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))))))
```


14. Les vecteurs

Comment regrouper des objets en une structure ?

```
(list 4 'x 3 '(7 . a))              (4 x 3 (7 . a))
(vector 4 'x 3 '(7 . a))           #(4 x 3 (7 . a))
```

L'accès aux éléments d'une liste est séquentiel ;
l'accès aux éléments d'un vecteur est direct

```
(vector-length '#(4 x 3 (7 . a)))   4
(vector-ref '#(4 x 3 (7 . a)) 0)    4
(vector-ref '#(4 x 3 (7 . a)) 3)    (7 . a)
(vector-ref '#(4 x 3 (7 . a)) 4)    Object 4 out of range
(subvector '#(4 x 3 (7 . a)) 1 2)   #(x)
(vector->list '#(4 x 3 (7 . a)))    (4 x 3 (7 . a))
(list->vector '(4 x 3 (7 . a)))     #(4 x 3 (7 . a))
```


Instructions altérantes

On utilise les vecteurs plutôt que les listes si l'accès aléatoire est important.

En scheme pur, on n'altère pas les objets, on en crée une copie modifiée.

C'est plus lent et cela prend de la place ... scheme comporte aussi des instructions altérantes.

```
(define v '#(0 2 4 6 8 10 12 14 16 18))    v
(define w v)                               w
(vector-set! v 3 -99)                      No value
v                                          #(0 2 4 -99 8 10 12 14 16 18)
w                                          #(0 2 4 -99 8 10 12 14 16 18)
```

C'est dangereux (effets de bord) ... mais c'est utile !

Remarque. On peut aussi altérer des variables simples, des listes, etc. au moyen de `set!`, `set-car!`, `set-cdr!`.

Tri par insertion (rappel)

```
(define insertsort
  (lambda (ls)
    (if (null? ls)
        ls
        (insert (car ls) (insertsort (cdr ls))))))

(define insert
  (lambda (a ls)
    (cond ((null? ls) (cons a '()))
          ((< a (car ls)) (cons a ls))
          (else (cons (car ls) (insert a (cdr ls)))))))
```


Version altérante pour vecteurs

```
(define vector-insertsort!
  (lambda (v)
    (let ((size (vector-length v)))
      (letrec ((loop
                (lambda (k)
                  (if (< k size)
                      (begin (vector-insert! k v) (loop (+ k 1)))))))
        (loop 1)))))

(define vector-insert!
  (lambda (k vec)
    (let ((val (vector-ref vec k)))
      (letrec ((insert-h
                (lambda (m)
                  (if (zero? m)
                      (vector-set! vec 0 val)
                      (let ((c (vector-ref vec (- m 1))))
                        (if (< val c)
                            (begin (vector-set! vec m c) (insert-h (- m 1)))
                            (vector-set! vec m val)))))))
        (insert-h k)))))
```


Documentation et essais I

Spécification. Si `v` est (lié à) un vecteur dans l'environnement courant avant l'exécution de `(vector-insertsort! v)`, alors `v` est (lié à) la version triée de ce vecteur après l'exécution.

Fonctionnement. Si s est la taille de `v`, exécuter `(vector-insertsort! v)` revient à exécuter la séquence

```
(vector-insert! 1 v)
(vector-insert! 2 v)
⋮      ⋮      ⋮      ⋮
(vector-insert! n v)
```

où $n = s-1$ est l'index du dernier élément du vecteur `v`.
Rappel : un vecteur de taille s est indexé de 0 à $s-1$.

Spécification. Si le préfixe `v`$[0:k-1]$ est trié avant l'exécution de la forme `(vector-insert! k v)`, alors cette exécution a pour effet d'insérer `v`$[k]$ à sa place. Le suffixe `v`$[k+1:n]$ n'est pas altéré.

Documentation et essais II

Exemple. Pour un vecteur de longueur 6, on appelle 5 fois la procédure d'insertion.

