Give it a go!

In your lab class you should be working on the exercises that reinforce the topics that are given in the pre-recorded videos each week. This means that you really need to watch those videos before your lab class. There will be under an hour of videos to watch each week, so make sure you set aside some time for it before your lab class.

Some of these exercises are simple, but others are considerably more challenging. You really need to work through them all to ensure that you really understand the material.

You won't necessarily finish all the exercises in your scheduled lab class each week, but you *should* be spending more than just your scheduled class time on this module. If you need extra explanations, come to the drop-in session or ask in a lab class in a later week.

Solutions to the lab exercises will be made available at the end of the *following* week (so week 2 exercise solutions will be made available at the end of week 3). **Please** have a go at the exercises before you look at the solutions; understanding a solution is not the same as being able to write it.

Note: exercises labelled [Hutton, ex N.X] refers to exercise X from chapter N of Graham Hutton's "Programming in Haskell, Second Edition" (2016).

Remember: if you are writing a function, you should include a type definition for it (and any subsidiary functions that you write as part of the definition). Also remember to use descriptive names and to write comments to explain your code.

Week 2

You will need to explore the standard library to develop solutions to some of these problems. In many cases, there is more than one possible solution. Talk to your peers and see if you are coming up with the same solutions. If you come up with different ones, discuss the pros and cons of each solution.

1. [Hutton, ex 2.3] The script below contains three syntactic errors. Correct these errors and then check that your script works properly.

```
N = a 'div' length xs
  where
    a = 10
   xs = [1, 2, 3, 4, 5]
```

2. What are the types of the following functions?

```
second xs = head (tail xs)
swap (x,y) = (y,x)
pair x y = (x,y)
double x = x*2
palin xs = reverse xs == xs
twice f x = f (f x)
```

(You can easily check this by putting the definitions in a file, loading the file, then entering the command

:type <function>

where <function> is the function name you want to check. But before you do this, see if **you** can work it out. When you check, if there are differences, try to work out why.)

3. [Hutton, ex 4.1] Using library functions, define a function

```
halve :: [a] -> ([a],[a])
```

that splits a list of even length into two halves. For example:

```
> halve [1, 2, 3, 4, 5, 6]
([1, 2, 3], [4, 5, 6])
```

4. Implement a function for computing the Euclidean distance between two points, (x1,y1) and (x2,y2).
   Reminder: the Euclidean distance is given by:

$$\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

5. Define a function that uses library functions to return the first word in a string. For example:

```
> firstWord "a test string"
"a"
>firstWord "   the trickier test string"
“the”
```

6. Write a new function, safeTail, that instead of generating an error when the input is an empty list (as tail does), simply returns an empty list. See if you can provide two different solutions:
   a) Using a conditional expression
   b) Using guarded equations

7. Define the following functions in a file with their type signatures
   a) stack takes the first element of a list and puts it on the end of a list
   b) range takes a numerical value and checks to see if it is between 0 and 10, returns True if it is False otherwise
   c) addc takes a Char and a String and adds the Char to the beginning of the String
   d) halves takes a list and divides each element in the list by two
   e) capitalizeStart that takes a string as input and returns the same string with the first character capitalized. (If the first character is not a lowercase letter, it should simply return the input string.)

8. Write a function **roots**, with type:

```
roots :: (Float, Float, Float) -> (Float, Float)
```

   to return the roots of an equation, where the input tuple represents the values of a, b and c in the above equation.

9. Write a function that returns all the odd items in a list using the library function filter and an appropriate where clause
   oddItems :: [Int] -> [Int]

10. For the following lists, predict the values that will be in the list, then check your answer using gchi
   a) [-1..1]
   b) [0.1..1]
   c) [1.2,0.9..0]
   d) [10..1]

Where your prediction was wrong, do you understand why? *(Don't worry about floating point inaccuracies!)*

11. A triple of positive integers (x, y, z) is *Pythagorean* if it satisfies the equation $x^2 + y^2 = x^2$. Define a function
   pyths :: Int -> [(Int, Int, Int)]
   that returns the list of all such triples whose components are at most a give limit. For example:

```
> pyths 10
  [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
```

12. [Hutton, ex. 4.8] The *Luhn algorithm* is used to check bank card numbers for simple errors, such as mistyping a digit, and proceeds as follows:
    a) Consider each digit as a separate number;
    b) Moving left, double every other number from the second last;
    c) Subtract 9 from each number that is now greater than 9;
    d) Add all the resulting numbers together;
    e) If the total is divisible by 10, the card number is valid.

