Christoph Lüth
Dominik Dietrich
Marcus Ermler
Klaus Hartke
Christian Maeder
Dominik Luecke
Ewaryst Schulz

2. Übungsblatt

Ausgabe: 17.11.08
Abgabe: 01.12.08

4 .... .- ... -.- . .-.. .-.. .. ..-. ..- -. *7 Punkte*

Ein *Kodierungsbaum* ist ein Baum, der an den Knoten mit einzelnen Zeichen (Typ `Char`) markiert ist. Jeder Pfad in einen Kodierungsbaum entspricht einer Kodierung eines Zeichens in einem binären Code. Weil der Baum an den Knoten, nicht den Blättern markiert ist, muss der Code nicht präfix-frei sein.

Ein Beispiel für solch einen Code und Kodierungsbaum ist der Morsecode und der dazugehörige Baum (Abb. 1). Wenn wir links mit `-` und rechts mit `.` kodieren, ist beispielsweise `-..-` die Kodierung des Pfades zum Knoten mit der Markierung `X` (Pfad links-rechts-rechts-links), und damit die Kodierung von `X`. Die Wurzel des Baumes ist unmarkiert; in dem Beispielbaum sind die Knoten der untersten Ebene keine Blätter, sondern Knoten mit jeweils zwei leeren Unterbäumen.

Für unsere Zwecke enthält eine Nachricht im Morsecode drei Zeichen: kurz (`.`), lang (`-`) und Pause (Leerzeichen). Um eine als Zeichenkette vorliegende Nachricht zu dekodieren, trennt man die Zeichenkette zuerst entlang der Leerzeichen in die Kodierung einzelner Buchstaben, und dekodiert diese dann durch Traversion entlang des Kodierungsbaums.

1. Entwerfen Sie einen Datentyp `CodeTree`, der wie oben beschrieben Kodierungsbäume implementiert, und eine Konstante `morseTree`, welche den Kodierungsbaum für den Morsecode repräsentiert.

2. Implementieren Sie eine Funktion

   ```
   decode :: String-> String
   ```

   welche eine im Morsecode gesendete Nachricht dekodiert. Beispiel:

   ```
   decode ".... .- ... -.- . .-.. .-.."
   ```

   sollte zu der Zeichenkette `HASKELL` auswerten.

   Hinweise: Mit der vordefinierten Funktion

   ```
   words :: String-> [String]
   ```

   kann eine Zeichenkette in ihre durch Leerzeichen getrennten Wörter aufgetrennt werden. Jedes dieser Worte ist dann die Kodierung eines einzelnen Zeichens. Eine zu implementierende Funktion

   ```
   dec1 :: CodeTree -> String-> Char
   ```

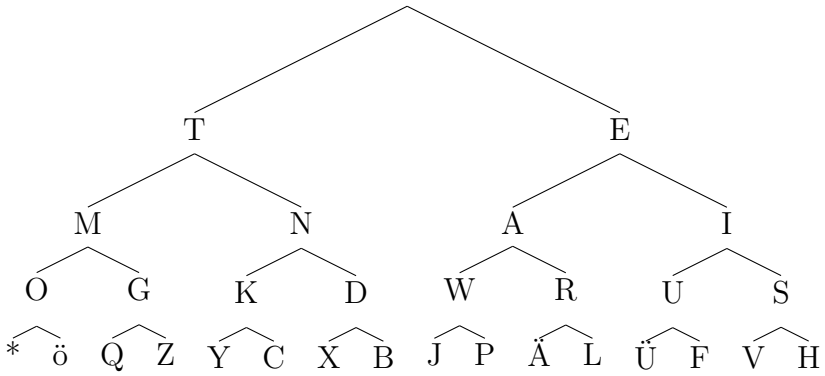

Abbildung 1: Kodierungsbaum für den Morsecode

dekodiert ein einzelnes Zeichen durch rekursiven Abstieg in den Baum.

3. Implementieren Sie eine Funktion

   ```
   encode :: String-> String
   ```

   welche eine Nachricht in den Morsecode kodiert. Leerzeichen, Ziffern und Interpunktion in der Nachricht können ignoriert werden.

   Hinweise: Eine Hilfsfunktion

   ```
   enc1 :: CodeTree-> Char-> String
   ```

   ist nützlich, die ein einzelnes Zeichen kodiert, indem sie rekursiv in den Baum hinabsteigt, bis das Zeichen gefunden wird, und dabei die Kodierung aufbaut. Eine Funktion

   ```
   findIn :: CodeTree-> Char-> Bool
   ```

   die ein Zeichen in einem Kodierungsbaum sucht, hilft zu entscheiden, in welchen Unterbaum man rekursiv absteigt.

