Ce TP (et a priori les suivants) est prévu pour être traité de manière linéaire et contient trois paliers. Lorsque vous franchissez un palier, vous êtes invité à me solliciter pour que je puisse voir votre travail.

Concernant mes attentes :

- idéalement, tout le monde doit franchir le palier 1 pendant la séance de TP. Si ce n'est pas le cas, il faut travailler de son côté pour gagner en autonomie ;
- j'attends que tout le monde atteigne le palier 2, quitte à ce qu'une partie du travail soit faite en dehors de la séance de TP ;
- atteindre le palier 3 montre une très bonne aisance et autonomie ;
- les questions au delà du palier 3, lorsqu'il y en a, sont uniquement pour occuper les plus rapides.

On s'attend à ce que le code écrit soit interprété ou compilé/exécuté, et testé correctement.

Exercice 0

1. Créer un fichier d'extension `.ml`.
2. Écrire un programme qui affiche « Hello World ! ».

La suite de l'exercice est à faire dans une console (intégrée à l'éditeur ou non).

3. Dans un interpréteur (on utilisera `utop` de préférence, mais `ocaml` convient aussi), interpréter le fichier et vérifier l'affichage. On rappelle que l'interprétation se fait avec la commande :

```
#use "nom_de_fichier.ml";;
```

4. En dehors de l'interpréteur, compiler puis exécuter le fichier. On rappelle que la compilation se fait avec la commande :

```
ocamlc nom_de_fichier.ml
```

On peut rajouter l'option de compilation `-o nom_executable` si on veut changer le nom du fichier exécutable obtenu (`a.out` par défaut). On rappelle que si on se trouve dans le bon répertoire dans la console, on peut exécuter un fichier exécutable avec la commande :

```
./nom_executable
```

1 Listes

On écrira les fonctions en échouant avec un message d'erreur avec `failwith` si la dimension de la liste ne permet pas de renvoyer de résultat. On utilisera des fonctions auxiliaires locales si nécessaire.

Exercice 1

1. Écrire une fonction `liste_alea : int -> int -> int list` qui prend en arguments deux entiers k et n et renvoie une liste de taille n contenant des entiers choisis aléatoirement et uniformément entre 0 et $k-1$. On rappelle que `Random.int k` renvoie un entier choisi aléatoirement et uniformément entre 0 et $k-1$. On pourra utiliser `Random.self_init ()` pour choisir une graine aléatoire.

   La fonction précédente sera utilisée pour tester les suivantes. Il est demandé de tester chaque fonction écrite.
2. Sans utiliser de fonction du module `List`, réécrire des fonctions récursives pour les fonctions déjà implémentées suivantes :
   - `length : 'a list -> int` (calcul de la taille) ;
   - `mem : 'a -> 'a list -> bool` (test d'appartenance) ;
   - `append : 'a list -> 'a list -> 'a list` (concaténation) ;

  - `map : ('a -> 'b) -> 'a list -> 'b list` (liste des images par une fonction) ;
  - `iter : ('a -> unit) -> 'a list -> unit` (application d'une fonction à tous les éléments d'une liste).
3. Écrire une fonction `somme : int list -> int` qui calcule la somme des éléments d'une liste d'entiers.
4. Écrire une fonction `maximum : 'a list -> 'a` qui renvoie l'élément maximal dans une liste.
5. Écrire une fonction `partition : ('a -> bool) -> 'a list -> 'a list * 'a list` telle que `partition p lst` renvoie une partition `l1, l2` telle que `l1` est l'ensemble des éléments de `lst` vérifiant le prédicat `p`, et `l2` ceux qui ne le vérifient pas. Par exemple, `partition (function x -> x < 0) [-1; -2; 3; -4; 5]` renvoie `[-1; -2; -4], [3; 5]`.

Exercice 2

Écrire une fonction `fibo : int -> int list` qui prend en argument un entier n et renvoie la liste contenant les $n+1$ premiers premiers termes de la suite de Fibonacci définie par $f_0 = 0$, $f_1 = 1$ et $f_{k+2} = f_{k+1} + f_k$. Calculer `fibo 60` (il faudra réécrire la fonction plus efficacement si nécessaire).

Palier 1

2 Arbres

Dans cette partie, on attend des fonctions récursives.

On considère le type d'arbres binaires étiquetés par des entiers défini par :

```
type arbre_bin = Vide | N of int * arbre_bin * arbre_bin
```

On considère également le type d'arbres non vides d'arité quelconque défini par :

```
type arbre = Noeud of int * arbre list
```

Le fichier `affichage_arbres.ml` contient une fonction pour afficher un arbre dans la console. Cette fonction peut être facilement adaptée à d'autres types d'arbres en modifiant les fonctions auxiliaires de calcul de la racine et des enfants.

