Programmation Fonctionnelle
Répétition 1
11 février 2010

Rappels

```
(cons 'a 'b) -> (a . b)
(cons 'a '(b)) -> (a . (b)) ;;= (a b)
(list '(a) 'b) -> ((a) b)
(append '(a) '(b)) -> (a b)

Définir une fonction:

(define f
   (lambda (arg1 arg2 ...)
      (... ))
ou

(define (f arg1 arg2 ...)
   (... ))
```

Évaluations

Exercice 1

```
3 ----------------------------------------> 3
#t ---------------------------------------- -> #t
(+ 1 2) ----------------------------------- -> 3
(/ 2 3) ----------------------------------- -> 2/3
(+ (* 3 4) 10) ---------------------------- -> 22
(* 3 (- 12 5)) ---------------------------- -> 21
(+ (+ 5 8) (+ 2 4)) ----------------------- -> 19
(define a 4) ------------------------------ -> NS  // non spécifié
a ----------------------------------------- -> 4
(quote a) --------------------------------- -> a
'a ---------------------------------------- -> a
(define b a) ------------------------------ -> NS
b ----------------------------------------- -> 4
(define a 6) ------------------------------ -> NS
a ----------------------------------------- -> 6
b ----------------------------------------- -> 4
(define c (quote a)) ---------------------- -> NS
c ----------------------------------------- -> a
(define d #t) ----------------------------- -> NS
(define Robert 'Bob) ---------------------- -> NS
Robert ------------------------------------ -> Bob
(car '(a b)) ------------------------------ -> a
(car (quote (a b))) ----------------------- -> a
(cdr '(a b)) ------------------------------ -> (b)
(cdr (quote (a b))) ----------------------- -> (b)
(cons 'a '()) ----------------------------- -> (a)
(cons 'a '(b)) ---------------------------- -> (a b)
```

```
(cons '() '()) ---------------------------- -> (())
(cons '(a) '(b)) ------------------------- -> ((a) b)
(list '(a) '(b)) ------------------------- -> ((a) (b))
(append '(a) '(b)) ----------------------- -> (a b)
(cons 'a (cons 'b (cons 'c '()))) --------- -> (a b c)
(car (cons 'a '())) ---------------------- -> a
(cdr (cons 'a '())) ---------------------- -> ()
(cons a '(a b)) -------------------------- -> (4 a b) ou erreur si a n'est pas défini
(cons x '(a b)) -------------------------- -> erreur
(cons (car '(a b c)) (cons
      (car (cdr '(a b c)))
      (cons (car (cdr (cdr '(a b c)))) '())))) -> (a b c)
(cadr '(a b c d)) ------------------------ -> b
(cadar '((a b) (c d) (e f))) ------------- -> b
```

Premières formes

Exercice 2

```
(* 60 60 24 365) -> 31536000
```

Exercice 3

```
(define sayit
  (lambda (x)
     (cond ((= x 1) 'un)
           ((= x 2) 'deux)
           ((= x 3) 'trois)
           ((= x 4) 'quatre)
           ((= x 5) 'cinq)
           (else 'inconnu))))
```

Exercice 4

```
(define sumOfIntegers
   (lambda (n)
      (if (zero? n) 0
         (+ n (sumOfIntegers (- n 1))))))

(define sumOfIntegers_opt
   (lambda (n)
      (quotient (* n (+ n 1)) 2)))
```

Exercice 5

```
(define modul
   (lambda (a b)
      (if (> b a) a
         (modul (- a b) b))))
```

Exercice 6

```
(define pgcd (lambda (a b)
               (cond ((= b 0) a)
```

```
                    ((< a b) (pgcd b a))
                    (else (pgcd b (modulo a b))))))
```

Exercice 7

```
(define power
   (lambda (x n)
      (cond ((= n 0) 1)
            ((even? n) (power (* x x) (quotient n 2)))
            (else (* x (power x (- n 1)))))))
```

Récursion sur les listes

Exercice 8

apply : fonction qui applique une fonction à tous les éléments d'une liste. (apply + l) : fonction qui somme tous les éléments de la liste → (+ l_1 l_2 l_3 ...)

```
(define (sumList l)
  (apply + l))

(define sumListNaif
  (lambda (l)
    (if (null? l) 0
        (+ (car l) (sumListNaif (cdr l))))))
```

Exercice 9

```
(define (removeFirst l x)
   (cond ((null? l) l)
         ((= (car l) x) (cdr l))
         (else (cons (car l) (removeFirst (cdr l) x)))))
```

Spécification

Exercice 10

```
(define xxx
  (lambda (u)
    (if (null? u) 0
        (if (> (car u) 0)
            (+ (car u) (xxx (cdr u)))
            (xxx (cdr u))))))
```

Spécifier => nombre d'arguments ? Type ? Qu'est ce que ça fait ?
Ici : fonction à 1 argument "liste" qui renvoie la somme des éléments positifs de cette liste.

