Haskell Programming

Haskell Practice Driving Test

Data Compression

The maximum total is 25.

Credit will be awarded throughout for the style, clarity and the appropriate use of list comprehensions, higher-order functions, etc.

It is important that you **comment out** any code that does not compile before you complete the test.

There will be a THREE mark penalty applied to any program that does not compile.

NOTE: `Data.List`

The template file imports the `Data.List` module and this contains the functions `sort :: Ord a => [a] -> [a]`, `nub :: Eq a => [a] -> [a]` and `insert :: Ord a => a -> [a] -> [a]` that you should use in this exercise. The `nub` function removes duplicate items from a list and the `insert` function inserts an item into an *ordered* list of items, producing an ordered list as a result. For example,

```
*Compression> sort "elements"
"eeelmnst"
*Compression> nub [3,5,2,7,2,5,2,6]
[3,5,2,7,6]
*Compression> insert 5 [2,4,6,8]
[2,4,5,6,8]
```

Data Compression using Huffman Coding

The standard ASCII code (a subset of Unicode) uses 7 bits to encode each character, so, for example, a string with 5 characters requires 35 bits in total. Data compression is all about representing the same information using fewer bits. As an example, one way to compress English text is to exploit the fact that some letters in the language are much more common than others[1]. The idea is to use a binary tree – a so-called *Huffman coding tree* – and to arrange for the most commonly-occurring letters to appear close to the root of the tree. A character is then encoded by a path through the tree: the more common the character, the shorter the path.

Note that we can build a Huffman coding tree from any object type, so long as we can compare any two such objects for equality. As an example, Figure **??** shows the tree for the string "mississippi is missing", i.e. where the object type is `Char`. Using this tree the character `'m'` would be encoded by the path `011`, where `0` and `1` respectively mean "go left" and "go right" through the tree. A list of characters can then be encoded by concatenating the paths corresponding to each character. For example, using the same tree, the string "sips" would be coded as: `111000011`; this uses just 9 bits (binary digits) cf 28 bits for standard ASCII.

When building a Huffman tree we need to keep track of the frequency counts of each item in the list used to build it. Each leaf (`Leaf`) will store both an item from the source list and a count of the number of times that item appeared. Each internal node (`Node`) will store, in addition to its two subtrees, the sum of the frequency counts of all the `Leaf` items in those subtrees. In this sense the tree is *well formed* and all trees in this exercise can be assumed to be well formed. Figure **??** shows the frequency counts in each node, including the leaf nodes, for which there is one for each unique character in the input string. For

[1] It turns out that the most common letter in English is '*e*', followed by '*t*', followed by '*a*', and so on.

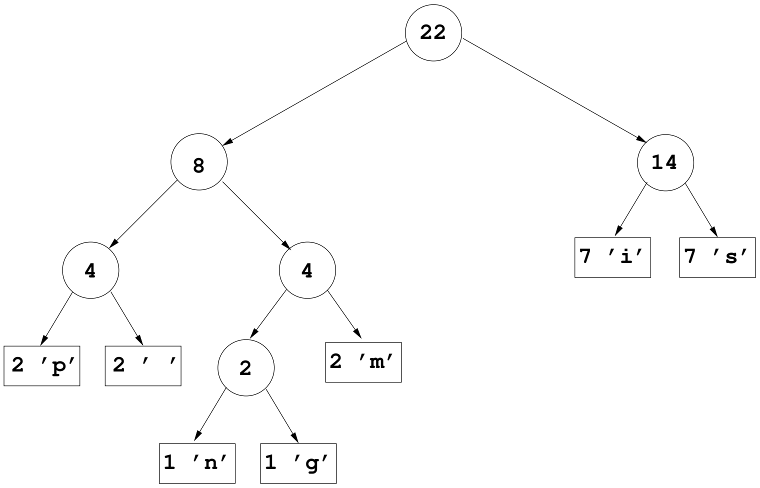

Figure 1: The Huffman coding tree for "mississippi is missing"

example, the tree tells us that the original string had two 'p's, two 'm's, one 'g', seven 'i's and so on. The most common characters were 'i' and 's', which is why they appear nearer the top of the tree than, say, 'g'.

Huffman trees with these added counts can be represented in Haskell by the polymorphic data type `HTree`:

```
data HTree a = Leaf Int a | Node Int (HTree a) (HTree a)
             deriving (Show)
```

Note that this allows trees to contain any item type – the items do not necessarily have to be `Char`s.

