DEUG 1^{ere} année
MIAS 1 et MASS 1 – Programmation fonctionnelle

TP n° 8-9 : Arbres binaires

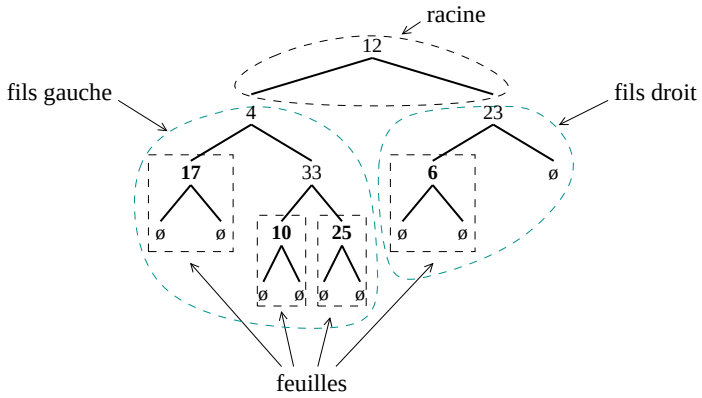
But : Aborder une structure de données arborescente. Définir et implanter un TAD «arbre binaire». Écrire des fonctions manipulant des arbres binaires.

Définition : Un arbre binaire est :

- soit l'arbre vide : Ø
- soit un nœud, c'est à dire un triplet $< v, g, d >$ où v est une *valeur*, g est un arbre binaire appelé *fils gauche* et d un arbre binaire appellé *fils droit*

On appelle *racine* d'un arbre le nœud unique qui n'est fils d'aucun autre nœud.
Si les deux fils d'un nœud sont des arbres vides alors on dit que ce nœud est une *feuille*.

Exemple 1 : arbre binaire où les valeurs sont des entiers :



On définit le TAD arbres binaires d'entiers naturels, *Arbre*, par :

les fonctions génératrices :

arbre-vide : $\rightarrow$ *Arbre*, qui renvoie l'arbre vide

cons-arbre : *Entier* , *Arbre* , *Arbre* $\rightarrow$ *Arbre*

$n \quad g \quad d \quad \mapsto \quad < n, g, d >$

qui renvoie l'arbre dont la valeur de la racine est n, de fils gauche g et de fils droit d.

les fonctions caractéristiques :

est-vide? : *Arbre* $\rightarrow$ *Booleen*, qui est vrai si un arbre est vide

val-racine : *Arbre* $\rightarrow$ *Entier*, qui renvoie la valeur de la racine

fils-gauche : *Arbre* $\rightarrow$ *Arbre*, qui renvoie le fils gauche d'un arbre

fils-droit : *Arbre* $\rightarrow$ *Arbre*, qui renvoie le fils droit d'un arbre

feuille? : *Arbre* $\rightarrow$ *Booleen*, qui est vrai si un arbre est une feuille

1) Définir en scheme les fonctions permettant d'implanter le TAD arbres binaires d'entiers en utilisant les listes.

Dans la suite du TP, toutes les fonctions manipulant des arbres doivent être indépendantes de l'implantation du TAD, on écrira ces fonctions en utilisant les fonctions génératrices et caractéristiques du TAD *Arbre* .

2) Définir en scheme les fonctions :

arbre=? : *Arbre* , *Arbre* $\rightarrow$ *Booleen* , qui rend vrai si deux arbres sont égaux.

nb-noeuds : *Arbre* $\rightarrow$ *Entier* , qui rend le nombre de nœuds constituant un arbre (un arbre vide est constitué de 0 nœud). L'arbre de l'exemple 1 est constitué de 8 nœuds.

hauteur : *Arbre* $\rightarrow$ *Entier* , qui rend la hauteur d'un arbre, c'est à dire le nombre maximum de nœuds reliant la racine de l'arbre à un nœud vide. La hauteur d'un arbre vide est 0. La hauteur de l'exemple 1 est 4.

valmax : *Arbre* $\rightarrow$ *Entier* , qui rend la plus grande valeur présente dans un arbre. Pour un arbre vide cette fonction rend la valeur -1. Pour l'exemple 1, la fonction **valmax** rend la valeur 33.

meme-struct : *Arbre*, *Arbre* $\rightarrow$ *Booleen*, qui rend vrai si deux arbres ont la même structures (on ne tient pas compte des valeurs).

3) Parcours d'un arbre en profondeur

Un arbre est un structure contenant un ensemble de données. Pour utiliser ces données, il faut parcourir l'arbre. Parcourir un arbre en profondeur consiste à passer ses nœuds en revue en commençant toujours par le même fils et en descendant le plus profondément possible dans l'arbre. Lorsqu'on arrive à un nœud vide, on remonte jusqu'au nœud supérieur et on continu le parcours à partir du fils encore inexploré.

Cela peut paraître compliqué, mais la structure récursive des arbres nous permet d'utiliser la récursivité pour les manipuler. Le principe général est le suivant : étant donnée un fonction **f** à appliquer sur les valeurs d'un arbre **A**, si **A** est vide alors le résultat est immédiat, sinon, on applique d'abord la fonction **f** à tous les nœuds du fils gauche de **A**, ce qui donne un résultat **R1**, puis à tous les nœuds du fils droit, ce qui donne un résultat **R2** et on calcul le résultat pour l'arbre **A** en fonction de la valeur de sa racine et des résultats **R1** et **R2**.

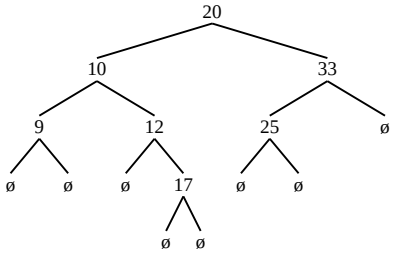
En utilisant ce principe, écrire les fonctions suivantes :

> **somme-noeuds** : *Arbre* → *Entier* , qui rend la somme des valeurs des nœuds d'un arbre. Pour un arbre vide, cette somme est nulle. Pour l'arbre de l'exemple 1, somme-noeuds rend 130.
>
> **fois-n** : *Arbre* , *Entier* → *Arbre* , qui, étant donné un arbre A, rend un arbre de structure identique où la valeur de chaque nœud a été multipliée par **n**.
>
> **liste-valeurs** : *Arbre* → *Liste*, qui rend la listes des valeurs des nœuds d'un arbre en parcourant celui-ci en profondeur. Pour l'exmple 1, liste-valeurs rend (12 4 17 33 10 25 23 6).

4) Arbres binaires ordonnées

On dit qu'un arbre binaire **A** est ordonnée quand, pour chacun de ses nœuds **N**, la valeur de **N** est supérieure à la valeur maximale présente dans son fils gauche et inférieure à la valeur minimale présente dans son fils droit.
Le premier exemple d'arbre n'est un arbre ordonné mais le suivant en est un.

Exemple 2 : arbre binaire ordonné.



Écrire les fonctions suivantes qui opèrent sur des arbre binaires ordonnés d'entiers :

> **est-dans?** : *Arbre* ,*Entier* → *Booleen*, qui rend vrai si l'entier n est une valeur présente dans un arbre binaire ordonné.
>
> **insere** : *Arbre* , *Entier* → *Arbre*, qui insère un nouveau nœud de valeur n dans un arbre ordonné, de telle façon que le résultat est un arbre binaire ordonné.
>
> Par exemple, l'insertion de la valeur 30, puis de la valeur 11 dans l'arbre de l'exemple 2 rend successivment les arbres suivants :