CHALMERS TEKNISKA HÖGSKOLA
Datavetenskap
Jan-Willem Roorda

Friday, January 14th, 2005
Functional Programming
INN040/TDA450

Exam Functional Programming

Friday, January 14th, 2005, 8.30-12.30.
Examiner: John Hughes.
Questions during the exam will be answered by Jan-Willem Roorda, tel 031-7721023.

Permitted aids:
English-Swedish or English-other language dictionary.

- Begin each question on a new sheet. Write your personal number on every sheet.
- You may lose marks for unnecessarily long, complicated, or unstructured solutions.
- Full marks are awarded for solutions which are elegant, efficient, and correct.
- You are free to use any Haskell standard functions, including those whose definitions are attached, unless the question specifically forbids you to do so.
- You may use the solution of an earlier part of a question to help solve a later part, even if you did not succeed in solving the earlier part.
- The exam consists of 3 questions with a total of 60 points. Chalmers students need 24 points to pass the exam. GU students need 28 points to pass.

1. (a) Define the concept of a higher-order function and give an example. (3 p)

   (b) Define the concept of a polymorphic function and give an example. (3 p)

   (c) The following two programs look very similar. Rewrite them using a higher-order polymorphic help function. (You can either define the help function yourself or find it in the Haskell prelude.)

```
findEvens :: [Int] -> [Int]

findEvens [] = []

findEvens (x:xs) = if even x then x : findEvens xs else findEvens xs


findDigits :: String -> String

findDigits [] = []

findDigits (x:xs) = if isDigit x then x : findDigits xs else findDigits xs
```

   (3 p)

   (d) What are the types of the following functions? You do not need to give the most general types—just ones that are correct.

```
f1 n m = (n+m,n==m)

f2 x y z = [[x]] : [y] : z

f3 = map words
```

   (3 p)

   (e) Give an example of an expression that terminates under lazy evaluation, but would not terminate under eager evaluation. (3 p)

2. This question concerns finding change. Suppose you have a collection of coins in your pocket, which we represent by a list of numbers such as `[1,1,5,5,5,10,10]`. Then you can pay 16kr by paying three 5kr coins and a 1kr, but you cannot pay 18kr at all. Suppose the person you are paying also has some coins, say `[1,1,1,5]`. Then you can pay 18kr by paying 20kr, and receiving 2kr in change. We will design functions to decide whether one person can pay another a specific amount, with or without giving change.

   (a) Define a function

   ```
   subsets :: [a] -> [[a]]
   ```

   which given a list, returns a list of all the ways of choosing some of the list elements. For example,

   ```
   subsets [1,2,3] ==
     [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
   ```

   (the order of this list doesn't matter here). (3 p)

   (b) Define a function

   ```
   amounts :: [Int] -> [Int]
   ```

   which given a list of the coins in your pocket, returns a list of all the amounts you can pay using those coins. (3 p)

   (c) Define a function

   ```
   withChange :: [Int] -> [Int] -> [Int]
   ```

   which, given a list of the coins in the first person's pocket, and a list of the coins in the second person's pocket, returns a list of all the amounts the first person can pay the second assuming that the second person is willing to provide change. The result should not contain negative numbers. (3 p)

   (d) The `subsets` function you wrote in part (*a*) is inefficient for this problem, because it ignores the fact that we may have many coins with the same value in our pockets. For example,

   ```
   subsets [1,1] ==
     [[],[1],[1],[1,1]]
   ```

   where the two subsets `[1]` represent choosing different coins, but with the same value. Since we do not care which coin we actually pay with, we do not need to consider these subsets separately.
   We can improve the efficiency of our program by working with *bags* instead, which we represent by lists of pairs of a coin value, and the number of coins of that value in the bag. For example, `[1,1,5,5,5,10,10]` corresponds to the bag `[(1,2),(5,3),(10,2)]`.

      i. Define a function

`bag :: Eq a => [a] -> [(a,Int)]`

which converts a set to a bag. (3 p)

ii. Define a function

`set :: [(a,Int)] -> [a]`

which converts a bag back into a set. (3 p)

iii. Define a function

`subbags :: [(a,Int)] -> [[(a,Int)]]`

which returns a list of every bag we can make by choosing some of the elements of the given bag. For example,

```
subbags [(1,2)] ==
  [[],[(1,1)],[(1,2)]]
```

