UE LI101

Université Pierre et Marie Curie
Programmation récursive
Cours 10

A.Brygoo, C.Queinnec, M.Soria, P.Manoury

PLAN du COURS 10

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, etc.

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.

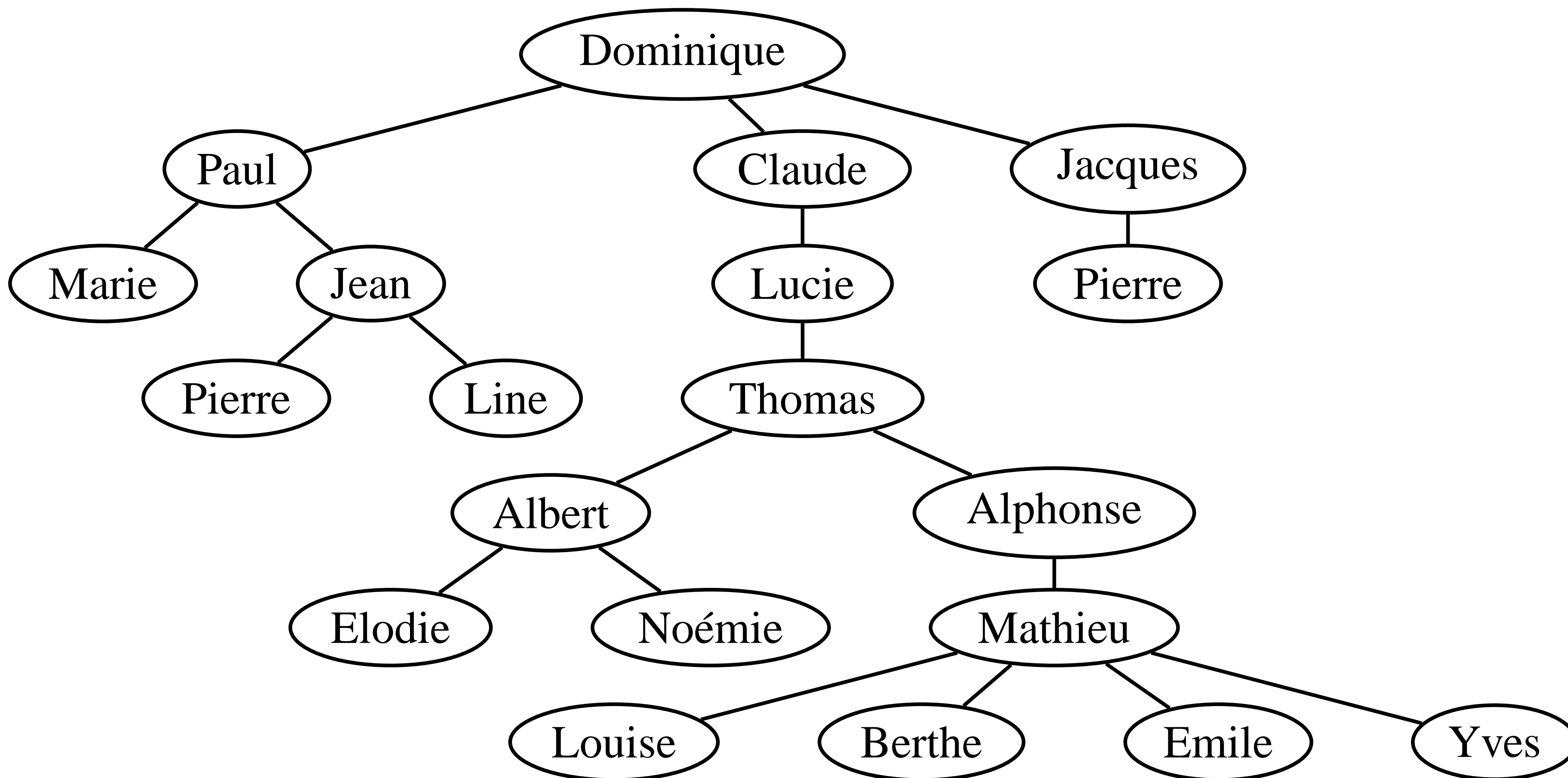
Définition (récursive) Un arbre général est formé

- d'un *nœud* portant une *étiquette* d'un type donné
- et d'une liste de sous-arbres généraux appelée *forêt*

Remarques :

