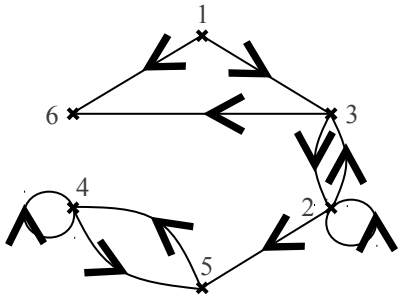
ISIMA 1ère année

18 février 2010
Durée : 2 heures
Documents de cours autorisés

# ***PROGRAMMATION FONCTIONNELLE***

Exercice 1 :

Dans cet exercice, un graphe orienté est représenté par une liste de couples du type ($x$ ($s_1$ … $s_n$)) dans lequel le premier élément est un sommet $x$ du graphe et le second élément une liste de sommets qui sont les successeurs de $x$ dans ce graphe.
Par exemple, la liste `((1 (3 6)) (2 (2 3 5)) (3 (2 6)) (4 (4 5)) (5 (4)) (6 ()))` représente le graphe orienté ci-dessous :




1°) Ecrire les fonctions utilitaires suivantes.

a) Une fonction `membre?` ayant comme arguments un élément `x`, de type quelconque, et une liste `L`, et telle que l'évaluation de l'expression `(membre? x L)` retourne `#t` si `x` est un élément de `L` et `#f` sinon.

Solution :

```
(define membre (lambda (x L)
   (if (null? L) #f
       (if (equal? (car L) x) #t (membre x (cdr L))) ) ))
```

b) Une fonction `SD` ayant comme argument une liste `L`, et telle que l'évaluation de l'expression `(SD L)` retourne la liste `L` sans doublon.

Solution :

```
(define SD (lambda (L)
   (if (null? L) ()
       (let ((SDcdrL (SD (cdr L))))
            (if (membre? (car L) SDcdrL)
                SDcdrL
               (cons (car L) SDcdrL) ) ) ) ))
```

c) Une fonction `moins` ayant comme arguments deux listes `L1` et `L2`, et telle que l'évaluation de l'expression `(moins L1 L2)` retourne la liste contenant les éléments de `L1` qui n'appartiennent pas à `L2`.

Solution :

```
(define moins (lambda (L1 L2)
   (if (null? L1) ()
       (if (membre? (car L1) L2)
           (moins (cdr L1) L2)
           (cons (car L1) (moins (cdr L1) L2)) ) ) ))
```

d) Une fonction `successeurs` ayant comme arguments une liste de couples, `G`, représentant un graphe orienté suivant la convention décrite ci-dessus et un sommet `s` de `G`, et telle que l'évaluation de l'expression `(successeurs G s)` retourne la liste des successeurs de `s` dans `G` (dans un ordre quelconque).

Par exemple, en prenant comme argument `G` le graphe donné en exemple ci-dessus, l'évaluation de l'expression `(successeurs G 1)` doit retourner `(3 6)`.

Solution :

```
(define successeurs (lambda (G s)
   (if (null? G) ()
       (if (equal? s (caar G))
           (cadar G)
           (successeurs (cdr G) s) ) ) ))
```

2°) Ecrire une fonction `Rn` ayant trois arguments :

- une liste de couples, `G`, représentant un graphe orienté suivant la convention décrite ci-dessus ;
- un sommet `s` de `G` ;
- et un entier `n` ;

et telle que l'évaluation de l'expression `(Rn G s n)` retourne la liste des sommets de `G` (dans un ordre quelconque) qui sont accessibles à partir du sommet `s` en suivant un chemin de longueur `n`.

Par exemple, en prenant comme argument `G` le graphe donné en exemple ci-dessus :

- l'évaluation de l'expression `(Rn G 1 0)` doit retourner `(1)` ;
- l'évaluation de l'expression `(Rn G 1 1)` doit retourner `(3 6)` ;
- l'évaluation de l'expression `(Rn G 1 2)` doit retourner `(2 6)` ;
- l'évaluation de l'expression `(Rn G 1 3)` doit retourner `(2 3 5)`.

Solution :

```
(define Rn (lambda (G s n)
   (if (= n 0) (list s)
       (SD (append_map (lambda (x) (successeurs G x))
                       (Rn G s (- n 1)) )) ) ))
```

3°) Ecrire une fonction `R*n` ayant trois arguments :

- une liste de couples, `G`, représentant un graphe orienté suivant la convention décrite ci-dessus ;
- un sommet `s` de `G` ;
- et un entier `n` ;

et telle que l'évaluation de l'expression `(R*n G s n)` retourne la liste des sommets de `G` (dans un ordre quelconque) qui sont accessibles à partir du sommet `s` en suivant un chemin de longueur inférieure ou égale à `n`.

Par exemple, en prenant comme argument `G` le graphe donné en exemple ci-dessus :

- l'évaluation de l'expression `(R*n G 1 0)` doit retourner `(1)` ;
- l'évaluation de l'expression `(R*n G 1 1)` doit retourner `(1 3 6)` ;
- l'évaluation de l'expression `(R*n G 1 2)` doit retourner `(1 2 3 6)` ;

- l'évaluation de l'expression `(R*n G 1 3)` doit retourner `(1 2 3 5 6)`.

Remarque : ***il est demandé d'écrire la fonction de telle façon que les successeurs d'un sommet ne soient recherchés qu'une seule fois***.

