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.

The Type of Equality

- We test for equality, using infix operator `==`

```
GHCi> 1 == 2
False
GHCi [1,2,3] == (reverse [3,2,1])
True
```

- What is the type of `==` ?
  - It compares things of the same type to give a boolean result:
    `(==) ::  a -> a -> Bool`
    (so it's polymorphic, then?)
  - What does Haskell think ?

```
GHCi> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

    It says `==` is defined for types that are instances of the the `Eq` Class.

Constraints

- The declaration `Eq a => a -> a -> Bool` contains what is known as a *type constraint* (here, `Eq a =>`)
- The constraint says that the type `a` must belong to the *class of types* `Eq`
- A number of predefined type classes:
  - `Eq` : Defines `==`.
    (Hint: try `:i Eq` in GHCi).
  - `Num` : Defines `+` and `-`, among others
  - `Ord` : Defines comparisons, `<=`
  - `Show` : Can convert to `String`
    (think of implementing `.toString()` in Java).
  - many more...
- The mention of the class name is a promise that some set of functions will work on the values of that class.
- A type class is an *interface* that the compiler will check for you, allowing you to say things like "this function accepts anything that `(+)` works on"

Ad-Hoc Polymorphism

- Equality is "polymorphic"
  `(==) ::  a -> a -> Bool`
- However it is *ad-hoc*:
  - There has to be a specific (different) implementation of it for each type

```
primIntEq :: Int -> Int -> Bool
primFloatEq :: Float -> Float -> Bool
...
```

- Contrast with the (parametric) polymorphism of `length`:
  - The same program code works for all lists, regardless of the underlying element type.

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

Ad-hoc polymorphism is ubiquitous

- Ad-hoc polymorphism is very common in programming languages:

operators	types
$= \neq < \leq > \geq$	$T \times T \rightarrow \mathbb{B}$, for (almost) all types T
$+ - * /$	$N \times N \rightarrow N$, for numeric types N

- The use of a single symbol ($+$, say) to denote lots of (different but related) operators, is also often called "overloading"
- In many programming languages this overloading is built-in
- In Haskell, it is a language feature called "type classes", so we can "roll our own".

Defining (Type-)Classes in Haskell (Overloading)

- In order to define our own name/operator overloading, we:
  - need to specify the name/operator involved (e.g. `==`);
  - need to describe its pattern of use (e.g. `a -> a -> Bool`);
  - need an overarching "class" name for the concept (e.g. `Eq`).
- In order to use our operator with a given type (e.g. `Bool`, we:
  - need to give the implementation of `==` for that type (`Bool -> Bool -> Bool`).
  - In other words, we define an instance of the type for the class.

Defining The Equality Class

- We define the class `Eq` as follows:

```
class Eq a where
  (==) :: a -> a -> Bool
```

- The first line introduces `Eq` as a class characterising a type (here called `a`).
- The second line declares that a type belonging to this class must have an implementation of `==` of the type shown.
- `class` and `where` are Haskell keywords

Giving an instance of the Equality Class

- We define an instance of `Eq` for booleans as follows

```
instance Eq Bool where
  True == True    =  True
  False == False  =  True
  _ == _          =  False
```

  (here `_` is a wildcard pattern matching anything).
- Now all we do is define instances for the other types for which equality is desired.
  - (In fact, in many cases, for equality, we simply refer to a primitive builtin function to do the comparison)
  - Most of this is already done for us as part of the Haskell *Prelude*.
- `instance` is a Haskell keyword

The "real" equality class

- In fact, Eq has a slightly more complicated definition:

```
class  Eq a  where
  (==), (/=) :: a -> a -> Bool
     -- Minimal complete definition: (==) or (/=)
  x /= y     =  not (x == y)
  x == y     =  not (x /= y)
```

- First, an instance must also provide /= (not-equal).
- Second, we give (circular) definitions of == and /= in terms of each other
  - The idea is that an instance need only define one of these
  - The other is then automatically derived.
  - However we may want to explicitly define both (for efficiency).

How Haskell handles a class name/operator (I)

- Consider the following (well-typed) expression:
  `x == 3 && y == False`
  (So `x` has type `Int`, and `y` is of type `Bool`).
- The compiler sees the symbol `==`, notes it belongs to the `Eq` class, and then . . .
  - seeing `x::Int` deduces (via type inference) that the first `==` has type `Int -> Int -> Bool`
    This is acceptable as it knows of such an instance of `==`
  - Generates code using that instance for that use of equality
  - Does a similar analysis of the second `==` symbol, and generates boolean-equality code there.

How Haskell handles a class name/operator (II)

- Now consider the following (well-typed) expression:
  `x == 3 && y == False || z == MyCons`
  (here `z` has a user defined `data` type `MyType`, with `MyCons` as a constructor).
  Assume we have *not* declared an instance of `Eq` for this type
- The compiler, seeing the 3rd `==`, looks for an instance for `MyType` of `Eq`, and fails to find one
- It generates a error message of the form

```
No instance for (Eq MyType)
  arising from a use of ‘==’ at ...
Possible fix: add an instance declaration for (Eq MyType)
```

- Note the helpful suggestion !