

```

1 {-# OPTIONS -Wall -fwarn-tabs -fno-warn-type-defaults -Werror #-}
2 {-# LANGUAGE NoImplicitPrelude #-}
3 {-
4     CIS552 HW 2
5     Yuexi Ma
6     Sebastian Messier
7 -}
8
9 module Main where
10 import Prelude hiding (takeWhile, all)
11 import Test.HUnit      -- unit test support
12
13 import XMLTypes        -- support file for XML problem (provided)
14 import Play            -- support file for XML problem (provided)
15
16 doTests :: IO ()
17 doTests = do
18     _ <- runTestTT $ TestList [ testFoldr, testTree, testXML ]
19     return ()
20
21 main :: IO ()
22 main = do
23     doTests
24     return ()
25
26 -----
27
28 testFoldr :: Test
29 testFoldr = TestList [tintersperse, tinvert, ttakeWhile, tfind, tall]
30
31 -- The intersperse function takes an element and a list
32 -- and 'intersperses' that element between the elements of the list.
33 -- For example,
34 --     intersperse ',' "abcde" == "a,b,c,d,e"
35
36
37 intersperse :: a -> [a] -> [a]
38 intersperse _ [] = []
39 intersperse sep list = init $ concat $ map (:[sep]) list
40
41 tintersperse :: Test
42 tintersperse = "intersperse" ~: TestList
43     [ intersperse ',' "abcde" ~?= "a,b,c,d,e"
44     , intersperse 1 [2,3,4] ~?= [2,1,3,1,4]
45     , intersperse 1 [2] ~?= [2]
46     , intersperse 1 [] ~?= []
47     , intersperse 1 [4,5,6,7,8] ~?= [4,1,5,1,6,1,7,1,8]
48     , intersperse True [False, False] ~?= [False, True, False]
49     , intersperse 3.2 [1.2,14.2,0.2] ~?= [1.2,3.2,14.2,3.2,0.2]
50     ]
51

```

```

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

```

```

103 tfind = "find" ~: TestList
104   [ find odd [0,2,3,4]      ~= Just 3
105     , find even [0,2,3,4]   ~= Just 0
106     , find odd [0,2,6,4]    ~= Nothing
107     , find odd []           ~= Nothing
108     , find (<3) [-1,0,1,2,3] ~= Just (-1)
109     , find even [0,5,7,9]   ~= Just 0
110     , find (/= 'a') "aaaaaa1" ~= Just '1'
111     , find (/= 2) [2,2,2,2,2] ~= Nothing
112   ]
113
114 -- all pred lst returns False if any element of lst
115 -- fails to satisfy pred and True otherwise.
116 -- for example:
117 --   all odd [1,2,3] returns False
118
119 all :: (a -> Bool) -> [a] -> Bool
120 all f = foldr (\value res -> (f value) && res) True
121
122 tall :: Test
123 tall = "all" ~: TestList
124   [ all odd [0,2,3,4]      ~= False
125     , all even [2,2,2,4]   ~= True
126     , all even [2,2,2,1]   ~= False
127     , all odd []           ~= True
128     , all (==0) [0,0]      ~= True
129     , all (==0) [0,1]      ~= False
130     , all null [[],[[]]]   ~= True
131     , all null [[],[1]]    ~= False
132     , all (<3) [1,1,1,1,1] ~= True
133     , all (/= 1) [0,1,2,3] ~= False
134     , all null [[],[[]],[[]],[[]]] ~= True
135   ]
136
137 -----
138
139 testTree :: Test
140 testTree = TestList [ tinvertTree, ttakeWhileTree,
141                      tallTree, tmap2Tree, tzipTree ]
142
143 -- | a basic tree data structure
144 data Tree a = Leaf | Branch a (Tree a) (Tree a) deriving (Show, Eq)
145
146 foldTree :: b -> (a -> b -> b -> b) -> Tree a -> b
147 foldTree e _ Leaf      = e
148 foldTree e n (Branch a n1 n2) = n a (foldTree e n n1) (foldTree e n n2)
149
150 mapTree :: (a -> b) -> Tree a -> Tree b
151 mapTree f = foldTree Leaf (\x t1 t2 -> Branch (f x) t1 t2)
152
153 -- The invertTree function takes a tree of pairs and returns a new tree

