Révisions - Programmation OCaml

I. Pied à l'étrier

1. Écrire une fonction `factorielle : int -> int` qui prend en entrée un entier positif n et renvoie $n!$.

2. Écrire une fonction `est_premier : int -> bool` qui indique si un entier naturel est premier.

3. Écrire une fonction `pgcd : int -> int -> int` qui prend en entrée deux entiers naturels et renvoie leur plus grand diviseur commun en utilisant l'algorithme d'Euclide.

4. Écrire une fonction `somme_chiffres : int -> int` qui prend en entrée un entier naturel et renvoie la somme des chiffres de son écriture décimale.
Par exemple, `somme_chiffres 512` devra renvoyer `5 + 1 + 2 = 8`.

II. Manipulation de listes

5. Écrire une fonction `retire : 'a -> 'a list -> 'a list` prenant en argument un élément `x` et une liste `l` et calculant la liste `l` privée de toutes les occurrences de l'élément `x`.

6. Écrire une fonction `dernier : 'a list -> 'a option` retournant le dernier élément d'une liste d'un type quelconque. Cette fonction doit retourner `None` si la liste est vide et `Some(x)` si la liste contient au moins un élément et que le dernier élément est `x`.

7. Écrire une fonction récursive `min_max : 'a list -> 'a * 'a` prenant en argument une liste et retournant un couple dont la première composante est le minimum de la liste et la seconde composante est le maximum. Cette fonction doit être récursive et ne pas utiliser de fonctions récursives auxiliaires. Dans le cas où la liste est vide on lèvera une exception.

8. Écrire une fonction `scinde : 'a list -> 'a list * 'a list` prenant en argument une liste `l` et la découpant en deux listes, dans la première on rangera les éléments d'indices pairs de `l`, dans la seconde on rangera les éléments d'indices impairs de `l`. L'ordre des éléments dans les listes résultats doit être celui de la liste d'entrée.

9. Écrire une fonction `liste_bool : int -> bool list list` prenant en argument un entier naturel n et retournant la liste des 2^n listes contenant n booléens. Ainsi dans la liste résultat on devra pouvoir trouver chaque liste de n booléens.

III. Manipulation de tableaux

10. Écrire une fonction `premier_zero : int array -> int` calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `while`. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

11. Écrire une fonction `premier_zero : int array -> int` calculant l'indice du premier `0` dans un tableau d'entiers parcouru par boucle `for`, arrêtée au moyen d'une levée d'exception. La fonction devra retourner `-1` si le tableau ne contient aucun `0`.

12. Écrire une fonction `int array -> int array` prenant en argument un tableau d'entiers `t` et renvoyant le tableau des sommes cumulées du tableau `t`. Ainsi la case d'indice `i` du tableau résultat doit contenir `t.(0) + t.(1) + . . . + t.(i)`. On fournira une implémentation en $O(n)$, où n est la taille de `t`.

13. Écrire une fonction `recherche : int array -> int -> bool` prenant en argument un tableau d'entiers triés par ordre croissant, un entier `x` et testant si `x` apparaît dans le tableau. L'algorithme devra être récursif et ne pas manipuler de boucles. On assurera une complexité en $O(\log(n))$, où n désigne la taille du tableau.

14. Écrire une fonction `succ : bool array -> unit` prenant en argument un tableau de booléens représentant un entier écrit en base 2 (les bits de poids faibles sont à droite) et modifiant le tableau pour qu'il représente l'entier suivant. On lèvera une exception s'il n'est pas possible d'incrémenter l'entier.

15. Écrire une fonction `majoritaire : int array -> int -> int` prenant en argument un tableau d'entiers `t`, contenant des valeurs entières dans un intervalle $[\![0, m]\!]$, l'entier m en question, et renvoyant l'entier de ayant le plus d'occurrences dans le tableau. On fournira une implémentation en $O(n + m)$, où n désigne la taille du tableau.

IV. Implémentation de tris

16. Écrire une fonction `tri_selection : 'a list -> 'a list` permettant de trier une liste au moyen de l'algorithme du tri par sélection.

