Les Arbres

Partie 1 : le paquetage Paq_arbre

Un arbre binaire est une structure récursive, où chaque nœud porte un label, de type `Natural` ici, et a deux fils, eux-mêmes des arbres.

```
type Noeud;
type Arbre is access noeud;
type Noeud is record
    label: Natural;
    fils_gauche, fils_droit : Arbre;
end record;
```

Le but de cette partie est de construire un paquetage pour la manipulation d'arbres binaires, avec les fonctionnalités ci-dessous.

Question 1. Écrire les fonctions

```
function Racine(A:Arbre) return Natural;
function Fils_droit(A: Arbre) return Arbre;
function Fils_gauche(A: Arbre) return Arbre;
```

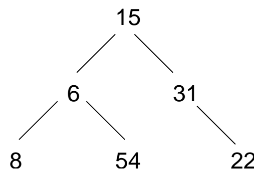
qui retournent respectivement le label de la racine de l'arbre `A`, le fils droit de `A` et le fils gauche de `A`. Dans les trois cas, si `A` est vide, vous devez déclencher une exception : `raise constraint_error;`

Question 2. Écrire une fonction

```
function Construire_arbre(N:Natural; A, B : Arbre) return Arbre
```

qui retourne l'arbre dont la racine est labellée par `N`, le fils gauche est `A` et le fils droit `B`.

Question 3. Ecrire une procédure récursive `Put(A : in Arbre)` qui affiche un arbre à l'écran sous la forme d'une expression parenthésée. Par exemple, l'expression $15(6(8, 54), 31(\ \ , 22))$ correspond à l'arbre



Question 4. La taille d'un arbre est définie comme le nombre de nœuds constituant cet arbre. Par exemple, la taille de l'arbre vide est 0. Ecrire une fonction récursive `Taille` qui calcule la taille d'un arbre.

Testez-bien votre paquetage avant de passer aux parties suivantes.

Partie 2 : les arbres binaires de recherche

Un *arbre de binaire recherche*, ou un *ABR*, est un arbre binaire qui satisfait la propriété

> pour chaque nœud, le label est strictement supérieur à tous les labels du sous-arbre gauche, et inférieur ou égal à tous les labels du sous-arbre droit.

Pour les questions suivantes, vous devez utiliser votre paquetage `paq_arbre`, sans le modifier.

Question 5. Écrire une fonction

```
function Appartient(A:Arbre; N:Natural) return Boolean
```

qui teste si un élément `N` appartient à l'arbre `A`, `A` étant supposé être un ABR. Quelle est la complexité de cette fonction, dans le meilleur des cas, dans le pire des cas ?

Question 6. Où se trouvent l'élément minimal et l'élément maximal d'un ABR ? Écrire des fonctions itératives `MinABR` et `MaxABR` qui retournent respectivement le plus petit et le plus grand élément d'un ABR.

La partie 3 utilise les ABR, comme la partie 2, mais n'a pas besoin des fonctions de la partie 2. Par contre, vous devez de nouveau utiliser le paquetage `paq_arbre`.

Partie 3 : *"L'entier mystérieux"*

Le jeu de l'*Entier mystérieux* n'est pas un jeu très intelligent. Il se joue à deux joueurs : le premier choisit un entier compris entre 1 et n, et le second doit deviner cet entier en posant des questions du style "5 ?", la réponse pouvant être "gagné", "trop petit" ou "trop grand".

La meilleure stratégie pour minimiser le nombre de questions à poser est évidemment de procéder par dichotomie : on commence par tester $n/2$. Si l'énigme à trouver est inférieure à $n/2$, on teste $n/4$, etc. Cette dichotomie peut être représentée par un ABR contenant tous les entiers de 1 à n: la racine de l'ABR contient le premier coup à jouer, puis suivant la réponse ("plus petit", "plus grand"), on joue la racine du fils gauche ou celle du fils droit. Et ainsi de suite. Le nombre de coups à jouer avant de gagner est la profondeur du nœud labellé par l'énigme dans cet ABR.

Question 7. Ecrire une fonction récursive

```
function Construire_Abr(I, J: Natural) return Arbre
```

qui construit l'ABR du jeu contenant les entiers de I à J. Faites attention aux cas particuliers où $I > J$, $I = J$ et $I = J - 1$.

Question 8. Ecrire une procédure

```
procedure Jouer(N : in Natural)
```

qui permet de faire jouer l'ordinateur à l'*Entier Mystérieux* en utilisant un ABR : vous choisissez un nombre et vous le faites deviner à l'ordinateur (sans tricher, bien sûr).