# Programmation fonctionnelle

Répétition 1

11 février 2010

## Avant propos

Informations pratiques :

- Assistant : Stéphane Lens
- `lens @ montefiore.ulg.ac.be`, 04/3662619, bureau I82a.
- `http://www.montefiore.ulg.ac.be/~lens`

- L'interpréteur scheme conseillé est *DrScheme*. (`http://www.plt-scheme.org`)
  Vous pouvez aussi utiliser *Guile* qui est installé sur les ordinateurs de Montefiore.

Quelques points délicats (qui seront répétés, rerépétés, ...) :

- Attention à ne pas confondre `cons`, `list`, et `append`.
- Savoir spécifier est aussi important que savoir programmer. Les énoncés des exercices sont de bons exemples de spécifications.
- Il faut privilégier les solutions simples et courtes.
- Les solutions doivent avoir une complexité acceptable.

## Evaluations simples

**Exercice 1.**
Donner le résultat de l'évaluation des expressions suivantes :

```
3
#t
(+ 1 2)
(/ 2 3)
(+ (* 3 4) 10)
(* 3 (- 12 5))
(+ (+ 2 3) (+ 4 5))
(- (+ 5 8) (+ 2 4))
(define a 4)
a
(quote a)
'a
(define b a)
```

```
b
(define a 6)
a
b
(define c (quote a))
c
(define d #t)
(define Robert 'Bob)
Robert
(car '(a b))
(car (quote (a b)))
(cdr '(a b))
(car (cdr '(a b)))
(cdr (cdr '(a b)))
(cons 'a '())
(cons 'a '(b))
(cons '() '())
(cons '(a) '(b))
(list '(a) '(b))
(append '(a) '(b))
(cons 'a (cons 'b (cons 'c '())))
(car (cons 'a '()))
(cdr (cons 'a '()))
(cons a '(a b))
(cons x '(a b))
(cons (car '(a b c)) (cons
                       (car (cdr '(a b c)))
                       (cons (car (cdr (cdr '(a b c)))) '())))
(cadr '(a b c d))
(cadar '((a b) (c d) (e f)))
```

# Premières formes

---

**Exercice 2.**
Calculer en une seule forme SCHEME le nombre de secondes dans une année (non bissextile).

---

**Exercice 3.**
Ecrire une forme qui rend `un` si `x` est égal à 1,...`cinq` si `x` est égal à 5 et `inconnu` sinon.

De là, définir une fonction *sayit* qui rend `un` si l'argument est 1, ...`cinq` si l'argument est égal à 5 et `inconnu` sinon.

# Récursion sur les nombres

---

**Exercice 4.**
Ecrire une fonction $sumOfIntegers$ prenant en argument un naturel $n$ et calculant la somme des naturels inférieurs ou égaux à $n$.

---

**Exercice 5.**
Définir la fonction `mod` qui renvoie le modulo de deux nombres.

Remarque : la fonction *modulo* est prédéfinie, nous la redéfinissons sous un autre nom à titre d'exercice.

---

**Exercice 6.**
Définir une fonction `pgcd` qui renvoie le plus grand commun diviseur de deux nombres.

---

**Exercice 7.**
Ecrire une fonction prenant en argument deux nombres $x$ et $n$ et renvoyant $x^n$.

Remarque : la fonction *expt* est prédéfinie, nous la redéfinissons sous un autre nom à titre d'exercice.

# Récursion sur les listes

---

**Exercice 8.**
Définir la fonction *sumList* qui calcule la somme des éléments d'une liste de nombres.

---

**Exercice 9.**
Ecrire une fonction `removeFirst` à deux arguments `l` et `n` dont les valeurs sont respectivement une liste et un nombre entier, qui retourne la liste privée de la première occurence de `n`.

```
(removeFirst '(2 7 1 7 3 1) 1) ⇒ (2 7 7 3 1)
```

# Spécification

---

**Exercice 10.**
Spécifier la fonction suivante :

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

# Exercices proposés

---

**Exercice 11.**
La variable `phrase` est définie comme suit :

```
(define phrase
  '(((e) x (e r (c (i c e) c (o m (p l (i q))) u e)))))
```

Extraire le `m` au moyen de `car`, `cdr`, `cadr`, ...

---

**Exercice 12.**
Construire la liste `(s c h e m e)` au seul moyen de la liste `ls` définie comme suit :

```
(define ls '(C E H M S))
```

et des fonctions `cons`, `car`, `cdr`, `cadr`, ...

