal eine bestimmte Zeit lang an ist, wobei ein *Dah* so lange ist wie drei *Dits*. Zwischen zwei *Dits*/*Dahs* ist eine Pause der Länge eines *Dits*.

Ein Zeichen wird, wie bereits erwähnt, durch mehrere solche *Dits*/*Dahs* hintereinander kodiert. Um zwei Zeichen voneinander abzugrenzen, wird zwischen Ihnen eine Pause in Länge eines *Dahs* (also drei *Dits*) eingefügt. Zwischen zwei Wörtern wird eine Pause in der Länge von 7 *Dits* eingefügt.

Beispiel: Im nachfolgenden Beispiel steht = für „Signal an“ und _ für „Signal aus“. Der kodierte String ist „O HAI“, was der Morse-Sequence `--- .... .- ..` entspricht.

```
Signal:     ===_===_===_______=_=_=_=___=_===___=_=
Morse:      -   -   -         . . . .   . -     . .
Text:       O                 H         A       I
```

Ihre Aufgabe ist es nun, aus einem solchen Signal die zugehörige Morse-Sequenz zu rekonstruieren:

```
bitStreamToMorse :: Int -> [Bool] -> MorseSequence
```

Ihre Funktion erhält dabei das abgetastete Signal als eine Liste von Booleans, wobei `True` für „Signal an“ und `False` für „Signal aus“ steht. Außerdem erhält sie die Länge eines *Dits*. Da Signalübertragung immer leichte Abweichungen hervorrufen kann, kannes vorkommen, dass ein *Dah* nicht immer exakt drei mal so lang ist wie ein *Dit*. Ihre Funktion soll daher für die gegebene *Dit*-Länge l folgende Schwellwerte anwenden:

- Ein Signal mit einer Länge $< 2l$ ist ein *Dit*
- Ein Signal mit einer Länge $\geq 2l$ ist ein *Dah*
- Eine Pause mit einer Länge $< 2l$ ist eine Pause zwischen zwei *Dit*/*Dahs*
- Eine Pause mit einer Länge $\geq 2l$, aber $< 5l$ ist eine Pause zwischen zwei Zeichen
- Eine Pause mit einer Länge $\geq 5l$ ist eine Pause zwischen zwei Wörtern

Beispiel:

```
bitStreamToMorse 1 "____" == []
bitStreamToMorse 1 "=" == [[[Dit]]]
bitStreamToMorse 1 "===" == [[[Dah]]]
bitStreamToMorse 2 "=_==_===_====_=====" == [[[Dit,Dit,Dit,Dah,Dah]]]
bitStreamToMorse 2 "==__=======_____=====_____________=======__=_______=="
    == [[[Dit,Dah], [Dah]], [[Dah,Dit], [Dit]]]
```

(Aus Platzgründen werden die Signal hier anstelle von `Bool`-Listen in der oben verwendeten Notation geschrieben.)

Aufgabe H12.3 Multiple Mengen

In der Mathematik wird eine Menge M üblicherweise durch die *Element*-Relation definiert. Das heißt, für alle Objekte x gilt entweder $x \in M$ oder $x \notin M$. Der Elementbegriff kann aber auch erweitert werden, in dem man nicht mehr fordert, dass ein Objekt höchstens einmal in einer Menge enthalten sein darf, sondern auch mehrfach.

Implementieren Sie ein Modul `Bag` mit einem abstrakten Datentyp `Bag a`.[1] Folgende Operationen sollen unterstützt werden:

```
empty :: Bag a
insert :: Ord a => a -> Bag a -> Bag a
union :: Ord a => Bag a -> Bag a -> Bag a
diff :: Ord a => Bag a -> Bag a -> Bag a
isSubbag :: Ord a => Bag a -> Bag a -> Bool
count :: Ord a => a -> Bag a -> Integer
```

[1] *Bag* ist der englische Begriff für *Multimenge*

Folgende Semantik muss erfüllt sein:

