COMP105 Lecture 14

Anonymous Functions

Anonymous functions

Sometimes it is convenient to define a function **inline**

```
ghci> (\x -> x + 1) 2
3
```

```
ghci> :t (\x -> x+1)
(\x -> x+1) :: Num a => a -> a
```

```
ghci> apply_twice (\x -> 2 * x) 2
8
```

These are called **anonymous** functions: they have no name

Anonymous functions syntax

The **syntax** for an anonymous function is:

```
\ [arg1] [arg2] ...  -> [expression]
```

The \ is supposed to resemble a lambda (λ)

- Anonymous functions are sometimes called λ-functions

Examples:

```
\ x y -> x + y + 1
```

```
\ list -> head list + last list
```

Functions that return functions

Higher order functions can also **return** other functions

```
f_that_adds_n :: Int -> (Int -> Int)
f_that_adds_n n = (\ x -> x + n)
```

```
ghci> let f = (f_that_adds_n 10) in (f 1)
11

ghci> (f_that_adds_n 20) 1
21

ghci> (f_that_adds_n 2 . f_that_adds_n 3) 0
5
```

Functions that take and return functions

Higher order functions can take **and** return functions

```
swap :: (a -> b -> c) -> (b -> a -> c)
swap f = \ x y -> f y x
```

```
ghci> take 4 [1..10]
[1,2,3,4]
```

```
ghci> (swap take) [1..10] 4
[1,2,3,4]
```

Currying revisited

Previously we've seen that it is possible to **partially** apply a function

```
add_two = (+2)

ghci> add_two 2
4
```

```
drop_six = drop 6

ghci> drop_six [1..10]
[7,8,9,10]
```

Currying revisited

This is just **nicer syntax** for a function that returns a function

```
add_two = (+2)

add_two' = (\ x -> x + 2)
```

```
drop_six = drop 6

drop_six' = (\ x -> drop 6 x)
```

Exercises

1. In ghci, write an anonymous function that takes two numbers and adds them together

2. Write a function that takes a character c, and returns a function `f :: [Char] -> [Char]` that adds c to the end of the input string

3. Write a function `curry'` that implements the following type signature `((a, b) -> c) -> a -> b -> c`