```
(*
                                CS51 Lab 5
            Variants, algebraic types, and pattern matching
 *)

(*
Objective:

In this lab you'll get practice with built-in algebraic data types,
including product types (like tuples and records) and sum types (like
variants), and the expressive power that arises from combining both. A
theme will be the requirement of and checking for consistency with
invariants.

NOTE: Since we ask that you define types in this lab, you must complete
certain exercises before this will compile with the testing framework on
the course grading server. Exercises 1, 2, 6, and 9 are required for
full compilation. If you want to "peek" at the right type definitions,
you can check out <http://url.cs51.io/lab5-1>.  *)

(*======================================================================
Part 1: Colors as an algebraic data type

In this lab you'll use algebraic data types to create several data
structures.

Ultimately, you'll define several types and structures that allow you to
create a family tree. To do this, you need to create a type to store a
set of biographical information about a person, like name, birthdate,
and favorite color. This set of data is different from the enrollment
data from the prior lab, so you'll need to create a new type.

You might be tempted to do something simple like

  type person = { name : string; favorite : string; birthday : string } ;;

Let's consider why this may not be appropriate by evaluating the type
for each record field individually.

It seems reasonable for a name to be a string (though personal name
structures can be quite complex; see
<https://en.wikipedia.org/wiki/Personal_name>), so let's declare that
complete and move on.

The "favorite" field is more problematic. Although we named it such for
simplicity, it doesn't convey very well that we intended for this field
to represent a person's favorite *color*. This could be resolved with
some documentation, but is not enforced at any level other than hope.
Further, it's very likely that many people would select one of a subset
of simple colors. Let's fix this issue first.

......................................................................
Exercise 1: Define a new algebraic type, called `color_label`, whose
values can be any of red, orange, yellow, green, blue, indigo, or
violet, each specified by an appropriate constructor.
......................................................................*)

type color_label = NotImplemented ;;

(* This is a good start, but doesn't allow for definition of *all* of
the colors that, say, a computer display might be able to present. Let's
make it more expressive.

One of the most commonly used methods of representing color in digital
devices is as an "RGB" value: a triplet of values to represent red,
```

green, and blue components that, through additive mixing, produce the wide array of colors our devices render.

Commonly, each of the red, green, and blue values are made up of a single 8-bit (1-byte) integer. Since one byte represents 2 ** 8 = 256 discrete values, there are over 16.7 million (256 * 256 * 256) possible colors that can be represented with this method.

The three components that make up an RGB color are referred to as "channels". In this 8-bit-per-channel model, a value of 0 represents no color and a value of 255 represents the full intensity of that color. Some examples:

