```
(*
                              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 2: Binary search trees and Gorn addresses

Recall from Chapter 11 of the textbook that binary trees are a data
structure composed of nodes that store a value from some base type as
well as a left and a right subtree. To well-found this recursive
definition, a binary tree can also be empty. Defined in this way, binary
trees resemble lists, but with two "tails".

We'll use the definition of polymorphic binary trees from the
textbook, reproduced and annotated here to clarify the arguments of
the `Node` value constructor:
 *)

type 'a bintree =
  | Empty
  | Node of 'a * 'a bintree * 'a bintree ;;
  (*         ^         ^            ^
             |         |            |
           value     left         right
           at node   subtree      subtree      *)

(*.....................................................................
Exercise 7: Define a function `node_count : 'a bintree -> int`, which
returns the number of internal nodes (that is, not including the empty
trees) in a binary tree.
.....................................................................*)

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

(* A *binary search tree* is a binary tree that obeys the following
invariant:

    ....................................................................
    For each node in a binary search tree, all values stored in its
    left subtree are less than the value stored at the node, and all
    values stored in its right subtree are greater than the values
    stored at the node.
    ....................................................................

(For our purposes, we'll take "less than" to correspond to OCaml's
polymorphic `<` operator.)

For example, the following integer binary tree is a binary search
tree:
 *)

let example1 =
  Node (10, Node (5, Empty,
                     Node (7, Empty,
                              Node (9, Empty, Empty))),
```

```
        Node (15, Empty, Empty))

(* This tree can be depicted graphically (given the limitations of
ascii art) as

      10
      ^
     / \
    5   15
    ^   ^
     \
      7
      ^
       \
        9
        ^
 *)

(* The binary tree in Figure 11.3(b) in the textbook, duplicated here as
the `string bintree` `example2`, also happens to be a binary search
tree. Do you see why it obeys the invariant?  *)

let example2 =
  Node ("red",
        Node ("orange",
              Node ("green", Node ("blue", Empty, Empty),
                             Node ("indigo", Empty, Empty)),
              Empty),
        Node ("yellow", Node ("violet", Empty, Empty),
              Empty)) ;;

(* Binary search trees are useful because, as indicated by the name,
searching for a value in a binary search tree is especially efficient.
Rather than needing to search for a value throughout the whole tree,
the value stored at a node tells you determinately whether to search
in the left or the right subtree. Other functionality, like finding
the minimum or maximum value, are also especially efficient in binary
search trees. *)

(*.....................................................................
Exercise 8: Define a function `find_bst` for binary search trees, such
that `find_bst tree value` returns `true` if `value` is stored at some
node in `tree`, and `false` otherwise. For instance,

    # find_bst example1 9 ;;
    - : bool = true
    # find_bst example1 10 ;;
    - : bool = true
    # find_bst example1 100 ;;
    - : bool = false
.....................................................................*)

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

(*.....................................................................
Exercise 9: Define a function `min_bst`, such that `min_bst tree`
returns the minimum value stored in binary search tree `tree` as an
option type, and `None` if the tree has no stored values. For
instance,

    # min_bst example1 ;;
    - : int option = Some 5
    # min_bst Empty ;;
    - : 'a option = None
.....................................................................*)
```

```
let min_bst (tree : 'a bintree): 'a option =
  failwith "min_bst not implemented" ;;

(* Constructing binary search trees must be done carefully so that the
invariant is always preserved. Next, you'll implement a function for
adding a value to a binary search tree, while maintaining the
invariant. *)

(*.....................................................................
Exercise 10: Define a function `insert_bst : 'a -> 'a bintree -> 'a
bintree` such that if `tree` is a binary search tree, `insert_bst
value tree` returns a tree with the same elements as `tree` but also
with the new `value` inserted. (If the value is already in the tree,
the tree can be returned unchanged.) Make sure that the tree that is
returned maintains the binary search tree invariant.

For instance, your function should have the following behavior.

    # let example1_again =
        Empty
        |> insert_bst 10
        |> insert_bst 5
        |> insert_bst 15
        |> insert_bst 7
        |> insert_bst 9 ;;
    val example1_again : int bintree =
      Node (10, Node (5, Empty, Node (7, Empty, Node (9, Empty, Empty))),
       Node (15, Empty, Empty))

(The returned tree is the same one as `example1` depicted above.)
.....................................................................*)

let rec insert_bst (value : 'a) (tree : 'a bintree) : 'a bintree =
  failwith "insert_bst not implemented" ;;

(* The *Gorn address* of a node in a tree (named after the early
computer pioneer Saul Gorn of University of Pennsylvania
<https://en.wikipedia.org/wiki/Saul_Gorn>, who invented the technique)
is a description of the path to take from the root of the tree to the
node in question. For a binary tree, the elements of the path specify
whether to go left or right at each node starting from the root of the
tree. We'll define an enumerated type for the purpose of recording the
left/right moves. *)

type direction = Left | Right ;;

(* Thus, for the tree `example1` defined above, the Gorn address of the
root is `[]` and the Gorn address of the node containing the value `9`
is `[Left, Right, Right]`. *)

(*.....................................................................
Exercise 11: Define a function `gorn : 'a -> 'a bintree -> direction
list` that given a target value and a binary search tree returns the
Gorn address of the target value in the tree. It should raise a
`Failure` exception if the value doesn't occur in the tree. For
instance,

    # gorn 9 example1 ;;
    - : direction list = [Left; Right; Right]
    # gorn 10 example1 ;;
    - : direction list = []
    # gorn 100 example1 ;;
    Exception: Failure "gorn: value not found".
.....................................................................*)
```

```
let rec gorn (target : 'a) (tree : 'a bintree) : direction list =
  failwith "gorn not implemented" ;;
```