COMP105 Lecture 4

If

If

If we want to do more than just simple maths, we'll need some decision making

In imperative languages `if` changes the **control flow**

```
if (x == 10)
{
    // do lots of stuff
}
else
{
    // do some other stuff
}
```

But there is no control flow in functional programming ...

Functional `If`

Functional programming still has `if`

```
gt10 x = if x > 10 then "yes" else "no"
```

Think of functional `if` as a function

- We test some boolean condition `x > 10`
- If it's true then the function returns "yes"
- If it's false then the function returns "no"

`If` really is a function

We can treat `if` like any other function:

```
f x = (if x > 10 then 1 else 0) + 2
```

So `f(11) = 3` and `f(9) = 2`

Rather than controlling flow, functional `if` chooses between **two alternatives**

If as a function

```
f x = (if x > 10 then 1 else 0) + 2
```

Both branches must **always** be present

- You cannot skip the `else`
- The following code will cause an **error**

  ```
  f x = if x > 10 then 1
  ```

Remember that functions **always** return a value

If as a function

Both branches **must** have the **same type**

- `if x > 10 then 1 else "no"` will cause an error

The following `if`s are correct:

```
if x == 1 then 1 else 0

if x /= 1 then "Yes" else "No"
```

This is because Haskell is **strongly typed**

- We will discuss this further in an upcoming lecture

The structure of an `If`

The syntax is

```
if A then B else C
```

where

- A is an expression that evaluates to True or False
- B and C are expressions that evaluates to the same type

Reminder: an **expression** is anything you can put

- in a function body
- as a query in ghci

More complex `if`s

```
if (a > 10 && b < 20) then (a + 12) else (min a b)

if (test c) then (answer c) else (answer c + answer d)
```

You really can put **any function** in an `if`

- You can do complex tests in part A
- Parts B and C can be the result of other functions, or combinations of them

You can even put `ifs` in your `ifs`

```haskell
if (a /= b) then (if a == 1 then a else b) else 0
```

The point is you can put **any** expression in an `if`

Nested `ifs` are not used that frequently in Haskell

- There are better ways to do this kind of thing
- We will discuss them in future lectures

Functional `if` in imperative languages

The functional `if` is more commonly known as the **ternary operator** in imperative languages

In C or Java:

```
int x = (y == 1) ? 1 : 0;
```

In Python:

```
x = 1 if y == 1 else 0
```

Exercises

1. Write a function `between36` that takes one input number and returns `"yes"` it is greater than 3 and less than 6, and `"no"` otherwise

2. Write a function `min'` that takes two numbers and returns the minimum of the two

3. Write a function `max3` that takes three arguments and returns their maximum