R	G	B	Color
255	0	0	Red
164	16	52	Crimson
255	165	0	Orange
255	255	0	Yellow
0	64	0	Dark green
0	255	0	Green
0	255	255	Cyan
0	0	255	Blue
75	0	130	Indigo
240	130	240	Violet

```
.....................................................................
Exercise 2: Define an algebraic color type that supports either
`Simple` colors (from the `color_label` type you defined previously)
or `RGB` colors, which would incorporate a tuple of values for the
three color channels.  You'll want to use `Simple` and `RGB` as the
value constructors in this new variant type.
.....................................................................*)

type color = NotImplemented ;;

(* There is an important assumption about the RGB values that determine
whether a color is valid or not. The RGB type presupposes an
*invariant*, that is, a condition that we assume to be true in order for
the type to be valid:

  INVARIANT: The red, green, and blue channels must each be a
  non-negative 8-bit int. Therefore, each channel must be in the range
  [0, 255].

Since OCaml, unlike some other languages, does not have native support
for unsigned 8-bit integers, you should ensure the invariant remains
true in your code. (You might think to use the OCaml `char` type --
which is an 8-bit character -- but this would be an abuse of the type.
In any case, thinking about invariants will be useful practice for
upcoming problem sets.)

We'll want a function to validate the invariant for RGB color values.
(Simple colors are assumed always valid.) There are several approaches
to building such functions, which differ in their types, and for which
we'll use different naming conventions:

* valid_rgb : color -> bool

  Returns true if the color argument is valid, false otherwise.

* validated_rgb : color -> color

  Returns its argument unchanged if it is a valid color, and raises an
  appropriate exception otherwise.
```

```
* validate_rgb : color -> unit

  Returns unit; raises an appropriate exception if its argument is not
  a valid color.

The name prefixes "valid_", "validated_", and "validate_" are intended
to be indicative of the different approaches to validation.

In this lab, we'll use the "validated_" approach and naming
convention, though you may want to think about the alternatives. (In
the next lab, we use the "valid_" alternative approach.)

.....................................................................
Exercise 3: Write a function `validated_rgb` that accepts a `color` and
returns that color unchanged if it's valid. However, if its argument is
not a valid color (that is, the invariant is violated), it raises an
`Invalid_color` exception (defined below) with a useful message.

For instance:

    # validated_rgb (RGB (10, 20, 30)) ;;
    - : color = RGB (10, 20, 30)
    # validated_rgb (RGB (10, 20, 300)) ;;
    Exception: Invalid_color "bad blue channel".
.....................................................................*)

exception Invalid_color of string ;;

let validated_rgb =
  fun _ -> failwith "validated_rgb not implemented" ;;

(*.....................................................................
Exercise 4: Write a function `make_color` that accepts three integers
for the channel values and returns a value of the color type. Be sure
to verify the invariant.
.....................................................................*)

let make_color =
  fun _ -> failwith "make_color not implemented" ;;

(*.....................................................................
Exercise 5: Write a function `rgb_of_color` that accepts a `color` and
returns a 3-tuple of integers representing that color. This is trivial
for `RGB` colors, but not quite so easy for the hard-coded `Simple`
colors. Fortunately, we've already provided RGB values for simple
colors in the table above.
.....................................................................*)

let rgb_of_color =
  fun _ -> failwith "rgb_of_color not implemented" ;;

(*=====================================================================
Part 2: Dates as a record type

Now let's move on to the last data type that will be used in the
biographical data type: the date field.

Above, we naively proposed a string for the date field. Does this make
sense for this field? Arguably not, since it will make comparison and
calculation extremely difficult.

Dates are frequently needed in programming, and OCaml (like many
languages) supports them through a library module; the `Unix` module
provides a `tm` data type for dates and times. Normally, we would
```

```
reduce duplication of code by relying on that module (the edict of
irredundancy), but for the sake of practice you'll develop your
own simple version.

.....................................................................
Exercise 6: Create a type `date` that supports values for years,
months, and days. First, consider what types of data each value should
be. Then, consider the implications of representing the overall data
type as a tuple or a record.
.....................................................................*)

type date = NotImplemented ;;

(* After you've thought it through, go to <http://url.cs51.io/lab5-1> to
see how we implemented the `date` type. If you picked differently, why
did you choose that way? Why might our approach be preferable?

.....................................................................
Exercise 7: Change your `date` data type, above, to implement it in a
manner identical to our method. If no changes are required...well, that
was easy.
.....................................................................

Like the color type, above, date values obey invariants. In fact, the
invariants for this type are more complex: we must ensure that days
fall within an allowable range depending on the month, and even on the
year.

The invariants are as follows:

- For our purposes, we'll only support non-negative years.

- January, March, May, July, August, October, and December have 31
  days.

- April, June, September, and November have 30 days.

- February has 28 days in common years, 29 days in leap years.

- Leap years are years that can be divided by 4, but not by 100,
  unless by 400.

You may find Wikipedia's leap year algorithm pseudocode useful:
<https://en.wikipedia.org/wiki/Leap_year#Algorithm>

.....................................................................
Exercise 8: Create a `validated_date` function that raises
`Invalid_date` if the invariant is violated, and returns the date
unchanged if valid.
.....................................................................*)

exception Invalid_date of string ;;

let validated_date =
  fun _ -> failwith "validated_date not implemented" ;;

(*.....................................................................
Exercise 9: Define a function `string_of_date` that returns a string
representing its date argument. For example,

    # string_of_date {year = 1706; month = 1; day = 17} ;;
    - : string = "January 17, 1706"
.....................................................................*)

let string_of_date =
```

```
  fun _ -> failwith "validated_date not implemented" ;;

(*======================================================================
Part 3: Persons as an algebraic data type

Now, combine all of these different types to define a person record,
with a name, a favorite color, and a birthdate.

......................................................................
Exercise 10: Define a `person` record type. Use the field names
`name`, `favorite`, and `birthdate`.
......................................................................*)

type person = NotImplemented ;;

(* Just for fun, here's a function to open up a graphics window and
display a person's information. Try it out! *)

(* display_person person -- Displays `person`'s name and birth date in
   a graphics window on a background of their favorite color. *)

let display_person ({name; favorite; birthdate} : person) : unit =
  let open Graphics in
  open_graph "";
  resize_window 200 60;
  let r, g, b = rgb_of_color favorite in
  set_color (rgb r g b);
  fill_rect 0 0 200 60;
  set_color white;
  moveto 20 40;
  draw_string name;
  moveto 20 20;
  draw_string (string_of_date birthdate);
  ignore (read_key ()) ;;

(*
let () = display_person {name = "Ben Franklin";
                         favorite = Simple Blue;
                         birthdate = {year = 1706; month = 1; day = 17}} ;;
 *)
```