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. It achieves this via foldr which allows myContains to terminate even on an infinite list (the optimiser will realise that once a true value is found it can end evaluation). `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. It will never terminate on an infinite list as every element in an infinite list must be evaluated.

```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. `merge` is a helper function which merges two ordered lists.

```haskell
mySort :: Ord a => [a] -> [a]
mySort [] = []
mySort [a] = [a]
mySort xs = merge (mySort xs1) (mySort xs2)
            where
```

```haskell
                    (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. foldr was used to maintain ordering and to allow handling of infinite lists. `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. It works by folding from the left of a list and always carrying the last element evaluated.

```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 (the reason for choosing foldr) and infinite lists can be handled.

```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 a Num 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

```haskell
scrabblify :: Num a => String -> a
scrabblify x = foldr ((+) . scrabLetter) 0 x

scrabLetter :: Num a => Char -> a
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.

```haskell
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. It works by placing the given element in front of every element aside from the first element. foldr is used which maintains ordering and allows the function to terminate even on empty lists.

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