Cours de Programmation fonctionnelle et logique

19.04.2000

Chapitre 1 : Introduction générale

- Lisp : c'est un langage de programmation inventé au MIT par John MAC Carty et Coll (≈1960), c'est aussi ancien que le fortran.

- Il existe 3 grandes familles:
  - MAC LISP (MIT) Le Lisp de Franz Lisp
  - INTER LISP (Xerox: Pala Alto)
  - Commun Lisp (futur standard MIT) à donner Scheme

On travaillera sous:
- Dr Scheme
- Scheme du MIT

Auteur : Steele, Abelson, Sussman
Il existe aussi Open Scheme (MAC)

- **Différence entre Scheme et les autres Lisp**
  - Portée statique (lexicale) des variables (liaisons lexicale) en opposition à dynamique
  - Notion d'environnement
  - On peut utiliser les programmes (comme données) en entrée et en sortie à d'autres programmes, grâce au lambda calcul (λ-calcul)

- **Classification rapide des langages de programmation** :
  - **Langages fonctionnels** (Lisp, Scheme : non typé, ML, CAML: très fortement typé)
    Pas d'effet de bord sauf pour les E/S
    On y programme essentiellement par opposition de fonction (Un programme = un ensemble de fonctions, sans fonction principale) les informations circulant entre les programmes par l'intermédiaire des paramètres des fonctions.

  - **Langages applicatifs** (?Cobol, SQL)
    Programmation fonctionnelle sans variable

  - **Langages impératifs**
    On programme avec les instructions qui agissent par effet de bord en modifiant le contenu de la mémoire.

  - **Langages logiques** (relationnels) Prolog, Datalog

  - **Langages orientés objets** (Small talk, ……, C++, Java)

Remarque :
2 grandes approche dans la programmation
- Une approche "impérative" (opérationnelle) totalement déterministe, où l'on décrit à la machine, exactement ce qu'elle doit faire (un programme = suite d'instruction à exécuter par la machine)
- Une approche déclarative, totalement non déterministe : on décrit à la machine ce que l'on veut qu'elle calcule (c'est à elle de trouver l'algo) des langages s'en approche Prolog, CAML, Lisp

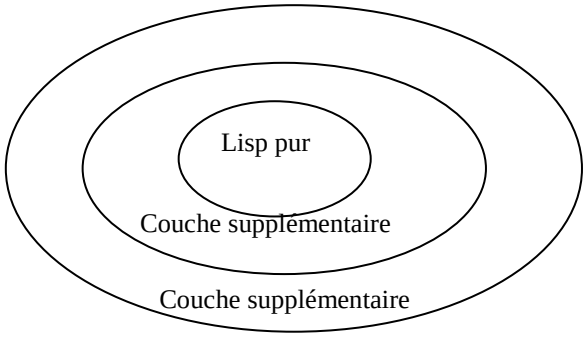
Les langages impératifs au départ été voué au calcul numérique opposé au calcul symbolique.

Chapitre 2 : Principale caractéristique de Lisp / Scheme

Lisp Pur:
Un noyau de langage très réduit existe dans lequel presque tout lisp est définissable
(sauf pour les E/S)

Lisp contient le λ-calcul (un système logique qui permet de représenter simplement, seulement 3 sortes de termes, tout ce qui est calculable par la machine)

Boot strap



Modules
La programmation modulaire est tout à fait naturelle puisqu'un programme est un ensemble de fonctions:

```
; commentaire
; …

F1
```

module

```
; commentaire
; …

F2
```

module

Récursion:
La programmation récursive est tout à fait naturelle en Lisp

Typage:
Pas de type en Lisp:
Il y a un contrôle dynamique du typage, c'est à dire lors de l'exécution des programmes
Il existe quelques test de type qui permettent de programmer le typage
Les listes sont hétérogènes

Mémoire:
De même, il y a une allocation dynamique de la mémoire, pas de déclaration préalable à faire, fixant la taille des structures de données utilisées : on donne des listes de longueur arbitraire

- Les systèmes Lisp sont **interactifs** :
  Une fois sous Unix (taper Scheme), le système demande à l'utilisateur de lui taper une EXPRESSION qu'il va se charger d'EVALUER, une fois la valeur de l'expression retournée, le système (l'interpréteur et/ou le compilateur + machine exécutant le code) en redemande une autre, jusqu'à ce que l'utilisateur lui demande d'arrêter (^D)

Noter que la valeur d'une "constante" c'est à dire une donnée basique comme un entier, un booléen est cette constante

- Un tel déroulement (une SESSION) se déroule de la manière suivante :
  On entre dans une boucle : "Read - eval - print" c'est à dire : "lecture - évaluation - impression" ou encore : " boucle top level" dont on ne sort qu'en interrompant tout le processus lié au système.

Exemple:

```
$ scheme
>(+ 2 3)                    ; ceci est un commentaire
                            ; le prompt Scheme "⇒" s'affiche
                            ; l'utilisateur lui tape une expression à évaluer
5                           ; réponse de l'interprète : évaluation de l'expression

⇒ (+ 2(* 3 5))              ; la notation est préfixe parenthésée
17

⇒ (eq? (< 3 2) #t)          ; Sous le Scheme du MIT #t ⇔ #! true
#f                          ; écriture abrégée de #! false

> #t                        ; écriture abrégée de #! true
#t

> 5                         ; la valeur d'une constante est cette constante
5                           ; constante = données de base comme dans les autres langages de
                            ; programmation

> '(1 2 3)                  ; si une liste précédée du signe '(quote) l'interpréteur la répète
(1 2 3)                     ; elle n'est pas évalué (à l'intérieur)

⇒ (1.(2 3))                 ; le . permet de construire des listes (ne marche pas sous scheme)
(1 2 3)


> (cons 1 '(2 3))                ; cons fait la même chose que le .
(1 2 3)                          ; . et cons sont des constructeur de liste
```

```
> (car '( 1 2 3))          ; car et cdr sont des destructeur de listes
1

> (cdr '(1 2 3))
(2 3)

> (reverse '(1 2 3))
(3 2 1)

> (reverse '(a b a))
(a b a)

> (reverse '((1 2) 3 4))
(4 3 (1 2))

> (equal? (cons 3 '(2 1)) (reverse '(1 2 3)))          ; equal? compare deux liste
#t

⇒ (if (equal? () #!null) "liste-vide" (liste non vide))
liste-vide                                              ; ne marche pas sous MzScheme

⇒ ^D                       ; on sort de Sheme
$                          ;on est sous Unix

> cons                     ; cons est une fonction prédéfinie
#<primitive:cons>

> #f                       ; #f est une constante prédéfinie
#f

⇒ x                        ; une variable non définie = non liée à une valeur dans
Unbounded variable x       ; l'environnement
Error! Level 2:
Error ⇒ ^G (rtn)           ; pour retourner au Top Level

> (define x (+ 1  1))      ; une définition de variable simple
x

> x
2

> (+ x 3)
5

> x                        ; x a garder sa valeur 2
2

> (define (f x y)(+ x (* x y)))      ; une définition de fonction
                                     ; f est le nom de la fonction
                                     ; x et y sont les paramètres de la fonction
                                     ; (+ x (* x y)) est le corps de la fonction et ne
                                     ; comporte pas d'autres variables en dehors des
                                     ; paramètres de f
f
```

```
> f
#<procedure:f>

⇒ (pp f)                                ; pp = pretty printer
(named-lambda (f x y)                   ; ne marche pas sous MzScheme
  (+ x (* x y)))
;No value

> (f 5 3)                 ; appel de la fonction sur des arguments
20

> x                       ; x a garder sa valeur 2
2

> (define x (+ x 3))      ; on redéfinit x
x

> x                       ; on voit que la valeur de x a maintenant changée
5

> (define (fact n)                ; définition de fonction récursive : la fonction factorielle
    (if (= n 0)                   ; 0!=1 et n! = n(n-1)! Si n>0
      1
       (* n (fact (- n 1)))))
fact

> (fact 100)
933262154439441526816992388562667004907159682643816214685929638952175999932
299156089414639761565182862536979208272237582511852109168640000000000000000
00000000

> (define (somme-liste l)                   ; On calcul la somme des éléments d'une liste
    (if (null? l)                           ; d'entiers
     0
     (+ (car l) (somme-liste (cdr l)))))
somme-liste

> (somme-liste '(1 2 3 4 5))
15

⇒ (sys : oblist)  ; ne marche pas sous Scheme
(……)              ; retourne la (très longue) liste des variables liées dans l'environnement
⇒ ^D              ; courant (dont les prédéfinies)
$
```