Programmation Fonctionnelle
Répétition 2
18 février 2010

Encore quelques évaluations

Exercice 1

```
(cons '(1 2 3) '(4 5))                          -> ((1 2 3) 4 5)
(append '(1 2 3) '(4 5))                        -> (1 2 3 4 5)
(list '(1 2 3) '(4 5))                          -> ((1 2 3) (4 5))
(list? (cons 'a 'b))                            -> #f
(list? (cons 'a (cons 'b '())))                 -> #t
(list? (cons (cons 'b '()) 'a))                 -> #f
(list? (cons (cons 'b '()) (cons 'b '())))      -> #t

(null? 'a)                                      -> #f
(null? (car '(a)))                              -> #f
(null? (cdr '(a)))                              -> #t

(number? 1)                                     -> #t
(number? '1)                                    -> #t
(number? #t)                                    -> #f
(number? 'a)                                    -> #f
(number? a)                                     -> error

(boolean? 3)                                    -> #f
(boolean? #t)                                   -> #t
(boolean? #f)                                   -> #t
(boolean? '#t)                                  -> #t
(boolean? 'd)                                   -> #f
(boolean? d)                                    -> #f
(boolean? '())                                  -> #f ou #t (dépend du compil)

(null? #t)                                      -> #f
(null? '())                                     -> #t
(null? '(a b))                                  -> #f
(null? (car '(())))                             -> #t
(null? (car '((()))))                           -> #f (car liste de liste vide)
(length '(()))                                  -> 1
(null? #f)                                      -> #f ou #t (dépend du compil)

(symbol? a)                                     -> error si pas lié sinon #f ou #t
(symbol? 'b)                                    -> #t
(symbol? (car '(a b c)))                        -> #t
(symbol? (cons '() '()))                        -> #f
(symbol? #f)                                    -> #f

eq : même objet en mémoire
eqv : idem eq mais aussi si même valeur
equal : idem eqv mais aussi si même structure
```

```
(equal? 'a (car '(a b)))                          -> #t
(equal? '(a b c) '(a b c))                        -> #t
(equal? '(a (b c)) '(a b c))                      -> #f (pas même structure)
(equal? (cdr '(a c d)) (cdr '(b c d)))            -> #t
(equal? '(car '((b) c)) (cdr '(a b)))             -> #f

(lambda (y x) (cons x y))                         -> #<procedure>
((lambda (y x) (cons x y)) '() 'a)                -> (a)
(define id (lambda (x) x))                        -> NS (non spécifié)
(id 1)                                            -> 1 (identité)
(id '(1 2 3))                                     -> (1 2 3)
(id id)                                           -> #<procedure:id>
((id id) (id id))                                 -> #<procedure:id>
(((id id) (id id)) 3)                             -> 3
```

Correction exercices proposés

Exercice 2

```
(car (cdr (car (cdr (cdr (cdr (car (cdr (cdr (car (cdr (cdr (car phrase))))))))))))

(cadr (cadddr (caddar (cddar phrase))))
```

En scheme, on peut concaténer au maximum 4 opérateurs

Exercice 3

Solution du répétiteur :

```
(cons (car (cddddr ls))
   (cons (car ls)
      (cons (caddr ls)
         (cons (cadr ls)
            (cons (cadddr ls)
               (cons (cadr ls) '()))))))
```

Solution avec list (pour le fun)

```
(list
   (car (cdr (cdr (cdr (cdr ls)))))
   (car ls)
   (car (cdr (cdr ls)))
   (car (cdr ls))
   (car (cdr (cdr (cdr ls))))
   (car (cdr ls))
   )
```

Exercice 4

```
(define min
  (lambda (l)
    (cond ((null? (cdr l)) (car l))
          ((< (car l) (min (cdr l))) (car l))
          (else (min (cdr l))))))
```

OK mais très inefficace
Rappel :

```
(let ((x a) (y b)) g)
        |||
((lambda (x y) g) a b)
```

(let peut se traduire par "soit")
On a donc :

```
(define min
  (lambda (l)
    (if (null? (cdr l))(car l)
      (let ((min_tail (min (cdr l))))    ;; deux parenthèses après le let
        (if (< (car l) min_tail) (car l) ;; car on peut définir plusieurs
            (min_tail)))))               ;; valeurs (ici une seule min_tail)
```

Version 2009 avec accumulateur

```
(define min_a
  (lambda (ls a)
    (cond
      ((null? ls) a)
      ((< (car ls) a) (min_a (cdr ls) (car ls)))
      (else (min_a (cdr ls) a)))))

(define min
   (lambda (ls) (min_a (cdr ls) (car ls)))
```

Spécification : min_a est une fonction qui prend 2 arguments :

1. une liste l
2. un nombre a

Si la liste est vide, min_a revoie a, sinon ça revoit le minimum des éléments de la liste l et a

Exercice 5

```
(define big
  (lambda (n l)
    (cond ((null? l) '())
          ((> (car l) n) (cons (car l) (big n (cdr l))))
          (else (big n (cdr l))))))
```

