```haskell
{-# OPTIONS -Wall -fwarn-tabs -fno-warn-type-defaults -Werror #-}
{-# LANGUAGE NoImplicitPrelude #-}
{-
  CIS552 HW 2
  Yuexi Ma
  Sebastian Messier
-}

module Main where
import Prelude hiding (takeWhile,all)
import Test.HUnit      -- unit test support

import XMLTypes        -- support file for XML problem (provided)
import Play            -- support file for XML problem (provided)

doTests :: IO ()
doTests = do
  _ <- runTestTT $ TestList [ testFoldr, testTree, testXML ]
  return ()

main :: IO ()
main = do
       doTests
       return ()

-----------------------------------------------------------------------

testFoldr :: Test
testFoldr = TestList [tintersperse, tinvert, ttakeWhile, tfind, tall]

-- The intersperse function takes an element and a list
-- and `intersperses' that element between the elements of the list.
-- For example,
--    intersperse ',' "abcde" == "a,b,c,d,e"


intersperse ::  a -> [a] -> [a]
intersperse _ []       = []
intersperse sep list = init $ concat $ map (:[sep]) list

tintersperse :: Test
tintersperse = "intersperse" ~: TestList
      [ intersperse ',' "abcde"          ~?= "a,b,c,d,e"
      , intersperse 1 [2,3,4]            ~?= [2,1,3,1,4]
      , intersperse 1 [2]                ~?= [2]
      , intersperse 1 []                 ~?= []
      , intersperse 1 [4,5,6,7,8]        ~?= [4,1,5,1,6,1,7,1,8]
      , intersperse True [False, False] ~?= [False, True, False]
      , intersperse 3.2 [1.2,14.2,0.2]  ~?= [1.2,3.2,14.2,3.2,0.2]
      ]

```

```haskell
-- invert lst returns a list with each pair reversed.
-- for example:
--   invert [("a",1),("a",2)] returns [(1,"a"),(2,"a")]

invert :: [(a,b)] -> [(b,a)]
invert = map (\(x,y) -> (y,x))

tinvert :: Test
tinvert = "invert" ~: TestList
      [ invert [("a",1),("a",2)]    ~?= [(1,"a"),(2,"a")]
      , invert ([] :: [(Int,Char)]) ~?= []
      , invert [("a",1)]            ~?= [(1,"a")]
      , invert [((),())]            ~?= [((),())]
      , invert [("a",1),("b",2)]    ~?= [(1,"a"),(2,"b")]
      , invert ([] :: [(Int,Char)]) ~?= []
      ]

-- takeWhile, applied to a predicate p and a list xs,
-- returns the longest prefix (possibly empty) of xs of elements
-- that satisfy p:
-- For example,
--     takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
--     takeWhile (< 9) [1,2,3] == [1,2,3]
--     takeWhile (< 0) [1,2,3] == []

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile f l = foldr (\x res -> if f x then x:res else []) [] l

ttakeWhile :: Test
ttakeWhile = "takeWhile" ~: TestList
      [ takeWhile (< 3) [1,2,3,4,1,2,3,4]     ~?= [1,2]
      , takeWhile (< 9) [1,2,3]               ~?= [1,2,3]
      , takeWhile (< 0) [1,2,3]               ~?= []
      , takeWhile (< 0) []                    ~?= []
      , takeWhile (== 'a') "aabc"             ~?= "aa"
      , takeWhile (== 'a') ""                 ~?= ""
      , takeWhile (< 3) [1,2,3,4,-1,-2,-3,-4] ~?= [1,2]
      , takeWhile (/= 'a') "godard"           ~?= "god"
      , takeWhile (< 0) [1,2,3]               ~?= []
      ]

-- find pred lst returns the first element of the list that
-- satisfies the predicate. Because no element may do so, the
-- answer is returned in a "Maybe".
