Assignment 1

Rory Stephenson 300160212 stepherory

5 March 2014

Module Setup

```haskell
module Helper where
```

Core

`myContains` accepts an element and a list of the same type and returns `True` if the element is in the list and False otherwise. The type of the element and list must implement `Eq`.

```haskell
myContains :: Eq a => [a] -> a -> Bool
myContains x y = foldr (\ x0 y0 -> y0 || (x0 == y)) False x
```

`mySum` accepts a list of elements (which must be a subtype of `Num`) and returns the sum of all the elements in the list using foldr.

```haskell
mySum :: Num a => [a] -> a
mySum x = foldr (+) 0 x
```

`mySort` accepts a list of elements which are a subtype of `Ord` and returns that list in ascending order. It uses recursive mergesort to achieve the sorting.

```haskell
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort [a] = [a]
mySort xs = merge (mySort xs1) (mySort xs2)
            where
                    (xs1, xs2) = splitAt (quot (length xs) 2) xs

merge :: Ord a => [a] -> [a] -> [a]
merge [] y = y
merge x [] = x
merge (x:xrest) (y:yrest) | x <= y = x: merge xrest (y:yrest)
merge (x:xrest) (y:yrest) | otherwise = y: merge (x:xrest) yrest
```

`myFilter` accepts a function and a list and returns a new list containing all of the elements which evaluate to `True` when the provided function is applied to them (order is maintained). The provided function must accept an element of the provided list and return a `Bool`. `evaluateElem` is a helper function which takes a function, an element and a list and appends the element to the list if the function returns true for the given element.

```haskell
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter f list = foldr (evaluateElem f) [] list

evaluateElem :: (a -> Bool) -> a -> [a] -> [a]
evaluateElem f e list | f e == True = e:list
                      | otherwise = list
```

`myLast` returns the last element of a list. If the provided list is empty it will throw an "Empty List" exception.

```haskell
myLast :: [a] -> a
myLast [] = error "Empty List"
myLast (x:xs) = foldl (\ _ b -> b) x xs
```

`myUnzip` accepts a list of pairs and returns a pair of lists such that the first element of every pair is in the first list and the second element of every pair is in the second list. Ordering is maintained.

```haskell
myUnzip :: [(a, b)] -> ([a],[b])
myUnzip x = foldr (\ (a,b) (c,d) -> (a:c,b:d)) ([],[]) x
```

Completion

`powerSet` accepts a list and returns a list which contains the powerset of the provided list (every possible combination of the provided elements plus empty list). It (intentionally) does not remove duplicate elements and therefore does not return an actual set.

```haskell
powerSet :: [a] -> [[a]]
powerSet [] = [[]]
powerSet (x:xs) = powerSet xs ++ map (x:) (powerSet xs)
```

`scrabblify` accepts a string and returns an Int which is the score that the provided string would earn if played in scrabble (without bonus tiles). `scrabLetter` is a helper function which maps a `Char` to its scrabble value

```
scrabblify :: String -> Int
scrabblify x = foldr ((+) . scrabLetter) 0 x

scrabLetter :: Char -> Int
scrabLetter x | myContains "aAeEiIlLnNoOrRsStTuU" x = 1
              | myContains "dDgG" x = 2
              | myContains "bBcCmMpP" x = 3
              | myContains "fFhHvVwWyY" x = 4
              | myContains "kK" x = 5
              | myContains "jJxX" x = 8
              | myContains "qQzZ" x = 10
              | otherwise = 0
```

`sortBy` sorts a list with an ordering determined by the provided function. The provided function must accept two elements of the list given and return an `Ordering`. `mergeComp` is a helper function which merges two two lists according to the `ordering` specified by the provided function.

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
sortBy _ [] = []
sortBy _ [a] = [a]
sortBy f xs = mergeComp f (sortBy f xs1) (sortBy f xs2)
            where
                    (xs1, xs2) = splitAt (quot (length xs) 2) xs

mergeComp :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
mergeComp _ [] y = y
mergeComp _ x [] = x
mergeComp f (x:xrest) (y:yrest) | f x y == LT =
  x: mergeComp f xrest (y:yrest)
mergeComp f (x:xrest) (y:yrest) | otherwise =
  y: mergeComp f (x:xrest) yrest
```

`intersperse` takes an element and a list of the same type and returns a list where the provided element is inserted between each of the original elements. If the list is empty or only has one element this means the returned list will be the same as the given list.

```haskell
intersperse :: a -> [a] -> [a]
intersperse _ [] = []
intersperse x (y:ys) = y: foldr (\ a b -> x:a:b) [] ys
```