1 Haskell: An Introduction

1.1 Lists

1.1.1 The Type of a List

A list has type `[A]` if each one of its elements has type `A`. Thus, Haskell lists are homogeneous – you cannot put elements of different types in the same list. For example,

- The following is a list of `Int` elements.

  ```
  myList :: [Int]
  myList = [1, 2, 3, 4]
  ```

- The following is a list of `Char` elements.

  ```
  myList :: [Char]
  myList = ['h', 'e', 'l', 'l', 'o']
  ```

  As a side note, a list of `Char`s is the same thing as a `String`; that is, the following will work:

  ```
  myList :: String
  myList = ['h', 'e', 'l', 'l', 'o']
  ```

  This is the same thing as (**A**):

  ```
  myList :: String
  myList = "hello"
  ```

  Note that `[Char]` being the same thing as `String` is an example of **type synonyms**. We can actually define our own type synonyms[1] like so:

  ```
  type Apple = [Char]
  ```

  And then we can use it like so (**B**):

  ```
  myList :: Apple
  myList = ['h', 'e', 'l', 'l', 'o']
  ```

  Remark: If you define a type synonym for some variable x and then ask GHCi what type x is, GHCi will give you the type synonym. However, if you don't have a type synonym, then GHCi will give you the underlying type. So, for example, in (A), GHCi would say that `myList` is a `[Char]`; for (B), GHCi would say that `myList` is an `Apple`.

- The following is not allowed in Haskell.

  ```
  myList = [1, 'h']
  ```

  Here, `myList` has two elements of different types.

- You need to specify a type, but that type can be anything.

[1]Note that all type names must start with a captial letter.

```
myList :: [Char]
-- or
myList :: [[Char]]
-- or
myList :: [Int]
-- ...
myList = []
```

What would the *best* type for this be? We can use polymorphism (also known as parametric polymorphism). So, we would define this like so:

```
myList :: [a]
myList = []
```

Here, `a` is the type variable[2]. So, literally, it's a variable that can stand for a type. *In actuality*, we don't need to specify a type at all; that is, we can just have

```
myList = []
```

GHCi will infer that `myList` is of type `[a]`.

1.1.2 Functions on List

We will now implement some basic functions. One thing to keep in mind is that functions on lists in Haskell will most likely be recursive. Generally, you will have

- A base case: for a function that produces a list, the base case will most likely be the empty list.
- The recursive case: for a function that produces a list, this will most likely produce a non-empty list by *cons*ing an element to a list.

Now, let's implement some functions (without using library functions).

- **Range:** Given a lower l and upper bound u, this returns a list of integers between l and u. For example, `upto 1 3` returns `[1, 2, 3]`.

  ```
  upto :: Int -> Int -> [Int]
  upto lo up
      | lo > up       = []
      | otherwise     = lo : upto (lo + 1) up
  ```

  To see how this works, consider a simple example `upto 1 3`; roughly speaking, the steps taken to produce the final list might look like:

  ```
  upto 1 3
  => 1 : upto 2 3
  => 1 : (2 : upto 3 3)
  => 1 : (2 : (3 : upto 4 3))
  => 1 : (2 : (3 : ([])))
  ```

  Remark: Because this function is often used, this function is actually built-in. Indeed, you can do something like `[1..10]` to get `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. You can also do something like `[1,3..10]` to get `[1, 3, 5, 7, 9]`.

[2]We know that it's a type variable because it's lowercase.

- **Length:** Given a list ℓ, find the length of ℓ, i.e. the number of elements in ℓ.

  The idea is that:

  - If the list is empty, then return `0`.
  - Otherwise, return the length of the tail plus 1.

  So, implementing this idea, we have:

  ```
  length :: [a] -> Int
  length []       = 0
  length (_ : xs) = 1 + length xs
  ```

  Here, we can destruct the list into two components; the head and the tail.

  With this in mind, we can redefine what a pattern is. In particular, a pattern is either:

  - A variable, including _.
  - *Or*, a constructor applied to other *patterns*.

  Pattern matching attempts to match values against patterns and, if desired, binds variables to successful matches.

  (Quiz.) Which of the following is **not** a pattern?

  (a) `(1:xs)`
  (b) `(_:_:_)`
  (c) `[x]`
  (d) `[1+2,x,y]`
  (e) All of the above.

  The answer is **D**.

  - **A** matches any list that start with 1.
  - **B** matches any list with at least 2 elements. We can rewrite B as `x:(y:z)`.
  - **C** matches a list with one element. We can rewrite C as `x:[]`.
  - **D** is invalid. Remember that you can only use a constructors and variables in a pattern. `+` is not a constructor; it is a function that computes something. Now, if you wrote `[3,x,y]`, then this would match a list with exactly 3 elements where the first element is 3.

1.1.3 Library Functions

These are some useful library functions for lists just to[3] "tell you that you aren't allowed to use them on the homework."

- Checking if List is Empty: You can use the function `null` to see if a function is empty.

  ```
  null :: [t] -> Bool
  ```

- Get Head of List: You can use the function `head` to get the head of a list.

[3] As the professor said.

```
head :: [t] -> t
```

Warning: This is a partial function; they can produce a runtime error on some inputs. In this case, the empty list `[]` will cause said runtime error.

- Get Tail of List: You can use the function `tail` to get the tail of a list.

  ```
  tail :: [t] -> [t]
  ```

  Warning: This is a partial function; they can produce a runtime error on some inputs. In this case, the empty list `[]` will cause said runtime error.

- Get Length of List: You can use the function `length` to get the length of a list.

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

- Append Two Lists: You can use the function `(++)` to append two lists.

  ```
  (++) :: [t] -> [t] -> [t]
  ```

- Check if Two Lists are Equal: You can use the function `(==)` to see if two lists are equal.

  ```
  (==) :: [t] -> [t] -> Bool
  ```