Homework

Haskell Formalities

We declare that this is the Hw1 module and import some libraries:

```
module Solution where
import qualified Hw1 as H
import SOE
import Play
import XMLTypes
```

Part 0: All About You

Tell us your name, email and student ID, by replacing the respective strings below

```
myName  = "Write Your Name  Here"
myEmail = "Write Your Email Here"
mySID   = "Write Your SID   Here"
```

Preliminaries

Before starting this assignment:

- Download and install the Haskell Platform.
- Download the SOE code bundle.
- Verify that it works by changing into the `SOE/src` directory and running `ghci Draw.lhs`, then typing `main0` at the prompt:

```
cd SOE/src
ghci Draw.lhs
*Draw> main0
```

You should see a window with some shapes in it.

NOTE: If you have trouble installing SOE, see this page

5. Download the required files for this assignment: hw1.tar.gz. Unpack the files and make sure that you can successfully run the main program (in `Main.hs`). We've provided a `Makefile`, which you can use if you like. You should see this output:

```
Main: Define me!
```

Part 1: Defining and Manipulating Shapes

You will write all of your code in the `hw1.lhs` file, in the spaces indicated. Do not alter the type annotations — your code must typecheck with these types to be accepted.

The following are the definitions of shapes:

– > data Shape = Rectangle Side Side – > | Ellipse Radius Radius – > | RtTriangle Side Side – > | Polygon [Vertex] – > deriving Show

– > type Radius = Float – > type Side = Float – > type Vertex = (Float, Float)

1. Below, define functions `rectangle` and `rtTriangle` as suggested at the end of Section 2.1 (Exercise 2.1). Each should return a Shape built with the Polygon constructor.

```
rectangle :: Float -> Float -> H.Shape
rectangle s1 s2 = H.Polygon [(0, 0), (s1, 0), (s1, s2), (0, s2)]

rtTriangle :: Float -> Float -> H.Shape
rtTriangle s1 s2 = H.Polygon [(0, 0), (s1, 0), (0, s2)]
```

2. Define a function

```
sides :: H.Shape -> Int
```

which returns the number of sides a given shape has. For the purposes of this exercise, an ellipse has 42 sides, and empty polygons, single points, and lines have zero sides.

```
sides (H.Rectangle _ _)     = 4
sides (H.Ellipse _ _)       = 42
sides (H.RtTriangle _ _)    = 3
```

```haskell
{- Assume non-overlapping sides.  Consider the bowtie shape defined by
     Polygon [(0, 0), (1, 1), (1, 0), (0, 1)]
   This has four vertices but six sides. -}
sides (H.Polygon vs)       = if length vs <= 2 then 0 else fromIntegral $ length vs
```