1. `empty` ist die leere Multimenge, d.h. für alle Werte x muss `count` $x = 0$ gelten
2. `insert` fügt ein Element in die Multimenge ein. Existiert es bereits, ist die Anzahl der Vorkommen zu inkrementieren.
3. `union` bildet die Vereinigung zweier Multimengen. Kommt ein Wert in beiden Multimengen vor, so sind die Anzahl der Vorkommen zu addieren.
4. `diff` berechnet die Differenz zweier Multimengen. Die Anzahl der Vorkommen sind zu subtrahieren. Elemente, die in der zweiten Multimenge häufiger vorkommen als in der ersten, sind im Ergebnis komplett zu eliminieren.
5. `isSubbag` ermittelt, ob die erste Multimenge in der zweiten enthalten ist. Dazu müssen alle Elemente der ersten Multimenge mindestens ebenso häufig in der zweiten Multimenge auftauchen.
6. `count` gibt die Anzahl der Vorkommen eines Elements in einer Multimenge aus. Diese Anzahl ist eine nichtnegative Ganzzahl, d.h. negative Vorkommen existieren nicht.

Aufgabe H12.4 Schnurfinger Grasgnomzucht (Wettbewerbsaufgabe)

In dieser Aufgabe sollen Sie eine Funktion schreiben, die Anagramme von Phrasen erzeugt.

```
anagram :: [String] -> String -> Maybe String
```

Als Eingabe erhält Ihre Funktion eine Liste von bekannten Wörtern und eine Phrase. Sie dürfen davon ausgehen, dass die Phrase nur aus lateinischen Groß- und Kleinbuchstaben sowie Leerzeichen besteht und mindestens ein nicht-Leerzeichen enthält. Die Wortliste kann leer sein, aber jedes Wort enthält mindestens Zeichen und besteht nur aus lateinischen Groß- und Kleinbuchstaben. Die Ausgabe soll ein Anagramm der Phrase sein, welches nur aus bekannten Wörtern besteht. Leerzeichen in der Phrase können ignoriert werden; Groß- und Kleinschreibung zählen sowohl in der Wortliste als auch in der Phrase als äquivalent.[2] Wenn kein solches Anagramm existiert, soll `Nothing` zurückgegeben werden.

Beispiele:

```
anagram english "Jasmin Blanchette" ==
  Just "Mental Bitch Jeans"

anagram english "Master of Competition" ==
  Just "Mince Tomatoes Profit"

anagram [] "Hausaufgaben" ==
  Nothing
```

Eine Beispielwortliste (z.B. `english`) wird als separate Datei bereitgestellt.

[2] Insbesondere ist es daher auch für die Ausgabe egal, ob Sie Groß- oder Kleinschreibung nutzen.

Als Hauptkriterium für den Wettbewerb zählt Skalierbarkeit. Außerdem werden wir völlig subjektiv die Lustigkeit der generierten Anagramme bewerten. Für die Hausaufgabe genügt es, sofern existent, ein beliebiges Anagramm zurückzugeben.

Hinweis: Sie dürfen Ihre Implementation der Multimengen hier verwenden. Allerdings müssen Sie dann darauf achten, dass Sie Dateinamen nach Haskell-Konvention benutzen: `Bag.hs` für die Multimengen und `Exercise_12.hs` für den Rest der Hausaufgaben. Andernfalls wird Ihre Abgabe u.U. nicht in der korrekten Reihenfolge kompiliert.

Wichtig: Wenn Sie diese Aufgabe als Wettbewerbsaufgabe abgeben, stimmen Sie zu, dass Ihr Name ggf. auf der Ergebnisliste auf unserer Internetseite veröffentlicht wird. Sie können diese Einwilligung jederzeit widerrufen, indem Sie eine Email an `fp@fp.in.tum.de` schicken. Wenn Sie nicht am Wettbewerb teilnehmen, sondern die Aufgabe allein im Rahmen der Hausaufgabe abgeben möchten, lassen Sie bitte die {`-WETT-`} . . . {`-TTEW-`} Kommentare weg. Bei der Bewertung Ihrer Hausaufgabe entsteht Ihnen hierdurch kein Nachteil.