PLAN DU COURS 11

« Programmation récursive » - Saison 2

Arbres généraux et S-Expressions

- Une application des arbres généraux : le système de fichiers
  - Barrière d'abstraction des descripteurs
  - Algorithmes de quelques commandes d'un système de fichiers
- S-Expression
  - Définition d'une S-Expression
  - Evaluation d'un expression arithmétique

SYSTÈMES DE FICHIERS

Systèmes de fichiers sous Linux, Windows ou Mac

- Répertoires (dossiers) et fichiers
- Commandes intéressantes : `du`, `ll`, `find`

Un système de fichiers est représenté par un arbre général d'étiquettes de type `Descripteur` :

`Systeme` = `ArbreGeneral[Descripteur]`

BARRIÈRE D'ABSTRACT. DE Descripteur

Constructeurs

```
;;; fichier: string * nat -> Descripteur
;;; (fichier nom t) rend le descripteur du fichier «nom» de taille «t»

;;; repertoire: string -> Descripteur
;;; (repertoire nom) rend le descripteur du répertoire «nom»
```

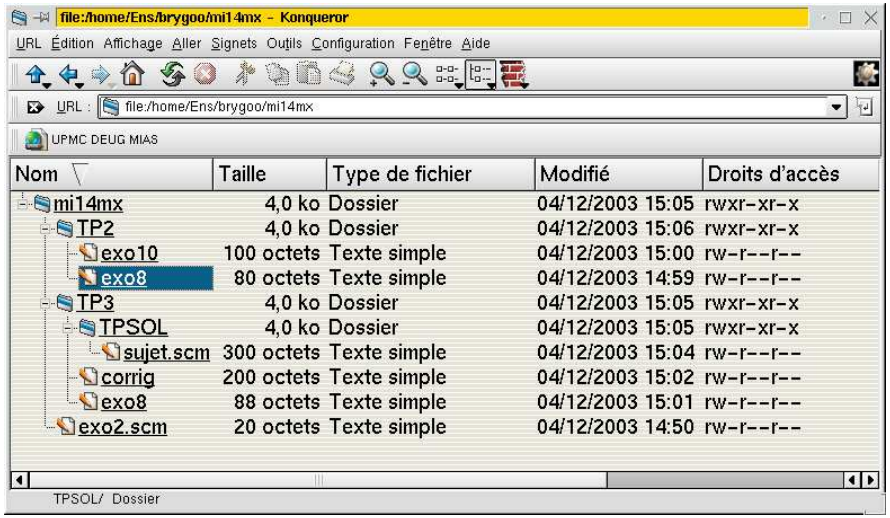
Reconnaisseur

```
;;; fichier?: Descripteur -> bool
;;; (fichier? desc) vérifie si «desc» est le descripteur d'un fichier
```


Accesseur

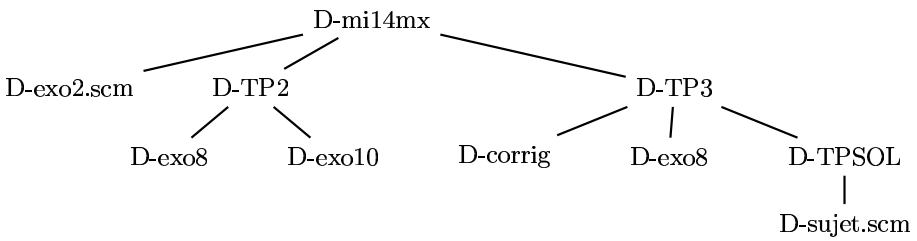
```
;;; nom: Descripteur -> string
;;; (nom desc) rend le nom du répertoire ou du fichier dont le
;;; descripteur est «desc»

;;; taille: Descripteur -> nat
;;; ERREUR lorsque le descripteur correspond à un répertoire
;;; (taille desc) rend la taille du fichier dont le descripteur est «desc»
```

Implantation de la barrière d'abstraction : à vous de la faire !
(avec des N-uplets ou des vecteurs)




```
(define (ex1-fichier)
  (ag-noeud
   (repertoire "mi14mx")
   (list (ag-noeud (fichier "exo2.scm" 20)'())
         (ag-noeud
          (repertoire "TP2")
          (list (ag-noeud (fichier "exo8" 80) '())
                (ag-noeud (fichier "exo10" 100) '())))
         (ag-noeud
          (repertoire "TP3")
          (list (ag-noeud (fichier "corrig" 200) '())
                (ag-noeud (fichier "exo8" 88) '())
                (ag-noeud
                 (repertoire "TPSOL")
                 (list (ag-noeud (fichier "sujet.scm" 300) '()))))))))
```


Visualisation d'un systeme de fichiers



Taille d'un répertoire

```
;;; du-s: Systeme -> nat
;;; (du-s systeme) rend la quantité d'espace disque utilisée par les fichiers
;;; du système «systeme»
(define (du-s systeme)
  (if (fichier? (ag-etiquette systeme))
      (taille (ag-etiquette systeme))
      (reduce + 0 (map du-s (ag-foret systeme)))))
```