L'environnement est une liste de liaisons:
Tout ce qui sert habituellement de déclaration dans les langages non itératifs et qu'on retrouve en tête des programmes se retrouve ici défini au fur et à mesure des besoins de l'utilisateur d'où la nécessité d'un tel environnement continuellement mis à jour.

Il y a une gestion dynamique des valeurs des variables et donc de sémantique dynamique des programmes.

Chapitre 3 : Syntaxe et structures de données Lisp / Scheme

Définition des S-exp : l'arbre binaire

- Dans Lisp on a essentiellement une seule structure de donnée non atomique : l'arbre binaire encore appelé S-exp
  Aux feuilles de ces arbres il y a des valeurs atomiques comme:
  - Des nombres
  - Des booléens
  - Des chaînes de caractères
  - Des symboles

  Etc …

- Les autres types de variables non atomiques sont:
  - Les vecteurs
  - Les tableaux
  - Les pointeurs

- On a la grammaire suivante:

Sexp =	constante Symbole	S-expressions atomiques = ATOMES (littéraux dans le 2eme cas)
	(Sexp . Sexp)	S-expressions non atomiques

Constante =	#! null	; S-exp vide (autre écriture : (), nil)
	#! truc | #! false	; booléens (autre écriture : #t, t et #f)
	suite de chiffres (et de caractères éventuellement)	; nombres
	#! c	; caractère c
	"suite de caractères"	; chaînes de caractères

Symbole =	identificateur	; suite de caractères ne commencant pas par un chiffre ; quelques caractères sont réservés

Remarque :
Dans une bibliothèque à chargée à chaque fois on a

```
(define nil ())
```

Il faut, pour charger la bibliothèque faire

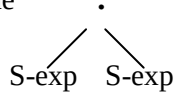
```
load "nom_fichier.scm"
```

Vérifier si l'on a un fichier init.scm ?

Note :
- Le caractère spécial : `'` utilisé pour empêcher l'évaluation de certaines expressions ne figure pas dans cette grammaire, c'est une abréviation pour une fonction prédéfinie (cf section VI)
- Sera pris comme commentaire dans un programme tout ce qui est sur la même ligne derrière un `;`.

Dans cette syntaxe la représentation linéaire des S-exp est utilise sur machine, mais on peut avoir besoin d'une représentation arborescente des S-exp

TS-exp = Atome



Exemples de S-exp :

- S-exp atomiques:

  `2`, `3`, très grands entiers …, `3.14`, … } constantes

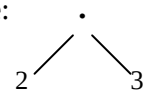
  `()`, `#t`, `#f`,… } constantes

  `+`, `cons`, `if`, `define`, `x`, `it`, `lisp` } symboles

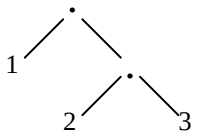
  `#!x`, `"chaine"`, `constantes` } constantes

- S-exp non atomiques:

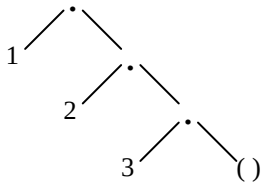
  `(2.3)` représenté par l'arbre:



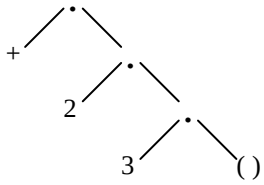
  `(1.(2.3))` représenté par l'arbre :



  `(1.(2.(3.() )))` représenté par l'arbre :



Le dernier exemple est ce que l'on appelle une liste propre en anglais p-list, et on peut l'écrire plus simplement sous la forme `(1 2 3)`
Autre exemple de liste propre : `(+.(2.(3.()))` ⇔ `(+ 2 3)` représenter par:



On communique avec l'interpréteur au moyen des listes (des S-exp) et pas avec les arbres, mais une simplification d'écriture est indispensable : c'est la notation de liste.

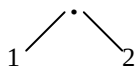
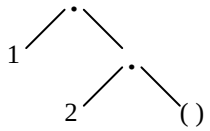
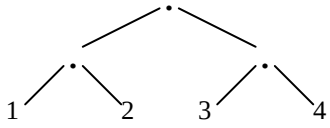
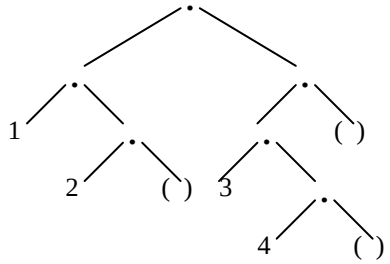
Règle pour passer de la représentation linéaire des S-exp, à la notation de liste :
Si e est une S-exp quelconque alors on remplace dans e tous les

(s_1 . (s_2)) par (s_1 s_2)

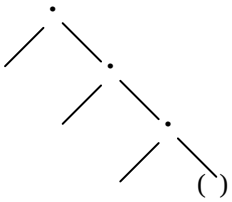
(s_1 . ( )) par (s_1)

Si il ne reste plus de point . dans la notation de liste d'une S-exp on dit alors que c'est une liste propre (liste)

Exemples :

Représentation linéaire générale	Arbre binaire	Notation liste
(1 . 2)		(1 . 2)
(1. (2. ()))		(1 2)
(1. 2) . (3 . 4)		((1 . 2) 3 . 4)
((1. (2. ( ))) . ((3. (4( ))) . ( )))		((1 2) (3 4))

Remarque :
On voit que pour les listes propre on a toujours un schéma du type suivant:

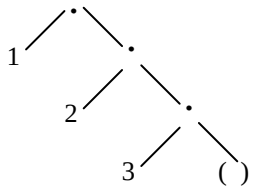


Toute les feuilles droite sont
Toujours ( )

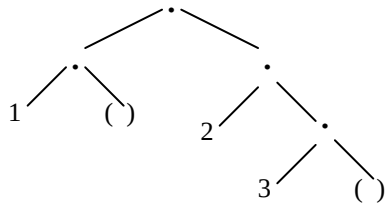
Exercice :

On considère les listes suivantes, les écrire sous formes de S-exp et sous forme d'arbre:
1) (1 2 3)
2) ((1) 2 3)
3) (1 (2) 3)
4) (1 2 (3))
5) ((1 2) 3)
6) (1 (2 3))
7) ((1) 2 (3))
8) ((1 2 3))

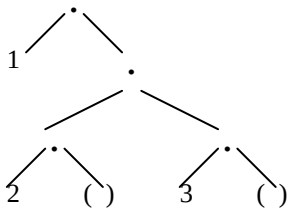
1) (1. (2. (3. ( )))



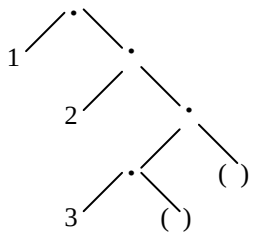
2) ((1. ( )) . (2. (3. ( )))



