Tutorial 2

James and Aaron

This is latex and haskell!

1 Example

```
1 included = "code include"
```

Description can go between.

```
1 excluded = "code exclude"
```

Note have text (% in this case) immediately follow the `\begin{code}` is important to prevent the interpreter from executing the code that should not be executed.

2 Tests

You can setup your tests so that by running testIncluded your code is tested, if you are going for a more automated approach.

```
1 testIncluded = included == "code include"
2 -- test a b = ...
3 -- testSuite = test 1 2 && test 2 1
```

You could also show what you executed in the terminal when you tested your code.

```
ghci> included == "code include"
true
```

3 Tail recursion

Defining a length function that uses backwards propagation.

```
1 len1 [] = 0
2 len1 (x:xs) = 1 + len1 xs
```

Showing some expansion

```
len1 [1,2,3,4]
1 + len1 [2,3,4]
1 + 1 + len1 [3,4]
1 + 1 + 1 + len1 [4]
1 + 1 + 1 + 1 + len1 []
...
4
```

If we instead define a length function that uses tail recursion (forwards propagation), we get:

```
1 len2 ls = len2' ls 0
2
3 len2' [] c = c
4 len2' (x:xs) c = len2' xs (c + 1)
```

Which expands like

```
len2 [1,2,3,4]
len2' [1,2,3,4] 0
len2' [2,3,4] 1
len2' [3,4] 2
len2' [4] 3
len2' [] 4
4
```

The important difference is that the first way expanded sideways (reflecting that it was having to keep track of several values in memory) while the second way simply did not.

```
1 fact n | n > 0      = n * fact(n-1)
2        | otherwise = 1
```

Note with lists, you still want to add to the front of the list ($O(1)$ with cons) instead of the back (which is $O(n)$). This works naturally for backwards propagation and for tail recursion you need to do a simple trick of building the list backwards and then reversing when you return it.

```
1 ones 0 = []
2 ones x | x > 0      = x:ones (x - 1)
3        | otherwise = error "No negatives!"
```