Programmation fonctionnelle

⚠ Il est interdit ici, sauf précision contraire, d'utiliser les instructions d'affectation, mais également les boucles `while` ou `for` (y compris en utilisant les définitions de liste par compréhension).

Exercice 1
Écrire une fonction `trouve(p, t)` qui reçoit en arguments une fonction `p` (pour « propriété ») et un tableau `t` et retourne le premier élément `x` de `t` tel que `p(x)` vaut `True`. Si aucun élément de `t` ne satisfait `p`, alors la fonction retourne `None`.

Exercice 2
Écrire une fonction `applique(f, t)` qui prend en paramètres une fonction `f` et un tableau `t` et retourne un nouveau tableau dont les éléments sont les `f(x)` pour tous les éléments de `t`.

Exercice 3
La fonction (impérative) `tri_insertion(t)` ci-dessous prend en paramètre un tableau `t` et trie, en place, ses éléments par ordre croissant selon l'algorithme de tri par insertion vu en première.

```
def tri_insertion(t):
    for i in range(1, len(t)):
        v = t[i]
        j = i
        while j>0 and t[j-1]>v:
            t[j] = t[j-1]
            j = j-1
        t[j] = v
```

1. Modifier cette fonction (la laisser impérative) pour qu'elle prenne comme argument supplémentaire une fonction `inf` prenant deux arguments et telle que `inf(x,y)` retourne un booléen indiquant si `x` est inférieur à `y`.
   Autrement dit, définir une fonction `tri_insertion(t, inf)` sur le modèle de la fonction `tri_insertion(t)` qui trie `t` selon l'ordre défini par la fonction `inf` donnée en argument.
2. On souhaite à présent écrire une fonction `tri_insertion` dans le paradigme fonctionnel.
   Cette fonction doit retourner une liste triée, et pas modifier la liste donnée en argument.
   Compléter le programme suivant pour qu'il réalise l'algorithme de tri par insertion.

```
def queue(liste):
    '''la queue de la liste'''
    return ...

def tete(liste):
    '''la tête de la liste'''
    return ...
```

```
def insere(v, t):
    '''insère la valeur v dans le tableau trié t'''
    if t==[] or v<=tete(t):
        return ...
    else:
        return ...

def tri_insertion(t):
    if ...:
        return []
    else:
        return insere(tete(t),tri_insertion(queue(t)))
```

3. Reprendre et modifier la définition précédente en ajoutant en argument supplémentaire une fonction `inf` à la fonction de tri comme pour la question 1.

   On pourra éventuellement placer la définition de la fonction `insere` dans celle de la fonction `tri_insertion(t, inf)` (quel en est l'intérêt ?).

Exercice 4

1. Écrire une fonction `double(f)` qui reçoit une fonction `f` en argument et retourne une fonction qui applique deux fois de suite la fonction `f` à son argument. Autrement dit, la fonction `double(f)` appliquée à `x` retourne `f(f(x))`.
2. Que vaut `double(double(lambda x: x*x))(2)` ? (Trouver le résultat de tête, puis lancer le calcul pour vérifier.

Exercice 5

Écrire une fonction `compose(f, g)` qui prend en arguments deux fonctions `f` et `g` et retourne leur composition, c'est à dre la la fonction `h` définie par `h(x)=f(g(x))`.

Exercice 6

À l'aide de la fonction `map` écrire les fonctions suivantes :

1. une fonction `triple(t)` qui retourne une liste sont les éléments sont les triples des éléments de la liste `t` donnée.
2. une fonction `maj_min(t)` qui, pour un tableau de caractères `t`, retourne la liste des même caractères en majuscule et minuscule sous forme de couples :

   ```
   assert maj_min(['a','F','G','y']) == [('A','a'),('F','f'),('G','g'),('Y','y')]
   ```

Exercice 7

À l'aide de la fonction `filter` écrire :

1. une fonction `plus_petit_plus_grand(t, n)` qui partage le tableau d'entiers `t` en deux tableaux, l'un contenant les éléments plus petit que `n` et l'autre contenant les éléments plus grand ou égal à `n`.

   ```
   assert plus_petit_plus_grand([2,24,3,14,42], 10) == ([2,3], [24,14,42])
   ```

2. une fonction `partition(p, t)` qui partage la liste `t` en deux listes : les éléments de `t` qui vérifient la condition `p` et les autres.
3. une fonction `long_mots(t, n)` qui prend en paramètres une liste de mots `t` et un nombre entier `n` et retourne la liste des mots dont la longueur est supérieure à `n`.

Exercice 8

À l'aide de la fonction `reduce` écrire les fonctions suivantes :

1. Une fonction `fact(n)` qui retourne la valeur de $n! = 1 \times 2 \times 3 \times ... \times n$, c'est à dire le produit des n premiers entiers.

```
assert fact(5) == 120
assert fact(12) == 479001600
```

2. Une fonction `maximum(t)` qui calcule le maximum d'une liste d'entiers `t`, puis une fonction `minimum(t)` qui calcule le minimum d'une liste d'entiers `t`.