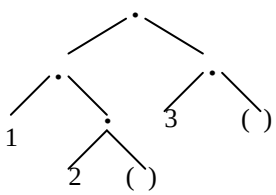
3) (1 . ((2. ( )) . (3 . ( ))))



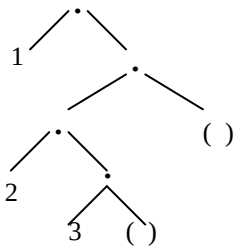
4) (1. (2. (3. ( )) . ( )))



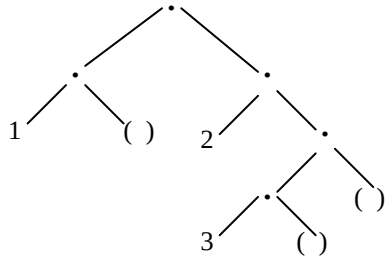
5) ((1. (2. ( ))) . (3 . ( )))



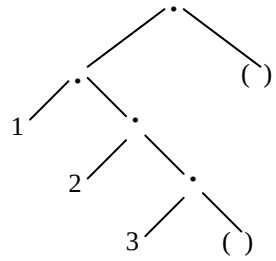
6) (1 . ((2 . (3 . ( ))) . ( ))



7) ((1 . ( )) . (2 . ((3 . ( )) . ( ))))



8) ((1 . (2 . (3. ( )))) . ( ))



Cours de Programmation fonctionnelle et logique

26.04.2000

Note: Avec DrScheme il est préférable d'utiliser MzScheme

Chapitre 3 : Syntaxe et structures de données Lisp / Scheme (suite)

Exemple:

```
>'(1 2 3)
(1 2 3)
⇒(quote 1 2 3)      ; ' ou ` sont des macros caractères c'est à dire des caractères utilisés comme
(1 2 3)             ; abréviation pour ces fonctions prédéfinies
                    ; combinaison du ` et de la , (ne marche pas sous MzScheme)

>'( (+ 2 3), (+ 2 3) 5 '5 `, (+ 2 3) '(+ 2 3))
((+ 2 3) (unquote (+ 2 3)) 5 (quote 5) (quasiquote (unquote (+ 2 3)))
(quote (+2 3)))

> 5                 ; les constantes n'ont pas besoin d'être quotées
5
> 5`,               ; mais on peut les quoter si l'on veut
5

>(1 2 3)
"erreur" …
```

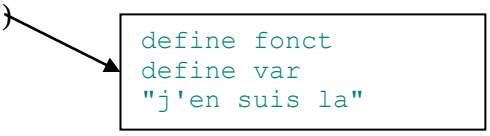
Scheme considère que dans une liste de la forme (e_1 e_2 e_3 ……e_n)

- e_1 est le nom pour une fonction prédéfini, ou nom chargée dans l'environnement
- e_2 e_3 ……e_n sont les arguments de la fonction e_1.

Remarque:

Quand on est dans l'environnement courant, on définit des fonctions avec `define`, ces fonctions peuvent être sauvegardées dans un fichier : toto.scm et rechargées dans un environnement ultérieur:

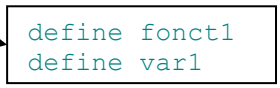
(load "toto.scm")

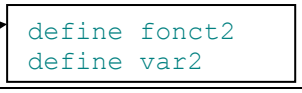


Exemple de programme Scheme:

Fichier général : projet.scm

```
(load "fich1.sc")
        ⋮
        ⋮
        ⋮
        ⋮
(load "fichn.scm)
```





Tout ceci est aussi valable pour des variables globale ("constantes")
Il ne vaut mieux pas modifier les variables globale pendant le déroulement du programme

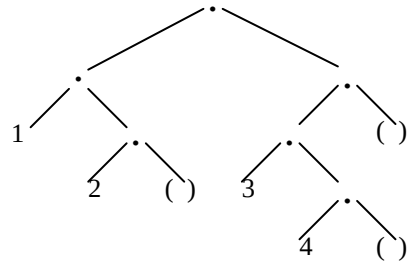
Sous DrScheme:
Trace de l'éxécution des programmes sous DrScheme
```
(require-library "trace.ss")
```
; sa donne acces au fonction `trace untrace`
```
>(trace + car cdr)
```
essayer aussi directement `(step nom_de_fonction)`

Exercice sur les S-exp:

1) Donner la représentation sous forme d'arbre binaire de la S-exp suivante:
((1**.** (2**.** ( ))) **.** ((3**.** (4**.** ( ))) **.** ( ))) ainsi que sa notation de liste

2) On considère la session Scheme suivante
```
⇒(cons 0 (quote (1 2) ))
(0 1 2)
```
   1. Ecrire cette session avec un **.** à la place du cons et un ' à la place du quote
   2. Ecrire cette session avec des S-exp

1) L'arbre conrespondant est le suivant:



La notation liste correspondante est la suivante:
((1 2) (3 4))

2) 1. On peut écrire l'expression sous la forme suivante:
   (cons 0 ' (1 2)) ou encore
   `(0 **.** (1 2))
   2. Sous forme de S-exp cette session s'écrit:
```
>( cons . (0 . (quote . (1. (2 . ( )) .  ( ) ))  )
( 0. (1. (2. ( )) ) )
```
   D'ou l'intérêt de la notation de liste pour simplifier cette écriture

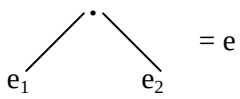
Atomes et syntaxe de base du langage Scheme

Il y a deux sortes de symbole Scheme:
1. Ceux qui sont prédéfinis et constituent la syntaxe de Scheme avec les constantes
2. Ceux qui sont définis par l'utilisateur: très souple, on peut redéfinir des symboles prédéfinis, tout se retrouve dans le même environnement

Comme symbole prédéfinis, on a en particulier ceux qui constituent ce que l'on a appelé le "Lisp pur"
- Les constantes précédemment décrites : S-exp vide `() nil`, booléens `#f #t`, nombres, chaîne de caractères, caractères
- Les fonctions opérant sur les nombres `+ * - / < =`
- Les fonctions opérant sur les S-exp non atomiques `cons .` (constructeur), `car cdr` (selecteur/déconstructeur) avec les règles suivantes:
  (car (cons e_1 e2)) → e_1
  (cdr (cons e_1 e2)) → e_1

Remarque:
(car (e_1 . e_2)) → e_1
(car (e_1 . (e_2 . (e_3 . ( ))))) → e_1 car (e_1 . (e_2 . (e_3 . ( )))) = (e_1 e_2 e_3)
(cdr (e_1 . e_2)) → e_2
(cdr (e_1 . (e_2 . (e_3 . ( ))))) → (e_2 e_3)
(cdr (e_1 e_2)) → (e_2) car (e_1 e_2) = (e_1 . (e_2 . ( )))

- avec les arbres:



e_1 = car e
e_2 = cdr e

Exemple:

```
⇒ (car '(1 . 2) )                    ; (ne marche pas sous MzScheme)
1
⇒ (cdr '(1 . 2) )                    ; (ne marche pas sous MzScheme)
2
⇒ (car '(1   2) )                    ; (ne marche pas sous MzScheme)
(2)
```

- Les fonctions de test de type `symbol? Pair?` (→`atom?`) `nomber? Char? String?`
- Comparaison pour les atom `eq?`
- La conditionelle `if`
- Une fonction servant à dfinir des fonctions sans les nommer lambda

La notation traditionnelle d'une fonction est la suivante:
x→x+1
Cette notation n'est pas pratique au plan symbolique:
Dans le λ-calcul on note: λx.x+1
En scheme on notera : (lambda(x) (+ x 1)) on remarque que la fonction n'est pas nommé