---

**Exercice 13.**
Définir la fonction `min` qui renvoie le minimum de la liste non vide de nombres donnée en argument.

---

**Exercice 14.**
Ecrire une fonction `big` qui prend comme arguments un nombre entier `n` et une liste `l` de nombres entiers, telle que `(big n l)` est la liste des éléments de `l` plus grands que `n`.

`(big 5 '(7 -3 6 1 -2 5 6))` $\Rightarrow$ `(7 6 6)`

# Programmation fonctionnelle

Répétition 2

18 février 2010

## Encore quelques évaluations

**Exercice 1.**

```
(cons '(1 2 3) '(4 5))
(append '(1 2 3) '(4 5))
(list '(1 2 3) '(4 5))
(list? (cons 'a 'b))
(list? (cons 'a (cons 'b '())))
(list? (cons (cons 'b '()) 'a))
(list? (cons (cons 'b '()) (cons 'b '())))
(null? 'a)
(null? (car '(a)))
(null? (cdr '(a)))
(number? 1)
(number? '1)
(number? #t)
(number? 'a)
(number? a)
(boolean? 3)
(boolean? #t)
(boolean? #f)
(boolean? '#t)
(boolean? 'd)
(boolean? d)
(boolean? '())
(null? #t)
(null? '())
(null? '(a b))
(null? (car '(())))
(null? (car '((()))))
(null? #f)
(symbol? a)
(symbol? 'b)
(symbol? (car '(a b c)))
(symbol? (cons '() '()))
(symbol? #f)
(equal? 'a (car '(a b)))
```

```
(equal? '(a b c) '(a b c))
(equal? '(a (b c)) '(a b c))
(equal? (cdr '(a c d)) (cdr '(b c d)))
(equal? '(car '((b) c)) (cdr '(a b)))
(lambda (y x) (cons x y))
((lambda (y x) (cons x y)) '() 'a)
(define id (lambda (x) x))
(id 1)
(id '(1 2 3))
(id id)
((id id) (id id))
(((id id) (id id)) 3)
```

# Correction exercices proposés

---

**Exercice 2.**
La variable `phrase` est définie comme suit :

```
(define phrase
  '(((e) x (e r (c (i c e) c (o m (p l (i q))) u e)))))
```

Extraire le `m` au moyen de `car`, `cdr`, `cadr`, ...

---

**Exercice 3.**
Construire la liste `(S C H E M E)` au seul moyen de la liste `ls` définie comme suit :

```
(define ls '(C E H M S))
```

et des fonctions `cons`, `car`, `cdr`, `cadr`, ...

---

**Exercice 4.**
Définir la fonction `min` qui renvoie le minimum de la liste non vide de nombres donnée en argument.

---

**Exercice 5.**
Ecrire une fonction `big` qui prend comme arguments un nombre entier `n` et une liste `l` de nombres entiers, telle que `(big n l)` est la liste des éléments de `l` plus grands que `n`.

`(big 5 '(7 -3 6 1 -2 5 6))` $\Rightarrow$ `(7 6 6)`

# Compréhension de la structure interne

---

**Exercice 6.**
Représentation interne des listes :
Dessiner l'arbre correspondant à la représentation interne de la liste

(a b c)

---

**Exercice 7.**
Différence entre cons, list et append :

Illustrer graphiquement les opérations suivantes :

- Ajout de l'élement `a` au début de la liste (4 5)
  → (cons 'a '(4 5))

- Création d'une liste à deux éléments, respectivement `a` et la liste (4 5)
  → (list 'a '(4 5))

- Concatenation des listes (1 2 3) et (4 5)
  → (append '(1 2 3) '(4 5))

- Ajout de la liste (1 2 3) au début de la liste (4 5)
  → (cons '(1 2 3) '(4 5))

- Création d'une paire pointée dont les membres sont a et 2
  → (cons 'a 2)

# Notions d'efficacité

---

**Exercice 8.**
Définir la fonction `length-list` qui renvoie la longueur de la liste passée en argument.

Remarque : la fonction *length* est prédéfinie, nous la redéfinissons sous un autre nom à titre d'exercice.

---

**Exercice 9.**
Définir la fonction `reverse-list` qui prend en argument une liste *ls* et renvoie une liste contenant les éléments de *ls* dans l'ordre inverse.

