13. Abstraction procédurale, exemple

On résout deux problèmes classiques,

— le problème des 8 reines,

— le problème du voyageur de commerce.

On voit qu'en dépit des différences apparentes, ces deux problèmes sont du type "recherche et optimisation", puis on montre que les programmes les résolvant sont des instances d'un schéma très général, ce qui amène à résoudre

— le problème de la recherche abstraite.

On donne en même temps de nouvelles applications de la récursion, de la structuration en procédures et de l'usage de la forme `let` et de ses variantes.

On travaille en données concrètes et on laisse à l'étudiant le problème de la réécriture avec données abstraites.

Introduction au problème des 8 reines I

Un échiquier est un carré de 8 cases de côté.
Deux reines sur un échiquier sont en prise si elles se trouvent sur une même horizontale, verticale ou diagonale.
Si les positions des reines sont (a, b) et $(a + i, b + j)$, les reines sont donc en prise si $i = 0$, $j = 0$ ou $i = \pm j$.
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 posé consiste à construire toutes les possibilités de placer huit reines sur un échiquier sans que deux d'entre elles soient en prise.
Une solution (s'il en existe) comportera une reine par colonne, chacune se trouvant sur une ligne distincte.
On pourra la représenter par une liste de type $(a_1, \ldots, a_8)$, permutation de la liste $(1, \ldots, 8)$. Les positions correspondantes des reines forment l'ensemble $\{(a_i, i) : i = 1, \ldots, 8\}$.
Toute solution est une permutation ; une permutation vérifie naturellement les conditions relatives aux horizontales et aux verticales, et sera une solution si elle vérifie en outre la condition relative aux diagonales.

Introduction au problème des 8 reines II

On pourrait penser à l'approche récursive habituelle, et exprimer les solutions du problème des n reines en fonction de celles du problème des $n-1$ reines. Il apparaît cependant que cette approche n'est pas facile. On observe d'ailleurs que le problème admet une solution pour $n = 1$, aucune solution pour $n = 2$ et $n = 3$, et deux solutions pour $n = 4$.

Une approche alternative naturelle est de construire les $n!$ permutations représentant tous les "candidats-solutions" (ceux-ci respectent les conditions horizontales et verticales, mais pas nécessairement les conditions diagonales) et ensuite de "filtrer" ceux-ci pour ne retenir que les (vraies) solutions. Cette approche est facile à mettre en œuvre mais très inefficace. Pour $n = 8$, il y a en effet $8! = 40\,320$ candidats-solutions mais seulement 92 solutions.

Introduction au problème des 8 reines III

Il suffit d'imaginer une mise en œuvre concrète, à la main, pour découvrir une optimisation élémentaire. L'expérimentateur place les reines une à une, dans des colonnes contigues. S'il place, par exemple, une reine en colonne 1, ligne 4, puis une seconde en colonne 2, ligne 5, la condition diagonale est déjà violée, et l'adjonction de nouvelles reines dans les six autres colonnes n'arrangera rien. Les $6! = 720$ candidats-solutions correspondants peuvent être éliminés d'un coup.

Le but de l'exercice est de montrer comment l'usage d'une stratégie clairement procédurale peut s'intégrer méthodiquement dans le paradigme fonctionnel. D'autre part, la stratégie "generate-and-test" et ses diverses variantes sont intéressantes pour elles-mêmes, vu leurs applications nombreuses et importantes.

Le problème des 8 reines I

On représentera naturellement un placement de reines sur l'échiquier par une liste. On considèrera par exemple que, si $n = 8$, la liste `(7 3 1)` identifie un échiquier comportant des reines en 6ième, 7ième et 8ième colonnes, placées respectivement sur les lignes 7, 3 et 1. Comme un élément s'ajoute commodément en tête de liste, on considèrera que l'échiquier est rempli de la droite (col 8) vers la gauche (col 1).

Le placement `(7 3 1)` est *légal* car la condition diagonale est respectée. Un placement légal est une *configuration*. Une première fonction à réaliser est le prédicat `legal?` tel que `(legal? try conf)` soit vrai si l'adjonction, dans la colonne libre la plus à droite de la configuration `conf`, d'une reine en ligne `try`, fournit encore une configuration.

