Programmation Fonctionnelle
TD N°4

Arbres binaires

Un arbre binaire est une structure définie récursivement. Il est :
– soit vide,
– soit un noeud formé :
  – d'une valeur (d'un type primitif de scheme)
  – d'un sous-arbre gauche qui est un arbre binaire,
  – d'un sous-arbre droit qui est un arbre binaire.

Ecrire les fonctions permettant :

- de créer un arbre vide **(a-vide)**
- de créer un arbre à partir d'une valeur, d'un sous-arbre droit et d'un sous-arbre gauche **(a-noeud v sag sad)**
- de savoir si un arbre est vide **(a-vide? A)**
- de récupérer la valeur de la racine d'un arbre **(a-valeur A)**
- de récupérer le sous arbre gauche **(a-gauche A)**
- de récupérer le sous arbre droit **(a-droit A)**

Bravo ! Vous venez de définir une abstraction d'arbre. A partir de maintenant, toutes les manipulations d'arbre binaires devront se faire par ces fonctions (barrière d'abstraction).

Ecrire une fonction permettant d'afficher un arbre

```
>(def A (a-noeud 1 (a-noeud 2 (a-vide) (a-noeud 3 (a-vide) (a-vide) )) (a-noeud 4 (a-vide)
(a-vide) )))
>(a-print A)
1
        2
                ()
                3
                        ()
                        ()
        4
                ()
                ()
```

Modifier la fonction pour qu'elle affiche l'arbre sous cette forme :

```
> (a-print-inf A)
                ()
        2
                        ()
                3
                        ()
1
                ()
        4
                ()
```

Ecrire une fonction permettant de faire la liste préfixe d'un arbre A :

- si A est l'arbre vide alors sa liste préfixe est vide,
- sinon la liste préfixe de A est égale à la concaténation de l'étiquette de la racine de A et des listes préfixes des sous-arbres gauche et droit de A

```
> (parcours-prof A)
'(1 2 3 4)
```

Cette fonction définit un parcours en profondeur préfixé. Chercher ce que pourrait être, et implémenter les, un parcours en profondeur suffixé et un parcours en profondeur infixé.

```
> (parcours-prof-suf A)
'(2 3 4 1)
> (parcours-prof-inf A)
'(2 3 1 4)
```

Ecrire une fonction permettant de faire le parcours en largeur d'un arbre.

```
> (parcours-largeur A)
'(1 2 4 3)
```

Ecrire une fonction permettant de calculer le nombre d'éléments d'un arbre.
Ecrire une fonction permettant de calculer la hauteur d'un arbre.

BONUS - Arbres syntaxiques

Lorsque les arbres représentent des expressions, on les appelle arbres syntaxiques. Par définition, les étiquettes des noeuds sont des opérateurs, les feuilles sont des opérandes (dans la suite nous nous limiterons aux expressions arithmétiques avec opérateurs binaires). Exemple : l'expression **`(+ (- 3 4) (* 1 (+ 5 6)))`** peut se représenter (pour simplifier, j'ai supprimé les ()) :

```
                3
        -
                4
+
                1
        *
                        5
                +
                        6
```

L'intérêt d'une telle représentation est qu'elle permet de manipuler facilement les expressions. Par exemple, si on transforme l'expression (liste) précédente en arbre et qu'on la retransforme en expression, mais en changeant le type de parcours, on obtient :

```
> (a-list-inf (arbrifier '(+ (- 3 4) (* 1 (+ 5 6)))))
'((3 - 4) + (1 * (5 + 6)))
```

Ou même, si on procéde dans l'autre sens :

```
> (a-list (arbrifier-inf '((3 - 4) + (1 * (5 + 6)))))
'(+ (- 3 4) (* 1 (+ 5 6)))
```

on obtient une expression scheme, qu'on peut alors évaluer :

```
> (eval (a-list (arbrifier-inf '((3 - 4) + (1 * (5 + 6))))))
10
```

Autre exemple, si on souhaite simplifier l'expression **`'(+ (- 3 4) (* b (+ 5 6)))`** :

```
> (a-list-inf (simplifier (arbrifier '(+ (- 3 4) (* b (+ 5 6))))))
'(-1 + (b * 11))
```