One tree is defined to be 'equal' to another if their frequency counts are the same. Similarly, one tree is defined to be 'smaller' than another if its frequency count is smaller. In order to implement this, the type `HTree` has been made an instance of the `Eq` and `Ord` classes in the template file so that the operators `==`, `/=`, `<`, `<=`, `>`, `>=`, `max` and `min` are all defined appropriately on `HTree`s. Thus, for example,

```
Compression> Leaf 3 'c' <= Leaf 1 'd'
False
Compression> max (Leaf 4 'a') (Node 12 (Leaf 8 'x') (Leaf 4 'y'))
Node 12 (Leaf 8 'x') (Leaf 4 'y')
```

Make sure you understand this before reading on.

A function `freqCount :: HTree a -> Int` that returns the frequency count associated with a given tree, has also been defined in the template:

```
freqCount :: HTree a -> Int
freqCount (Leaf n a)
  = n
freqCount (Node n t1 t2)
  = n
```

What to do

1. Define a function `count :: Eq a => a -> [a] -> Int` that given an item and a list of items of the same type will return a count of the number of times the item occurs in a given list. For example, `count 's' "mississippi"` should return `4`.
   (2 marks)

2. Using `count`, define a function `countAll :: Eq a => [a] -> [a] -> [(a, Int)]` that, given two lists, `s1` and `s2`, will compute the number of times each element of the first list occurs in the second. The result should be a list of *pairs*, each of the form `(x,n)`, where `x` is an element (of `s1`) and `n` is the number of times `x` occurs in `s2`. For example `countAll "is" "mississippi"` should return `[('i',4),('s',4)]`. The order of the elements in the result is unimportant.
   (2 marks)

3. Using `countAll` define a function `buildTable :: Eq a => [a] -> [(a, Int)]` that given a list `s` will return, for each element, `e`, of `s`, the pair `(e,n)`, where `n` is the number of times `e` occurs in `s`. For example, `buildTable "mississippi"` should return `[('m',1), ('i',4), ('s',4), ('p',2)]`, in some order. Notice that you will need to remove duplicate list entries at some point. To do this, use the `nub` function from the `Data.List` module.
   (2 marks)

Building a Huffman Tree

You're now going to build a (Huffman) tree from a given source list, `s`. To do this you start by using `buildTable` to build a count of the number of times each element occurs in `s`. Now, for each element `(x,n)` in the result, build the tree `Leaf n x`. You now have a list of trees, all of which are leaves. Now use the `sort` function from the module `Data.List` to sort the trees into *ascending* order of their frequency counts. Note that `sort` will do the right thing here because of the defined ordering on trees in terms of the frequency counts (see above). As an example, given the string `"mississippi is missing"` (this string is predefined and called `testString` in the template file), the `buildTable` function first generates the list `[('m',2), ('i',7), ('s',7), ('p',2), (' ',2), ('n',1), ('g',1)]`. After turning each element into a leaf and sorting the list of leaves you get: `[Leaf 1 'n', Leaf 1 'g', Leaf 2 'm', Leaf 2 'p', Leaf 2 ' ', Leaf 7 'i', Leaf 7 's']`.

Now you do the following to the above list of trees: take the two smallest (i.e. leftmost) trees in the list, *merge* them to form a bigger tree, and then use the `insert` function from the `Data.List` module to insert the merged tree into the remaining elements of the list, i.e. preserving the frequency count order. Observe that the new list now contains one fewer elements than the original. Now repeat the process until the list contains just a single tree – this tree is the required result.

To merge two trees `t1` and `t2` with frequency counts `n1` and `n2` respectively you simply build a new node (`Node`) whose frequency count is `n1+n2`, whose left subtree is the 'smaller' of `t1` and `t2` (i.e. `t1` by virtue of the fact that the original list is ordered) and whose right subtree is the 'larger' of `t1` and `t2` (i.e. `t2`). Note that it's a simple *non-recursive* function.

As an example, starting with the above list we first combine the two leftmost trees giving `Node 2 (Leaf 1 'n') (Leaf 1 'g')`. We then insert this tree into the remainder of the original list giving the new list `[Node 2 (Leaf 1 'n') (Leaf 1 'g'), Leaf 2 'm', Leaf 2 'p', Leaf 2 ' ', Leaf 7 'i', Leaf 7 's']`. Continuing the process we eventually end up with a *singleton* list comprising just the tree shown in Figure **??** – this is called `fig` in the template. Thus...