Remarques:

- En Lisp / scheme toute expression différente de `()` et `#f` peut être considéré comme le booléen #t
  `(if '(1 2 3) 'oui 'non)` → `oui`
  `(if '() 'oui 'non)` → `non` (sous MzScheme la réponse est oui)
- Tout ce qui attend une réponse booléenne (prédicat) ce termine par un ? (number? E)
- On peut combiner les "a" et les "d" dans car et cdr
  `(car (cdr '(1 2 3)))` → 2
  `(cadr '(1 2 3))` → 2
  Le nombre de "a" et de "b" accepter varie suivant les Scheme

Exercice:

1. Calculer (cadadr '((1 . 2) ((3 . 4) 5) .6))
2. Comment obtenir 3 dans (1 (1 2 3)), (3) dans (1 (1 2 3))

1. (cadadr '((1 . 2) ((3 . 4) 5) .6)) → 5
2. (caddadr '(1 (1 2 3))) → 3 car:
   (cdr '(1 (1 2 3))) → ((1 2 3))
   (cadr '(1 (1 2 3))) → (1 2 3)
   (cddadr `(1 (1 2 3))) → (3)

Autres fonctions prédéfinies très utiles, mais définissables au moyen des précédentes

`1+  -1+`	(incrémentation, décrémentation)
`>  <=  >=`	(nombres)
`list append length`	(S-expressions)
`reverse last-pair`	(S-expressions)
`pair?`	(test de type)
`null? equal?`	(égalité pour les S-expressions quelconque)
`memq member`	(appartenance à une liste)
`assq assoc`	(appartenance à une **liste d'association**: traditionnellement ce sont des listes de la forme (($symb_1$. e_1) …… ($symb_n$. e_n)) où pour $1 \le i \le n$ $symb_i$ est un symbole et e_i une S-exp quelconque. Ces listes jouent le rôle des "Record" ("enregistrements") en Pascal.
`not and or`	(fonctions booléennes)
`cond case`	(fonctions booléennes)

Exemple d'utilisation:

```
> (define (atom? n) (not (pair? n)))        ; Sous MzScheme on n'a pas atom?
> (atom? '(1 2 3))
#f                                          ; C'est une liste
> (atom? '() )                              ; () est considéré comme un atom
#t
> (pair? '(1 2 3) )
#t
> (pair? '( () ) )
#t
> (symbol? (+ 2 3) )                        ; 5 est une constante
#f
> (symbol? (car '(a b)) )                   ; a est un symbol
#t
> (eq? 'a (car '(a b)) )                    ; a eq a
#t
> (eq? '(a) '(a) )                          ; eq? ne teste pas les égalité sur les liste
#f
> (equal? '(a) '(a) )
#t
>(eq? (+ 2 3)  5)
#t
> (null? '(1 2 3) )
#f
⇒ (null? nil)                               ; ne marche pas sous Scheme
#t
> (member '(b) '( (a) (b) (c)))             ; reprend la liste à partit de b
((b) (c))                                   ; utilise equal? , memp  utilise eq?
> (assoc 'b '( (a 1) (b 2) (a 3)))          ; utilise equal? , assq utilise eq?
(b 2)
> (or #f #f 3 4)                  ; retourne le premier élément vrai, ici 3
3                                           ; 4 ne sera pas évalué
> (and 1 2 #f 4)                            ; retourne le premier élément faux, ici #f
#f
> (cond (#f 1 2) (#t 3 4 5))                ; le premier élément de la première liste étant faux
5                                           ; il passe a la suivante, quand le premier élément est
                                            ; vrai il évalue toute la liste et retourne le dernier
                                            ; élément
```

```
; on peut se servir du mot réservé :else
> (cond (#f 1 2) (#f 3 4) (else (+ 2 3)))        ; ici les deux première liste on un
5                                                 ; premier élément qui est faux donc il
                                                  ; évalué le else
> (case (cadr '(a b))
        ( (a e i o u y) 'voyelle)                 ; b n'appartient pas a la liste on évalue
        ( else 'consonne))                        ; donc le else
consonne
> (list 'a 'b 'c)
(a b c)
⇒ (length (list 'a 'b 'c))                        ; retourne la longueur
3                                                 ; sous MzScheme il faut définir lenght
> (append '(a b c) '(d e))                        ; concatène les listes
(a b c d e)
> (last-pair (append '(a b c) '(d e)))            ; retourne le dernier élément avec un
(e)                                               ; niveau de ( )
; sous MzScheme il faut définir last-pair
```

Exercice :

Redéfinir les fonction prédéfinies suivantes:

1. `equal?` à partir de eq?
2. `length`
3. `member?` à partir de equal?
4. `last-pair`

```
1. (define (equal? x y)
          (or (eq? x y)
             (and (pair? x) (pair? y))
                (equal? (car x) (car y))
                  (equal? (cdr x) (cdr y) ))))

2. (define (length l)
         (if (null? l)
              0
            (+ 1 length (cdr l)))))
```

On va maintenant redéfinir la fonction reverse qui renverse une liste au premier niveau:

```
> (reverse '((1 2) 3 (4 5))
((4 5) 3 (1 2))

(define (reverse l)
  (if (null? l)
       l
    (append (reverse(cdr l))  (list (car l)))))
```

On va maintenant écrire la fonction nreverse qui renverse une liste à tout les niveau

```
(define (nreverse l)
  (if (atom? l)
      l
    (append (nreverse (cdr l)) (list (nreverse (car l) )))))

>  (nreverse ((1 2) 3 (4 5))
((5 4) 3 (2 1)
```

Exercice:

Définir les fonction suivantes:

1. `(last '(1 2 3))` → `3`
2. `(nth '((1) (2) (3 4) 5) 3)` → `3 4`
3. `(flatten '((1) (2) (3))` → `(1 2 3)`
4. `(deflatten '(1 2 3))` → `((1) (2) (3))`

Chapitre 4 : fonctionnalité de Lisp et Scheme

Le style de programmation:

Les programmes ne sont pas constitués de suite d'instructions agissant par effet de bord en modifiant globalement ou localement des variables comme en programmation impérative: ils sont constitué d'ensemble de fonctions.
La programmation se faisant alors par composition de ces fonctions, celle-ci communiquant entre elles aux moyens de leurs paramètres

Ceci est valable pour tout ce qui n'est pas E/S, c'est à dire pour tout ce qui n'est pas interaction avec l'utilisateur.

Not effect property

Après exécution d'un programme donné la mémoire se retrouve dans le même état qu'avant l'exécution (Prolog, Scheme)

Exemple:

Imaginons que l'on veuille calculer le dernier élément d'une liste.
Posons la définition suivante:

```
> (define (f l)
     (reverse l)   } 1
         (car l))  }
f
> (f '(1 2 3))
1
```

On voit que la fonction ne fonctionne pas car le bloc 1 sera évalué dans un environnement ou l=(1 2 3) mais `(reverse l)` n'affecteras pas `l` donc `(car l)` sera effectué avec `l` valant (1 2 3)

La bonne définition est la suivante:

```
> (define (g l)
      (car (reverse l)))
g
> (g '(1 2 3))
3
```

On peut aussi définir la fonction de façon impérative de la manière suivante:

```
> (define x '())
x
> (define (h l)
     (set! x (reverse l))
       (car x))
h
> (h '(1 2 3))
3
> x
(3 2 1)
```

La définition impérative ne respecte pas la Not effect property, et est inutile pour ce genre de programme, mais set! Est indispensable pour le dialogue avec l'utilisateur.

Remarque:
x est une variable globale ce qui ne pose pas de problème, c'est fonctionnel. C'est le fait de la modifier durant l'exécution de h qui n'est pas fonctionnel.