```

```

154 -- with each pair reversed.  For example:
155 --     invertTree (Branch ("a",1) Leaf Leaf) returns Branch (1,"a")
156 --     Leaf Leaf
157
158 invertTree :: Tree (a,b) -> Tree (b,a)
159 invertTree = mapTree (\(x,y) -> (y,x))
160
161 tinvertTree :: Test
162 tinvertTree = "invertTree" ~: TestList
163   [ invertTree Leaf
164     ~?= (Leaf :: Tree(Int,Int))
165   , invertTree (Branch (1,"2") Leaf Leaf)
166     ~?= (Branch ("2",1) Leaf Leaf :: Tree([Char],Int))
167   , invertTree (Branch (1,"2") Leaf (Branch (1,"2") Leaf Leaf))
168     ~?= ((Branch ("2",1) Leaf (Branch ("2",1) Leaf Leaf))
169         :: Tree([Char], Int))
170   , invertTree (Branch ((Leaf), Branch 1 Leaf Leaf) Leaf
171     (Branch ((Leaf), Branch 2 Leaf Leaf) Leaf Leaf))
172     ~?= ((Branch (Branch 1 Leaf Leaf, (Leaf)) Leaf
173         (Branch (Branch 2 Leaf Leaf, (Leaf)) Leaf Leaf))
174         :: Tree(Tree Int,(Tree Int)))
175   ]
176
177 -- takeWhileTree, applied to a predicate p and a tree t,
178 -- returns the largest prefix tree of t (possibly empty)
179 -- where all elements satisfy p.
180 -- For example, given the following tree
181
182 tree1 :: Tree Int
183 tree1 = Branch 1 (Branch 2 Leaf Leaf) (Branch 3 Leaf Leaf)
184
185 takeWhileTree :: (a -> Bool) -> Tree a -> Tree a
186 takeWhileTree f = foldTree Leaf (\x t1 t2 -> if f x then Branch x t1 t2
187                                         else Leaf)
188
189 ttakeWhileTree :: Test
190 ttakeWhileTree = "takeWhileTree" ~: TestList
191   [ takeWhileTree (<0) Leaf ~?= (Leaf :: Tree Int)
192   , takeWhileTree (< 3) tree1 ~?= (Branch 1 (Branch 2 Leaf Leaf) Leaf
193                                     :: Tree Int)
194   , takeWhileTree (< 9) tree1 ~?= tree1
195   , takeWhileTree (< 0) tree1 ~?= (Leaf :: Tree Int)
196   ]
197
198
199 -- allTree pred tree returns False if any element of tree
200 -- fails to satisfy pred and True otherwise.
201 -- for example:
202 --     allTree odd tree1 returns False
203
204 countNodes :: Num(b) => a -> b -> b -> b

```

```

205 countNodes _ counter _ = counter + 1
206
207 allTree :: (a -> Bool) -> Tree a -> Bool
208 allTree f = foldTree True (\x lm rm -> (f x) && lm && rm )
209
210 tallTree :: Test
211 tallTree = "allTree" ~: TestList
212     [ allTree (>0) Leaf ~?= True
213     , allTree (>2) (Branch 3 Leaf Leaf) ~?= True
214     , allTree (null) (Branch [] Leaf (Branch [] Leaf Leaf)) ~?= True
215     , allTree (>5) (Branch 3 Leaf Leaf) ~?= False
216     ]
217
218
219 -- WARNING: This one is a bit tricky! (Hint: the value
220 -- *returned* by foldTree can itself be a function.)
221
222 -- map2Tree f xs ys returns the tree obtained by applying f to
223 -- to each pair of corresponding elements of xs and ys. If
224 -- one branch is longer than the other, then the extra elements
225 -- are ignored.
226 -- for example:
227 --     map2Tree (+) (Branch 1 Leaf (Branch 2 Leaf Leaf)) (Branch 3 Leaf Leaf)
228 --     should return (Branch 4 Leaf Leaf)
229
230 map2Tree :: (a -> b -> c) -> Tree a -> Tree b -> Tree c
231 map2Tree f t1 t2 = (foldTree (\_ -> Leaf)
232     (\x lm rm -> superTreeFunction f x lm rm) t1) t2
233
234 where
235     superTreeFunction _ _ _ _ Leaf = Leaf
236     superTreeFunction f x lm rm (Branch v lb rb)
237         = Branch (f x v) (lm lb) (rm rb)
238
239 tmap2Tree :: Test
240 tmap2Tree = "map2Tree" ~: TestList
241     [ map2Tree (+) (Branch 1 Leaf (Branch 2 Leaf Leaf))
242         (Branch 3 Leaf Leaf)
243         ~?= (Branch 4 Leaf Leaf :: Tree Int)
244     , map2Tree (+) (Branch 1 Leaf Leaf)
245         (Branch 3 Leaf (Branch 2 Leaf Leaf))
246         ~?= (Branch 4 Leaf Leaf :: Tree Int)
247     , map2Tree (\a b -> [a, b, a+b]) (Branch 1 Leaf Leaf)
248         (Branch 3 Leaf (Branch 2 Leaf Leaf))
249         ~?= (Branch [1,3,4] Leaf Leaf :: Tree [Int])
250     , map2Tree (\a b -> [a, b, a+b]) (Branch 1 (Branch 2 Leaf Leaf)
251         (Branch 2 Leaf Leaf)) (Branch 3 Leaf (Branch 2 Leaf Leaf))
252         ~?= (Branch [1,3,4] Leaf (Branch [2,2,4] Leaf Leaf)
253             :: Tree [Int])
254     , map2Tree (+) Leaf (Branch 3 Leaf Leaf)
255         ~?= Leaf
256     , map2Tree (+) (Branch 3 Leaf Leaf) Leaf

