

Attribuer des valeurs à des variables avec set!

- ▶ Cette fonction permet d'attribuer une valeur à une variable

```
(set! nombre (+ 3 4))  
(set! nombre (+ 1 nombre))
```

En SCHEME, les fonctions dont le nom se termine par ! sont des fonctions qui modifient la valeur de l'un des arguments (opérations destructives).

Exemple

```
(define interrupteur  
  (let ((etat #f)) (lambda ()  
    (set! etat (not etat))  
    (if etat 'on 'off))))
```

Encapsulation de la variable lit dans la définition de light-switch

Compter le nombre d'appels

```
(define nombreDappels 0)
```

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

Exemple impératif

```
(define secondesImp
  (lambda (h m s)
    (let ((sh 0) (sm 0) (total 0))
      (set! sh (* 60 (* 60 h)))
      (set! sm (* 60 m))
      (set! total (+ s (+ sh sm)))
      total)))
```

Exemple plus fonctionnel

```
(define secondes
  (lambda (h m s)
    (let ((sh (* 60 (* 60 h))))
      (sm (* 60 m)))
    (+ s (+ sh sm))))))
```

Jouer contre Racket...

```
(define inf 0)
(define sup 1000)
(define (devine)
  (truncate (/ (+ inf sup) 2)))
(define (plus-petit)
  (set! sup (devine))
  (devine))
(define (plus-grand)
  (set! inf (devine))
  (devine))
(define (devine-un-nombre inferieur superieur)
  (set! inf inferieur)
  (set! sup superieur)
  (devine))
```

```
> (devine-un-nombre 10 20)
15
> (plus-petit)
12
> (plus-grand)
13
> (plus-grand)
14
```

Pile en Scheme version impérative

```
(define PILE `())  
(define (vide?)  
  (null? PILE))
```

```
(define empiler  
  (lambda (element)  
    (set! PILE (cons  
              element PILE))))
```

```
(define depiler  
  (lambda ()  
    (let (( res (car PILE)))  
      (set! PILE (cdr PILE))  
      res)))
```

```
(define (top)  
  (vide? `())  
  (else (car PILE)))
```

Évaluation différée

- ▶ Ou évaluation paresseuse, consiste à ne pas évaluer immédiatement une expression

(delay exp) ; retourne une 'promesse' d'évaluation

(force promesse) ; force l'évaluation promise

Exemple

```
(define (produit x y)
  (if (negative? x) (* x x) (* x (force y))))  
  
(produit v (delay (sqrt v)))
```

Avec les listes

- ▶ Dans certains cas, il est possible d'obtenir un résultat sans avoir à évaluer tous les éléments d'une liste.

```
(define (suite n1 n2 N)
  (if (zero? N) '()
    (cons (+ n1 n2) (suite n2 (+ n1 n2) (- N 1)))))
```

```
(suite 0 1 10)
(1 2 3 5 8 13 21 34 55 89)
```

Exemple sans delay

```
(define (membre-ord? nbre L)
  (cond ((null? L) '())
        ((< nbre (car L)) #f)
        ((= nbre (car L)) #t)
        (#T (membre-ord? nbre (cdr L)))))

(membre-ord? 15 (suite 0 1 200))
; tous les éléments sont générés!
()
```

Exemple avec delay

```
(define (suite n1 n2 N)
  (if (zero? N) '()
    (cons (+ n1 n2)
          (delay (suite n2 (+ n1 n2)
                        (- N 1))))))

(define (membre-ord? nbre L)
  (let ((premier (car L)))
    (cond ((null? L) #f)
          ((< nbre premier) #f)
          ((= nbre premier) #t)
          (#t (membre-ord? nbre (force
                                  (cdr L)))))))
```

Caractères

- ▶ Constante:

`#\a #\A #\(` #\space #\newline`

- ▶ Prédicats:

`(char? obj)
(char-alphabetic? char)
(char-numeric? char)
(char-whitespace? char)
(char-upper-case? char)
(char-lower-case? char)`

Caractères

▶ Fonctions:

```
(char=? char_1 char_2)
(char<? char_1 char_2)
(char>? char_1 char_2)
(char<=? char_1 char_2)
(char>=? char_1 char_2)
```

▶ Avec **-ci**, ces fonctions deviennent indépendantes à la casse:

```
> (char=? #\a #\A)
#f
> (char-ci=? #\a #\A)
#t
```

Manipulation des caractères

```
(char->integer #\a)
```

```
97
```

```
(integer->char (+ 1 (char->integer #\a)))  
#\b
```

Chaine de caractères

```
(string=? string_1 string_2)
(string<? string_1 string_2)
(string>? string_1 string_2)
(string<=? string_1 string_2)
(string>=? string_1 string_2)
```

```
(string=? "Coco" "coco")
#f
(string-ci=? "Coco" "coco")
#t
(string-length "Bravo")
5
(string->list "Bravo")
(#\B #\r #\a #\v #\o)
(substring "computer" 3 6)
"put"
```

Code César

```
(define (cesar-char char k)
  (if (char-alphabetic? char)
      (let ((base (if (char-upper-case? char)
                      (char->integer #\A)
                      (char->integer #\a))))
        (integer->char
         (+ base
            (modulo (+ k (- (char->integer char) base)) 26)))
      )
    (char)))
```



```
> (cesar-char #\r 7)
#\y
```

