ANSWERS

The Functional Programming Paradigm—with Haskell

A Level Computer Science (AQA 4.12.1)

Starter: A Python Investigation…

(i) Try to run the following snippets of Python code:

```
sum( “the” )
[1, 2] – [3, 4]
ord( 10 )
“300” > 200
```

Explain, in your own words, what is going wrong.

The inputs to the functions are of the wrong type. For example, “sum” expects a list of numbers but gets a string; “-“ expects numbers but gets lists; “ord” expects a character but gets a number, and “>” expects two values that are comparable but gets a string and an integer, which are not comparable.

(ii) Try the code snippets again, this time giving appropriate arguments or argument combinations to each operator: sum, -, ord, and >. Write down your code snippets and the results below:

```
sum( __[1, 2, 3]___ )
__7___ – __5___
ord( __”C”___ )
__100___ > ___200___
```

Explain, in your own words, how you chose appropriate arguments or argument combinations for each operator.

For “sum”, I chose a list of integers as these can be added together; for “-“, I chose two numbers as these can be subtracted in Python; for “ord”, I chose a character as the ASCII number of it can be found; for “>”, I chose two integers which can be compared according to order of size.

ANSWERS

Section 1: Function Types (part 1)

To run Haskell code snippets in an interpreter, you can:

- **create a repl on https://replit.com, making sure you select "Haskell" as the programming language (see Appendix), or**
- **install the Glasgow Haskell Compiler (GHC) on your device and start up the interactive environment with "ghci" in your command prompt.**

TASK 1

Enter the following commands into the Haskell interpreter and record the results. (Look up the action of any functions that you do not remember from the previous workbook.)

```
:t True
:t 42
:t 'A'
:t "A"
:t head
:t sqrt
:t fst
:t even
:t min
:t (+)
:t (&&)
:t filter
```

Question 1: Based on your results, describe the purpose of "`:t`" in your own words below:

":t" returns the type of the input given

KEY POINTS

- Recall that programming languages have data types such as String, Boolean, Integer, Float/Real, and Character/Char. For example, "Hello, world!" is a String, 'H' is a Char, 5 is an Integer, 5.0 is a Float and True is a Boolean.

- Recall that in (set-theoretic) mathematics, a function is a rule that, for each element in some set A, assigns a **unique** output value chosen from some set B (without necessarily using every member of B).

- **Key Point: In programming languages, functions can be assigned types. For function `f:A -> B` ,**
  "A" represents the type of the argument and "B" represents the type of the returned value.

ANSWERS

- Recall from mathematics that this notation also represents the notion that that A is the domain of the function (the set of inputs for which the function is defined) and B is the codomain of the function (the set from which the outputs are chosen). Therefore, there is a fundamental link between the concept of the domain and codomain of a function in mathematics and the type of a function in programming.

- Example 1: in Python, the function "`ord`" can be assigned type `Char -> Int` , because it takes a character, such as 'A', as input, and returns an integer corresponding to the Unicode decimal value for that character, such as 65.

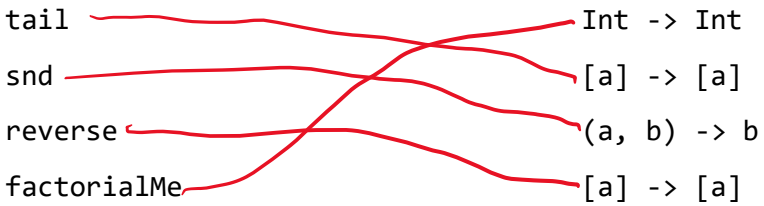
- Example 2: in Haskell, the function "`even`" has type signature `even :: Integral a => a -> Bool`. The type of the function, `a -> Bool`, with a single-line arrow, indicates that "`even`" takes an input of some type "a" and returns a Boolean value. The "Integral a =>" part, with the double-line arrow, specifies that "a" is any data type with "Integral" properties, such as an Int or an Integer.

- Example 3: in Haskell, the function "`head`" has type `[a] -> a`. This indicates that its input is a list of elements that each have the same type, "a", and that the output of head is a single value, also of type "a". But "a" itself is not specified; it could be Int, Char, etc. Using the **type variable** "a" in this way shows that "head" can get the first element of a list of *anything*. This is called parametric polymorphism.