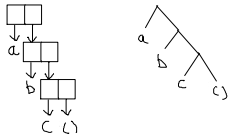
Exercices de compréhension de la structure interne

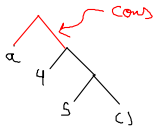
Exercice 6

```
(a b c) => (a . (b . (c . '())))
```

Exercice 7

- `(cons 'a '(4 5))`
- `(list 'a '(4 5))`
- `(append '(1 2 3) '(4 5))`





- `(cons '(1 2 3) '(4 5))`
- `(cons 'a 2)`

Notions d'efficacité

Exercice 8

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

Solution pas très efficace car il faut un cache pour l'implémenter, il faudrait plutôt utiliser un accumulateur

```
(define length
   (lambda (ls)
      (length-iter ls 0)))

(define length-iter
   (lambda (ls acc)
      (if (null? ls) acc
         (length-iter (cdr ls) (+ 1 acc)))))
```

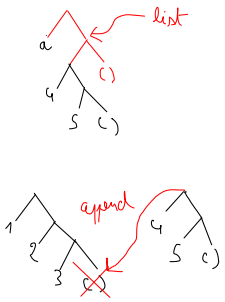
Spécification : lengthiter est une fonction qui prend deux arguments : une liste ls et un entier naturel acc et renvoie la longueur de la liste ls + acc

Exercice 9

```
(define reverse_list
  (lambda (l)
    (if (null? l) '()
      (append (reverse_list (cdr l)) (list (car l))))))
```

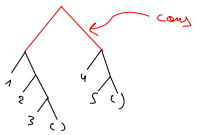
Inefficace car parcours de toute la liste à chaque append

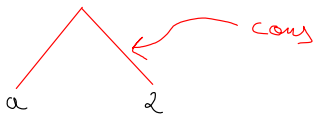
```
(define reverse_aux
   (lambda (l u)
      (if (null? l) u
         (reverse_aux (cdr l) (cons (car l) u)))))

(define reverse
```




```
(lambda (l)
   (reverse_aux l '()))
```

Spécification : reverse_aux est une fonction qui prend deux listes l et u en argument et renvoie la concaténation de l retournée et de u


cons


cons

Programmation Fonctionnelle
Répétition 3
25 février 2010

Exercices sur les listes

Exercice 1

```
(define nombre-de
   (lambda (c ls)
      (cond ((null? ls) 0))
            ((equal? (car ls) c) (+ 1 (nombre-de c (cdr ls))))
            (else (nombre-de c (cdr ls)))))
```

Exercice 2

```
(define make-list
   (lambda (n x)
      (if (zero? n) '()
        (cons x (make-list ((- n 1) x)))))
```

Version améliorée qui garde les résultats intermédiaires

```
(define make-list_acc
   (lambda (n x acc)
      (if (zero? n) acc
         (make-list_acc (- n 1) x (cons x acc)))))

(define make-list
   (lambda (n x)
      (make-list_acc n x '())))
```

Spécification : fonction à 3 arguments (nombre naturel, liste et liste acc) renvoie une liste avec n fois x concaténé avec acc

Exercice 3

```
(define gribomont
   (lambda (n)
      (f-aux n 0 1 2)))

(define f_aux
   (lambda (n a b c)
     (if (zero? n) a
        (f_aux (- n 1) b c (+ a b c)))))
```

Spécification : fonction à 4 arguments (nombre naturels). Si le deuxième est f(i), le troisième f(i+1) etc alors elle revoie f(i+n)

Correction exercices proposés

Exercice 4

```
(define removeAll
```

```
    (lambda (n l)
      (cond ((null? l)'())
            ((equal? (car l) n) (removeAll n (cdr l)))
            (else (cons (car l)(removeAll n (cdr l)))))))
```

Exercice 5

```
(define suffix
   (lambda (Sf ls)
      (or (null? Sf)
      (and (not (null? ls))
           (or (equal? Sf ls)
               (suffix Sf (cdr ls)))))))

(define prefix
  (lambda (pf sf)
    (or (null? pf)
        (and (equal? (car pf) (car sf))
             (prefix (cdr pf) (cdr sf))))))
```

prefix est une procédure à 2 arguments, des listes qui renvoie #t si pf est le préfixe de sf
Autre version de suffix :

```
(define suffix (l1 l2)
   (prefix (reverse l1) (reverse l2)))
```

Exercice 6

```
(define frequency
   (lambda (l)
      (frequency_aux l '())))

(define frequency_aux
   (lambda (li lo)
      (if (null? li) lo
         (frequency_aux (cdr li) (add-elem (car li) lo))))))

(define add-elem
   (lambda (elem l)
     (cond ((null? l) (list (cons elem 1)))
           ((equal? elem (caar l)) (cons (cons elem (+ 1 (cdar l)) (cdr l)))
           (else (cons (car l)(add-elem elem (cdr l)))))))
```

Spécification frequency_aux : fonction à deux arguments (listes) : une liste d'éléments et une liste de fréquence d'apparitions. Renvoie la liste d'apparitions mise à jour par rapport à la liste d'apparition donnée en argument

