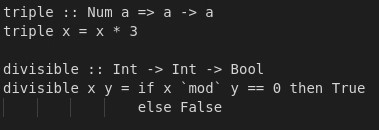
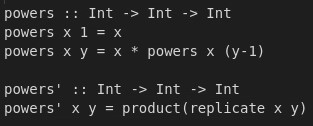
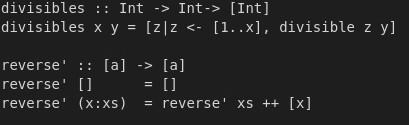
**Haskell/Functional: Mechanisms for Defining Functions in Haskell**

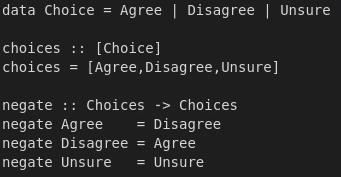
Like many other languages, there is a clear syntax you must follow to define functions. The difference compared to other languages is that it is much clearer and easier to read than object oriented as functions tend to be more concise and smaller, therefore making it easier for programmer to understand the code.

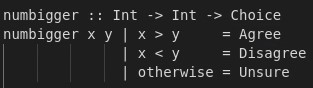
Firstly, we will look at the syntax. Function names always start from a lower-case letter and usually variable names are one letter like mathematical functions. Another variable convention is the use of “xs” for variables that are lists. We can see that the function type is defined above. For the triple, we can read that when given a number it triples it. The type of the function is a Numerical type meaning it can take and return both integer and float values, this

function is called a polymorphic function, meaning it can take any variable types. However, this is an example of an overloaded function where there is a class constraint, in this case numerical type. For the second example, we can see a function that takes more than one variable, this is called a curried function and works like a nested function. The type for this function is taking 2 integers and returning a Boolean value. We can also see that we can use if statements in Haskell like other languages, this is called conditional expressions.

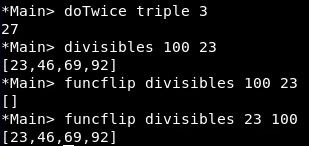
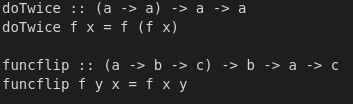
Having reviewed the basic syntax, we can explore list comprehension and the use of recursion. The first function, we can see it is a curried function. It generates a list of numbers 1 to x (first integer inputted), then there is a guard where it uses the previous function and when it does return true, generate that

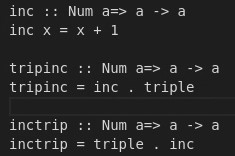


number, otherwise skip it. This then outputs a list of integers from 1-x, divisible by y using the previous helper function. The second function uses recursion. Haskell is a pure functional language so there are no for loops as the data is immutable. We can see that there is a base case where the function returns “x” so it terminates. And there is a second recursive case, which calls upon itself decrementing “y” until at the end taking the product using “\*”. The function itself raises the first integer to the second, e.g. “powers 2 2 => 4”. However, though you can use explicit recursion, Haskell has built in functions where you can replicate a given integers and product it all up as shown above using product and replicate functions. An extra feature with Haskell is that you can define your own data types. From the example to the left, you can see that I have defined a new data type called choices, where it is “Agree”, “Disagree”, “Unsure”. You are then able to use this with function with the right function types. There is a function defined as negate which flips the data around. This is a new way of defining functions and is called pattern matching. So,

you apply the function to parameters and has case for all different value and mapped to another new value. The final way of defining functions is guarded functions, this case we define it using our previous new data Choice and this function will have a choice if x is bigger than y.

There is another type of function which is a higher order function. This is a function which takes in another function as a parameter. One good example of the filter function which is defined in the Haskell standard prelude. This function essentially produces a new list from the old list where the function/predicate is valid. So, from the example, it outputs all the odd numbers from 1-12. We can do more with this like the first function, it applies that certain twice to the parameter making it triple twice. A more interesting example is the funcflip which just flips the parameters around when applying the function. From the output below we can see that it really did flip the function. It gave the same output when the parameters were flipped from the previous output.

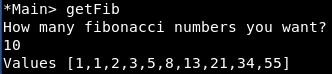
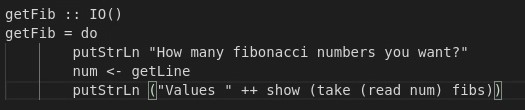


All the function examples have been what we call pure. With this, there is no side effects to using any of these functions, so you can use two functions together where it can work properly. You can also compose two functions together and use “.” to use composition. Show in the example to the right, we can fuse functions together. This is very similar to math functions where you can apply two functions together “f(g(x))” and the function further

right is applied first. As well as composing functions, we can use lazy evaluation and infinite lists. This

fibs function uses lazy evaluation by generating only as it is asked. So this is a helper function which generates all values of the sequence and you can get any size list by just using “take x . fibs”. zipWith is a function where it takes a function and a lists and applies between

the list e.g zipWith (+) [1,2] [1,2] = [2,4], the laziness of Haskell lets us do this as numbers are only created when asked. Finally, there is a feature



for when you want user inputs. This can cause issues as Haskell is a pure language, and with IO, you are now impure, so you will have to have a bridge to the impure world. To do this you are need to define the type IO() in your function. With this you can take inputs from the user. putStrLn function allows the print to screen and num is a string input from the user. Read converts the string to an integer value and show converts the list to a string allowing putStrLn to output.

This concludes on ways you can define functions in Haskell, we have seen many ways using guarded equations, conditional expressions etc. All these functions are concise, clear, and easy to read which is what Haskell aims to do.