- NB. In Haskell, when you define a function, you can specify its type in the type signature, using "::" (see Task 3 Extension). However, you do not have to do this, because Haskell's type system can perform **type inference** and infer a suitable type for most functions.

TASK 2

Match the following Haskell functions/operators with their possible type. If you cannot remember what these operators do, look them up. (NB. The function factorialMe returns the factorial of a number, i.e., `factorialMe( n )` = n!).



When you have finished, check your answers for `tail`, `snd` and `reverse` using, e.g., "`:t tail`" in your Haskell interpreter. Functions can be correctly assigned more than one type (the interpreter will often give the most general type possible), and the types provided here are simplified, so ask for help if you are not sure.

Extension: Write three different functions that can have a type of Int -> Int. Hint: In your Haskell file, write the required type signature above the function definition then try to load the file (see Appendix) to check whether your function can work with that type signature. Use the example below to guide you. ANSWER CODE PROVIDED.

```
doubleMe :: Int -> Int
doubleMe x = x * 2
```

ANSWERS

Section 2: Function Types (part 2)

STARTER

(i) **Look carefully at the function "halveMe" defined below, without running it. Why is the given type signature wrong for this function? Illustrate your answer with an example.**

```
halveMe :: Int -> Int -- This is the wrong type signature
halveMe x = x / 2
```

The type signature implies that the returned value will always be an integer, which is not true. For example, halveMe 7 would return 3.5, which is a floating-point data type, not an integer.

(ii) **Copy the above code into a Haksell file and load the file (see Appendix). Observe the error message produced by the type system and copy it down below. Then comment out the incorrect type signature (with --) and load the file again without a type signature. Enter the command ":t halveMe" to find out what type Haskell has inferred for this function. Explain your results below.**

If the file is loaded with the incorrect type signature Int -> Int, a type error is obtained: "No instance for (Fractional Int) arising from the use of '/' in the expression: x / 2.

This shows that Haskell has recognised that a fractional value might arise from division by 2, and this conflicts with the Int (whole number) type assigned to the return value for halveMe.

When Haskell is allowed to infer a type, the type is halveMe :: Fractional a => a -> a, which allows both the argument and returned value of halveMe to be fractional.

KEY POINTS

- What about types for functions that take more than one argument? Last lesson, you may have noticed that functions min, (+), (&&), and `filter` have multiple single-line "->" arrows in their type. The type (or type variable) after the final arrow represents the type of the value returned by the function. The other types (or type variables) represent the types of the input arguments.

- Example 1: the OR operator (||) requires two arguments, which are both Boolean values, and it returns a Boolean value. Accordingly, its type is Bool -> Bool -> Bool. For example,

`>> False || False` (with arguments False and False, this returns the Boolean value False)

`>> True || False` (with arguments True and False, this returns the Boolean value True)

ANSWERS

- Example 2: the function “take” has type Int -> [a] -> [a]. This shows it has two input arguments, an Int and a list [a] of a’s, and it returns a list [a] of a’s. Indeed, the function “take” can been called like so:

```
>> take 5 [1, 4..100]
```

That is, with arguments 5 and `[1, 4..100]` , it will return `[1, 4, 7, 10, 13]`, the first five elements of the list argument.

- The function “take” can also be called like so:

```
>> take 5 [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’]
```

which will return `[‘a’, ‘b’, ‘c’, ‘d’, ‘e’]`, the first five elements of the list argument. Note that in both cases, the types of the elements inside the lists (Int or Char in these examples) don’t matter, but we must ensure that the first argument to `take` is an integer (5 in this case). Furthermore, we are guaranteed to get out the same type of list out as we put in.

