Code (Week 1 Wed)

Functions from the standard library

All of the information about how these functions work is in their type signature.

When reading types in Haskell, names that start with a lowercase letter are type variables and names that start with an uppercase letter are concrete types.

We can't create values or operate on values of a type variable without restricting the variable in some way. Type classes let us do that by specifying operations must be implemented for the type (`Ord` requires `a <= a` and `Eq` requires `a == a`).

Type variables help generic functions express more of their properties in their type signature. For example, you know that a function with the type `[a] -> [a]` can re-order, duplicate, or remove elements from the list, but without knowing the type of the elements it cannot operate on them or add new elements.

`id` takes a value and returns it. This is more useful than you might expect.

```
id :: a -> a
id a = a
```

`flip` swaps the order of the first two arguments to a function.

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f left right = f right left
```

`undefined` allows your code to compile but will crash if it is ever evaluated. This can be useful to incrementally build out parts of your system.

```
undefined :: a
```

`error` will crash your program with a particular error if it is ever evaluated.

```
error :: String -> a
```

```
oops :: a
oops = error "Oops!"
```

Operators

Operators are normal functions in Haskell. An operator is merely a function with a name that consists of non-alphanumeric characters.

An operator can be used as a function by surrounding it with parentheses, so `(.) f g` is the same as `f . g`.

A normal function can be used as an infix operator by surrounding it with backtiks, so `div a b` is the same as ``a `div` b``.

Infix operators have associativity, which determines the order in which different operators implicitly bind and the direction in which multiple uses of the same operator implicitly bind.

The `->` operator on types is right-associative, so `a -> b -> c -> d` is equivalent to `a -> (b -> (c -> d))`.

Function application in expressions is written as a whitespace between to expressions. It is left-associative so `a b c d` is equivalent to `((a b) c) d`.

The `$` reverses the associativity of function application so `a $ b $ c $ d` is equivalent to `a (b (c d))`. This makes it useful to remove trailing parentheses.

Custom Operators

You can also define your own infix operators. You may want to explicitly specify whether they are right associative with `infixr` or left associative with `infixl`

This definition of `(.>)` is like `(.)` but with arguments in reverse order.

```
infixl 0 .>
(.>) :: (a -> b) -> (b -> c) -> a -> c
a .> b = b . a
```

This definition of `(|>)` is like `($)` but with arguments in reverse order.

```
infixl 0 |>
(|>) :: a -> (a -> b) -> b
a |> f = f a
```

List

`append` places all the elements in the first list before all the elements in the second list, appending the two together.

This is equivalent to the infix operator `(++)`.

```
append :: [a] -> [a] -> [a]
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

Indexing `(!!)` is a partial function that gets the nth element in the list:

```
(!!) :: [a] -> Int -> a
(x:_ ) !! 0 = x
(_:xs) !! n = xs !! n - 1
[]     !! _ = error "Index too large"
```

`foldl` and `foldr` are two different ways to 'collapse' a list into a single value. Both take some operation that will 'fold' each value in the list into the working state and some state to start with.

The key difference is that `foldl` will take elements at the head of the list until none are left and `foldr` will go to the end of the least and will step backwards from the end taking each element.

```
-- foldr f base [a, b, c] == a `f` (b `f` (c `f` base))
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f base []     = base
foldr f base (x:xs) = x `f` foldr f base xs

-- foldl f acc [a, b, c] == ((acc `f` a) `f` b) `f` c
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     = acc
foldl f acc (x:xs) = foldl f (acc `f` x) xs
```

As lists are constructed by starting at the end and pushing elements onto the start of the list, `foldr` tends to be more useful in operating on lists in a way that preserves the order. In fact, any basic recursive function on lists can be expressed with `foldr`.

For `map`, the 'base' is an empty list and the operation is one that maps the value then pushes it on the start of the list.

```
map :: (a -> b) -> [a] -> [b]
map f = foldr (\next mapped -> f next:mapped) []
```

For `append`, the 'base' is the list that will end up at the end and the operation is `(:)` which pushes an item onto the front of the list.

```
append :: [a] -> ([a] -> [a])
append front back = foldr' (:) back front
```

For `filter`, the 'base' is an empty list and the operation is one that only includes an element if the predicate `p` returns true for that item.

```
filter'' :: (a -> Bool) -> [a] -> [a]
filter'' p = foldr' step []
  where
    step next filtered
      | p next     = next:filtered
      | otherwise = filtered
```

Binding and scope

In `append` below, the first argument (with type `[a]`) is 'bound' to the name `front` and the second argument (with type `[a]`) to `back`.

The names `front` and `back` are only 'visible' (i.e. they can only be used) within the definition of `concat`, on the right side of `=`.

```
concat :: [a] -> [a] -> [a]
concat front back = foldr (:) back front
```

The following are equivalent ways of writing the same definition.

They show how functions in Haskell fundamentally only take a single argument, even if we can write them as though they take more than one.

```
append :: [a] -> [a] -> [a]
append = \front -> \back -> foldr (:) back front
append = \front back -> foldr (:) back front
append front = \back -> foldr (:) back front
append front back = foldr (:) back front
```

The two definitions of `filter` below are equivalent, the choice of which you would want to use is a matter of your personal preference for style.

In each, a helper function called `step` is defined with the type `a -> [a] -> [a]` (where `a` is actually the `a` from the type of `filter` rather than 'any type'.

`step` is in scope for the entire definition of filter, however `next` and `filtered`, the two arguments of `step` are only in scope for the definition of `step`, including its guards.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr step []
  where
    step next filtered
      | p next     = next:filtered
      | otherwise = filtered

filter :: (a -> Bool) -> [a] -> [a]
filter p =
  let
    step next filtered
      | p next     = next:filtered
      | otherwise = filtered
  in
    foldr step []
```

Partial application and currying

In the following definition of `map`, the type suggests that it takes 2 arguments but it only binds one to a name. `foldr` also takes 3 arguments but only two have been applied.

In this case, we have a 'partially applied' `foldr`. As functions in Haskell only take a single argument, multiple arguments are passed by having each argument but the last produce a function that accepts the next argument.

This definition of `map` works as it provides the `step` function and the 'base' state to `foldr` that will apply the mapping function then push the result onto the start of the working list.

As the last argument to `foldr` would also be the last argument to `map`, `map` returns the function produced by `foldr` with only the first two arguments applied.

The second definition is equivalent to the first:

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

map :: (a -> b) -> [a] -> [b]
map f = foldr (\next mapped -> f next:mapped) []

map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\next mapped -> f next:mapped) [] xs
```