3. Define a function

```haskell
bigger :: H.Shape -> Float -> H.Shape
```

that takes a shape `s` and expansion factor `e` and returns a shape which is the same as (i.e., similar to in the geometric sense) `s` but whose area is `e` times the area of `s`.

```haskell
-- bigger s e returns a new shape where each side is scaled by e
sBigger :: H.Shape -> Float -> H.Shape
sBigger (H.Rectangle s1 s2) e  = H.Rectangle (e * s1) (e * s2)
sBigger (H.Ellipse r1 r2) e    = H.Ellipse (e * r1) (e * r2)
sBigger (H.RtTriangle s1 s2) e = H.RtTriangle (e * s1) (e * s2)
sBigger (H.Polygon vs) e       = H.Polygon (expandVertices vs)
  where expandVertices [] = []
        expandVertices ((x, y) : vs) = (e * x, e * y) : expandVertices vs

-- bigger s e returns a new shape where the area is scaled by e
bigger s e = sBigger s (sqrt e)

dist :: (Float, Float) -> (Float, Float) -> Float
dist (x1, y1) (x2, y2) = sqrt $ (x1 - x2)^2 + (y1 - y2)^2

epsilon = 0.0001 :: Float

perimeter :: H.Shape -> Float
perimeter (H.RtTriangle s1 s2) = s1 + s2 + sqrt (s1^2 + s2^2)
perimeter (H.Rectangle s1 s2)  = 2 * s1 + 2 * s2
perimeter (H.Polygon vs)       = foldl (+) 0 $ sides vs
  where sides vs = zipWith dist vs (tail vs ++ [head vs])
perimeter (H.Ellipse r1 r2) | r1 > r2   = ellipsePerim r1 r2
                            | otherwise = ellipsePerim r2 r1
  where ellipsePerim r1 r2 =
          let e       = sqrt (r1^2 - r2^2) / r1
              s       = scanl aux (0.25 * e^2) [2..]
              aux s i = nextEl e s i
              test x  = x > epsilon
              sSum    = foldl (+) 0 (takeWhile test s)
          in 2 * r1 * pi * (1 - sSum)
        nextEl e s i = s * (2 * i - 1) * (2 * 1 - 3) * (e^2) / (4 * i^2)
```

```haskell
area :: H.Shape -> Float
area (H.Rectangle s1 s2)             = s1 * s2
area (H.RtTriangle s1 s2)            = 0.5 * s1 * s2
area (H.Ellipse r1 r2)               = pi * r1 * r2
area (H.Polygon (v1 : v2 : v3 : vs)) = triArea v1 v2 v3 + area (H.Polygon (v1 : v3 : vs))
  where triArea v1 v2 v3 = let a = dist v1 v2
                               b = dist v2 v3
                               c = dist v3 v1
                               s = 0.5 * (a + b + c)
                           in sqrt (s * (s - a) * (s - b) * (s - c))
area (H.Polygon _) = 0
```

4. The Towers of Hanoi is a puzzle where you are given three pegs, on one of which are stacked n discs in increasing order of size. To solve the puzzle, you must move all the discs from the starting peg to another by moving only one disc at a time and never stacking a larger disc on top of a smaller one.

To move n discs from peg a to peg b using peg c as temporary storage:

1. Move $n - 1$ discs from peg a to peg c.
2. Move the remaining disc from peg a to peg b.
3. Move $n - 1$ discs from peg c to peg b.

Write a function

```haskell
hanoi :: Int -> String -> String -> String -> IO ()
```

that, given the number of discs n and peg names a, b, and c, where a is the starting peg, emits the series of moves required to solve the puzzle. For example, running `hanoi 2 "a" "b" "c"`

should emit the text

```
move disc from a to c
move disc from a to b
move disc from c to b
```

```haskell
hanoi 0 _ _ _        = return ()
hanoi 1 src dst _    = putStr ("move disc from " ++ src ++ " to " ++ dst ++ "\n")
hanoi n src dst tmp = do
  hanoi (n - 1) src tmp dst
  hanoi 1 src dst tmp
  hanoi (n - 1) tmp dst src
```

Part 2: Drawing Fractals

1. The Sierpinski Carpet is a recursive figure with a structure similar to the Sierpinski Triangle discussed in Chapter 3:

Write a function `sierpinskiCarpet` that displays this figure on the screen:

```
sierpinskiCarpet :: IO ()
sierpinskiCarpet = main0

fillSquare w x y l =
  drawInWindow w
    (withColor Blue (polygon [(x, y), (x + l, y), (x + l, y + l), (x, y + l)]))

smin = 2

sierpinski w x y l =
  if l <= smin
  then fillSquare w x y l
  else do
    sierpinski w x y nl
    sierpinski w (x + nl) y nl
    sierpinski w (x + 2 * nl) y nl
    sierpinski w x (y + nl) nl
    sierpinski w (x + 2 * nl) (y + nl) nl
    sierpinski w x (y + 2 * nl) nl
    sierpinski w (x + nl) (y + 2 * nl) nl
    sierpinski w (x + 2 * nl) (y + 2 * nl) nl
      where nl = l `div` 3

main0 =
  runGraphics (
    do w <- openWindow "Sierpinski Carpet" (300, 300)
       sierpinski w 10 10 290
       k <- getKey w
       closeWindow w
    )
```

Note that you either need to run your program in `SOE/src` or add this path to GHC's search path via `-i/path/to/SOE/src/`. Also, the organization of SOE has changed a bit, so that now you use `import SOE` instead of `import SOEGraphics`.