`(du-s (ex1-fichier))` → 788

Contenu d'un répertoire

```
;;; ll: Systeme -> Paragraphe
;;; (ll systeme) rend le paragraphe contenant, pour chaque sous-éléments
;;; immédiats de «systeme» une ligne formée:
;;; pour un fichier, de sa taille, d'une tabulation et de son nom
;;; pour un répertoire, de «-», d'une tabulation, de son nom suivi de «/»

(ll (ex1-fichier)) →


"
20      exo2.scm
--      TP2/
--      TP3/
"
```



```
(define (ll systeme)
  ;; desc-ll: Descripteur -> Ligne
  ;; (desc-ll descripteur) rend l'image du descripteur «descripteur»
  (define (desc-ll descripteur)
    (string-append (->string (if (fichier? descripteur)
                                 (taille descripteur)
                                 "--"))
                   (string #\tab)
                   (nom descripteur)
                   (if (fichier? descripteur) "" "/")))

  ;; expression de (ll systeme):
  (paragraphe (map desc-ll
                   (map ag-etiquette (ag-foret systeme)))))
```


12 Recherche dans un répertoire

```
;;; find: string * Systeme -> Paragraphe
;;; (find ident systeme) rend le paragraphe dont les lignes sont constituées
;;; par les noms complets des fichiers ou répertoires du système «systeme»
;;; dont le nom est «ident».

(find "exo8" (ex1-fichier)) →


"
mi14mx/TP2/exo8
mi14mx/TP3/exo8
"
```

```
(define (find ident systeme)
  ;; find-ss-systeme: string * Systeme -> Paragraphe
  ;; (find-ss-systeme path-ss-systeme ss-systeme) a la même sémantique que
  ;; «find» mais, en plus, chaque ligne est préfixée par «path-ss-systeme»,
  ;; le chemin d'accès au «ss-systeme»
  (define (find-ss-systeme path-ss-systeme ss-systeme)
     ...

  ;; find-foret: string * Foret[Descripteur] -> Paragraphe
  ;; (find-foret path-ss-systeme F) rend le paragraphe obtenu en
  ;; concaténant, pour tout arbre «A» de la forêt «F»,
  ;; la valeur de (find-ss-systeme path-ss-systeme A)
  (define (find-foret path-ss-systeme F)
    ...

  ;; expression de (find ident systeme):
```



```
(find-ss-systeme "" systeme))
```



```
  (define (find-ss-systeme path-ss-systeme ss-systeme)
        ...

  ;; find-foret: string * Foret[Descripteur] -> Paragraphe
  ;; (find-foret path-ss-systeme F) rend le paragraphe obtenu en concaténant,
  ;; pour tout arbre «A» de la fôret «F», (find-ss-systeme path-ss-systeme A)
  (define (find-foret path-ss-systeme F)

    ;; find-ss-systeme-path: ArbreGeneral[Descripteur] -> Paragraphe
    ;; (find-ss-systeme-path G) même sémantique que «find-ss-systeme»
    ;; mais fonction rendue unaire pour utilisation dans un map
    (define (find-ss-systeme-path G)
      (find-ss-systeme path-ss-systeme G))
```



```
(reduce paragraphe-append (paragraphe '())
        (map find-ss-systeme-path F)))
```

```
(define (find-ss-systeme path-ss-systeme ss-systeme)
    (paragraphe-append

     (if (equal? ident (nom (ag-etiquette ss-systeme)))
         (paragraphe (list (string-append path-ss-systeme
                                          ident)))
         (paragraphe '()))

     (if (not (fichier? (ag-etiquette ss-systeme)))
         (find-foret
          (string-append path-ss-systeme
                         (nom (ag-etiquette ss-systeme))
                         "/")
          (ag-foret ss-systeme))
         (paragraphe '()))))
```


Implantation des arbres généraux

Il existe différentes possibilités pour implanter la barrière d'abstraction des arbres généraux (c'est-à-dire écrire les définitions des fonctions constituant la barrière) :

- en utilisant les vecteurs (cf. sur le cédérom)
- en utilisant les S-expressions

S-expressions

L'expression Scheme

```
(* (+ (* 90 10) 100)
   (+ 20 80)
   36)
```

ressemble à une liste (de quatre éléments) de type `LISTE[`α`]` mais les éléments de la liste ne sont pas de même type.

C'est une **S-expression**

Définition d'une S-expression

<*Sexpression*> → <*atome*>
`LISTE[`<*Sexpression*>`]`

<*atome*> → <*Nombre*>
<*bool*>
<*string*>
<*Symbole*>

(<*Nombre*>, <*bool*>, <*string*> et <*Symbole*> sont définis dans la carte de référence)

Implantation par les S-expressions

Un arbre général peut être représenté par une S-expression dont le premier élément est l'étiquette de la racine et dont le reste est constitué de la forêt de ses sous-arbres immédiats.

```
(define (ag-noeud e F)
  (cons e F))

(define (ag-etiquette g)
  (car g))

(define (ag-foret g)
  (cdr g))
```


Pourquoi avoir une barrière d'abstraction pour les arbres généraux ? Pour s'abstraire des listes, pour se concentrer sur la récursion dans les arbres généraux.

