List and Option Types

We will discuss in this section two new values and corresponding types, that are emblematic of many functional programming ideas.

Lists

Lists are essentially a sequence of elements of the same type, that can only be accessed in the order they appear. Coming from more mainstream languages, the closest analog would be single-linked lists.

Technically, a value of type `list` is:

- either the value `[]`, representing an empty list, or
- the expression `a :: l` where `l` is a previously constructed list of elements of some type, and `a` is another element of that same type. We say that we *prepend* `a` to `l`.

For example, we could have the list `1 :: 2 :: 3 :: []`, which parenthesized would look like `1 :: (2 :: (3 :: []))`.

A syntactic sugar for the same thing would be `[1; 2; 3]`. We will refer to these as *list literals*.

Essentially lists are *containers* for values of another type. For that reason saying a list has type `list` is not enough; we must specify the contained type. We do this by prepending. So:

- A list of type `int list` contains integers.
- A list of type `string list` contains strings.
- A list of type `(int * string) list` contains pairs of an int and a string. The parentheses are necessary. In abuse of notation, in literal notation the parentheses for the pairs can be omitted (but we will never do so). For example, `[1, "hi"; 2, "this"]` instead of `[(1, "hi"); (2, "this")]`.
- A list of type `int list list` contains a list of “lists of integers”. An example value would be `[[1; 2]; [3; 4]]`.

We will learn how to work with lists in a little while.

Option types

Option types are a feature not often found in mainstream languages. Option types are effectively containers for another type that allow for the possibility that there is no value. More precisely, a value of option type would be:

- either the keyword None, indicating the absense of a value, or
- the expression Some v where v is some value of the desired type.

As with lists, the actual type needs to specify the contained type. So we can have:

- Some 5, a value of type int option.
- Some "string", a value of type string option.
- Some [1; 2; 3], a value of type int list option.

and so on. So a value of option type is either a value of the contained type, or no value at all.

This is the kind of thing that you could handle in other language via null or nil. The approach using option types, and the lack of something like null makes it so that the type signatures of our functions can tell us when we are performing something that might not have a value. For example, we could imagine a function that looks for a integer in a list of integers, and returns that integer if it finds it. What should it do if it does not find it? The option type gives us a way around that. The function would have signature:

```
val f: int * int list -> int option
```

A function whose signature says that it returns an int must in fact always return an int, it can't fail silently by returning null. This is actually a very powerful feature, we get **more expressive types**. Our types tell us more about the behaviours of our functions.

Type practice

Before we move on, and in order to practice some of the above, write a literal value of each of the following types:

1. int * int option
2. (int * int) option
3. int option list
4. int list option

5. int list $*$ int option

6. int $*$ int option list

7. (int $*$ int option) list

8. (int $*$ int) option list