(4 p)

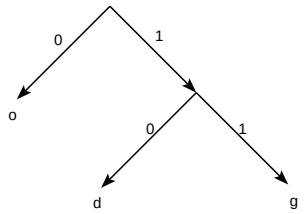
How would you use these functions to improve your function `withChange`? (1 p)

3. This question concerns *Huffman coding*, a form of data compression used, among other things, as part of JPEG image compression. Huffman coding works by analyzing the input to be compressed and assigning each character a *code* (sequence of bits), so that the most common characters are assigned short codes, and rarer characters longer ones. Each character is then replaced by its code, which usually results in a shorter bitstring. For example, if the text to be compressed were just the word "good", the codes assigned might be

'd'	10
'g'	11
'o'	0

and the compressed text would then be "110010" — six bits, which is much less than the 32 bits the uncompressed string requires. (In practice the table of codes must also be stored with the compressed text, which occupies space, so Huffman coding only pays off for larger inputs).

The code is constructed by building a data structure (tree) such as the one below, in which all the characters from the text appear.



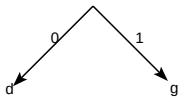
The code for each character is read off by tracing a path from the top (root) of the tree to the character, and outputting a '0' every time one traces an arrow pointing left, and a '1' every time one traces an arrow pointing right. For example, to reach 'd' from the top, we go first right and then left, so the code is "10". Frequently occurring characters are placed higher up in the tree than less common ones, and so are assigned shorter codes.

To construct the tree, we first construct a "trivial tree" (containing only one character) for each character in the input, and assign it a "weight" of the number of times that character occurs. In our example, we construct

- a tree just containing 'd', with weight 1,

- a tree just containing 'g', with weight 1,
- a tree just containing 'o', with weight 2.

Then we combine the two trees with *least* weight, obtaining a new tree whose weight is the sum of the weights of the original trees. In this case, we combine the trees containing 'd' and 'g' to obtain



with weight 2. This step is then repeated until only one tree remains. The codes can be read off from this final tree.

We shall represent these trees in Haskell using the type

```
data Huffman = Leaf Char | Branch Huffman Huffman
```

For example,

- a tree containing one character is represented as a `Leaf` — e.g. `Leaf 'd'`,
- the final tree for our example is represented as `Branch (Leaf 'o') (Branch (Leaf 'd') (Leaf 'g'))`.

Notice that we do not need to store the '0's and '1's in our pictures: given a tree `Branch left right` we know anyway that all the characters in the left branch have codes starting with '0', while all the characters in the right branch have codes starting with '1'.

We shall represent *code tables* using the type

```
type CodeTable = [(Char, String)]
```

where each character is paired with its code, as a string of bits. In our example, the code table would be

```
exampleCodeTable = [('d',"10"), ('g',"11"), ('o',"0")]
```

(a) Define a function

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

such that `encode code text` replaces all the characters in `text` with their codes, taken from `code`. For example

```
Main> encode exampleCodeTable "good"
"110010"
```

(4 p)

(b) Define a function

```
extractCodes :: Huffman -> CodeTable
```

which converts a tree into the corresponding code table. For example,

```
Main> extractCodes (Branch (Leaf 'o') (Branch (Leaf 'd') (Leaf 'g')))
[('o',"0"),('d',"10"),('g',"11")]
```

Define `extractCodes` by pattern matching; that is, write a definition of the form

```
extractCodes (Leaf x) = ...
extractCodes (Branch l r) = ...
```

(5 p)

(c) Define a function

```
buildTree :: [(Int,Huffman)] -> Huffman
```

which converts a list of partial trees and their weights into a complete tree, by repeatedly combining the two trees with least weights. For example,

```
Main> buildTree [(1,Leaf 'd'), (1,Leaf 'g'), (2,Leaf 'o')]
Branch (Leaf 'o') (Branch (Leaf 'd') (Leaf 'g'))
```

(6 p)

(d) Define a function

```
huffmanCode :: String -> CodeTable
```

which constructs the Huffman code table from the given input string. For example,

```
Main> huffmanCode "good"
[('d',"10"),('g',"11"),('o',"0")]
```

(7 p)

(e) Define a function

```
compress :: String -> String
```

which converts an input string to its encoding. For example,

```
Main> compress "good"
"110010"
```

(3 p)