On ne peut pas ramener le cas `(legal? try conf)` au cas `(legal? try (cdr conf))` parce qu'il y a décalage d'une colonne pour le placement de la nouvelle reine. Pour appliquer cette récursion habituelle, il faut tenir compte de ce décalage, soit d. Pour ce faire, on définit un prédicat auxiliaire `good?` ; on pourrait, avec des notations évidentes, ramener le cas `(good? try conf d)` au cas `(good? try (cdr conf) (add1 d))` mais on préfère introduire deux paramètres, `up` et `down`, correspondant respectivement à `(+ try d)` et `(- try d)`.

Le problème des 8 reines II

```
(define legal?
  (lambda (try conf) (good? try conf (add1 try) (sub1 try)))))

(define good?
  (lambda (try conf up down)
    (or (null? conf)
        (let ((next-pos (car conf)))
          (and (not (= next-pos try))
               (not (= next-pos up))
               (not (= next-pos down))
               (good? try (cdr conf) (add1 up) (sub1 down)))))))
```

La fonction auxiliaire `good?` peut être rendue locale à la fonction principale `legal?`. Comme la définition de `good?` est récursive, il conviendra d'utiliser `letrec`. Puisque le premier argument de `good?` ne varie pas lors des appels récursifs, on peut l'omettre.

A titre d'exercice, le lecteur pourra préciser la spécification de `legal?` donnée précédemment, et spécifier la fonction auxiliaire `good?`.

Le code de `legal?` est donné page suivante ; le prédicat `solution?` détermine si une configuration est une solution.

Le problème des 8 reines III

L'adjonction de la reine `try` à la configuration `conf` produit-elle une configuration ?

```
(define legal?
  (lambda (try conf)
    (letrec
      ((good?
         (lambda (new-pl up down)
           (or (null? new-pl)
               (let ((next-pos (car new-pl)))
                 (and (not (= next-pos try))
                      (not (= next-pos up))
                      (not (= next-pos down))
                      (good? (cdr new-pl) (add1 up) (sub1 down))))))))
      (good? conf (add1 try) (sub1 try)))))

(legal? 2 '(4 8))   #t
(legal? 6 '(4 8))   #f
(legal? 8 '(4 8))   #f

(define solution? (lambda (conf) (= (length conf) fresh-try)))
```

Le problème des 8 reines IV

Le cœur de notre stratégie consiste à prolonger une configuration en une solution. Pour éviter tant les omissions que les répétitions, on doit ordonner totalement l'ensemble des configurations, de manière compatible avec l'ordre dans lequel elles seront générées et testées. L'ordre *total* le plus communément utilisé pour les domaines de listes est l'ordre lexicographique. Nous l'adoptons ici avec deux modifications mineures : il est renversé (l'élément de poids le plus élevé est à la fin) et inversé (un élément plus grand précède un élément plus petit). Notons aussi qu'une configuration partielle précédera toujours ses prolongements.

Pour fixer les idées, considérons le cas $n = 8$. A chaque configuration partielle α, nous associons le nombre dont la liste renversée des chiffres est α', obtenue en complétant α par des occurrences de 9. Par exemple, aux configurations $\alpha =$ `(7 3 1)` et $\beta =$ `(2 6 3 1)` on associe respectivement les nombres $\alpha' = 13799999$ et $\beta' = 13629999$. L'ordre des configurations est l'ordre numérique inverse ; la configuration α précède la configuration β (on écrit $\alpha \prec \beta$) puisque l'on a $\alpha' > \beta'$. La première configuration est donc la configuration vide `()`, représentée par le nombre 99999999.

Le problème des 8 reines V

Prolongement d'une configuration sans "backtracking". La fonction `build-solution` calcule le premier *prolongement* d'une configuration donnée (s'il existe).

```
(define fresh-try 8)

(define build-solution
  (lambda (conf)
    (newline) (blank-write conf)
    (if (solution? conf)
        conf
        (forward fresh-try conf))))

(define forward
  (lambda (try conf)
    (cond ((zero? try) 'none)
          ((legal? try conf) (build-solution (cons try conf)))
          (else (forward (sub1 try) conf)))))
```

Le problème des 8 reines VI

```
(build-solution '(2 6 3 1 4 8))
        (2 6 3 1 4 8)
      (7 2 6 3 1 4 8)
    (5 7 2 6 3 1 4 8)
:-) (5 7 2 6 3 1 4 8)

(build-solution '(3 1 7 2 4 8))
        (3 1 7 2 4 8)
      (5 3 1 7 2 4 8)
:-) none

(build-solution '(6 2 7 1 4 8))
        (6 2 7 1 4 8)
      (3 6 2 7 1 4 8)
:-) none   ;; induit par forward
```

