COMP304 Tutorial 3
28/07/2016

1 Lambdas

A lambda is a function which has no name (sometimes called an anonymous function). The functions we've defined in Haskell thus far have consisted of a name, arguments, and a body. A lambda just has arguments and a body. We've seen two ways to define the `max` function.

```
1 mymax x y = if y > x then y else x
2
3 mymax x y
4    | y > x = y
5    | otherwise = x
```

Here's a third, equivalent way, by giving an appropriate lambda the name `mymax`.

```
1 mymax :: Ord a => a -> a -> a
2 mymax = \x y -> if y > x then y else x
```

Note that in this case you need the explicit type annotation or `GHCI` will complain. The slash denotes the beginning of a lambda. It's supposed to resemble the Greek lambda λ. Practically you wouldn't use lambdas in this way, but they are tremendously useful for higher-order functions.

Lastly, we must mention the importance of lambdas in computability theory. Lambda functions on their own are a *model of computation*. A model of computation attempts to capture the notion of what it means for something to be computable. Another model of computation you may have heard of is the Turing machine.

A natural question to ask is whether some models are more powerful than others. The answer is yes, but there *seems* to be a cap at a set of models which are called Turing complete. This cap is known as the Church-Turing thesis.

This group includes Turing machines and the lambda calculus. Each model describes a simple "programming language". Any program you write in one can be written in the other. Modern languages like Java or Haskell or C are also in this group. Turing machine and lambda functions are *so* simple that even trivial programs (such as adding two numbers) are of enormous length. However, their simplicity makes them amenable to mathematical reasoning. Most functional programming languages are based on (some variant of) the lambda calculus, which includes Haskell.

2 Currying

We've talked briefly about currying as a means of "fixing inputs" to a function. Recall the type of `max`.

```
Prelude> :type max
max :: Ord a => a -> a -> a
```

Multi-variable functions such as `max` are *syntactic sugar* really a sequence of nested lambdas which have been given a name. `max` is *really* a lambda which takes an argument x of type `a` and returns a lambda of type $\texttt{a} \to \texttt{a}$. If you give that lambda an argument y of type $\texttt{a} \to \texttt{a}$ it will then return the maximum of x and y. We illustrate this below by supplying the first input to `max` and then binding the resulting function to the name `mymax`.

```
Prelude> :type max
max :: Ord a => a -> a -> a
Prelude> let mymax = max 3
Prelude> mymax 4
4
Prelude> mymax 2
3
```

In light of this we can view function application in another way. Recall that function application is left associative, so the following are equivalent

```
max 3 4
(max 3) 4
```

It's now clear why function application has to be left associative: `max 3 4` is *really* `(max 3) 4` which is the result of passing the argument `4` to the lambda returned by `max 3` .

Until now we've done nothing to really warrant caring about lambdas or currying, but together with higher-order functions we can do some neat stuff. There are a slew of higher-order functions that you will use over and over in functional programming languages, so let's talk about some.

3 Filter

Intuitively, `filter` takes a collection and produces a new collection containing only those elements which satisfy some predicate. Here is its signature.

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

Let's say we wanted the positive elements in a list. Our function for checking if an element is positive will be $>$, with the right-side argument fixed to 0 (so we've curried it into a function which checks if an input is greater than 0).

```
Prelude> filter (>0) [-2,-1,0,1,2,3]
[1,2,3]
```

A palindrome is a word that is spelled the same forwards as it is backwards. If `s` is a palindrome, then `s` $==$ `reverses`. If we pass that into `filter` using a lambda, we can extract the palindromes in a list.

```
Prelude> filter (\s -> s == reverse s) [``racecar'', ``lindsay'', ``carpet'', ``mum'']
[``racecar'', ``mum'']
```

We might like to give that function a name so it can be used over and over again. We can achieve this by currying: we only supply the predicate to `filter`. This gives us a function which, when given a list, returns the sublist of items satisfying the predicate.

```
palindromes xs = filter (\s -> s == reverse s) xs
```

Here's something cool: because `filter` (\s → s == `reverses`) is a function which takes a list as input and returns the sum, the `xs` parameter is redundant. We can define `palindromes` like this:

```
palindromes = filter (\s -> s == reverse s)
```

This way of defining functions is called "point-free programming". Some people think it is needlessly obscure and prefer the name "pointless programming". It's up to you.

4 Map

Intuitively `map` takes a collection of values and produces a new collection by applying some function to each element. Your standard kind of `map` applies to a list, so let's inspect the type signature for `map`.

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]
```

The first argument is a function from `a` → `b`; the second argument is a list [`a`]; the result is a list [`b`]. So the output list doesn't have to have the same type as the input list. Here we map a list of strings to lengths.

```
Prelude> map length [``higher'', ``order'', ``functions'']
[6,5,9]
```

Here we make use of a lambda to fix the second parameter of the > operator to check if each element is positive.

```
Prelude> map (>0) [4,7,5]
[True, True, True]
```

If we wanted to check if a list only contained positive numbers we could combine that with `and`.

