Programmation fonctionnelle
TD n° 3 : Programmer avec des arbres

Exercice 1 (Complexes)

On se donne deux représentations différentes des complexes. L'une sous leur forme cartésienne via le type `cartesien` et l'autre sous leur forme polaire via le type `polaire`.

```
type cartesien = {re : float; im : float}
type polaire = {modulus : float; arg : float}
```

1. Définissez une fonction `polaire_to_cartesien` qui convertit un complexe sous forme polaire vers sa représentation cartésienne.
2. Définissez une fonction `cartesien_to_polaire` qui convertit un complexe sous forme cartésienne vers sa représentation polaire. On rappelle l'existence d'une fonction `atan2` telle que `atan2 y x` renvoie une mesure de l'angle entre le vecteur directeur de l'axe des abscisses et le vecteur $\overrightarrow{OM}$ où M est le point de coordonnées (x, y).
3. On cherche maintenant à manipuler les complexes indifféremment de leur représentation. Définissez un type `complexe` qui permet de travailler simultanément sur les deux représentations possibles.
4. Définissez une fonction `conj` qui renvoie le conjugué d'un complexe. Le type de la fonction doit donc être `complexe -> complexe`.
5. Définissez une fonction `add` qui réalise la somme de deux complexes.
6. Définissez une fonction `mul` qui réalise le produit de deux complexes.

Exercice 2 (Tas d'appariement)

Un tas d'appariement (*pairing heap* en anglais) est une implémentation purement fonctionnelle d'un tas. Pour rappel, un tas est une structure de données qui dispose (a minima) des opérations suivantes :

— `get_min` qui renvoie le plus petit élément d'un tas,
— `remove_min` qui enlève le plus petit élément d'un tas,
— `insert` qui permet d'ajouter un élément à un tas.

Un arbre d'appariement est un arbre d'arité quelconque tel que :
— sa racine est minimale parmi tous les éléments de l'arbre,
— ses fils sont eux-mêmes des arbres d'appariement.

Un tas d'appariement est soit vide soit un arbre d'appariement.

1. Définissez un type `pairing_tree` qui représente un arbre d'appariement.
2. Définissez un type `pairing_heap` qui représente un tas d'appariement.
3. Définissez la fonction `get_min`.
4. Définissez une fonction `merge` qui fusionne deux tas d'appariement. Si l'un des tas est vide, on renvoie l'autre. Si les deux tas sont non vides, on ajoute le tas dont le plus petit élément est maximal aux sous-tas du second. Par exemples, si `t1` et `t2` sont deux tas dont les plus petits éléments sont respectivement `1` et `2`. On renvoie `t1` avec `t2` comme sous-tas supplémentaire.

5. Définissez la fonction `insert`. Pour cela, on crée un nouveau tas ne contenant que l'élément à insérer et on le fusionne avec le tas passé en paramètre.
6. La fonction `remove_min` n'est définie que sur les tas non vides. Le tas est donc un arbre d'appariement. Il suffit alors de fusionner tous ses fils pour éliminer l'élément minimal du tas (la racine). Pour des raisons d'efficacité, une stratégie en deux passes est adoptée. Une première passe va fusionner les fils par paires (d'où le nom de la structure de données). Une deuxième passe va fusionner récursivement les tas obtenus.

   Définissez la fonction `remove_min` (une fonction auxiliaire est bienvenue).
7. Définissez une fonction `from_list` qui crée un tas d'appariement à partir d'une liste.
8. Définissez une fonction `to_list` qui renvoie la liste (triée) des éléments composant un tas d'appariement.
9. En déduire une fonction `heap_sort` qui réalise le tri par tas d'appariement d'une liste.

Exercice 3 (Polynômes creux)

On représente un monôme par son coefficient (non nul) et par son degré. Un polynôme est représenté par la liste de ses monômes triée par degrés décroissants.

```
type monome = {coeff : float; deg : int}
type polynome = monome list
```

Remarquez que le type `polynome` est un *alias* pour le type `monome list`. OCaml autorise en effet de renommer des types. Dans notre cas, il s'agit surtout de mettre en avant qu'un polynôme a des contraintes particulières (monômes non nuls et degrés décroissants) qui le distingue d'une liste de monômes quelconque. Lorsqu'on manipulera ce type, il faudra s'assurer que les contraintes sont préservées.

1. Comment est représenté le polynôme nul ?
2. Définissez une fonction `degre` qui renvoie le degré d'un polynôme.
3. Quel est le type de la fonction `degre` ? Pouvons-nous y faire quelque chose ?
4. Définissez l'opérateur `++` qui réalise la somme de deux polynômes.
5. Définissez l'opérateur `~~` qui renvoie l'opposé d'un polynôme. En déduire l'opérateur `--` qui réalise la différence de deux polynômes.
6. Définissez la fonction `mult_monome` qui réalise le produit d'un monôme et d'un polynôme. En déduire l'opérateur `**` qui réalise le produit de deux polynômes.
7. Définissez l'opérateur `//` qui réalise la division euclidienne de deux polynômes.
8. Définissez une fonction `monomes_to_polynome` qui prend en paramètre une liste de monômes (sans aucune contrainte) et renvoie le polynôme qui correspond à leur somme.