17. Écrire une fonction `tri_insertion : 'a list -> 'a list` permettant de trier une liste au moyen de l'algorithme du tri par insertion.

18. Écrire une fonction `tri_fusion: 'a list -> 'a list` permettant de trier une liste au moyen de l'algorithme du tri fusion.

19. Écrire une fonction `partition : 'a array -> int -> int -> int` prenant en argument un tableau `t`, deux indices `d` (début) et `f` (fin), et qui permute `t[d, f]` de manière à placer d'abord les éléments inférieurs à la valeur pivot `p = t.(f)`, puis cette valeur pivot, et enfin ceux strictement supérieurs à cette valeur. On renverra le nouvel indice de `p`.

20. Écrire une fonction `tri_rapide : 'a array -> unit` permettant de trier un tableau au moyen de l'algorithme du tri rapide.

V. Manipulation de types personnalisés

21. On définit le type énuméré suivant :

```
type coup = Pierre | Feuille | Ciseau
```

Définir une fonction `victoire : (coup * coup)list -> bool` qui prend en entrée une liste de matchs au Pierre-Feuille-Ciseau, et renvoie `true` si le joueur 1 a gagné, et `false` sinon.

22. On définit le type enregistrement suivant :

```
type 'a array_dyn = {mutable t : 'a array; mutable n : int}
```

permettant de définir un tableau dynamique (ou redimensionnable). L'entier n indique le nombre de cases $0, \dots, n-1$ occupées du tableau t.
Écrire une fonction `ajout : 'a -> 'a array_dyn -> unit` qui modifie ajoute un élément dans un tableau dynamique. Si le tableau est plein, on double sa capacité.

VI. Manipulation d'arbres

On définit les types suivants pour représenter respectivement un arbre binaire et un arbre d'arité quelconque non vide, tout deux étiquetés par un type quelconque.

```
type 'a arbreB = Vide | NoeudB of 'a * 'a arbreB * 'a arbreB

type 'a arbreG = NoeudG of 'a * 'a arbreG list
```

Fonctions générales

23. Définir une fonction `hauteur : 'a arbreB -> int` renvoyant la hauteur d'un arbre binaire. On rappelle que la hauteur de l'arbre vide est -1.

24. Définir une fonction `appartient : 'a -> 'a arbreB -> bool` permettant de tester si une étiquette apparaît dans un arbre binaire.

25. Définir une fonction `taille : 'a arbreG -> int` calculant la taille (i.e. le nombre de nœuds) d'un arbre général. On itérera sur l'arbre général au moyen de deux fonctions mutuellement récursives.

26. Réécrire la fonction précédente en utilisant la fonction `List.fold_left`.

27. On rappelle qu'un arbre binaire est parfait si tous ses nœuds ont la même profondeur. Écrire une fonction `est_parfait : 'a arbreB -> bool` indiquant si un arbre binaire l'est.

Parcours

28. Définir une fonction `parcours_prefixe : 'a arbreB -> 'a list` retournant un parcours en profondeur préfixe d'un arbre binaire. On fournira une implémentation récursive n'utilisant pas de fonction récursive auxiliaire, et de complexité quadratique.

29. Comment modifier la fonction précédente pour obtenir un parcours en profondeur suffixe ? Infixe ?

30. Définir une fonction `parcors_prefixe : 'a arbreB -> 'a list` retournant un parcours en profondeur préfixe d'un arbre binaire. On doit fournir une implémentation de complexité linéaire en la taille de l'arbre.

31. Définir une fonction `parcours_largeur : 'a arbreB -> 'a list` retournant un parcours en largeur d'un arbre binaire (la liste de ses étiquettes, triées par profondeur, et à profondeur équivalente de gauche à droite). On attend une complexité linéaire en la taille de l'arbre.

Arbre Binaire de Recherche (ABR)

32. Rappeler la définition d'un arbre binaire de recherche.

33. Définir une fonction `recherche : 'a -> 'a arbreB -> bool` qui indique si une étiquette est présente dans un ABR.

34. Définir une fonction `ajout : 'a -> 'a arbreB -> 'a arbreB` qui ajoute un nouveau nœud à un ABR, et renvoie le nouvel ABR ainsi créé.