Le problème des 8 reines VII

```
(define forward
  (lambda (try conf)
    (cond ((zero? try) (backtrack conf))
          ((legal? try conf) (build-solution (cons try conf)))
          (else (forward (sub1 try) conf)))))

(build-solution '(6 2 7 1 4 8))
        (6 2 7 1 4 8) ;; configurations
      (3 6 2 7 1 4 8)
      (3 6 2 7 1 4 8) ;; intermediaires
        (6 2 7 1 4 8)
          (2 7 1 4 8) ;; on pourrait
            (7 1 4 8)
            (3 1 4 8) ;; les afficher
          (6 3 1 4 8)
        (2 6 3 1 4 8) ;; en utilisant
      (7 2 6 3 1 4 8)
    (5 7 2 6 3 1 4 8) ;; newline-display
:-) (5 7 2 6 3 1 4 8)
```

Le problème des 8 reines VIII

```
(define backtrack
  (lambda (conf)
    (newline) (blank-write conf) ;; optionnel
    (if (null? conf)
        '()
        (forward (sub1 (car conf)) (cdr conf)))))

(build-solution '(6 1 4 8))
            (6 1 4 8)
          (2 6 1 4 8)
        (7 2 6 1 4 8)
        (5 2 6 1 4 8)
      (7 5 2 6 1 4 8)
      (7 5 2 6 1 4 8)
        (5 2 6 1 4 8)
          (2 6 1 4 8)
            (6 1 4 8)
            (3 1 4 8)
          (6 3 1 4 8)
        (2 6 3 1 4 8)
      (7 2 6 3 1 4 8)
    (5 7 2 6 3 1 4 8)
:-) (5 7 2 6 3 1 4 8)
```

Le problème des 8 reines IX

Construction progressive de toutes les solutions.

```
(build-solution '()) :-) (5 7 2 6 3 1 4 8) ;; solution 01

(backtrack '(5 7 2 6 3 1 4 8)) :-) (4 7 5 2 6 1 3 8) ;; solution 02

(backtrack '(4 7 5 2 6 1 3 8)) :-) (6 4 7 1 3 5 2 8) ;; solution 03

(backtrack '(6 4 7 1 3 5 2 8)) :-) (6 3 5 7 1 4 2 8) ;; solution 04

... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...

(backtrack '(3 5 2 8 6 4 7 1)) :-) (5 2 4 7 3 8 6 1) ;; solution 91

(backtrack '(5 2 4 7 3 8 6 1)) :-) (4 2 7 3 6 8 5 1) ;; solution 92

(backtrack '(4 2 7 3 6 8 5 1)) :-) ()           ;; plus de solution
```

Il existe donc 92 solutions.

Le problème des 8 reines X

Automatiser en une boucle le “backtracking” sur les solutions.

```
(define build-all-solutions
  (lambda ()
    (letrec
      ((loop
        (lambda (sol)
          (if (null? sol)
              '()
              (cons sol (loop (backtrack sol)))))))
      (loop (build-solution '()))))))

(build-all-solutions)
:-) ((5 7 2 6 3 1 4 8) (4 7 5 2 6 1 3 8) ... (4 2 7 3 6 8 5 1))
```

Le problème des 8 reines XI

```
(define step  ;; sans récursion mutuelle
  (lambda (try conf)
    (cond ((zero? try)
           (if (null? conf) '() (step (sub1 (car conf)) (cdr conf))))
          ((legal? try conf)
           (let ((lp (cons try conf)))
             (if (solution? lp) lp (step fresh-try lp))))
          (else (step (sub1 try) conf)))))

(define build-all-solutions
  (lambda ()
    (letrec
      ((loop
        (lambda (sol)
          (if (null? sol)
              '()
              (cons sol (loop (step (sub1 (car sol)) (cdr sol))))))))
      (loop (step fresh-try '()))))))
```

Voyageur de commerce I

Données : un ensemble de villes numérotées de `1` à `n`,
une carte `dmap` des distances entre villes,
une longueur maximale de circuit `lmax`

```
0   200 316 200 316 282 447 412 423 632 141 223
    0   141 282 316 200 400 223 316 600 141 100
        0   423 447 316 510 223 400 707 282 100
            0   141 200 282 412 316 447 141 360
                0   141 141 360 200 316 200 412
                    0   200 223 141 400 141 300
                        0   360 141 200 316 500
                            0   223 538 300 282
                                0   316 282 412
                                    0   510 700
                                        0   223
                                            0
```

