

# La fonction map

- ▶ Permet d'appliquer une fonction aux éléments d'une liste
  - peut accepter plusieurs arguments

```
(map abs '(1 -2 3 -4 5 -6))  
(1 2 3 4 5 6 )
```

```
(map (lambda (x y) (* x y))  
      '(1 2 3 4) '(8 7 6 5))  
(8 14 18 20)
```

# Extraction des préfixes d'une liste

```
(define (extraire-prefixes l)
  (define (extraire-prefixes2 l l1)
    (if (null? l) l1
        (extraire-prefixes2 (cdr l)
                            (map (lambda (u) (cons (car l) u))
                                 (cons '() l1)))))

  (extraire-prefixes2 (reverse l) '())))

> (extraire-prefixes '( 1 2 3 4 ))
' ((1) (1 2) (1 2 3) (1 2 3 4))
```

# La fonction filter

- ▶ Permet d'éliminer des éléments ne répondant pas à la condition définie par le filtre

```
(filter (lambda (x) (> x 0))  
       '(1 -2 3 -4 5 -6))  
'(1 3 5)
```

# La fonction filter

```
(define (notre-filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst))
         (cons (car lst) (notre-filter pred (cdr lst)))))
        (else (notre-filter pred (cdr lst))))))
```

Ou...

```
(define (notre-filter pred lst)
  (reverse (filter-help pred lst '())))

(define (filter-help pred lst res)
  (cond ((null? lst) res)
        ((pred (car lst))
         (filter-help pred (cdr lst) (cons (car lst) res)))
        (else
         (filter-help pred (cdr lst) res))))
```

# La fonction build-list

- ▶ Elle permet de construire une liste d'entiers à partir de la séquence des nombres naturels

```
> (build-list 10 (lambda (n) (* n 2)))  
'(0 2 4 6 8 10 12 14 16 18)
```

```
> (build-list 10 (lambda (n) n))  
'(0 1 2 3 4 5 6 7 8 9)
```

# Définitions locales: let, let\*, letrec

## ▶ let

- permet de définir une liste de variables locales à un bloc
- à chaque nom de variable est associé une valeur
- **let** retourne la dernière expression dans le bloc

```
> (let ((a 2) (b 3)) ; variables locales  
      (+ a b)) ; bloc où les variables sont définies
```

5

```
> a          => Error: variable a is not bound.  
> b          => Error: variable b is not bound.
```

# Définitions locales: let, let\*, letrec

$$f(x, y) = x^* (1+x*y)^2 + y^* (1-y) + (1+x*y) * (1-y)$$

$$\begin{aligned}a &= 1+x*y \\b &= 1-y\end{aligned}$$

$$f(x, y) = x*a^2 + y*b + a*b$$

```
>(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x a a) (* y b) (* a b))))
```

```
> (f 1 2)  
4
```

# Définitions locales: let, let\*, letrec

- ▶ **let** permet aussi de définir des fonctions locales:

```
>(let ((a 3)
       (b 4)
       (square (lambda (x) (* x x))))
  (plus +))
(sqrt (plus (square a) (square b)))
=> 5
```

# Definitions locales: let, let\*, letrec

- let permet une assignation en parallèle:

```
> (define x 'a)
```

```
> (define y 'b)
```

```
> (list x y)           => (a b)
```

```
> (let ((x y) (y x)) (list x y)) => (b a)
```



1. d'abord évaluer toutes les expressions dans la liste
2. Ensuite associer les noms aux valeurs.

# Définitions locales: let, let\*, letrec

```
> (let ((x 1)
        (y (+ x 1)))
    (list x y)) => Error: variable x is not bound.
```

- ▶ Pour permettre de définir y en termes de x:
  - besoin d'utiliser **let\***
- ▶ **let\*** – similaire à **let**, mais permet une association séquentielle

# Définitions locales: let, let\*, letrec

```
> (let* ((x 1)
          (y (+ x 1)))
    (list x y)) => (1 2)
```

- ▶ Comment on peut utiliser let seulement?

```
> (let ((x 1))
      (let ((y (+ x 1)))
        (list x y)))
```

# Exemple

```
(let ((x 2) (y 3))  
  (let ((x 7)  
        (z (+ x y)))  
    (* z x)))
```

35

```
(let ((x 2) (y 3))  
  (let* ((x 7)  
         (z (+ x y)))  
    (* z x)))
```