Interro 2008

Question 1

```
(lambda (x) (cons x '()))              => #procedure
((lambda (x) (cons x '())))            => error: #procedure: attend un argument!
((lambda (x) (cons x '())) x)          => error: x non défini
((lambda (x) (cons x '())) 'x)         => (x)
```

```
((lambda (x) (cons x '())) '())           => (())
(map (lambda (x) (cons x '())) '(0 1 2))  => ((0)(1)(2))

(define f ((lambda (x y) (x y)) (lambda (x) x) (lambda (y) y)))   => NS
```

On remarque que :

```
((lambda (x y) (x y)) (lambda (x) x) (lambda (y) y))
```

est identique à

```
((lambda (x) x) (lambda (y) y))
```

est identique à

```
(lambda (y) y)
```

car on "donne déjà l'argument" à chaque fois. Donc

```
(f 0)  => 0
```

```
(cons (list '(a b) '(c)) (append '(a b) '(c)))
= (cons (((a b)(c)) (a b c) => (((a b)(c)) a b c)

(list (append '(a b) '(c)) (cons '(a b) '(c)))
= (list (a b c) ((a b) c) => ((a b c)((a b) c))
```

Question 2

Première observation : pas de récursion sur le m

```
f(0,m) = m
f(1,m) = m^2
f(2,m) = m f(0,m) + (m+1) f(1,m)
       = (m+2) f(1,m)
```

Pour $n > 1$:

```
f(n,m) = (m+n) f(n-1,m)
```

Donc :

```
(define f
  (lambda (n m)
    (cond ((zero? n) m)
          ((equal? n 1) (* m m))
          (else (* (+ m n) (f (- n 1) m))))))
```

On peut faire mieux avec un accumulateur

```
(define f
  (lambda (n m)
    (if (zero? n) m
        (f_aux n m 2 (* m m)))))

(define f_aux
  (lambda (n m i fim1) ;;fim1 = f(i-1)
     (if (> i n) fim1
         (f_aux n m (+ i 1) (* (+ m i) fim1)))))
```

Spécification : fonction à 4 arguments naturels. Si fim1 est f(i-1,m) alors la fonction renvoie f(n,m)

Question 3

Exemple :

```
(sym? '((a . b) (c . c) (b . a)))   => #t
(sym? '((a . b) (c . c)))           => #f
```

On a en scheme :

```
(member x l)
```

Renvoie le premier suffixe de l contenant x et renvoie false sinon. Exemple :

```
(member 2 '(3 4 2 6 7))  => (2 6 7)
```

Donc on a :

```
(define sym?
  (lambda (r)
    (and-map (lambda (pair)
      (or (eq? (car pair) (cdr pair))
          (member (cons (cdr pair)(car pair))r)))
      r)))

(define and-map
   (lambda (pred l)
      (or (null? l)
          (and (pred (car l))
          (and-map pred (cdr l))))))
```

Programmation Fonctionnelle
Répétition 4
11 mars 2010

Expressions à évaluer

Exercice 1. Quelle est la valeur de cette expression ?

```
((lambda (x) (list x (list (quote quote) x)))
  (quote (lambda (x) (list x (list (quote quote) x)))))

=> ((lambda (x) (list x (list 'quote x))) '(lambda (x) (list x (list 'quote x))))
```

Exercice sur les prédicats

Exercice 2

```
(define filter
  (lambda (l p)
    (cond ((null? l) '())
          ((p (car l)) (cons (car l) (filter (cdr l) p )))
          (else (filter (cdr l) p)))))

(define greater
  (lambda (l a)
    (filter l (lambda (x) (> x a)))))
```

Récursion profonde sur les listes

Exercice 3

```
(define count-all
  (lambda (x ls)
     (cond ((null? ls) 0)
       ((pair? (car ls))(+ (count-all x (car ls))
                           (count-all x (cdr ls))))
       ((eq? (car ls) x)(+ 1 (count-all x (cdr ls))))
       (else (count-all x (cdr ls))))))
```

Exercice sur les compositions de fonctions

Exercice 4

```
(define compose-n
  (lambda (f n)
    (if (zero? n) (lambda (x) x)
      (lambda (x) (f ((compose-n f (- n 1)) x))))))

;; car fonction composée zéro fois avec elle même: identité

((compose-n (lambda (x) (+ x 1)) 10) 3) => 13
```

```
(define compose-fgf
  (lambda (f)
    (lambda (g)
      (lambda (x)
        (f (g (f x)))))))

(define compose fgab
  (lambda (f g a b)
    (lambda (x)
      ((compose-n f a) ((compose-n g b) x)))))
```

