Exercice 1 : Définitions

On définit le type `'a arbre_bin`, correspondant aux arbres binaires, ainsi :

```
type 'a arbre_bin = Vide | Noeud of 'a arbre_bin * 'a * 'a arbre_bin
```

On remarque qu'ici l'étiquette d'un arbre se situe entre les deux sous-arbres (choix différent de celui fait en cours). Ceci change évidemment l'écriture des fonctions, mais n'a aucune conséquence sur les algorithmes utilisés ou leur complexité.

1. Écrire des fonctions correspondant à :
   - la construction d'un nouvel arbre binaire étant donné ses sous-arbres gauche et droit et son étiquette ;
   - le calcul de l'étiquette d'un arbre binaire non vide ;
   - le calcul du sous-arbre gauche d'un arbre binaire non vide ;
   - le calcul du sous-arbre droit d'un arbre binaire non vide.

   Remarque : Dans la suite il n'est pas pertinent d'utiliser ces fonctions. Ceci sera en général plus lourd que d'utiliser directement la syntaxe OCaml qui a permis d'écrire ces fonctions.
2. (a) Écrire une fonction `taille : 'a arbre_bin -> int` qui prend en argument un arbre binaire et renvoie son nombre de nœuds (c'est à dire sa taille).

   (b) Quelle est sa complexité dans le pire cas en fonction de la taille de l'arbre ? En fonction de la hauteur de l'arbre ?
3. (a) Écrire une fonction `hauteur : 'a arbre_bin -> int` qui prend en argument un arbre binaire et renvoie sa hauteur.

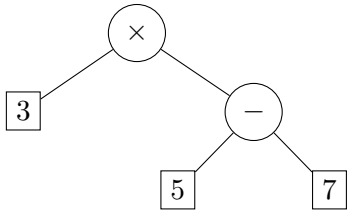
   (b) Quelle est sa complexité dans le pire cas en fonction de la taille de l'arbre ? En fonction de la hauteur de l'arbre ?

Exercice 2 : Arbres binaires stricts non vides : application aux expressions arithmétique

On considère ici une expression arithmétique (utilisant les opérateurs $+$, $-$, $\times$ et $/$) sur les entiers.

On peut représenter l'expression arithmétique par un arbre binaire strict non vide dont les feuilles sont étiquetées par des entiers et les nœuds par des opérateurs.

Par exemple, l'expression $3 \times (5 - 7)$ correspond à l'arbre ci-contre :



On définit les types suivants, ainsi que les fonctions de conversion :

```
type op  = Addition | Soustraction | Multiplication | Division
type arbre_arith = F of int | NI of arbre_arith * op * arbre_arith

let string_of_op op =
  match op with
   | Addition -> "+"
   | Soustraction -> "-"
   | Multiplication -> "*"
   | Division -> "/"

let op_of_string op =
  match op with
   | "+" -> Addition
   | "-" -> Soustraction
   | "*" -> Multiplication
   | "/" -> Division
   | _ -> failwith "pas un opérateur"
```

0. Écrire une fonction `eval : arbre_arith -> int` qui évalue l'expression représentée par un arbre.
1. On remarque que si on écrit l'arbre, qu'on remplace les opérateurs par leur symbole, et qu'on retire les constructeurs, on retrouve l'expression écrite sous forme infixe (la forme usuelle). Or une expression arithmétique comme $(((1 + 2) \times (5 - 7)) - 1)$ peut aussi être écrite :
   - Sous forme suffixe, où les opérateurs suivent les opérandes : $(((1\ \ 2\ \ +)\ \ (5\ \ 7\ \ -)\ \ \times)\ \ 1\ \ -)$;
   - Sous forme préfixe, où les opérateurs précèdent les opérandes : $(-\ \ (\times\ \ (+\ \ 1\ \ 2)\ \ (-\ \ 5\ \ 7))\ \ 1)$.

   Écrire trois fonctions `arbre_vers_infixe`, `arbre_vers_suffixe` et `arbre_vers_prefixe` qui prennent chacune en argument un `arbre_arith` et renvoient une chaîne de caractère correspondant à l'écriture infixe / suffixe / préfixe [1] de l'expression arithmétique. On mettra des parenthèse à l'extérieur de toute expression non réduite à un entier.

   Pour cela on utilisera l'opérateur de concaténation des chaînes `^` et la fonction `string_of_int` qui convertit un entier en chaîne de caractères.
2. Soit $s = s_0, \dots, s_{n-1}$ est la suite des étiquettes d'un `arbre_arith` A dans l'ordre préfixe.
   (a) Rappeler pourquoi s contient exactement $\frac{n+1}{2}$ entiers et $\frac{n-1}{2}$ opérateurs.
   (b) Montrer par induction structurelle sur A que pour tout $1 \leqslant k < n$, $s_0, \dots, s_{k-1}$ contient au moins autant d'opérateurs que d'entiers.
   (c) En déduire qu'il est impossible que deux arbres distincts aient la même suite d'étiquette dans l'ordre préfixe. Modifier la `arbre_vers_prefixe` pour ne pas inclure de parenthèses, qui ne sont donc pas nécessaires [2].
3. Écrire une fonction `prefixe_vers_arbre : string list -> arbre_arith` qui lit une expression arithmétique en écriture préfixe, avec les entiers et opérateurs dans une liste, et calcule l'arbre correspondant à cette expression.

   Pour cela, il peut être utile d'écrire une fonction récursive auxiliaire
   `aux : string list -> arbre_arith * list` qui lit l'expression (partielle) en entrée à partir d'un certain indice, calcule le premier arbre bien défini, et le renvoie ainsi que la partie non lue de la liste.

   On suppose que l'entrée est une expression arithmétique sous forme préfixe correcte.
4. Faire de même avec une fonction `infixe_vers_arbre`. Ici on supposera que toute sous-expression non réduite à un entier est entourée de parenthèses (y compris l'expression tout entière).

1. Pour les écritures infixes et suffixe, il faut séparer les opérandes par des espaces pour ne pas confondre `+ 3 12` et `+ 31 2`.
2. Il en va de même pour `arbre_vers_suffixe`, avec un argument similaire.