

## CAS London Conference: February 2020

### Practical Sheet: Functional Programming

This sheet is a set of exercises for introducing functional programming using Haskell. An extended version and answers are available from

<http://www.eecs.qmul.ac.uk/~william/CAS-London-2020.html>

#### Notes on using WinGHCi

- Use a text editor (such as notepad) to create a file.
- The file should have extension .hs
- Start WinGHCi and load the file
- Add to your program using the editor; reload it in WinGHCi

#### Online alternative

You can use Haskell online at <https://repl.it/languages/haskell>

- Limited documentation: <https://repl.it/site/blog/haskell>
- Type into the right hand box!

## 1 Simple Functions

In this section, you learn to define some simple functions.

### Exercise 1.1: Enter a Simple Function

Enter the following function and test it using WinGHCi

```
-- Find the larger of two numbers  
bigger a b = if a > b then a else b
```

Here is an example of running the program:

```
*Main> bigger 10 20  
20  
*Main> bigger 12 6  
12  
*Main>
```

### Exercise 1.2: Bigger of Three

Create a second function that finds the largest of three arguments. It starts like this:

```
True -- Find the largest of three numbers  
bigger3 a b c = add text here
```

**Exercise 1.3: More functions**

Practice defining more functions (see the operators on the next page). Here are some suggestions:

- Square a number
- Test whether a number is odd
- The area of a circle (or cylinder) from the radius (using constant `pi`)

Some operators in Haskell:

Operators	Notes	(Interactive) Example
<code>+ * / - ^</code>	Arithmetic	<code>*Main&gt; 3^27</code> <code>7625597484987</code>
<code>== /=</code>	Equals; not equals	<code>*Main&gt; 3 /= 5</code> <code>True</code>
<code>&amp;&amp;    not</code>	Logic	<code>*Main&gt; not (3 == 4)</code> <code>True</code> <code>*Main&gt; (3 == 1 + 2) &amp;&amp; (3 /= 5)</code> <code>True</code>
<code>&lt; &lt;= &gt; &gt;=</code>	Comparison	<code>*Main&gt; (3 &gt;= 6)    (-1 &lt; -2)</code> <code>False</code>
<code>`mod` `div`</code>	Modulo	<code>*Main&gt; 27 `mod` 7</code> <code>6</code> <code>*Main&gt; 27 `div` 7</code> <code>3</code>

**Additional Exercise 1.4: Is it a triangle?**

Three lengths can form a triangle provided that none of them exceeds the sum of the other two. Define a function to check that three lengths make a triangle. Here are some examples of the function being used:

```
*Main> isTriangle 10 10 10
True
*Main> isTriangle 10 10 100
False
*Main> isTriangle 10 40 10
False
```

## 2 Function Composition

This section introduces combining ('composing') functions. This is the way that complex programs are made from simple ones.

**Exercise 2.1:** Given the following function definition (add them to your file):

```
-- Simple functions
double a = 2 * a
square a = a * a
inc a = a + 1
```

Predict the result of the following (and then check your answers):

```
> double (double 5)
> double (square 3)
> square (double 3)
> square (square 3)
> double (double (double 6))
> square (inc (inc 3))
```

### **Additional Exercise 2.2: Heron's Formula**

Heron's Formula for the area of a triangle from the lengths of the sides (a, b, c) is:

$$Area = \sqrt{p(p-a)(p-b)(p-c)}$$

where p is half the perimeter:  $p = \frac{a+b+c}{2}$

Define a function to compute the area. Note however, that the formula assumes a, b and c do define a triangle; use the function 'isTriangle' to check this, returning -1 if the lengths are not a triangle.

An outline of the program is given below, using a new form of syntax:

```
halfP a b c = add text here

area a b c =
  if isTriangle a b c
  then
    let p = halfP a b c in sqrt (add text here)
  else -1
```

The new syntax is a **Let expression**. We can avoid repeating an expression multiple times by giving it a name and using the name. Here 'p' has been used to stand for the value of the half perimeter. The function could be defined without using 'let' by 'halfP a b c' in place of 'p'.

### 3 Recursions

Function definitions can be recursive. Using recursion to define a function is a key technique in Functional Programming.

**Exercise 3.1:** Enter and try the `trigNum` function. All the forms below are equivalent – try them all out.

```
-- Definition using a pattern to distinguish the two cases
trigNum 1 = 1
trigNum n = n + trigNum (n-1)

-- Definition using guard conditions instead of a pattern
trigNum1 n | n == 1     = 1
           | otherwise = n + trigNum1 (n-1)

-- Definition using if expression
trigNum2 n = if n == 1
             then 1
             else n + trigNum2 (n-1)
```

Note that only one definition of a function is allowed so we have used different names for each definition.

**Exercise 3.2:** What happens if the `trigNum` function is called with an argument of zero or less?

**Additional Exercise 3.3:** The familiar factorial function can be defined in a similar way to `trigNum`. In mathematical notation, the factorial is defined by:

$$\text{factorial } n = n \times n - 1 \times \dots \times 2 \times 1$$

Define a Haskell function `fact`. Note that the base case is shown below:

```
-- factorial
fact 0 = 1
fact n = add text here
```

### 4 Lists

Like Python, Haskell has lists (but with the important difference that elements cannot be updated). Lists are often processed recursively:

- The base case is the empty list
- The recursive call is applied to the tail of the list.

