TP 3 : Structures de données

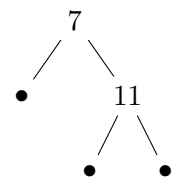
Exercice 1 - Les arbres binaires

Un arbre binaire est soit une feuille ; soit un nœud interne composé d'une valeur, d'un fils gauche et d'un fils droit. Pour définir un tel arbre en OCaml, on peut utiliser le type suivant :

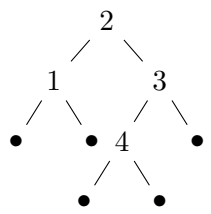
```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree ;;
```

1.1 Définir en OCaml les deux arbres binaires suivants, où les feuilles sont représentées par •.

a1 :



a2 :



1.2 Écrire une fonction `tree_max` qui, sur la donnée d'un arbre d'entiers, retourne la plus grande valeur stockée dans un nœud.

1.3 Écrire une fonction `hauteur` qui calcule la hauteur (longueur plus long chemin vers une feuille) d'un arbre donné en argument.

1.4 Écrire une fonction `list_of_tree` qui, sur la donnée d'un arbre `a`, renvoie la liste des valeurs dans les nœuds de `a`. La liste contiendra d'abord les valeurs dans le fils gauche, puis la valeur du nœud courant et enfin celles dans le fils droit.

bonus : Le faire sans utiliser `@` (souvenez-vous qu'une liste se construit de droite à gauche).

Exercice 2 - Expressions arithmétiques

On définit les expressions arithmétiques à l'aide du type OCaml suivant :

```
type expr = Const of int | Var of string | Plus of expr * expr
          | Neg of expr | Minus of expr * expr | Mult of expr * expr ;;
```

Pour calculer la valeur d'une expression `e`, il faut connaître les valeurs associées à chacune des variables dans `e`. On stockera ces informations dans un environnement dont le type est :

```
type env = (string * int) list ;;
```

2.1 Représenter l'expression $(x + 4) \times y - 5$ en utilisant le type `expr` défini précédemment.

2.2 Écrire la fonction `vars: expr -> string list` qui renvoie la liste des variables présentes dans une expression.

2.3 Écrire la fonction `eval: expr -> env -> int` qui calcule la valeur entière d'une expression. Par exemple, `eval (Plus (Var "x", Var "y")) [("x",11); ("y",14)]` doit renvoyer la valeur 25.

2.4 [⋆] Proposer une fonction `simpl` qui simplifie au maximum une expression arithmétique en effectuant les calculs quand c'est possible, et en utilisant les propriétés liées à 0 et à 1. Par exemple, l'expression `(1 - 1) * y + (1 + 1) * x` doit être simplifiée en `2 * x`.

Exercice 3 - Le module `Set`

Pour cette exercice, nous allons manipuler des ensembles d'entiers en OCaml par le biais du module `Set`. 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) ;;
```

3.1 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).

3.2 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.

3.3 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.

3.4 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 ?

3.5 Écrire une fonction `powerset: intSet.t -> IntSet.t list` qui, sur la donnée d'un ensemble E, renvoie la liste des sous-ensembles de E.

Exercice 4 - Manipulation de graphes

Pour cet exercice, on utilsera le type `graph` vu en cours.

4.1 Écrire une fonction `is_successor` telle que `is_successor u v g` renvoie `true` si `v` est un successeur de `u` dans `g`, et `false` sinon.

4.2 Écrire `nb_vertices: graph -> int` renvoyant le nombre de sommets d'un graphe.

4.3 Écrire la fonction `nb_edges: graph -> int` renvoyant le nombre d'arêtes d'un graphe.

4.4 Écrire la fonction `max_degree: graph -> int` renvoyant le degré sortant maximal d'un noeud du graphe.

4.5 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.