```haskell
{-# OPTIONS -Wall -fwarn-tabs -fno-warn-type-defaults #-}
{-# LANGUAGE NoImplicitPrelude, ScopedTypeVariables #-}

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 ins =
  foldr (\ x re -> if null re then [x] else x : ins : re) []
tintersperse :: Test
tintersperse = "intersperse" ~:
  TestList [ intersperse ',' "abcde"     ~?= "a,b,c,d,e",
             intersperse 2 ([] :: [Int]) ~?= ([] :: [Int]),
             intersperse 'a' "b"         ~?= "b",
             intersperse 3 [4, 5]        ~?= [4, 3, 5],
             intersperse ([] :: [Bool]) [[True, False], [True]] ~?=
               [[True, False], [], [True]],
             intersperse '.' "abc"       ~?= "a.b.c",
             intersperse 0 [1, 2, 3, 4, 5] ~?= [1, 0, 2, 0, 3, 0, 4, 0, 5],
             intersperse '-' []          ~?= [] ]


-- 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)]
```

```
52  invert = map (\ (x, y) -> (y, x))
53  tinvert :: Test
54  tinvert = "invert" ~:
55    TestList [ invert ([] :: [(Int, Char)]) ~?= ([] :: [(Char, Int)]),
56               invert [("a", 1), ("a", 2)]  ~?= [(1, "a"), (2, "a")],
57               invert [(2, 1), (1, 1)]      ~?= [(1, 2), (1, 1)],
58               invert [("a", 1), ("a", 2)]  ~?= [(1, "a"), (2, "a")],
59               invert ([] :: [(Int, Char)]) ~?= [] ]
60
61
62  -- takeWhile, applied to a predicate p and a list xs,
63  -- returns the longest prefix (possibly empty) of xs of elements
64  -- that satisfy p:
65  -- For example,
66  --     takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
67  --     takeWhile (< 9) [1,2,3] == [1,2,3]
68  --     takeWhile (< 0) [1,2,3] == []
69
70  takeWhile :: (a -> Bool) -> [a] -> [a]
71  takeWhile f = foldr (\ x re -> if f x then x : re else []) []
72  ttakeWhile :: Test
73  ttakeWhile = "takeWhile" ~:
74    TestList [ takeWhile (< 3) [1, 2, 3, 4, 1, 2, 3, 4] ~?= [1, 2],
75               takeWhile (< 9) [1, 2, 3]                 ~?= [1, 2, 3],
76               takeWhile (< 0) [1, 2, 3]                 ~?= [],
77               takeWhile null [[], "", "abc", "az"]      ~?= [[], ""],
78               takeWhile odd [1, 3, 5, 2, 4]             ~?= [1, 3, 5] ]
79
80  -- find pred lst returns the first element of the list that
81  -- satisfies the predicate. Because no element may do so, the
82  -- answer is returned in a "Maybe".
83  -- for example:
84  --     find odd [0,2,3,4] returns Just 3
85
86  find :: (a -> Bool) -> [a] -> Maybe a
87  find f = foldr (\ x re -> if f x then Just x else re) Nothing
88  tfind :: Test
89  tfind = "find" ~:
90    TestList [ find odd [0, 2, 3, 4]          ~?= Just 3,
91               find (< 3) [5, 3, 2, 1]        ~?= Just 2,
92               find null ["ab", "cd", "eff"] ~?= Nothing,
93               find odd ([] :: [Int])         ~?= Nothing,
94               find even [1, 3, 5, 7]         ~?= Nothing,
95               find (> 100) []                ~?= Nothing ]
96
97  -- all pred lst returns False if any element of lst
98  -- fails to satisfy pred and True otherwise.
