1 Haskell: An Introduction

Haskell is essentially Lambda Calculus plus

- Better syntax.
- Types.
- Built-in features like primitives (booleans, numbers, etc.), records, lists, recurison, and more.

1.1 Types

In Haskell, every expression either **has a type** or is **ill-typed** and rejected statically (at compile-time, before execution begins). This is similar to Java (which is statically typed), and differs from languages like Python (which is dynamically typed).

1.1.1 Type Annotations

While the Haskell compiler can infer types, you should still annotate your bindings with their types using `::`, like so:

```
aBoolean :: Bool
aBoolean = True

message :: String
message = if aBoolean
            then "True!"
            else "False."

rating :: Int
rating = if aBoolean
           then 10
           else 0
```

Note that we can use the GHCi command `:t` to inspect types. For example,

```
> :t if x then 'a' else 'b'            -- Char
```

1.1.2 Function Types

Functions have arrow types. That is,

- `\x -> e` has type `A -> B`.
- In other words, `e` has type `B` if `x` has type `A`.

For example,

```
> :t \b -> if b then 'a' else 'n'      -- Bool -> Char
```

Note that we can also have functions with mulitple parameters; that is:

```
pair :: String -> (String -> (Bool -> String))
pair :: String -> String -> Bool -> String      -- Same as above.
pair x y b = if b then x else y                 -- Definition of function.
```

(Quiz.) With `pair :: String -> String -> Bool -> String`, what would GGHCi say to

```
> :t pair "apple" "orange"
```

(a) Syntax error

(b) The term is ill-typed.

(c) `String`

(d) `Bool -> String`

(e) `String -> String -> Bool -> String`

The answer is **D**. Recall that the annotation given is just syntactic sugar for

```
pair :: String -> (String -> (Bool -> String)).
```

So, if we pass in two `String`s, we get back a `Bool -> String`.

Like with general variables, function bindings should be annotated.

1.2 Lists

A list is either:

- An empty list

```
[]              -- pronounced "nil"
```

- Or, a head element attached to a tail list.

```
x:xs            -- pronounced "x cons xs"
```

Remarks:

  - Note that this is pronounced `cons` for historical element.
  - The head element is just the first element. The *tail* list is everything after the head element (not the last element).

1.2.1 Example List Declarations

```
[]                              -- A list with 0 elements.

1:[]                            -- A list with 1 element.
                                -- This is essentially just a cons of a head
                                -- which is the number 1, followed by an empty
                                -- list.

(:) 1 []                        -- Same thing as above. Any infix operator 'op' in
                                -- Haskell can be transformed into a regular
                                -- function which can be called in infix notation
                                -- by putting it in parentheses ('op').

1:(2:(3:(4:[])))                -- A list with 4 elements.

1:2:3:4:[]                      -- Same thing (cons, :, is right associative).

[1, 2, 3, 4]                    -- Same thing (syntactic sugar for
                                -- 1:(2:(3:(4:[]))) ).
```

Remark: With regards to the infix operator in the third example, this is not just exclusive to lists. For example, `2 + 3` can be equivalently written as `(+) 2 3`. Likewise, `cmpSquare 2 3` can be equivalently written[1] as `2 ‘cmpSquare‘ 3`.

1.2.2 Constructors and Values

`[]` and `(:)` are known as the list **constructors**. We've seen constructors before; for example,

- `True` and `False` are both `Bool` constructors.
- `0`, `1`, `2`, and so can be *thought of* as `Int` constructors.
- These constructors didn't take any parameters; so, we call then *values*.

In general, a **value** is a constructor applied to other values. The list examples above are list *values*.

[1] Using backticks to denote the function name.