Voyageur de commerce II

Données : un ensemble de villes numérotées de `1` à `n`,
une carte `dmap` des distances entre villes,
une longueur maximale de circuit `lmax`

```
(define dmap
  (lambda (i j)
    (cond ((= i j) 0)
          ((> i j) (dmap j i))
          ((= i 1) (cond ((= j 2) 200)
                         ...       ...
                         ((= j 12) 316)))
          ((= i 2) ...)
          ...       ...
          ((= i 11) (cond ((= j 12) 223))))))
```

Voyageur de commerce III

Essai d'adjonction de la ville `try` au circuit partiel légal `legal-c`.
La solution `(3 2 4 1)` représente le circuit $1-3-2-4-1$.

```
(define legal?
  (lambda (try legal-c dmap lmax)
    (letrec ((circ? (lambda (t l-c)
                      (or (null? l-c)
                          (and (not (= t (car l-c)))
                               (circ? t (cdr l-c))))))
             (size-c (lambda (l-c)
                       (cond ((= (car l-c) 1) 0)
                             (else (+ (size-c (cdr l-c))
                                      (dmap (car l-c) 1)
                                      (dmap (car l-c) (cadr l-c))
                                      (- (dmap (cadr l-c) 1)))))))))
      (and (circ? try legal-c) (<= (size-c (cons try legal-c)) lmax)))))
```

Voyageur de commerce IV

```
(define fresh-try 12)

(define solution?
  (lambda (legal-c) (= (length legal-c) fresh-try)))

(define build-solution
  (lambda (legal-c dmap lmax)
    (if (solution? legal-c)
        legal-c
        (forward fresh-try legal-c dmap lmax))))

(define forward
  (lambda (try legal-c dmap lmax)
    (cond ((zero? try) (backtrack legal-c dmap lmax))
          ((legal? try legal-c dmap lmax)
           (build-solution (cons try legal-c) dmap lmax))
          (else (forward (sub1 try) legal-c dmap lmax)))))

(define backtrack
  (lambda (legal-c dmap lmax)
    (if (null? legal-c)
        '()
        (forward (sub1 (car legal-c)) (cdr legal-c) dmap lmax))))
```

Voyageur de commerce V

```
(define fresh-try 5)   fresh-try
  200 316 200 316
      141 282 316
          423 447
              141

(define length-c
  (lambda (l-c dmap)
    (if (= (car l-c) 1)
        0
        (+ (length-c (cdr l-c) dmap)
           (dmap (car l-c) 1)
           (dmap (car l-c) (cadr l-c))
           (- (dmap (cadr l-c) 1))))))
```

Voyageur de commerce VI

```
(build-solution '(1) dmap 1000)  ()
(build-solution '(1) dmap 2000)  (2 3 4 5 1)
(length-c '(2 3 4 5 1) dmap)     1221
(backtrack '(2 3 4 5 1) dmap 1221) > (3 2 4 5 1)
(length-c '(3 2 4 5 1) dmap)     1196
(backtrack '(3 2 4 5 1) dmap 1196) > (2 3 5 4 1)
(length-c '(2 3 5 4 1) dmap)     1129
(backtrack '(2 3 5 4 1) dmap 1129) > (3 2 5 4 1)
(length-c '(3 2 5 4 1) dmap)     1114
(backtrack '(3 2 5 4 1) dmap 1114) > (4 5 2 3 1)
(length-c '(4 5 2 3 1) dmap)     1114
(backtrack '(4 5 2 3 1) dmap 1114) > ()
```

Voyageur de commerce VII

```
(define fresh-try 8)   fresh-try

  200 316 200 316 282 447 412
      141 282 316 200 400 223
          423 447 316 510 223
              141 200 282 412
                  141 141 360
                      200 223
                          360

(build-solution '(1) dmap 1500)              (2 3 8 6 7 5 4 1)

(length-c '(2 3 8 6 7 5 4 1) dmap)           1469

(backtrack '(2 3 8 6 7 5 4 1) dmap 1469)     (4 5 7 6 8 3 2 1)

(length-c '(4 5 7 6 8 3 2 1) dmap)           1469

(backtrack '(4 5 7 6 8 3 2 1) dmap 1469)     ()
```

Recherche abstraite I

Idée.

On observe que le problème des reines et celui du voyageur de commerce, quoique très différents en apparence, se résolvent par la même technique.

