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.

Reading

- Sections 7.3, 7.4
- Practice exercises (7.9): 3, 4, 6

Folding Lists

Folding is meant to capture a quite generic pattern when traversing lists. This pattern could go as follows:

- 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: "'haskell 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:

```
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:

```
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:

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

Its type and standard implementation follow:

```
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:

```
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!):

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

Folding Trees

Recall how we defined trees in the past:

```
data Tree a = E | N (Tree a) a (Tree a)
```

It is natural for us to want to traverse the trees. The most universal way to do so is to define folding functions analogous to foldr or foldl. We will need three such functions, as trees can be traversed in three ways:

Inorder With *inorder traversal*, the nodes on the left child are visited first, then the root, then the nodes on the right child (left-root-right).

Preorder With *preorder traversal*, the root is visited first, then the nodes on the left child, then the ones on the right child (root-left-right).

Postorder with *postorder traversal*, the nodes on the left child are visited first, then the ones on the right child, and finally the root (left-right-root).

Let's take a look at how we can implement each of these:

```
foldin :: (a -> b -> b) -> Tree a -> b -> b
foldin _ E v                = v
foldin f (N left x right) v = v3
    where v1 = foldin f left v
          v2 = f x v1
          v3 = foldin f right v2
```

We could actually also write these in a "point-free" way, avoiding direct references to v:

```
foldin _ E              = id          -- The identity function
foldin f (N left x right) = foldin f right . f x . foldin f left
```

Practice: Implement the other two traversals, foldpre and foldpost.

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