2. Write a function `myFractal` which draws a fractal pattern of your own design. Be creative! The only constraint is that it shows some pattern of recursive self-similarity.

```haskell
myFractal :: IO ()
myFractal = error "Define me!"
```

Part 3: Recursion Etc.

First, a warmup. Fill in the implementations for the following functions.

(Your `maxList` and `minList` functions may assume that the lists they are passed contain at least one element.)

Write a *non-recursive* function to compute the length of a list

```haskell
lengthNonRecursive :: [a] -> Int
lengthNonRecursive = foldr (\_ n -> n + 1) 0
```

`doubleEach [1,20,300,4000]` should return `[2,40,600,8000]`

```haskell
doubleEach :: [Int] -> [Int]
doubleEach []     = []
doubleEach (x:xs) = 2*x : doubleEach xs
```

Now write a *non-recursive* version of the above.

```haskell
doubleEachNonRecursive :: [Int] -> [Int]
doubleEachNonRecursive = map (* 2)
```

`pairAndOne [1,20,300]` should return `[(1,2), (20,21), (300,301)]`

```haskell
pairAndOne :: [Int] -> [(Int, Int)]
pairAndOne []     = []
pairAndOne (x:xs) = (x,x+1) : pairAndOne xs
```

Now write a *non-recursive* version of the above.

```haskell
pairAndOneNonRecursive :: [Int] -> [(Int, Int)]
pairAndOneNonRecursive = map (\a -> (a, a + 1))
```

`addEachPair [(1,2), (20,21), (300,301)]` should return `[3,41,601]`

```haskell
addEachPair :: [(Int, Int)] -> [Int]
addEachPair []     = []
addEachPair ((x,y):xs) = x+y : addEachPair xs
```

Now write a *non-recursive* version of the above.

```
addEachPairNonRecursive :: [(Int, Int)] -> [Int]
addEachPairNonRecursive = map (uncurry (+))
```

`minList` should return the *smallest* value in the list. You may assume the input list is *non-empty*.

```
minList :: [Int] -> Int
minList []       = 0
minList [x]      = x
minList (x:y:xs) = minList $ min x y : xs
```

Now write a *non-recursive* version of the above.

```
minListNonRecursive :: [Int] -> Int
minListNonRecursive = foldr min (head l) l
```

`maxList` should return the *largest* value in the list. You may assume the input list is *non-empty*.

```
maxList :: [Int] -> Int
maxList []       = 0
maxList [x]      = x
maxList (x:y:xs) = maxList $ max x y : xs
```

Now write a *non-recursive* version of the above.

```
maxListNonRecursive :: [Int] -> Int
maxListNonRecursive = foldr max (head l) l
```

Now, a few functions for this `Tree` type.

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

`fringe t` should return a list of all the values occurring as a `Leaf`. So: `fringe (Branch (Leaf 1) (Leaf 2))` should return `[1,2]`

```
fringe :: Tree a -> [a]
fringe = treeFold (\x -> [x]) (++)
```

`treeSize` should return the number of leaves in the tree. So: `treeSize (Branch (Leaf 1) (Leaf 2))` should return 2.

```
treeSize :: Tree a -> Int
treeSize = treeFold (const 1) (+)
```

`treeSize` should return the height of the tree. So: `height (Branch (Leaf 1) (Leaf 2))` should return 1.

```
treeHeight :: H.Tree a -> Int
treeHeight (H.Leaf _) = 0
treeHeight (H.Branch l r) = 1 + max (treeHeight l) (treeHeight r)

treeFold :: (a -> b) -> (b -> b -> b) -> H.Tree a -> b
treeFold f g (H.Leaf x)     = f x
treeFold f g (H.Branch l r) = treeFold f g l `g` treeFold f g r
```

Now, a tree where the values live at the nodes not the leaf.

```
data InternalTree a = ILeaf | IBranch a (InternalTree a) (InternalTree a)
                      deriving (Show, Eq)
```