    Define a function

    luhnDouble :: Int -> Int

    that doubles a digit and subtracts 9 if the result is greater than 9. For example:

    ```
    > luhnDouble 3
    6
    > luhnDouble 6
    3
    ```

    Using luhnDouble and the integer remainder function mod, define a function

    luhn :: Int -> Int -> Int -> Int -> Bool

    that decides if a four-digit bank card number is valid. For example:

    ```
    > luhn 1 7 8 4
    True
    > luhn 4 7 8 3
    False
    ```

Week 3

13. Here is alternative definition for a function that has already been introduced:
    (\ (_:xs) -> xs)

What function is it? Make sure that you understand how this lambda expression works.

14. Write lambda expressions to perform the following tasks:
    a) Increment an integer value
    b) Decrement a value
    c) (Harder!) Check if a value is a prime number

15. Given the list of tuples **records** below, write the **choose** function using filter and a lambda expression so that it returns the tuples that have **True** and a number below 10:

```
records = [(True,5), (False,7), (True,12), (True,8), (False,15), (True,4)]

choose :: (Bool, Num a) => [(Bool, a)] -> [(Bool, a)]
choose = undefined
-- choose records should return
-- [(True,5),(True,8),(True,4)]
```

16. Rewrite the list comprehension:
    [x ^ 2 | x <- [1..20], even x]

using the library functions map and filter.

17. In the recordings, two definitions of factorial were given. It was remarked that both have a problem: they won't work for negative input. In fact, the factorial of a negative number is undefined. Modify either definition of factorial so that it reports an appropriate error if the input is negative.

18. In this week's recordings, we examined the following implementation of length:

```
length :: [a] -> Int

length [] = 0

length (x:xs) = 1 + length xs
```

If we were to evaluate the function call **`length [1,2,3]`** we would get:

```
length [1,2,3]
  = length (1:[2,3])
  = 1 + length [2,3]
  = 1 + length (2:[3])
  = 1 + 1 + length [3]
  = 1 + 1 + length (3:[])
  = 1 + 1 + 1 + length []
  = 1 + 1 + 1 + 0
```

Look at the definition of nth from the same lecture, and using the same approach as above, show the evaluation of **`nth 5 [9,4,10,1,2,6,9]`**

19. Write a function that returns a list of the indices of all instances of x in a list. For example

    indices 5 [5,7,9,9,5,1,2,5]

    should return [0,4,7] and

    indices 'a' "test text"

    should return []

20. [Hutton, ex 6.4] Define a recursive function Euclid :: Int -> Int -> Int that implements *Euclid's algorithm* for calculating the greatest common divisor of two non-negative integers: if the two numbers are equal, this number is the result; otherwise, the smaller number is subtracted from the larger, and the same process is repeated. For example:

    > euclid 6 27

    3

21. [Hutton, ex 6.8] Construct definitions for the library functions that:
    a. calculate the **`sum`** of a list of numbers;
    b. **`take`** a given number of elements from the start of a list;
    c. select the **`last`** element of a non-empty list.

22. Construct a definition for the library function **`zip`.** This function takes two lists and "zips" them together, producing a list of pairs, such that the first pair contains the first item from the first list and the first item from the second, and so on. If one list is shorter than the other, the function stops when the shorter list runs out of items. For example:

    zip [1,2,3] "abcdefg" = [(1,'a'), (2,'b'), (3,'c')]

23. Write a function that takes a list of (String, Int) pairs, representing student names and grades, and returns a tuple of how many students fall in the following grade bands: < 30, from 30 to < 40, from 40 to < 50, from 50 to < 60, from 60 to < 70, and 70+.

24. [Hutton, ex 5.2] Suppose that a *coordinate grid* of size $m \times n$ is given by the list of all pairs (*x, y*) of integers such that $0 \leq x \leq m$ and $0 \leq y \leq n$. Using a list comprehension, define a function grid :: Int -> Int -> [(Int, Int)] that returns a coordinate grid of a given size. For example,

    > grid 1 2

    [(0,0), (0,1), (0,2), (1,0), (1, 1), (1,2)]

25. [Hutton, ex 5.6] A positive integer is *perfect* if it equals the sum of all its factors, excluding the number itself. Using a list comprehension and the function factors (which you will need to define if you have not already), define a function

    perfects :: Int -> [Int]

    that returns the list of all perfect numbers up to a given limit. For example:

    > perfects 500

    [6, 28, 496]

26. As mentioned in the recordings, there are several library functions that are available to operate upon lists. You have two techniques now for working with lists: list comprehension and recursion. Practice these techniques by writing your own versions of some of the library functions. Good ones to try are: replicate, take, sum, unzip, reverse. Which ones are better suited to a recursive solution? Which ones a list comprehension?

