PLAN DU COURS 7

« Programmation récursive »

- La citation
- La notion de n-uplet
- La notion de liste d'associations
- La notion de barrière d'abstraction

LISTES

En Scheme, les programmes sont essentiellement représentés par des S-expressions (ou expressions symboliques).

Les parenthèses (...) servent à noter en Scheme :

- des applications fonctionnelles,
- des définitions,
- des formes spéciales (`if`, `and`, `or`, `let`, ...).

Comment faire apparaître une liste dans un programme ?

EXPRESSION ET VALEUR D'UNE EXPRESSION

Faire la différence entre :
– **expression** : `(list 1 2 3)`
– **valeur** : `(1 2 3)` la liste formée des éléments 1, 2 et 3
Comment exprimer (*citer*) la **valeur** de l'**expression** `(list 1 2 3)` ?

- Syntaxe de la citation

  `(quote` *exp*`)` *ou* `'`*exp*

- La citation est une **forme spéciale** : on n'évalue pas l'argument mais on le retourne tel quel. La citation d'une expression signifie « J'ai pour valeur ce qui suit »

LA CITATION

- la citation d'une constante est la constante elle-même `(quote 2)` ≡`'2` ≡`2` →`2`
- la citation d'un symbole : `(quote a)` ≡`'a` →`a`
- la citation d'une liste est la liste des citations de ses éléments

```
(quote (e1 e2 e3)) ≡ '(e1 e2 e3) ≡ (list 'e1 'e2 'e3)
(quote ()) ≡ '() ≡ (list)
```

Exemples :

`(cons 'a '())` →`(a)`

`(quote (a1 a2))` ≡`'(a1 a2)` →`(a1 a2)`

`(quote (1 2 3))` ≡`'(1 2 3)` →`(1 2 3)`

Distinguer variable et symbole

Dans vos programmes Scheme, vous avez utilisé des symboles

- comme mots clef : `if`, `and`, `or`, `let`, `define`
- comme identificateurs : `*`, `map`, `n`, ...

Un symbole représente une variable mais on peut aussi manipuler les symboles pour eux-mêmes.


```
(let ((a 3))
  a)

(let ((a 3))
  (number? a) )

(let ((a 3))
  (symbol? a) )

(let ((a 3))
  'a)

(let ((a 3))
  (symbol? 'a) )
```


`(list 1 2 3)` →`(1 2 3)`

`(list a1 a2 a3)` →?????

```
(let ((a1 1)
      (a2 2)
      (a3 3))
  (list a1 a2 a3))
(let ((a1 1)
      (a2 2)
      (a3 3))
  '(a1 a2 a3))

(quote ((1 I) (2 II) (3 III)))
       ≡ (list '(1 I) '(2 II) '(3 III))
       ≡ (list (list 1 'I) (list 2 'II)(list 3 'III))
       → ((1 I) (2 II) (3 III))
```

Notion de NUPLET

Un **n-uplet** est une **structure de données** qui comporte un nombre fixé d'éléments, de types a priori différents.

Le **type** `NUPLET[alpha beta gamma delta]` représente une structure de 4 éléments, de types `alpha`, `beta`, `gamma` et `delta`.

Exemples de types NUPLET :

Soit `Note`, le type `Nombre/entre 0 et 20/`

`Notes` est le type `NUPLET[Note Note Note]` pour les notes de CC, TP et d'examen d'un étudiant.

`Etudiant` est le type `NUPLET[Nat string string Notes]` pour représenter un étudiant par son numéro de dossier, son nom, son prénom et ses notes.

Les listes d'associations

- Association
- Liste d'associations
- Ajout d'une association
- Recherche d'une association
- Recherche d'une valeur

Une association

Définition : étant donnés deux types **Clef** et **Valeur**, une **association** est un élément de type `NUPLET[Clef Valeur]`, représentée par une liste de 2 éléments.

```
; association de type NUPLET[string string]
        ("chat" "cat")
; associations de type NUPLET[Nat Nat]
        (2003512    192)
        (2003513    567)
; associations de type NUPLET[Nat string]
        (192   "Virginie")
        (567   "Paul")
```


Un autre exemple d'associations

Seulement pour ceux qui désirent approfondir la notion de citation.

```
; associations de type NUPLET[symbole   fonction]
        (list  '+    +)
        (list  '*    *)
        (list  '^    puissance)
```

`(list '+ +)` →`(+ #<primitive:+>)`

`((cadr (list '+ +)) 3 2)` → ? ? ?