`takeTree n t` should cut off the tree at depth `n`. So `takeTree 1 (IBranch 1 (IBranch 2 ILeaf ILeaf) (IBranch 3 ILeaf ILeaf)))` should return `IBranch 1 ILeaf ILeaf`.

```
takeTree :: Int -> InternalTree a -> InternalTree a
takeTree _ H.ILeaf
  = H.ILeaf
takeTree n (H.IBranch x l r)
  | n > 0     = H.IBranch x (takeTree (n - 1) l) (takeTree (n - 1) r)
  | otherwise = H.ILeaf
```

`takeTreeWhile p t` should cut of the tree at the nodes that don't satisfy `p`. So: `takeTreeWhile (< 3) (IBranch 1 (IBranch 2 ILeaf ILeaf) (IBranch 3 ILeaf ILeaf)))` should return `(IBranch 1 (IBranch 2 ILeaf ILeaf) ILeaf)`.

```
takeTreeWhile :: (a -> Bool) -> InternalTree a -> InternalTree a
takeTreeWhile p H.ILeaf
  = H.ILeaf
takeTreeWhile p (H.IBranch x l r)
  | p x       = H.IBranch x (takeTreeWhile p l) (takeTreeWhile p r)
  | otherwise = H.ILeaf
```

Write the function map in terms of foldr:

```haskell
myMap :: (a -> b) -> [a] -> [b]
myMap f xs = foldr (\x xs -> f x : xs) [] xs
```

Part 4: Transforming XML Documents

The rest of this assignment involves transforming XML documents. To keep things simple, we will not deal with the full generality of XML, or with issues of parsing. Instead, we will represent XML documents as instances of the following simplified type:

```haskell
data SimpleXML =
   PCDATA String
 | Element ElementName [SimpleXML]
 deriving Show

type ElementName = String
```

That is, a `SimpleXML` value is either a `PCDATA` ("parsed character data") node containing a string or else an `Element` node containing a tag and a list of sub-nodes.

The file `Play.hs` contains a sample XML value. To avoid getting into details of parsing actual XML concrete syntax, we'll work with just this one value for purposes of this assignment. The XML value in `Play.hs` has the following structure (in standard XML syntax):

```xml
<PLAY>
  <TITLE>TITLE OF THE PLAY</TITLE>
  <PERSONAE>
    <PERSONA> PERSON1 </PERSONA>
    <PERSONA> PERSON2 </PERSONA>
    ... -- MORE PERSONAE
    </PERSONAE>
  <ACT>
    <TITLE>TITLE OF FIRST ACT</TITLE>
    <SCENE>
      <TITLE>TITLE OF FIRST SCENE</TITLE>
      <SPEECH>
        <SPEAKER> PERSON1 </SPEAKER>
        <LINE>LINE1</LINE>
        <LINE>LINE2</LINE>
        ... -- MORE LINES
```

```
      </SPEECH>
      ... -- MORE SPEECHES
    </SCENE>
    ... -- MORE SCENES
  </ACT>
  ... -- MORE ACTS
</PLAY>
```

- `sample.html` contains a (very basic) HTML rendition of the same information as `Play.hs`. You may want to have a look at it in your favorite browser. The HTML in `sample.html` has the following structure (with whitespace added for readability):

```
<html>
  <body>
    <h1>TITLE OF THE PLAY</h1>
    <h2>Dramatis Personae</h2>
    PERSON1<br/>
    PERSON2<br/>
    ...
    <h2>TITLE OF THE FIRST ACT</h2>
    <h3>TITLE OF THE FIRST SCENE</h3>
    <b>PERSON1</b><br/>
    LINE1<br/>
    LINE2<br/>
    ...
    <b>PERSON2</b><br/>
    LINE1<br/>
    LINE2<br/>
    ...
    <h3>TITLE OF THE SECOND SCENE</h3>
    <b>PERSON3</b><br/>
    LINE1<br/>
    LINE2<br/>
    ...
  </body>
</html>
```

