CS 496: Homework Assignment 2
Due: 11 February, 11:55pm

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment involves implementing a series of operations on general trees. Binary trees, as seen in class, are trees whose nodes may have at most two children. In contrast, *general tree* are trees in which each node can have any number of children. Moreover, in this assignment general trees will be assumed to be **non-empty**. An example of a general tree is given in Figure 1. Note that not all nodes have the same number of children. For example, the root node 33 has two children, but node 77 has three children. Also, node 12 has no children, it is a leaf node. The general tree of Figure 1 is a general tree of integers since the nodes hold integers. One could have a general tree holding values of other types. Indeed, general trees are polymorphic in the type of their contents.

General trees may be modeled in OCaml by means of the following algebraic data type:

```
type 'a gt = Node of 'a*('a gt) list
```

The type `'a gt` has only one constructor, namely `Node`, whose argument is a tuple of type `'a*('a gt) list`. Thus a general tree is a node that has a data value of type `'a` and a list of children (each of which are general trees). The simplest example of a general tree is a leaf. Leaves have an empty list of children. For example, the expression `Node(12,[])`, of type `int gt`, models a leaf that holds the integer 12 as data. The following expression `t`, of type `int gt`, represents the tree depicted in Figure 1:

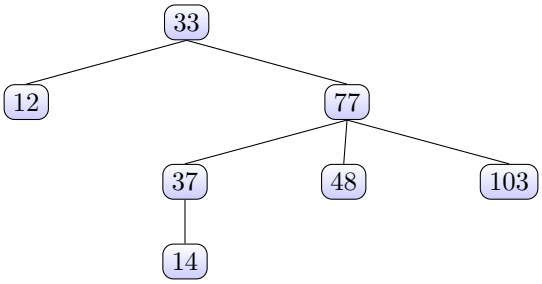

Figure 1: Sample value of type `int gt`

```
let t : int gt =
 Node (33,
       [Node (12,[]);
        Node (77,
              [Node (37,
                     [Node (14, [])]);
               Node (48, []);
               Node (103, [])])
       ])
```

Here is a sample function that given `n` builds a general tree that is a leaf holding `n` as data.

```
let mk_leaf : 'a -> 'a gt =
  fun n ->
  Node(n,[])
```

For example, `mk_leaf 12` returns the leaf `Node(12,[])`.

3 Operations to be Implemented

Please implement the following operations. For each operation, indicate its type by annotating the function definition as seen in class.

1. `height`: that given a general tree returns its height. The height of a tree is the length of the longest (in terms of number of nodes) path from the root to a leaf. Eg.

   ```
   # height t;;
   - : int = 4
   ```

2. `size`: that given a general tree returns its size. The size of a general tree consists of the number of nodes.

   ```
   # size t;;
   - : int = 7
   ```

3. `paths_to_leaves t`: returns a list with all the paths from the root to the leaves of the general tree `t`. Let n be the largest number of children of any node in `t`. A path is a list of numbers in the set $\{0, 1, \ldots, n-1\}$ such that if we follow it on the tree, it leads to a leaf. The order in which the paths are listed is irrelevant. Eg.

```
# paths_to_leaves t;;
- : int list list = [[0]; [1; 0; 0]; [1; 1]; [1; 2]]
```

4. `is_leaf_perfect`: that determines whether a general tree is *leaf perfect*. A general tree is said to be *leaf perfect* if all leaves have the same depth. Eg.

```
# is_leaf_perfect t;;
- : bool = false
```

5. `preorder`: that returns the pre-order traversal of a general tree. Eg.

```
# preorder t;;
- : int list = [33; 12; 77; 37; 14; 48; 103]
```

6. `mirror`: that returns the mirror image of a general tree. Eg.

```
# mirror t;;
- : int gt =
Node (33,
 [Node (77, [Node (103, []); Node (48, []); Node (37, [Node (14, [])])]);
  Node (12, [])])
```

7. `map f t`: that produces a general tree resulting from `t` by mapping function `f` to each data item in `d`. Eg.

```
# map (fun i -> i>20) t;;
- : bool gt =
Node (true,
 [Node (false, []);
  Node (true,
   [Node (true, [Node (false, [])]); Node (true, []); Node (true,
[])])])
```

8. `fold f t`: that encodes the recursion scheme over general trees. Its type is

```
fold: ('a -> 'b list -> 'b) -> 'a gt -> 'b
```

For example, here is how one may define `sum` and `mem` using `fold`:

```
let sum t =
  fold (fun i rs -> i + List.fold_left (fun i j -> i+j) 0 rs) t

let mem t e =
  fold (fun i rs -> i=e || List.exists (fun i -> i) rs) t
```

Here is the result of applying these functions in some examples involving `t`,

```
# sum t ;;
- : int = 324
# mem t 12;;
- : bool = true
# mem t 33;;
- : bool = true
# mem t 35;;
- : bool = false
```

9. Implement `mirror'` using `fold`. It should behave just like Exercise 6.

10. `degree`: that returns the maximum number of children that a node in the tree has. Eg.

```
# degree t;;
- : int = 3
```

4 Submission instructions

Submit a file `gt.ml` through Canvas. Each exercise is worth ten points.