Overview

Haskell is a statically typed language, so all variables have types at compile time. However, it uses type inference to allow programs to skip specifying the specific types a function takes (it infers this based on how you use the function).

This means a function definition will typically look like the following, especially for what we are using it for in this course.

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem y (x:xs)
  | y == x = True
  | otherwise = elem y xs
```

Going through line by line:

```
elem :: Eq a => a -> [a] -> Bool
```

The first line is the function signature. It has several parts: - `elem`: the name of the function - `::`: indicates this is a function signature - `Eq a =>`: type qualifier. This is any item behind the `=>` (if it exists). This one says that the `a` type implements equality: you can check if elements of type `a` are equal to each other. - `a -> [a] -> Bool`: the function arguments and return value. This one takes an argument of type `a`, a list of elements of type `a`, and returns a boolean value (true or false).

Note that the names used for the types have no relation to the names used for the parameters.

```
elem _ [] = False
```

The second line is one implementation of the function. Haskell allows multiple implementations, and they are resolved top to bottom by *pattern matching* the arguments against the definition. In this case, the first implementation takes anything as the first argument (`_` stands for an argument that can be anything but is unused), but only accepts an empty list as the second argument (`[]`). If the arguments are of this form, the function returns `False`.

> Pattern matching is a very important part of Haskell, and is used extensively in this course too. Questions will almost always have multiple implementations of a function, and you must determine which is appropriate to use based on your conditions on the values you are working with.

```
elem y (x:xs)
```

The third line is a second implementation. This will be reached if the second argument is not empty. In the parameters, we name the first parameter `y`, and decompose the second parameter into the first element `x` and the rest of the

elements `xs` (it is common to give list names an `s` suffix). So `x` is a variable of type `a`, and `xs` is a variable of type `[a]`.

> The `:` operator is call *cons*. Used as an operator, it is a function with the signature `a -> [a] -> [a]` that takes an element of type `a` and prepends it to a list of elements of type `a`, returning the new list. Used in pattern matching, it decomposes a non-empty list into the first element and a list of the rest of them.

Note the body of this implementation is defined over the next couple of lines.

```
  | y == x = True
  | otherwise = elem y xs
```

The vertical lines below this implementation are *guards*. They are of the form `<condition> = <result>`. Evaluated from top to bottom, if a condition is true, the result for that condition is returned. In this case, if the first parameter (`y`) is equal to the first element of the second parameter (`x`), then it returns `True`. `otherwise` is a catch-all condition that always succeeds. In this case, it means that if `y != x` then it recursively calls `elem` with `y` and the tail of the list in the second parameter (`xs`).

That's about as complicated as it gets in this course in terms of function definitions. The only other feature used in the tutorial 4 sheet is parentheses around the function name, which indicates it can be used as an *infix* function. This means it can appear between its two arguments, instead of to the left. So `a ++ b` is equivalent to `(++) a b`.

Examples

```
foo :: a -> b
foo n = 4

foo 'a' -- 4
foo 5   -- 4
```

A function `foo` takes an argument `n` and returns `4`.

```
foo :: a -> a -> a
foo a b = a + b

foo 4 5  -- 9
foo 8 -3 -- 5
```

A function `foo` takes arguments `a` and `b` and returns `a + b`

```
start :: Num a => [a] -> a
start [] = 0
start x:xs = x
```

```haskell
start [1, 2, 3]  -- 1
start []         -- 0
start [1.3, 2.7] -- 1.3
```

A function `start` that takes a list of numbers, and returns the first one in the list (or 0 if the list is empty).

```haskell
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys                  -- A1
(x:xs) ++ ys = x : (xs ++ ys) -- A2

[1, 2, 3] ++ [4, 5, 6] -- [1, 2, 3, 4, 5, 6]
```

An infix function `++` that concatenates two lists.

Currying

There is a feature in Haskell where you can provide some of the arguments to a function, and it will return a function that takes the remaining arguments.

E.g.,

```haskell
add :: a -> a -> a
add a b = a + b

add2 :: a -> a
add2 = add 2

add 4 5 -- 9
add2 3  -- 5
```

Here we defined a function of two arguments `add`, and passed only one to define a new function `add2`. This derived function takes a single argument, and adds it with `2`.

I don't know if this is used in 1600, but it's good to keep in mind. Note how this works cleanly with the type signatures; passing the first argument resolves `a -> a -> a` to a function `a -> a`, and the second argument resolves `a -> a` to just a value `a`.

Proofs

In proofs, function definitions go both ways. For example, if you are using `++`, then using definition `A1` you can go from `[] ++ ys` to `ys`, but you can also do the reverse and go from `ys` to `[] ++ ys`.

Further reading

- COMP1100 labs: https://cs.anu.edu.au/courses/comp1100/labs/
- Learn You a Haskell for Great Good!: http://learnyouahaskell.com/chapters