70

# Définitions locales: let, let\*, letrec

- ▶ **letrec** – similaire à **let\***, mais permet de définir des fonctions récursives
- ▶ Définir la factorielle localement:

```
> (letrec ((fact (lambda (n)
                     (if (= n 1)
                         1
                         (* n (fact (- n 1)))))))
      (fact 5))
```

=> 120

# Application récursive d'une fonction à une liste

```
(define (application fct)
  (letrec ((app (lambda (L)
                  (if (null? L) ()
                      (cons (fct (car L)) (app (cdr L)))))))
    app))

((application traite) L) ; la fonction traite est
; appliquée à tous les éléments de la liste
```

# named let

(let *name* ((*var val*) ...)  
  *exp*<sub>1</sub> *exp*<sub>2</sub> ...)

est équivalent à:

(letrec ((*name* (lambda (*var* ...) *exp*<sub>1</sub> *exp*<sub>2</sub> ...)))  
  (*name val* ...))

# exemple

```
(define divisors
  (lambda (n)
    (let f ((i 2))
      (cond
        ((>= i n) '())
        ((integer? (/ n i)))
        (cons i (f (+ i 1))))))
    (else (f (+ i 1)))))))
```

(divisors 32)  
(2 4 8 16)

# Un autre exemple

```
(let loop ((numbers ' (3 -2 1 6 -5)) (nonneg ' ()) (neg ' ()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers) (cons (car numbers) nonneg) neg))
        ((< (car numbers) 0)
         (loop (cdr numbers) nonneg (cons (car numbers) neg))))))

((6 1 3) (-5 -2))
```

# Le tri de listes

- ▶ Avec le tri fusion
  - Extraire des sous-listes
  - Subdiviser une liste
  - Fusion de listes
- ▶ Avec le tri rapide
  - Choisir un pivot
  - Partitionner une liste
  - Trier une sous-liste

# Extraire une sous-liste

```
(define (sub L start stop ctr)
; extract elements start to stop into a list
(cond ( (null? L) L)
      ( (< ctr start) (sub (cdr L) start stop (+ ctr 1)))
      ( (> ctr stop) '() )
      (else (cons (car L)
                   (sub (cdr L) start stop (+ ctr 1)))) ) ) )
```

# Diviser une liste en deux

```
(define (split L)
; division de la liste en 2:
; retourne ((1ere moitié) (2nde moitié))
(let ((len (length L)))
  (cond ((= len 0) (list L L) )
        ((= len 1) (list L '() ))
        (else (list (firstHalf L (/ len 2))
                     (lastHalf L (/ len 2)))))))
(define (firstHalf L N)
  (if (= N 0)
      null
      (if (or (= N 1) (< N 2))
          (list (car L))
          ;else
          (cons (car L) (firstHalf (cdr L) (- N 1))))))
(define (lastHalf L N)
  (if (= N 0) L
      (if (or (= N 1) (< N 2))
          (cdr L)
          ;else
          (lastHalf (cdr L) (- N 1)))
      )))
```

# Fusion de 2 listes triées

```
(define (mergelists L M)
; supposer L et M déjà triés
(cond ( (null? L) M)
      ( (null? M) L)
      ( (< (car L) (car M)) (cons (car L)
                                     (mergelists (cdr L)M)))
      (else (cons (car M) (mergelists L (cdr M)))) ) )
```

# Tri fusion

```
(define (mergesort L)
  (cond ((null? L) '())
        ((= 1 (length L)) L)
        ((= 2 (length L)) (mergelists (list (car L))
                                         (cdr L)))
        (else (mergelists (mergesort (car (split L)) )
                           (mergesort (car (cdr (split L))))))))
  ))
```

# quicksort

```
(define (qsort e)
  (if (or (null? e) (<= (length e) 1)) e
      (let loop ((left null) (right null) ; named let
                (pivot (car e)) (rest (cdr e)))
        (if (null? rest)
            (append (append (qsort left) (list pivot))
                    (qsort right))
            (if (<= (car rest) pivot) ; partition
                (loop (append left (list (car rest)))
                      right pivot (cdr rest))
                (loop left
                      (append right (list (car rest)))
                      pivot (cdr rest)))))))
```