Homework2

* 1. Permutations (35 pts)

(a)

Code

insertIn:: Int->a->[a]->[a]

--Three arguments are Int, a element, and a list. The output is a list.

insertIn 0 x y= x:y

--If the position which the element will insert is first, the list is put x and y.

insertIn n x y= take n y++[x]++drop n y

--If the position is n+1, we take first n elements from y. Then put x.

--we put the rest of y in the end.

insert :: Int->a->[a]->[[a]]

--Three arguments are Int, a element, and a list. The output is a list.

insert (-1) \_ \_ = []

--It's because any list has one more space can be insert than the element.

--We have to let the position stop less than 0, and the position is Int.

--So I let the position is -1, and then return []

insert n x y = insertIn n x y : insert (n-1) x y

--Let the x insert in every position in y.

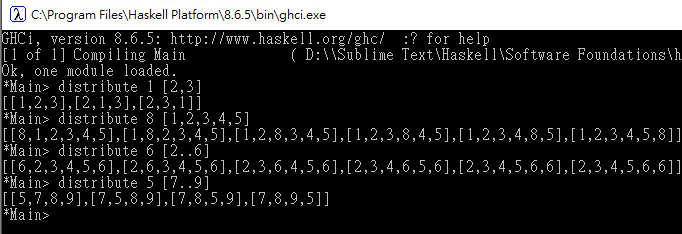
distribute:: a -> [a]->[[a]]

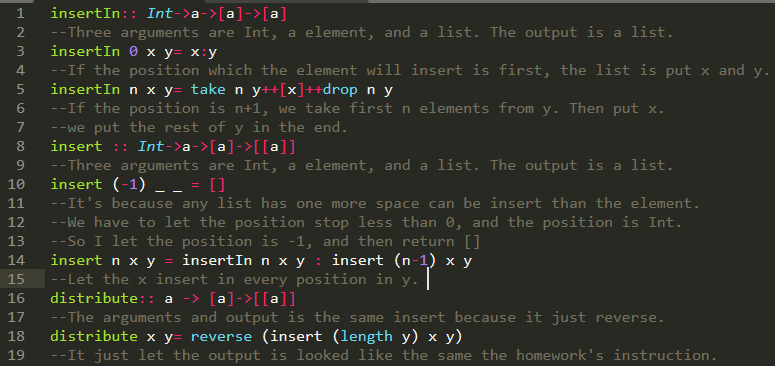
--The arguments and output is the same insert because it just reverse.

distribute x y= reverse (insert (length y) x y)

--It just let the output is looked like the same the homework's instruction.

Code, comment and test(Screenshot)





(b)

Code

permutation::[a]->[[a]]

--The argument is a list, and output is a list of lists.

permutation []=[]

--If the input list is empty, and the output should be empty.

permutation al@(x:xs)

--Let the origin be al.

| length al==1 =[[x]]

| otherwise = concatMap (distribute x) (permutation xs)

--The idea is from permutation in Math. We permutation the last and second to last element.

--Then we get two lists. --Next we put the third element into diffent position in each list.

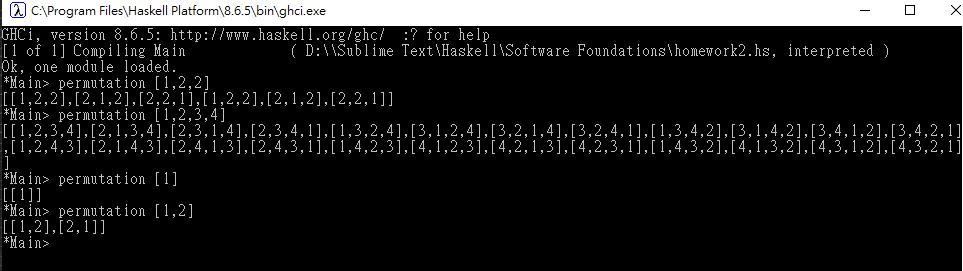
--So the recursive function is we put the nth element into diffent position of each list before we get. ----For example, the input is[a,b,c,d,e]. First, the e is [[e]].

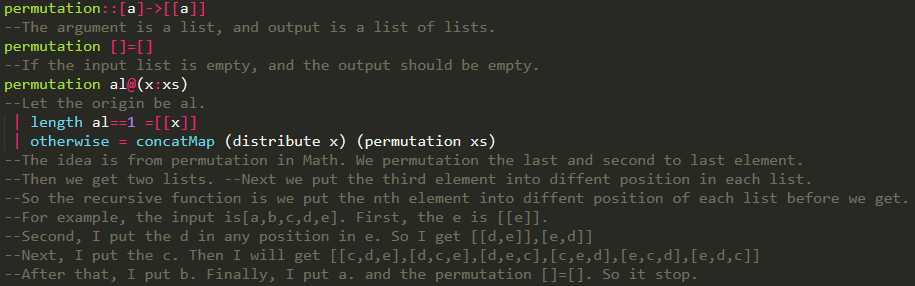
--Second, I put the d in any position in e. So I get [[d,e]],[e,d]]

--Next, I put the c. Then I will get [[c,d,e],[d,c,e],[d,e,c],[c,e,d],[e,c,d],[e,d,c]]

--After that, I put b. Finally, I put a. and the permutation []=[]. So it stop.

