TP 3 : OCaml. Listes et types avancés

Exercice 1 – Suite sur les listes

▶ **Question 1** Définir une fonction `vectoriser : ('a -> 'b) -> 'a list -> 'b list` qui pour une fonction `f : 'a -> 'b` et une liste `l` renvoie la liste des images de ses éléments. *Remarque : c'est la fonction* `List.map`.

▶ **Question 2** *En une petite ligne,* définir une fonction `liste_est_pair : int list -> bool list` tel que `liste_est_pair l` renvoie la liste des booléens indiquant si l'élément au même index dans `l` est pair.

▶ **Question 3** Écrire une fonction `select : ('a -> bool) -> 'a list -> 'a list` tel que `select c l` renvoie la liste des éléments de `l` respectant la condition `c`.

▶ **Question 4** Écrire une fonction `pour_tout : ('a -> bool) -> 'a list -> bool` telle que `pour_tout p l` vérifie si le "prédicat" `p` est vérifié sur tous les éléments de la liste `l`. De la même façon, écrire `il_existe` qui vérifie qu'il existe un élément vérifiant le prédicat. *Ces fonctions correspondent aux fonctions prédéfinies* `List.for_all` et `List.exists`.

▶ **Question 5** Écrire une fonction `compose : ('a -> 'a) list -> 'a -> 'a` tel que `compose l x` compose toutes les fonctions de `l` dans l'ordre de la liste et les applique à `x`. De même, écrire la fonction `applique : ('a -> 'a) list -> 'a -> 'a` tel que `applique l x` applique successivement les fonctions de `l` à `x`.

▶ **Question 6** Pourquoi les fonctions dans la liste en argument de `compose` et `applique` ne peuvent pas avoir un type `'a -> 'b` avec deux types différents (non compatibles)? Par exemple, `compose [int_of_float; float_of_int] 0`.

▶ **Question 7** Écrire une fonction `est_sans_doublons_triee : 'a list -> bool` qui, en supposant que `l` est une liste triée, renvoie `true` si elle contient deux éléments égaux et `faux` sinon.

▶ **Question 8** Écrire une fonction `supprime_doublons_triee : 'a list -> 'a list` qui, en supposant que la liste `l` est triée, renvoie la liste des éléments distincts de `l` dans le même ordre (c'est-à-dire triée).

▶ **Question 9** Écrire les fonctions `est_sans_doublon` et `supprime_doublons` qui répondent aux mêmes spécifications, mais sans supposer que l'entrée est triée. *Indication : on pourra utiliser la fonction* `List.mem`.

▶ **Question 10** Pour chaque question, donner une borne supérieure du nombre de tests d'égalité (sur `'a`, on ne compte pas les tests de motif) nécessaires. *Indication : la fonction* `List.mem` *utilise au plus* $|l|$ *tests d'égalité.*

Maintenant, on va utiliser le même principe de suppression de doublons pour écrire un algorithme de compression naïf (bien que beaucoup utilisé). Le principe est d'encoder k éléments x successifs égaux par le couple (k, x).

▶ **Question 11** Écrire les fonctions `encode : 'a list -> (int * 'a) list` et `decode : (int * 'a) list -> 'a list` qui implémentent la compression et la décompression.

Exercice 2 – Types somme

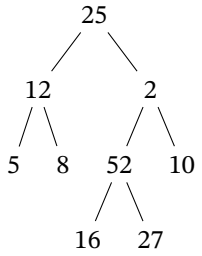
On va s'entraîner à écrire des types avancés et à les utiliser, d'abord en construisant des types implémentant des structures déjà vus en cours. On commence par l'Exercices VI et VIII. Ensuite, on reprend le type `arbre_etiquete` du cours, que l'on nomme simplement `arbre` et que l'on définit comme suis :

OCaml
```
type 'a arbre =
  | Vide
  | Noeud of 'a arbre * 'a * 'a arbre
```

Ensuite, on fait l'Exercice VII.

▶ **Question 1** Écrire une fonction `nombre_feuilles : 'a arbre -> int` qui calcule le nombre de feuilles d'un arbre (c'est-à-dire de nœuds sans fils).

On va implémenter des *parcours* d'arbre, c'est-à-dire des fonctions qui évaluent une fonction sur les nœuds d'un arbre dans un certain ordre. On définit l'arbre suivant :



▶ **Question 2** Écrire cet arbre avec le type `int arbre`.

▶ **Question 3** Écrire une fonction `prefixe : 'a arbre -> string list` qui renvoie la liste des nœuds de l'arbre dans l'ordre suivant : d'abord la racine, puis les nœuds du parcours préfixe du fils gauche, et enfin les nœuds du parcours préfixe du fils droit.

▶ **Question 4** Faire de même pour `infixe` où l'on visite d'abord les nœuds du fils gauche, puis la racine, enfin le fils droit.

▶ **Question 5** Faire de même pour `suffixe` où l'on visite d'abord le fils gauche, puis le fils droit, et enfin la racine.

▶ **Question 6** Appliquer ces trois parcours à l'exemple précédent.

Exercice 3 – Type enregistrement

On reprend par le type défini pour l'Exercice IX. Ensuite :

▶ **Question 1** Écrire une fonction `duel : carte -> carte -> int` qui calcule le résultat d'un duel de bataille entre deux cartes, et qui renvoie -1 si la première carte gagne, 1 si la seconde gagne, et 0 en cas d'égalité. *On considère que l'as est la carte la plus faible.*

▶ **Question 2** On va jouer une partie de cartes semblable à la bataille, avec deux joueurs : chaque joueur a une liste de cartes, et au tour n chaque joueur sa $n$ième carte, jusqu'à ce que l'un d'entre eux n'ait plus de cartes. Celui qui a gagné le plus de tours (au sens de la fonction `duel`) est déclaré vainqueur. Écrire une fonction `partie : carte list -> carte list -> int` qui renvoie un entier strictement positif si le premier joueur gagne, strictement négatif si le second gagne, et 0 en cas d'égalité.

▶ **Question 3** (Bonus) Écrire une fonction `partie_bataille` qui joue une partie de bataille et renvoie le gagnant. *On ne cherchera pas à détecter les parties infinies. Pour simplifier, on peut considérer que dans les duels égaux, chaque joueur récupère sa carte.*

▶ **Question 4** Définir les opérateurs usuels sur les nombres rationnels ($+ - / *$) en utilisant le type `type quotient {den : int; num : int}`.

▶ **Question 5** Même chose, pour le `type complexes = {re : float; im : float}`.