La constante `fresh-try`, ainsi que les procédures `legal?` et `solution?` diffèrent d'un problème de recherche à l'autre ; ces objets seront donc des arguments de la procédure de recherche abstraite `searcher`.

Par contre, le principe de construction des solutions ne varie pas, donc les procédures `build-solution`, `forward`, `backtrack` et `build-all-solutions` seront écrites une fois pour toutes. Elles pourront être internes à la procédure de recherche abstraite `searcher`.

L'unification de plusieurs problèmes concrets en un problème plus abstrait peut impliquer quelques changements de détail.

Recherche abstraite II

```
(define searcher
  (lambda (legal? solution? fresh-try)
    (letrec
      ((build-solution
         (lambda (conf)
           (if (solution? conf) conf (forward fresh-try conf))))
       (forward
         (lambda (try conf)
           (cond ((zero? try) (backtrack conf))
                 ((legal? try conf) (build-solution (cons try conf)))
                 (else (forward (sub1 try) conf)))))
       (backtrack
         (lambda (conf)
           (if (null? conf) '() (forward (sub1 (car conf)) (cdr conf)))))
       (build-all-solutions
         (lambda ()
           (letrec
             ((loop
                (lambda (sol)
                  (if (null? sol) '() (cons sol (loop (backtrack sol)))))))
             (loop (build-solution '())))))))
      (build-all-solutions))))
```

Recherche abstraite III

```
(define legal?-reines
  (lambda (try conf)
    (letrec
      ((good?
         (lambda (new-pl up down)
           (cond ((null? new-pl) #t)
                 (else (let ((next-pos (car new-pl)))
                         (and (not (= next-pos try))
                              (not (= next-pos up))
                              (not (= next-pos down))
                              (good? (cdr new-pl)
                                     (add1 up)
                                     (sub1 down)))))))))
      (good? conf (add1 try) (sub1 try)))))

(define solution?-reines
  (lambda (s) (= (length s) fresh-try-reines)))

(define fresh-try-reines 8)
```

Problème des 8 reines : essais

```
(searcher legal?-reines solution?-reines fresh-try-reines)
:-) ((5 7 2 6 3 1 4 8) ... (4 2 7 3 6 8 5 1))

(length (searcher legal?-reines solution?-reines fresh-try-reines))
:-) 92

(define fresh-try-reines 6)
:-) ...

(searcher legal?-reines solution?-reines fresh-try-reines)
:-) ((2 4 6 1 3 5) (3 6 2 5 1 4) (4 1 5 2 6 3) (5 3 1 6 4 2))

(define fresh-try-reines 10)
:-) ...

(length (searcher legal?-reines solution?-reines fresh-try-reines))
:-) 724
```

Paramètres pour le problème du voyageur

```
(define dmap
  (lambda (i j)
    (cond ((= i j) 0)
          ((> i j) (dmap j i))
          ((= i 0) (cond ((= j 1) 200)
                         ((= j 2) 316))
                         ...  ...  ...
                         ((= j 10) 141))
                         ((= j 11) 223)))
          ((= i 1) (cond ((= j 2) 141))
                         ...  ...  ...
          ((= i 2) (cond ((= j 3) 423))
          ... ... ... ... ... ... ...
          ((= i 9) (cond ((= j 10) 510))
                         ((= j 11) 700)))
          ((= i 10)(cond ((= j 11) 223))))))

(define lmax ...)

(define fresh-try-voyageur ...)
```

Procédures pour le problème du voyageur

```
(define legal?-voyageur
  (lambda (try leg-c)
    (letrec
      ((circ?
         (lambda (t l-c)
           (cond ((null? l-c) #t)
                 (else (and (not (= t (car l-c))) (circ? t (cdr l-c)))))))
       (size-c
         (lambda (l-c)
           (cond ((null? l-c) 0)
                 ((null? (cdr l-c)) (* 2 (dmap (car l-c) 0)))
                 (else (+ (size-c (cdr l-c))
                          (dmap (car l-c) 0)
                          (dmap (car l-c) (cadr l-c))
                          (- (dmap (cadr l-c) 0)))))))))
      (and (circ? try leg-c) (<= (size-c (cons try leg-c)) lmax)))))

(define solution?-voyageur (lambda (s) (= (length s) fresh-try-voyageur)))
```

