Some Haskell conveniences, tips and tricks

1 Minor pieces of "syntactic sugar"

(1) For "chained" function applications, e.g. `(f Rock) Z`, we can omit the parentheses and just write `f Rock Z`. Similarly for the type of a function like `f` here, such as `Shape -> (Numb -> Bool)`, we can just write `Shape -> Numb -> Bool`.

(2) The case expression for `Bool` has a friendly abbreviation. Instead of writing:

```
case b of {True -> x; False -> y}
```

you can just write:

```
if b then x else y
```

(3) When you are giving a name to a lambda abstraction, you can write the argument on the left hand side instead of writing an explicit lambda. For example, instead of writing:

```
isZero = \n -> case n of {Z -> True; S n' -> False}
```

you can just write:

```
isZero n = case n of {Z -> True; S n' -> False}
```

This works just the same with multiple lambdas too (because they're just one lambda inside another), so the following are all equivalent:

```
add = \n -> \m -> case m of {Z -> n; S m' -> S (add n m')}

add n = \m -> case m of {Z -> n; S m' -> S (add n m')}

(add n) m = case m of {Z -> n; S m' -> S (add n m')}

add n m = case m of {Z -> n; S m' -> S (add n m')}
```

(4) When the right hand side of a definition is a case expression, you can abbreviate by instead writing a separate "equation" for each case. This avoids giving a name to the thing you want to look at the insides of. For example, instead of writing:

```
isZero n = case n of {Z -> True; S n' -> False}

add n m = case m of {Z -> n; S m' -> S (add n m')}
```

you can just write:

```
isZero Z = True
isZero (S n') = False

add n Z = n
add n (S m') = S (add n m')
```

(5) If a “two-place” function has a name that begins with `(`, ends with `)`, and contains only non-alphanumeric symbols in between — for example, a name like `(&&)` or `(#$#)` — then you can use it without the parentheses as an infix operation. For example instead of

```
(#$#) a b
```

you can just write

```
a #$# b
```

2 Polymorphism and type classes

If we ask for the types of `map` and `filter`, here's what we find:

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> :t filter
filter :: (a -> Bool) -> [a] -> [a]
```

The `a` and `b` are type variables. (Anything that begins with a lowercase letter in a type is a type variable; type constants begin with uppercase letters, e.g. `Bool` and `Shape`.) These type variables are bound by an implicit universal quantifier. So you can read the type for `filter` as “For all types `a`, type `(a -> Bool) -> [a] -> [a]`”. Types that contain variables like this are called *polymorphic types*, and functions that have these types are called *polymorphic functions*.

Now suppose we wanted to write a polymorphic function which, given a list and a particular thing of the appropriate type, checks whether this thing is in the list. It seems like it should be possible to write a function like this which would have type `a -> [a] -> Bool`. But we would get stuck, because we wouldn't know how to “look at” the elements of the list:

```
isElement y [] = False
isElement y (x:xs) = if (...???...) then True else (isElement y xs)
```

One solution is to explicitly “outsource” this work of checking whether `x` and `y` match in the manner appropriate for the relevant type, by having a function that checks this passed as an additional argument. This way the function has a slightly different type, namely `(a -> a -> Bool) -> a -> [a] -> Bool`.

```
isElement equal y [] = False
isElement equal y (x:xs) = if (equal y x) then True else (isElement equal y xs)
```

Or, just to make things a bit prettier, we could use that handy trick for writing binary functions infix (see (5) above):

```
isElement (#=) y [] = False
isElement (#=) y (x:xs) = if (y #= x) then True else (isElement (#=) y xs)
```

This is, in a sense, the *only* solution — it's the way any polymorphic function for checking whether something is in a list has to work. But Haskell provides some machinery to allow us to do it a bit more conveniently. Comparing things for equality is such a common thing to do that it would be annoying to have to pass around the appropriate `(a -> a -> Bool)` functions everywhere they were needed. So instead Haskell allows us to specify an equality function just once for a particular type, and then that function is implicitly “carried around” with every value of that type. This “global” equality function is called `(==)`, so we can use it to write our `isElement` function like this:

```
isElement y [] = False
isElement y (x:xs) = if (y == x) then True else (isElement y xs)
```

But now we can't say that this function has type `(a -> a -> Bool)` *for all* types `a`, because it is only compatible with types that carry around with them a function called `(==)`. The types that carry around such a function are the types that belong to the *type class* called `Eq`.[1] So the type of this final `isElement` function is

```
Eq a => a -> [a] -> Bool
```

where you can read the `Eq a` part as a restrictor on the universal quantifier. This function is also predefined for us, with the name `elem`.

There are two different ways that we can put a type that we define (e.g. `Numb`) into the type class `Eq`:

- The "manual" option is to write the appropriate function like this (the name doesn't matter):

  ```
  foo Z Z = True
  foo Z (S n) = False
  foo (S m) Z = False
  foo (S m) (S n) = foo m n
  ```

  and then specify that this is the `(==)` function for this type like this:

  ```
  instance Eq Numb where (==) = foo
  ```

- The "automatic" option is to simply add `Eq` to the `deriving` list at the end of the type's declaration:

