The University of Hong Kong

COMP3258: Functional Programming

Assignment 2

Deadline: 23:59, March 24, 2017 (HKT)

---

1 Enum

In Haskell we have several ways to construct lists. One of the simplest is to use ranges. The syntax is to make a list that has the element you want to start the list from followed by two dots followed by the value you want as the final element in the list. Here are some examples using the range syntax:

```
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> ['t'..'z']
"tuvwxyz"
```

You can also specify a step. For example if you want all even numbers between 1 and 10, or every third number between 1 and 10:

```
Prelude> [1,2..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> [1,3..10]
[1,3,5,7,9]
```

The syntax is to separate the first two elements with a comma and then specify what the upper limit is. This also makes possible if you want all the numbers from 10 to 1:

```
Prelude> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
```

You may be wondering how all of these are implemented in Haskell. Well, why not roll up your sleeves and do it yourself!

The types of the functions (simplified a bit) underlying the range syntax are:

```haskell
enumIntFromTo :: Int -> Int -> [Int]
-- Used in Haskell's translation of [n..m]

enumIntFromThenTo :: Int -> Int -> Int -> [Int]
-- Used in Haskell's translation of [n,n'..m]
```

Be aware that `enumIntFromTo` must have its first argument be lower than the second argument, otherwise we'll get an empty list:

```
Prelude> enumFromIntTo 3 1
[]
Prelude> enumFromIntTo 1 3
[1,2,3]
```

Problem 1. (20 pts.) Write your own `enumIntFromTo` and `enumIntFromThenTo` for the types provided. *Do not use range syntax to do so.* They should return the same results if you did using range syntax.

Hint: There are many ways to do, one of them is to use the library function `iterate`.

2 Ciphers

The science of writing secret codes is called *cryptography*. For thousands of years cryptography has made secret messages that only the sender and recipient could read, even if someone captured the messenger and read the coded message. A secret code system is called a *cipher*. In this problem, you'll be writing two ciphers.

2.1 Caesar cipher

One of the simplest is the so called "Caesar cipher", also know as a shift cipher. It is a substitution cipher where each letter in the original message (called the *plaintext*) is replaced with a letter corresponding to a certain number of letters up or down in the alphabet.

(A historical note: The Caesar cipher is named after Julius Caesar, who, according to Suetonius, used it with a shift of 3 to protect messages of military significance.)

You will find variations on this all over the internet. A rightward shift of 3 means that 'A' will become 'D' and 'B' will becomes 'E' for example.

Problem 2. (10 pts.) Your first goal is to write a basic Caesar cipher that shifts rightward:

```
caesar :: Int -> String -> String
```

The first argument is the number of spaces to shift. Note that you want your shift to wrap back around to the begining of the alphabet, so that if you have a rightward shift of 3 from 'z', you end up back at 'c', not somewhere in the vast Unicode hinterlands.

Here is one running example:

```
> caesar 3 "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG"
"WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ"
```

You can assume that messages only use uppercase letters (A-Z) and white spaces.

Hint: You may find the following two functions `ord` and `chr` helpful. These two functions can be used to associate a `Char` with its `Int` representation in the Unicode system and vice versa.

```
chr :: Int -> Char
ord :: Char -> Int
```

Problem 3. (10 pts.) You should also include an `unCaesar` function that will decipher your text as well:

```
unCaesar :: Int -> String -> String
```

2.2 Vigenère cipher

Next we are going to expand on that idea by writing a Vigenère cipher. A Vigenère cipher is another substitution cipher, based on a Caesar cipher, but it uses a series of Caesar ciphers for polyalphabetic substitution. The substitution for each letter in the plaintext is determined by a fixed keyword.

So for example, if you want to encode the message "meet at dawn", the first step is to pick a keyword that will determine which Caesar cipher to use. We'll use the keyword "ALLY" here. You repeat the keyword for as many characters as there are in your original message:

```
MEET AT DAWN
ALLY AL LYAL
```

Now the number of rightward shifts to make to encode each character is set by the character of the keyword that lines up with it. The 'A' means a shift of 0, so the initial M will remain M. But the 'L' for our second character sets a rightward shift of 11, so 'E' becomes 'P'. And so on, so "meet at dawn" encoded with the keyword "ALLY" becomes "MPPR AE OYWY".