Remarque : la fonction *reverse* est prédéfinie, nous la redéfinissons sous un autre nom à titre d'exercice.

# Exercices proposés

---

**Exercice 10.**
Ecrire une fonction `removeAll` à deux arguments `l` et `n` dont les valeurs sont respectivement une liste et un nombre entier, qui retourne la liste privée de toutes les occurences de `n`.

```
(removeAll '(2 7 1 7 3 1) 1) ⇒ (2 7 7 3)
```

---

**Exercice 11.**
Définir la fonction `suffix` à deux arguments `l1` et `l2` dont les valeurs sont des listes et qui retourne `#t` si `l1` est suffixe de `l2` et `#f` sinon.

```
(suffix '(1 2) '(3 x 1 2)) ⇒ #t
(suffix '(1 2) '(3 x 2)) ⇒ #f
```

---

**Exercice 12.**
Ecrire une fonction `frequency` prenant en argument une liste d'atomes `ls` et renvoyant une table d'apparition de chacun des atomes dans la liste `ls`. Cette table sera représentée par une liste de paires pointées, dont le car est un atome et le cdr la fréquence d'apparition de cet atome. Tous les atomes de `ls` apparaissent une et une seule fois dans la table.

```
(frequency '(a b c b a b d a c b b))
   ⇒ ((a . 3) (b . 5) (c . 2) (d . 1))
```

# Programmation fonctionnelle

Répétition 3

25 février 2010

## Exercices sur les listes

**Exercice 1.**
Définir la fonction `nombre-de` à deux arguments qui renvoie le nombre de fois que l'on a rencontré le premier argument dans la liste en second argument.

**Exercice 2.**
Définir la fonction `make-list` qui prend comme arguments un nombre `n` et une s-expression quelconque `x` et qui construit une liste composée de `n` fois l'élément `x`.

**Exercice 3.**
Les nombres de Gribomont sont définis comme suit :

si $n < 3$, $f(n) = n$
si $n \geq 3$, $f(n) = f(n-1) + f(n-2) + f(n-3)$.

Ecrire une fonction qui permet de calculer les nombres de Gribomont.

## Correction exercices proposés

**Exercice 4.**
Ecrire une fonction `removeAll` à deux arguments `l` et `n` dont les valeurs sont respectivement une liste et un nombre entier, qui retourne la liste privée de toutes les occurences de `n`.

`(removeAll '(2 7 1 7 3 1) 1)` ⇒ `(2 7 7 3)`

---

**Exercice 5.**
Définir la fonction `suffix` à deux arguments `l1` et `l2` dont les valeurs sont des listes et qui retourne `#t` si `l1` est suffixe de `l2` et `#f` sinon.

```
(suffix '(1 2) '(3 x 1 2)) ⇒ #t
(suffix '(1 2) '(3 x 2)) ⇒ #f
```

---

**Exercice 6.**
Ecrire une fonction `frequency` prenant en argument une liste d'atomes `ls` et renvoyant une table d'apparition de chacun des atomes dans la liste `ls`. Cette table sera représentée par une liste de paires pointées, dont le car est un atome et le cdr la fréquence d'apparition de cet atome. Tous les atomes de `ls` apparaissent une et une seule fois dans la table.

```
(frequency '(a b c b a b d a c b b))
   ⇒ ((a . 3) (b . 5) (c . 2) (d . 1))
```