5 *Was für ein Typ!* — *3 Punkte*

Geben Sie eine Typherleitung (Vorlesung vom 12.11.08, Regeln auf Folie 14) für folgende Ausdrücke an, oder begründen Sie, warum es keine geben kann. (Wir verwenden hier die vordefinierten Konstruktoren für Listen und Tupel statt der selbstdefinierten `Cons`, `Empty` and `Pair` aus der Vorlesung. Sie können ferner annehmen, dass `'a' :: Char` und `"a" :: [Char]`. `++` ist Konkatenation von Listen mit der Signatur `(++) :: [a]-> [a]-> [a]`.)

1. ```
   \x -> case x of [] -> []
                   y:ys -> ys ++ [y]
   ```

2. ```
   \x y-> ('a' : x, x) : (y, "a") : []
   ```

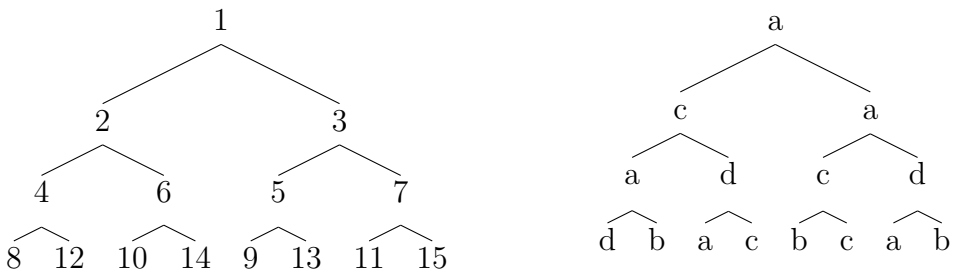

Abbildung 2: Funktionales Array: Indexpositionen und Beispielbaum.

3. `\x -> case x of []     -> x`
   `(ys:y) -> ys`

6 *Alles für Bad und WC.* *10 Punkte*

Die Firma *Sanitär-Kuhlke* hat ein Problem: ihre Lagerverwaltungssoftware läuft noch in COBOL auf einem Rechner, für den Ersatzteile mittlerweise auf dem Archäologieschwarzmarkt gehandelt werden, und ist so langsam, dass die Laufzeiten in Mondzyklen gemessen werden. Haskell to the rescue!

Die Lagerverwaltung identifiziert jeden Artikel mit einer eindeutigen Artikelkennung (AKN). Sie besteht zum einen aus dem *Katalog*, der jeder AKN die Daten des Artikels (Beschreibung, ersteller und Preis in Eurocent) zuordnet, sowie der eigentlichen *Bestandsliste*, die für jede AKN den Bestand enthält.

Um den Zugriff auf Artikel und Bestand schnell zu halten, implementieren wir sowohl Katalog als auch Lagerbestand als ein *funktionales Array* oder Indexbaum, d.h. ein Baum, dessen Knoten eindeutig indiziert sind. Diese Datenstruktur erlaub den Zugriff in $O(\log n)$, und kann gleichzeitig unbeschränkt wachsen. Abb. 2 zeigt links die Indizes der Knoten, und rechts ein Beispiel; in diesem Beispiel gibt der Zugriff mit dem Index 9 den Wert `b`, mit dem Index 6 den Wert `d`, und mit dem Index 3 den Wert `a`.

1. Zuerst implementieren wir den Indexbaum. Dieses ist ein polymorpher Datentyp mit folgenden Operationen:

```
data Array a

empty :: Array a
get   :: Array a-> Int-> Maybe a
upd   :: Array a-> Int-> a-> Array a
add   :: Array a-> a-> (Int, Array a)
list  :: Array a-> [(Int, a)]
```

   - `empty` ist der leere Baum;
   - `get` liest den Baum am angegeben Index aus;
   - `upd` überschreibt den Wert an dem angegeben Index, oder fügt einen neuen Wert ein. Nicht definiert für Werte, die um mehr als eins größer sind als der maximale Index:

- `add` fügt einen Wert an dem nächsten freien Index ein, und gibt diesen Index zusammen mit dem veränderten Baum zurück;
- `list` traversiert den Baum in der Breite (*breadth-first*), und gibt eine dem Index nach geordnete Liste von Paaren aus Index und dem Wert an dem Index zurück.

Welche Korrektheitseigenschaften können Sie über den Operationen formulieren und testen?