1. il n'y a pas d'arbre général vide
2. une forêt peut être vide

Exemple



Barrière d'abstraction des arbres généraux

Le type est noté des arbres généraux (à étiquettes de type α est noté

`ArbreGeneral[α]`

On pose

$$\texttt{Foret}[\alpha] \equiv \texttt{LISTE}[\texttt{ArbreGeneral}[\alpha]]$$

- **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

Remarque : les forêts sont des listes donc sont manipulées avec les primitives sur les listes

Spécification du constructeur

Un seul constructeur

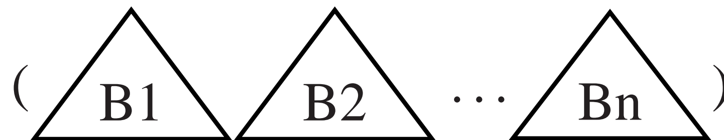
; ; ; ***ag-noeud:*** *α * Foret[α] -> ArbreGeneral[α]*

; ; ; ***(ag-noeud e F)*** *rend l'arbre formé de la racine d'étiquette*

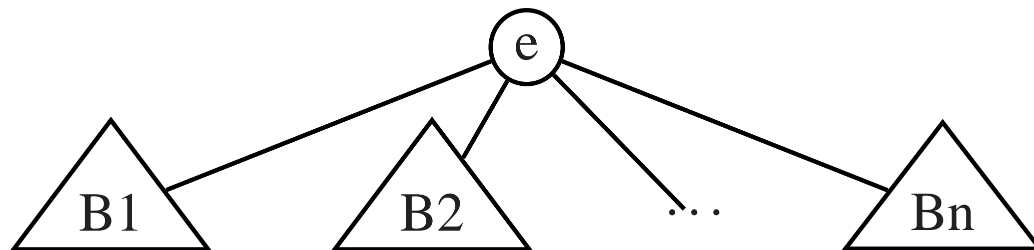
; ; ; ***e*** *et, comme sous-arbres immédiats, les arbres de la*

; ; ; *forêt* ***F****.*

Si la forêt `F` est



On obtient l'arbre général

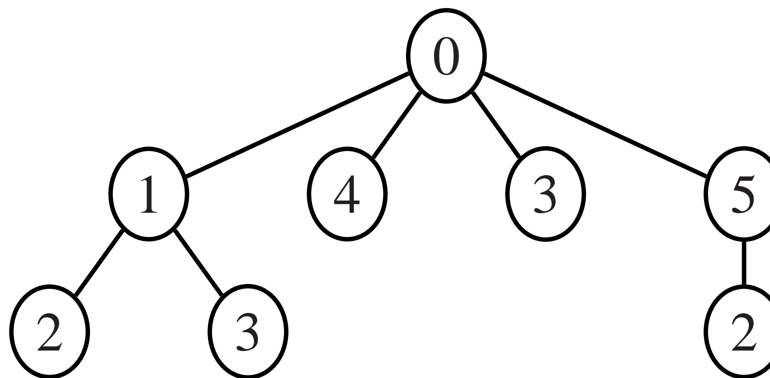


Exemple

Expression scheme :

```
(ag-noeud 0
          (list
            (ag-moeud 1 (list (ag-noeud 2 (list))
                              (ag-noeud 3 (list))))
            (ag-noeud 4 (list))
            (ag-noeud 5 (list (ag-noeud 6 (list))))))
```

Représentation graphique :



Spécification des accesseurs

Étiquette de la racine

;;; ***ag-etiquette:*** *ArbreGeneral[α] -> α*

;;; ***(ag-etiquette G)*** *rend l'étiquette de la racine de l'arbre* ***G***.

Forêt associée à un nœud

;;; ***ag-foret:*** *ArbreGeneral[α] -> Foret[α]*

;;; ***(ag-foret G)*** *rend la forêt des sous-arbres immédiats de* ***G***.

Propriétés algébriques

- 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
```

Constructeur/reconnaisseur dérivés

Feuille : arbre général dont la forêt est vide.

Pseudo constructeur feuille

```
;;; ag-feuille: α -> ArbreGeneral[α]
;;; (ag-feuille e) construit la feuille d'étiquette e
(define (ag-feuille e)
   (ag-noeud e (list)))
```

Pseudo reconnaissuer associé

```
;;; 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

Exemple

```
(ag-expression
  (ag-noeud 1 (list (ag-feuille 2)
                    (ag-feuille 3))))
```

```
→ (ag-noeud 1 (list (ag-noeud 2 (list))
                    (ag-noeud 3 (list))))
```

Récursion sur les arbres généraux

Un arbre général est constitué :

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

Traitement d'un arbre :

- Pour **traiter un arbre**, il faut **traiter la forêt** de ses descendants directs.
- Pour **traiter la forêt**, il faut **traiter chaque arbre** de cette liste.

