TP numéro 4

Programmation fonctionnelle, ENSIIE

Semestre 4, 2019–20

Exercice 1 : Arbres binaires

On définit le type des arbres binaires :

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

1. Écrire une fonction `prefix : 'a tree -> 'a list` qui prend un arbre et qui retourne la liste des valeurs contenues dans ses nœuds en suivant l'ordre préfixe.
2. Écrire une fonction `infix : 'a tree -> 'a list` qui prend un arbre et qui retourne la liste des valeurs contenues dans ses nœuds en suivant l'ordre infixe.
3. Écrire une fonction `mirror : 'a tree -> 'a tree` qui retourne l'image dans le miroir de l'arbre passé en argument.
4. Montrer (sur papier) la propriété suivante : $\forall a$, `mirror (mirror` a`)` $= a$

Exercice 2 : Ensembles d'entiers

On peut définir leur type par :

```
module Int = struct
  type t = int
  let compare = fun x y -> x - y
end
module IntSet = Set.Make(Int)
```

1. Écrire une fonction récursive `range: int -> int -> IntSet.t` qui, sur la donnée de deux entiers `a` et `b`, renvoie un ensemble contenant tous les entiers compris entre `a` et `b` (inclus).
2. Écrire une fonction `nub: int list -> int list` qui, sur la donnée d'une liste `l` renvoie `l` sans les doublons. Par exemple, `nub [1;1;2;3;1;4]` retournera `[1;2;3;4]`.

   Note : utiliser une fonction auxiliaire `nub_aux: int list -> IntSet.t -> int list` qui prend comme argument supplémentaire l'ensemble des entiers déjà vu.
3. Écrire une fonction `from_list: int list -> IntSet.t` qui, sur la donnée d'une liste `l` renvoie l'ensemble des entiers présents dans `l`.
4. On définit la fonction `f: int list -> int list` par

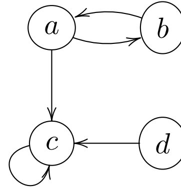
   ```
   let f = IntSet.elements (from_list l)
   ```

   Que fait la fonction `f` ? Quel est son coût pour une liste de taille n en entrée ?

Exercice 3 : Graphes

On reprend la définition des graphes vue en cours.

1. Définir le graphe suivant :



2. Écrire une fonction `to_dot : graph -> unit` qui affiche un graphe au format DOT (Graphviz) de la forme

```
digraph MonGraph {
  a -> b;
  a -> c;
  c -> a;
}
```

   Note : On pourra utiliser les fonctions `StringMap.iter` et `StringSet.iter`.

   On peut afficher un tel graphe avec la commande `dot -Tx11 fichier.dot`.

3. Écrire la fonction `reverse: graph -> graph` qui, sur la donnée d'un graphe `g`, renvoie une copie de `g` où le sens de chaque arête a été inversé.

4. Écrire une fonction `accessible : string -> string -> graph -> bool` qui prend en argument deux sommets `u`, `v` et un graphe `g` et qui teste s'il existe un chemin de `u` vers `v` dans `g`.

   Note : utiliser une fonction auxiliaire qui prend en paramètre supplémentaire l'ensemble des nœuds déjà visités, pour lesquels il n'est plus nécessaire de chercher un chemin.