PLAN DU COURS 10

« Programmation récursive » - Saison 2

Structures Arborescentes : arbres généraux

- Définition et exemples
- Barrière d'abstraction des arbres généraux
- Schéma de récursion sur les arbres généraux
- Exemples : nombre de nœuds, profondeur, liste préfixe, affichage

DÉFINITION D'UN ARBRE GÉNÉRAL

Dans un arbre général, chaque nœud porte une information (étiquette de type α) et a un nombre quelconque de descendants immédiats.

Le type est noté `ArbreGeneral[`α`]`

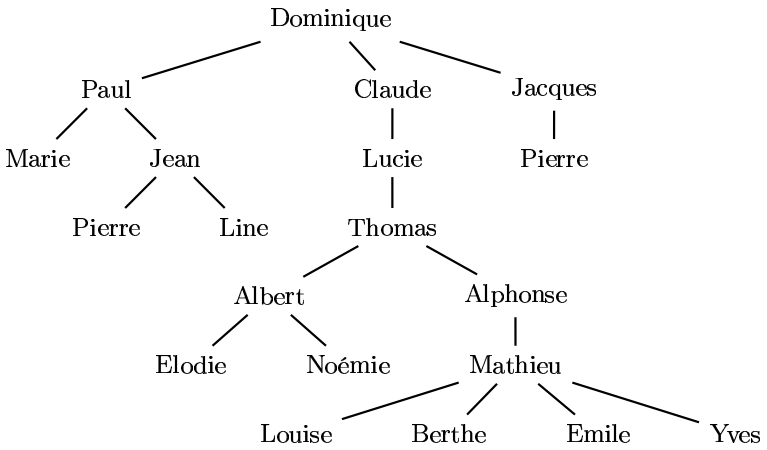
Définition (récursive) Un arbre général est formé

- d'un nœud (portant une étiquette de type α)
- et d'une liste de sous-arbres de type `ArbreGeneral[`α`]`

Remarque : une liste d'arbres généraux est aussi appelée *forêt*, soit pour les types : `Foret[`α`]` $\equiv$ `LISTE[ArbreGeneral[`α`]]`

Il n'y a pas d'arbre général vide, mais une forêt peut être vide (la liste vide)

EXEMPLE



BARRIÈRE D'ABSTRACTION DES ARBRES GÉNÉRAUX

- **Constructeur** pour construire un arbre général : `ag-noeud`
- **Accesseurs** pour accéder aux parties d'un arbre général : `ag-etiquette` et `ag-foret`
- **Reconnaisseur** inutile, puisqu'il n'y a qu'un seul constructeur

On manipule les arbres généraux uniquement à travers leur **barrière d'abstraction**. On ne connait que la **spécification** des fonctions.

Remarque : les forêts sont des listes → utiliser les primitives sur les listes

Spécification du constructeur

```
;;; ag-noeud: α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] = LISTE[ArbreGeneral[α]]
;;; (ag-nœud e foret) rend l'arbre formé de la racine d'étiquette «e» et,
;;; comme sous-arbres immédiats, les arbres de la forêt «foret».
```

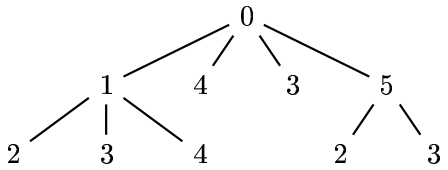
Exemples : arbres généraux de nombres

```
(ag-noeud 3 '())}    → #<object>
```

et construit l'arbre avec un unique nœud d'étiquette 3.


```
(let* ((g1 (ag-noeud 2 '()))
       (g2 (ag-noeud 3 '()))
       (g3 (ag-noeud 4 '()))
       (g4 (ag-noeud 1 (list g1 g2 g3)))
       (g5 (ag-noeud 0 (list g4 g3 g2 (ag-noeud 5 (list g1 g2))))))
  g5)      → #<object>
```

et construit l'arbre dont la représentation graphique est :



Spécification des accesseurs

```
;;; ag-etiquette: ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine de l'arbre «g».

;;; ag-foret: ArbreGeneral[α] -> Foret[α]
;;; avec Foret[α] = LISTE[ArbreGeneral[α]]
;;; (ag-foret g) rend la forêt des sous-arbres immédiats de «g».

(let* ((g1 (ag-noeud 2 '()))
       (g2 (ag-noeud 3 '()))
       (g3 (ag-noeud 4 '()))
       (g4 (ag-noeud 1 (list g1 g2 g3))))
  (ag-etiquette (cadr (ag-foret g4))))     → ???
```


Propriétés

- Pour toute forêt d'arbres généraux F, et toute valeur v

```
(ag-etiquette (ag-noeud v F)) ≡ v
(ag-foret (ag-noeud v F))     ≡ F
```