4. Define a function `merge :: HTree a -> HTree a -> HTree a` that will merge two trees in the manner described above.
   (2 marks)

5. Define a function `reduce :: [HTree a] -> HTree a` that given a non-empty, sorted list of trees will return a single tree by repeated application of `merge` and the `insert` function from `Data.List` as described above. Hint: You can use pattern matching to pick off the first *two* elements of a given list thus:

   ```
   reduce (t1 : t2 : ts) = ...
   ```

   The base case is of the form `reduce [t] = ...`. A precondition is that the argument list is non-empty and already sorted.
   (3 marks)

6. Define a function `buildTree :: Eq a => [a] -> HTree a` that will combine `buildTable`, `sort` and `reduce` to generate the tree corresponding to the given list. A precondition is that the list is non-empty. For example `buildTree "mississippi is missing"` should generate the `HTree` in Figure **??**, i.e. `buildTree testString == fig` should return `True`.
   (4 marks)

7. Define a function `encode :: Eq a => [a] -> HTree a -> Code`, which takes a list of items and a tree and produces the Huffman encoding of the

list in the form of a list of 0s and 1s (type `Code`) representing the bitstring encoding of the list of items. For example, `encode "sign" fig` should return `[1,1,1,0,0,1,0,1,0,1,0,0]`. A precondition is that the given tree can encode each of the items in the list, i.e. each item occurs somewhere in the leaves of the tree.

Hint: there are several ways of doing this: you could use a helper function that checks whether a given item (of type `a`) is in a given `HTree`, or (better?) you could design a helper function that carries round an accumulating parameter representing the path from the root of the `HTree` to the current node.
(4 marks)

8. Define a function `decode :: Code -> HTree a -> [a]`, which takes a Huffman code and a tree and returns the (decoded) list corresponding to the code. For example, `decode [1,1,1,0,0,0,0] fig` should return `"sip"`. Note that, in general, if tree `t` can encode list `xs` then `decode (encode xs t) t` should return `xs`. A precondition is that the code is valid with respect to the tree.
(4 marks)

The hard bit

Only two marks are allocated to these last two questions, so **you should only spend time on them when you have completed the above**.

9. In order to use a Huffman tree for data compression you need to be able to compress the tree as well as the data you are trying to encode; otherwise you can't decode the data! Thus define a function `compressTree :: HTree Char -> [Int]` that will compress a tree of characters into a bitstring (here a list of 0s and 1s) using the following scheme: a `Leaf` is encoded as a 1 followed by 7 bits representing the ordinal value of the character at the leaf, itself encoded as a bitstring; a `Node` is encoded as a 0 followed by the encoding of the left subtree followed by the encoding of the right subtree. For example:

```
*Compression> compressTree fig
[0,0,0,1,1,1,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,1,1,1,0,1,1,1,0,1,1,
1,0,0,1,1,1,1,1,1,0,1,1,0,1,0,1,1,1,0,1,0,0,1,1,1,1,1,0,0,1,1]
```

10. Define a function `rebuildTree :: [Int] -> HTree Char` that will build a Huffman tree of characters from its bitstring encoding. You don't need the frequency counts once a tree has been built so you should just set these to 0 when reconstructing the tree. For example:

```
*Compression> buildTree "suss"
Node 4 (Leaf 1 'u') (Leaf 3 's')
```

```
*Compression> compressTree (buildTree "suss")
[0,1,1,1,1,0,1,0,1,1,1,1,1,0,0,1,1]
*Compression> rebuildTree (compressTree (buildTree "suss"))
Node 0 (Leaf 0 'u') (Leaf 0 's')
```

Note that `rebuildTree .  compressTree` is not the same as `id` because compression loses the frequency counts. Finally, the following constant is defined in the template:

```
secretString :: String
secretString
  = decode code (rebuildTree tree)
  where
    code = [0,1,0,0,0,0,1,1,0,1,0,1,1,1,1,0,0,0,1,
            1,0,0,1,1,1,1,0,0]
    tree = [0,0,0,1,1,1,0,1,0,0,0,1,1,1,0,0,0,0,1,
            0,1,1,1,0,1,1,1,1,1,1,1,1,1,0,0,1,0,1,
            1,1,0,0,0,1,0,0,0,1,0,1,0,0,0,0,1,0,1,
            0,1,0,1,1,0,0,1,1,1,1,0,0,1,1,1,0,1,0,
            0,0,0,0]
```

A SPECIAL PRIZE WILL BE AWARDED TO THE FIRST STUDENT WHO CAN DECODE THE SECRET STRING!

(2 marks in total for the last two questions)

Start coding!

To begin working on your code, go to your **Home folder** where you will find a subfolder called **"Compression"** which contains the **skeleton files**. A sample **Tests.hs** file with some tests has also been provided. Good luck!