Haskell basics

Joe Moore 07508617059, Joe.Moore@warwick.ac.uk

Lists

List examples:

```
[1,2,3]
['a','b','c','d']
["cat","dog"]
[True,False,False,True,True,False]
```

You can have a list of everything including tuples

Built-in functions that can be applied to lists:

```
Head [1,2,3]
  => 1
Tail [1,2,3]
  => [2,3]
Take 2 [1,2,3]
  => [1,2]
Drop 1 [1,2,3]
  => [2,3]
```

Since strings are lists of chars everything that works on list will work on strings:

```
take 2 "cake"
  => "ca"
```

This is because cake can be represented as a list of chars, `['c','a','k','e']` and so by applying the function `take 2` we return the first 2 elements, `['c','a']` or in other words the string `"ca"`

Boolean expressions of the length of lists can be applied in console like as below:

```
null []
  => True
```

```
null [ 1 , 2 , 3]
  => False
```

Replication is a built in command that can be used on elements to return lists:

```
replicate 0 'a'
  => []

replicate 5 'a'
  => ['a','a','a','a','a']

replicate 2 "Earl Grey"
  => ["Earl Grey","Earl Grey"]
```

Write a `splitAt` function that given a list and an integer splits the list into 2 at the position specified by the given integer:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

Mathematical operations can also be used in lists:

```
[1+2,2*3]
  => [3,6]
```

Alongside Boolean operations:

```
[even 5, odd 3, True, not False]
  => [False,True,True,False]
```

Operators can even be used in lists not just operands:

```
(head [(+),(-)]) 5 1
   => (+) 5 1
   => 6
```

Ranges can be given by filling in 2 or more in a list accompanied by a `..` such as below:

```
[1..4]
  => [1,2,3,4]
['D'..'H']
  => ['D','E','F','G','H']
```

Set intervals (known in Haskell as **ranges**) can be implemented by including 2 values at the start:

```
['a','d'..'m']
  => ['a','d','g','j','m']
[1,3..10]
  => [1,3,5,7,9]
[1.0,1.5..3.0]
  => [1.0,1.5,2.0,2.5,3.0]
```

An example of a list **comprehension** can be seen below:

```
[even n | n <- [0..5]
  => [True,False,True,False,True,False]
```

Multiple generators can be included:

```
[n*m | n <- [0..2], m <- [0..2]]
  => [0,0,0,0,1,2,0,2,4]
  =  [0*0,0*1,0*2,1*0,1*1,1*2,2*0,2*1,2*2]

[n*m | n <- [0..2], m <- [0..n]]
  => [0,0,1,0,2,4]
  =  [0*0,1*0,1*1,2*0,2*1,2*2]
```

Here we see that the each time the multiplications increase up to the bigger number so if we had `n` equal to `[0..4]` Then the final list would include up to `4*4` or `16`

Statements can also contain patterns in the left hand side of the generator:

```
[x | (c,x) <- [('a',5),('b',7)]]
  => [5,7]
```

In this example above the pattern is the `x | (c,x)` which is saying that the function should return the second element, `x` of any pair `(c,x)` passed into it. In the example above those `x` values are `5` and `7` so the function returns `[5,7]`.

Let's look at another example:

```
[length xs | x:xs <- [[1,2],[3,4,5]]]
  => [1,2]
```

Here the pattern is showing that the length of the tail, `xs` of list `x:xs` is what should be returned by the function for all elements passed into it. In the above example the tails of the 2 lists are `2` and `[4,5]` and so have lengths `1` and `2` respectively. Ergo the function returns `[1,2]`

Functions involving lists may also contain predicates.

```
[n | n <- [0..4], mod n 2 = 0]
  => [0,2,4]
```

Here the predicate is `mod n 2 = 0` and so only values between 0 and 4 where, $\frac{n}{2} = 0$ are returned. Hence the result of `[0,2,4]`

Recursive Functions

How do we express loops without mutable state? Recursive functions.
Let's define a `factorial` function which given a number `n` calculates the factorial of n,

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Let us see how this is evaluated:

```
factorial 2
  => 2 * factorial (2-1)
  => 2 * factorial 1
  => 2 * 1 * factorial (1-1)
  => 2 * 1 * factorial 0
  => 2 * 1 * 1
  => 2
```

