L'intérêt du processus

```
(* 1 (fact 4))
(* 4 (fact 3))
(* 12 (fact 2))
(* 24 (fact 1))
(* 24 (fact 0))
24
```

est que chaque état est caractérisé par deux paramètres seulement. On peut générer un tel processus très simplement, en faisant de ces paramètres les arguments d'une fonction `fact-a` :

```
(fact-a 4 1)
(fact-a 3 4)
(fact-a 2 12)
(fact-a 1 24)
(fact-a 0 24)
24
```


La définition de `fact-a` est évidente :

```
(define fact-a
  (lambda (n a)
    (if (zero? n) a (fact-a (- n 1) (* a n)))))
```

Si [[n]] $= n$ et [[a]] $= a$,
alors [[`(fact-a n a)`]] $= n!a$.

Le processus généré est analogue à celui associé à la boucle

```
while n > 0 do (a,n) := (a*n,n-1)
```

Processus itératif : espace-mémoire de contrôle constant.

```
(define fact (lambda (n) (fact-a n 1)))
```


Variante.

On observe qu'au cours de l'exécution l'accumulateur a pour valeur un produit partiel, tel $4*3$ ou $4*3*2$ (si on calcule 4!). Rien n'empêche d'utiliser plutôt des produits partiels du type $1*2$ ou $1*2*3$.

On définit `(fact-b n i b)` par la spécification suivante :

Si [[n]] $= n$, [[i]] $= i$ et [[b]] $= (i-1)!$ et si $1 \leq i \leq n+1$,
alors [[`(fact-b n i b)`]] $= n!$.

On peut écrire :

```
(define fact-b
  (lambda (n i b)
    (if (> i n) b (fact-b n (+ i 1) (* b i)))))

(define fact
  (lambda (n) (fact-b n 1 1)))
```


Un exemple classique

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

Problème : recalcul inutile de résultats intermédiaires.
Solution : utiliser des accumulateurs.

```
(define fib-a
  (lambda (n a b)
    (if (zero? n)
        a
        (fib-a (- n 1) b (+ a b)))))
```

```
(fib-a  9  0  1)
(fib-a  8  1  1)
(fib-a  7  1  2)
(fib-a  6  2  3)
(fib-a  5  3  5)
(fib-a  4  5  8)
(fib-a  3  8 13)
(fib-a  2 13 21)
(fib-a  1 21 34)
(fib-a  0 34 55)
34
```

Processus itératif, comme pour la boucle

```
while n > 0 do (n,a,b) := (n-1,b,a+b)
```

Autres exemples numériques I

```
(define expt
 (lambda (m n)
  (cond ((zero? n) 1)
        ((even? n) (expt (* m m) (/ n 2)))
        ((odd? n) (* m (expt m (- n 1)))))))

(define expt-a  ;; à spécifier
 (lambda (m n a)
  (cond ((zero? n) a)
        ((even? n) (expt-a (* m m) (/ n 2) a))
        ((odd? n) (expt-a m (- n 1) (* m a))))))

(expt 5 4)              625
(expt-a 5 4 1)          625
(expt-a 5 4 10)        6250
```


Autres exemples numériques II

```
(define cbin
 (lambda (n u)
  (if (zero? n)
      1
      (/ (* (cbin (- n 1) (- u 1)) u) n))))

(define cbin-a  ;; à spécifier
 (lambda (n u a)
  (if (zero? n)
      a
      (cbin-a (- n 1) (- u 1) (/ (* u a) n)))))

(cbin 4 6)              15
(cbin-a 4 6 1)          15
(cbin-a 4 6 10)        150
```


Exemple avec exploration superficielle I

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

(reverse '(0 1 2))
(append (reverse '(1 2)) '(0))
(append (append (reverse '(2)) '(1)) '(0))
(append (append (append (reverse '()) '(2)) '(1)) '(0))
(append (append (append '() '(2)) '(1)) '(0))
(append (append '(2) '(1)) '(0))
(append '(2 1)) '(0))                          (2 1 0)
```

Temps d'exécution quadratique

Exemple avec exploration superficielle II

```
(define reverse-a
  (lambda (u a)
    (if (null? u)
        a
        (reverse-a (cdr u) (cons (car u) a)))))
```

Temps d'exécution linéaire, processus itératif

```
(reverse-a '(0 1 2) '())
(reverse-a '(1 2) '(0))
(reverse-a '(2) '(1 0))
(reverse-a '() '(2 1 0))                 (2 1 0)
```

$\boxed{\texttt{[[(append (reverse u) a)]]} = \texttt{[[(reverse-a u a)]]}}$

Exemple avec exploration profonde