Pour l'utiliser, on peut soit la charger dans l'interpréteur en tapant la commande (en supposant que le fichier se trouve dans le bon répertoire) :

```
#use "affichage_arbres.ml";;
```

soit ouvrir le module dans le fichier `.ml` avant compilation :

```
open Affichage_arbres
```

Attention, dans ce cas il faudra bien indiquer les deux fichiers pour compiler et obtenir un exécutable. Par exemple par :

```
ocamlc -o nom_executable affichage_arbres.ml TP1.ml
```

La fonction à utiliser pour afficher un arbre binaire est `afficher_arbre_bin : arbre_bin -> unit`. Celle pour un arbre d'arité quelconque est `afficher_arbre : arbre -> unit`.

Exercice 3

1. Écrire une fonction `arbre_bin_alea : int -> arbre_bin` qui prend en argument un entier n et renvoie un arbre aléatoire de taille n, dont les nœuds sont exactement tous les entiers compris entre 0 et $n-1$, et étiqueté de telle sorte que le parcours infixe de l'arbre est croissant. On choisira aléatoirement la taille de l'arbre gauche.

   Indication : on pourra écrire une fonction de signature `int -> int -> arbre_bin` prenant un argument supplémentaire k et renvoyant un arbre aléatoire similaire, mais dont les nœuds sont ceux compris entre k et $n+k-1$.

   La fonction précédente sera utilisée pour tester les suivantes. Il est demandé de tester chaque fonction écrite.
2. Écrire une fonction `taille_bin : arbre_bin -> int` qui calcule et renvoie la taille d'un arbre binaire.
3. Écrire de même une fonction `hauteur_bin : arbre_bin -> int`.
4. Écrire une fonction `affiche_prefixe : arbre_bin -> unit` qui prend en argument un arbre binaire et affiche les étiquettes de ses nœuds dans l'ordre d'un parcours en profondeur préfixe. On rappelle que `print_int : int -> unit` permet l'affichage d'un entier.
5. Écrire une fonction `infixe : arbre_bin -> int list` qui calcule et renvoie la liste des nœuds d'un arbre binaire dans l'ordre d'un parcours en profondeur infixe. On autorisera l'utilisation de `@` dans un premier temps, avant d'améliorer la fonction sans utiliser `@`, avec une complexité linéaire en la taille de l'arbre.

Exercice 4

Pour tester les fonctions sur les arbres d'arité quelconque, on pourra créer des arbres aléatoires de la manière suivante :

```
let rec vers_lst = function
    | Vide       -> []
    | N(x, g, d) -> Noeud(x, vers_lst g) :: vers_lst d

let arbre_alea n =
    let a = arbre_bin_alea (n - 1) in
    Noeud(n - 1, vers_lst a)
```

1. Écrire des fonctions `taille : arbre -> int` et `hauteur : arbre -> int` qui calculent la taille et la hauteur d'un arbre quelconque.
2. Écrire une fonction `suffixe : arbre -> int list` qui calcule et renvoie la liste des nœuds d'un arbre quelconque dans l'ordre d'un parcours en profondeur suffixe, avec une complexité linéaire en la taille de l'arbre.
3. Écrire une fonction `bin_vers_arbre : arbre_bin -> arbre` qui transforme un arbre binaire non vide de type `arbre_bin` en un arbre binaire de type `arbre` (c'est-à-dire où chaque liste de fils est de taille au plus 2).

3 Programmation impérative

Dans cette partie, on demande de ne pas utiliser de programmation récursive.

Exercice 5

1. Écrire une fonction `tab_alea : int -> int -> int list` qui prend en arguments deux entiers k et n et renvoie un tableau de taille n contenant des entiers choisis aléatoirement et uniformément entre 0 et $k-1$.

   La fonction précédente sera utilisée pour tester les suivantes. Il est demandé de tester chaque fonction écrite.
2. Écrire une fonction `init : int -> (int -> 'a) -> 'a array` qui prend en argument un entier n et une fonction f et renvoie le tableau de taille n contenant les valeurs $f(0)$, $f(1)$, …, $f(n-1)$.
3. Écrire une fonction `minimum : 'a array -> 'a` qui renvoie le minimum d'un tableau.