# Première interrogation

---

**Exercice 7.**
Correction interros du 6 mars 2008 et du 5 mars 2009.

# Programmation fonctionnelle

Répétition 4

11 mars 2010

## Expression 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)))))
```

## Prédicats

**Exercice 2.**
Écrire une fonction `filter` prenant deux arguments, une liste `l` d'objets et un prédicat unaire `p`, renvoyant la liste des éléments de `l` pour lesquels `p` est vrai.

Définir alors au moyen de la fonction précédente, la fonction `greater` prenant comme arguments une liste `l` de nombres entiers et un nombre entier `a`, et renvoyant la liste des éléments de `l` strictement supérieurs à `a`.

`(greater '(5 2 7 4 3 6) 4)` ⇒ `(5 7 6)`

## Récursion profonde sur les listes

**Exercice 3.**
Définir la procédure `count-all` à deux arguments, un élément et une liste, et qui compte, en profondeur, le nombre de de fois que l'élément est contenu dans la liste.

`(count-all 1 '(0 (1 2 (3 4 (1)) (3 (2 1) 1) 1) 0 (1 2 (1 2 3))))` ⇒ `7`

# Compositions de fonctions

---

**Exercice 4.**
Définir une fonction `compose-n` qui renvoie la fonction unaire donnée en argument `n` fois composée avec elle-même.

*Variante* :
Écrire une fonction `compose-fgf` qui prend comme argument une fonction $f$ et renvoie une fonction qui prend comme argument une fonction $g$ et qui renvoie la fonction $f \circ g \circ f$.

*Variante 2* :
Écrire une fonction `compose-fgab` qui prend comme argument deux fonctions $f$ et $g$, ainsi que deux entiers $a$ et $b$ et renvoie la fonction

$$\underbrace{f \circ \ldots f}_{a} \circ \underbrace{g \circ \ldots g}_{b}$$

*Variante 3* :
Écrire une fonction `compose-fa` qui prend comme argument une fonction $f$ et deux entier $a, b$ et renvoie la fonction

$$x \mapsto \underbrace{f \circ \ldots f}_{a}(b^x)$$

# Arbres

---

**Exercice 5.**
Tout nœud interne d'un `arbre arithmétique` a exactement deux fils et est étiqueté par l'un des symboles `add`, `sub`, `mul` et `div` ; toute feuille est étiqueté par un nombre entier.

Écrire la fonction `value` qui prend comme argument un arbre arithmétique et qui renvoie la valeur de l'expression arithmétique associée si l'évaluation de celle-ci n'implique aucune division par 0 ; sinon, `value` renvoie `#f`.

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

# Somme

---

**Exercice 6.**

$$f(n) = \sum_{i=0}^{n-1} \Big( ((f(i) + 2) * (f(n - i - 1) + 3)) \, mod \, \big(n^2 + i + 5\big) \Big)$$

# Exercices proposés

---

**Exercice 7.**
Écrire un prédicat `prime?` qui détermine si un entier strictement positif est premier ou non.

---

**Exercice 8.**
Un carré magique de dimension n est un tableau de taille $n \times n$ dans lequel chaque nombre de 1 à $n \times n$ apparaît une fois, et dont les sommes des lignes, des colonnes et des diagonales sont égales. Écrire un prédicat magic ? qui vérifie si un carré de nombres est magique ou non. On représente un carré de nombres de dimension `n` par une liste de `n` listes contenant chacune `n` éléments.

```
(magic? '((4 9 2) (3 5 7) (8 1 6))) ⇒ #t

(magic? '((4 4 7) (5 3 7) (8 1 6))) ⇒ #f
```

# Programmation fonctionnelle

Répétition 5

18 mars 2010

## Correction exercices proposés

**Exercice 1.**
Écrire un prédicat `prime?` qui détermine si un entier strictement positif est premier ou non.

**Exercice 2.**
Un carré magique de dimension n est un tableau de taille $n \times n$ dans lequel chaque nombre de 1 à $n \times n$ apparaît une fois, et dont les sommes des lignes, des colonnes et des diagonales sont égales. Écrire un prédicat magic ? qui vérifie si un carré de nombres est magique ou non. On représente un carré de nombres de dimension `n` par une liste de `n` listes contenant chacune `n` éléments.

`(magic? '((4 9 2) (3 5 7) (8 1 6)))` $\Rightarrow$ `#t`

`(magic? '((4 4 7) (5 3 7) (8 1 6)))` $\Rightarrow$ `#f`

## Abstraction sur les données

**Exercice 3.**
On décide de représenter les nombres complexes par des paires pointées dont le `car` est la partie réelle et le `cdr` la partie imaginaire. Ecrire les fonctions