  ```
  data Numb = Z | S Numb deriving (Show,Eq)
  ```

3 `map` and extending binary operations to collections

3.1 Reminder of some mathematical notation

Suppose $S = \{3, 4, 5, 6\}$. Then using what's sometimes known as "set-builder" or "set comprehension" notation, we can write:

(6) a. $\{x + 10 \mid x \in S\} = \{3 + 10, 4 + 10, 5 + 10, 6 + 10\} = \{13, 14, 15, 16\}$
   b. $\{y^2 + 2y \mid y \in S\} = \{3^2 + 2 \times 3, 4^2 + 2 \times 4, 5^2 + 2 \times 5, 6^2 + 2 \times 6\} = \{15, 24, 35, 48\}$

The general idea here is that we write $\{\mathbf{E} \mid v \in S\}$, where $\mathbf{E}$ is some expression in which v is a free variable. We understand $\mathbf{E}$ as a recipe for getting some widget from an element of the set S. Then $\{\mathbf{E} \mid v \in S\}$ is the set of widgets we get by following this recipe for every element of S.[2]

Now supposing still that $S = \{3, 4, 5, 6\}$, a common piece of closely-related notation for summations works like this:

(7) a. $\sum_{x \in S}(x + 10) = (3 + 10) + (4 + 10) + (5 + 10) + (6 + 10)$
   b. $\sum_{y \in S}(y^2 + 2y) = (3^2 + 2 \times 3) + (4^2 + 2 \times 4) + (5^2 + 2 \times 5) + (6^2 + 2 \times 6)$

So this summation/sigma notation combines the idea of the set comprehension notation with the binary **addition** operation.

We can similarly combine the set comprehension idea with the binary operation of **multiplication**; this is usually written with a big pi, as follows:

[1] Perhaps confusingly, *type classes* correspond to the general idea of what are called *interfaces* in some other programming languages (e.g. Java) — but *not* classes in object-oriented languages, which are generally more analogous to types.

[2] So the variable x is free in the expression x^2, but in the larger expression $\{x^2 \mid x \in S\}$, it is bound. This is why $\{x^2 \mid x \in S\}$ means the same thing as $\{y^2 \mid y \in S\}$. Of course, the variable S is free in these larger expressions. So just as y^2 is a recipe for getting one number from another, $\{y^2 \mid y \in S\}$ is a recipe for getting one set from another.

(8) a. $\prod_{x \in S}(x+10) = (3+10) \times (4+10) \times (5+10) \times (6+10)$

b. $\prod_{y \in S}(y^2+2y) = (3^2+2\times 3) \times (4^2+2\times 4) \times (5^2+2\times 5) \times (6^2+2\times 6)$

So the sigma and pi notation "extend" certain binary operations on numbers, namely addition and multiplication respectively.[3]

We can do the same thing for binary operations on booleans (or truth values), in particular **disjunction** ("or") and **conjunction** ("and"). For the analogous extensions of these — which act a bit like existential and universal quantification, respectively — we often just write big versions of the familiar $\vee$ and $\wedge$ symbols:

(9) a. $\bigvee_{x \in S}(x \leq 5) = (3 \leq 5) \vee (4 \leq 5) \vee (5 \leq 5) \vee (6 \leq 5)$

b. $\bigvee_{y \in S}(y^2 \geq 10) = (3^2 \geq 10) \vee (4^2 \geq 10) \vee (5^5 \geq 10) \vee (6^2 \geq 10)$

(10) a. $\bigwedge_{x \in S}(x \leq 5) = (3 \leq 5) \wedge (4 \leq 5) \wedge (5 \leq 5) \wedge (6 \leq 5)$

b. $\bigwedge_{y \in S}(y^2 \geq 10) = (3^2 \geq 10) \wedge (4^2 \geq 10) \wedge (5^5 \geq 10) \wedge (6^2 \geq 10)$

Notice that in (9) and (10), the expressions $x \leq 5$ and $y^2 \geq 10$ specify recipes for getting a *boolean* from a number, whereas in (7) and (8) we had recipes for getting a *number* from a number — because disjunction and conjunction work on booleans, whereas addition and multiplication work on numbers.

3.2 Equivalents with Haskell lists

Mathematical notation	**Haskell**	**Related functions**
$\{x+10 \mid x \in S\}$	`map (\x -> x + 10) s`	
$\sum_{x \in S}(x+10)$	`sum (map (\x -> x + 10) s)`	`sum :: [Int] -> Int` `(+) :: Int -> Int -> Int`
$\prod_{x \in S}(x+10)$	`product (map (\x -> x + 10) s)`	`product :: [Int] -> Int` `(*) :: Int -> Int -> Int`
$\bigvee_{x \in S}(x \leq 5)$	`or (map (\x -> x <= 5) s)`	`or :: [Bool] -> Bool` `(||) :: Bool -> Bool -> Bool`
$\bigwedge_{x \in S}(x \leq 5)$	`and (map (\x -> x <= 5) s)`	`and :: [Bool] -> Bool` `(&&) :: Bool -> Bool -> Bool`

One quick warning: this table conveys the important idea, but glosses over the difference between sets and lists, i.e. a list can contain multiple occurrences of something, whereas a set cannot. So, for example:

`sum (map (\x -> x + 10) [1,1,2])` $\Longrightarrow^*$ `34` $\qquad \sum_{x \in \{1,1,2\}}(x+10) = 23$

This turns out to not matter with disjunctions and conjunctions though, because they have the nice property that $\phi \vee \phi = \phi$ and $\phi \wedge \phi = \phi$. (The fact that lists, unlike sets, are *ordered* also never matters, because all these operations are *commutative*. It's also a good thing that all these operations are *associative*, if you think about it.)

[3]The hidden underlying shared idea here is known as a *fold*. See e.g. `https://en.wikipedia.org/wiki/Fold_(higher-order_function)`.