```
(define v '#(9 3 7 0 5 1))      #(9 3 7 0 5 1)
                                   ^ ^

(vector-insert! 1 v)            #(3 9 7 0 5 1)
                                   ^ ^

(vector-insert! 2 v)            #(3 7 9 0 5 1)
                                     ^ ^

(vector-insert! 3 v)            #(0 3 7 9 5 1)
                                       ^ ^

(vector-insert! 4 v)            #(0 3 5 7 9 1)
                                         ^ ^

(vector-insert! 5 v)            #(0 1 3 5 7 9)
```

Documentation et essais III

Fonctionnement. L'appel de `(vector-insert! k v)` crée la liaison `val: v[k]` puis provoque le nombre adéquat de "décalages", suivi de l'écrasement de la dernière case recopiée par `val`.

```
...   ...
  ;; (insert-h m) ::=
  (let ((c (vector-ref vec (- m 1))))
    (if (< val c)
        (begin (vector-set! vec m c) (insert-h (- m 1)))
        (vector-set! vec m val)))
...   ...
```

```
k: 4   v: #(0 3 7 9 5 1))    val: 5
init   v: #(0 3 7 9 5 1))    val: 5    c: 9
then   v: #(0 3 7 9 9 1))    val: 5    c: 7
then   v: #(0 3 7 7 9 1))    val: 5    c: 3
else   v: #(0 3 5 7 9 1))
```


"Reverse"

"Reverse", version fonctionnelle simple

```
(define reverse
  (lambda (l)
    (if (null? l)
        '()
        (append (reverse (cdr l)) (list (car l))))))
```

"Reverse", version fonctionnelle accumulante

```
(define rev-it
  (lambda (l)
    (letrec ((r0 (lambda (u v)
                   (if (null? u)
                       v
                       (r0 (cdr u) (cons (car u) v))))))
      (r0 l '()))))
```


Version vectorielle altérante

```
(define vector-reverse!
  (lambda (v)
    (let ((s (vector-length v)) (t 0))
       (letrec
         ((switch (lambda (i j)
                    (begin (set! t (vector-ref v i))
                           (vector-set! v i
                           (vector-ref v j))
                           (vector-set! v j t))))
          (loop (lambda (i j)
                  (if (< i j)
                      (begin (switch i j)
                             (loop (+ i 1) (- j 1)))))))
         (loop 0 (- s 1))))))
```


Générateur aléatoire

```
(make-vector 4)   #(() () () ())

(random 1000)   624    ;; [0...999]

(define random-vector
  (lambda (n)
    (let ((v (make-vector n)))
      (letrec ((fill
                (lambda (i)
                  (if (< i n)
                      (begin (vector-set! v i (random 1000))
                             (fill (+ i 1)))))))
        (fill 0))
      v)))

(random-vector 5)    #(90 933 656 240 587)
(random-vector 5)    #(666 943 203 632 512)
```

15. Mécanismes particuliers, tabulation

Mesure du temps d'exécution I

Définition d'un "chronomètre".

```
(define time
  (lambda (proc arg) ;; arg : argument unique
    (let* ((t0 (runtime))
           (val (proc arg))
           (t1 (runtime))
           (del (round (* (- t1 t0) 100))))
      (newline) (display "Time = ") (write del)
      val)))
```

Fonctions de test.

```
(define fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))

(define fib2
  (lambda (n)
    (letrec ((f (lambda (n a b) (if (= n 0) a (f (- n 1) b (+ a b))))))
      (f n 0 1))))
```


Mesure du temps d'exécution II

Essais.

