TP : Arbres

Dans ce TP, on travaille avec des arbres *persistants* : les données ne sont pas modifiables, toute fonction renvoyant un arbre construira un nouvel arbre.

1 Arbres binaires entiers et énumération postfixe

On travaille dans cette section avec des arbres binaires entiers (semblables à ceux du cours) définis par le type :

```
type ('a, 'b) abe = F of 'b | N of 'a * ('a, 'b) abe * ('a, 'b) abe ;;
```

On travaillera souvent avec des `(operateur, int) abe`, avec le type énuméré suivant :

```
type operateur = Plus | Moins | Fois | Div | Mod ;;
```

Ces types sont disponibles dans l'annexe qui se trouve sur mon site web, vous trouverez également quelques arbres.

Exercice 1. *Traduction.* Écrire une fonction `traduit op a b` de type `operateur -> int -> int -> int` calculant le résultat de l'opération de nom « op » sur les entiers a et b.

```
# traduit Moins 5 2 ;;
- : int = 3
```

Exercice 2. *Évaluation.* Écrire une fonction `evalue a` de type `(operateur, int) abe -> int` évaluant l'expression arithmétique associée à l'arbre.

```
# evalue ex_abe1 ;;
- : int = 15
# evalue ex_abe2 ;;
- : int = 28
```

On définit le type `enum` comme dans le cours :

```
type ('a, 'b) enum = A of 'a | B of 'b ;;
```

Exercice 3. *Énumération postfixe.* Écrire une fonction `enum_postfixe a` de type `(opetateur, int) abe -> (operateur, int) enum list` renvoyant l'énumérations postfixe de l'arbre, sous la forme d'une liste. Rappel : l'énumération postfixe d'un arbre non réduit à une feuille consiste en l'énumération :

— du sous-arbre gauche ;

— du sous-arbre droit ;

— de la racine.

```
# enum_postfixe ex_abe1 ;;
- : (operateur, int) enum list =
[B 5; B 7; A Mod; B 1; B 0; A Plus; A Div; B 9; B 1; A Plus; B 5; B 4;
 A Fois; A Mod; A Plus]
```

On peut évaluer une expression arithmétique directement à partir de l'énumération postfixe de son arbre. Pour cela, il suffit de parcourir l'énumération à l'aide d'une pile initialement vide :

— à la lecture d'un entier, on empile l'entier correspondant ;

— à la lecture d'un opérateur, on dépile les deux premiers éléments de la pile, on réalise l'opération, et on empile le résultat.

Si l'énumération correspond bien à l'énumération postfixe de l'arbre associé à une expression airthmétique, le processus ne fait pas d'erreur, et la pile contient un unique élément à la fin du parcours : le résultat de l'expression.

Exercice 4. *Évaluation via l'énumération.* Écrire une fonction `evalue_enum q`, de type `(operateur, int) list -> int` effectuant l'algorithme précédent. On utilisera une fonction auxiliaire et un accumulateur (une liste) pour implémenter la pile décrite précédemment. Attention, lorsqu'on lit un opérateur, dans quel sens effectuer l'opération ?

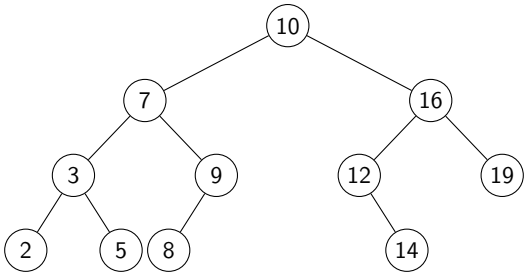
```
# evalue_enum (enum_postfixe ex_abe1) ;;
- : int = 15
```

*R*emarque : on peut reconstruire l'arbre en modifiant légèrement la fonction précédente, en manipulant une liste d'arbres plutôt qu'une liste d'entiers.

2 Quelques questions sur les ABR

2.1 Fonctions de base

Un arbre binaire de recherche (ABR) est un arbre binaire, dont les nœuds sont étiquetés par des clés, avec la propriété suivante : pour un nœud donné de clé k, les clés de son sous-arbre gauche sont strictement inférieures à k, et celles de son sous-arbre droit sont strictement supérieures. La figure suivante montre un arbre binaire de recherche dont les nœuds sont étiquetés par les entiers.



Le type utilisé pour les arbres binaires de recherche est le suivant :

```
type 'a abr = Vide | N of 'a abr * 'a * 'a abr ;;
```

Ainsi, un arbre binaire de recherche (ABR) est implémenté comme un arbre binaire entier, les nœuds « Vide » n'ayant pas été représentés dans la figure précédente. En Caml, on écrit pour construire cet arbre (un `int abr`) :

```
let ex_abr = N
  (N
    (N (N (Vide, 2, Vide), 3, N (Vide, 5, Vide)), 7,
     N (N (Vide, 8, Vide), 9, Vide)),
   10, N (N (Vide, 12, N (Vide, 14, Vide)), 16, N (Vide, 19, Vide)))  ;;
```

Pour travailler sur ces arbres, on rappelle que l'on procède par filtrage. Voici par exemple la fonction de recherche d'un élément dans un arbre binaire de recherche :

```
let rec recherche a x=match a with
  | Vide -> false
  | N(_,y,_) when y=x -> true
  | N(g,y,_) when y>x -> recherche g x
  | N(_,_,d) -> recherche d x
;;
```

Comme on le voit, la propriété d'ABR permet, lorsqu'on recherche un élément, de se diriger à chaque étape soit dans le sous-arbre droit, soit dans le sous-arbre gauche. La fonction de `recherche` ainsi écrite est de complexité $O(h)$, où h est la hauteur de l'arbre.