```
(make-from-real-imag re im)
(make-from-ang-imag rho ang)
(add-complex z1 z2)
(sub-complex z1 z2)
(mul-complex z1 z2)
(div-complex z1 z2)
(real-part z)
(imag-part z)
(magnitude z)
(angle z)
```

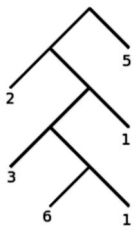
# Exercices sur les arbres

---

**Exercice 4.**

Ecrire une fonction `depth-first` qui prend comme argument un arbre binaire complet — chaque nœud a 0 (une feuille) ou 2 fils (un nœud interne) — dont uniquement les feuilles sont étiquetées, et renvoie la liste des étiquettes des feuilles, obtenue par un parcours en profondeur d'abord, et de gauche à droite, de l'arbre.

Par exemple, `depth-first` appliquée à l'arbre




renvoie la liste `(2 3 6 1 1 5)`.

On choisira et spécifiera une représentation adéquate des arbres.

---

**Exercice 5.**

On considère des arbres binaires complets (chaque nœud a exactement 0 ou 2 fils) dont les feuilles sont étiquetées par des symboles atomiques. Les nœuds internes ne sont pas étiquetés.

Soit `a` un tel arbre. Simplifier `a` consiste à supprimer dans cet arbre les feuilles redondantes. Ainsi tout nœud ayant deux fils étiquetés par le même symbole est remplacé par ce symbole et ainsi de suite.

Ecrire une fonction `simplify` qui prend pour argument un arbre binaire complet `a` et qui retourne l'arbre binaire complet simplifié correspondant.

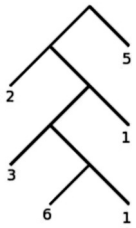
# Exercices proposés

---

**Exercice 6.**

Ecrire une fonction `breadth-first` qui prend comme argument un arbre binaire complet — chaque nœud a 0 (une feuille) ou 2 fils (un nœud interne) — dont uniquement les feuilles sont étiquetées, et renvoie la liste des étiquettes des feuilles, obtenue par un parcours en largeur d'abord, et de gauche à droite, de l'arbre.

Par exemple, `breadth-first` appliquée à l'arbre




renvoie la liste `(5 2 1 3 6 1)`.

On choisira et spécifiera une représentation adéquate des arbres.

*Variante* : même exercice avec un arbre non plus binaire mais avec un nombre quelconque de fils, dont tous les nœuds sont étiquetés.

---

**Exercice 7.**

Un entier naturel est *parfait* s'il est égal à la somme de ses diviseurs positifs propres. Ecrire un prédicat `perfect?` déterminant si son argument est un nombre parfait.

Le nombre 28 est parfait parce que $28 = 1 + 2 + 4 + 7 + 14$ ;
le nombre 32 n'est pas parfait parce que $32 \neq 1 + 2 + 4 + 8 + 16$.

# Programmation fonctionnelle
# 6 mars 2008

**Question 1.** Evaluer les expressions suivantes

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

(define f ((lambda (x y) (x y)) (lambda (x) x) (lambda (y) y)))
(f 0)

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

**Question 2.** Soit la fonction $f$ qui prend comme arguments deux entiers naturels et qui vérifie les équations suivantes, pour toutes les valeurs possibles des variables $m$ et $n$ :

$$f(0,m) = m\,,$$
$$f(n+1,m) = mf(0,m) + (m+1)f(1,m) + \cdots + (m+n)f(n,m)\,.$$

Ecrire un programme `f` pour cette fonction.

**Question 3.** Une relation binaire sur un ensemble $E$ est un ensemble de couples d'éléments de $E$. Une relation binaire sur $E$ est *symétrique* si, pour tout couple $(a,b)$ appartenant à la relation, le couple $(b,a)$ appartient aussi à la relation. En Scheme, on représentera un couple par une paire pointée et un ensemble par une liste (sans répétition). Ecrire un prédicat `sym ?` prenant comme argument une relation binaire et renvoyant `#t` si cette relation est symétrique.

**Consignes.** Spécifier les fonctions auxiliaires éventuelles, même celles définies localement.
Les fonctions prédéfinies ne doivent pas être spécifiées ni redéfinies.
Répondre à chaque question sur une feuille A4 séparée.
Ne pas utiliser de crayon, ne pas utiliser de rouge.
Mentionner nom, prénom, *section* et numéro de la question sur chaque feuille.

# Programmation fonctionnelle
# 5 mars 2009

**Question 1.** Un $\mathbb{N}$-arbre est un nombre ou une liste non vide dont chaque élément est un $\mathbb{N}$-arbre. Ecrire une fonction `f` prenant comme argument un $\mathbb{N}$-arbre et renvoyant comme résultat un $\mathbb{N}$-arbre de même structure, dans lequel chaque nombre $x$ a été remplacé par $x + 1$.

**Question 2.** Le *nombre de Bell* $B_n$ est le nombre de partitions d'un ensemble de $n$ éléments. On a $B_0 = 1$ et, pour tout $n > 0$, $B_n = \sum_{i=0}^{n-1} B_i \mathbf{C}_{n-1}^{i}$. Ecrire une fonction `bell` prenant comme argument un entier naturel $n$ et renvoyant comme résultat $B_n$. (On écrira d'abord une version naïve puis une version efficace.)

**Consignes.** Spécifier les fonctions auxiliaires éventuelles, même celles définies localement.
Les fonctions prédéfinies ne doivent pas être spécifiées ni redéfinies.
Répondre à chaque question sur une feuille A4 séparée.
Ne pas utiliser de crayon, ne pas utiliser de rouge.
Mentionner nom, prénom, *section* et numéro de la question sur chaque feuille.

# Programmation fonctionnelle

Répétition 6

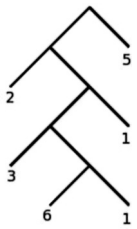
25 mars 2010

## Correction exercices proposés

**Exercice 1.**

Ecrire une fonction `breadth-first` qui prend comme argument un arbre binaire complet — chaque nœud a 0 (une feuille) ou 2 fils (un nœud interne) — dont uniquement les feuilles sont étiquetées, et renvoie la liste des étiquettes des feuilles, obtenue par un parcours en largeur d'abord, et de gauche à droite, de l'arbre.

Par exemple, la fonction `breadth-first` appliquée à l'arbre




renvoie la liste `(5 2 1 3 6 1)`.

On choisira et spécifiera une représentation adéquate des arbres.

*Variante* : même exercice avec un arbre non plus binaire mais avec un nombre quelconque de fils, dont tous les nœuds sont étiquetés.

**Exercice 2.**

Un entier naturel est *parfait* s'il est égal à la somme de ses diviseurs positifs propres. Écrire un prédicat `perfect?` déterminant si son argument est un nombre parfait.

Le nombre 28 est parfait parce que $28 = 1 + 2 + 4 + 7 + 14$ ;
le nombre 32 n'est pas parfait parce que $32 \neq 1 + 2 + 4 + 8 + 16$.

# N-arbre

---

**Exercice 3.**

Un $N$-*arbre* est un arbre dont :

- chaque nœud est étiqueté par un nombre naturel, son label ;
- chaque nœud a 0 ou 1 fils gauche ;
- chaque nœud a 0 ou 1 fils droit.

Un $N$-arbre est représenté par une liste de trois éléments. Le premier est le label de la racine, le deuxième est la représentation du fils gauche s'il existe et la liste vide sinon. De même, le troisième élément est la représentation du fils droit s'il existe et la liste vide sinon. Un $N$-arbre est *conditionné* si le label de tout nœud interne est plus grand ou égal aux labels de tous ses descendants de gauche, et plus petit ou égal aux labels de tous ses descendants de droite.

1. Ecrire un prédicat `ntree?` à un argument, qui renvoie `#t` si son argument représente un $N$-arbre et `#f` sinon.
   `(ntree? '(5 (4 () (7 () ())) (3 () ())))` ⇒ `#t`

2. Ecrire un prédicat `condit?` prenant comme argument un $N$-arbre et renvoyant `#t` si ce $N$-arbre est conditionné et `#f` sinon.
   `(condit? '(5 (4 () (7 () ())) (3 () ())))` ⇒ `#f`

3. Ecrire un prédicat `present?` prenant comme arguments un $N$-arbre conditionné et un nombre naturel et renvoyant `#t` si le nombre est le label d'un nœud de l'arbre, et `#f` sinon.

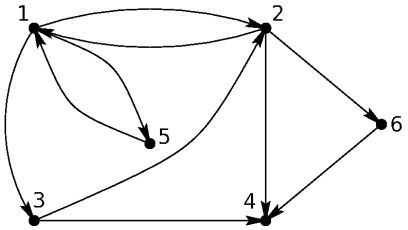
# Exercice sur les graphes

---

**Exercice 4.**

Écrire une fonction `inv` qui, à partir d'un graphe orienté, construit le graphe retourné correspondant. Nous convenons de représenter un graphe comportant $n$ nœuds par une liste de $n$ listes ; la liste correspondant au nœud $x$ a pour premier élément $x$ et pour éléments suivants les successeurs (immédiats) de $x$.

Par exemple le graphe




sera représenté par la liste