- Pour tout arbre général G

```
(ag-noeud (ag-etiquette G)(ag-foret G)) ≡ G
```

Reconnaisseur dérivé

Définir le prédicat `ag-feuille?`

```
;;; ag-feuille?: ArbreGeneral[α] -> bool
;;; (ag-feuille? G) rend vrai ssi G est une feuille
(define (ag-feuille? G)
  (not (pair? (ag-foret G))) )
```


Une fonction d'affichage

La barrière d'abstraction contient aussi une fonction d'affichage

```
;;; ag-expression: ArbreGeneral[α] -> Sexpression
;;; (ag-expression G) rend une Sexpression construisant l'arbre G
(let* ((g1 (ag-noeud 2 '()))
       (g2 (ag-noeud 3 '()))
       (g3 (ag-noeud 4 '()))
       (g4 (ag-noeud 1 (list g1 g2 g3))))
  (ag-expression g4))
                              →
          (ag-noeud 1
                    (list (ag-noeud 2 (list))
                          (ag-noeud 3 (list))
                          (ag-noeud 4 (list))))
```


Récursion sur les arbres généraux

Un arbre général est constitué :

- d'une étiquette
- et d'une forêt (liste) d'arbres généraux

Pour définir une fonction sur un arbre, il faut traiter la liste (forêt) de ses sous-arbres, et pour définir une fonction sur une forêt (liste), il faut traiter chacun des arbres la composant => Récursion Croisée

On peut aussi exploiter le fait qu'une forêt est une LISTE d'arbres en utilisant les itérateurs map et reduce

Schéma récursif sur les arbres généraux

```
;;; ArbreRec: ArbreGeneral[alpha] -> ...
(define (ArbreRec G)
   (combinaison (ag-etiquette G)
                (ForetRec (ag-foret G))))

;;; ForetRec: LISTE[ArbreGeneral[alpha]] -> ...
(define (ForetRec F)
   (if (pair? F)
     (combinaison (ArbreRec (car F))
                  (ForetRec (cdr F)))
     base))
```

```
(define (ForetRec F)
  (reduce combinaison base (map ArbreRec F)))

(define (ArbreRec G)
  (combinaison (ag-etiquette G)
               (reduce combi
                       base
                       (map ArbreRec (ag-foret G)))))
```


Nombre de nœuds

```
;;; nombre-noeuds-arbre: ArbreGeneral[α] -> nat
;;; (nombre-noeuds-arbre G) rend le nombre de noeuds de G
(define (nombre-noeuds-arbre G)
  ;; nombre-noeuds-foret: Foret[α] -> nat
  ;; (nombre-noeuds-foret F) rend le nombre de noeuds de F
  (define (nombre-noeuds-foret F)
    (if (pair? F)
        (+ (nombre-noeuds-arbre (car F))
           (nombre-noeuds-foret (cdr F)))
        0 ) )
  (+ 1 (nombre-noeuds-foret (ag-foret G))) )
```


Exemple

```
(let* ((g1 (ag-noeud 2 '()))
       (g2 (ag-noeud 3 '()))
       (g3 (ag-noeud 4 '()))
       (g4 (ag-noeud 1 (list g1 g1 g2 g2 g3 g3))))
  (nombre-noeuds-arbre g4))
```


Avec un map

```
;;; nombre-noeuds-arbre: ArbreGeneral[α] -> nat
;;; (nombre-noeuds-arbre G) rend le nombre de nœuds de G
(define (nombre-noeuds-arbre G)
  ;; nombre-noeuds-foret: Foret[α] -> nat
  ;; (nombre-noeuds-foret F) rend le nombre de nœuds de F
  (define (nombre-noeuds-foret F)
    (reduce + 0 (map nombre-noeuds-arbre F)))
  (+ 1 (nombre-noeuds-foret (ag-foret G))))
```

ou encore :

```
(define (nombre-noeuds-arbre G)
  (reduce + 1 (map nombre-noeuds-arbre (ag-foret G))))
```

Profondeur

```
;;; ag-profondeur: ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
(define (ag-profondeur G)
  ;; profondeurForet: Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F
  (define (profondeurForet F)
    (if (pair? F)
        (max (ag-profondeur (car F))
             (profondeurForet (cdr F)))
        0))
  (+ 1 (profondeurForet (ag-foret G))))
```

ou avec des itérateurs :

```
(define (ag-profondeur G)
  (+ 1 (reduce max 0 (map ag-profondeur (ag-foret G)))))
```


UPMC MIAS1-Info1 Année 2003-2004 « Programmation récursive » Cours 10 Anne Brygoo, Michèle Soria

Liste préfixe

```
;;; ag-liste-prefixe: ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe des étiquettes de l'arbre G
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret: Foret[α] -> LISTE[α] rend la concaténation
  ;; des listes préfixes des étiquettes des arbres de F
  (define (liste-prefixe-foret F)
    (if (pair? F)
        (append (ag-liste-prefixe (car F))
                (liste-prefixe-foret (cdr F)))
        '()))
    ; (reduce append '() (map ag-liste-prefixe F)))
  (cons (ag-etiquette G)
        (liste-prefixe-foret (ag-foret G))))
```


