COMP105 Lecture 3

Writing Our Own Functions

Defining your own functions

So far we've just used the interpreter to evaluated predefined functions.

For the most part we will program in Haskell by **defining** our own functions

We do this by saving our functions in a file, and then loading them into ghci

Alternatively we can **compile** our file to create a runnable program

A first function

Let's write a function that adds two to its input

```
addTwo x = x + 2
```

We can save it in `our_code.hs` and then load into ghci

```
Prelude> :l our_code.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> addTwo 2
4
```

Function syntax

The syntax for defining a function is

```
function_name arg_1 arg_2 arg_3 ... = body
```

Where

- `function_name` must start with a small letter
- all arguments must start with small letters
- `body` is some **expression**

An expression is a combination of functions

- Anything you can type into ghci is an expression
- Think of it as an imperative function that immidiately uses `return <something>`

Examples of functions

```
addTwo x = x + 2

addSquares x y = x**2 + y**2

maxThree x y z = max (max x y) z
```

Further example

Converting a two-bit binary number $b_1 b_2$ to decimal:

$$b_1 * 2^1 + b_2 * 2^0$$

So 11 in binary becomes $1 \times 2 + 1 = 3$

In Haskell:

```
pow2 x = 2^x

bin_to_dec b1 b2 = b1 * pow2 1 + b2 * pow2 0
```

Comments

It's a good idea to **comment** your code

```
-- A single line comment looks like this

1 + 1 -- It can go after code on the same line

{- A multi-line comment uses this syntax
    It can
    go accross
    many lines -}
```

Comparison with imperative languages

Compared to imperative languages, functions in Haskell tend to look **much shorter**

- Imperative functions do lots of things. Set up variables, go around loops, etc.
- Since we only care about the return value of a function, the body tends to be much shorter, because the function usually only does **one** thing

It will take time to get used to the idea of

```
f x y z = <one thing>
```

Compilation

Instead of loading into ghci, we could also **compile** our code

```
addTwo x = x + 2

main = putStrLn (show (addTwo 4))
```

- `show` turns its input into a string
- `putStrLn` prints out a string (side effect!)
- `main` is the function that will be run by our program

This is **I/O** code, which we will not cover until later in the course

- For now it's fine to stick with ghci

Where are we?

We have so far seen how to

- Do basic maths in Haskell
- Use some predefined functions
- Write our own functions

This allows us to use Haskell as a fancy calculator

- Of course it can do much more
- We will expand on this base of knowledge over the coming weeks
- We will unlock the full power of the language when we start doing recursion

Exercises

1. Write a function `double` that takes one argument x and returns two times x

2. Write a function `pythagoras` that takes arguments a and b, and returns the square root of a squared plus b squared. The `sqrt` function in prelude will give a square root

3. Write a function `maxFour` that takes four arguments and returns the maximum of them