Expression arithmétique

Les expressions arithmétiques, bien formées, opérateurs `+` et `*`

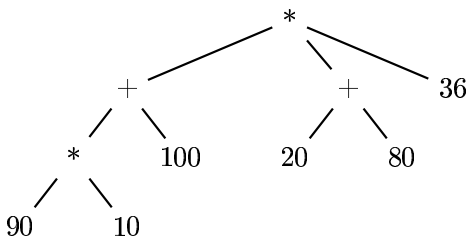
`(((90 * 10) + 100) * (20 + 80) * 36)`

- Visualisation et représentation par un arbre général (type `ArbreExpr`)
- Représentation par une S-expression (type `SchemeExpr`)

```
(* (+ (* 90 10) 100) (+ 20 80) 36)
```

- Évaluer (opérateurs `+`, `*` et constantes)
- Évaluer dans un **environnement** (opérateurs `+`, `*`, constantes et variables auxquelles sont associées des valeurs).

Visualisation par un arbre général



Arbre représentant une expression

```
(define (G-S)
  (ag-noeud '*
            (list (ag-noeud '+
                            (list (ag-noeud
                                   '*
                                   (list (ag-noeud 90 '())
                                         (ag-noeud 10 '())))
                                  (ag-noeud 100 '())))
                  (ag-noeud '+
                            (list (ag-noeud 20 '())
                                  (ag-noeud 80 '())))
                  (ag-noeud 36 '()))))
```


Définition de evaluationExprArbre

```
;;; evaluationExprArbre: ExprArbre-> Nombre
;;; (evaluationExprArbre G) rend la valeur de l'expression arithmétique
;;; représentée par G
(define (evaluationExprArbre G)
  (if (ag-feuille? G)
      (ag-etiquette G)
      (let ((LArgs (map evaluationExprArbre (ag-foret G))))
        (application (operation (ag-etiquette G)) LArgs))))
```


Définition de operation

```
;;; operation: Symbole -> Opération
;;; (operation operateur) rend la fonction correspondant au symbole
;;; «operateur»
(define (operation operateur)
  (cond ((equal? operateur '+) +)
        ((equal? operateur '*) *)
        (else (erreur 'operation operateur "pas défini"))))
```


Définition de application

```
;;; application: Opération * LISTE[Nombre]/non vide/ -> Nombre
;;; (application op args) rend le résultat de l'application de l'opération «op»
;;; à la liste d'arguments «args»
(define (application op args)
  (if (pair? (cdr args))
      (op (car args) (application op (cdr args)))
      (op (car args))))
```

Définition de evaluationExprS

```
;;; evaluationExprS: ExprScheme -> Nombre
;;; (evaluationExprS E) rend la valeur de l'expression arithmétique «E»
(define (evaluationExprS E)
  (if (list? E) ; pourquoi avons-nous mis list? et non pair?
      (let ((LArgs (map evaluationExprS (cdr E))))
        (application (operation (car E)) LArgs))
      E))


(evaluationExprS '(* (+ (* 90 10) 100) (+ 20 80) 36))
```

```
(* (+ (* 90 10) 100) (+ 20 80) 36)
```


Evaluation dans un environnement

L'environnement est représenté par une liste d'associations

```
(Symbole, Nombre)
```

Par exemple

```
(define (MonEnv) '((x 3) (y 4) (z 5)))
```

```
(valeur-de 'y (MonEnv)) -> 4
```

La fonction d'évaluation doit donc avoir 2 arguments :

- l'expression elle même
- et la liste d'associations représentant l'environnement.

Spécification et application

```
;;; evaluationExpr: ExprScheme * LISTE[N-UPLET[Symbole Nombre]] ->
Nombre
;;; (evaluationExpr E env) rend la valeur de l'expression arithmétique «E»
;;; dans l'environnement «env»


(define (MonEnv) '((x 3) (y 4) (z 5)))

(evaluationExpr '(+ x (* 3 y 5) (+ 1 2 y (* z 10)))
                (MonEnv))
```


Définition de evaluationExpr

```
(define (evaluationExpr E env)
  ;; evaluationExpr-env: ExprScheme -> Nombre
  ;; (evaluationExpr-env E): même sémantique que «evaluationExpr»
  ;; mais fonction rendue unaire pour utilisation dans un map
  (define (evaluationExpr-env E)
    (evaluationExpr E env))

  (if (list? E)
      (let ((LArgs  (map evaluationExpr-env (cdr E))))
        (application (operation (car E)) LArgs))
      (if (symbol? E)
          (valeur-de E env)
          E)))
```