Cours de Programmation fonctionnelle et logique

03.05.2000

Chapitre 4 : fonctionnalité de List et Scheme (suite)

➢ Les fonctions nommées

On a vu la "fonction instruction" `define`, cette fonction
- modifie la variable globale "environnement" prise
- trouve au méta niveau c'est à dire celui de l'implantation, le niveau objet étant celui de l'utilisateur

Pour définir :
- des varibles simples (par exemple x=2)
- des fonctions
- des variables fonctionnelles (on le verra plus tard)

➢ Syntaxe de la construction

`( define ( f  x1 ..... xn) e1 ..... em)`

`f` : nom de la fonction

`x1 ..... xn` : paramètres

`e1 ..... em` : corps de la fonction

En général m=1, Si m>1 les expressions `e1 ,....., em` sont alors évaluées pour l'effet de bord qu'elles produisent jusqu'à `em` non compris. `em` est vraiment évaluée pour retourner une valeur.

➢ Mode d'emploi :

Utilisation de `define` au top level et garder le `let`, `let*`, `let rec` à l'intérieur des fonctions.

➢ Question :

Quel type de programme peut-on écrire en Scheme ?

Tout ce qui est programmable

➢ Attention :

Il faut que la définition informatique du programme en question aie un certain contenu opérationnel

Exemple :

Racine_carrée(x) = la valeur y tel que $y \geq 0$ et $y^2=x$ Cette définition est traduisible en Scheme, mais non exécutable car trop déclarative Ceci n'est pas un programme en Scheme

Exercice :

Prolog étant plus déclaratif que Scheme, trouver un exemple de Programme Prolog qui ne soit pas un programme Scheme.

Exemple de définition déclarative qui sont des programmes exécutables en Scheme (et dans les autres langages de programme actuels) : Les définitions inductives ou récursives
On a écrit des interpréteurs capables d'exécuter de telles définitions.

Exemple :

Somme (n, 0) = n
Somme (n, m+1) = (Somme (n, m) +1)

Cette définition mathématique déclarative de la somme de 2 entiers issue du schéma de récursion sur les entiers

```
( define ( somme n m)
      ( if ( = m 0 ) n
           ( + (somme  n (-1+ m) ) 1 )))  ; sous MzScheme il faut définir -1+
```

Remarque :
La fonction somme est "totale", en effet elle s'arrêtera sur toute entrée d'entiers car issue du schéma de récursion.
En informatique, on appelle une fonction récursive une fonction définie à partir d'elle même et qui s'appellera donc durant son exécution et qui peut fournir des fonctions qui ne s'arrêtent pas pendant leur exécution.

Il existe 3 façons de bouclé pour une fonction :
- Une fonction qui boucle sans rien modifier
  ```
  ( define (f n )  (f n) )
  ```
- Une fonction qui boucle en remplissant la mémoire de plus en plus
  ```
  ( define (f n) ( + 1 (f n) )
  ```
- Une fonction qui boucle en construisant une structure de donnée de plus en plus grande (par exemple un entier)
  ```
  ( define (f n) (f (+n) ) )
  ```

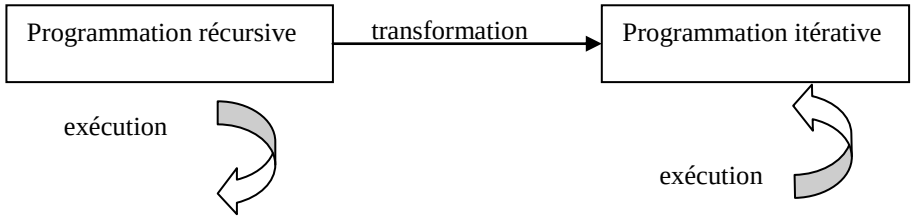
Remarque :
Quel problème pose l'utilisation des fonctions récursive ?
Il y a besoin de gérer une pile (des appels laissés en attente) pendant l'exécution de la fonction
Par exemple : `(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))`

Exercice :

Ecrire l'exécution (avec trace) de (fact 5)

```
> (fact 5);
fact 4 5
( * ( * ( fact 3) 4 ) 5)
( * ( * ( * ( fact 2) 3) 4) 5)
( *( *( *( * ( fact 1) 2) 3) 4) 5)
( *( *( *( *( * ( fact 0) 1) 2) 3) 4) 5)
( *( *( *( *( * ( 1 ) 1) 2) 3) 4) 5)
( *( *( *( * ( 1) 2) 3) 4) 5)
( *( *( * ( 2) 3) 4) 5)
( *( * ( 6) 4) 5)
( * 24  5)
120
```

Remarque :
Pour éviter la gestion de cette pile, on utilise une technique, dite "des paramètres d'accumulation" ou "récursion terminale", ces paramètres permettant de jouer le rôle de la pile des calculs laissés en attente



Ce qui sera incrémenté, ce sont des paramètres de fonctions et non pas des variables globale comme en programmation impérative

Exemple:

```
(define ( fact_it n) ( f_it 1 n))
(define (f_it x n)                          ; x est le paramètre d'accumulation
     (if ( = n 0)
           x
              (f_it (* x n) (- n 1) )))
```

Exercice :

1. Tracer l'appel de (fact_it 5)
2. Idem avec la fonction de fonction de Fibonnaci
   a) Definir (fib n) de manière "normale"
   b) Definir (fib_it n) avec un paramètre d'accumulation

1.

```
> (fact_it 5)
f_it (1 5)
f_it (5 4)
f_it (20 3)
f_it (60 2)
f_it (120 1)
f_it (120 0)
120
```

2. a)

```
( define (fib n)
   (if (< n 2 )
     1
     ( + (fib (- n 1)) (fib (- n 2)))))
```

2. b)

```
( define (fib_it n) (f_it 1 1 n))
(define ( f_it x y n)
    (if (< n 2)
       x
         (f_it (+ x y) x (- n 1)))))
```

Exercice :

1. Définir 2 prédicats ent_pair? et ent_impair? Qui testent respectivement si un entier donné est pair ou impair au moyen de fonctions mutuellement récursives.

1.

```
(define (ent_pair? n)
   (if (= n 0) #t (ent_impair? (- n 1))))
(define (ent_impair? n)
   (if (= n 0) #f (ent_pair? (- n 1))))
```

2. programmer Reverse avec accumulateur

➢ Les fonctions sans nom, les lambda expressions.

On peut en Scheme définir des fonctions sans les nommer explicitement au moyen de la construction suivante :

(lambda (x_1 …… x_n) e_1 …… e_m)

pour exécuter une telle fonction, on procédera comme pour les fonctions nommées :

(f E_1 …… E_m) ((lambda (x_1 …… x_n) e_1 …… e_m) E_1 …… E_m))

Il est possible de nommer ces fonctions anonymes

(define f (lambda (x_1 …… x_n) e_1 …… e_m))

f est une variable fonctionnelle

≈ (define (f x_1 …… x_n) e_1 …… e_m)) sémantiquement

Cas particulier n=m=1

(define f (lambda (x) e)) ≈ (define (f x) e)

(f E) ≈ ((lambda (x) e E)

≈ ((λ(x) e) E) ; x→ e

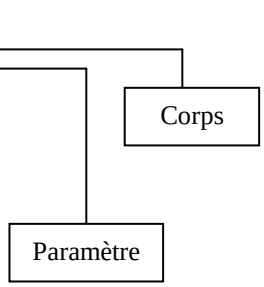
≈ e [E/x] ;e dans lequel x est remplacer par E

≈ val (e [$val[E]/x$]) ;évalue dans un environnement ou x→ val(E)