```
(define flat-l
  (lambda (l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (if (null? (car l)) (flat-l (cdr l)) (cons (car l) (flat-l (cdr l)))))
      ((list? (car l)) (append (flat-l (car l)) (flat-l (cdr l)))))))

(define flat-a
  (lambda (l a)
    (cond
      ((null? l) a)
      ((atom? (car l))
       (if (null? (car l))
           (flat-a (cdr l) a)
           (cons (car l) (flat-a (cdr l) a))))
      ((list? (car l)) (flat-a (car l) (flat-a (cdr l) a))))))
```

$\boxed{[[\texttt{(append (flat-l l) a)}]] = [[\texttt{(flat-a l a)}]]}$

Gain de temps et d'espace, processus *non* itératif.

Exemple, récursivité non structurelle I

```
(define M (lambda (x) (if (> x 100) (- x 10) (M (M (+ x 11))))))
```

On peut démontrer que la valeur de `(M x)` existe
quelle que soit la valeur de l'entier (relatif) `x`.

On peut aussi prouver que le graphe de `M` coïncide avec celui de `M91` :

```
(define M91 (lambda (x) (if (> x 100) (- x 10) 91)))
```

```
(define M-c
  (lambda (x c)
    (cond ((= c 0) x)
          ((> x 100) (M-c (- x 10) (- c 1)))
          (else (M-c (+ x 11) (+ c 1))))))
```

[[(M-c x 0)]] = [[x]] ;
[[(M-c x 1)]] = [[(M x)]] ;
[[(M-c x 2)]] = [[(M (M x))]] ;
[[(M-c x 3)]] = [[(M (M (M x)))]] ;
. . .

Exemple, récursivité non structurelle II

```
(define ack   ;; Fonction totale sur NxN
  (lambda (m n)
    (cond
      ((zero? m) (+ n 1))
      ((zero? n) (ack (- m 1) 1))
      (else (ack (- m 1) (ack m (- n 1)))))))

(define acl
  (lambda (l)
    (cond
      ((null? (cdr l)) (car l))
      ((zero? (cadr l)) (acl (cons (+ (car l) 1) (cddr l))))
      ((zero? (car l)) (acl (cons 1 (cons (- (cadr l) 1) (cddr l)))))
      (else (acl (cons (- (car l) 1) (cons (cadr l) (cons (- (cadr l) 1) (cddr l)))))))))
```

[[(acl '(n))]] = [[n]] ;
[[(acl '(n m))]] = [[(ack m n)]] ;
[[(acl '(n m u))]] = [[(ack u (ack m n))]] ;
[[(acl '(n m u v))]] = [[(ack v (ack u (ack m n)))]] ;
. . .

Programmation CPS I

Reconsidérons deux états homologues des processus de calcul associés aux évaluations des formes `(fact 4)` et `(fact-a 4 1)`, soient

`(* 4 (* 3 (fact 2)))`   `(fact-a 2 12)`

On a trois paires de constituants homologues :

fact	fact-a
2	2
(* 4 (* 3 ...))	12

Il est plus économique de mémoriser le nombre 12 que la fonction `(* 4 (* 3 ...))` ou, plus exactement, l'expression `(lambda (k) (* 4 (* 3 k)))`.

Programmation CPS II

On peut cependant utiliser un argument fonctionnel, et on le devrait si la multiplication n'était pas associative :

```
(define fact-c
  (lambda (n c)
    (if (zero? n)
        (c 1)
        (fact-c (- n 1)
                (lambda (k) (c (* n k)))))))

(define fact
  (lambda (n) (fact-c n (lambda (k) k))))
```

> Si n est un naturel
> et si c est une fonction de $\mathbb{N}$ dans D,
> alors $[\![$`(fact-c` n c `)`$]\!] = c(n!)$.

Programmation CPS III

On voit que l'associativité n'est pas utilisée :

```
(fact-c 4 (lambda (k) k))
(fact-c 3 (lambda (k) ((lambda (k) k) (* 4 k))))
(fact-c 3 (lambda (k) (* 4 k)))
(fact-c 2 (lambda (k) ((lambda (k) (* 4 k)) (* 3 k))))
(fact-c 2 (lambda (k) (* 4 (* 3 k))))
...
(fact-c 0 (lambda (k) (* 4 (* 3 (* 2 (* 1 k))))))
((lambda (k) (* 4 (* 3 (* 2 (* 1 k))))) 1)
(* 4 (* 3 (* 2 (* 1 1))))
24
```


Programmation CPS IV

Remarque. Scheme n'effectue les réductions qu'à la fin du processus. La dernière étape n'est donc pas l'évaluation de