-- for example:
--     find odd [0,2,3,4] returns Just 3

find :: (a -> Bool) -> [a] -> Maybe a
find f = foldr (\value res -> if f value then Just value else res) Nothing

tfind :: Test
```

```
tfind = "find" ~: TestList
      [ find odd [0,2,3,4]       ~?= Just 3
      , find even [0,2,3,4]      ~?= Just 0
      , find odd [0,2,6,4]       ~?= Nothing
      , find odd []              ~?= Nothing
      , find (<3) [-1,0,1,2,3]   ~?= Just (-1)
      , find even [0,5,7,9]      ~?= Just 0
      , find (/= 'a') "aaaaaa1" ~?= Just '1'
      , find (/= 2) [2,2,2,2,2] ~?= Nothing
      ]

-- all pred lst returns False if any element of lst
-- fails to satisfy pred and True otherwise.
-- for example:
--    all odd [1,2,3] returns False

all  :: (a -> Bool) -> [a] -> Bool
all f = foldr (\value res -> (f value) && res) True

tall :: Test
tall = "all" ~: TestList
      [ all odd [0,2,3,4]        ~?= False
      , all even [2,2,2,4]       ~?= True
      , all even [2,2,2,1]       ~?= False
      , all odd []               ~?= True
      , all (==0) [0,0]          ~?= True
      , all (==0) [0,1]          ~?= False
      , all null [[],[]]         ~?= True
      , all null [[],[1]]        ~?= False
      , all (<3) [1,1,1,1,1]     ~?= True
      , all (/= 1) [0,1,2,3]     ~?= False
      , all null [[],[],[],[]]   ~?= True
      ]

-----------------------------------------------------------------------

testTree :: Test
testTree = TestList [ tinvertTree, ttakeWhileTree,
                      tallTree, tmap2Tree, tzipTree ]

-- | a basic tree data structure
data Tree a = Leaf | Branch a (Tree a) (Tree a) deriving (Show, Eq)

foldTree :: b -> (a -> b -> b -> b) -> Tree a -> b
foldTree e _ Leaf     = e
foldTree e n (Branch a n1 n2) = n a (foldTree e n n1) (foldTree e n n2)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f = foldTree Leaf (\x t1 t2 -> Branch (f x) t1 t2)

-- The invertTree function takes a tree of pairs and returns a new tree
```

```
-- with each pair reversed.  For example:
--     invertTree (Branch ("a",1) Leaf Leaf) returns Branch (1,"a")
--     Leaf Leaf

invertTree :: Tree (a,b) -> Tree (b,a)
invertTree = mapTree (\(x,y) -> (y,x))

tinvertTree :: Test
tinvertTree = "invertTree" ~: TestList
      [ invertTree Leaf
            ~?= (Leaf :: Tree(Int,Int))
      , invertTree (Branch (1,"2") Leaf Leaf)
            ~?= (Branch ("2",1) Leaf Leaf :: Tree([Char],Int))
      , invertTree (Branch (1,"2") Leaf (Branch (1,"2") Leaf Leaf))
            ~?= ((Branch ("2",1) Leaf (Branch ("2",1) Leaf Leaf))
                  :: Tree([Char], Int))
      , invertTree (Branch ((Leaf), Branch 1 Leaf Leaf) Leaf
                (Branch ((Leaf), Branch 2 Leaf Leaf) Leaf Leaf))
            ~?= ((Branch (Branch 1 Leaf Leaf, (Leaf)) Leaf
                  (Branch (Branch 2 Leaf Leaf, (Leaf)) Leaf Leaf))
                  :: Tree(Tree Int,(Tree Int)))
      ]

-- takeWhileTree, applied to a predicate p and a tree t,
-- returns the largest prefix tree of t  (possibly empty)
-- where all elements satisfy p.