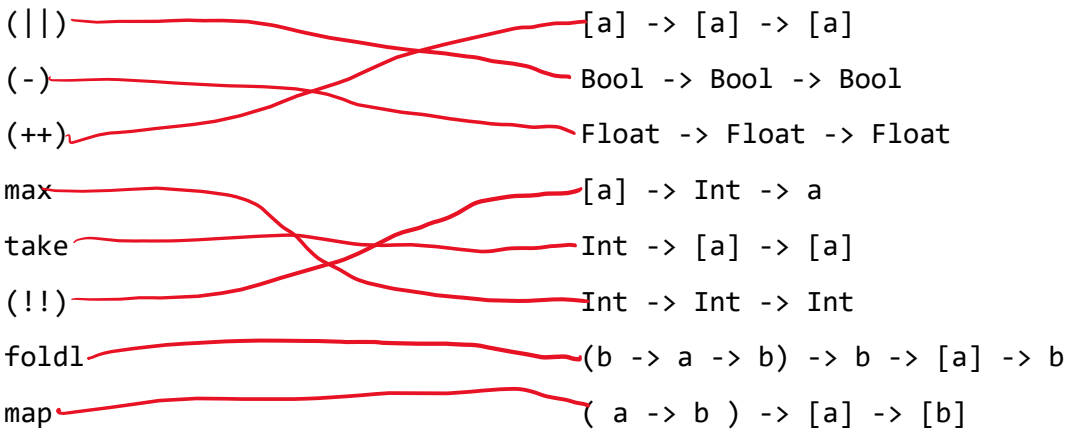
- What about functions that take functions as inputs? This is a focus of Section 3, but for now, consider the type of the function “`filter`”, which is (a -> Bool) -> [a] -> [a]. This indicates that “`filter`” takes two inputs, a function with type a -> Bool , and a list [a], and returns a list [a]. Indeed, the function “`filter`” can be called like so:

  ```
  filter even [0..16]
  ```

  which will return the list [0, 2, 4, 6, 8, 10, 12, 14, 16]. As another example, the function “even” has type Integral a => a -> Bool (with the extra proviso that type a must have “Integral” characteristics).

TASK 1

Match the following functions/operators with their appropriate type. Hint: try analysing the types first. For example, [a] -> Int -> a means you are looking for a function that takes two arguments, a list [a] and an integer, and returns a single element of type “a” . Which one could it be? POSSIBLE ANS:



ANSWERS

When you have finished, check your answers using ":t" in your Haskell interpreter. Functions can be correctly assigned with more than one type, and the types provided in the exercise are simplified, so ask if you are not sure.

TASK 2

For each type below, create a function that could be given that type. Check your answers by writing the type into the type signature above the definition of your function and seeing whether it is accepted by the Haskell type system (as in the Section 1 Extension example). Be original and creative! Insert screenshots of your code and the result of your functions running on example inputs.

```
(i)      Float -> Float
(ii)     Float -> Float -> Float
(iii)    [Char] -> Char -> [Bool]
(iv)     Int -> Int -> Int -> [Int]
(v)      a -> a -> (a -> b) -> (b, b)
```

Example answer code provided.

ANSWERS

Section 3: Functions as first-class citizens

STARTER

Match each phrase from the list with an appropriate code snippet example:

- Function in an expression (can be Example 3; function in Boolean expression)
- Function assigned to a variable (can be Example 1)
- Function assigned as argument (Example 2; square is argument to map)
- Function returned in a function call (Example 4; a function is returned using lambda)

```
-- Example 1
enthusiastically f x = f (f (f x) )
very = enthusiastically


-- Example 2
square x = x * x
newList = map square [1..100]


-- Example 3
isAllowed n
  | even n || n > 35 = True
  | otherwise = False


-- Example 4
foo n = \x -> n**x
twoToPowerOf = foo 2
threeToPowerOf = foo 3

-- What is the result of twoToPowerOf 10?
-- What is the result of threeToPowerOf 3?
```

KEY POINTS

- In functional programming languages, such as Haskell, Common Lisp, Scheme, OCaml, and F#, functions are first-class citizens (referred to as **first-class objects** in the AQA syllabus). This means functions can **appear in expressions**, **be assigned to variables**, **be assigned as arguments**, and **be returned in function calls**.

- Most modern imperative programming languages, such as Python, also support functions as first-class objects.

ANSWERS

- Arguably, the languages C and C++ do not support functions as first-class objects https://stackoverflow.com/questions/10777333/functions-are-first-class-values-what-does-this-exactly-mean/10782920#10782920

TASK 1

Give three of your own examples of functions used as arguments to other functions in Haskell. Insert screenshots of your code and the results of running it.

Answer code provided.

TASK 2

Load this code snippet below into your Haskell interpreter then answer the questions below:

```
-- Define a function that cubes its input:
cube x = x * x * x

-- Define a polynomial function:
poly x = 5 * x * x  - 3 * x + 1


deriv f = \x -> (f (x + dx) - f x) / dx
  where dx = 0.01


derivCube = deriv cube
derivPoly = deriv poly
```

Question 1: What is the result of running >> `derivCube 4`?

```
=> 48.120099999998445
```

Question 2: What is the result of running >> `derivPoly 8`?

```
=> 77.049999999997
```

Question 3: Explain, in your own words, what the function "`deriv`" does and how it does it, using the answers to Questions 1 & 2 as examples.

