ISIMA 1ère année

3 février 2004
Durée : 2 heures
Documents autorisés

PROGRAMMATION FONCTIONNELLE

Problème : **Le problème des huit reines**

Un échiquier est un carré de huit cases de côté. Deux reines sur un échiquier sont en prise si elles se trouvent sur une même ligne, sur une même colonne ou sur une même diagonale. Une case de l'échiquier est repérée par un couple (i, j) où i est le numéro de la ligne (1 pour la ligne inférieure, 8 pour la ligne supérieure) et j est le numéro de la colonne (1 pour la colonne de gauche, 8 pour celle de droite). Il est clair que l'on ne peut placer plus de huit reines sur un échiquier sans que deux d'entre elles soient en prise. Le problème des huit reines consiste à construire toutes les possibilités de placer exactement huit reines sur un échiquier sans que deux d'entre elles soient en prise.
Une configuration acceptable comportera une reine par colonne, chacune se trouvant sur une ligne distincte. On pourra la représenter par une liste de type $(i_1, \ldots, i_8)$, permutation de la liste (1 … 8), les positions correspondantes des reines formant l'ensemble $\{ (i_1, 1), \ldots, (i_8, 8) \}$. Toute liste de ce type satisfait les conditions relatives aux lignes et aux colonnes et sera une solution si elle satisfait en outre la condition relative aux diagonales.
Le but de ce problème est d'écrire une fonction Scheme qui retourne la liste de toutes les configurations acceptables. Une solution consiste à utiliser des configurations intermédiaires de type $(i_k, \ldots, i_8)$, avec $1 \leq k \leq 8$. Par exemple `()` représente l'échiquier vide, `(4)` représente la configuration avec une seule reine en position (4, 8), `(2 4)` représente la configuration avec deux reines en positions respectives (4, 8) et (2, 7). Notons que l'échiquier est donc rempli de la droite (colonne 8) vers la gauche (colonne 1) parce qu'il est plus facile d'insérer des éléments en tête de liste plutôt qu'en fin de liste. Une configuration intermédiaire est acceptable si elle est vide, ou bien avec une seule reine, ou bien si deux reines de cette configuration ne sont jamais en prise.

1°) Ecrire une fonction `acc?` ayant comme argument un entier `i`, représentant un numéro de ligne, et une liste d'entiers `C`, représentant une configuration intermédiaire acceptable, et telle que l'évaluation de `(acc? i C)` retourne une valeur logique vraie si la configuration obtenue en mettant `i` en tête de `C` est acceptable, et une valeur logique fausse sinon.
Exemples :
L'évaluation de `(acc? 7 '(2 4))` doit retourner une valeur logique vraie ;
L'évaluation de `(acc? 3 '(2 4))` doit retourner une valeur logique fausse ;

Solution :
```
(define acc? (lambda (i C)
   (and (not (member i C)) (a? i C 1)) ))
(define a? (lambda (i C k)
   (if C
      (and (not (= i (+ (car C) k)))
           (not (= i (- (car C) k)))
           (a? i (cdr C) (+ k 1)) )
      #t ) ))
```

2°) Ecrire une fonction `liste_acc` ayant comme argument une liste d'entiers `C`, représentant une configuration intermédiaire acceptable de type $(i_k, \ldots, i_8)$, avec $1 < k \leq 8$, et telle que l'évaluation de `(liste_acc C)` retourne la liste de toutes les configurations acceptables construites à partir de `C` en ajoutant une reine dans la colonne $k - 1$.
Exemples :
L'évaluation de `(liste_acc ())` doit retourner `((1) (2) (3) (4) (5) (6) (7) (8))` ;
L'évaluation de `(liste_acc '(4))` doit retourner `((1 4) (2 4) (6 4) (7 4) (8 4))` ;
L'évaluation de `(liste_acc '(2 4))` doit retourner `((5 2 4) (7 2 4) (8 2 4))` ;

Solution :

```
(define liste_acc (lambda (C) (l_a C 1)))
(define l_a (lambda (C n)
   (if (> n 8)
       ()
       (let ((NC (cons n C)) (rec (l_a C (+ n 1))))
          (if (acc? NC) (cons NC rec) rec) ) ) ))
```