#### Some useful functions on lists

Function	Description	Example
<code>:</code>	'Cons' an element to a list	Main> 42:[4,3,2,1,0] [42,4,3,2,1,0] Main> 1:2:[] [1,2]
<code>elem</code>	Member of a list	Main> elem 4 [1,2,3,4,5] True Main> elem 4 [1,3,5] False
<code>head</code>	First element of a list	Main> head [2,4,6,8] 2

Function	Description	Example
tail	List without the first element	Main> tail [3,5,7,9] [5,7,9]
last	Last element of a list	Main> last [3,5,7,9] 9
init	List without the last element	Main> init [2,4,6,8] [2,4,6]
++	Concatenate two lists	Main> [1,2,3] ++ [7,9] [1,2,3,7,9]
null	Test if a list is empty	Main> null [] True Main> null [1] False
length	Length of a list	Main> length [1,3,5] 3
reverse	Reverse a list	Main> reverse [2,7,5] [5,7,2]

**Exercise 4.1:** Head and tail of a list. Predict the result of the following expressions. Check your answers.

```
Main> head [1,2,3,4]
Main> tail [4,3,2,1]
Main> head (tail [2,4,6,8])
```

**Exercise 4.2:** Ranges. Predict the result of the following expressions. Check your answers.

```
Main> [1..5]
Main> [7,11 .. 27]
Main> [7,6 .. 1]
Main> reverse [7,9 .. 21]
Main> length [0 .. 100]
```

**Exercise 4.3:** Sum of a list.

A function to sum the numbers in a list can be defined recursively. The key ideas are:

- The sum of the elements of an empty list is 0
- The sum of the elements of a non-empty list is the head of the list added to the sum of the tail.

Here is an outline of the function. Complete the definition and try it out. Note: the function is called 'addUp' as there is already a function 'sum' defined.

```
-- Add up a list of numbers
addUp [] = 0
addUp (x:xs) = add text here
```

**Additional Exercise 4.4:** Redefining `trigNum` and `fact`. You may have noticed that

- `trigNum n` is the sum of the numbers from 1 to `n`
- `fact n` is the product of the numbers from 1 to `n`

Using this, `trigNum` can be redefined as follows

```
-- Alternative trigNum
trigNumAlt n = sum [1..n]
```

Enter this new definition and test it out. Make a similar redefinition of `fact`. (Hint: you need a function `product`. This function is already defined.)

## 5 Map, Filter and Fold

The functions `map`, `filter` and (various forms of) `fold` capture some common forms of recursive operation on lists. Many recursive definitions can be written using these functions rather than explicit recursion.

**Exercise 5.1:** Map: applying a function to each item in a list.

Given the following functions

```
inc n = n + 1
double n = n * 2
square n = n * n
```

Predict the result of the following.

```
Main> map inc [2,4 .. 8]
Main> map double [2,4 .. 8]
Main> map square [1 .. 4]
Main> map double (map inc [1 .. 4])
Main> map double (map square [1 .. 4])
Main> map square (map double [1 .. 4])
```

**Exercise 5.2:** Filter: picking out a subset of a list.

Given the following function

```
even n = n `mod` 2 == 0
```

Predict the result of the following.

```
Main> filter even [1,2,3,4]
Main> sum (filter even [1..10])
```

**Exercise 5.3:** Using `foldr`

The `sum` function can be defined using `foldr`.

```
sum xs = foldr (+) 0 xs
```

Enter this function and try using it.

**Exercise 5.4:** Using `foldr` to define `product`

The `product` function can be defined using `foldr`, in a similar way to `sum`.

**Additional Exercise 5.5:** Any and All. The functions ‘`and`’ and ‘`or`’ are used to combine a list of Boolean values. Here are some examples of their use:

```
Main> and [True, True, True]
True
Main> and [True, True, False]
False
Main> or [True, True, False]
True
```

- Explain whether these functions are defined using `map` or `foldl`.
- Give a definition of the functions using `map` or `foldl`.

## 6 Challenge Problems

These problems are selected from <https://projecteuler.net> See if you can solve any using Haskell.

### 6.1 Problem 1: Multiples of 3 and 5

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000

#### Outline

- Define a function that test whether a number is divisible by 3 or by 5
- Use the function to filter a list of numbers
- Find the sum of the filtered numbers

### 6.2 Problem 6: Sum square difference

The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \dots + 10^2 = 385$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \dots + 10)^2 = 55^2 = 3025$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is  $3025 - 385 = 2640$ .

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

### 6.3 Problem 4: Largest palindrome product

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is  $9009 = 91 \times 99$ .

Find the largest palindrome made from the product of two 3-digit numbers.

#### Outline

- Define a function that gives the digits of a number as a list. For example 'digits 1815' give `[5, 1, 8, 1]`.
- A palindrome is a number whose list of digits is the same when reversed. For example, 91719 is a palindrome. Define a function to test for palindromes.
- You need to make a list of products of all pairs of number from a pair of lists. This can be done using the `makeProd` function shown below.
- Filter the list of products using the palindrome tester and use the function `maximum` to find the largest number in the list of palindromes.

The `makeProd` function can be defined like this:

```
makeProd as bs = [a*b | a<-as, b<-bs]
```

Here is an example of its use:

```
Main> makeProd [1..5] [1..5]
[1,2,3,4,5,2,4,6,8,10,3,6,9,12,15,4,8,12,16,20,5,10,15,20,25]
```



## 7 Further Reading

You can try any of the following to find out more about Haskell:

- <https://tryhaskell.org/> An outline tutorial. Simple Haskell expressions can be tried in a browser interface.
- An introductory book that can be read online: <http://learnyouahaskell.com/>
- Download the most popular Haskell system: <https://www.haskell.org/platform/>
- Haskell language home page: <https://www.haskell.org/>