2. Für den Katalog modellieren wir den Datentyp `Artikel`, zusammen mit den Selektoren:

```
data Artikel
beschreibung :: Artikel-> String
hersteller   :: Artikel-> String
preis        :: Artikel -> Int
```

Das Lager besteht aus einem `Array Artikel` für den Artikelkatalog (Verzeichnis der bekannten Artikel, indiziert durch die AKN), sowie einem `Array Int` für den Bestand (Anzahl im Lager befindlicher Artikel mit dieser AKN). Eine Invariante ist, dass alle im Katalog eingetragenen Waren auch im Bestand eingetragen sind.

Nach außen (d.h. für die unten zu implementierenden Funktionen) ist die AKN ist eine Verkapselung von `Int`:

```
data AKN = AKN Int
```

Die eigentliche Lagerbuchaltung soll folgende Funktionalität bieten:

```
data Lager
leer         :: Lager
neuerArtikel :: Lager-> String-> String-> Int-> (AKN, Lager)
artikel      :: Lager-> AKN-> String
eingang      :: Lager-> AKN-> Int-> Lager
auslieferung :: Lager-> AKN-> Int-> Lager
inventur     :: Lager-> String
```

- `neuerArtikel` legt einen neuen Artikel mit Beschreibung, Hersteller und Preis im Katalog und im Bestand (mit Bestand 0) an, und gibt die AKN des neuen Artikels sowie das neue Lager zurück;
- `artikel` liefert eine textuelle Beschreibung des Artikels aus dem Katalog, in folgendem Format:

  ```
  *Test> putStrLn (artikel lager a4)
  AKN: 00000004  Preis: 551,00  Bestand: 4
  Badewanne Derby-Top Duo, Kunststoff Acryl, Mittelablauf
  Hersteller: Derby
  ```

  Die AKN soll immer achtstellig mit führenden Nullen angezeigt werden.

```
*Test> putStrLn (inventur lager)
AKN      Artikel                                        Bestand      Preis       Summe
------------------------------------------------------------------------------------
00000001 Wandklosett KARLA                                3 @   147,00 =    441,00
00000002 Standklosett KARLA 24, Abgang waagrecht, *      13 @   185,00 =   2405,00
00000003 Einbauwaschtisch ROSA, Keramik, mit Uebe *       9 @   198,00 =   1782,00
00000004 Badewanne Derby-Top Duo, Kunststoff Acry *       4 @   551,00 =   2204,00
00000005 Badewanne Acryl ATLAS Whirlpool, System  *       2 @  5152,00 =  10304,00
00000006 Stahlduschenwanne MAYA 6,5, 80 x 80 x 6. *       1 @   444,00 =    444,00
00000007 Duschen-Set CROMA-VARIO / UNICA'S                2 @   127,00 =    254,00
00000008 Einbau-Waschtisch LARGO, 49 x 40,5 cm           11 @   269,00 =   2959,00
00000009 Schlauchnippel NEOMATIC 1/2", 19mm              83 @     8,00 =    664,00
00000010 Schlauchnippel NEOMATIC 1/2", 13mm             127 @     6,90 =    876,30
00000011 Schlauchnippel NEOMATIC 1/2", 11mm              67 @     8,90 =    596,30
00000012 Mischdüse NEOMATIC-NEOPERL 1/2"                113 @    12,50 =   1412,50
00000013 Schlauchnippel NEOMATIC 1/2", 16mm              25 @     6,30 =    157,50
------------------------------------------------------------------------------------
                                                                Gesamtsumme:   24499,60
```

Abbildung 3: Beispielinventur

- `eingang` und `auslieferung` verändern den Bestand um die angegebene Menge. Die Menge muss immer positiv sein, und der Bestand soll nie negativ werden können.
- `inventur` macht eine Inventur des Lagers, und liefert eine textuelle Beschreibung in Kurzform aller Artikel im Lager mit ihrem Bestand, sowie am Ende den Gesamtwert des Lagers, in dem Format in Abb. 3. Wichtig ist hierbei die durchgehend bündige Darstellung in Spalten.

Hinweis: Sie können die Funktion `error :: String-> a` benutzen, um Fehler (undefinierte AKN etc.) in der Lagerbuchhaltung (aber nicht in dem anderen Modul) abzufangen.

Die Funktion `putStrLn` gibt eine Zeichenkette auf der Konsole aus:

```
Prelude> let s= "\thello\n\tworld!"
Prelude> s
"\thello\n\tworld!"
Prelude> putStrLn s
        hello
        world!
```