27. The Leibniz formula for π is given by:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

Write a function approx_pi that takes a single argument: the tolerance you want for the return value. (That is, if the tolerance is 0.001, the return value should be no more than 0.001 from the true value of π.) The output should be a tuple: the first value in the tuple should be the approximation of the value of π, accurate at this tolerance. The second item in the tuple should be the number of recursive steps that were required to get this accuracy.
For example,

```
> approx._pi 0.001
(3.1420924036835256,2000)
```

(Yes, it converges **very** slowly.)

28. From the recordings: You were shown the way to encode a single character using a Caesar cipher. Write three different versions of a function to encode an entire string:
    a) Using map
    b) Using list comprehension
    c) Using recursion

29. Express the comprehension [f x | x ← xs, p x] using the functions map and filter.

Week 4

30. In the lecture on user-defined types, the Shape type was introduced:

```
data Shape = Circle Float | Rect Float Float
```

Define a function

```
scale :: Float -> Shape -> Shape
```

that takes a scaling factor and a shape, and returns the shape with the scaling factor applied. For example:

```
> scale 2.0 (Circle 1)
Circle 2.0
> scale 2.0 (Rect 1 2)
Rect 2.0 4.0
```

Note that in order to test your code, you will have to add "deriving Show" to the end of the type definition:

```
data Shape = Circle Float | Rect Float Float
                deriving Show
```

Don't worry about why this is now; we will come to it in later weeks.

31. Change the Shape definition so that the ordering is based upon the perimeter of the shape rather than the area.

(OK, so for the two types of shape that we have defined there is actually no difference, but you could add further shape variants where it would make a difference!)

32. A binary tree is complete if the two sub-trees of every node are of equal size. Define a function that decides if a binary tree is complete.

33. Define a function balance :: [a] -> Tree a that converts a non-empty list into a balanced tree.

34. Redefine map f and filter p using foldr.

35. In the recordings on higher order functions, we looked at a new definition for mergesort. What function would you pass as an argument to this function to sort a list of (name, grade) pairs by grade order, from lowest to highest?
For example,

```
> mergesort my_function [("Sam", 46), ("Bob", 22), ("Alice", 65), ("George", 87),
("Jason", 77)]
[("Bob", 22), ("Sam", 46), ("Alice", 65), ("Jason", 77), ("George", 87)]
```

(assuming my_function is the function you have defined)

36. Implement your own version of maximum using one of the fold functions.

37. Rewrite the following expressions as list comprehensions:
    a. map (+3) xs
    b. filter (>7) xs
    c. concat (map (\x -> map (\y -> (x,y)) ys) xs)
    d. filter (>3) (map (\(x,y) -> x+y) xys)

38. What does this function do:

```
mystery xs = foldr (++) [] (map sing xs)
                    where
                    sing x = [x]
```

Hint: Work through what it does for a simple list, e.g. [1,2,3]

39. Without looking at the definitions from the standard prelude, define the higher-order library function curry that converts a function on pairs into a curried function, and, conversely, the function uncurry that converts a curried function with two arguments into a function on pairs.
Hint: start with the type definitions.
(The solution is simple, but reaching it might not be so simple.)

40. [Hutton, ex 7.10] Using altMap, define a function luhn :: [Int] -> Bool that implements the *Luhn algorithm* from the earlier exercise set for bank card numbers of any length. Test your new function using your own bank card.

41. **A more substantive problem to work on**:

A bag is a collection of items. Each item may occur one or more times in the bag. All the items are of the same type. The order of the items is unimportant. For instance,

- over a 40-game season a football team might score 5 goals in 2 games, 4 goals in 1 game, 3 goals in 4 games, 2 goals in 10 games, 1 goal in 12 games and 0 goals in 11 games;
- a student's assessment profile might comprise 3 A grades, 4 B grades and 2 C grades.

