Type classes in more detail

Some Haskell options we will use in this file

We'll be using the following Haskell option to have more flexibility in the ways we are allowed to define things:

```
{-# Language FlexibleInstances #-}
```

We will also use the option below

```
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
```

to ask `ghc` to give us warning when we don't have enough patterns in a definition by pattern matching.

Motivation

Consider the following interaction at the `ghci` command prompt:

```
Prelude> "abcd" == "ab" ++ "cd"
True
Prelude> "abcd" == reverse ("ab" ++ "cd")
False
Prelude>
```

Good. We successfully compared some strings for equality.

But now consider the Haskell code below

```
f1, f2, f3 :: Bool -> Bool
f1 x = if x then False else True
f2 x = not x
f3 x = x
```

and try to see whether these two functions are equal at the `ghci` prompt:

```
Prelude> f1 == f2

<interactive>:14:1: error:
```

Why do we get an error? Because Haskell doesn't know how to compare functions for equality. In this particular case, it is easy to *see* that that these two functions are equal. But, from Computability Theory, we know that, unfortunately, there is no algorithm that can decide whether or not two given functions are equal, in general

But Haskell gives us a more cryptic error message instead:

```
    • No instance for (Eq (Bool -> Bool)) arising from a use of ‘==’
        (maybe you haven't applied a function to enough arguments?)
     • In the expression: f1 == f2
       In an equation for ‘it’: it = f1 == f2
```

What does this mean? It means that there is no equality function `==` defined for the function type `Bool -> Bool`

For this particular type `Bool -> Bool`, it is easy to compare functions for equality, and we can implement it:

```
instance Eq (Bool -> Bool) where
  f == g = f True == g True && f False == g False
```

```
*Main> f1 == f2
True
*Main> f1 == f3
False
```

We can do better, in the sense of being more general:

```
instance Eq a => Eq (Bool -> a) where
  f == g = f True == g True && f False == g False
```

In English, this says the following:

- If we know how to compare elements of the type `a` for equality,
- then we can compare functions `f,g :: Bool -> a` for equality using the algorithm `f True == g True && f False == g False`.

Self-contained example

The following Haskell code is self-contained and deliberately doesn't use any Haskell library, not even the prelude, so that we can have a full and clear picture of

- what type classes are,
- what they are for, and
- how they are used.

Example 1

We now define our own equality class from scratch as follows.

We only include an equality function in our definition, written `===`:

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

We implement this equality for some types. Booleans first:

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

Now pairs. What is different here is that assuming we have equality on the types `a` and `b`, we define equality on the type `(a,b)` of pairs:

```


instance (MyEq a , MyEq b) => MyEq (a , b) where
  (x , y) === (u , v) = x === u && y === v
```

Similarly, if we have equality in the type `a`, then we can define equality in the type `[a]` of lists of elements of type `a`:

```
instance MyEq a => MyEq [a] where
  []     === []     = True
  (x:xs) === (y:ys) = x === y && xs === ys
  _      === _      = False

instance MyEq a => MyEq (Bool -> a) where
  f === g = f True === g True && f False === g False
```

Here are some examples of functions using the above:

```
allEqual :: MyEq a => [a] -> Bool
allEqual []       = True
allEqual (x:[])   = True
allEqual (x:y:zs) = x === y && allEqual (y:zs)

someDifferent :: MyEq a => [a] -> Bool
someDifferent []       = False
someDifferent (x:[])   = False
```

Convince yourself that `allEqual xs === not (someDifferent xs)` is `True` for all lists `xs :: [a]`. (Later in the module we will discuss induction on lists and be able to argue rigorously to show this.)

Example 2

We now illustrate default methods:

```
class YourEq a where
  (====) :: a -> a -> Bool      -- (1)
  (=//=) :: a -> a -> Bool      -- (2)

  a ==== b = not (a =//= b)     -- Default definition of (1) using (2)
  a =//= b = not (a ==== b)     -- Default definition of (2) using (1)
```

The last two are the *default* definitions:

- If we define (1), then we don't need to define (2).
- If we define (2), then we don't need to define (1).
- But we can define both if we want.

Some examples follow:

- Only define (1):

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

- We only define (2):

```
instance (YourEq a , YourEq b) => YourEq (a , b) where
  (x , y) =//= (u , v) = x =//= u || y =//= v
```

- We only define (1):

```
instance YourEq a => YourEq [a] where
  []     ==== []     = True
  (x:xs) ==== (y:ys) = x ==== y && xs ==== ys
  _      ==== _      = False
```

We define both (1) and (2):

```
instance YourEq a => YourEq (Bool -> a) where

  f ==== g = f True ==== g True && f False ==== g False

  f =//= g = f True =//= g True || f False =//= g False
```

Example - adding `Bool` to the type class `Num`

Consider the following example:

```
ghci> True + 0

<interactive>:1:6: error: [GHC-39999]
    • No instance for ‘Num Bool’ arising from a use of ‘+’
    • In the expression: True + 0
      In an equation for ‘it’: it = True + 0
```

This is saying that `True + 0` requires the type `Bool` to be in the type class `Num`, but it isn't. But we can, if we want, to add `Bool` to the type class `Num` as follows:

```
instance Num Bool where

  False + y  = y
  (*)    = (&&)
  negate = id
  abs    = id
  signum = id

  fromInteger 0 = False
  fromInteger _ = True
```

Here we are saying that addition is `exclusive or`, multiplication is `and`, and that the negative of a boolean is itself. Notice that we encode booleans as integers as in the programming language `C`. These operations form a so-called Boolean ring. We can check the Boolean ring property as follows:

```
bools = [False,True]

testPlusAssoc = and [ (x + y) + z == x + (y + z)   | x <- bools, y <- bools, z <-bools]
testMulAssoc  = and [ (x * y) * z == x * (y * z)   | x <- bools, y <- bools, z <-bools]
testPlusComm  = and [ x + y == y + x               | x <- bools, y <- bools]
testMulComm   = and [ x * y == y * x               | x <- bools, y <- bools]
testNeg       = and [ x + (- x) == 0               | x <- bools]
testBoolean   = and [ x * x == x                   | x <- bools]
test0         = and [ x + 0 == x                   | x <- bools]
test1         = and [ x * 1 == x                   | x <- bools]
testDistrL    = and [ a * (x + y) == a * x + a * y | a <- bools, x <- bools, y <- bools]
testDistrR    = and [ (x + y) * a == x * a + y * a | a <- bools, x <- bools, y <- bools]
```

The operations `abs` and `signum` are not part of the definition of a Boolean ring, and they are required to satisfy the following equation:

```
testSignum    = and [ abs x * signum x == x        | x <- bools]
```

We can test everything as follows:

```
testAll = and [testPlusAssoc, testMulAssoc, testPlusComm, testMulComm,
               testNeg, testBoolean, test0, test1 , testDistrL , testDistrR ,
               testSignum]
```

This evaluates to `True`, which shows that the above equations hold.

```
ghci> testAll
True
```

And now we don't get an error any more when we do `True + 0`. This expression evaluates to `True`:

```
ghci> True + 0
True
```