Folding

In this section we will look at the important idea of a folding operation, which provides a systematic way to process all elements in a list, or any other recursive structure. Folding effectively captures in a nicely generic way the idea of accumulating the values of a list/array.

Folding Lists

We start with a special case, called foldr1, which only works for lists that have at least one element. As an example, folding the addition operator over a list of numbers will result in adding those numbers:

```
foldr1 (+) [1..5] = 1 + (2 + (3 + (4 + 5)))
```

While the parentheses are not needed in this case, they indicate the way in which the function is applied.

As another example, we can implement the minimum function which finds the smallest element in an array simply as:

```
minimum lst = foldr1 (min) lst
```

where min is the function that given two numbers returns their maximum.

Practice

1. Write a foldr1 call which will determine if *all* the elements in a list of booleans are True, as well as one that will determine if *any* of them are True.
2. Write a foldr1 call that will concatenate together a list of strings.
3. Determine the type of foldr1 and then provide an implementation for it.

foldr

A more general pattern is implemented via the function foldr (without the 1). This is the overall pattern we want to employ, which replicates the idea of accumulation:

- We want to process the elements of a list of type [a] and return a value of a certain type b.
- We have an initial value to get as the result for the case of the empty list.
- For a non-empty list:
  - We get a value of type b from recursively working on the tail of the list.

- We have a way to combine that value with the head of the list to produce a new value. This would be done via a function of type: `a -> b -> b`.

There are many examples of this pattern: Computing the sum of numbers, the product of numbers, reversing a list, etc.

All these functions have the following "generic" implementation:

```
f []     = v
f (x:xs) = x # f xs     -- "#" is the function a -> b -> b
```

This is exactly what the function `foldr` does for us. Here is its type and definition:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

It takes in order:

- A function to be used for combining an `a` value with a `b` value, to produce a new *updated* `b` value.
- An initial `b` value.
- A list ofa` values to process.

And here is the implementation:

```
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Visually you should think of `foldr (#) v` as replacing the list "colon" operator with `#`, and the empty list with `v`, like so:

```
1 : (2 : (3 : []))      -- A list
1 # (2 # (3 # v))       -- The "foldr (#) v" of that list

As an example, `foldr (+) 0` is the same as `sum`:
```haskell
sum []     = 0
sum (x:xs) = (+) x (sum xs)    --- usually written as "x + sum xs"
-- visually:
1 + (2 + (3 + 0))
```

Let us think of how we can write the function `map` using `foldr`. It would look in general something like this:

```
map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\x ys -> ...) [] xs
```

where the function in the parentheses must be of type `a -> [b] -> [b]` (the "result type" that `foldr` calls `b` is in our case `[b]`).

So, we provide the empty list as an initial value: After all that should be the result if the `xs` is an empty list. Then we tell `foldr` that we will iterate over the list of the `xs`. Finally we need to tell it how to combine the current `a` value (`x`), and the list that is the result of processing the rest of the values, (`ys`), into the new list:
```

```haskell
map f xs = foldr (\x ys -> f x : ys) [] xs
--- We can also write this as:
map f = foldr (\x ys -> f x : ys) []
-- We can also write it as:
map f = foldr (\x -> (f x :)) []
```

Practice: Implement length and filter via foldr.

foldl

foldl is the sibling of foldr. It performs a similar process but does so in the opposite direction, from left to right. Symbolically we could say something like:

```haskell
foldl (#) y [x1, x2, x3] = (((y # x1) # x2) # x3)
```

Its type and standard implementation follow:

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ v []       = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Practice: Understand the above definition and make sure it typechecks.

Practice: Implement reverse using foldl:

```haskell
reverse = foldl (\ys y -> ...) []
```

Challenge: For those particularly motivated, there is a remarkable way to implement foldl via actually using foldr. The essential idea is to foldr appropriate functions, each new function building on the previous one. When these functions get called on the initial value, they end up performing the folds in the left-to-right order. If you are interested in learning more about this, here are two relevant links: Foldl as foldr alternative[1], A tutorial on the universality and expressiveness of fold[2]. But for now here is the implementation (Just understanding how the types work is an exercise in its own right, note how foldr appears to be applied to 4 arguments!):

```haskell
foldl f yinit xs = foldr construct id xs yinit
    where construct x g y = g (f y x)
          id y = y
```

[1] https://wiki.haskell.org/Foldl_as_foldr_alternative
[2] http://www.cs.nott.ac.uk/~pszgmh/fold.pdf