-- For example, given the following tree

tree1 :: Tree Int
tree1 = Branch 1 (Branch 2 Leaf Leaf) (Branch 3 Leaf Leaf)

takeWhileTree :: (a -> Bool) -> Tree a -> Tree a
takeWhileTree f = foldTree Leaf (\x t1 t2 -> if f x then Branch x t1 t2
                                             else Leaf)

ttakeWhileTree :: Test
ttakeWhileTree = "takeWhileTree" ~: TestList
      [ takeWhileTree (<0) Leaf   ~?= (Leaf :: Tree Int)
      , takeWhileTree (< 3) tree1 ~?= (Branch 1 (Branch 2 Leaf Leaf) Leaf
                                                 :: Tree Int)
      , takeWhileTree (< 9) tree1 ~?= tree1
      , takeWhileTree (< 0) tree1 ~?= (Leaf :: Tree Int)
      ]


-- allTree pred tree returns False if any element of tree
-- fails to satisfy pred and True otherwise.
-- for example:
--    allTree odd tree1 returns False

countNodes :: Num(b) => a -> b -> b -> b
```

```
countNodes _ counter _ = counter + 1

allTree :: (a -> Bool) -> Tree a -> Bool
allTree f = foldTree True (\x lm rm -> (f x) && lm && rm )

tallTree :: Test
tallTree = "allTree" ~: TestList
      [ allTree (>0) Leaf                                          ~?= True
      , allTree (>2) (Branch 3 Leaf Leaf)                          ~?= True
      , allTree (null) (Branch [] Leaf (Branch [] Leaf Leaf)) ~?= True
      , allTree (>5) (Branch 3 Leaf Leaf)                          ~?= False
      ]


-- WARNING: This one is a bit tricky!  (Hint: the value
-- *returned* by foldTree can itself be a function.)

-- map2Tree f xs ys returns the tree obtained by applying f to
-- to each pair of corresponding elements of xs and ys. If
-- one branch is longer than the other, then the extra elements
-- are ignored.
-- for example:
--    map2Tree (+) (Branch 1 Leaf (Branch 2 Leaf Leaf)) (Branch 3 Leaf Leaf)
--        should return (Branch 4 Leaf Leaf)

map2Tree :: (a -> b -> c) -> Tree a -> Tree b -> Tree c
map2Tree f t1 t2 = (foldTree (\_  -> Leaf)
                        (\x lm rm -> superTreeFunction f x lm rm) t1) t2
  where
      superTreeFunction _ _ _ _ Leaf = Leaf
      superTreeFunction f x lm rm (Branch v lb rb)
          = Branch (f x v) (lm lb) (rm rb)

tmap2Tree :: Test
tmap2Tree = "map2Tree" ~: TestList
      [ map2Tree (+) (Branch 1 Leaf (Branch 2 Leaf Leaf))
                (Branch 3 Leaf Leaf)
            ~?= (Branch 4 Leaf Leaf :: Tree Int)
      , map2Tree (+) (Branch 1 Leaf Leaf)
                (Branch 3 Leaf (Branch 2 Leaf Leaf))
            ~?= (Branch 4 Leaf Leaf :: Tree Int)
      , map2Tree (\a b -> [a, b, a+b]) (Branch 1 Leaf Leaf)
                (Branch 3 Leaf (Branch 2 Leaf Leaf))
            ~?= (Branch [1,3,4] Leaf Leaf :: Tree [Int])
      , map2Tree (\a b -> [a, b, a+b]) (Branch 1 (Branch 2 Leaf Leaf)
                (Branch 2 Leaf Leaf)) (Branch 3 Leaf (Branch 2 Leaf Leaf))
            ~?= (Branch [1,3,4] Leaf (Branch [2,2,4] Leaf Leaf)
                    :: Tree [Int])
      , map2Tree (+) Leaf (Branch 3 Leaf Leaf)
            ~?= Leaf
      , map2Tree (+) (Branch 3 Leaf Leaf) Leaf
```

```
            ~?= Leaf
      ]

-- zipTree takes two trees and returns a tree of corresponding pairs. If
-- one input branch is smaller, excess elements of the longer branch are
-- discarded.