```
(time fib 20)   Time = 455.   ;Value:   6765
(time fib 21)   Time = 735.   ;Value:  10946

(time fib2 20)  Time =   0.   ;Value:   6765
(time fib2 30)  Time =   0.   ;Value: 832040
```

	300	1000	3000	10000
reverse	20	170	1435	15428
rev-it	5	15	43	140
vector-reverse!	8	27	72	237


```
(eval '(+ 5 1 3) '())   9
(apply + '(5 1 3))      9
```

Fonctions d'arité variable

```
(define list (lambda v v))
(list 1 2 3 4)              (1 2 3 4)

(define sort (lambda args (apply insertsort (list args))))

(sort)                      ()
(sort 3 1 5 2)              (1 2 3 5)

(define g-expt
  (lambda (x . l) (if (null? l) x (expt x (apply g-expt l)))))

(g-expt)                    Error: at least 1 argument required
(g-expt 5)                  5
(g-expt 2 3)                8
(g-expt 2 3 2)              512
```


Mesure du temps d'exécution III

```
(define time*
  (lambda (proc . args) ;; args: liste d'arguments
    (let* ((t0 (runtime))
           (val (apply proc args))
           (t1 (runtime))
           (del (round (* (- t1 t0) 100))))
      (newline) (display "Time = ") (write del)
      val)))


(time* fib 20)
 Time = 457.   ;Value: 6765

(time* + (fib 18) (fib 19))
 Time =   0.   ;Value: 6765
```

Seul le temps consommé par l'addition a été comptabilisé !

```
(time* g-expt 2 2 2 2)     Time =   0.    65536
(time* g-expt 2 2 2 2 2)   Time = 868.    **too big!**
```

Simulation de `letrec`

```
(letrec
  ((even? (lambda (n) (or  (= n 0) (odd?  (- n 1)))))
   (odd?  (lambda (n) (and (> n 0) (even? (- n 1))))))
  (list (even? 11) (odd? 7)))
;Value: (#f #t)

(let ((even? 'any) (odd?  'any))
  (let ((e (lambda (n) (or  (= n 0) (odd?  (- n 1)))))
        (o (lambda (n) (and (> n 0) (even? (- n 1))))))
    (set! even? e)
    (set! odd?  o)
    (list (even? 11) (odd? 7))))
;Value: (#f #t)
```


Une curiosité . . .

On peut se passer de `letrec`. Le code

```
(letrec
  ((fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))))
  (fact x))
```

peut être simulé par le code

```
((lambda (n)
   ((lambda (fact) (fact fact n))
    (lambda (f k) (if (= k 0) 1 (* k (f f (- k 1)))))))
 x)
```


Tables et tabulation I

Idée : ne pas recalculer un élément déjà calculé.

Exemple : rendre efficace le programme naïf `fib`.

Stratégie : maintenir une table de valeurs calculées.

Une table est une liste de paires pointées (variable,valeur)

```
(define *TABLE-fib* '((1 . 1) (0 . 0)))

(define m-fib
  (lambda (n)
    (let ((v (assv n *TABLE-fib*)))
      (if v
          (cdr v)
          (let ((a (m-fib (- n 1))) (b (m-fib (- n 2))))
            (let ((val (+ a b)))
              (set! *TABLE-fib* (cons (cons n val) *TABLE-fib*))
              val))))))
```


Tables et tabulation II

Essais.

```
(time m-fib 100)   Time = 15.         354224848179261915075

(time fib-it 100)  Time = 2.          354224848179261915075

(time m-fib 100)   Time = 0.          354224848179261915075

(time m-fib 90)    Time = 0.          2880067194370816120
```

Problèmes.

Version tabulée à écrire pour chaque fonction.

Introduction d'une variable globale.

Tables et tabulation III

Fonctions de tabulation.

```
(define lookup
  (lambda (obj table succ-p fail-p)
    (if (null? table)
        (fail-p)
        (let ((pr (car table)))
          (if (equal? (car pr) obj)
              (succ-p pr)
              (lookup obj (cdr table) succ-p fail-p))))))

(define assoc
  (lambda (obj table)
    (lookup obj table (lambda (pr) pr) (lambda () #f))))
```


Tables et tabulation IV

Un tabulateur générique.

