```
(*
                              CS51 Lab 6
     Variants, algebraic types, and pattern matching (continued)
 *)

(* Objective: This lab is intended to reinforce core concepts in
   typing in OCaml, including:

     Algebraic data types
     Using algebraic data types to enforce invariants
     Implementing polymorphic algebraic data types
 *)

(*======================================================================
                          Part 1: Camlville
                  Variants and invariants revisited

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

First, you'll complete instructive examples on variant types and
enforcing invariants to model residences in Camlville. Then, you'll
revisit an example from the reading to implement a polymorphic list as
an algebraic data type. Building on the reading, we'll add some more
functionality to this type.

NOTE: As was the case last time, since we ask that you define types in
this lab, you must complete certain exercises (1 and 9) before this will
compile with the testing framework on the course grading server. We'll
provide hints so you'll be sure to get those types right.

......................................................................
Exercise 1: There are two kinds of residences in Camlville.  A *house*
in Camlville has a street name, zip code, and building number. An
*apartment* in Camlville has a street name, zip code, building number
(the same for all units in the building), and a unit number.

Define a variant type to capture the idea of a residence. What
representation and types make sense? Is there one right answer or
several possibilities?

We will start by assuming there are no restrictions on the zip code and
building and unit numbers other than those given by their types (zip
codes must be strings and building and unit numbers must be integers).
Though zipcodes on first glance are numbers, they are generally not
treated as numbers. (What would be the conceptual meaning of averaging
zipcodes?) They also can contain leading zeros (Cambridge, 02138).
Consequently, we choose to represent zipcodes as strings.

However, there are only four streets in Camlville (regardless of
zipcode) which are the following:

    High Street
    Hauptstrasse
    Rue Principale
    Main Street

How might you use algebraic data types to enforce this invariant on
street names?

**********************************************************************
Try to do this exercise first. There may be more than one way to solve
this problem, so if your solution doesn't compile against our unit
tests, see <https://url.cs51.io/lab6-1> for our solution and use that.
Do not proceed until your code compiles cleanly against the unit tests.
```

```
**********************************************************************
......................................................................*)

type residence = NotImplemented ;;

(* After implementing the residence type, compare it with our type
definition at <https://url.cs51.io/lab6-1>. Consider the tradeoffs we
may have considered if you find our definition differs from your own.

To compile against our unit tests, please change your definition to
match ours. You may comment out your original type definition if you
would like to keep it.

Valid zip codes in Camlville are given as five digits. For example,
12345, 63130, and 02138 are valid zipcodes, but -0004, 2138, and F69A
are not. We'll represent zip codes with strings, but will want to be
able to validate them appropriately. In this lab, we'll use the
`valid_` validation convention from lab 5. *)

(*......................................................................
Exercise 2: Define a function `valid_zip` that takes a `string` and
returns a `bool` indicating whether or not the string represents a valid
zip code. You may find the function `Stdlib.int_of_string_opt` and the
`String` or `Str` modules to be useful.

(For the purpose of defining a "valid zip code", you don't have to worry
about what the function does on strings interpreted as non-base-10
numbers. For example, `0x100` (hexadecimal) may or may not pass your
test but `abcde` definitely should not.)
......................................................................*)

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

(*......................................................................
Exercise 3: Define a function `valid_residence` that enforces proper
zipcodes, and verifies that building and unit numbers are greater than
0. It should return `true` if its argument is valid and `false`
otherwise.
......................................................................*)

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

(*......................................................................
Exercise 4: Time to get neighborly. Define a function `neighbors` that
takes two residences and returns a `bool` indicating whether or not
they are neighbors. In Camlville, a neighbor is someone living on the
same street in the same zipcode.

Note: By this definition, a residence is considered to be its own
neighbor.
......................................................................*)

let neighbors (place1 : residence) (place2 : residence) : bool =
  failwith "neighbors not implemented" ;;

(*......................................................................
Exercise 5: When buyers purchase a new residence in Camlville, they
must register the residence with the town hall, which creates a record
of the residence location and owner.

Implement a function `record_residence` to perform this bookkeeping. It
should accept a residence and a name (which should be a string) and
return the corresponding entry to be made as a value of the type
`town_record`, defined below. The town works hard to prevent fraudulent
```

```
residences from being entered into historical records and has asked you
to do the same by raising an `Invalid_argument` exception when
appropriate.
.....................................................................*)

type town_record = { residence : residence; name : string } ;;

let record_residence (res : residence) (name : string) : town_record =
  failwith "record_residence not implemented" ;;

(*.....................................................................
Exercise 6: Neighbor search.

As part of Bob’s promotion, he has been moved to the next floor up at
work. He doesn’t yet know any of his coworkers, and so he decides to
search through Camlville’s records to determine which of them are his
neighbors. Camlville keeps extensive records, so he doesn’t want to
have to look them up manually. Instead, he asks you to do it for him,
since he heard you were learning a lot of useful skills in CS51.

Write a function `named_neighbors` that, given two names (strings
again) and a `town_record list`, searches though the list to
determine if the two people are neighbors, as defined above, and
returns a `bool`. Return a `Failure` exception in the event that
either of the names does not appear in the list of records. You can
assume that no two town records have the same name.

Hint: You may find the `List.find` function to be useful.
.....................................................................*)

let named_neighbors =
  fun _ -> failwith "named_neighbors not implemented" ;;
```