```
((lambda (k) (* 4 (* 3 (* 2 (* 1 k))))) 1)
```

mais celle de

```
((lambda (k)
   ((lambda (k)
      ((lambda (k)
         ((lambda (k)
            ((lambda (k) k) (* 4 k)))
          (* 3 k)))
       (* 2 k)))
    (* 1 k)))
 1)
```


Programmation CPS V

L'argument fonctionnel auxiliaire `c` est une *continuation*. Cette fonction, appliquée à un résultat intermédiaire du calcul en cours, fournit le résultat final.

Dans le cas de `fact-c`, l'exécution construit itérativement une fonction de plus en plus complexe, représentant l'enchaînement des multiplications à faire. A chaque étape, l'argument continuation est transformé en une fonction impliquant une multiplication supplémentaire ; cette fonction est l'argument de l'appel suivant. Les calculs numériques n'ont lieu que quand l'enchaînement complet des multiplications a été formé. Cette technique est le *Continuation-Passing Style* (CPS). Elle ressemble à la technique de séparation fonctionnelle vue précédemment.

Programmation CPS VI

Produit de liste : $\ell \mapsto \prod_{x\in\ell} x$.

```
(define pl
  (lambda (l)
    (cond ((null? l) 1)
          ((zero? (car l)) 0)
          (#t (* (car l) (pl (cdr l)))))))
```

Si un facteur est nul, comment éviter *toutes* les multiplications ?
Séparation fonctionnelle, ou encore CPS :

```
(define pl-c
  (lambda (l c)
    (cond ((null? l) (c 1))
          ((zero? (car l)) 0)
          (#t (pl-c (cdr l) (lambda (k) (c (* (car l) k))))))))

(define pl (lambda (l) (pl-c l (lambda (k) k))))
```

On comparera les couples (`pl-c` , `pl`) et (`p2` , `pl2`), tr. 86.

Programmation CPS VII

On peut remplacer la continuation par un argument non fonctionnel. Pour `pl`, cela revient à tester l'absence de facteur nul avant d'opérer les multiplications.

```
(define pl-l
  (lambda (l a) ;; la liste a ne comporte pas de 0
    (cond ((null? l) (p a))
          ((zero? (car l)) 0)
          (else (pl-l (cdr l) (cons (car l) a))))))

(define pl (lambda (l) (pl-l l '())))

;; p est une procedure calculant le produit
;; d'une liste dont aucun facteur n'est nul.
```

Inconvénient : double parcours de liste si aucun facteur nul n'est présent.

Programmation CPS VIII

Double comptage – rappel. La séparation fonctionnelle permet d'éviter efficacement le double parcours de la liste :

```
(define c2
  (lambda (l s c)
    (if (null? l)
        c
        (if (eq? (car l) s)
            (c2 (cdr l)
                s
                (lambda (u) (c (cons (1+ (car u)) (cdr u)))))
            (c2 (cdr l)
                s
                (lambda (u) (c (cons (car u) (1+ (cdr u))))))))))

(define id (lambda (v) v))

(define count2 (lambda (l s) ((c2 l s id) '(0 . 0))))
```


Programmation CPS IX

La solution en CPS est semblable :

```
(define c3
  (lambda (l s c)
    (if (null? l)
        (c '(0 . 0))
        (if (eq? (car l) s)
            (c3 (cdr l)
                s
                (lambda (u) (c (cons (1+ (car u)) (cdr u)))))
            (c3 (cdr l)
                s
                (lambda (u) (c (cons (car u) (1+ (cdr u))))))))))

(define count3 (lambda (l s) (c3 l s id)))
```

Programmation CPS X

Une variante permet l'économie des opérations sur les paires :

```
(define c3
  (lambda (l s c)
    (if (null? l)
        (c 0 0)
        (if (eq? (car l) s)
            (c3 (cdr l)
                s
                (lambda (u v) (c (1+ u) v)))
            (c3 (cdr l)
                s
                (lambda (u v) (c u (1+ v))))))))

(define count3 (lambda (l s) (c3 l s cons)))
```


Programmation CPS XI

Spécification :

> Si s est un symbole,
> si ℓ est une liste de symboles
> et si c est une fonction de $\mathbb{N} \times \mathbb{N}$ dans D,
> alors (c3 ℓ s c) vaut $c(a, b)$,
> où a est le nombre d'occurrences de s dans ℓ
> et où b est le nombre d'occurrences
> de symboles distincts de s dans ℓ.

Programmation CPS XII

L'argument fonctionnel évolue comme suit.

Si sa valeur “avant” un appel récursif est celle de `c`, soit