```
(define memoize
  (lambda (proc)
    (let ((table '()))
      (lambda (arg)
        (lookup
          arg
          table
          cdr
          (lambda ()
            (let ((val (proc arg)))
              (set! table (cons (cons arg val) table))
              val)))))))
```


Tables et tabulation V

Fonctions de test.

```
(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2))))))

(define m-fib1 (memoize fib))

(define m-fib2
  (memoize (lambda (n)
             (if (< n 2)
                 n
                 (+ (m-fib2 (- n 1)) (m-fib2 (- n 2)))))))

(define fib-it
  (lambda (n)
    (letrec
      ((f0 (lambda (n p q)
             (if (= n 0) p (f0 (- n 1) q (+ p q))))))
      (f0 n 0 1))))
```


Tables et tabulation VI

Essais.

```
(time fib 20)     Time: 161. ;Value:  6765
(time fib 20)     Time: 161. ;Value:  6765
(time fib 21)     Time: 268. ;Value: 10946

(time m-fib1 20)  Time: 165. ;Value:  6765
(time m-fib1 21)  Time: 268. ;Value: 10946
(time m-fib1 20)  Time:   0. ;Value:  6765
(time m-fib1 19)  Time: 100. ;Value:  4181
(time m-fib1 21)  Time:   0. ;Value: 10946

(time m-fib2 20)  Time:   2.
(time m-fib2 20)  Time:   0.
(time m-fib2 22)  Time:   0. ;Value: 17711
(time m-fib2 100) Time:  29. ;Value: ...
(time m-fib2 100) Time:   0. ;Value: ...

(time fib-it 100) Time:   1. ;Value: ...
(time fib-it 100) Time:   1. ;Value: ...
```

```
(define num-part
  (lambda (n k)
    (if (or (= k n) (= k 1))
        1
        (+ (num-part (- n 1) (- k 1))
           (* k (num-part (- n 1) k))))))

(time* num-part 10  5)   Time = 0.01
(time* num-part 20 10)   Time = 4.87
```

indice	0	1	2	3	4	5	6	...
(n,k)	(1,1)	(2,1)	(2,2)	(3,1)	(3,2)	(3,3)	(4,1)	...

La fonction

$$(n,k) \;\mapsto\; \frac{n(n-1)}{2} + k - 1$$

calcule l'indice associé à des arguments donnés.

Si on refuse l'usage des variables globales, on peut écrire

```
(define num-part-loc
  (lambda (n k)
    (let ((table       ;; table locale
           (make-vector (index n n) #f)))
      (letrec
          ((aux
            (lambda (m q)
              (let ((w (vector-ref table (index m q))))
                (if w
                    w
                    (let
                      ((x
                        (if (or (= q 1) (= q m))
                            1
                            (+ (aux (- m 1) (- q 1))
                               (* q (aux (- m 1) q))))))
                      (vector-set! table (index m q) x)
                      x))))))
        (aux n k)))))
```



```
(define index
  (lambda (n k)
    (+ (/ (* n (- n 1)) 2) k -1)))

(define *MAX* 300)    ;; valeur maximale de n

(define *MEMO* (make-vector (index *MAX* *MAX*) #f))

(define num-part-glob
  (lambda (n k)
    (let ((w (vector-ref *MEMO* (index n k))))
      (if w
          w
          (let
            ((x
              (if (or (= k n) (= k 1))
                  1
                  (+ (num-part-glob (- n 1) (- k 1))
                     (* k
                        (num-part-glob (- n 1) k))))))
            (vector-set! *MEMO* (index n k) x)
            x)))))
```


Il est intéressant d'observer la manière dont la table se remplit en cours d'exécution ; cela peut se faire en intercalant dans le code un ordre d'impression après chaque modification de la table :

