PLAN DU COURS 8

« Programmation récursive » - Saison 2

Structures Arborescentes : Arbres binaires

- Exemples d'arbres
- Définition d'un arbre binaire
- Barrière d'abstraction des arbres binaires
- Schéma de récursion des arbres binaires
- Exemples : nombre de noeuds, profondeur, liste préfixe
- Affichage sous forme d'un paragraphe

DES ARBRES

- Arbres binaires
  - Arbre généalogique des ascendants
  - Arbre représentant une expression arithmétique simple
- Arbres généraux
  - Arbre généalogique de la descendance
  - Table des matières d'un document
  - Arbre des fichiers
  - Noms d'ordinateur : *machine.domaine.pays*
  - Organigramme d'une société

Représentation graphique

Vocabulaire : noeud, étiquette, père-ascendant immédiat, fils-descendant immédiat, sous-arbre ...

DÉFINITION D'UN ARBRE BINAIRE

Un **arbre binaire** est une **structure de données** qui permet de représenter des éléments de même type, ordonnés hiérarchiquement.

Le type d'un tel arbre se note : **ArbreBinaire[α]**

Dans un arbre binaire non vide, chaque noeud porte une information (étiquette de type α), et a exactement 2 descendants immédiats.

Définition récursive : Un arbre binaire de type `ArbreBinaire[`α`]` est

- soit vide
- soit formé
  - d'un noeud (portant une étiquette de type α)
  - d'un sous-arbre gauche, qui est de type `ArbreBinaire[`α`]`
  - d'un sous-arbre droit, qui est de type `ArbreBinaire[`α`]`

BARRIÈRE D'ABSTRACTION

La barrière d'abstraction des arbres binaires doit contenir :

- **Constructeurs** pour construire un arbre binaire : `ab-vide`, `ab-noeud`
- **Accesseurs** pour accéder aux parties d'un arbre binaire : `ab-etiquette`, `ab-gauche` et `ab-droit`
- **Reconnaisseur** pour déterminer si un arbre binaire est non vide `ab-noeud?`

1. On manipule les arbres binaires uniquement à travers leur **barrière d'abstraction**.
2. Barrière d'abstraction => on ne connait que la **spécification** des fonctions. (Plus tard implantation)

Spécification des constructeurs

```
;;; ab-vide:  -> ArbreBinaire[α]
;;; (ab-vide) rend l'arbre binaire vide.

;;; ab-noeud: α * ArbreBinaire[α] * ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-noeud e B1 B2) rend l'arbre binaire formé de la racine d'étiquette
;;; «e», du sous-arbre gauche «B1» et du sous-arbre droit «B2».
```

Exemples : arbres binaires de nombres

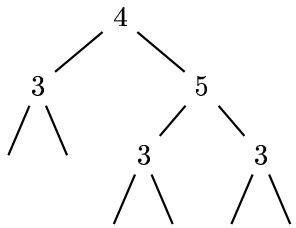
```
(ab-noeud 3 (ab-vide) (ab-vide))

(ab-noeud 8
          (ab-noeud 10 (ab-vide) (ab-vide))
          (ab-noeud 12 (ab-vide) (ab-vide)))
```


Exemple

```
(let* ((b0 (ab-noeud 3 (ab-vide) (ab-vide)))
       (b1 (ab-noeud 5 b0 b0))
       (b2 (ab-noeud 4 b0 b1)))
  b2)
```

renvoie la valeur `#<object>`, c'est-à-dire l'arbre



Spécification des accesseurs

```
;;; ab-etiquette: ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine de l'arbre «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»

;;; ab-gauche: ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»

;;; ab-droit: ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de «B»
;;; ERREUR lorsque «B» ne satisfait pas «ab-noeud?»
```


Exemple

```
(let* ((b01 (ab-noeud 3
                      (ab-vide)
                      (ab-vide)))
       (b02 (ab-noeud 2
                      (ab-vide)
                      (ab-vide)))
       (b1 (ab-noeud 5
                     b01
                     b02))
       (b2 (ab-noeud 4
                     b01
                     b1)))
  (ab-etiquette (ab-gauche (ab-droit b2))))   renvoie ? ? ?
```

Propriétés

- Pour tout couple d'arbres binaires G et D, et toute valeur v

```
(ab-etiquette (ab-noeud v G D))     → v
(ab-gauche (ab-noeud v G D))        → G
(ab-droit (ab-noeud v G D))         → D
```

- Pour tout arbre binaire non vide B

```
(ab-noeud (ab-etiquette B)
          (ab-gauche B)
          (ab-droit B))     → B
```


Spécification du reconnaisseur

```
;;; ab-noeud?: ArbreBinaire[α] -> bool
;;; (ab-noeud? B) rend vrai ssi «B» n'est pas l'arbre vide.
```

Par exemple

```
                    (ab-noeud? (ab-vide)) → #f
(ab-noeud? (ab-noeud 3 (ab-vide) (ab-vide))) → #t
```