Code, comment and test(Screenshot)





* 1. Polymorphic datatypes (40 pts)

(a)

Code

data Tree a = Leaf a | Node a (Tree a) (Tree a) deriving (Show)

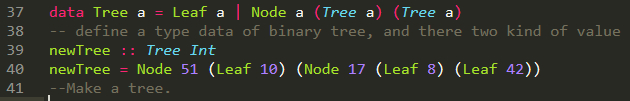
-- define a type data of binary tree, and there two kind of value

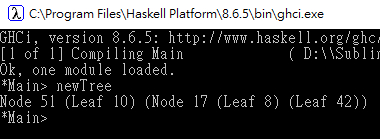
newTree :: Tree Int

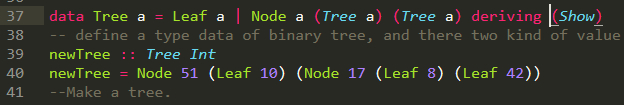
newTree = Node 51 (Leaf 10) (Node 17 (Leaf 8) (Leaf 42))

--Make a tree.

Code, comment and test(Screenshot)







(b)

Code

data Direction = L | R deriving (Show)

--define a type data. There are only L and R

element :: [Direction] -> Tree a -> [a]

element (L:ds) (Node \_ l \_) =element ds l

element (R:ds) (Node \_ \_ r) =element ds r

element (L:\_) (Leaf \_)=[]

element (R:\_) (Leaf \_)=[]

element [] (Node x \_ \_) = [x]

element [] (Leaf x) = [x]

--The idea is because the one node has less two child node or leaf.

--Use the R and L to trace the footprint to find the special element.

listRL::Int->[[Direction]]

listRL 0 = [[]]

--It's because the if the tree level is 0, and the footprint is [[]]

listRL n

| n==1 = [[L],[R]]

| otherwise = map (L:) (listRL (n-1)) ++ map (R:) (listRL (n-1))

--The idea is the footprint of all element on the row.

--Every node have two choice. So put the L and R into each list before we get.

--For example, the root is [[]]. the second is [[R],[L]].

--Third put L and R into [[R],[L]] because the right side and left side both has right and left side.

--So,[[L,R],[L,L],[R,R],[R,L]]. And next level also has right side and left side in each node.

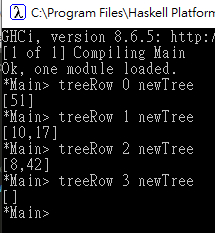
treeRow :: Int -> Tree a -> [a]

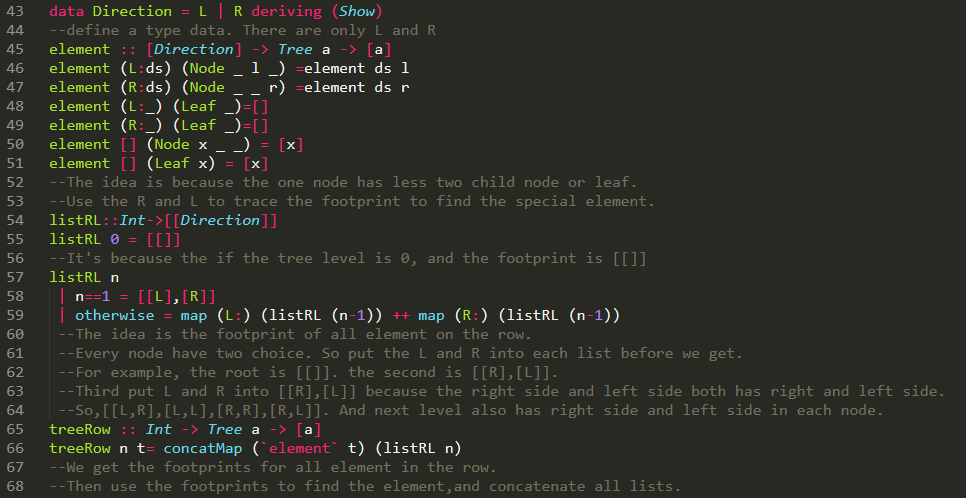
treeRow n t= concatMap (`element` t) (listRL n)

--We get the footprints for all element in the row.

--Then use the footprints to find the element,and concatenate all lists.

Code, comment and test(Screenshot)





©

Code

breadthFirst :: Eq a => Tree a ->[a]

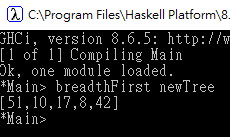
breadthFirst t= concat (takeWhile (/=[]) (map(`treeRow` t) [0..]))

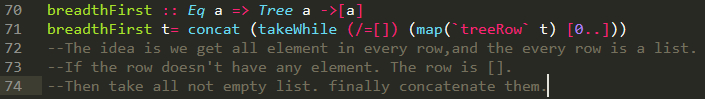
--The idea is we get all element in every row,and the every row is a list.

--If the row doesn't have any element. The row is [].

--Then take all not empty list. finally concatenate them.

Code, comment and test(Screenshot)





It’s because for all lists l :: [a]. So all x which belongs set l. x:: a

Let l = [x1,x2,….xn] => map f (filter p [x1,x2,….xn]) = map f [filter p x1, filter p x2,..filter p xn]

=> [f ( filter p x1 ),f ( filter p x2),….f (filter p xn)]

It’s because f :: a -> a, p::a->bool, and f (p x)= p (f x).

=> [filter p ( f x1), filter p ( f x2),….filter p (f xn)] = filter p [(f x1), (f x2),…(f xn)]

= filter p (map f [x1,x2,….xn] = filter p (map f l)

So, map f ( filter p l) = filter p (map f l)

Reference

(n.d.). Learn You a Haskell for Great Good! Retrieved from <http://learnyouahaskell.com/chapters>

(n.d.). Haskell Reference. Retrieved from <http://zvon.org/other/haskell/Outputprelude/index.html>

The Lecture Notes from the course in learn.unm.edu.