Exemple de session

```
> '(lambda (x y) (+ x y))
(lambda (x y) (+ x y))
> (lambda (x y) (+ x y))
#<procedure>
> (lambda (x y) (+ x y) 2 3)
#<procedure>
> (define f ( lambda (x y) (+ x y)))
f
> (f 2 3)
5
> (define g (lambda (x) (lambda (y) ( + x y))))
g

> ((lambda (f) (f 2)) (lambda (x) (+ 1 x)))
3
> (g 2)              ; λ(y) (+ x y) avec x→2
#<procedure>         ; λ(y) (+ 2 y)
> ((g 2) 3)
5
```



Une fonctionnelle est une fonction dont un des paramètres est lui même une fonction

➢ Les fonctions comme données. Les Fonctions comme valeurs

En Scheme, on peut passer une fonction en argument à une autre fonction (fonctionnelle)

Exemple de fonctionnelle très utile : map

```
> (map 1+ '(1 2 3 4 5))
(2 3 4 5 6)
> (map (lambda (n) (* n n)) '(1 2 3 4 5))
(1 4 9 16 25)
; on peut redéfinir map très facilement
(define (map f l)
   (if (null? l)
      l
         (cons (f (car l)) (map f (cdr l)))))
```

On peut aussi retourner des fonctions comme valeurs résultant de l'évaluation d'expressions

Exemples :

```
> (define g (lambda (x) (lambda (y) (+ x y))))
g
> ( (g 2) 3)
5
> ( g 2)
#<procedure>

; on peut définir la composition de fonction
> (define (compose f g) (lambda (x) (f (g x) )))
compose
```

Exercice:

Trouver le résultat de la commande suivante:

```
> ( ( ( lambda (x) (lambda (y) (x y))) (lambda (z) (+ 1 z)) ) 3)
```

x est une fonction

`(lambda (z) (1+z))` est évalué dans un environnement ou x= λ(z) (1+z)

`(x y)` est évalué dans un environnement ou x= λ(z) (1+z) et y = z

`(+ 1 z)` est évalué dans un environnement ou z=3

```
4
```

Cours de Programmation fonctionnelle et logique

10.05.2000

Chapitre 4 : fonctionnalité de List et Scheme (suite)

➢ Le lambda calcul

On a vu les fonctions nommées et anonymes (lambda calcul)
On a également vu les fonctionnelles :

- Fonction comme donnée en entrée
- Fonction comme valeur en Sortie
- Données comme fonction (c'est à dire comme programme)

Exemple : les booléens

On définit:
vrai = (lambda (x) (lambda (y) x))
faux = (lambda (x) (lambda (y) y))
On montre

$((\text{vrai } e_1)\ e_2) \rightarrow e_1$	$\text{vrai} \Leftrightarrow x \rightarrow (y \rightarrow x)$
$((\text{faux } e_1)\ e_2) \rightarrow e_2$	$\text{faux} \Leftrightarrow x \rightarrow (y \rightarrow y)$

Démonstration :
$((\text{faux } e_1)\ e_2) = (\ (\ (\lambda(x)\ (\lambda(y)\ y)\ e_1)\ e_2)$
$= (\ (\lambda(y)\ y\ )\ [\ e_1/x\ ]\ e_2)$
$= (\ (\lambda(y)\ y)\ e_2) = y\ [\ e_2/y\ ]$ en conclusion $((\text{faux } e_1)\ e_2) \rightarrow e_2$

$((\text{vrai } e_1)\ e_2) = (\ (\ (\lambda(x)\ (\lambda(y)\ x)\ e_1)\ e_2)$
$= (\ (\lambda(x)\ x\ )\ [\ e_1/x\ ]\ e_2)$
$= (\ (\lambda(y)\ y)\ e_1) e_2) = e_1\ [\ e_2/y\ ]$ en conclusion $((\text{vrai } e_1)\ e_2) \rightarrow e_1$

On définit alors la conditionnelle "si" de façon à avoir l'équation:
$(\text{si b } e_1\ e_2) =$ e_1 si b→vrai
e_2 si b→faux
On peut prendre
si = λ(x y z) ((x y) z)
si' = λ(x) (λ(y) (λ(z) ((x y) z)))

Exercice pour le dossier :

Programmer tout ceci en Scheme Après avoir programmer : vrai, faux , si on aura par exemple : > (si ((lambda (x) x) vrai) "c'est vrai" "c'est faux") vrai Essayer de programmer la factorielle avec cette conditionnelle, idem avec les entiers

On peut de même coder les entiers positifs avec des λ-exp
L'idée: l'entier n sera par exemple codé par :
λ(f) (λ(x) (f (f (…… (f x) ……)))))
D'autre codage sont possible

- Quelles sont les équation que l'on voudra avoir :
  On voudra pouvoir représenter l'addition + par un codage dans le λ-calcul et avoir l'égalité :
  x + 0 = x
  x + (y + 1) = (x +y) + 1

On peut tout représenter dans le λ-calcul

Autre exemple:
De même on représente les primitives cons, car et cdr par des λ-exp
Equations a respecter :
(car (cons e_1 e_2)) → e_1
(cdr (cons e_1 e_2)) → e_2

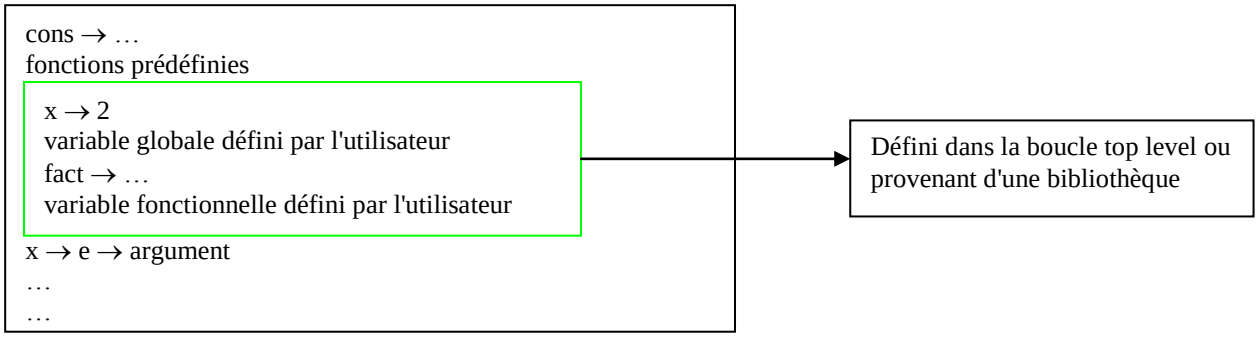
Exercice :

Trouver cons, car et cdr avec des λ-calcul

Chapitre 5 : Evaluation des programmes, les définitions locales

On va décrire comment sont évalués les expressions (et programmes) Scheme, l'idée essentielle :
Toute l'exécution se déroule dans un environnement ou chaque symbole définit dans la session en cours est lié à une valeur (simple ou fonctionnelle). Ce même environnement servira lors des appels de fonction pour lier des paramètres à leur arguments ainsi que dans les liaisons locales.

Environnement:



Remarque :
Les S-exp servent à représenter :

- La structure de donnée fondamentale (non atomique) de Lisp (c a d l'arbre binaire)
- Les définitions de fonctions (et donc les programme)
- Les définitions de constantes
- L'appel de fonction (et l'exécution des programme)

Forme générale des appels de fonction :
(e e_1…… e_n)
e : expression dont la valeur est une valeur fonctionnelle c'est à dire soit :

- une λ-exp explicite
- un nom pour une fonction prédéfini ou défini au préalable par l'utilisateur

e_1…… e_n : argument pour la fonction ($n \geq 0$)

A partir du moment ou il n'y a pas de ' Scheme considère que c'est un appel de fonction.

Que se passe t-il lors d'un tel appel ?