$\Rightarrow$ **récursion croisée** $\Leftarrow$

Schéma récursif sur les arbres généraux

Sur les arbres

```
;;;ArbreRec: ArbreGeneral[α] -> β
(define (ArbreRec G)
   (combinaison1 (ag-etiquette G)
                 (ForetRec (ag-foret G)) ) )
```

Sur les forêts

```
;;;ForetRec: Foret[α] -> β
(define (ForetRec F)
   (if (pair? F)
     (combinaison2 (ArbreRec (car F))
                   (ForetRec (cdr F)) )
     base ) )
```

Nombre de nœuds d'un arbre général

Arbre : 1 + nombre de nœuds de la forêt

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

Forêt : somme des nombres de nœud des arbres de la forêt

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

Profondeur

Arbre : 1 + max des profondeurs des arbres de la forêt

```
;;; ag-profondeur: ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de l'arbre «G»
(define (ag-profondeur G)
  (+ 1 (profondeurForet (ag-foret G))))
```

Forêt : max des profondeurs des arbres de la forêt

```
;;; 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))
```

Liste préfixe

Arbre

```
;;; 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)
  (cons (ag-etiquette G) (liste-prefixe-foret (ag-foret G)) )
```

Forêt

```
;;; liste-prefixe-foret: Foret[α] -> LISTE[α]
;;; (liste-prefixe-foret F) concaténation des listes préfixes des étiquettes de F
(define (liste-prefixe-foret F)
  (if (pair? F)
      (append (ag-liste-prefixe (car F))
              (liste-prefixe-foret (cdr F)))
      '() ) )
```

Schéma récursif sur les forêts

```
;;;ForetRec: Foret[α] -> β
(define (ForetRec F)
   (if (pair? F)
      (combinaison2 (ArbreRec (car F)) (ForetRec (cdr F)) )
      base ) )
```

Soit

```
;;;combinaison3: ArbreGeneral[α] * β -> β
(define (combinaison3 G r)
   (combinaison2 (ArbreRec G) r))
```

On peut utiliser la fonctionnelle `reduce`

```
;;;ForetRec: Foret[α] -> β
(define (ForetRec F)
   (reduce combinaison3 base F))
```

Schéma récursif sur les arbres généraux (II)

Avec **fonctionnelle** et **définition locale**

```
;;;ArbreRec: ArbreGeneral[α] -> β
(define (ArbreRec G)
  ;;fonction-foret: ArbreGeneral[α] * β -> β
  (define (fonction-foret G r)
     (combinaison2 (ArbreRec G) r))
  ;;corps de la définition de ArbreRec
  (combinaison1 (ag-etiquette G)
                (reduce fonction-foret Base (ag-foret G))))
```

Récurrence croisée **«interne»**

Liste préfixe (II)

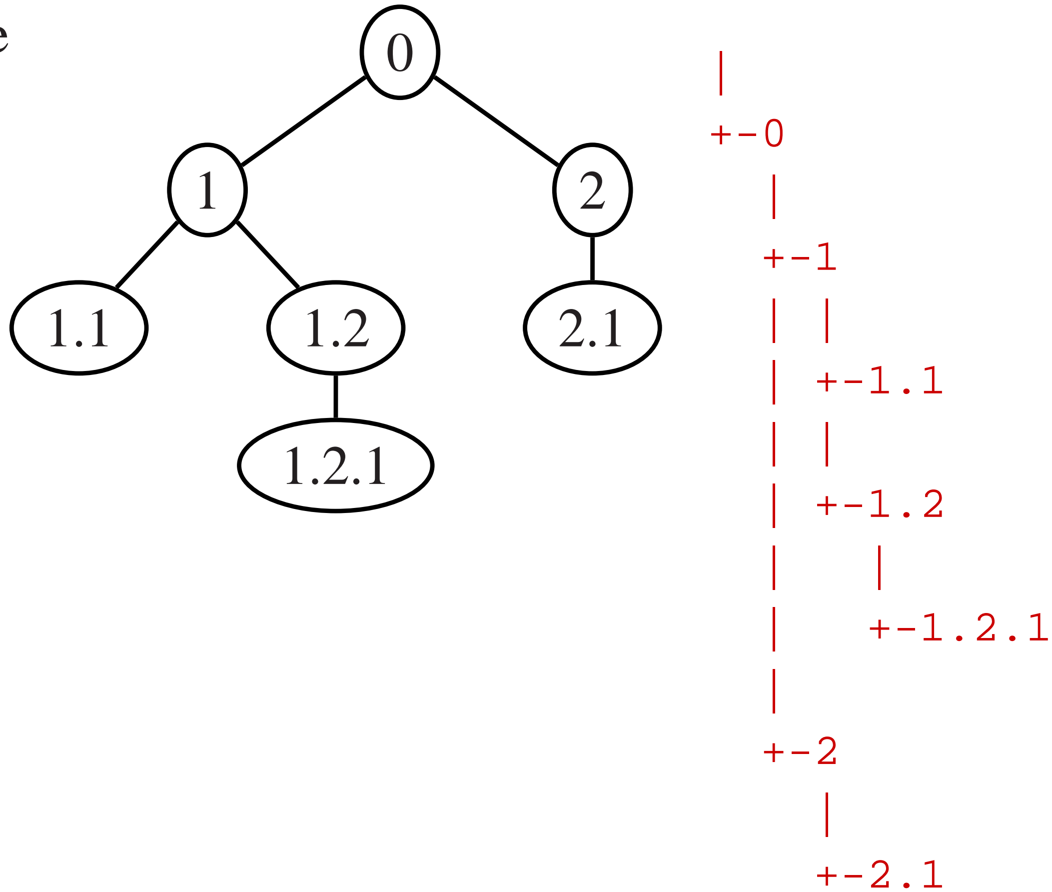
Redéfinition de la liste préfixe

```
(define (liste-prefixe G)

  (define (append-prefixes G r)
     (append (liste-prefixe G) r))

  (cons (ag-etiquette G)
        (reduce append-prefixes (list) (ag-foret G))))
```