Entrees / Sorties

- ▶ **read** – retourne une entrée du clavier

```
> (read)  
234      ; entrée de l'utilisateur  
234      ; valeur renvoyée par la fonction  
> (read)  
"hello world"  
"hello world"
```

- ▶ **display** – affiche son paramètre sur écran

```
> (display "hello world")  
hello world  
> (display (+ 2 3))  
5
```

- ▶ **newline** – affiche une nouvelle ligne

Entrees / Sorties

- ▶ Définir une fonction qui lit un nombre (si ce n'est pas un nombre, elle continue de demander un nombre):

```
> (define (ask-number)
  (display "Entrez un nombre: ")
  (let ((n (read)))
    (if (number? n)
        n
        (ask-number))))
```

```
> (ask-number)
Entrez un nombre: a
Entrez un nombre: (5 6)
Entrez un nombre: "mais pourquoi ?"
Entrez un nombre: 12
12
```

Entrees / Sorties

- ▶ Une fonction qui lit un entier, fait appel à la factorielle et affiche le résultat:

```
(define (fact-interactif)
  (display "Entrez un entier: ")
  (let ((n (read)))
    (display "La factorielle de ")
    (display n)
    (display " est ")
    (display (fact n))
    (newline)))
```

```
> (fact-interactif)
Entrez un entier: 4
La factorielle de 4 est 24
```

Port d'entrée

```
(let ((p (open-input-file "fichier.txt")))
  (let f ((x (read p)))
    (if (eof-object? x)
        (begin
          (close-input-port p)
          '())
        (cons x (f (read p)))))))
```

Appel avec un port d'entrée

```
(call-with-input-file "fichier.txt"
  (lambda (p)
    (let f ((x (read p)))
      (if (eof-object? x)
          '()
          (cons x (f (read p)))))))
```

Définition équivalente

```
(define call-with-input-file
  (lambda (filename proc)
    (let ((p (open-input-file filename)))
      (let ((v (proc p)))
        (close-input-port p)
        v))))
```

Sauvegarder une liste

```
(let ((p (open-output-file "fichier.txt")))
  (let f ((ls list-to-be-printed))
    (if (not (null? ls))
        (begin
          (write (car ls) p)
          (newline p)
          (f (cdr ls))))))
  (close-output-port p))
```

Ou encore

```
(call-with-output-file "myfile.ss"
  (lambda (p)
    (let f ((ls list-to-be-printed))
      (if (not (null? ls))
          (begin
            (write (car ls) p)
            (newline p)
            (f (cdr ls)))))))
```

Définition équivalente

```
(define call-with-output-file
  (lambda (filename proc)
    (let ((p (open-output-file filename)))
      (let ((v (proc p)))
        (close-output-port p)
        v))))
```

Vecteurs en Scheme

- ▶ Constructeurs and accesseurs

```
> (define v (vector 1 (+ 1 2)))  
 #(1 3)  
> (vector 'a 'b 'c)  
 #(a b c)  
> (vector-ref v 0)  
 1  
> (vector-length v)  
 2  
> (vector-set! v 3 10)  
 2
```

- ▶ L'index commence à 0.

vector-set!

```
(let ((v (vector 'a 'b 'c 'd 'e)))
  (vector-set! v 2 'x) v)
```

```
#(a b x d e)
```

```
(define vector-fill!
  (lambda (v x)
    (let ((n (vector-length v)))
      (do ((i 0 (+ i 1)))
          ((= i n))
        (vector-set! v i x)))))
```

```
(let ((v (vector 1 2 3)))
  (vector-fill! v 0) v)
 #(0 0 0)
```

conversion de listes à vecteurs

```
(list->vector L)
(vector->list L)
```

```
(define vector->list
  (lambda (s)
    (do ((i (- (vector-length s) 1) (- i 1))
          (ls '() (cons (vector-ref s i) ls)))
        ((< i 0) ls)))

(vector->list '#(a b c))
(a b c)

(let ((v '#(1 2 3 4 5)))
  (apply * (vector->list v)))
```

Exemple

```
(let ((v '#(1 2 3 4 5)))
  (let ((ls (vector->list v)))
    (list->vector (map * ls ls))))  
  
#(1 4 9 16 25)
```

Exemple

```
(define vector-sum (lambda (vec)
  (let ((size (vector-length vec))
        (position 0) (total 0))
    (do () ((= position size) total)
      (set! total (+ total
                      (vector-ref vec position)))
      (set! position (+ position 1))))))
```

```
(define vector-sum (lambda (vec)
  (do ((remaining (vector-length vec)
                  (- remaining 1))
       (total 0 (+ total
                      (vector-ref vec (- remaining 1)))))
    ((zero? remaining) total))))
```

Tri des vecteurs et des listes

```
(quick-sort '#(3 4 2 1 2 5) <)
 #(1 2 2 3 4 5)
```

```
(merge-sort '(0.5 1.2 1.1) >)
 (1.2 1.1 0.5)
```

On doit avoir:

```
(and (test x y) (test y x))
 #f
```