```
((1 2 3 5) (2 1 4 6) (3 2 4) (4) (5 1) (6 4))
```

# Fonctions

---

**Exercice 5.**

Définir les fonctions `symmetrize` et `anti-symmetrize`, qui prennent comme argument une fonction $f : R \to R$ et qui renvoient respectivement les fonctions

$$f' : R \to R \qquad x \mapsto \frac{f(x) + f(-x)}{2}$$

et

$$f' : R \to R \qquad x \mapsto \frac{f(x) - f(-x)}{2}$$

Définir ensuite une fonction `func-op` à trois arguments, un opérateur `op` de $R \times R \to R$, et deux fonctions unaires `f` et `g`. `func-op` renvoie la fonction unaire $h$ telle que

$$\forall x \in R : h\,(x) = \texttt{op}\,(\texttt{f}(x), \texttt{g}(x))$$

Redéfinir ensuite `symmetrize` et `anti-symmetrize` à partir de `func-op`.

# Sous-listes

---

**Exercice 6.**

Écrire une fonction `sublists` prenant comme argument une liste `l` et retournant la liste des sous-listes de `l`. (Une sous-liste de `l` est une liste de 0, 1 ou plusieurs éléments consécutifs de `l`.)

# Exercices proposés

---

**Exercice 7.**
Un arbre n-aire est un arbre dont chaque nœud admet un nombre quelconque de fils ; seules les feuilles sont étiquetées, l'étiquette étant un nombre naturel. La fonction $f$ prend comme argument un arbre n-aire $a$ et renvoie l'arbre n-aire $f(a)$ obtenu en remplaçant chaque feuille de $a$ étiquetée $n > 0$ par un nœud interne dont les $n$ fils sont des feuilles étiquetées $n - 1$. On demande de programmer la fonction $f$.

---

**Exercice 8.**
Ecrire une fonction `longest-inc` retournant la plus longue sous-liste croissante de la liste d'entiers donnée en argument.

---

**Exercice 9.**
Écrire une fonction qui prend en argument une liste et renvoie cette liste dont on a supprimé 3 éléments sur 5. On commence par garder les deux premiers éléments, puis on supprime les trois suivants, puis on garde les deux suivants, ...

# Programmation fonctionnelle

Répétition 7

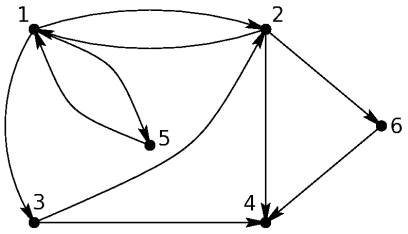
22 avril 2010

## Correction exercices proposés

**Exercice 1.**

Écrire une fonction `inv` qui, à partir d'un graphe orienté, construit le graphe retourné correspondant. Nous convenons de représenter un graphe comportant $n$ nœuds par une liste de $n$ listes ; la liste correspondant au nœud $x$ a pour premier élément $x$ et pour éléments suivants les successeurs (immédiats) de $x$.

Par exemple le graphe




sera représenté par la liste

