Exercise One

- Working with Expressions
- Implement `eval` and `simp`
- Test driven:
  - Code compiles, all tests pass: `Full Marks`
  - Code compiles, some tests pass: `Marks reduce pro-rata.`
  - Code compiles, no tests pass: `Zero Marks`
  - Code does not compile: `Zero Marks`

Turning common "shapes" into functions

Remember these?

```
sum [] = 0
sum (n:ns) = n + sum ns

length [] = 0
length (_:xs) = 1 + length xs

prod [] = 1
prod (n:ns) = n * prod ns
```

They have a common pattern,
which is typically referred to as "`folding`".
Can we abstract this?
Can we produce something (`<abs-fold>`) that captures folding?

Common aspects (I)

They all have the empty list as a base case

```
sum [] = 0
sum (n:ns) = n + sum ns

length [] = 0
length (_:xs) = 1 + length xs

prod [] = 1
prod (n:ns) = n * prod ns


<abs-fold> [] = ....
```

Common aspects (II)

They all have a non-empty list as the recursive case

```
sum [] = 0
sum (n:ns) = n + sum ns

length [] = 0
length (_:xs) = 1 + length xs

prod [] = 1
prod (n:ns) = n * prod ns


<abs-fold> [] = ...
<abs-fold> (a:as) = ... <abs-fold> as
```

Common aspects (III)

The base case returns a fixed "unit" value, which we will call u.

```
sum [] = 0
sum (n:ns) = n + sum ns

length [] = 0
length (_:xs) = 1 + length xs

prod [] = 1
prod (n:ns) = n * prod ns


<abs-fold> [] = u
<abs-fold> (a:as) = ... <abs-fold> as
```

Common aspects (IV)

The recursive case combines the head of the list with the result of the recursive call, using a binary operator we shall call op

```
sum [] = 0
sum (n:ns) = n + sum ns

length [] = 0
length (x:xs) = x `incr` length xs
  where x `incr` y = 1 + y

prod [] = 1
prod (n:ns) = n * prod ns


<abs-fold> [] = u
<abs-fold> (a:as) = a `op` <abs-fold> as
```

Common aspects (V)

So we have the following abstract form

```
<abs-fold> [] = u
<abs-fold> (a:as) = a `op` <abs-fold> as
```

But how do we instantiate <abs-fold> ?

Our concrete fold needs to be a function that is supplied with u and op as arguments, and then builds a function on lists as above.

So <abs-fold> becomes fold u op

```
fold u op [] = u
fold u op (a:as) = a `op` fold u op as
```

This is a HOF that captures a basic recursive pattern on lists.

Common aspects (VI)

We have <abs-fold>fold u op
So how do we use fold to save boilerplate code?

```
-- sum [] = 0,                      u = 0
-- sum (n:ns) = n + sum ns,         op = (+)
sum = fold 0 (+)

-- length [] = 0,                   u = 0
-- length (_:xs) = 1 + length xs,   op = incr
length = fold 0 incr where _ `incr` y = 1 + y

-- prod [] = 1,                     u = 1
-- prod (n:ns) = n * prod ns,       op = (*)
prod = fold 1 (*)
```

The type of `fold`

```
fold u op [] = u
fold u op (a:as) = a `op` fold u op as

--  a :: t,  as :: [t]
--  u :: r   -- result type may differ, e.g. length
-- op :: t -> r ->  r   -- 1st from list, 2nd a "result"

fold :: r -> (t -> r -> r) -> [t] -> r
```

Fold in Haskell

- Haskell has a number of variants of `fold`
- "Fold-Right" (`foldr`) is like our `fold` in that the uses of `op` are nested on the *right*.

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

  ```
  foldr (+) 0 [10,11,12] = 10 + (11 + (12 + 0))
  ```

  Note: The order of `u` and `op` are also different!
- "Fold-Left" (`foldl`) is different in that the uses of `op` are nested on the *left*.

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

  ```
  foldl (+) 0 [10,11,12] = ((0 + 10) + 11) + 12
  ```

  We shall see reasons for the distinction later.
- There are also variants that don't require the unit `u` to be specified, but which are only defined for non-empty lists.