   Écrire ensuite une fonction `min_max(t)` qui donne le minimum et le maximum d'une liste d'entiers `t`.

```
assert min_max([2,1,5,3]) == (1, 5)
```

3. Une fonction `somme_sup(t, n)` qui prend en paramètres un tableau d'entier `t` et un entier `n` retourne la somme des éléments de `t` supérieur à `n`.

```
assert somme_sup([4,2,56,-12,1],3) == 60
```

4. Une fonction `longueur(t)` qui calcule le nombre d'éléments de la liste `t`.

```
assert longueur([4,2,56,-12,1]) == 5
```

5. Une fonction `all_true(t)` qui prend en paramètre un tableau de booléen `t` et retourne `True` si tous les éléments de `t` sont `True` et `False` sinon.

```
assert all_true([True,True,True])
assert not all_true([True,False,True])
```

   Écrire ensuite une fonction `check_all(p, t)` qui reçoit en argument une fonction `p` et un tableau `t` et revoie `True` si pour tous les éléments `x` de `t`, `p(x)` vaut `True` et `False` sinon.

```
assert check_all(lambda x: x%2==0, [2,4,12,42])
assert not check_all(lambda x: x%2==0, [2,3,12,42])
```

6. Une fonction `any_true(t)` qui prend en paramètre un tableau de booléen `t` et retourne `True` si au moins un des éléments de `t` vaut `True` et `False` sinon.

```
assert not any_true([False,False,False])
assert any_true([True,False,True])
```

   Écrire ensuite une fonction `check_any(p, t)` qui reçoit en argument une fonction `p` et un tableau `t` et revoie `True` s'il existe au moins une valeur `x` de `t` telle que `p(x)` vaut `True` et `False` sinon.

```
assert check_any(lambda x: x%2==0, [1,3,11,42])
assert not check_any(lambda x: x%2==0, [1,3,11,41])
```

7. Une fonction `occurence(mot)` qui prend une chaîne de caractère `mot` en paramètre et retourne le dictionnaire des occurences des lettres formants ce mot.

```
assert occurence("banane") == {'a': 2, 'b': 1, 'e': 1, 'n': 2}
```

Indications (version facile) :

- Commencer par définir une fonction (utilisant `reduce` qui détermine l'occurrence d'un caractère donné dans un mot.
- On pourra ensuite utiliser la fonction `map`.
- De plus, il sera utile d'utiliser la fonction `dict` dont l'usage est le suivant :

```
>>> dict([('a', 4139), ('b', 4127), ('c', 4098)])
{'a': 4139, 'b': 4127, 'c': 4098}
```

En fait le code obtenu par cette méthode n'est pas efficace, puisque l'on recalcule, à chaque lettre du mot, son nombre d'occurrence. Sa complexité n'est alors pas linéaire mais quadratique.

Indications (version plus efficace) : Utiliser `reduce` avec comme argument une fonction, qui prend comme arguments un dictionnaire et un caractère, et qui retourne un dictionnaire mis à jour. L'idée est :

- Si le caractère est déjà une clé du dictionnaire, on ajoute 1 à sa valeur ;
- Sinon, on ajoute la clé avec pour valeur 1.

On pourra utiliser la syntaxe :

```
dico1 | dico2
```

qui retourne un dictionnaire dont le contenu est celui de `dico1` mis à jour avec les éléments de `dico2`.

8. À l'aide de la fonction `insere(v, t)` de l'exercice 3 :
   écrire une nouvelle fonction `tri_insertion(t)` qui utilise `reduce`.

Exercice 9

En partant d'une liste de `n` éléments (`l=range(n)`), on veut produire une nouvelle liste qui contient :

cas 1 : `e+1` pour tous les éléments `e` de la liste de départ (`[0,1,2,...]`→`[1,2,3,...]`)

cas 2 : seulement les éléments pairs de la liste de départ (`[0,1,2,...]`→`[0,2,4,...]`)

cas 3 : `e+1` pour tous les éléments pairs `e` de la liste de départ (`[0,1,2,...]`→`[1,3,5,...]`)

1. Pour chacun de ces cas, écrire :
   - Une version avec une boucle `for` ;
   - Une version avec `map` et/ou `filter` (sans transformer en liste) ;
   - Une version avec une définition de liste par compréhension.
2. **(Facultatif)** Comparer les temps de parcours du résultat pour ces trois variantes dans chacun des trois cas, pour différentes longueurs de listes (jusqu'à une longueur de 10^6). On pourra afficher les résultats sous forme de courbe en utilisant le module `matplotlib`.
   Note : les fonctions `map` et `filter` retournent des itérateurs, il est donc nécessaire de les parcourir (ou de les transformer en liste, ce qui revient sensiblement au même) pour avoir une idée du temps de calcul.