-- for example:
--    zipTree (Branch 1 (Branch 2 Leaf Leaf) Leaf) (Branch True Leaf Leaf)
--        returns
--            (Branch (1,True) Leaf Leaf)

-- To use foldTree, you'll need to think about this one in
-- the same way as part (d).

zipTree :: Tree a -> Tree b -> Tree (a,b)
zipTree = map2Tree (\x y -> (x, y))

tzipTree :: Test
tzipTree = "zipTree" ~: TestList
      [ zipTree (Branch 3 Leaf (Branch 0 (Branch 1 Leaf Leaf) Leaf))
                (Branch False (Branch True Leaf Leaf) (Branch True Leaf Leaf))
          ~?= (Branch (3,False) Leaf (Branch (0,True) Leaf Leaf)
                  :: Tree(Int, Bool))
      , zipTree Leaf Leaf
          ~?= (Leaf :: Tree (Int, Bool))
      , zipTree (Branch [] Leaf (Branch [] Leaf Leaf))
                (Branch 1 (Branch 2 (Branch 3 Leaf Leaf) Leaf) Leaf)
          ~?= (Branch ([],1) Leaf Leaf :: Tree ([Int],Int))
      , zipTree (Branch Leaf Leaf Leaf) (Branch Leaf Leaf Leaf)
          ~?= (Branch (Leaf,Leaf) Leaf Leaf :: Tree (Tree Int,Tree Int))
      ]
-------------------------------------------------------------------------
foldXML :: (String -> b)
           -> (String -> [b] -> b)
           -> SimpleXML
           -> b
foldXML fLeaf _ (PCDATA val) = fLeaf val
foldXML fLeaf fBranch (Element val lst)
        = fBranch val $ map (foldXML fLeaf fBranch) lst


linebreak :: SimpleXML
linebreak = Element "br" []

formatPlay :: SimpleXML -> SimpleXML
formatPlay xml = Element "html" (foldXML fLeaf fBranch xml)
  where
    fLeaf val = [PCDATA val]
    fBranch "PLAY" lst      = [Element "body" $ concat lst]
    fBranch "TITLE" lst     = [Element "h1" $ concat lst]
    fBranch "PERSONAE" lst  = (Element "h2" [PCDATA "Dramatis Personae"])
```

```
                                    : (concat lst)
    fBranch "PERSONA" [[persona]]
                              = [persona, linebreak]
    fBranch "ACT" ([Element _ [act]]:xs)
                              = (Element ("h2") [act]):(concat xs)
    fBranch "SCENE" ([Element _ [scene]]:xs)
                              = (Element ("h3") [scene]):(concat xs)
    fBranch "SPEECH" ([Element _ [speech]]:xs)
                              = (Element "b" [speech]):linebreak:(concat xs)
    fBranch "LINE" [[line]] = [line, linebreak]
    fBranch val lst         = [Element val $ concat lst]

firstDiff :: Eq a => [a] -> [a] -> Maybe ([a],[a])
firstDiff [] [] = Nothing
firstDiff (c:cs) (d:ds)
    | c==d = firstDiff cs ds
    | otherwise = Just (c:cs, d:ds)
firstDiff cs ds = Just (cs,ds)

-- | Test the two files character by character, to determine whether
-- they match.
testResults :: String -> String -> IO ()
testResults file1 file2 = do
  f1 <- readFile file1
  f2 <- readFile file2
  case firstDiff f1 f2 of
    Nothing -> return ()
    Just (cs,ds) -> assertFailure msg where
      msg  = "Results differ: '" ++ take 20 cs ++
             "' vs '" ++ take 20 ds

testXML :: Test
testXML = TestCase $ do
  writeFile "dream.html" (xml2string (formatPlay play))
  testResults "dream.html" "sample.html"
```