La barrière d'abstraction contient aussi une fonction d'affichage, qui permet de « visualiser » les objets arbres binaires par des S-expressions

Une fonction d'affichage

```
;;; ab-expression: ArbreBinaire[α] -> Sexpression
;;; (ab-expression B) rend une Sexpression reflétant la construction de «B».

(let* ((b0 (ab-noeud 3 (ab-vide) (ab-vide)))
       (b1 (ab-noeud 5 b0 b0))
       (b2 (ab-noeud 4 b0 b1)))
  (ab-expression b2))
                          →
(ab-noeud
  4
  (ab-noeud 3 (ab-vide) (ab-vide))
  (ab-noeud 5
            (ab-noeud 3 (ab-vide) (ab-vide))
            (ab-noeud 3 (ab-vide) (ab-vide))))
```

Récursion sur les arbres binaires

Un arbre binaire est :
- soit vide
- soit constituée d'une étiquette, d'un sous-arbre gauche et d'un sous-arbre droit

```
;;; fRec: ArbreBinaire[alpha] -> ...
(define (fRec B)
  (if (ab-noeud? B)
      (combinaison (ab-etiquette B)
                   (fRec (ab-gauche B))
                   (fRec (ab-droit B)))
      cas-arbre-vide ) )
```


Nombre de noeuds

```
;;; nombre-noeuds:  ArbreBinaire[α] -> nat
;;; (nombre-noeuds B) rend le nombre de noeuds de B (la taille de l'arbre)
(define (nombre-noeuds B)
  (if (ab-noeud? B)
      (+ 1
         (nombre-noeuds (ab-gauche B))
         (nombre-noeuds (ab-droit B)))
      0))
```

Exemple

```
(let* ((b01 (ab-noeud 3 (ab-vide) (ab-vide)))
       (b02 (ab-noeud 2 (ab-vide) (ab-vide)))
       (b1 (ab-noeud 5 b01 b02))
       (b2 (ab-noeud 4 b01 b1)))
   (nombre-noeuds b2))
```


Trace de l'exécution

```
|(nombre-noeuds #<object>)
| (nombre-noeuds #<object>)
| |(nombre-noeuds #<object>)
| |0
| |(nombre-noeuds #<object>)
| |0
| 1
| (nombre-noeuds #<object>)
| |(nombre-noeuds #<object>)
| | (nombre-noeuds #<object>)
| | 0
| | (nombre-noeuds #<object>)
| | 0
| |1
```



```
| |(nombre-noeuds #<object>)
| | (nombre-noeuds #<object>)
| | 0
| | (nombre-noeuds #<object>)
| | 0
| |1
| 3
|5
```

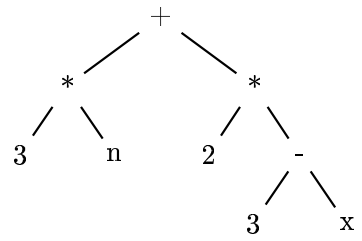
Profondeur

Définition récursive de la profondeur d'un arbre :

- la profondeur de l'arbre vide est 0,
- la profondeur d'un arbre non vide est égale au maximum des profondeurs de ses sous-arbres immédiats, augmenté de 1.

```
;;; profondeur:  ArbreBinaire[α] -> nat
;;; (profondeur B) rend la profondeur de B
(define (profondeur B)
  (if (ab-noeud? B)
      (+ 1
         (max (profondeur (ab-gauche B))
              (profondeur (ab-droit B))))
      0))
```


Différentes écritures d'une exp. arithm.



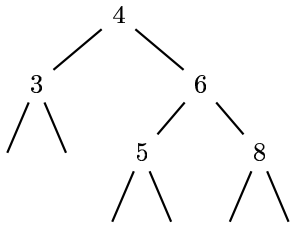
- infixe complètem^t parenthésée : `((3 * n) + (2 * (3 - x)))`
- préfixe Scheme : `(+ (* 3 n) (* 2 (- 3 x)))`
- polonaise préfixe : `+ * 3 n * 2 - 3 x`
- liste préfixe : `(+ * 3 n * 2 - 3 x)`
- polonaise postfixe : `3 n * 2 3 x - * +`

Liste préfixe

```
;;; liste-pref:  ArbreBinaire[α] -> LISTE[α]
;;; (liste-pref B) rend la liste préfixée de B
(define (liste-pref B)
  (if (ab-noeud? B)
      (cons (ab-etiquette B)
            (append (liste-pref (ab-gauche B))
                    (liste-pref (ab-droit B))))
      (list)))
```


Affichage en paragraphe

```
"
4
-3
--vide
--vide
-6
--5
---vide
---vide
--8
---vide
---vide
"
```