99  -- for example:
100 --    all odd [1,2,3] returns False
101
102 all  :: (a -> Bool) -> [a] -> Bool
```

```haskell
all f = foldr (\ x re -> f x && re) True
tall :: Test
tall = "all" ~:
  TestList [ all odd [1, 2, 3]       ~?= False,
             all null [[], []]       ~?= True,
             all (< 3) []            ~?= True,
             all (< 9) [2, 3, 4, 8] ~?= True,
             all odd [1, 3, 5]       ~?= True,
             all even [2, 4, 6]      ~?= True,
             all (> 5) [4, 5, 6]     ~?= False,
             all (> 3) [4, 5, 6]     ~?= 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)

-- | trees for testing
lecTree :: Tree Int
lecTree = Branch 5 (Branch 2 (Branch 1 Leaf Leaf) (Branch 4 Leaf Leaf))
  (Branch 9 Leaf (Branch 7 Leaf Leaf))

pairTree :: Tree (Int, Int)
pairTree = Branch (1,2) ( Branch (3,4) Leaf Leaf)
  (Branch (5,6) (Branch (7,8) Leaf Leaf) Leaf)

pairTreeInv :: Tree (Int, Int)
pairTreeInv = Branch (2,1) ( Branch (4,3) Leaf Leaf)
  (Branch (6,5) (Branch (8,7) Leaf Leaf) Leaf)

-- 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 (\ (a, b) -> (b, a))
tinvertTree :: Test
tinvertTree = "invertTree" ~:
  TestList [ invertTree (Leaf :: Tree (Int, Int)) ~?= (Leaf :: Tree (Int, Int)),
             invertTree pairTree ~?= pairTreeInv,
```

```
              invertTree (Leaf :: Tree (Char, Bool)) ~?=
                (Leaf :: Tree (Bool, Char))]


-- 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 (< 3) tree1  returns Branch 1 (Branch 2 Leaf Leaf) Leaf
--     takeWhileTree (< 9) tree1  returns tree1
--     takeWhileTree (< 0) tree1  returns Leaf

takeWhileTree :: (a -> Bool) -> Tree a -> Tree a
takeWhileTree f =
  foldTree Leaf (\ x lt rt -> if f x then Branch x lt rt else Leaf)
ttakeWhileTree :: Test
ttakeWhileTree = "takeWhileTree" ~:
  TestList [ takeWhileTree (< 3) tree1  ~?= Branch 1 (Branch 2 Leaf Leaf) Leaf,
             takeWhileTree (< 9) tree1  ~?= tree1,
             takeWhileTree (< 0) tree1  ~?= Leaf,
             takeWhileTree (> 5) (Leaf :: Tree Int)
                                        ~?= Leaf ]


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

allTree :: (a -> Bool) -> Tree a -> Bool
allTree f = foldTree True (\ x lt rt -> f x && (lt && rt))
tallTree :: Test
tallTree = "allTree" ~:
  TestList [ allTree odd tree1                ~?= False,
             allTree ((< 8) . fst) pairTree   ~?= True,
             allTree (< 4) tree1              ~?= True,
             allTree (>= 1) lecTree           ~?= True,
             allTree (< 3) tree1              ~?= False,
             allTree (< 9) tree1              ~?= True,
             allTree (< 0) tree1              ~?= False,
             allTree (> 5) (Leaf :: Tree Int) ~?= True ]

-- | trees for testing
tree2 :: Tree Int
tree2 = Branch 1 Leaf (Branch 2 Leaf Leaf)
tree3 :: Tree Int
tree3 = Branch 3 Leaf Leaf
```

```haskell
tree4 :: Tree Int
tree4 = Branch 4 Leaf Leaf
prod1AndLec :: Tree Int
prod1AndLec = Branch 5 (Branch 4 Leaf Leaf) (Branch 27 Leaf Leaf)

-- 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 :: forall a b c . (a -> b -> c) -> Tree a -> Tree b -> Tree c
map2Tree f = foldTree (const Leaf) funcMaker where
  funcMaker :: a -> (Tree b -> Tree c) -> (Tree b -> Tree c) ->
    (Tree b -> Tree c)
  funcMaker x lFn rFn = processTree where
    processTree :: Tree b -> Tree c
    processTree Leaf = Leaf
    processTree (Branch v lt rt) = Branch (f x v) (lFn lt) (rFn rt)

tmap2Tree :: Test
tmap2Tree = "map2Tree" ~:
  TestList [ map2Tree (+) tree2 tree3   ~?= tree4,
             map2Tree (-) tree2 tree3   ~?= Branch (-2) Leaf Leaf,
             map2Tree (-) tree3 tree2   ~?= Branch 2 Leaf Leaf,
             map2Tree (*) lecTree Leaf  ~?= Leaf,
             map2Tree (*) tree1 lecTree ~?= prod1AndLec,
             map2Tree (\ n1 n2 -> show (div n1 n2)) lecTree tree2
                                        ~?= Branch "5" Leaf
                                              (Branch "4" 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 =
```

```
  TestList [ zipTree tree2 tree3                  ~?= Branch (1, 3) Leaf Leaf,
             zipTree tree3 tree2                  ~?= Branch (3, 1) Leaf Leaf,
             zipTree lecTree (Leaf :: Tree Bool)  ~?= Leaf,
             zipTree (Leaf :: Tree Char) lecTree  ~?= Leaf,
             zipTree lecTree tree2                ~?=