A bag may be implemented by a list (e.g. ['A', 'A', 'A', 'B', 'B', 'B', 'B', 'C, 'C'] for the student grades) but this is clearly inefficient. In this assignment you will implement and test a Haskell module Bags.hs to handle bags.

Bags.hs should contain

- A definition for a polymorphic Haskell datatype **Bag** to represent bags efficiently.
- The following functions

a) **listToBag** takes a list of items (such as ['A', 'A', 'A', 'B', 'B', 'B', 'B', 'C, 'C'] above) and returns a bag containing exactly those items; the number of occurrences of an item in the list and in the resulting bag is the same.
b) **bagEqual** takes two bags and returns True if the two bags are equal (i.e., contain the same items and the same number of occurrences of each) and False otherwise.
c) **bagInsert** takes an item and a bag and returns the bag with the item inserted; bag insertion either adds a single occurrence of a new item to the bag or increases the number of occurrences of an existing item by one.
d) **bagSum** takes two bags and returns their bag sum; the sum of bags X and Y is a bag which contains all items that occur in X and Y; the number of occurrences of an item is the sum of the number of occurrences in X and Y.
e) **bagIntersection** takes two bags and returns their bag intersection; the intersection of bags X and Y is a bag which contains all items that occur in both X and Y; the number of occurrences of an item in the intersection is the number in X or in Y, whichever is less.

Week 5

42. Write functions to return those properties of a BST:
    a) minimum
    b) maximum
    c) successor
    d) predecessor

43. Using '.', foldr, map and the identity function id, write a function pipeline which given a list of functions, each of type a -> a will form a *pipeline* function of type [a] -> [a]. In such a pipeline, each function in the original function list is applied in turn to each element of the input (assume the functions are applied from right to left in this case). You can imagine this as being like a conveyor belt system in a factory where goods are assembled in a fixed number of processing steps as they pass down a conveyor belt. Each process performs a part of the assembly and passes the (partially completed) goods on to the next process. Test your function by forming a pipeline from the function list [(+ 1), (* 2), pred] with the resulting pipeline being applied to the input list [1, 2, 3].
    Hint: Notice that if f :: a -> a then (map f) is a *function* of type[a] -> [a].

44. Write a function find that finds the first item in a list for which a predicate is True. If there is no such item, it should return Nothing. The type of your function should be:

```
find :: (a -> Bool) -> [a] -> Maybe a
```

45. In the lectures, we considered the maybeHalf function:

```
maybeHalf :: Int -> Maybe Int
    maybeHalf a
      | even a = Just (div a 2)
      | otherwise = Nothing
```

    Based upon this, define a function maybeEighth that will return one eight of the input value, if it is a value for which an integer eighth can be calculated.

46. Write a recursive function:

```
sumInts :: Integer -> IO Integer
```

    that repeatedly reads integer numbers from IO until the number 0 is given. At that point, the function should return the sum of all the entered numbers plus the original (default) value, which is given as a function parameter.

47. Write a function which prompts for a filename, reads the file with this name, splits the file into the lines contained within it, splits each line into the component words separated by ' ' (a space), and prints the total number of lines and words.

48. Implement the Sieve of Eratosthenes for computing prime numbers in Haskell. Recall that the sieve works as follows. Starting from the natural numbers from 2 and up, keep 2 as it is a prime, then strike out all multiples of 2 from the rest of the numbers. The smallest remaining number is also a prime, so keep it, then strike out all multiples of it form the the remaining numbers. And so on.

49. Define the infinite list fibonacci of Fibonacci numbers. Use the predefined function zipWith. It takes as input a function, a list, and another list, and gives back as output a list consisting of the function applied to both first elements, the function applied to both second elements, etcetera.

50. In an older version of the standard library, the function intersperse, which places an element between all elements of a list, was defined as:

```
intersperse _ []      = []
    intersperse _ [x]     = [x]
    intersperse e (x:y:ys)
      = x : e : intersperse e (y:ys)
```

Notice that the result of the expression

```
intersperse 'a' ('b' : undefined)
```

is an immediate error. Can you write a definition that would give at least the initial character in the output string? I should be able to get a sensible result from

```
take 1 (intersperse 'a' ('b' : undefined) )
```

but with the above definition I get an error.

What Now?

You've finished the provided exercises, but don't stop practising. The grading assignment will draw upon everything here and build your understanding even further.

This module is just an introduction to functional programming, and there are many aspects that are glossed over or not mentioned at all. If you are interested in learning more, you could use functional programming in your third-year dissertation.