Types Ligne et Paragraphe

```
;;; paragraphe: LISTE[Ligne] -> Paragraphe
;;; (paragraphe L) construit le paragraphe formé des lignes de la liste L

;;; paragraphe-cons: Ligne * Paragraphe -> Paragraphe
;;; (paragraphe-cons ligne para) construit le paragraphe formé de la ligne
;;; « ligne » suivie des lignes du paragraphe « para »

;;; paragraphe-append: Paragraphe * Paragraphe * ... -> Paragraphe
;;; (paragraphe-append para1 para2 ...) construit le paragraphe corres-
;;; pondant à la concaténation des paragraphes donnés en argument

;;; lignes: Paragraphe -> LISTE[Ligne]
;;; (lignes para) rend la liste des lignes contenues dans le paragraphe « para ».
```


Fonctions sur les chaînes

Concaténation de chaînes

```
;;; string-append: string * string * ... -> string
;;; (string-append s1 s2 ...) construit la concaténation de toutes les
;;; chaînes reçues en argument
```

Transformation d'une valeur quelconque en chaîne

```
;;; ->string: Valeur -> string
;;; (->string v) construit la chaîne représentant l'argument v, qui peut être
;;; de type quelconque
```


Fonction d'affichage

```
;;; ab-affichage: ArbreBinaire[α] -> Paragraphe
;;; (ab-affichage B) renvoie B sous la forme d'un paragraphe.
;;; les indentations du paragraphe reflètent la hiérarchie de l'arbre
;;; un noeud à profondeur p est précédé de p tirets
```

- si B est l'arbre vide, renvoyer le paragraphe réduit à la ligne `"vide"`
- sinon renvoyer le paragraphe formé de
  - l'étiquette de l'arbre
  - le paragraphe représentant le sous-arbre gauche de B, dont toutes les lignes sont précédées d'un tiret
  - le paragraphe représentant le sous-arbre droit de B, dont toutes les lignes sont précédées d'un tiret

Appel à une fonction interne, récursive, qui étant donnés un arbre et une chaîne, affiche un paragraphe représentant l'arbre « préfixé par » cette chaîne.

```
;;; aff-Aux: Ligne * ArbreBinaire[α] -> Paragraphe
;;; (aff-Aux pref B) rend le paragraphe obtenu en préfixant par la chaine
;;; «pref» chaque ligne du paragraphe représentant l'affichage de B.
(define (aff-Aux pref B)
  (if (ab-noeud? B)
      (paragraphe-cons
       (string-append pref (->string (ab-etiquette B)))
       (let ((pref2 (string-append pref "-")))
         (paragraphe-append
          (aff-Aux pref2 (ab-gauche B))
          (aff-Aux pref2 (ab-droit B)))))
      (paragraphe (list (string-append pref "vide")))))
;;; ab-affichage: ArbreBinaire[α] -> Paragraphe
;;; (ab-affichage B) renvoie B sous la forme d'un paragraphe.
(define (ab-affichage B)
  (aff-Aux "" B))
```


Avec fonction interne

```
;;; ab-affichage: ArbreBinaire[α] -> Paragraphe
;;; (ab-affichage B) renvoie B sous la forme d'un paragraphe.
(define (ab-affichage B)
  ;; aff-Aux: Ligne * ArbreBinaire[α] -> Paragraphe
  ;; (aff-Aux pref B) rend le paragraphe ...
  (define (aff-Aux pref B)
    (if (ab-noeud? B)
        (paragraphe-cons
         (string-append pref (->string (ab-etiquette B)))
         (let ((pref2 (string-append pref "-")))
           (paragraphe-append
            (aff-Aux pref2 (ab-gauche B))
            (aff-Aux pref2 (ab-droit B)))))
        (paragraphe (list (string-append pref "vide")))))
  (aff-Aux "" B))
```

Une meilleure solution

Pas de fonction interne récursive

- si B est l'arbre vide, renvoyer le paragraphe réduit à la ligne `"vide"`
- sinon renvoyer le paragraphe formé de
  - l'étiquette de l'arbre
  - le paragraphe obtenu en faisant précéder d'un tiret toutes les lignes du paragraphe représentant le sous-arbre gauche de B,
  - le paragraphe ... le sous-arbre droit de B

Avec tiret

Faire précéder toutes les lignes par un tiret :

```
(map tiret ListeLignesParagrapheSousArbre)
```

```
;;; tiret: Ligne -> Ligne
;;; (tiret s) ajoute un tiret au debut de la ligne s
(define (tiret s)
  (string-append "-" s) )
```


Implantation

```
;;; ab-affichage: ArbreBinaire[α] -> Paragraphe
(define (ab-affichage B)
  (if (ab-noeud? B)
      (paragraphe
       (cons
        (->string (ab-etiquette B))
        (append
         (map tiret (lignes (ab-affichage (ab-gauche B))))
         (map tiret (lignes (ab-affichage (ab-droit B)))) ) ) )
      (paragraphe (list "vide")) ) )
```