Arbres

Exercice 5

On représente un noeud par une liste de 3 éléments : (étiquette fils1 fils2). On représente une feuille par l'atome étiquette.

```
(value '(mul (add (add 3 5)(sub 3 4))(div 3 2.0))) => 10.5

(define (value aa)
  (cond ((null? aa) 0)
        ((not (pair? aa)) aa)
        (else (let* ((op (car aa))
                     (arg1 (value (cadr aa)))
                     (arg2 (value (caddr aa))))
                (cond ((or (not arg1) (not arg2)) #f)
                      ((eq? op 'add)(+ arg1 arg2))
                      ((eq? op 'sub)(- arg1 arg2))
                      ((eq? op 'mul)(* arg1 arg2))
                      (else (if (zero? arg2) #f (/ arg1 arg2)))))))))
```

Exercice Somme

$$f(n) = \sum_{i=0}^{n-1}(([f(i)+2] * [f(n-i-1)+3]) \bmod (n^2+i+5))$$

```
;; f(0)=0 car somme de 0 à -1 est une somme de rien
;; f(1)=0
(define (f0 n)
  (apply +
         (map (lambda (i)
                (modulo (* (+ (f0 i) 2)(+ (f0 (- n i 1)) 3))
                          (+ (* n n) i 5)))
                (enum 0 n))))

(define (enum a b)
  (if (< a b)
     (cons a (enum (+ a 1) b)) '() ))

(define (f2 n)
  (car (f2_aux 0 (+ n 1) '() '())))
```

```
(define (f2_aux r p l li)
  (if (zero? p) l
      (let ((fr (apply +
                  (map (lambda (l_i inv_i i)
                     (modulo (* (+ l_i 2) (+ inv_i 3)) (+ (* r r) i 5)))
                       l (reverse l) li))))
      (f2_aux (+ r 1) (- p 1)(cons fr l)(cons r li)))))
```

Si $l_i = [r-1, r-2, ..., 0]$ et $l = [f(r-1), f(r-2), ..., f(0)]$
alors f2_aux (r p l li) $= [f(r-1+p), f(r-2+p), ..., f(0)]$

Programmation Fonctionnelle
Répétition 5
18 mars 2010

Correction des exercices proposés

Exercice 1

```
; Quelques restrictions:
; Inutile d'aller plus loin que la racine du nombre.
; On peut éliminer tous les diviseurs paires

(define (divisor? m n)
  (zero? (remainder m n)))
; dviseurs prends deux arguments m et n, et renvoi vrai si n est le diviseur de m
; m, n sont des nombres entiers positifs.

(define (prime? n)
  (or (= n 2)
  (and (not (zero? n))
       (not (= n 1))
       (odd? n)
       (not (prime_aux n 3)))))

(define (prime_aux n m)
  (or (> (* m m) n) ; Car calculer la racine est beaucoup plus long
      (and (not (divisor? n m))
      (prime_aux n (+ 2 m)))))

; prime_aux prends deux arguments
; n nombre entier plus grand que 2 et impair
; m nombre entier < n
; elle renvoi faux si il existe un diviseur de n dans l'interval {m, sqrt(n)}
```

Exercice 2

```
; La somme doit nécessairement faire
; n(n+1)/2
; car la somme classique se fait sur
; n(n+1)/2
; et on a n lignes, n colonnes, etc...
; D'abord on vérifie qu'on a pas deux fois le même nombre:
(define (unique? l)
  (or (null? l)
  (and  (not (member (car l) (cdr l)))
  (unique? (cdr l)))))

(define (sumline ll)
  (let ((sh (map (lambda (x) (apply + x)) ll))) ; Me sort une liste de somme
  (if (apply = sh) (car sh) #f)))

; Fonction à un argument ll (liste de liste), qui renvoi la somme de chaque ligne si les sommes s
```

```
(define (sumcol ll)
  (sumline (transpose ll)))

(define (transpose ll)
  (if (null? (car ll)) '()
      (cons (map car ll) (transpose (map cdr ll)))))

; Fonction qui prends une liste de liste de même taille et renvoi sa transposée.
; Elle renvoi une liste vide si la liste de départ est une liste de listes vides.

(define (sumdiag_1 ll)
  (cond ((null? (cdar l)) (caar l))
(else (+ (caar l) (sumdiag (cdr (map cdr l)))))))
; Renvoi la somme de la diagonale principale. ll est une liste de liste représentant un carré.

(define (sumdiag_2 ll)
  (sumdiag (map reverse l)))

(define (magic? ll)
  (and (unique? (apply append c))
       (= (let ((n length c))
       (/ (* n (+ 1 (* n n))) 2)
       (sumine c) (sumcol c) (sumdiag1 c) (sumdiag2 c)))))
```

Abstraction sur les données

Exercice 3

```
; On représente par des paires pointées les nombres complexes:
; 2 + 3i -> (2.3)

(define make_from_real_img cons)
(define real_part car)
(define img_part cdr)
(define (add_complex z1 z2)
  (make_from_real_img (+ (real_part z1) (real_part z2)) (+ (img_part z1) (img_part z_2))))
(define (magintude z)
  (sqrt (+ (* (real_part z) (real_part z)) (* (img_part z) (img_part z)))))
```