```
((1 2 3 5) (2 1 4 6) (3 2 4) (4) (5 1) (6 4))
```

**Exercice 2.**

Écrire une fonction `longest-inc` retournant la plus longue sous-liste croissante de la liste d'entiers donnée en argument.

**Exercice 3.**

Écrire une fonction qui prend en argument une liste et renvoie cette liste dont on a supprimé 3 éléments sur 5. On commence par garder les deux premiers éléments, puis on supprime les trois suivants, puis on garde les deux suivants, ...

# Exercice sur les listes de listes

---

**Exercice 4.**
Écrire une fonction `lpref` prenant comme argument une liste `u` et retournant la liste des préfixes de `u`. (La liste vide et la liste `u` elle-même sont des préfixes de `u`.)

---

**Exercice 5.**
Écrire une fonction `nbsum` qui prend comme argument un nombre $n$ et qui renvoie le nombre de façons d'écrire une somme égale à $n$ (on comptera une seule fois les commutations).

---

**Exercice 6.**
Une *tricoupure* d'une liste $\ell$ est une liste de trois listes non vides dont la concaténation (dans l'ordre) vaut $\ell$. Écrire une fonction `tricoup` qui à toute liste $\ell$ associe la liste des tricoupures de $\ell$.

Par exemple, si $\ell$ est `(a b)` la liste des tricoupures de $\ell$ est la liste vide ; si $\ell$ est `(a b c d)` la liste des tricoupures de $\ell$ comporte, dans un ordre quelconque, les trois listes `((a b) (c) (d))`, `((a) (b c) (d))` et `((a) (b) (c d))`.

**Variante :**

Une *tricoupure* d'une liste $\ell$ est une liste de trois listes dont la concaténation (dans l'ordre) vaut $\ell$. Écrire une fonction `tricoup-lv` qui à toute liste $\ell$ associe la liste des tricoupures de $\ell$.

Par exemple, si $\ell$ est `(a)` la liste des tricoupures de $\ell$ comporte, dans un ordre quelconque, les trois listes `((a) () ())`, `(() (a) ())` et `(() () (a))`.

---

**Exercice 7.**
Écrire une fonction qui renvoie la liste des permutations d'une liste donnée.

# Exercice proposé

---

**Exercice 8.**
Écrire une fonction qui renvoie la liste des sous-ensembles d'un ensemble donné.

`(lset '(a b c))` ⇒ `(() (a) (a b) (a b c) (a c) (b) (b c) (c))`

# Programmation fonctionnelle

Répétition 8

20 mai 2010

## Correction exercices proposés

**Exercice 1.**
Écrire une fonction qui renvoie la liste des permutations d'une liste donnée.

**Exercice 2.**
Écrire une fonction qui renvoie la liste des sous-ensembles d'un ensemble donné.

`(lset '(a b c))` ⇒ `(() (a) (a b) (a b c) (a c) (b) (b c) (c))`

## Exercices

**Exercice 3.**
Soit un alphabet décrit par une liste de symboles. Écrire une fonction `words` qui engendre la liste (dans un ordre quelconque) des mots de longueur n écrits dans cet alphabet.