- Scheme commence par évaluer les arguments e_1…… e_n (appel par valeur) dans un ordre indifférencié
- Puis l'expression e est évalué, l'interpréteur reconnaît qu'il s'agit d'une valeur fonctionnelle (sinon "erreur") à n paramètres
- Il calcule alors le résultat de l'application de e aux valeurs e_1…… e_n en évaluant le corps de la fonction dans un nouvel environnement provisoire ou en plus de l'environnement courant (celui existant lors de l'appel de la fonction) chaque paramètre x_i est lié à la valeur de l'argument e_i évalué.

L'appel par valeur entraîne des problèmes avec la définition de la condition si par exemple sur la factorielle.

A la fin d'une telle évaluation l'environnement initial est restauré (Not effect Property) sauf si la fonction comporte des affectations de variables (set!, do, …)

Bien voir que les paramètres d'une fonction sont des variables locales à la fonction : leur portée est le corps de la fonction.

Exemple de session :

```
> (define x 2)
x
> x
2
> (define y 5)
y
> (define ( f x y) ( x (* y y)))            ; f est une fonctionnelle
f
> (f ( lambda (x) (1+ x)) 3)
```

- évaluation de `( x (* y y))` dans un environnement ou
  `x → ( lambda (x) (1+ x))`
  `y → 3`
- évaluation de `(* y y)` en 9
- évalue x et voit que `x → ( lambda (x) (1+ x))`
- évalue `(1+ x)` dans un environnement ou `x → 9`

```
10
```

```
> x                                ;x et y retrouve leurs valeurs initiales
2
> y
5
> ( define (g) (+ 2 3))           ;cette fonction n'a pas d'argument
g
> g
#<procedure:g>
> (g)
5
```

Le cas des fonctions récursives

C'est un peu plus compliqué, la fonction est défini dans un environnement ou elle est censée être lié et faire référence à sa propre définition. (implémentation: on construit des boucles dans l'environnement)

Exemple:

```
> (define (f n) (f (+ 1 n)))
f
> (f 5)                            ;dans un tel appel le corps de f (f (1+ n)) est évalué
                                   ;dans un environnement ou f → λ(n) (f (1+ n))
                                   ;et n →5, ce qui aura pour effet d'appeler (f 6)
core
```

Portée lexicale des variables, liaison statique

Que se passe-t-il si le corps de la fonction contient des variables qui ne sont pas des paramètres de celle-ci (ni des variables locales : let…)

- Ou bien ces variables sont liées à la valeur qu'elles ont lors de la définition de la fonction (ça les oblige à être définis à ce moment là) ⇒ liaison statique
- Ou bien ces variables sont liées à la valeur qu'elles ont lors de l'exécution de la fonction ⇒ liaison dynamique

Liaison statique tous sauf Lisp et Scheme
Liaison dynamique Lisp, Scheme

Exemple :

```
> (define y 2)
y
> (define (p x) (+ x y))           ;y est libre dans p mais liée précédemment dans
                                   ; dans l'environnement
p
> (p 3)
5
> (define y 5)                     ; on change la valeur de y
y
> (p 3)
8                                  ; la valeur de p a changé: il y a donc eu une
                                   ; liaison dynamique de variables
```

```
;autre exemple avec une variable fonctionnelle
> (define (f n) (+ n (g (- n 1))))
f                                       ; g n'est pas lié mais Scheme accepte, liaison
                                        ; dynamique
> (define (g m) (if (<= m 0) 1 (f (- m 1))))
g
> (f 5)
10
> (define (g m) (if (<= m 0) 10 (f (- m 1))))
g
> (f 5)
19
; les variables libres des fonctions (que ce soient des variables simples ou fonctionnelles) sont liées
; dynamiquement dans l'environnement global.

;Lorsqu'il s'agit de variables globales, apparaissant au Top Level, la fonction define opère des liaisons
; statiques, ce qui confère à ces variables, un statut de constantes
> (define x 2)
x
> (define y x)
y
> y
2
> (define x 3)                         ;on redéfinit x
x
> y
2                                       ; la valeur de y reste inchangée
                                        ; même chose pour des variables fonctionnelles
> (define f (lambda (x) x))
f
> (define g f)
g
⇒ (pp g)                                ; pp ne marche pas sous MzScheme
(define (g x) x)                        ; g vaut (lambda (x) x)
#t
> (define f (lambda (x) (xx)))          ; on redéfinit f
f
⇒ (pp g)                                ; pp ne marche pas sous MzScheme
(define (g x) x)                        ; la valeur de g reste inchangée
#t
; remarque Δ=( λ(x) (xx)) (λ(x) (xx)))
```

Les définitions locales

On peut en Scheme utiliser des variables (simples, fonctionnelles) locales dans le corps des fonctions au moyen des fonctions: `let`, `let*`, `letrec` (`define`)
Les liaisons opérées seront statiques

Liaison locale let

`( let ( (x1 e1) ..... (xn en) ) E1 ..... Em)` ≈ `(( lambda (x1 ..... xn )) E1.....Em) e1.....En)`

cas n=m=1
(let ((x e)) E) ≈ ((lambda (x) E) e)
⇔ "Soit x qui vaut la valeur de E dans E à évaluer"
⇔ val (E [val(e)/*x*])

Liaison locale let*

(let* ((x_1 e_1) …… (x_n e_n)) E_1 …… E_m)

≈ (let* ((x_1 e_1)) (let* ((x_1 e_1)) (…… (let* ((x_n e_n)) …… E_1 …… E_m)……)))

Liaison locale let rec

Chaque liaison $x_i \rightarrow$ val(e_i) intervient dans chaque liaison $x_j \rightarrow$ val (e_i) i, j ≤n

Exemples :

```
> (define x 2)
x
>(define y 3)
> (let ((x (+ 2 3)) (y (* 2 3))) (list x y))
(5 6)
> x
2                                                    ;x retrouve sa valeur initiale
> y
3                                                    ;y la sienne
> ( let ((x (+ 2 3)) (y (* x x))) (list x y))
(5 4)
> (let* ((x (+ 2 3)) (y (* x x))) (list x y))
(5 25)
> x
2
> y
3
> (define (f x) (let ((y x)) (+ x y)))
f
> (f  5)
10


> (define g (lambda (x) (let ((y x)) (+ x y))))
g
> (g 5)
10                                                   ;On évalue le corps de g dans un
                                                     ;environnement où x = 5
                                                     ;et donc évalue (+ x y) avec x = 5 et y = 5


> x
2
> y
3
> (define h (let ((y x)) (lambda (x) (+ x y))))
h
> (h 5)
7
> x
2
> y
3
> (letrec ((h (lambda (z) (+ x y z))) (x 5)) (h 10))
18
                                                     ;Evaluation de l'appel (k 10),  k→ λ(z) (+ x y z)
                                                     ;avec z = 10, x = 5,  y = 3
                                                     ;z: paramètre, x: variable locale, y: variable libre
> (letrec (( fact (lambda (n)
      (if (= n 0) 1
                  (* n (fact (- 1 n))) ))))  erreur ???
   (fact 5))
120
```