Exercices sur les arbres

Exercice 4

```
; On représente un noeud interne par une paire pointée, et une feuille par une étiquette

(define make-tree cons))
(define make-leaf (lambda (x) x))
(define leaf? (not pair?))
(define k (lambda (x) x))  ; Va rechercher une feuille
(define l_tree car)
(define r_tree cdr)  ; Va rechercher l'arbre de gauche ou de droite
(define (depth_first tree)
```

```
  (if (leaf? tree) (list (k tree))
      (append (depth_first (l-tree tree)) (dept_first (r-tree tree)))))
```

Exercice 5

```
(define (simplify tree)
  (if (leaf? tree) tree
      (let ((sim_left (simplify (l-tree tree))) (sim_right (simplify (r-tree tree))))
           (if (and (leaf? simp_l) (leaf? sim_r) (equal? (k simpl_l) (k simpl_r))) simpl_l ; deux
               (make_tree simpl_l simp_r))))) ; Je renvoi les arbres car pas simplifié
```

Programmation Fonctionnelle
Répétition 6
25 mars 2010

Correction exercices proposés

Exercice 1

```
; La fonction prends en argument une liste l d'arbre et renvoi la list
; des étiquettes des arbres de profondeur 0, puis 1...
; et ce dans l'ordre d'apparition de l et de gauche a droite.
(define breath_first_forest
  (lambda (trees)
    (append (apply append (map (lambda (x) (if (leaf? x) (k x) '())) trees)
    ; Le apply append vire les listes vides
           (breath_first_forest
             (apply append (map (lambda (x) (if (leaf? x) '()
                (list (l_tree x) (r_tree x)))) tress)))))))

(define breath_first
  (lambda (tree)
    (breath_first_forest (list tree))))

; On renvoie à la ligne 8 tous les nœuds, et la ligne 11 la liste de tous les fils.
```

Exercice 2

```
(define perfect?
   (lambda (n)
      (and (> n 3)
               (= n (+ 1 (sum-div n 2))))))

(define sum-div
  (lambda (n i)
  (cond ((> (* i 2) n) 0)
        ((= (* i 2) n ) i)
        ((zero? (modulo n i)) (+ i (sum-div n (+ 1 i))))
        (else (sum-div n (+ 1 i))))))
```

sum-div : fonction à deux arguments naturels n et i, renvoie la somme des diviseurs de n compris dans $[i, \frac{n}{i}]$

N-arbre

Exercice 3

```
; Remarque: on considère les labels comme étant des entiers naturels
(define ntree?
  (lambda (t)
    (or (null? t)
        (and (pair? t) ; On doit avoir un objet composé
  (integer? (car t))
```

```
  (>= (car t) 0)
  (pair? (cdr t))
  (ntree? (cadr t)) ; Suite est un narbre?
  (pair? (cddr t))
  (ntree? (caddr t))
  (null? (cdddr t))))))

(define condit?
  (lambda (t)
    (or (null? t)
        (and (condit? (cadr t)) ; Arbre de gauche
        (condit? (caddr t)) ; Arbre de droite
        (greater_or_eq? (car t) (cadr t))
        (less_or_eq? (car t) (caddr t))))))

; Fonction a deux arguments (un entier et un ntree) et qui renvoi
; true si n est plus grand que tous ses fils de gauche
(define greater_or_eq?
  (lambda (n t)
    (or (null? t)
        (and (>= n (car t))
             (greater_or_eq? n (cadr t)))))) ; Fils de gauche

(define less_or_eq?
(lambda (n t)
(or (null? t)
(and (<= n (car t))
 (less_or_eq? n (caddr t)))))) ; Fils de droite
```

Exercice sur les graphes

Exercice 4

```
; Prochaine répet
```

Fonctions

Exercice 5

```
(define symmetrize
   (lambda (f)
      (lambda (x)
         (/ (+ (f x) (f  (- x)) 2)))))

(define anti-symmetrize
   (lambda (f)
      (lambda (x)
         (/ (- (f x) (f  (- x)) 2)))))

(define func-op
   (lambda (op f g)
       (lambda (x) (op (f x) (g x)))))
```

```
(define symmetrize2
   (lambda (f)
      (func-op (lambda (x g) (/ (+ x g) 2))
                     f (lambda (x) (f (- x))))))

(define anti-symmetrize2
   (lambda (f)
      (func-op (lambda (x g) (/ (- x g) 2))
                     f (lambda (x) (f (- x))))))
```

Sous-listes

Exercice 6

```
; ex: (a b c) -> ((a b c) (a b) (b c) (a) (b) (c) '())
; ex: () -> (())
;     (c) -> (() (c))
;     (b c) -> (() (b) (c) (b c))
;     (a b c) -> (() (a) (a b) (a b c) (b) (b c) (c))

(define ajout
  (lambda (ll x n)
    (if (zero? n) '()
        (cons (cons x (car ll))
              (ajout (cdr l) x (- n 1))))))
; Prends 3 arguments: une liste de liste ll, un élément x et un
; nombre naturel n, renvoi les liste de listes correspondant à la hauteur n

(define sublist_aux
  (lambda (l n)
    (if (null? l) '(())
        (let (ll (sublist_aux (cdr l) (- n 1))))
          (cons '() (append (ajout ll (car l) n) cdr ll)))
          ; Pour pas recopier deux fois la  liste vide

; Calcul la liste des sous listes avec les n premiers éléments de ll
(define sublist
  (lambda (l)
    (sublist_aux l (length l))))
```