Un exemple moins simple

Affichage d'arbre général en ASCII ART

Exemple



Analyse

- l'arbre est représenté par une chaîne de caractères structurée en une **suite de lignes**
  - une ligne est une chaîne de caractères terminée par un *retour-charriot* (`"\n"`).
- il y a 2 sortes de lignes
  - celles qui portent une étiquette (`"  +-1"`)
  - celles qui font le *lien* avec les lignes précédentes (`"  |  |"`)
- chaque étiquette est *décalée* en fonction de sa profondeur

But : construire une chaîne de caractères constitué d'une alternance de lignes «lien» et de lignes «étiquette» qui repésente la structure d'un arbre général.

Premier essai

Schéma de programme : arbres généraux ⇒ récurrence croisée

Un paramètre pour le décalage (dec)

Sur les arbres (G) :

```
(define (lignes-arbre dec G)
   (string-append (ligne-lien dec)
                  (ligne-etiquette dec (ag-etiquette G))
                  (lignes-foret (decaler dec) (ag-foret G))))
```

Sur les forêts (F) :

```
(define (lignes-foret dec F)
  (if (pair? F)
      (string-append (lignes-arbre dec (car F))
                     (lignes-foret dec (cdr F)))
      ""))
```

Analyse (II)

Ligne «étiquette» : décalage + `"+-"` + étiquette + retour charriot

Ligne «lien» : décalage + `"|"` + retour-charriot

Le décalage :

hétérogène : des caractères blancs (espaces)
ou des caractères de liaison (barre verticale)

changeant : dépend de la profondeur
$\Rightarrow$ ajouter devant les arbres d'une forêt

Espace ou barre : dépend de la place de l'arbre dans une forêt

– ajouter un espace le *dernier* d'une forêt
– ajouter une barre sinon

$\Rightarrow$ Paramètre supplémentaire (`der`) : place dans une forêt

Programme (I)

Fonctions auxiliaires

```
;;; ligne-etiquette: string*string -> string
(define (ligne-etiquette dec e)
  (string-append dec "+-" e "\n"))

;;; ligne-etiquette: string*string -> string
(define (ligne-lien dec)
  (string-append dec "|\n"))

;;; ligne-etiquette: bool*string -> string
(define (decaler der dec)
  (if der
      (string-append dec "   ")
      (string-append dec "|  ")))
```

Programme (II)

```
;;; lignes-arbre: bool * string * ArbreGeneral[string] -> string
(define (lignes-arbre der dec G)
  (string-append (ligne-lien dec)
                 (ligne-etiquette dec (ag-etiquette G))
                 (lignes-foret (decaler der dec) (ag-foret G))
;;; lignes-foret: string * Foret[string] -> string
(define (lignes-foret dec F)
  (if (pair? F)
      (string-append (lignes-arbre (der? F) dec (car F))
                     (lignes-foret dec (cdr F)))
      ""))
```

Avec

```
(define (der? F) (not (pair? (cdr F))))
```