Palier 2

4. Écrire une fonction `tri_insertion : 'a array -> unit` qui trie en place un tableau par un tri par insertion.
5. Écrire une fonction `knuth : 'a array -> unit` qui mélange un tableau en place selon l'algorithme de Knuth, dont le principe est le suivant : on choisit aléatoirement l'élément à mettre en dernière position, puis celui à mettre en avant-dernière position, sans toucher au dernier, puis celui à mettre en avant-avant-dernière position, sans toucher aux deux derniers, etc.
6. Écrire une fonction `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a` qui prend en argument une fonction f, un élément a et un tableau $t = [|b_0, \ldots, b_{n-1}|]$ et renvoie la valeur :

$$f(f(\ldots f(f(a, b_0), b_1), \ldots), b_{n-2}), b_{n-1})$$

4 Pour aller plus loin

Exercice 6

Écrire une fonction `convertir : int list -> int` qui renvoie l'entier obtenu en écrivant les nombres dans l'ordre de la liste. Par exemple, `convertir [1; 23; 0; 4]` renvoie `12304`. On demande de ne travailler qu'avec des entiers.

Indication : on pourra commencer par transformer la liste de nombres en une liste de chiffres.

Exercice 7

La notation postfixe des expressions arithmétiques (appelée également notation polonaise inverse ou notation de Łukasiewicz) consiste à écrire les opérandes d'une expression avant les opérateurs. Par exemple, l'expression $(3 \times (5+4) - 2)$ s'écrira $3\,5\,4\,+\,\times\,2\,-$. Aucune parenthèse n'est requise pour écrire une telle expression.

Pour écrire une expression, nous utiliserons le type suivant où un symbole est soit un entier, soit un opérateur sur les entiers (addition, soustraction, multiplication ou division entière) :

```
type symb = I of int | Op of (int -> int -> int)
and expr = symb list
```

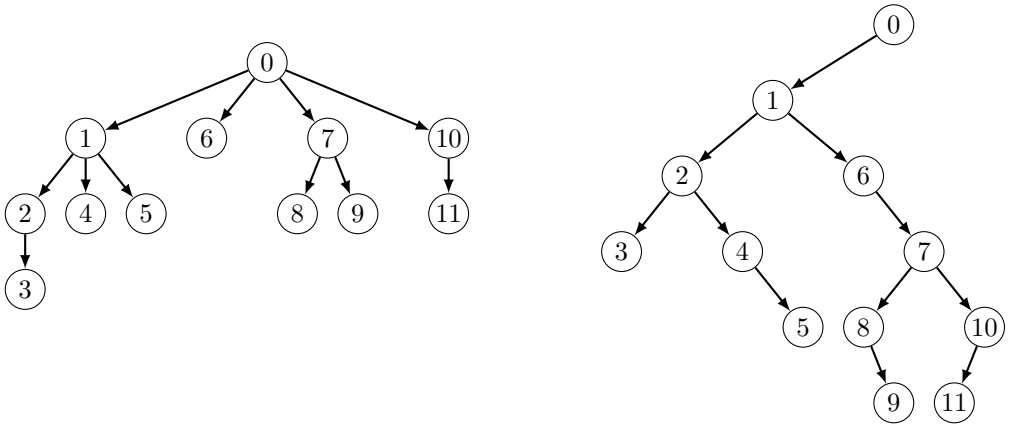
Écrire une fonction `postfixe` qui calcule l'évaluation d'une expression écrite en notation postfixe. Par exemple :

```
# postfixe [I 3; I 5; I 4; Op ( + ); Op ( * ); I 2; Op ( - )];;
- int = 25
```

Exercice 8

On cherche à transformer un arbre $\mathcal{A}$ d'arité quelconque en arbre binaire $\mathcal{B}$. Pour cela, on utilise la transformation canonique suivante appelée **codage enfant-adelphe** : un nœud x de $\mathcal{A}$ aura pour enfant gauche dans $\mathcal{B}$ son premier enfant (s'il existe) dans $\mathcal{A}$, et pour enfant droit son adelphe[1] suivant (s'il existe) dans $\mathcal{A}$, c'est-à-dire l'arbre apparaissant juste après x dans la liste d'enfants contenant x.

Par exemple, l'arbre de gauche sera transformé en l'arbre de droite :



1. Écrire une fonction `arbre_vers_bin : arbre -> arbre_bin` qui effectue cette transformation. *On pourra écrire une fonction auxiliaire prenant en argument une liste d'arbres et renvoyant un arbre binaire.*
2. Montrer que cette transformation conserve l'ordre préfixe.

Palier 3

1. Le mot *adelphe* signifie « enfant d'un même parent ». Il désigne indifféremment frère ou sœur et est une traduction littérale du mot anglais *sibling*.