Exercice 5. On rappelle certaines opérations de dictionnaire vues en cours. Les-écrire.

1. `creer_vide ()` qui crée un dictionnaire (ABR) vide ;
2. `abr_vide a` qui teste si l'ABR `a` est vide ;

3. `inserer a x` de type `'a abr -> 'a -> 'a abr`, qui insère `x` dans `a` : attention à bien respecter la structure d'ABR. On renverra l'arbre à l'identique si `x` est déja présent dans `a`.

```
# inserer ex_abr 4 ;;
- : int abr =
N
 (N (N (N (Vide, 2, Vide), 3, N (N (Vide, 4, Vide), 5, Vide)), 7,
   N (N (Vide, 8, Vide), 9, Vide)),
 10, N (N (Vide, 12, N (Vide, 14, Vide)), 16, N (Vide, 19, Vide)))
```

On garantira (pour l'insertion) une complexité en $O(h)$, où h est la hauteur de l'arbre.

Exercice 6. Écrire une fonction `enumeration_infixe a` prenant en entrée un ABR et retournant sous forme de liste l'énumération infixe de ses nœuds.

```
# enumeration_infixe ex_abr ;;
- : int list = [2; 3; 5; 7; 8; 9; 10; 12; 14; 16; 19]
```

Exercice 7. Écrire une fonction `hauteur a` prenant en entrée un ABR et retournant sa hauteur. On convient que la hauteur de l'arbre `Vide` est -1.

```
#hauteur ex_abr ;;
- : int = 3
```

2.2 Un peu de statistique

Exercice 8. En utilisant une référence (vers un ABR), créer un ABR contenant les clés de 1 à 100 insérées dans l'ordre croissant. Quelle est sa hauteur ?

On veut maintenant avoir une estimation de la hauteur moyenne d'un ABR dont les clés sont fixés. On va pour cela créer des ABR aléatoirement.

Exercice 9. Reprendre la question précédente pour écrire une fonction `creer_abr t` qui prend en entrée un tableau et renvoie un ABR dont les clés sont les éléments de `t` insérées dans l'ordre du tableau.

Les fonctions suivantes (disponibles sur le site web) permettent de générer aléatoirement un tableau contenant les entiers de $[\![0, n-1]\!]$. La distribution est uniforme sur toutes les permutations de `[|0; 1; ··· ; n-1|]`.

```
let echange t x y=
  let a=t.(x) in t.(x) <- t.(y) ; t.(y) <- a
;;

let permutation_aleatoire t=
  let n=Array.length t in
  for i=n-1 downto 1 do
    let j=Random.int (i+1) in
    echange t i j
  done ;;

let creer_aleatoire n=
  let w=Array.make n 0 in
  for i=0 to n-1 do w.(i) <- i done ;
  permutation_aleatoire w;
  w;;
```

Exercice 10. À partir de `creer_aleatoire` et `creer_abr`, écrire une fonction `moyenne n m` renvoyant la hauteur moyenne de m ABR à n noeuds générés aléatoirement. On utilisera `float_of_int`, `+.`, `/.` pour le calcul de la moyenne. Je trouve une hauteur moyenne d'environ 12 pour des ABR à 100 éléments, 21 pour 1000 éléments et 30 pour 10000 éléments (moyenne effectuée sur 100 arbres).

*R*emarque : on peut montrer que la hauteur moyenne d'un ABR dont les n clés sont insérées aléatoirement à une hauteur moyenne $O(\log n)$: on obtient ainsi une bonne implémentation de la structure de dictionnaire. Malheureusement, si les données sont (par exemple) insérées dans l'ordre croissant (ou décroissant), la hauteur de l'arbre est linéaire en n. Il existe des modifications de la structure permettant un équilibrage « automatique » de l'arbre garantissant une hauteur en $O(\log n)$, tout en ne changeant pas la complexité des opérations de dictionnaire (également en $O(\log n)$, du coup).

Exercice 11. *Suppression d'un élément.* Écrire une fonction `supprimer a x` de type `'a abr -> 'a -> 'a abr`, supprimant de `a` la clé `x` si elle est présente. On pourra se ramener au cas de la suppression de la racine, et réfléchir à comment fusionner deux ABR dont les nœuds de l'un sont strictement inférieurs au nœuds de l'autre. Garantir une complexité $O(h)$ où h est la hauteur de l'arbre.

```
# supprimer ex_abr 9 ;;
- : int abr =
N (N (N (N (Vide, 2, Vide), 3, N (Vide, 5, Vide)), 7, N (Vide, 8, Vide)), 10,
 N (N (Vide, 12, N (Vide, 14, Vide)), 16, N (Vide, 19, Vide)))
```