You will write a function `formatPlay` that converts an XML structure representing a play to another XML structure that, when printed, yields the HTML specified above (but with no whitespace except what's in the textual data in the original XML).

```
formatPlay :: SimpleXML -> SimpleXML
-- `head` is safe to use here because the flattening function `f` always
```

```haskell
-- returns a non-empty list
formatPlay = head . flattenWith f
  where
    f "PLAY" (Element "TITLE" t : xs) =
        [Element "html" [Element "body" $ formatTitle 1 t : xs]]
    f "PERSONAE" ps =
        Element "h2" [PCDATA "Dramatis Personae"] : ps
    f "PERSONA" [p]                          = [p, br]
    f "ACT"     (Element "TITLE" t : xs) = formatTitle 2 t : xs
    f "SCENE"   (Element "TITLE" t : xs) = formatTitle 3 t : xs
    -- this pattern match is a bit silly, but it ensures totality of the call
    -- to `head` above
    f "SPEECH"  (x:xs)                   = x:xs
    f "SPEAKER" s                        = [Element "b" s, br]
    f "LINE"    [l]                      = [l, br]
    -- f for TITLE is basically `id` so we can pattern-match above and assign
    -- the proper heading
    f e xs = [Element e xs]

foldXML :: (ElementName -> [b] -> b) -> (String -> b) -> SimpleXML -> b
foldXML _ b (PCDATA s)     = b s
foldXML f b (Element e xs) = f e $ map (foldXML f b) xs

showXML :: SimpleXML -> String
showXML = foldXML f id
  where
    f e ss = concat [ "<", e, ">", concat ss, "<", e, "/>" ]

flattenXML :: SimpleXML -> [SimpleXML]
flattenXML = flattenWith (\e xs -> Element e [] : xs)

flattenWith :: (ElementName -> [SimpleXML] -> [SimpleXML])
            -> SimpleXML
            -> [SimpleXML]
flattenWith f = foldXML (\e xs -> f e $ concat xs) (\s -> [PCDATA s])

br :: SimpleXML
br = PCDATA "<br/>"

formatTitle :: Int -> [SimpleXML] -> SimpleXML
formatTitle n t = Element ('h' : show n) t
```

The main action that we've provided below will use your function to generate a file `dream.html` from the sample play. The contents of this file after your program runs must be character-for-character identical to `sample.html`.

```haskell
mainXML = do writeFile "dream.html" $ xml2string $ formatPlay play
             testResults "dream.html" "sample.html"

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)

testResults :: String -> String -> IO ()
testResults file1 file2 = do
  f1 <- readFile file1
  f2 <- readFile file2
  case firstDiff f1 f2 of
    Nothing -> do
      putStr "Success!\n"
    Just (cs,ds) -> do
      putStr "Results differ: '"
      putStr (take 20 cs)
      putStr "' vs '"
      putStr (take 20 ds)
      putStr "'\n"
```

Important: The purpose of this assignment is not just to “get the job done” — i.e., to produce the right HTML. A more important goal is to think about what is a good way to do this job, and jobs like it. To this end, your solution should be organized into two parts:

1. a collection of generic functions for transforming XML structures that have nothing to do with plays, plus
2. a short piece of code (a single definition or a collection of short definitions) that uses the generic functions to do the particular job of transforming a play into HTML.

Obviously, there are many ways to do the first part. The main challenge of the assignment is to find a clean design that matches the needs of the second part.

You will be graded not only on correctness (producing the required output), but also on the elegance of your solution and the clarity and readability of your code and documentation. Style counts. It is strongly recommended that you rewrite this part of the assignment a couple of times: get something working, then step back and see if there is anything you can abstract out or generalize, rewrite it, then leave it alone for a few hours or overnight and rewrite it again. Try to use some of the higher-order programming techniques we've been discussing in class.

Submission Instructions

- If working with a partner, you should both submit your assignments individually.
- Make sure your `hw1.lhs` is accepted by GHC without errors or warnings.
- Attach your `hw1.hs` file in an email to `cse230@goto.ucsd.edu` with the subject "HW1" (minus the quotes). *This address is unmonitored!*

Credits

This homework is essentially Homeworks 1 & 2 from UPenn's CIS 552.