`'('+ +)` →`('+ +)`

`((cadr '('+ +)) 3 2)` → ? ? ?

Une liste d'association

Définition : **une liste d'associations** est une liste dont chaque terme est une **association (clef valeur)**

```
; LISTE[NUPLET[string string]]
(list (list "chat"  "cat")
      (list "chien" "dog"))
(list '("chat"  "cat")
      '("chien" "dog"))
'(("chat"  "cat")
  ("chien" "dog"))
```

Ces trois expressions rendent la liste d'associations : `(("chat" "cat")("chien" "dog"))`

Ajout dans une liste d'associations

```
;;; ajout : alpha * beta * LISTE[N-UPLET[alpha beta ]]
;;;                                -> LISTE[N-UPLET[alpha beta]]
;;; (ajout cle valeur a-liste) rend la liste d'association obtenue en ajoutant
;;; l'association (cle valeur) en tête de la liste d'association a-liste.
(define (ajout cle valeur a-liste)
  (cons (list cle valeur)
        a-liste) )
```


Un exemple d'ajout

```
(let ((mon-dico '(("souris" "mouse")
                  ("chat" "cat")
                  ("chien" "dog")
                  ("fromage" "cheese"))))
  (ajout "chat" "tabby" mon-dico))
```


Recherche

(cf. carte de référence)

```
;;; assoc: alpha * LISTE[N-UPLET[a b]] -> N-UPLET[a b] + #f
;;; (assoc cle aliste) rend la 1ère association de aliste dont le 1er
;;; élément est égal à cle. Retourne la valeur #f en cas d'echec.
```

- Rend la première association de la liste (raison d'efficacité) donc la plus récente
- C'est un **semi-prédicat**, elle rend :
  - `#f` lorsque l'association n'existe pas,
  - et sinon la valeur `Vrai` sous la forme de l'association elle-même.

```
(define (mon-dico)
  '(("chat"    "tabby")
    ("souris"  "mouse")
    ("chat"    "cat")
    ("chien"   "dog")
    ("fromage" "cheese")))

(mon-dico)

(assoc "manger" (mon-dico))

(assoc "chien" (mon-dico))

(assoc "chat" (mon-dico))
```


Définition de assoc

`assoc` est une primitive de Scheme.

Elle pourrait se définir comme :

```
(define (assoc cle  a-liste)
  (if (pair? a-liste)
      (if (equal? cle (caar a-liste))
          (car a-liste)
          (assoc cle (cdr a-liste)))
      #f))
```


Fonction valeur-de

La fonction `valeur-de` donne la valeur associée à une clé.

```
;;; valeur-de : alpha * LISTE[N-UPLET[alpha beta]]-> beta + #f
;;; (valeur-de cle aliste) rend la valeur de la 1ère association de
;;; aliste dont le 1er élément est égal à cle. Retourne Faux en cas d'echec.
(define (valeur-de cle a-liste)
  (let ((couple (assoc cle a-liste)))
    (if couple
        (cadr couple)
        #f) ) )
```

Rappel : toute valeur différente de `#f` est équivalente à `Vrai`.

Barrière d'abstraction

Une **barrière d'abstraction** est un ensemble de fonctions qui permet de manipuler des concepts sans se soucier de l'implantation de ces fonctions.

Exemple des types `Ligne` et `Paragraphe` qui servent à la mise en page d'un texte.

- Définition des types `Ligne` et `Paragraphe`
  - `Ligne` : type de chaînes de caractères ne comportant pas de caractères de « fin de ligne »
  - `Paragraphe` : type de chaînes formées d'une suite de lignes (`Ligne`) séparées par des caractères de « fin de ligne »
- Définition de constructeurs et d'accesseurs

On utilisera ces types pour l'affichage des arbres.