Solution :

```
(define R*n (lambda (G s n) (R*nacc G s n 0 (list s) ())))
(define R*nacc (lambda (G s n k Lnouveaux Lanciens)
   (if (< k n)
       (let ((Lnouv
(moins (append_map (lambda (x) (successeurs G x)) Lnouveaux)
        Lanciens ) ))
         (R*nacc G s n (+ k 1) Lnouv (append Lnouveaux Lanciens)) )
       (append Lnouveaux Lanciens) ) ))
```

Exercice 2 :

Nous considérerons ici qu'une image en niveau de gris est représentée par une liste de lignes de taille identique.

Chaque ligne est représentée par une liste de pixels dont la valeur est comprise entre 0 et 255. Par exemple la liste ((3 0 0 0 0 15 15 ) (9 9 9 9 0 0 0) (0 0 0 0 0 0 254)) est une image – elle sera notée I dans la suite.

1°) Une manière de compresser une image peut être de remplacer chaque suite contigüe de n fois le même pixel 'p' par une liste (n p)

Ecrire une fonction **`compress`** prenant en paramètre une image `Img` et telle que l'évaluation `(compress Img)` retourne une image compressée.

Ex : `(compress I)` retournera la liste (((1 3) (4 0) (2 15)) ((4 9) (3 0)) ((6 0) (1 254)))

Solution :

```
(define compress (lambda (I) (map comp I)))
(define comp (lambda (L)
   (if (null? (cdr L))
       (list (list 1 (car L)))
       (let ((compcdrL (comp (cdr L))))
          (if (equal? (car L) (cadr L))
              (cons (list (+ 1 (caar L)) (car L)) (cdr compcdrL))
              (cons (list 1 (car L)) compcdrL) ) ) ) ))
```

3°) Ecrire une fonction **`decompress`** qui à partir d'une image compressée fournit l'image initiale.

Solution :

```
(define decompress (lambda (I) (map decomp I)))
(define decomp (lambda (L)
   (if (null? L) ()
       (cons (listn (caar L) (cadar L)) (decomp (cdr L))) ) ))
(define listn (lambda (n e)
   (if (= n 0) () (cons e (listn (- n 1) e))) ))
```

3°) Ecrire une fonction **`renv_image`** qui permette de faire pivoter une image de 180 degrés. [Vous pourrez utiliser la fonction `renverse` telle que `(renverse L)` retourne la liste L dans l'ordre inverse de ses éléments.].

Solution :

```
(define renv_image (lambda (I) (map renv_ligne I)))
(define renv_ligne (lambda (L) (rl L ())))
(define rl (lambda (L LR)
   (if (null? L) LR (rl (cdr L) (cons (car L) LR))) ))
```

4°) Nous souhaitons écrire une fonction **`extrait`** permettant de vérifier qu'une image P est incluse dans une image `Img`, `Img` et P étant compressées et P plus petite que `Img`.

a) Proposer un algorithme en fonction des structures de données utilisées.

Solution :

Il faut d'abord décompresser les images. La fonction `ext` a comme arguments les images `P` et `Img` décompressée, et retourne la sous image (non compressée) de `Img` contenant `P`. Elle utilise également comme arguments les nombre de lignes et colonnes des images `P` et `Img`.

Si `P` est un "préfixe" de `Img` (c'est-à-dire si les `nblP` lignes de `P` sont les débuts des `nblP` premières lignes de `Img`), alors, il suffit de retourner `Img`. Sinon l'appel récursif peut être fait en "décalant" `P` vers la droite horizontalement (appel récursif sur `(map cdr Img)`) si cela est possible. Si cet appel récursif retourne une sous image, `ext` retourne cette sous image, sinon si l'appel récursif retourne `#f`, un autre appel récursif peut être fait en "descendant" `P` (appel récursif sur `(cdr Img)`).

b) Ecrire la fonction scheme correspondante telle que `(extrait P Img)` retourne, sous forme compressée, une sous image de `Img`, où P est en haut à gauche, si cela est possible et #f sinon.

Solution :

```
(define extrait (lambda (P Img)
   (let ((SI (ext (decompress P) (decompress Img)
                  (length P) (length Img)
                  (length (car P)) (length (car Img)) )))
      (if SI (compress SI) #f) ) ))

(define ext (lambda (P Img nblP nblImg nbcP nbcImg)
   (cond ((prefixe P Img) Img)
         ((> nbcImg nbcP)
          (extligne P (map cdr Img) nbcP (- nbcImg 1)) )
         ((> nblImg nblP)
          (ext P (cdr Img) nblP (- nblImg 1) nbcP nbcImg) )
         (else #f) ) ))

(define extligne (lambda (P Img nbcP nbcImg)
   (cond ((prefixe P Img) Img)
         ((> nbcImg nbcP)
          (extligne P (map cdr Img) nbcP (- nbcImg 1)) )
         (else #f) ) ))
```

```
(define prefixe (lambda (P Img)
   (if null? P) #t
       (and (ligneprefixe (car P) (car Img))
            (prefixe (cdr P) (cdr Img)) ) ) ))
(define ligneprefixe (lambda (LP Limg)
   (if (null? LP) #t
       (and (= (car LP) (car Limg))
            (ligneprefixe (cdr Lp) (cdr Limg)) ) ) ))
```