Folding: Motivation

We wrote a function to add up a list:

\[ \text{sumList \ } [] = 0 \]
\[ \text{sumList \ } (x:xs) = x + \text{sumList \ } xs \]

In the assignment, we also wrote a function to multiply up a list:

\[ \text{prodList \ } [] = 1 \]
\[ \text{prodList \ } (x:xs) = x \times \text{prodList \ } xs \]

There is a lot of similarity here; only the binary operator and the initial value are different.

We can generalize this pattern and reduce boring coding.
Folding

The library function `foldr` captures the pattern in `sumList` and `prodList`. Here is what it looks like. We need to give it as parameters:

- the initial value `init` for the empty list case, e.g., `0`
- the binary function `f` to be used, e.g., addition

```
foldr f init [] = init
foldr f init (x:xs) = x 'f' foldr f init xs
  -- same as f x (foldr f init xs)
foldr :: (a->b->b) -> b -> [a] -> b
```

Examples:

```
sumList xs = foldr (+) 0 xs
prodList xs = foldr (*) 1 xs
```
Folding: Left and Right

The \( r \) in \texttt{foldr} means it computes from the right hand side:

\[
\text{foldr} \ (+) \ 0 \ [1,2,3] = 1+ (2+ (3+0))
\]

Similarly, there is a \texttt{foldl} that computes from the left hand side:

\[
\text{foldl} \ (+) \ 0 \ [1,2,3] = ((0+1)+2)+3
\]

It looks like this:

\[
\text{foldl} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow \ [a] \rightarrow b \\
\text{foldl} \ f \ \text{init} \ [] = \text{init} \\
\text{foldl} \ f \ \text{init} \ (x:xs) = \text{foldl} \ f \ (\text{init} \ ‘f’ \ x) \ \text{xs}
\]
Folding: When to Use Which

Which of foldl and foldr should we use? It depends on the situation.

- We probably want to use foldl to add up integers. It is tail-recursive.
  
- But we probably want to use foldr to join a list of strings.

  \[
  \text{foldl (++) "" ["abc","abc","abc"]}
  \]
  
  takes quadratic time, while
  
  \[
  \text{foldr (++) "" ["abc","abc","abc"]}
  \]
  
  takes linear time. This is because (++) is linear in its first argument.
Currying: Introduction

Consider the following function:

\[
\text{myadd} :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Int} \\
\text{myadd} \ x \ y \ z = x+y+z
\]

The type could be read as “a function that takes three numbers and returns a number”. But it could also be read as:

- \(\text{myadd} :: \text{Int} \to \text{Int} \to (\text{Int} \to \text{Int})\)
  
  takes two numbers and returns a function \(\text{Int} \to \text{Int}\).

- \(\text{myadd} :: \text{Int} \to (\text{Int} \to \text{Int} \to \text{Int})\)
  
  takes one number and returns a function \(\text{Int} \to \text{Int} \to \text{Int}\).
Currying

So you can give one parameter at a time and get intermediate functions:

• myadd 1 :: Int -> Int -> Int
  a function that takes two numbers and add them to 1

• myadd 1 2 :: Int -> Int
  a function that takes a number and add it to 1 + 2

• myadd 1 2 3 :: Int
  finally the number 6

This ability is called *currying*.
Currying: Examples

Using currying, we can shorten the definition of `sumList` a bit. Recall:

```haskell
sumList :: [Int] -> Int
sumList xs = foldr (+) 0 xs
```

Look at the right hand side. If we omit the third parameter, we will have:

```haskell
foldr (+) 0 :: [Int] -> Int
```

This is precisely what we want for `sumList`, matching both the type specification and the content! So we will write:

```haskell
sumList = foldr (+) 0
prodList = foldr (*) 1
```
Composition

Recall that we had a function that sums up the areas of a list of shapes. It can now be written as:

\[
\text{areaList } xs = \text{sumList } (\text{map area } xs)
\]

This is saying: pass \(xs\) to a function \(f\) (\(\text{map area}\)), then take the result and pass it to another function \(g\) (\(\text{sumList}\)). This is function composition. There is an operator for this:

\[
(\cdot) :: (b\to c) \to (a\to b) \to (a\to c)
\]

\[
(g \cdot f) x = g (f x)
\]

So \(g \cdot f\) is a function that, when you give it a parameter \(x\), it will compute \(f(x)\), and then use it to compute \(g(f(x))\).
Composition: Example

Look at `areaList` again:

```haskell
areaList xs = sumList (map area xs)
```

Using composition, we can rewrite it as:

```haskell
areaList xs = (sumList . map area) xs
```

But then we can apply currying:

```haskell
areaList = sumList . map area
```
Anonymous Functions: Motivation

There are times when we want to write a function without giving it a name. E.g.,

\[
\text{square } n = n * n
\]

is silly if all we want is just:

\[
\text{map square } [1,2,3]
\]

Even this:

\[
\text{let square } n = n * n \text{ in map square } [1,2,3]
\]

is too tedious. We would like to write functions without giving them names.
Anonymous Functions

Here is how. A function that squares its parameter:

\[ n \rightarrow n \times n \]

So to square a list of numbers,

\[
\text{map (} n \rightarrow n \times n \text{)} \ [1,2,3]
\]

More parameters can be accommodated too, e.g.,

\[ x \ y \ z \rightarrow x+ y + z \]

This is a shorthand for:

\[ x \rightarrow \ y \rightarrow \ z \rightarrow x + y + z \]
Sections

Binary operators can be turned into unary functions by giving them a constant argument and using the following syntax:

(1+) means $\lambda x \rightarrow 1+x$
(+1) means $\lambda x \rightarrow x+1$

E.g., a function that increments every number in a list:

map (+1)

A function that tests if all numbers in a list are negative:

foldl (&&) True . map (<0)

The library has a function to do the foldl (&&) True part:

and . map (<0)