ANSWERS

The function deriv takes a function f as an argument and returns a function that will calculate the approximate derivative of the original function at a given input argument x. Specifically, deriv will calculate the gradient between the value of the original function f at x and at a nearby value, x + 0.01. This gradient an approximation to the derivative of f at x. Hence for derivCube 4, the value $(4.01^3 – 4^3) / 0.01$ was calculated, and for derivPoly, the value (f(8.01) – f(8))/0.01 was calculated, where f is the polynomial function $5x^2 – 3x + 1$.

The resulting values are approximations to the exact derivatives, which are

$3(4)^2 = 3 \times 16 = 48$, and

$10(8) – 3 = 80 – 3 = 77$, respectively.

Question 4: An engineer using the function "deriv" decides that it is not accurate enough. Explain what you can change in the definition of the function to make it more accurate.

Make the increment dx smaller, e.g. 0.0001 or 0.00001. This will reduce the distance over which the gradient of the function f at input x is approximated by deriv and hence is more likely to be accurate.

ANSWERS

Section 4: Function application and partial application

STARTER

(a) Run the following Haskell code snippets in the interpreter and record the results next to each line.

```
map (+3) [1..20]

map (&& True) [True, False, True, False]

map (<=4) [-6, -3..20]

map (min 5) [1..10]
```

(b) Explain, in your own words, the meaning of the first argument to map in each case above:

(+3) Add 3 to a vlue

(&& True) Perform logical AND using a value and TRUE

(<=4) Check whether a value is smaller than or equal to 4

(min 5) Check the minimum between the number 5 and a given value

NOTES – FUNCTION APPLICATION

- **Function application** means to apply a function to its argument(s).
- When a function is applied to all of its arguments, it will return an output of the type specified by its type signature.
- **Example 1:** In Haskell, you can find the maximum of two numbers using `max a b`, which will return the largest of a and b, and will return a if they are equal. For example:

```
max 5 10
=> 10
```

- In Haskell, the type signature for the function max is `max :: Ord a => a -> a -> a`, meaning it takes two arguments, each of type a, and returns a value, also of type a, with the only restriction that "a" is a type that can be ordered (such as an Int, Char, etc.). **In this example, max can be thought of as taking two arguments, 5 and 10, and returning the value 10.**
- But more generally in computing and mathematics, the "max" function can also be thought of as having the type a x a -> a, that is, a function that takes in a single argument that is an

element of the set a x a, with a x a being the Cartesian product of the set "a" with itself. In mathematics, this would be the set-theoretic way of describing the function in terms of a single domain and codomain. Under this type, **the function max from Example 1 above can be thought of taking just one argument, the coordinate (5,10) from the set RxR or R^2, where R represents the set of real numbers, and returning the value 10.** For A Level Computer Science, you need to be able to understand and use both formats of function type highlighted in yellow above.

- Example 2: The type of the AND function (&&) can be viewed as either (&&) :: Bool -> Bool -> Bool or as Bool x Bool -> Bool.

- Example 3: The type of function `foo x y z = [x * 2, y + 5, z - 100]` can be viewed as either Float -> Float -> Float -> [Float] or Float x Float x Float -> [Float]. The domain of this function can be viewed as a three-dimensional Cartesian product, because the function accepts three arguments.

TASK 1

Run each of the following code snippets below in the interpreter and record the result. What pattern do you find? Explain your findings in detail.

```
:t (+)
(+) :: Num a => a -> a -> a
```

```
:t (+3)
(+3) :: Num a => a -> a
```

```
:t (&&)
(&&) :: Bool -> Bool -> Bool
```

```
:t (&& True)
(&& True) :: Bool -> Bool
```

```
:t (<=)
(<=) :: Ord a => a -> a -> Bool
```

```
:t (<=4)
(<=4) :: (Ord a, Num a) => a -> Bool
```

There is one less argument in the function type signature after an argument is passed to the function.

For (<=4), there is also the additional constriction that type variable a has to have Num properties.