              Branch (5, 1) Leaf (Branch (9, 2) Leaf Leaf),
             zipTree tree1 pairTree               ~?=
               Branch (1, (1, 2)) (Branch (2, (3, 4)) Leaf Leaf)
                 (Branch (3, (5, 6)) Leaf Leaf) ]

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



-- | HTML tags
htmlTag :: ElementName
htmlTag = "html"
bodyTag :: ElementName
bodyTag = "body"
header :: Char
header = 'h'
h1Tag :: ElementName
h1Tag = "h1"
h2Tag :: ElementName
h2Tag = "h2"
h3Tag :: ElementName
h3Tag = "h3"
brTag :: ElementName
brTag = "br"
bTag :: ElementName
bTag = "b"

-- | XML tags
playTag :: ElementName
playTag = "PLAY"
titleTag :: ElementName
titleTag = "TITLE"
personaeTag :: ElementName
personaeTag = "PERSONAE"
personaTag :: ElementName
personaTag = "PERSONA"
actTag :: ElementName
actTag = "ACT"
sceneTag :: ElementName
sceneTag = "SCENE"
speechTag :: ElementName
speechTag = "SPEECH"
speakerTag :: ElementName
speakerTag = "SPEAKER"
lineTag :: ElementName
lineTag = "LINE"
```

```haskell

-- | other constants
errorMsg :: String
errorMsg = "ERROR"
brElem :: SimpleXML
brElem = Element brTag []
dramPerElem :: SimpleXML
dramPerElem = Element h2Tag [PCDATA "Dramatis Personae"]

-- | apply function and interleave an element
modifyInterleave :: (a -> a) -> a -> [a] -> [a]
modifyInterleave _ _ []  = []
modifyInterleave f ins l = intersperse ins (map f l) ++ [ins]

-- | convert an element to header element based on the number passed in
convertToHeader :: Int -> SimpleXML -> SimpleXML
convertToHeader lvl (Element _ body) = Element (header : show lvl) body
convertToHeader _     _                =
  PCDATA (errorMsg ++ "convertToHeader")

-- | get first child of an Element
getFirstChild :: SimpleXML -> SimpleXML
getFirstChild (Element _ (x : _)) = x
getFirstChild _                   = PCDATA (errorMsg ++ "getFirstChild")

-- | replace the name of an XML element
replaceName :: ElementName -> SimpleXML -> SimpleXML
replaceName newName (Element _ body) = Element newName body
replaceName _       _                = PCDATA (errorMsg ++ "replaceName")

-- | convert from tag of parent in XML play to level of header
headerLvl :: ElementName -> Int
headerLvl tag
  | tag == playTag  = 1
  | tag == actTag   = 2
  | tag == sceneTag = 3
  | otherwise       = -1

-- | convert speaker to html
convertSpeaker :: SimpleXML -> [SimpleXML]
convertSpeaker s = [replaceName bTag s, brElem]

-- | convert title to html given parent tag
convertTitle :: ElementName -> SimpleXML -> SimpleXML
convertTitle parentTag = convertToHeader (headerLvl parentTag)

-- | convert the non-repeating leading content in a play element
convertFirsts :: SimpleXML -> [SimpleXML]
convertFirsts (Element tag body)
  | tag == speechTag                = convertSpeaker (head body)
  | tag == actTag || tag == sceneTag = [convertTitle tag (head body)]
```

```
  | tag == playTag                    = convertTitle tag (head body) :
      dramPerElem : convertContent (head (tail body))
  | otherwise                         = [PCDATA (errorMsg ++ "convertFirsts")]
convertFirsts _ = [PCDATA (errorMsg ++ "convertFirsts")]

-- | the the first children and interleave with br elements
getFirstsAndInsBr :: [SimpleXML] -> [SimpleXML]
getFirstsAndInsBr = modifyInterleave getFirstChild brElem

-- | convert XML PLAY content to html and concat
convertListContent :: [SimpleXML] -> [SimpleXML]
convertListContent = concatMap convertContent

-- | convert all of the content in a play element
convertContent :: SimpleXML -> [SimpleXML]
convertContent e@(Element tag body)
  | tag == speechTag                  = convertFirsts e ++
      getFirstsAndInsBr (tail body)
  | tag == personaeTag                = getFirstsAndInsBr body
  | tag == sceneTag || tag == actTag  = convertFirsts e ++
      convertListContent (tail body)
  | tag == playTag                    = convertFirsts e ++
      convertListContent (tail (tail body))
  | otherwise                         = [PCDATA (errorMsg ++ "convertContent")]
convertContent _ = [PCDATA (errorMsg ++ "convertContent")]

-- | convert an XML PLAY into HTML
formatPlay :: SimpleXML -> SimpleXML
formatPlay orig = Element htmlTag [Element bodyTag (convertContent orig)]

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
```

```
409     writeFile "dream.html" (xml2string (formatPlay play))
410     testResults "dream.html" "sample.html"
```