```
Prelude> and (map (>0) [4,7,5])
True
```

5 Fold

The basic idea of a `fold` is that you take a collection of items and a unit value. Starting with the unit value, you combine everything in the list using a function and produce a final value. For example, summing a list of numbers is a kind of `fold` where, starting from 0, you repeatedly combine elements with + to produce the final sum of the elements in the list. repeatedly combine elements with + to produce a final value (the sum of the list). `fold` is sometimes called `reduce` in other languages.

In Haskell there is a left fold (`foldl`) and a right fold (`foldl`). The difference is in how the combination happens: `foldl` is left-to-right, while `foldr` is right-to-left. Taking `sum` on $[1, 2, 3]$ as our example, with unit value 0, a left-fold version would look like this:

```
(((0 + 1) + 2) + 3)
```

A right-fold version would look like this:

```
(1 + (2 + (3 + 0)))
```

Because + is associative, that is $(a + b) + c = a + (b + c) = a + b + c$. the two folds will do the same thing. A lot of functions aren't associative, like subtraction, so the choice of `foldl` or `foldr` is important.

First let's inspect the type signature of `foldl`.

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

`b` is the type of the elements is our list. `a` is the type that your combination produces. The fourth/return type is `a`. The third argument is the list to be folded. The second argument is the unit value. The first argument is the combination function. It takes the folded value so far (type `a`) and the next element in the list (type `b`) and returns the value of combining those two things.

With this in mind we can define `sum` as a special case of `foldl` by fixing the first two arguments. We're left with a function which takes a list as input and returns the sum.

```
mysum xs = foldl (+) xs
```

Here's something cool: because `foldl` (+) is a function which takes a list as input and returns the sum, we can simply define `mysum` to be `foldl` (+).

```
mysum = foldl (+)
```

This means the same as what we had before. This style of programming is called "point-free" programming. Some people think it is a bit too obscure and call it "pointless" programming. Up to you how you want to program.

One way to think about the unit value of fold is as the base case of the combination. Another way is that it is the value that you should get when you fold an empty list. Here's a fold which takes a list of list and combines them into a single list L. The base case is an empty list; at each step we want to take the next list and combine it with L by concatenation.

```
Prelude> foldl (++) [] [[1,2,3], [4,5], [6], [], [7,8,9]]
[1,2,3,4,5,6,7,8,9]
```

Sometimes it doesn't really make sense to have a unit value or fold on an empty list. For example, when taking the maximum of a list, what should the maximum of an empty list be? This isn't well-defined; there are versions of fold for non-empty lists called `foldl1` and `foldr1`. They're the same except they don't take a unit value. `foldl1` just takes the first element of the list as your "unit value".

```
Prelude> foldl1 max [1,2,3,4]
4
```

The beauty of these higher-order functions is in how you combine them. Here's how we can use `map` and `fold` to find the size of the longest string in a list.

```
Prelude> foldl1 max (map (\s -> length s) [``glass'', ``gypsum'', ``ash''])
6
```

Your intuition might tell you that this is inefficient: we construct a list of lengths, then we do a second pass to find the maximum. It would be more efficient to iterate over each string, check its length, then take the maximum of that with our current best. Because of lazy evaluation, this is exactly what happens.

To illustrate the difference between `foldl` and `foldr` let's try using subtraction as our combination function.

```
Prelude> foldl (-) 0 [1,2,3]
-6
Prelude> foldr (-) 0 [1,2,3]
2
```

If we expand the second we see why we get 2:

```
(1 - (2 - (3 - 0)))
(1 - (2 - 3))
(1 - (-1))
2
```

6 Zip

`Zip` takes two lists and returns the list of pairs.

```
Prelude> zip [1,2,3] [``higher'', ``order'', ``functions'', ``when?'']
[(1, ``higher''), (2, ``order''), (3, ``functions'')]
```

If the input lists are of differing length, the output list is only as long as the shorter of the two.

A more general function is `zipWith` which combines two lists into a third by applying a function pairwise. `zip` is the same as performing `zipWith` using the function which pairs its inputs.

```
Prelude> zipWith (\x y -> (x,y)) [1,2] [``haskell'', ``prolog'']
[(1, ``haskell''), (2, ``prolog'')]
```

Here's another example which takes two lists and produces a third by taking the pairwise maximum of the numbers.

```
Prelude> zipWith max [1,4, 0] [4, 2, 4]
[4,4,4]
```

7 List Comprehensions

There are useful way to build lists in Haskell. The first is ranges. Here is the list of all natural numbers:

```
Prelude> nats = [0, 1..]
```

This is an infinite list. Because of lazy evaluation, we never try to construct the list in its entirety; only the portion which we use. You can think of an infinite list as being a function which generates successive elements in the sequence. To actually use it we can use e.g. `take`.

```
Prelude> take 3 nats
[0,1,2]
```

Be careful not to e.g. try and take the last element `last nats`; there isn't one, so it will recurse forever. By specifying different starting points in the range we get a different list. Here are the negative numbers:

```
Prelude> take 3 [0, -1 ..]
[0, -1, -2]
```

List comprehensions let us build a list of elements satisfying some predicate (a bit like `filter`).

Here's a neat use of infinite lists. We're going to find the 1000th prime number. Let's say we had a function `isPrime` which could determine if a number was prime. To find the 1000th prime number we define the list of all prime numbers with a list comprehension and then ask for the number at index 1000.

```
Prelude> let primes = [x | x <- [0..], isPrime x]
Prelude> primes !! 1000
7927
```

Bonus: here's an example of how you might write `isPrime`.

```

-- Return true if a divides b (that is k*a = b for some k).
divides :: Integer -> Integer -> Bool
divides a b = b `mod` a == 0

-- Return True if the input is prime, or False otherwise.
isPrime :: Integer -> Bool
isPrime n
  | n == 2 = True
  | n == 1 = False
  | otherwise = isPrime' n 2
  where isPrime' :: Integer -> Integer -> Bool
        isPrime' n curr
          | curr == 2 = if 2 `divides` n
                        then False
                        else isPrime' n 3
          | otherwise = if curr > (ceiling (sqrt (fromIntegral n)))
                        then True
                      else if curr `divides` n
                        then False
                      else
                        isPrime' n (curr+2)
```