

```
(*
                                CS51 Lab 7
                                Modules and Abstract Data Types
*)

(* Objective: This lab practices concepts of modules, including files
as modules, signatures, and polymorphic abstract data types.

There are 4 total parts to this lab. Please refer to the following
files to complete all exercises:

    lab7_part1.ml -- Part 1: Implementing modules
-> lab7_part2.ml -- Part 2: Files as modules (this file)
    lab7_part3.ml -- Part 3: Polymorphic abstract types and
                        Interfaces as abstraction barriers
*)

(*=====
Part 2: Files as modules
```

A useful feature of OCaml is that it *automatically* wraps all of the functions and values that are defined in a single file into a module named after that file. The module name is the name of the file with the first letter capitalized. This functionality is in addition to the manual definition of modules as you've just used in Part 1, and it is a convenient way of separating code into separate namespaces when writing a large program.

There are other source files included in this lab, other than the 'lab7_partn.ml' files. The file 'color.ml' contains an implementation of a system for managing colors. (Recall from lab 5 the idea of colors as consisting of values for three color channels -- red, green, and blue.) Take a look at it to see what functions and values it contains, including a type for colors, and a means for converting values for the three color channels into the abstract 'color' type, extracting the channels individually from colors, and converting some standard color names to this color representation.

With the exception of Exercise 2A, you will need to modify *only* the file 'color.ml' to complete the exercises below.

A digression on accessing other modules:

You'll want to test this part of the lab using 'ocamlbuild', for instance, with

```
% ocamlbuild -use-ocamlfind lab7_part2.byte
% ./lab7_part2.byte
```

The 'ocamlbuild' command should automatically find modules that you've written that reside in the same directory as your source, compile those additional files, and link them to your compiled program. You can then access functions from those files under the module name, which (again) is the name of the file with the first letter capitalized. For instance, if 'color.ml' is in the same directory as 'lab7_part2.ml' (which it probably is), 'ocamlbuild' will find it and use it, since it is referenced through expressions like 'Color.red' and the like.

On the other hand, if you're testing with a top-level REPL, like utop or ocaml, it will *not* automatically find those modules and evaluate them for you. However, you can inform the REPL about those modules manually yourself with the '#mod_use' directive, like this:

```
# #mod_use "color.ml" ;;
```

allowing you to then refer to elements of the module as, for instance,

```
# Color.color_named ;;
- : Color.color_name -> int = <fun>
```

(Note the capitalized module name, as discussed above.)

Keep in mind however that the `#mod_use` and `#use` directives look only at their argument files, not at any corresponding `.mli` files, so the modules being used will *not* be restricted to the signatures in the corresponding `.mli` files. For that, you'll have to use the compiler (with `'ocamlbuild'` for instance).

.....
Exercise 2A: Replace the `'0'` in the expression below with an expression that extracts the red channel of the color named `'Red'`, thereby naming the result `'red_channel_value'`. The expression will be constructed from values in the `'Color'` module.
.....*)

```
let red_channel_value : int = 0 ;;
```

(* Let's investigate one way that a signature can be useful. Although `'color.ml'` contains an implementation of a basic color module, the implementation is unintuitive and obscure -- truly *horrid* in fact. (We did that on purpose.) You will want to change the implementation of `'color.ml'`, rewriting it wholesale. At the same time, you'll want to guarantee to users of the module (like this file itself!) that the functionality from the "outside" point of view stays the same; the way to do this is through module *signatures*.*

.....
Exercise 2B: Add a file `'color.mli'`, in which you define an appropriate signature for the `'Color'` module. Consider which types and values you want revealed to the user and which you would prefer to be hidden.

Once you have `'color.mli'` implemented, you should still be able to compile `'color.ml'` and run `'color.byte'`.

.....*)

(*.....
Exercise 2C:

In the file `'color.ml'`, modify the implementation of a color module as you see fit. Make the design choices that you think would be best for a color module implementation. In particular, you should be able to come up with a solution that is *much* SIMPLER, clearer, and more transparent than the one currently in `'color.ml'`. (Frankly, I recommend throwing out the `'color.ml'` contents entirely and starting over.)

To pass the unit tests, you'll want the RGB values for the colors in the `'color_name'` type to have the following values from lab 5:

R	G	B	Color
---	---	---	-----
255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	165	0	Orange
255	255	0	Yellow
75	0	130	Indigo

240 | 130 | 240 | Violet

.....*)

(* Here's the payoff: A user who uses the 'color' module, by virtue of having to stay within the 'color.mli' interface, will not notice *any difference at all* between the two implementations of 'color.ml', the horrid one we provided and the elegant one you've developed. The underlying implementation can be changed any time in any way, so long as the functionality provided stays consistent with the signature.

Correspondingly, consider the 'List' module that you are familiar with by now. You never needed to worry about how the 'List' module was implemented in order to use it; you only needed to understand the interface.

Compartmentalization ftw! *)