UPMC MIAS1-Info1 Année 2003-2004 « Programmation récursive » Cours 10 Anne Brygoo, Michèle Soria

Avec des itérateurs

```
(define (ag-liste-prefixe G)
  (cons (ag-etiquette G)
        (reduce append
                '()
                (map ag-liste-prefixe (ag-foret G)))))
```

Affichage en paragraphe indenté

;;; *ag-affichage: ArbreGeneral[α] -> Paragraphe*
;;; *(ag-affichage G) rend le paragraphe dont la première ligne est l'étiquette*
;;; *de la racine de l'arbre «G» et dont les lignes suivantes sont égales à la*
;;; *représentation de la suite des sous-arbres de «G», toutes les lignes étant*
;;; *préfixées par un tiret*

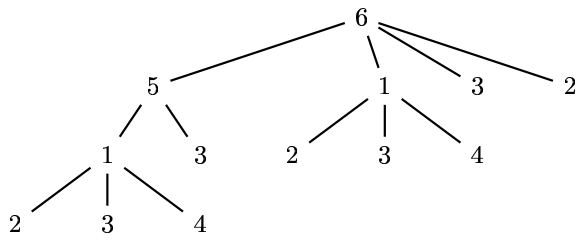
UTILISE

;;; *tiret: Ligne -> Ligne*
;;; *(tiret s) ajoute un tiret au debut de la ligne s*

;;; *paragraphe-append: Paragraphe * Paragraphe -> Paragraphe*
;;; *(paragraphe-append P1 P2) rend la concaténation de P1 et P2*

UPMC MIAS1-Info1 Année 2003-2004 « Programmation récursive » Cours 10 Anne Brygoo, Michèle Soria

```
"
6
-5
--1
---2
---3
---4
--3
-1
--2
--3
--4
-3
-2
"
```



MIAS1-Info1 Année 2003-2004 « Programmation récursive » Cours 10 Anne Brygoo, Michèle Soria

UPMC MIAS1-Info1 Année 2003-2004 « Programmation récursive » Cours 10 Anne Brygoo, Michèle Soria

```
;;; ag-affichage: ArbreGeneral[α] -> Paragraphe
;;; (ag-affichage G) rend ...
(define (ag-affichage G)
  ;; ag-affichage-foret: Foret[α] -> Paragraphe
  ;; (ag-affichage-foret F) rend ...
  (define (ag-affichage-foret F)
    (if (pair? F)
        (paragraphe-append (ag-affichage (car F))
                           (ag-affichage-foret (cdr F)))
        (paragraphe '())))
  ;; expression de (ag-affichage G)
  (paragraphe (cons
               (->string (ag-etiquette G))
               (map tiret
                    (lignes (ag-affichage-foret (ag-foret G)))))))
```

Une autre version

avec itérateurs pour `ag-foret` :

```
(define (ag-affichage G)
  ;; ag-affichage-foret: Foret[α] -> Paragraphe
  ;; (ag-affichage-foret F) rend ...
  (define (ag-affichage-foret F)
    (reduce paragraphe-append
            (paragraphe '())
            (map ag-affichage F)))
  ;; expression de (ag-affichage G)
  (paragraphe
   (cons
    (->string (ag-etiquette G))
    (map tiret
         (lignes (ag-affichage-foret (ag-foret G)))))))
```


Une troisième version (sans ag-foret)

```
(define (ag-affichage G)
  (paragraphe
   (cons
    (->string (ag-etiquette G))
    (map tiret
         (lignes (reduce paragraphe-append
                         (paragraphe '())
                         (map ag-affichage (ag-foret G))))))))
```


Un autre algorithme

```
(define (ag-affichage G)
  ;; aff-foret: Ligne * ArbreGeneral[α] -> Paragraphe
  ;; (aff-foret pref F) rend le paragraphe obtenu en préfixant chaque ligne
  ;; de la représentation des arbres de «F» par la chaîne «pref»
  (define (aff-foret pref F)
    ; aff-Aux-pref: ArbreGeneral[α] -> Paragraphe
    ; (aff-Aux-pref G) rend le paragraphe obtenu en préfixant chaque ligne
    ; de (ag-affichage G) par la chaîne «pref»
    (define (aff-Aux-pref G)
      (aff-Aux pref G))
    ; expression de (aff-foret pref F)
    (reduce paragraphe-append
            (paragraphe '())
            (map aff-Aux-pref F)))
```

```
;; aff-Arbre: Ligne * ArbreGeneral[α] -> Paragraphe
;; (aff-Arbre pref G) rend le paragraphe obtenu en préfixant chaque ligne
;; de (ag-affichage G) par la chaîne «pref»
(define (aff-Aux pref G)
  (paragraphe-cons
   (string-append pref (->string (ag-etiquette G)))
   (aff-foret (string-append pref "-")
              (ag-foret G))))

;; expression de (ag-affichage G)
(aff-Aux "" G))
```