KEY POINTS – PARTIAL APPLICATION

ANSWERS

- **Partial application** is when a function is applied to fewer than its full number of arguments. For example, the functions +, &&, <=, and min in the STARTER all accept two arguments but received only one before they were mapped to the list.

- **Partial application of a function returns a new function that takes fewer arguments than the original function.**

- Example 4: imagine "`addInt`" is a function that adds two integers together. In Haskell, its type signature is addInt :: Int -> Int -> Int , that is, it accepts two Int inputs.

- Technically, the type signature of "addInt" should be addInt :: Int -> (Int -> Int). This clarifies that after partial application of the function "addInt" to the first Int argument, one obtains a new function that takes one Int argument and returns an Int output. E.g., "`addInt 5`" is a new function that takes a single input argument, adds 5 to it, and returns the resulting value. Its type signature would be Int -> Int.

- In practice, we skip these brackets out of the type signature because they would cause clutter for complicated functions. We only need to include brackets when there is ambiguity, e.g., for map :: (a -> b) -> [a] -> [b], where we need the brackets to show that the first argument is itself a function of type (a -> b) that takes an argument of type a and returns an argument of type b, rather than two separate arguments of types a and b.

- **Do not confuse partial application with the concept of *currying*.** *Currying* is the technique of converting a function whose domain that is a Cartesian product of sets (like f :: Int x Int -> Int) into a sequence of functions that each take a single argument (like f :: Int -> Int -> Int). As you might have noticed from their type signatures, in Haskell all functions are considered curried and this form is more convenient as it allows partial application; however, conversion in either direction is possible. Like the programming language itself, "currying" is also named after mathematician and logician **Haskell Curry.**

TASK 2

Create the functions specified below, **using partial application**. Insert a screenshot of your definition of the function and the result of applying the function to two appropriate, contrasting arguments. The first one is done as an example to guide you:

Example answer code provided.

(i) Using (<=), create a function "foo" that takes a single input and returns the Boolean result of whether that input is at most 10.

```
ghci> foo = (<=10)
ghci> foo 11
=> False
ghci> foo 9
=> True
```

ANSWERS

(ii) Using (>), create a function “greaterThan100” that takes a single input and returns the Boolean result of whether that input is greater than 100.

(iii) Using (+), create a function “plus100” takes a single input and returns that input plus 100.

(iv) Using “max”, create a function “rectify” that takes a single input and returns that input if it is non-negative; otherwise, it returns zero.

(v) Using “map” and “square x = x * x”, create a function “squareList” that takes a list of numbers and returns the list of those same numbers squared.

(vi) Using “foldl” and two appropriate inputs, create a function “sumList” that will sum the numbers in the input list and return an integer.

(vii) Using “filter”, create a function “filterPos” which returns a list that contains only the positive numbers in the input list.

ANSWERS

Section 5: Function composition

STARTER

Three of the following function compositions will produce an error! Without running them in Haskell, work out which three, giving a reason for each answer.

```
even ( square x )
```

```
even ( sqrt x ) Error: sqrt(x) might not be an integer so cannot be checked for even-ness by "even".
```

```
not . square $ x Error: "square" returns a number, but "not" expects a Boolean.
```

```
not . even $ x
```

```
head ( tail ( tail [1..10] ))
```

```
tail . head . tail $ [1..10] Error: tail expects a list but the each element in [1..10] is a number not a list, so head will return a number.
```

KEY POINTS

- Function composition is itself an operation (i.e., a higher order function) which takes two functions and returns a new function, the result of applying one after the other.
- For example, in mathematics, if $f(x) = x^2$ and $g(x) = x + 1$, the composition $g \circ f$ is equal to applying f then g , i.e., "g after f". This means $(g \circ f)(x) = g(f(x)) = x^2 + 1$.
- You can think of $(g \circ f)(x)$ as *f* applied to *x*, and then function *g* applied to the result returned by *f*.
- On the other hand, the composition $(f \circ g)(x) = f\big(g(x)\big) = (x+1)^2 = x^2 + 2x + 1$.
- So, in general, the order of functions in function composition matters.
- What about function composition in programming? It works in the same way, but we must pay attention to types. If f: A -> B, and g: B -> C, then composition of "g after f" will be

  $g \circ f$: A -> C,

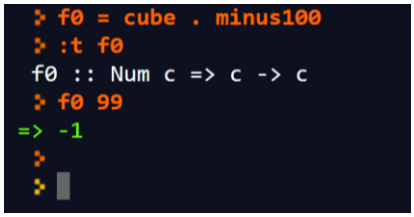
  because the input to the composition will be the same as the input type for *f* and the output of the composition will be the same as the output type for *g*. We no longer need to care about the intermediate type B.
- However, if f: A -> B, and g: B -> C, the function composition "f after g" $f \circ g$: B -> B will in general not be possible, unless type C is a subtype of type A, that is, function *f* can accept inputs of type C.