Problème du voyageur : essais

```
; Villes de 0 a 4, longueur maximale 1200
(define lmax 1200) ...
(define fresh-try-voyageur 4) ...

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((2 1 3 4) (1 2 4 3) (2 1 4 3) (3 4 1 2) (4 3 1 2) (3 4 2 1))
; Rappel: (2 1 3 4) est 0-2-1-3-4-0



; longueur maximale 1150
(define lmax 1150) ...

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 4 3) (2 1 4 3) (3 4 1 2) (3 4 2 1))
```

Problème du voyageur : essais (bis)

```
; Villes de 0 a 8 (longueur maximale 1150)
(define fresh-try-voyageur 8) ...

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ()
; pas de solution


; (Villes de 0 a 8) longueur maximale 1600
(define lmax 1600) lmax

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 7 5 8 6 4 3) (3 4 6 8 5 7 2 1))
; circuit 0-1-2-7-5-8-6-4-3-0 ou inverse
```

Problème du voyageur : essais (ter)

```
; Villes de 0 a 9, longueur maximale 2000
(define fresh-try-voyageur 9) (define lmax 2000)


(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 7 8 6 9 4 5 3) (1 2 7 8 9 6 4 5 3)
     (1 2 7 5 8 6 9 4 3) (1 2 7 8 5 6 9 4 3)
     (1 2 7 5 8 9 6 4 3) (1 2 7 8 9 6 5 4 3)
     (3 4 5 6 9 8 7 2 1) (3 5 4 6 9 8 7 2 1)
     (3 5 4 9 6 8 7 2 1) (3 4 9 6 5 8 7 2 1)
     (3 4 6 9 8 5 7 2 1) (3 4 9 6 8 5 7 2 1))


(define lmax 1925) ; longueur maximale 1925


(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ()


(define lmax 1926) ; longueur maximale 1926


(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 7 5 8 6 9 4 3) (1 2 7 5 8 9 6 4 3)
     (3 4 6 9 8 5 7 2 1) (3 4 9 6 8 5 7 2 1))
```

Variables globales

La valeur de `(searcher ...)` dépend des variables globales `dmap` et `lmax` ; il vaut mieux éliminer ce manque de transparence.

"Solution" naïve et **incorrecte**

```
(let ((lmax 3000))
     (searcher legal?-voyageur solution?-voyageur fresh-try-voyageur))
:-) ((1 2 7 5 8 6 9 4 3) (1 2 7 5 8 9 6 4 3)
     (3 4 6 9 8 5 7 2 1) (3 4 9 6 8 5 7 2 1))
```

Le `let` n'a eu aucun effet ;
la réponse correspond à la valeur globale de `lmax`.

Pour l'évaluateur, l'environnement pertinent pour la variable `lmax` est l'environnement global, où la variable `legal?-voyageur` a été LIEE, et non l'environnement d'APPLICATION, où la forme `(let ...)` a été évaluée.

Rappelons l'approche *lexicale* de Scheme en ce qui concerne la portée des identificateurs. L'identificateur `lmax` n'ayant aucune occurrence dans le *texte* du corps du `let`, il est normal que la liaison due au `let` soit restée sans effet.

Suppression des variables globales

Eviter les variables globales `dmap` et `lmax`

Solution correcte

On n'écrit pas directement la procédure `legal?-voyageur` mais le générateur de procédure `make-legal?-voyageur`, tel que la forme `(make-legal?-voyageur map lmax)` ait la valeur (prédicative) du `legal?-voyageur` antérieur.

On pourra alors définir

```
(define solve-voyageur
  (lambda (dmap lmax)
    (searcher (make-legal?-voyageur dmap lmax)
              solution?-voyageur
              fresh-try-voyageur)))
```

Il est préférable d'inclure les définitions de `make-legal?-voyageur`, `solution?-voyageur` et `fresh-try-voyageur` dans `solve-voyageur`.

```
(define make-legal?-voyageur
  (lambda (dmap lmax)
    (lambda (try leg-c)
      (letrec
        ((circ?
           (lambda (try leg-c)
             (cond ((null? leg-c) #t)
                   (else (and (not (= try (car leg-c)))
                              (circ? try (cdr leg-c)))))))
         (size-c
           (lambda (l-c)
             (cond ((null? l-c) 0)
                   ((null? (cdr l-c)) (* 2 (dmap (car l-c) 0)))
                   (else (+ (size-c (cdr l-c))
                            (dmap (car l-c) 0)
                            (dmap (car l-c) (cadr l-c))
                            (- (dmap (cadr l-c) 0)))))))))
        (and (circ? try leg-c)
             (<= (size-c (cons try leg-c)) lmax))))))
```