ISIMA 1ère année

3 septembre 2008

Durée : 2 heures

Documents autorisés

PROGRAMMATION FONCTIONNELLE

Problème : Les tours de Hanoï

Le jeu des tours de Hanoï est constitué par trois piles de disques, qui peuvent être numérotées 1, 2 et 3 comme sur la figure ci-dessus. Les disques sont toujours empilés de façon à ce que le diamètre des disques aille en décroissant quand on monte dans la pile : le plus grand disque d'un empilement se trouve toujours au fond, le plus petit au sommet, et tous les disques au dessus d'un disque donné sont de diamètre inférieur à celui-ci. Dans sa configuration initiale, tous les disques se trouvent sur une même pile. Le principe de ce jeu est de déplacer tout le contenu d'une pile sur une autre pile en effectuant uniquement des déplacements élémentaires. Un *déplacement élémentaire* consiste déplacer un disque se trouvant au sommet d'une pile sur le sommet d'une autre pile, tout en respectant la contrainte qui impose que tous les disques situés en dessous du disque déplacé soient de diamètre supérieur à celui-ci.
Ce jeu peut être représenté par une liste de trois sous-listes d'entiers, chaque sous-liste représentant une pile. Ces sous-listes sont triées par ordre décroissant, l'entier le plus grand, en tête de liste, représentant le disque le plus grand, au fond de la pile, et l'entier le plus petit, en fin de liste, représentant le disque le plus petit, au sommet de la pile.
Dans sa configuration initiale, le jeu a donc deux sous-listes vides, comme par exemple dans `((3 2 1) () ())`. Si l'on veut, à partir de cette configuration initiale, déplacer le contenu de la pile 1 (à gauche) sur la pile 3 (à droite) on va effectuer l'enchaînement de déplacements élémentaires suivant :

```
((3 2 1) ()     ()       )
((3 2)   ()     (1)      )
((3)     (2)    (1)      )
((3)     (2 1)  ()       )
(()      (2 1)  (3)      )
((1)     (2)    (3)      )
((1)     ()     (3 2)    )
(()      ()     (3 2 1)  )
```

On peut remarquer que cela revient à déplacer, de manière récursive, les disques 2 et 1 de la pile de gauche vers la pile du milieu, à effectuer le déplacement élémentaire du disque 3 vers la pile de droite, puis à redéplacer à nouveau, et toujours de manière récursive, les disques 2 et 1 de la pile du milieu vers la pile de droite.

Dans la suite de ce problème, on appellera *configuration* toute liste de trois sous-listes d'entiers représentant une configuration valide du jeu, et on appellera *indice de pile* un entier valant 1, 2 ou 3.

Le but de ce problème est d'écrire une fonction `Trace-Hanoi` qui a comme arguments une configuration initiale, `C`, et un indice de pile `a`, et telle que l'évaluation de l'expression `(trace-Hanoi C a)` retourne la liste de toutes les configurations de jeu menant, de

déplacement élémentaire en déplacement élémentaire, de la configuration initiale (où tous les disques sont empilés sur une pile `d` à déterminer à partir de `C`) à la configuration finale où tous les disques sont empilés sur la pile `a`.
Par exemple l'évaluation de `(Trace-Hanoi '((3 2 1) () ()) 3)` doit retourner

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

1°) Ecrire une fonction `Pile` qui a comme arguments une configuration, `C`, et un indice de pile, `i`, et telle que l'évaluation de l'expression `(Pile C i)` retourne la pile indicée par `i`, c'est-à-dire la `i`ème sous-liste de `C`.
Par exemple l'évaluation de `(pile '(() (2 1) (3)) 2)` doit retourner `(2 1)`.

2°) Ecrire une fonction `cdr2` qui a comme arguments une configuration, `C`, et un indice de pile, `i`, et telle que l'évaluation de l'expression `(cdr2 C i)` retourne la liste obtenue à partir de `C` en supprimant le premier élément de sa `i`ème sous-liste.
Par exemple l'évaluation de `(cdr2 '((3 2 1) () ()) 1)` doit retourner `((2 1) () ())`.

3°) Ecrire une fonction `cons2` qui a comme arguments un entier, `base`, une configuration, `C`, et un indice de pile, `i`, et telle que l'évaluation de l'expression `(cons2 base C i)` retourne la liste obtenue à partir de `C` en insérant l'entier `base` en tête de sa `i`ème sous-liste.
Par exemple l'évaluation de `(cons2 3 '((2 1) () ()) 1)` doit retourner `((3 2 1) () ())`.

4°) Ecrire une fonction `mapcons2` qui a comme arguments un entier, `base` , une liste de listes de configurations, `T`, et un indice de pile, `i`, et telle que l'évaluation de l'expression `(mapcons2 base T i)` retourne la liste obtenue à partir de `T` en insérant l'entier `base` en tête de la `i`ème sous liste de chaque élément de `T`.
Par exemple l'évaluation de
`(mapcons2 3 '(((2 1) () ()) ((2) () (1)) (() (2) (1)) (() (2 1) ()) 1)`
doit retourner `(((3 2 1) () ()) ((3 2) () (1)) ((3) (2) (1)) ((3) (2 1) ()).`

5°) Ecrire une fonction `Permute` qui a comme arguments une configuration, `C`, et deux indices de pile, `i` et `k`, et telle que l'évaluation de l'expression `(Permute C i k)` retourne la liste obtenue à partir de `C` en permutant la `i`ème et la `k`ème sous-liste.
Par exemple l'évaluation de `(permute '((2 1) () ()) 1 2)` doit retourner `(() (2 1) ()).`

6°) En utilisant les fonctions définies précédemment, écrire la fonction `Trace-Hanoi`.

Solution :
1°)
```
(define Pile (lambda (C i)
      (if (= i 1) (car C) (Pile (cdr C) (- i 1))) ))
```

2°)
```
(define cdr2 (lambda (C i)
      (if (= i 1)
         (cons (cdar C) (cdr C))
```

```
          (cons (car C) (cdr2 (cdr C) (- i 1))) ) ))
```

3°)
```
(define cons2 (lambda (base C i)
       (if (= i 1)
          (cons (cons base (car C)) (cdr C))
          (cons (car C) (cons2 base (cdr C) (- i 1))) ) ))
```

4°)
```
(define mapcons2 (lambda (base T i)
       (map (lambda (C) (cons2 base C i)) T) ))
```

5°)
```
(define Permute (lambda (C i k)
      (let ((Pi (pile C i)) (Pk (pile C k)))
         (case i
            (1 (if (= k 2) (list Pk Pi ()) (list Pk () Pi)))
            (2 (if (= k 1) (list Pk Pi ()) (list () Pk Pi)))
            (3 (if (= k 1) (list Pk () Pi) (list () Pi Pk))) ) ) ))
```

6°)
```
(define Trace-Hanoi (lambda (C a)
      (let ((d (Pile-depart C)))
         (if (cdr (pile C d))
             (let* ((t (- 6 a d))
                    (base (car (pile C d)))
                    (T1 (Trace-Hanoi (cdr2 C d) t))
                    (T2 (Trace-Hanoi (Permute (cdr2 C d) d t) a)) )
                (append (mapcons2 base T1 d)
                        (mapcons2 base T2 a) ) )
             (list C (Permute C d a)) ) ) ))
   (define Pile-depart (lambda (C)
      (if (car C) 1 (+ 1 (Pile-depart (cdr C)))) ))
```