Programmation Fonctionnelle
Répétition 7
22 mars 2010

Correction exercices proposés

Exercice 1

```
(define inv
  (lambda (g)
    (map (lambda (l) (cons (car l) (inv_aux (car l) g))) g)))

(define inv_aux
  (lambda (n g)
    (apply append (map (lambda (l) (if (member n (cdr l)) (list (car l)) '())) g))))

;; inv_aux est une fonction à deux arguments: n un élément, et g un
;; graphe; elle renvoi la liste des noeuds ayant pour extrémité n

;; member: renvoi faux si l'élément est dans la liste, sinon ça
;; renvoi la liste après la découverte de l'élément.
;; Exemple (member c '(a b c d)) => (c d);
```

Exercice 2

```
(define prefix_croiss
  (lamda (l)
    (cond ((null? l) '())
          ((null? (cdr l)) l)
          ((<= (car l) (cadr l)) (cons (car l) (prefix-croiss (cdr l))))
          (else (list (car l))))))

(define longest_inc
  (lambda (l)
    (if (null? l) '()
      (let ((a (prefix-croiss l))
            (b (longest_inc (cdr l))))
        (if (> (length a) (length b)) a b)))))

Remarque: on a calculé des choses qui ne servent a rien... Genre on
recalcule le longest préfix à chaque fois. une solution serait de
sauter la longueur de longest prefix...

Donc faire une petite fonction auxiliaire en ** qui remplace le cdr
par la liste enlevée du prefix
```

Exercice 3

```
(define sup
  (lambda (l)
    (sup_aux l o)))

(define sup_aux
```

```
  (lambda (l n)
    (cond ((null? l) '())
          ((< n 2) (cons (car l) (sup-aux (cdr l) (+ n 1))))
          (else (sup-aux (cdr l) (modulo (+ n 1) 5))))))
```

Fonction auxiliaire: prends en entrée la liste et un entier naturel

Exercice sur les listes de listes

Exercice 4

```
(define lpref
  (lambda (l)
    (if (null? l) '(())
        (cons '() (map (lambda (x) (cons (car l) x)) (lpref (cdr l)))))))
```

Exercice 5

```
Exemples:
1 -> 1
2 -> 1 + 1, 2
3 -> 1 + 1 + 1, 1 + 2, 3
4 -> 1 + 1 + 1 + 1, 1 + 1 + 2, 2 + 2, 3 + 1, 4

(define nbsum
  (lambda (n)
    (nbsum_aux 1 n)))

nbsum_aux: fonction à deux arguments (m et n) et qui renvoi le
nombre de somme qui font n avec des nombres compris entre n et m.

(define nbsum_aux
  (lambda (m n)
    (cond ((> m n) 0)
          ((= m n) 1)
          (else (+ (mbsum_aux m (- n m)) (mbsum_aux (+ 1 m) n))))))
```

Exercice 6

```
Exemple:
(b c d)
-> ((b) (c) (d))
-> (a b c d)
-> ((a b) (c) (d))
-> ((a) (b c) (d))
-> ((a) (b) (c d))

Donc on ajoute un élément au début, on obtient une autre liste de
tricoupure. Ensuite, les cas non considérés sont ceux ou x est tout seul au début.

(define (bicoup l)
  (if (or (null? l) (< (length l) 2)) '()
    (append (proc1 (car l) (bicoup (cdr l)))
            (proc2 (car l) (list (list (cdr l)))))))
```

```
Remarque: (list (list (cdr l))) = uncoup

(define (proc1 s lll)
  (map (lambda (ll) (cons (cons s (car ll)) (cdr ll))) lll))
proc1 est une fonction à 2 arguments s un élément et lll une liste
de liste de liste. On renvoi une liste de liste de liste ou l'on a
rajouté l dans chaque début de liste de sous liste de liste.

(((x) ()) ((x) ()) ((x) ()))

(define (proc2 s lll)
  (map (lambda (ll) (cons (list s) ll)) lll))
Ajoute une liste contenant l en tête de chaque sous liste

(define (tricoup l)
  (if (or null? l) (< (length 1) 3)) '()
    (append (proc1 (car l) (tricoup (cdr l)))
            (proc2 (car l) (bicoup (cdr l)))))

(define (ncoup l n)
  (cond ((or (null? l) (< (length l) n)) '())
        ((= 1 n) (list (list l)))
        (else (append (proc1 (car l) (ncoup (cdr l) n ))
                      (proc2 (car l) (ncoup (cdr l) (- n 1)))))))))
```