ANSWERS

TASK 1 – Creative composition

Using the functions specified in Haskell below, define three different new functions using function composition – you can compose more than two functions if you wish! For each of your answers, provide a screenshot that shows the new function's type signature (using *:t*) and the result of running it on an appropriate argument. An example screenshot for a function "f0" is provided as a guide.

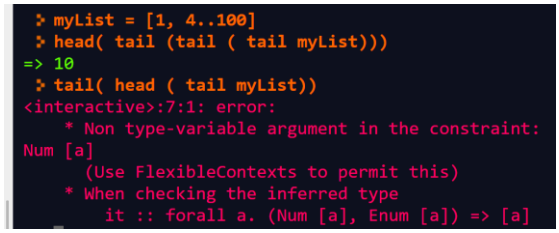
```
cube x = x * x
div10 x = x/10
minus100 x = x - 100
odd
not
sqrt
```



Students' individual answers should be checked.

TASK 2 – Heads or Tails

Look through this code and its output carefully, then answer the questions below:



Question 1: Write down the first five elements of "myList":

[1, 4, 7, 10, 13]

Question 2: Explain fully why the value returned in the second line of code is 10.

The function "tail" is applied three times to myList = [1, 4..100]. This means the tail of the list, that is, the list except the first element, is taken three times, so the first the elements 1, 4, 7, of the list are removed. Then, the head of the resultant list, head [10, 13, … 100] is taken. The head of this list is 10.

ANSWERS

Question 3: Explain fully why a type error is produced by the third line of code.

The third line of code takes the head of the tail of the list, and then tries to take the tail of that. The head of the tail of the list is the head of [4, 7, … 100], which is 4. It is not possible to take the tail of the number 4, because it is only possible to take the tail of a list, and the number 4 is not a list.

TASK 3 – Extension

We have seen previous examples where the order of function composition matters. This means function composition is not **commutative** in general. However, is it **associative**? That is, when composing any three functions *f, g, h*, does $h \circ (g \circ f) = (h \circ g) \circ f$? Use examples to illustrate your answer and provide a proof if you can, doing extra research if necessary.

See e.g. https://math.stackexchange.com/questions/523906/show-that-function-compositions-are-associative

APPENDIX: How to run Haskell on replit.com

How to run Haskell with Nix on www.replit.com/ for the purposes of the exercises in this workbook.

This is to account for the changes replit.com have made to the Haskell environment in Spring/Summer 2021, to embed Haskell within the Nix package manager (see https://blog.replit.com/nix).

GHCi is the Glasgow Haskell Compiler's interactive environment. It is useful for learning basic Haskell and debugging code. For more information about GHCi, with examples, follow the link below:

https://downloads.haskell.org/ghc/latest/docs/html/users_guide/ghci.html

INSTRUCTIONS FOR USING GHCI IN REPLIT.COM

1. Create a new repl, selecting Haskell as the language

2. There will be a default Nix folder structure created for you. The editor will show the file "src/Main.hs", which contains instructions for the Main module to print "hello world" when loaded.

3. Select the Shell tab on the right-hand side and navigate to the src folder (using "cd src"). This is so that you are in the correct directory to load the Main.hs file later.

4. Into the Shell command prompt, type "ghci" to start up the Glasgow Haskell Compiler interactive environment. This will automatically load up the "Prelude" module, which contains standard Haskell definitions.

5. The Shell prompt should change to "Prelude> ". Test GHCi by typing e.g.:

`reverse "abcd"` --which should reverse the string

`:t reverse` --which should return the type of reverse, i.e., [a] -> [a]

`map (+3) [1, 2 .. 10]` -- which should add 3 to all the numbers in the list 1-10

If this works it means you now have a working Haskell interactive environment.

