Functional Programming

5CM524

**A computer screen with gears and a monitor

AI-generated content may be incorrect.**

Define Simple Functions

Lab Instructions for 1 hour lab   
of week 3

Dr Youbing Zhao

Prof Stephan Reiff-Marganiec

# Aims

The attached instructions form content for 2 hours of labs. This lab contains no assessed element. Note that there are additional instructions for connecting to the Azure Lab.

1. Define simple functions
2. Checking credit card numbers for validity

# Overview

In this lab you will be learning how to get started in Haskell. We are using an Azure lab with the Haskell compiler preinstalled. The Azure lab has already been setup for you and there are instructions in the lab folder of how to connect. Please make sure that you stop the VM when you are done as otherwise time will keep ticking down!

Note: If you work on your own machine then you can remote connect to the Azure lab or install Haskell from <https://www.haskell.org/ghc/download.html>. Install takes a little while!

# Define Simple Functions

Let’s look at writing some simple functions. We will learn more about this in the next couple of lectures but doing it and it seeing it in action will greatly help you with the class later on.

Task: Create simple functions that take arguments and return a result.

Goal: Learn function definition, function types, and the concept of pure functions.

You can write your functions in a script file (extension .hs) in any editor of your choice and load that on starting GHCi; which then allows you to reload and go back to them easily. You can also load a script from within GHCi using the **:load** ***filename*** command or reload one you used before after making changes using **:reload**. If you have a simple one you can just type it on the command prompt in GHCi.

Let’s start at the prompt. Try this: **square x = x \* x**.

After oppressing enter you are back at the GHCi command prompt and you can now say **square 3**, see what happens.

Now put the function definition into an external script, load that into the environment and use the function. Add a function cube to the script, reload and try to use it.

# Now that you’ve been shown, practice on your own …

Ok here are a few ideas to try, maybe you can come up with more of your own.

Write a function that

… doubles a number

… doubles the squares of a number

… determines if a number is positive, negative, or zero using conditionals

… calculates the area of a circle given the radius (the maths is a=pi r2; Haskells knows **pi** as a constant)

… converts a given temperature in Fahrenheit to Celsius.

… the perimeter of a rectangle given its length and width.

… converts kilometres to miles (1 km = 0.621371 miles)

… converts minutes into hours and minutes (this is more advanced: you want the function to return a pair of numbers which you get by using (), e.g. **functionname x = (x, x)** would return two x’s as a pair.)

… given number is a perfect square (i.e. where the square root is an integer; 16 is a perfect square, 20 is not)

… returns True if a given number is even and False if it's odd. True and False exits as values in Haskell. (**a == b** is a test that will return true or false depending on whether a and b are equal)

Over to you, what can you come up with?

Which things do you want to do and can’t? What is missing in your Haskell repertoire?

# Credit Card Number checking

The final task for this lab is to write a programme that can check a credit card number for validity (to protect from e.g. input errors). Credit card number formats are checked using an algorithm called Luhn’s algorithm, which creates a simple checksum.

Luhn’s algorithm starts at the right and doubles every 2nd number; if doubling gives a 2 digit number the digits are added to give a single digit (e.g. 16 would be converted to 7, for numbers ranging between 10 and 18, it is equivalent to subtract 9 from the number). You then sum all values and if the number can be divided by 10 without a remainder you have a valid credit card number.

Here is an example (note that the algorithm doesn’t say a Credit Card number needs to have a specific length):

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Original number | 9 | 9 | 2 | 7 | 3 | 9 | 8 | 7 | 1 | 3 |
| Double every other | 18 |  | 4 |  | 6 |  | 16 |  | 2 |  |
| Remove 2 digits if needed | 9 |  |  |  |  |  | 7 |  |  |  |
| Numbers to be added for final check | 9 | 9 | 4 | 7 | 6 | 9 | 7 | 7 | 2 | 3 |

The sum of the numbers in the last row is 63; 63 cannot be divided by 10 without remainder so the number is invalid.

Task: Create a programme that checks a credit card number.

Goal: A programme that has a main function checkCCNumber :: Integer -> Bool which returns True if a number has a valid format.

Hints:

I am not providing you many details for this but here are some hints: 1. You want to split the problem into several functions, like splitting the input Integer into a list of Int, dealing with doubling every other number etc. You will need to use **div** and **mod** in the splitting and in some other places. Also, we said quite a few times that a list is **x:xs**, but if you need more detail it could also be seen as **x:y:xs**, which hopefully helps with ‘every other’!