```
(num-part-loc 5 3)  ==>
   #(#f #f #f #f #f  1 #f #f #f #f #f #f #f #f #f)
   #(#f #f  1 #f #f  1 #f #f #f #f #f #f #f #f #f)
   #(#f  1  1 #f #f  1 #f #f #f #f #f #f #f #f #f)
   #(#f  1  1 #f  3  1 #f #f #f #f #f #f #f #f #f)
   #(#f  1  1 #f  3  1 #f #f  6 #f #f #f #f #f #f)
   #(#f  1  1  1  3  1 #f #f  6 #f #f #f #f #f #f)
   #(#f  1  1  1  3  1 #f  7  6 #f #f #f #f #f #f)
   #(#f  1  1  1  3  1 #f  7  6 #f #f #f 25 #f #f)
25
```

L'inconvénient d'une table locale est que chaque appel à `num-part-loc` provoque la création d'une nouvelle table. On peut éviter cet inconvénient en utilisant une variable libre :

```
(define num-part-free
  (let ((table (make-vector (index *MAX* *MAX*) #f)))
    (letrec
        ((aux
          (lambda (m q)
            (let ((w
                   (vector-ref table (index m q))))
              (if w
                  w
                  (let
                    ((x
                      (if (or (= q 1) (= q m))
                          1
                          (+ (aux (- m 1) (- q 1))
                             (* q (aux (- m 1) q))))))
                    (vector-set! table (index m q) x)
                    x))))))
      aux)))
```


La table est créée lors de l'évaluation de la forme `define` et réutilisée à chaque appel, comme le montre la session suivante :

```
(timer* num-part-loc 200 100)   Time = 1.73
(timer* num-part-loc 200 100)   Time = 1.73
(timer* num-part-free 200 100)  Time = 1.73
(timer* num-part-free 200 100)  Time = 0.00
```

Avec la table locale, deux calculs successifs de la même valeur (non affichée ici car il s'agit d'un nombre de 235 chiffres) prennent le même temps ; avec la table non locale, le second "calcul" est instantané, car la valeur est trouvée dans la table.


```
(define *cons* 0)

(define kons
  (lambda (x y) (set! *cons* (1+ *cons*)) (cons x y)))

(define lpref1
  (lambda (l)
    (if (null? l)
        (kons l '())
        (kons '() (put (car l) (lpref1 (cdr l)))))))

(define put
  (lambda (x ll)
    (if (null? ll)
        '()
        (kons (kons x (car ll)) (put x (cdr ll))))))
```



```
(define lpref2
  (lambda (l)
    (if (null? l)
        (kons l '())
        (kons l (lpref2 (butlast l))))))

(define butlast
  (lambda (l)
    (if (null? (cdr l))
        '()
        (kons (car l) (butlast (cdr l))))))
```

```
(define lpref3
  (lambda (l) (lreverse (lsuff (reverse l)))))

(define reverse (lambda (l) (rev l '())))

(define lsuff
  (lambda (l)
    (if (null? l) (kons l '()) (kons l (lsuff (cdr l))))))

(define lreverse
  (lambda (ll)
    (if (null? ll)
        '()
        (kons (rev (car ll) '()) (lreverse (cdr ll))))))

(define rev
  (lambda (l a)
    (if (null? l)
        a
        (rev (cdr l) (kons (car l) a)))))
```



```
(define count-1 (lambda (n) (set! *cons* 0) (lpref1 (enum 1 n)) *cons*))
(define count-2 (lambda (n) (set! *cons* 0) (lpref2 (enum 1 n)) *cons*))
(define count-3 (lambda (n) (set! *cons* 0) (lpref3 (enum 1 n)) *cons*))

(define enum (lambda (p q) (if (> p q) '() (cons p (enum (+ p 1) q)))))

(define *list-length* '(1 2 5 10 50 100))
```

```
*list-length*                          (1  2  5  10   50   100)
(map count-1 *list-length*)            (4  9 36 121 2601 10201)
(map count-2 *list-length*)            (2  4 16  56 1276  5051)
(map count-3 *list-length*)            (6 11 32  87 1427  5352)
```