6. To define your own functions, type them into line 6 onwards in the Main.hs file.

E.g., the full Main.hs file could read:

```
module Main where

main :: IO ()

main = do

  putStrLn "hello world"

-- My new function is below:

factorial :: (Integral a) => a -> a

factorial n

  | n == 0 = 1

  | otherwise = n * factorial (n - 1)
```

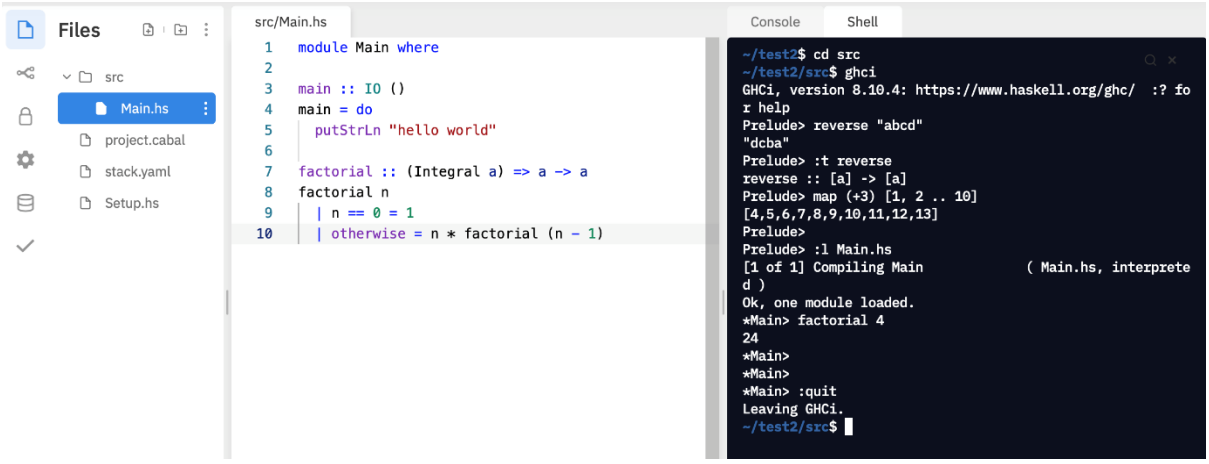
7. To load the newly defined factorial function into the interpreter, type "`:l Main.hs`" into the Shell prompt to load the file. The interpreter should respond with something like:

```
Prelude> :l Main.hs
[1 of 1] Compiling Main             ( Main.hs, interpreted )
Ok, one module loaded.
```

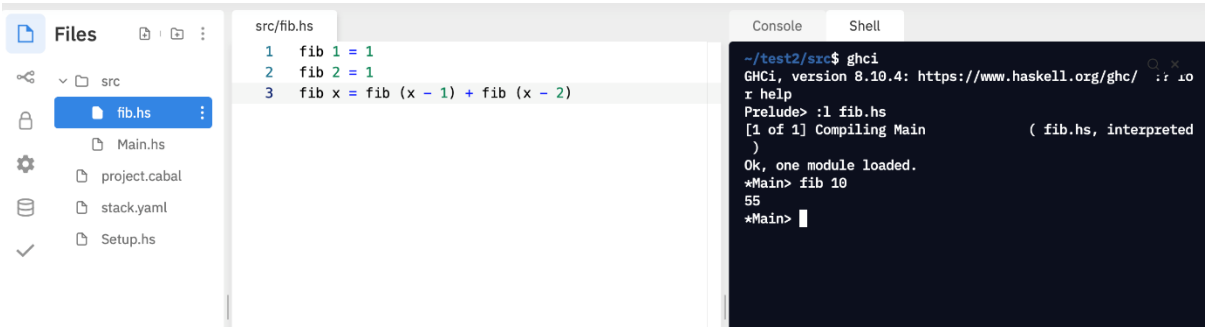
Then you can call your newly defined function(s) in the interpreter, e.g.:

`factorial 4` -- Should return 24.

See image below for guidance:



8. Whenever you made a change to the code inside Main.hs, you will need to load it into GHCi again using ":l Main.hs". You can create multiple *.hs files, with different names, to organise your code. You will need to load each one into GHCi using ":l <filename>.hs". See fib.hs example below:



9. To use the code files provided in this resource, you can upload them to the src directory in your repl, or copy/paste the text into a .hs file. Make sure to rename the files appropriately and delete the definition of "main" in the example code (as it is not necessary if you already have a main function defined in your Main.hs with Nix).

10. To quit GHCi, type ":quit".