Exercice 7

```
()      ->  (())
(c)     ->  ((c))
(b c)   ->  ((b c) (c b))
(a b c) ->  ((a b c) (b a c) (b c a)
            (a c b) (c a b) (c b a))

(define (permut l)
  (if (null? l) '(())
      (apply append (map (insert (car l))
                         (permut (car l))))))

(define (insert a)
  (lambda (l)
    (if (null? l) (list (list a))
        (cons (cons a l)
              (map (lambda (ll)
                     (cons (car l) ll))
                   ((insert a) (cdr l)))))))
```

Programmation Fonctionnelle
Répétition 8
20 mai 2010

Exercice 1 : Permutations

```
=> () -> (())
=> (c) -> ((c))
=> (b c) -> ((b c) (c b))
=> (a b c) - > ((a b c) (b a c) (b c a) (a c b) (c a b) (c b a))
--> à partir de (b c) on ajoute a à toutes les positions dans (b c) et (c b)
--> n! permutations. Si dérangements: d(n) = (n - 1) (d(n - 1) + d(n - 2))

(define (permut l)
  (if (null? l) '(())
      (let ((w (permut (cdr l))))
        (apply append (map (lambda (x)(all-insert (car l) x)) w)))))

(define (all-insert x l)
  (if (null? l) (list (list x))
    (cons (cons x l)
          (map (lambda (ll) (cons (car l) ll))(all-insert x (cdr l))))))
```

Exercice 2 : Sous-ensembles

```
ensemble => pas d'ordre donc (a b) et (b a) sont le même ensemble
=> () -> (())
=> (c) -> (() (c))
=> (b c) -> (()(b) (b c) (c))
=> (a b c) - > (() (a) (a b) (a b c) (a c) (b) (b c) (c))

(define (lset l)
  (if (null? l) '(())
      (let ((w (lset (cdr l))))
        (append (map (lambda (x) (cons (car l) x)) w)
                w))))
```

Exercice 3 : Words

```
=> (words 2 '(a b c))
=> ((a a) (b a) (c a) (a b) (b b) (c b) (a c) (b c) (c c))
récursion sur la longueur des mots

(define words
  (lambda (n s)
    (if (zero? n) '(())
        (apply append (map (lambda (l) (w_aux l s)) (words (- n 1) s))))))

(define w_aux
  (lambda (l s)
    (map (lambda (x) (cons x l)) s)))
```

Exercice 4 : Arbres binaires complets

```
(d) -> d

(c d) -> /\
         c  d

(b c d) -> /\      /\
           b /\    /\ d
             c  d  b  c

(a b c d) -> /\         /\         /\          /\           /\
             a /\       a /\      /\ /\       /\ d         /\ d
               b /\      /\ d    a b c d     a /\         /\ c
                 c  d    b c                   b  c       a  b
```

```
(define (bt l)
  (cond ((null? l) '())
        ((null? (cdr l))(list l))
        (else (apply append (map (lambda (t) (bt* t (car l))) (bt (cdr l)))))))

(define (bt* t x)
  (if (atom? t) (list (list x t))
      (cons (list x t)
            (map (lambda (t2) (list t2 (cadr t)))
                 (bt* (car t) x)))))
```

Exercice 5 : Lagrange

Écrire une fonction `lagrange`, prenant pour argument un entier naturel n et renvoyant la liste de tous les quadruplets d'entiers naturels (a, b ,c ,d) tels que $a^2 + b^2 + c^2 + d^2 = n$

```
(define lagrange
  (lambda (n)
    (lagrange_aux 4 0 (sqrt n) n)))

(define lagrange_aux
  (lambda (size n1 n2 sum)
    (cond ((and (zero? size) (zero? sum)) '(()))
          ((or (zero? size) (> n1 n2)) '() )
          (else (let ((newSum (- sum (* n1 n1))))
                  (append (map (lambda (x) (cons n1 x))
                               (lagrange_aux (- size 1) 0 (sqrt newSum) newSum))
                          (lagrange_aux size (+ n1 1) n2 sum)))))))
```

Exercice 6 : Polynômes

Dérivée

```
a + bx + cx^2 + dx^3 + ...   -> (a b c d ...)
b + 2cx + 3dx^2 + ...  -> (b 2c 3d ...)

(define (deriv p)
  (if (null? (cdr p)) '(0)
      (deriv_aux 1 (cdr p))))
```

```
(define (deriv_aux n p)
  (if (null? p) '()
      (cons (* n (car p)) (deriv_aux (+ n 1) (cdr p)))))
```

Exercice 7 : Formule

$$f(n) = \sum_{i=0}^{n-1} [f(n-i-1)^{f(i)}]$$

```
(define (f n)
  (car (f_aux (+ n 1) '())))

;;Si l = [f(n-p-1),f(n-p-2),...,f(0)]
-> (f_aux p l) = [f(n-p),...,f(0)]

(define (f_aux p l)
  (if (zero? p) l
      (let ((fn (apply + (map (lambda (x y) (expt x y))
                              l (reverse l)))))
      (f_aux (- p 1) (cons fn l)))))
```