```
(words 2 '(a b c)) ==>
((a a) (b a) (c a) (a b) (b b) (c b) (a c) (b c) (c c))
```

**Exercice 4.**
Générer à partir d'une liste, tous les arbres binaires complets dont la lecture des feuilles de gauche à droite donne la liste.

Par exemple, si on représente un nœud par une liste et une feuille par l'atome étiquette.

```
(binaryTrees '(a b c d)) ==>
((a (b (c d))) ((a b) (c d)) (a ((b c) d)) ((a (b c)) d)
 (((a b) c) d))
```

**Exercice 5.**
Écrire la fct `lagrange`, prenant comme argument un entier naturel n et renvoyant la liste de tous les quadruplets d'entiers naturels $(a, b, c, d)$ tels que $a^2 + b^2 + c^2 + d^2 = n$.

```
(lagrange 13) ==>
((0 0 2 3) (0 0 3 2) (0 2 0 3) (0 2 3 0) (0 3 0 2) (0 3 2 0)
 (1 2 2 2) (2 0 0 3) (2 0 3 0) (2 1 2 2) (2 2 1 2) (2 2 2 1)
 (2 3 0 0) (3 0 0 2) (3 0 2 0) (3 2 0 0))

(lagrange 18)
((0 0 3 3) (0 1 1 4) (0 1 4 1) (0 3 0 3) (0 3 3 0) (0 4 1 1)
 (1 0 1 4) (1 0 4 1) (1 1 0 4) (1 1 4 0) (1 2 2 3) (1 2 3 2)
 (1 3 2 2) (1 4 0 1) (1 4 1 0) (2 1 2 3) (2 1 3 2) (2 2 1 3)
 (2 2 3 1) (2 3 1 2) (2 3 2 1) (3 0 0 3) (3 0 3 0) (3 1 2 2)
 (3 2 1 2) (3 2 2 1) (3 3 0 0) (4 0 1 1) (4 1 0 1) (4 1 1 0))
```

---

**Exercice 6.**
Dérivée n-ième d'un polynôme.

---

**Exercice 7.**
Écrire une fonction qui à deux polynômes $P$ et $Q$ associe le polynôme composé $P \circ Q$.

Rappel : Si $P(x) = x^3 - x + 1$ et si $Q(x) = x^2 - 2$, on a
$(P \circ Q)(x) = P(Q(x)) = (x^2 - 2)^3 - (x^2 - 2) + 1$ et
$(Q \circ P)(x) = Q(P(x)) = (x^3 - x + 1)^2 - 2$.