35. Définir une fonction `retire : 'a -> 'a arbreB -> 'a arbreB` qui retire un nœud, et renvoie le nouvel ABR ainsi créé.
Indication : Lorsqu'on supprime un nœud interne ayant 2 fils, on le remplace par la plus grande valeur de son fils gauche.

36. Définir une fonction `est_ABR : 'a arbreB -> bool` qui indique si un arbre binaire est un ABR.
Indication : Un arbre binaire est un ABR ssi son parcours infixe est trié dans l'ordre croissant.

37. Implémenter une fonction `tri_abr : 'a list -> 'a list` qui trie une liste d'éléments, en les insérant successivement dans un arbre binaire de recherche, puis en renvoyant le parcours infixe de l'ABR ainsi créé.

VII. Manipulation de graphes

Représentation par liste d'adjacences

38. Définir un type `graphe` permettant de stocker un graphe sous forme de liste d'adjacences.

39. Écrire une fonction `sont_voisins : graphe -> int -> int -> bool` qui prend en entrée un graphe orienté à n sommets, deux sommet i et j, et renvoie un booléen indiquant si l'arc (i, j) est présent dans le graphe.

40. Écrire une fonction `ajoute_voisin : graphe -> int -> int -> unit` qui prend en entrée un graphe orienté à n sommets, deux sommet i et j non reliés, et ajoute l'arc (i, j) au graphe. Comment modifier la fonction dans le cas d'un graphe non orienté ?

41. Écrire une fonction `degre_max : graphe -> int` qui prend en entrée un graphe non orienté, et renvoyant le degré maximum parmi les sommets.

Représentation par matrice d'adjacences

42. Définir un type `graphe` permettant de stocker un graphe sous forme de matrice d'adjacences.

43. Écrire une fonction `sont_voisins : graphe -> int -> int -> bool` qui prend en entrée un graphe orienté à n sommets, deux sommet i et j, et renvoie un booléen indiquant si l'arc (i, j) est présent dans le graphe.

44. Écrire une fonction `ajoute_voisin : graphe -> int -> int -> unit` qui prend en entrée un graphe orienté à n sommets, deux sommet i et j non reliés, et ajoute l'arc (i, j) au graphe. Comment modifier la fonction dans le cas d'un graphe non orienté ?

45. Écrire une fonction `degre_max : graphe -> int` qui prend en entrée un graphe non orienté, et renvoyant le degré maximum parmi les sommets.

Parcours

46. Définir une fonction récursive `parcours_profondeur : graphe -> int -> int list` qui prend en entrée un graphe non orienté défini par listes d'adjacence, ainsi qu'un sommet source s et renvoie la liste des sommets atteignables depuis s, dans l'ordre d'un parcours en profondeur.

47. Écrire une fonction `nb_cc : graphe -> int` qui prend en entrée un graphe non orienté et renvoie son nombre de composantes connexes.

48. Écrire une fonction `parcours_largeur : graphe -> int -> int list` qui prend en entrée un graphe non orienté défini par listes d'adjacence, ainsi qu'un sommet source s et renvoie la liste des sommets atteignables depuis s, dans l'ordre d'un parcours en largeur.
Indication : On pourra utiliser le module `Queue`.

49. Écrire une fonction `pcc : graphe -> int -> int array` qui prend en entrée un graphe non orienté défini par listes d'adjacence, ainsi qu'un sommet source s et renvoie le tableau des plus courts chemins de s à tout sommet en nombre d'arêtes, grâce à un parcours en largeur.

50. Écrire une fonction `pcc : graphe -> int -> int array * int array` qui prend en entrée un graphe non orienté défini par listes d'adjacence, ainsi qu'un sommet source s et renvoie deux tableaux : celui des plus courts chemins depuis s en nombre d'arêtes, et celui des pères pour reconstruire le chemin.