# CA320 Computability and Complexity

## Lab 2: Defining Functions in Haskell

### Aim

The aim of this lab is to help you learn to define simple Haskell functions. Also, you will be introduced to some common error messages produced by GHCi.

### 1. GHCi Error Messages

The purpose of this exercise is to show you some common GHCi error messages and to get you to interpret them and correct them. The script `errors.hs` contains a few simple and common errors. You should begin by loading this into GHCi. GHCi will respond by reporting an error and a line number where the error was found. Using the editor, you should find that line and try to figure out what the error is, and correct it. Then reload the script to find the next error. You can just type `:reload` or even just `:r` at the GHCi prompt to reload the script.

Note that getting used to the error messages and what they really mean is an important and often quite difficult part of learning to use a programming language effectively. Often the messages require deep knowledge of the language to understand them fully. In languages like Haskell, the type mismatch information can be hard to fathom.

### 2. Area of a Triangle

The area of a triangle with sides $a$, $b$, $c$ is given by the formula:

$$\sqrt{s(s - a)(s - b)(s - c)}$$

where

$$s = (a + b + c)/2$$

Design a Haskell function `triangleArea` to calculate the area of a triangle given the lengths of its sides.

The type of `triangleArea` should be `Float -> Float -> Float -> Float`

The behaviour of `triangleArea` should be as follows:

```
> triangleArea 3 4 5
6.0
> triangleArea 1 2 2.5
0.94991773
> triangleArea 1 1 (sqrt 2)
0.5000001
```

To check your answer is correct (and to register your progress), you should save your program to a file named `triangleArea.hs`, and drag and drop this file to the upload link on the following page: https://ca320.computing.dcu.ie/einstein

### 3. Sum Test

Design a Haskell function `isSum` that takes three integer arguments and tests whether one of them is the sum of the other two.

The behaviour of `isSum` should be as follows:

```
> isSum 1 2 3
True
> isSum 4 9 5
True
> isSum 12 5 7
True
> isSum 23 23 23
False
```

You should start by declaring the type of `isSum` in your script.

To check your answer is correct (and to register your progress), you should save your program to a file named `isSum.hs`, and drag and drop this file to the upload link on the following page: https://ca320.computing.dcu.ie/einstein

**4. Area of a Triangle (revisited)**

Have you considered what your `triangleArea` function from exercise 2 will do with invalid data? For example, there is no triangle with sides 1, 2, 4. What does GHCi give as the value of the expression: `triangleArea 1 2 4`?

Add to your function definition some checks to handle such invalid data and report an error if appropriate. You can use the built-in `error` function for this purpose; it is called with a string argument which is the error message. For example:

```
error "Not a triangle!"
```

The `isSum` function from exercise 3 is also pretty close to what you need for checking that three numbers can really be the sides of a triangle. Begin by modifying this function.

To check your answer is correct (and to register your progress), you should save your program to a file named `triangleArea2.hs`, and drag and drop this file to the upload link on the following page: https://ca320.computing.dcu.ie/einstein

**5. Fibonacci Numbers**

Early in the 13$^{th}$ Century, Leonardo Fibonacci described the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Except for the first two numbers in the sequence, each is the sum of the two numbers preceding it in the sequence. Construct a Haskell function to calculate the $n^{th}$ element of the sequence of Fibonacci numbers. For example:

```
> fibonacci 1
0
> fibonacci 2
1
> fibonacci 3
1
> fibonacci 8
13
> fibonacci 20
```

You should take a similar approach to that used in the `factorial` function shown in lectures. To help jog your memory, here is the definition using guards:

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

`fibonacci` is more complicated than factorial, so here is a hint: your function definition will need to handle three cases: `n = 1`, `n = 2` and `n > 2`.

To check your answer is correct (and to register your progress), you should save your program to a file named `fibonacci.hs`, and drag and drop this file to the upload link on the following page: https://ca320.computing.dcu.ie/einstein

**6. Fibonacci (revisited)**

It would be better to define the `fibonacci` function using patterns rather than guards. Try to do this.

To check your answer is correct (and to register your progress), you should save your program to a file named `fibonacci2.hs`, and drag and drop this file to the upload link on the following page: https://ca320.computing.dcu.ie/einstein