```

```

256         ~?= Leaf
257     ]
258
259 -- zipTree takes two trees and returns a tree of corresponding pairs. If
260 -- one input branch is smaller, excess elements of the longer branch are
261 -- discarded.
262 -- for example:
263 --     zipTree (Branch 1 (Branch 2 Leaf Leaf) Leaf) (Branch True Leaf Leaf)
264 --     returns
265 --         (Branch (1,True) Leaf Leaf)
266
267 -- To use foldTree, you'll need to think about this one in
268 -- the same way as part (d).
269
270 zipTree :: Tree a -> Tree b -> Tree (a,b)
271 zipTree = map2Tree (\x y -> (x, y))
272
273 tzipTree :: Test
274 tzipTree = "zipTree" ~: TestList
275     [ zipTree (Branch 3 Leaf (Branch 0 (Branch 1 Leaf Leaf) Leaf))
276         (Branch False (Branch True Leaf Leaf) (Branch True Leaf Leaf))
277         ~?= (Branch (3,False) Leaf (Branch (0,True) Leaf Leaf)
278             :: Tree(Int, Bool))
279     , zipTree Leaf Leaf
280         ~?= (Leaf :: Tree (Int, Bool))
281     , zipTree (Branch [] Leaf (Branch [] Leaf Leaf))
282         (Branch 1 (Branch 2 (Branch 3 Leaf Leaf) Leaf) Leaf)
283         ~?= (Branch ([],1) Leaf Leaf :: Tree ([Int],Int))
284     , zipTree (Branch Leaf Leaf Leaf) (Branch Leaf Leaf Leaf)
285         ~?= (Branch (Leaf,Leaf) Leaf Leaf :: Tree (Tree Int,Tree Int))
286     ]
287 -----
288 foldXML :: (String -> b)
289         -> (String -> [b] -> b)
290         -> SimpleXML
291         -> b
292 foldXML fLeaf _ (PCDATA val) = fLeaf val
293 foldXML fLeaf fBranch (Element val lst)
294     = fBranch val $ map (foldXML fLeaf fBranch) lst
295
296
297 linebreak :: SimpleXML
298 linebreak = Element "br" []
299
300 formatPlay :: SimpleXML -> SimpleXML
301 formatPlay xml = Element "html" (foldXML fLeaf fBranch xml)
302     where
303         fLeaf val = [PCDATA val]
304         fBranch "PLAY" lst = [Element "body" $ concat lst]
305         fBranch "TITLE" lst = [Element "h1" $ concat lst]
306         fBranch "PERSONAE" lst = (Element "h2" [PCDATA "Dramatis Personae"])
```

```

307                                     : (concat lst)
308     fBranch "PERSONA" [[persona]]
309                     = [persona, linebreak]
310     fBranch "ACT" ([Element _ [act]]:xs)
311                     = (Element ("h2") [act]):(concat xs)
312     fBranch "SCENE" ([Element _ [scene]]:xs)
313                     = (Element ("h3") [scene]):(concat xs)
314     fBranch "SPEECH" ([Element _ [speech]]:xs)
315                     = (Element "b" [speech]):linebreak:(concat xs)
316     fBranch "LINE" [[line]] = [line, linebreak]
317     fBranch val lst        = [Element val $ concat lst]
318
319 firstDiff :: Eq a => [a] -> [a] -> Maybe ([a],[a])
320 firstDiff [] [] = Nothing
321 firstDiff (c:cs) (d:ds)
322     | c==d = firstDiff cs ds
323     | otherwise = Just (c:cs, d:ds)
324 firstDiff cs ds = Just (cs,ds)
325
326 -- | Test the two files character by character, to determine whether
327 -- they match.
328 testResults :: String -> String -> IO ()
329 testResults file1 file2 = do
330     f1 <- readFile file1
331     f2 <- readFile file2
332     case firstDiff f1 f2 of
333         Nothing -> return ()
334         Just (cs,ds) -> assertFailure msg where
335             msg = "Results differ: '" ++ take 20 cs ++
336                 "' vs '" ++ take 20 ds
337
338 testXML :: Test
339 testXML = TestCase $ do
340     writeFile "dream.html" (xml2string (formatPlay play))
341     testResults "dream.html" "sample.html"

```