Problem 4. (20 pts.) Write a Vigenère cipher that takes a keyword, a plaintext, and produces the encrypted message:

```
vigenere :: String -> String -> String
```

3 Binary Tree

You've already seen one representation of binary trees in Haskell:

```haskell
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
            deriving (Eq, Ord, Show)
```

This tree has a value of type `a` at each node. Each node could be a terminal node, called a *leaf*, or it could branch and have two subtrees. The subtrees are also of type `Tree a`, so this type is recursive, meaning it allows for trees of arbitrary depth.

A special binary tree called "Binary Search Tree" (BST) is useful in many applications. A BST keeps this invariant throughout all its nodes: The value stored in each node must be greater than or equal to any values stored in the left subtree, and less than or equal to any values stored in the right subtree.

BSTs are, in some cases, more efficient for structuring and accessing data than a list, because we know whether to look "left" or "right" to find what we want in a tree.

3.1 Inserting into trees

The `insert` function will insert a value into a tree or, if no tree exists yet, create a new tree. The invariant mentioned above should still hold after you insert a value into a BST.

Problem 5. (5 pts.) Write the function `insert`:

```haskell
insert :: Ord a => a -> Tree a -> Tree a
```

3.2 Convert binary trees to lists

There are three ways to traverse a binary tree:

- Pre-order
    1. Check the current node
    2. Traverse the left subtree by recursively calling the pre-order function
    3. Traverse the right subtree by recursively calling the pre-order function
- In-order
    1. Traverse the left subtree by recursively calling the in-order function
    2. Check the current node
    3. Traverse the right subtree by recursively calling the in-order function
- Post-order
    1. Traverse the left subtree by recursively calling the post-order function
    2. Traverse the right subtree by recursively calling the post-order function
    3. Check the current node

Problem 6. (10 pts.) Write functions to convert `Tree` values to lists.

```
preorder :: Tree a -> [a]
preorder = undefined

inorder :: Tree a -> [a]
inorder = undefined

postorder :: Tree a -> [a]
postorder = undefined
```

Problem 7. (5 pts.) Now we can implement a sorting function that first builds a BST from the input list, then uses one of the above functions to turn it into a sorted list.

```
sort :: Ord a => [a] -> [a]
sort = undefined
```

4 List Zipper

A zipper is a technique of representing an aggregate data structures so that it is convenient to traverse the structure arbitrarily and update its contents. In this problem, we consider a very simple zipper called the list zipper.

A ListZipper is a focused position, with a list of values to the left and to the right:

```
data ListZipper a =
  ListZipper [a] a [a]
  deriving (Eq, Show)
```

The middle value of type a is the element we are focusing on. The left sub-list is the list we have traversed, and the right yet to be traversed.

For example, taking the list [0,1,2,3,4,5], then move focus to the third position, the zipper looks like ListZipper [2,1,0] 3 [4,5,6]. Suppose then we move left on this zipper, it becomes ListZipper [1,0] 2 [3,4,5,6].

Problem 8. (10 pts.) Write a function that converts the given zipper back to a list.

```
toList :: ListZipper a -> [a]
toList = undefined
```

Problem 9. (5 pts.) Write two functions that move the zipper forward and backward.

```
moveLeft :: ListZipper a -> Maybe (ListZipper a)
moveLeft = undefined

moveRight :: ListZipper a -> Maybe (ListZipper a)
moveRight = undefined
```

Some running examples:

```
> moveLeft (ListZipper [3,2,1] 4 [5,6,7])
Just (ListZipper [2,1] 3 [4,5,6,7])
```

```
> moveLeft (ListZipper [] 1 [2,3,4])
Nothing

> moveRight (ListZipper [3,2,1] 4 [5,6,7])
Just (ListZipper [4,3,2,1] 5 [6,7])

> moveRight (ListZipper [3,2,1] 4 [])
Nothing
```

Code style and submission (5 pts.)

All functions should be implemented in a single Haskell file, named as A2_XXX.hs, with XXX replaced by your UID. Your code should be well-written (e.g. proper indentation, names, and type annotations) and documented. *Unless specified, you should not use functions from other libraries apart from the standard library Prelude.* Please submit your solution on Moodle before the deadline.

Plagiarism

Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.