3°) Etant donnée une configuration acceptable `C`, la liste de configurations acceptables retournée par l'évaluation de l'expression fonctionnelle `(liste_acc C)` est appelée la propagation de longueur 1 à partir de `C`, que l'on note PC_1. La propagation de longueur 2 à partir de `C`, notée PC_2, est obtenue en concaténant les propagations de longueur 1 à partir des éléments de PC_1. En général, la propagation de longueur n à partir de `C`, notée PC_n, est obtenue en concaténant les propagations de longueur 1 à partir des éléments de PC_{n-1}.
Ecrire une fonction `propagation` ayant comme argument une liste d'entiers `C`, représentant une configuration intermédiaire acceptable, et un entier `n` compris entre 1 et 8, et telle que l'évaluation de l'expression fonctionnelle `(propagation C n)` retourne la propagation de longueur n à partir de `C`.
Exemples :
L'évaluation de `(propagation '(2 4) 1)` doit retourner `((5 2 4) (7 2 4) (8 2 4))` ;
L'évaluation de `(propagation '(5 2 4) 1)` doit retourner `((3 5 2 4) (8 5 2 4))` ;
L'évaluation de `(propagation '(7 2 4) 1)` doit retourner `((3 7 2 4) (5 7 2 4))` ;
L'évaluation de `(propagation '(8 2 4) 1)` doit retourner `((3 8 2 4)(5 8 2 4)(6 8 2 4))` ;
L'évaluation de `(propagation '(2 4) 2)` doit retourner `((3 5 2 4) (8 5 2 4) (3 7 2 4) (5 7 2 4) (3 8 2 4) (5 8 2 4) (6 8 2 4))`;

Solution :
```
(define propagation (lambda (C n)
   (if (> n 1)
       (let ((PCn-1 (propagation C (- n 1))))
          (append_map liste_acc PCn-1) )
       (liste_acc C) ) ))
```

4°) Ecrire finalement la fonction `HR`, sans argument, qui retourne la liste de toutes les configurations acceptables.

Solution :
```
(define HR (lambda ()
   (propagation '(()) 8) ))
```

Exercice 1 :

Ecrire une fonction `f` ayant comme argument une fonction `g` à un seul argument et telle que l'évaluation de l'expression `(f g)` retourne la composition de `g` par elle même. Que retourne l'évaluation de l'expression `(f f)` ?

Exercice 2 :

On suppose que l'on dispose d'une fonction `separe` ayant comme argument une liste `L` et telle que l'évaluation de l'expression `(separe L)` retourne une liste de deux sous listes qui forment une partition en deux parties de `L`.
Par exemple l'évaluation de `(separe '(1 2 3 4))` retourne `((1 3) (2 4))`.
Soit le schéma suivant :
```
(define TF (lambda (L F)
   (letrec ((separe (lambda (L) (...))))
      (if (and L (cdr L))
         (let* ((C (separe L))
                (L1 (car C))
                (L2 (cadr C))
                (TFL1 (TF L1))
                (TFL2 (TF L2)) )
            (F TFL1 TFL2) )
       L ) ) ))
```
Remarque : le `letrec` ne sert qu'à éviter d'avoir à redéfinir `separe` chaque fois que l'on veut utiliser `TF`.

Ecrire, en utilisant le schéma `TF`, 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.

Par exemple l'évaluation de `(SD '(5 1 3 2 2 3))` retourne `(1 2 3 5)`.

Solution :

```
(define SD (lambda (L)
   (TF L FSD) ))
(define FSD (lambda (L1 L2)
   (if (and L1 L2)
       (cond ((= (car L1) (car L2))
              (cons (car L1) (FSD (cdr L2) (cdr L1))))
             ((> (car L1) (car L2))
              (cons (car L2) (FSD (cdr L2) L1)))
             ( else
              (cons (car L1) (FSD L2 (cdr L1)))) )
       (append L1 L2) ) ))
```