Corps de f

Corps de g

Corps de h de paramètre x

Autre session montrant que les fonctions `let`, `let*`, `define`, `letrec` employées localement fournissent des liaisons statiques pour les variables (toutefois certains emplois de define fournissent une liaison dynamique)

```
> (let ((x 1)) (let ((f (lambda (y) (+ x y)))) (let ((x 10)) (f 2))))
3                                                  ;liaison statique du let : x→1
> (define x 2)
x
> (define (g)                                      ;ne marche pas sous Scheme
       define x 1
            (define f (lambda (y) (+ x y)))        ;liaison statique du define employé
               (define x 10)                       ;comme un let
                    (f 2))
g
> (g)
3
> x                                                ;ceci montre que employé localement le
2                                                  ; define à une portée locale contrairement à set

> (define (h x)                                          ;ne marche pas sous Scheme
      (letrec (( f (lambda (n) (+n (g (-1+ n)))))       ;f et g sont mutuellement
              ((g ( lambda(m) (if (<= m 0) 1            ;récursive, liées statiquement
                                  (f (-1+ m) )))))      ;entre elles
                                      (f x))))
h
> (h 5)
10
> (define g (lambda (n) (if (<= m 0) 10
                            (f (-1+ m)))))
g
> (h 5)
10
> (g 3)
error: "unbounded variable f"……     ;portée locale du f définie dans le letrec de la fonction h
```

Cours de Programmation fonctionnelle et logique

17.05.2000

Chapitre 6 : Application: implantation du calcul propositionnel

Nous allons voir comment représenter des données du calcul propositionnel, puis l'écriture des programmes modulo cette représentation

Par Exemple:
Représentation d'une fonction de domaine fini f par une liste $((x_1\ e_1)...(x_n\ e_n))$ = Rep(f)
où f: $x_1 \rightarrow e_1$
$x_n \rightarrow e_n$
Comment on programme les opérations sur les fonctions modulo cette représentation: par exemple, comment écrire la composition de fonction gof.

➢ Formules propositionnelles, interprétations, substitution...

Représentation des formules propositionnelles:
- Une variable propositionnelle sera représentée par le symbole Scheme correspondant.
- Le connecteur ¬ sera représenté par le symbole Scheme `non`.
- Les connecteurs ∧, ∨, →, ↔ seront représentés par les symboles `et`, `ou`, `imp`, `ssi`.

Une formule propositionnelles sera alors représentée (en interne) par un arbre binaire (S-exp) en prenant la notation préfixée parenthésée.

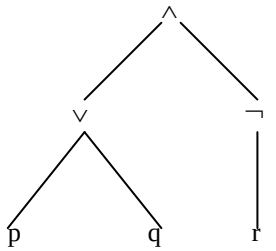
Exemple :

F = ∧ ∨ p q ¬ r
rep(F) = (et (ou p q) (non r))

```
> (define repF '(et (ou p q) (non r)))
> (car repF)
et
> (cadr repF)
(ou p q)
> (caddr repF)
(non r)
```

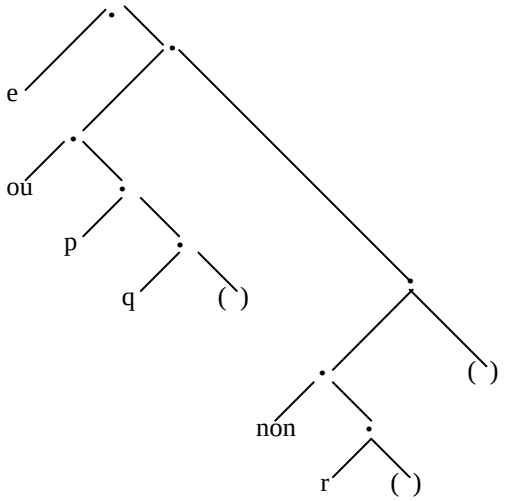
Exercice :

Ecrire l'arbre associé à F.



Exercice :

Ecrire l'arbre binaire associé à rep(F).



Exercice :

Définitions inductives
rep(v) = v si v∈P
rep(¬G) = (non rep(G))
rep(∧ G H) = (et rep(G) rep(H))
de même pour les autres connecteurs binaires
Formule notation préfixe ⇒ représentation préfixe parenthésée de Scheme

Autre possibilité de représentation des formules propositionnelles:
Formule notation infixe ⇒ représentation infixe parenthésée de Scheme
rep'(v) = v si v∈P
rep'(¬G) = (non rep'(G))
rep'(G ∧ H) = (rep'(G) et rep'(H))

- Programmes calculant sur les formules propositionnelles rep(F)

On utilise des variables globales pour ranger les connecteurs
%lc → liste des connecteurs
%lcu → liste des connecteurs unaires (`non`)
%lcb → liste des connecteurs binaires (`et`, `ou`, `imp`, `ssi`)

Exemple :

```
; fonction de test morphologique pour les formules propositionnelles
; écrites avec rep

(define (test s)
      (if (atom? s)
            (not (member s %lc))
            (let ((c (car s)) (l1 (cdr s)))
                  (if (null? l1)
                        #f
            (let ((G (car l1)) (l2 (cdr l1)))
                  (if (null? l2)
                        (and member c %lcu)
                              (test G))
                        (let ((H (car l2)) (l3 (cdr l2)))
                              (if l3
                                    #f
                                    (and (member c %lcb)
                                          (test G)
                                          (test H)))))))))
```

Exemple :

```
; fonction de traduction des formules propositionnelles de la forme
; préfixe parenthésée: rep vers la forme infixe parenthésée rep' et
; inversement: rep'(F)→ rep(F)
; la même fonction trad(F) marche dans les deux sens

(define (trad F)
      (if (atom? F)
            F
            (let ((x (car F)) (y (cadr F)) (z (cddr F)))
                  (if (null? z)
                        '( ,x,(trad y))
                        '( ,(trad y), (trad x), (trad(car z)))))))
```

Exercice (avec rep(F))

Ecrire une fonction "hauteur", qui calcule la hauteur de la formule F (= longueur de la branche la plus longue de l'arbre associé à la formule en comptant la racine)
Ecrire une fonction "longueur" qui calcule la longueur d'une formule F.
Ecrire une fonction "degre" qui calcule le degré d'une formule F.

```
(define (haut F)
      (if (atom? F)
            1
            (let ((G (cadr F)) (l (cddr F)))
                  if (null? l)
                        (1+ (haut G))
                        (1+ (max (haut G) (haut (car l))))))))


(define (long F)
      (if (atom? F)
            1
            (let ((G (cadr F)) (l (cddr F)))
                  if (null? l)
                        (1+ (long G))
                        (1+ (+ (long G) (long (car l))))))))
```

```
(define (degre F)
      (if (atom? F)
            0
            (let ((G (cadr F)) (l (cddr F)))
                  if (null? l)
                        (1+ (degre G))
                        (1+ (+ (degre G) (degre (car l))))))))
```

On se rend compte qu'on a écrit trois formules quasiment identiques.
On en déduit une fonctionnelle de décomposition des formules propositionnelles
Cas simple: les connecteurs ne seront pas distingués les uns des autre.

```
; fonctionnelle de décomposition
; F représente une formule propositionnelle
; x une expression quelconque
; y une fonction à 1 argument qui doit être une formule
; z -------------- 2 ----------------------------------

(define (decomp F x y z)
      (if (atom? F)
            x
            (let ((G (cadr F)) (l (cddr F)))
                  (if (null? l)
                        (y G)
                        (let ((H (car l)))
                              (z G H))))))

(define (haut F)
      (decomp F
            1
            (lambda (x) (1+ (haut x)))
            (lambda (xy) (1+ (max (haut x) (haut y))))))

(define (long F)
      (decomp F
            1
            (lambda (x) (1+ (long x)))
            (lambda (xy) (1+ (+ (long x) (long y))))))

(define (long F)
      (decomp F
            0
            (lambda (x) (1+ (long x)))
            (lambda (xy) (1+ (+ (long x) (long y))))))
```

Exercice :

Utiliser la fonctionnelle decomp pour calculer la liste des variables propositionnelles apparaissant dans F

```
(define (listvar F)
      decomp F
            (list F)
            (lambda (x) (listvar x))
            (lambda (xy) (append (listvar x) (listvar y)))))
```