Deux constructeurs :

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

;;; paragraphe-cons: Ligne * Paragraphe -> Paragraphe
;;; (paragraphe-cons ligne para) rend le paragraphe dont la première ligne
;;; est « ligne » et dont les lignes suivantes sont constituées par les
;;; lignes du paragraphe « para »
```

Un accesseur :

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


Peu nous importe, pour les **utiliser** de connaître leur **implantation** (la façon dont sont écrites les définitions).

Barrière d'abstraction pour les notes

`Notes` le type `NUPLET[Note Note Note]` pour les notes d'un étudiant.

Barrière d'abstraction pour les notes = un ensemble de fonctions pour manipuler les notes

- un **constructeur** `notes`, par exemple `(notes 8 12 14))`

```
;;; notes: Note * Note * Note -> Notes
;;; (notes note1 note2 note3) rend l'ensemble des notes d'un étudiant
;;; constitué par sa note de CC, sa note de TP et sa note d'exam
```


Les accesseurs de Notes

```
;;; tp: Notes -> Note
;;; (tp ns) rend la note de TP

;;; cc: Notes -> Note
;;; (cc ns) rend la note de contrôle continu

;;; exam: Notes -> Note
;;; (exam ns) rend la note d'examen
```

Il peut y avoir aussi d'autres fonctions sur les notes ...

Peu importe pour l'utilisateur la **représentation** du type `Notes`.
Elle peut être :

- sous forme de liste `(8 12 14)`
- sous forme de vecteur `#(8 12 14)`
- sous forme de Aliste `((exam 8)(cc 14)(tp 12))`
- ...

Barrière pour le type Etudiant

Soit `Etudiant` le type `NUPLET[Nat string string Notes]` pour représenter un étudiant par son numéro de dossier, son nom, son prénom et ses notes.

Cela nécessite un **constructeur** `etudiant` qui à partir d'un numéro de dossier d'un étudiant, de son nom, de son prénom et de ses notes va permettre d'obtenir une valeur de type `Etudiant`

```
(etudiant 231690 "Dupont" "Jean" (notes 8 12 14))
```


Différentes représentations possibles

- Une des représentations pourra être
  `(231690 "Dupont" "Jean" (8 12 14))` si l'on implante les constructeurs `etudiant` et `notes` avec l'aide des primitives `list` et `cons`
- Une autre représentation pourra être
  `#(231690 "Dupont" "Jean" #(8 12 14))` si l'on implante les constructeurs `etudiant` et `notes` avec des vecteurs
- ou encore
  `#(231690 ((nom "Dupont")(prenom "Jean") ((cc 8)(exam 12)(tp 14)))` si l'on implante les constructeurs `etudiant` et `notes` avec des vecteurs et des Alistes.

Les accesseurs de Etudiant

```
;;; num-dossier: Etudiant -> nat
;;; (num-dossier un-etudiant) rend le numero de dossier de un-etudiant

;;; nom: Etudiant -> string
;;; (nom un-etudiant) rend le nom de un-etudiant

;;; prenom: Etudiant -> string
;;; (prenom un-etudiant) rend le prenom de un-etudiant

;;; notes-etudiant: Etudiant -> Notes
;;; (notes-etudiant un-etudiant) rend les notes de un-etudiant
```

Peu importe, pour l'utilisateur de ces fonctions, comment est

implantée la fiche d'un étudiant

Une même interface de `nom` pour différentes définitions

- Si le NUPLET[nat string string Notes] de `Etudiant` est représenté par une S-expression comme `(231690 "Dupont" "Jean" (8 12 14))`

```
(define (nom un-etudiant)
  (if (pair? un-etudiant)
      (cadr un-etudiant)
      (erreur 'nom "pas de dossier")))
```


- Si le n-uplet est représenté par un vecteur comme `#4(231690 "Dupont" "Jean" (8 12 14))` :

```
(define (nom un-etudiant)
  (if (= 0 (vector-length un-etudiant))
      (erreur 'nom "pas de dossier")
      (vector-ref un-etudiant 1)))
```


Une même interface de `notes-etudiant` pour différentes définitions

```
;;; notes-etudiant: Etudiant -> Notes
;;; (notes-etudiant un-etudiant) rend les notes de un-etudiant
```

- Si le NUPLET[nat string string Notes] de `Etudiant` est représenté par une S-expression

```
(define (notes-etudiant un-etudiant)
  (cadddr un-etudiant) )
```

- Si le NUPLET[nat string string Notes] de `Etudiant` est représenté par un vecteur :

```
(define (notes-etudiant un-etudiant)
  (vector-ref un-etudiant 3))
```


Barrière d'abstraction

Une barrière est entre

- l'**utilisation** et
- l'**implantation** des fonctions

Une barrière est habituellement mise pour que l'on ne la traverse pas ...