HW 2: More Haskell

CS 440: Programming Languages and Translators
Due Sat Feb 9, 11:59 pm
v. Mon 2/4, 20:42

Notes
For all problems, if you define a function, also include a declaration of the function's type. E.g.,

```haskell
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

All of the problems involve Haskell programming. Turn in the whatever.hs file with your solutions; don't bother with a sample run.

Problems [50 points total] Haskell Programming

1. [4 points] Give the (polymorphic) types of the following functions. (Figure them out yourself first, then enter them into ghci to make sure you're right.) It's ok to uniformly rename variables (e.g. as types, a -> [a] and b -> [b] are equivalent). Include your answer in your *.hs file as a bunch of comments.
   a. `f1a = \(x, y\) -> (y, x)`  \textit{Hint: You need two type variables}
   b. `f1b = \(y\) \rightarrow \[x\]`
   c. `f1c = \(x y z\) \rightarrow \[x : z\]`
   d. `f1d f = length . (\(x\) \rightarrow \(x\))`  \textit{Hint: dot is function composition}

Problems 2 and 3 concern an `iterate` function: For integer n and function f, `iterate n f` should yield the n-fold composition of f. If n ≤ 0, this composition should be the identity function \(id : : a \rightarrow a\). (No need to define \(id\); it is built in.) Since `iterate n f` is a function, the easiest way to test it is to apply it to something. E.g.,

```haskell
iterate 2 sqrt 81 = 3.0
iterate 3 (\(y\) \rightarrow 8+y) 5 = 8+8+8+5 = 29
iterate 4 (\(x\) \rightarrow x-5) (-2) = -22
```

2. [4 points] Write `iterate` using a standard recursive definition that relies on `iterate n f = (f composed with iterate (n-1) f)`, for appropriate n. To avoid conflict with the built-in `iterate` function (which does something different), call yours `iterate1`. Mon 2/4
3. [5 points] Write iterate again (let's call it iterate2 to differentiate it from the previous problem), but this time by using foldl (foldr would works too): Build an n-long list of copies of f and fold the list using function composition as the operator and the id function as the starting value.

For Problems 4 – 7, we'll use this definition of binary trees that include values at the leafs and interior nodes. Tree is polymorphic, so you can have a Tree Int or Tree Float or Tree (Int, Int), etc.

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

4. [5 points] Write a treeEq function that is our own version of tree equality. It should behave just like ==: If x and y are trees, then (treeEq x == treeEq y) == (x == y). For the function type, you need two trees of an equality-testable type. (You can't test treeEq (Leaf sqrt) (Leaf sqrt), for example.)

5. [5 points] Write a treeShow function that is our own version of show. It should take a tree and return a string (a list of characters). It should behave just like show: If x is a tree, then treeShow x == show x. For the function type, you need a tree of showable values. (You can't treeShow (Leaf sqrt), for example.)

6. [5 points] Write a preorder traversal function for trees. E.g., if t is the tree

   (Branch 0 (Leaf 1) (Branch 2 (Leaf 3) (Leaf 4)))

then preorder t = [0, 1, 2, 3, 4]. Write your function so that it uses recursion in the usual way: For a branch, recursively calculate the preorder traversals of the left and right subtrees and join them together (prepending the node value).

7. [5 points] Now write preorder using a tail-recursive helper function: preorder' tree list should calculate the preorder traversal of the tree and prepend it to the list. E.g., if t = (Branch 0 left right) where left = (Leaf 1) and right = (Branch 2 (Leaf 3) (Leaf 4)), then the call preorder' t [5, 6] should return [0, 1, 2, 3, 4, 5, 6]. You still need two recursive calls: The first recursive call prepends the preorder traversal of the right subtree onto the given list; we would get [2, 3, 4, 5, 6]. The second
recursive call would prepend the left subtree to this list and get [1, 2, 3, 4, 5, 6].
Adding on the branch's value 0 gives us [0, 1, 2, 3, 4, 5, 6].
In addition to preorder', define a top-level traversal function preorder2 t =
preorder' t t [].

8. [8 points] The next data structure is for encoding patterns that describe lists of values.
   data Pattern a = P a | POr (Pattern a) (Pattern a)
       | PAnd (Pattern a) (Pattern a) deriving Show
   Your task will be to complete a partially-written match routine that tries to match a
pattern with a list of values. match pat vals = (False, vals) means that the
match failed, and the vals are being returned as is. match pat vals = (True, vals2) means that the pattern matched the head of vals and removed that head with
vals2 leftover.
   match (P 1) [1, 2, 3] = (True, [2, 3])
   match (P 1) [2, 3, 4] = (False, [2, 3, 4])
   match (P 1) [ ] = (False, [ ])
   -- You can't match the empty list
The POr constructor takes two pattern and returns a pattern that tries to match one or the
other. E.g.,
   match ((P 1) `POr` (P 2)) [1, 2, 3] = (True, [2, 3])
   match ((P 1) `POr` (P 2)) [2, 1, 3] = (True, [1, 3])
   match ((P 1) `POr` (P 2)) [7] = (False, [7])
The leftover list comes from whichever match was successful.

The PAnd constructor concatenates two patterns. E.g.,
   match ((P 1) `PAnd` (P 2)) [1, 2, 3] = (True, [3])
because [3] was leftover after matching 1 and then 2.
If the first pattern fails to match, we don't try the second pattern:
   match ((P 1) `PAnd` (P 2)) [4, 5] = (False [4, 5]
The only subtle part is that if the first pattern matches but the second one doesn't, we
don't return the list from the failed second pattern match, we return the original list.
   match ((P 1) `PAnd` (P 2)) [1, 4, 5] = (False, [1, 4, 5])
Here, \texttt{match (P 1) [1, 4, 5]} returned \texttt{(True, [4, 5])} but \texttt{match (P 2) [4, 5]} returned \texttt{(False, [4, 5])}. Nonetheless, the \texttt{PAnd} returned what it started with, namely \texttt{[1, 4, 5]}.

Here is the code for matching the empty list, \texttt{P}, and \texttt{POr}. Your job is to add the code for the \texttt{PAnd} case.

\begin{verbatim}
match _ [] = (False, [])

match (P x) (y : ys)
    = if x == y then (True, ys) else (False, y : ys)

match (POr pat1 pat2) xs
    = case match pat1 xs of
        (True, leftover) -> (True, leftover)  \hspace{1cm}  -- stop if pat1 succeeded
        (False, _ )   -> match pat2 xs
        -- else try pat2

    -- (You add PAnd)
\end{verbatim}