INFO0054-1
Programmation fonctionnelle

LIÈGE université

Chapitre 14 : Les vecteurs

Christophe Debruyne

(c.debruyne@uliege.be)

Cette présentation est basée sur les transparents de Prof. Dr. Em. Pascal Gribomont.

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))
```

"A *vector* is a fixed-length array with constant-time access and update of the vector slots, which are numbered from 0 to one less than the number of slots in the vector." (source: Racket Documentation)

Les vecteurs

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)    vector-ref: index is out of range
> (vector->list '#(4 x 3 (7 . a)))    (4 x 3 (7 . a))
> (list->vector '(4 x 3 (7 . a)))     #(4 x 3 (7 . a))
```

```
(require racket/vector)
> (vector-take '#(4 x 3 (7 . a)) 1)   #(4)
```

Certaines fonctions/procédures sont immédiatement accessible. Nous devons importer les autres.

Instructions altérantes

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

En Scheme (et Racket) *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 (*destructif*).

Attention : les structures de données modifiables (mutable) et non-modifiables (immutable) sont désormais séparées!

Instructions altérantes

Les vecteurs lu avec #(…), #[…], ou #{…} sont non-modifiables !
L'exemple dans les slides et livre de Prof. Gribomon est "obsolète".

```
> (define v '#(0 2 4 6 8 10 12 14 16 18))
> (define v (vector 0 2 4 6 8 10 12 14 16 18)) ...
> (define w v)                                  ...
> (vector-set! v 3 -99)                         ...
> 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 modifiables, etc. au moyen de set!, set-mcar!, set-mcdr!.

Tri par insertion (rappel)


```
> (insert 3 '(1 2 4 5))
(1 2 3 4 5)
```

```
(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)))))))
```

Tri par insertion (rappel)


```
> (insertsort '(5 4 2 1))
(1 2 4 5)
```

```
(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)
                  (cond ((< k size)
                         (begin
                           (vector-insert! k v)
                           (loop (+ k 1)))))))))
        (loop 1)))))
```

Version altérante pour vecteurs

LIÈGE université

```
(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


```
> (define v (vector 9 3 7 0 5 1)) ...
>  v                                '#(9 3 7 0 5 1)
> (vector-insert! 1 v)              ...
>  v                                '#(3 9 7 0 5 1)
> (vector-insert! 2 v)              ...
>  v                                '#(3 7 9 0 5 1)
> (vector-insert! 3 v)              ...
>  v                                '#(0 3 7 9 5 1)
> (vector-insert! 4 v)              ...
>  v                                '#(0 3 5 7 9 1)
> (vector-insert! 5 v)              ...
>  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 '()))))
```

Reverse, 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)
                     (cond ((< i j)
                            (begin
                              (switch i j)
                              (loop (+ i 1) (- j 1))))))))
      (loop 0 (- s 1))))))
```

Générateur aléatoire


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

```
> (random-vector 5)
'#(263 793 298 452 933)

> (random-vector 5)
'#(224 725 866 329 107)
```