Now we'll apply this principle to make a function `fib` which given some number `n` will return the n^{th} Fibonacci number

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n - 2)
```

Once more let's see this in action:

```
fib 3
  => fib (3-1) + fib (3-2)
  => fib 2 + fib 1
  => fib(2-1) + fib (2-2) + fib 1
  => fib 1 + fib 0 + fib 1
  => 1 + 1 + 1
  => 3
```

Each call of a function in a regular programming language will need a stack frame for each call. For example for a java function that calculates a factorial, `fac` for the number, `n` passed into it we have n stack frames. Say if `n` is 500 we have 500 stack frames, this is extremely poor memory wise and ergo one should lean towards use of a for, while or do while loop.

Haskell, however, optimises all recursive functions for us. It involves creating a second *prime* function. See the example below for the function `fac`

```
fac :: Int -> Int
fac 0 = 1
face n = n * fac (n-1)

fac' :: Int -> Int -> Int
fac' 0 m = m
fac' n m = fac' (n-1) (n*m)
```

Below is a worked example of how this is used to save memory:

```
face 500
  => fac' 500 1
  => fac' (500-1) * (500*1)
  => fac' 499 (500*1)
  => fac' 499 500
  => fac' (499-1) (499*500)
  => fac' 498 (499*500)
  => fac' 498 249500
```

Here we can see that 4 lines down we have a function call using no more data than in the first line of evaluation. And this continues in a pattern and again in line 6 of the evaluation we have a function that uses no more data than the one we started with. Therefore, the functional approach is much more memory efficient than the equivalent imperative approach. In fact, given a similar imperative method, `frac(n)` , it would be 500 times less memory efficient to evaluate `fac(500)` than `fac(1)` .

Let and where:

Let us revisit one of our programs from the first set of notes (01 Lists):

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

This is not as memory efficient as it could be since it isn't recursive, so we're going to redefine it:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt 0 xs     = ([], xs)
splitAt n []     = ([], [])
splitAt n (x:xs) = (x:ys, zs)
    where (ys, zs) = splitAt (n-1) xs
```

The final line here is called a where clause and allows us to define a set of values in the context of a single specific line. It could also be written like this:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (ys, zs)
    where
        ys = take n xs
        zs = drop n xs
```

Or even with use of a `let`:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt 0 xs     = ([], xs)
splitAt n []     = ([], [])
splitAt n (x:xs) =
    let (ys, zs) = splitAt (n-1) xs
    in (x:ys, zs)
```

This is called a let-binding or a let-expression. It is also possible to define functions with `let` and `where`:

```
fac :: Int -> Int
fac x = go x 1
    where go 0 m = n
    go n m = go (n-1) (n*m)
```

In the given example above `go` is not a global function. It cannot be called from the console. The best equivalent example in imperative programming would be say a private function within a class.

```
intercalate :: String -> [String] -> String
intercalate sep xs = go xs
    where go []     = ""
          go [x]    = x
          go (x:xs) = x ++ sep ++ go ++ xs
```

Another benefit is the fact that not all the parameters need to be implicitly defined or listed. (This is seen in the example above)

```
interclatae "," ["Red","Blue","Green"]
  => "Red,Blue,Green"
interclatae "|||" ["Red","Blue","Green"]
  => "Red|||Blue|||Green"
interclatae "Yellow" ["Red","Blue","Green"]
  => "RedYellowBlueYellowGreen"
```

Algebraic Data Types

Booleans

It is rather unorthodox but the Boolean type is not built into the Haskell language. It is imported but we could define it ourselves:

```
data Bool = True | False
```

And this is also how we can define functions such as the `not` function:

```
not :: Bool -> Bool
not True = False
not False = True
```

We can add parameters to definition too,

```
data Shape = Rect Double Double
           | Circle Double
```

And then we can use that in the definition of further shapes:

```haskell
square :: Double -> Shape
square x = Rect x x

area :: Shape -> DOuble
area (Rect w h) = w * h
area (Circle r) = pi * r^2
```

Exceptions

Let's return a value of a new type, `MaybeInt` if we need something to fail:

```haskell
data MaybeInt = Nothing | Just Int
```

Binary Trees in Haskell

```haskell
data BinTree = Leaf Int
             | Node BinTree BinTree

Leaf :: Int -> BinTree
Node :: BinTree a -> BinTree a -> BinTree a
```