```
(lambda (x y) (c x y))
```

sa “nouvelle” valeur sera celle d'une des expression

```
(lambda (x y) (c (1+ x) y))
(lambda (x y) (c x (1+ y)))
```


Programmation CPS XIII

On peut remplacer l'argument fonctionnel par deux arguments numériques. Cela donne :

```
(define c4
  (lambda (l s a1 a2)
    (if (null? l)
        (cons a1 a2)
        (if (eq? (car l) s)
            (c4 (cdr l) s (1+ a1) a2)
            (c4 (cdr l) s a1 (1+ a2))))))

(define count4 (lambda (l s) (c4 l s 0 0)))
```

Cette solution est simple et optimale.

On voit que la séparation fonctionnelle et le CPS sont des techniques utiles, donnant lieu à des solutions efficaces.

Parfois, le remplacement de l'argument fonctionnel par un ou plusieurs accumulateur(s) améliore encore le programme.

Un schéma accumulant I

Si à toute liste de naturels la fonction h associe un naturel, on définit la fonction gib (pour "generalized Fibonacci") par

$$gib(n,h) \;=\; h([gib(n-1,h),\ldots,gib(0,h)])\,.$$

Pour une fonction h appropriée, on retrouve Fibonacci :

```
(define hfib
  (lambda (l)
    (cond ((null? l) 0)
          ((null? (cdr l)) 1)
          (else (+ (car l) (cadr l))))))

(hfib '(8 5 3 2 1 1 0))   13
```


Un schéma accumulant II

Cas général :

```
(define rev-enum       ;; 5 -> (5 4 3 2 1 0)
  (lambda (n)
    (if (< n 0) '() (cons n (rev-enum (- n 1))))))
```

on peut programmer *gib* en utilisant `map` :

```
(define gib0
  (lambda (n h)
    (h (map (lambda (i) (gib0 i h))
            (rev-enum (- n 1))))))

(gib0 12 hfib)   144
```

C'est correct, mais extrêmement inefficace.

Un schéma accumulant III

En utilisant une fonction auxiliaire, on obtient :

```
(define gib1
  (lambda (n h) (h (gib1* n h))))

(define gib1*
  (lambda (n h)
    (if (= n 0)
        '()
        (cons (gib1 (- n 1) h) (gib1* (- n 1) h)))))
```

Cette solution reste inefficace car l'évaluation du premier argument du `cons` implique la (ré)évaluation du second. Plus précisément, si la valeur du second argument est r, la valeur du premier argument est $h(r)$.

```
(gib1 10 hfib)   55

(gib1* 10 hfib)  (34 21 13 8 5 3 2 1 1 0)

(hfib '(34 21 13 8 5 3 2 1 1 0))   55
```


Un schéma accumulant IV

La variante suivante évite ce gaspillage :

```
(define gib2 (lambda (n h) (h (gib2* n h))))

(define gib2*
  (lambda (n h)
    (if (= n 0)
        '()
        ((lambda (rec) (cons (h rec) rec)) (gib2* (- n 1) h)))))
```

La fonction auxiliaire s'écrit aussi (variante syntaxique vue plus loin)

```
(define gib2*
  (lambda (n h)
    (if (= n 0)
        '()
        (let ((rec (gib2* (- n 1) h))) (cons (h rec) rec)))))
```

On a par exemple

```
(gib2* 12 hfib)                (89 55 34 21 13 8 5 3 2 1 1 0)
```

Un schéma accumulant V

Version accumulante et itérative :

```
(define gib (lambda (n h) (h (gib-a n h '()))))

(define gib-a
  (lambda (n h l)
    (if (= n 0) l (gib-a (- n 1) h (cons (h l) l)))))
```

Si `n`, `h` et `l` ont pour valeurs respectives
n, h et $[gib(i-1,h),\ldots,gib(0,h)]$,
alors `(gib-a n h l)` a pour valeur
$[gib(n+i-1,h),\ldots,gib(0,h)]$.
En particulier, `(gib-a n h '())` a pour valeur
$[gib(n-1,h),\ldots,gib(0,h)]$.

```
(gib-a 4 hfib '(21 13 8 5 3 2 1 1 0))    (144 89 55 34 21 13 8 5 3 2 1 1 0)
(gib 12 hfib)                            144
```


8. Expressions symboliques

Représentation des listes en mémoire

Principe. La représentation en mémoire de la valeur de `(cons` α β`)` est un couple de pointeurs vers les représentations des valeurs de α et β. Dans le cas des listes, β est une liste, mais le cas où β n'est pas une liste est admis aussi.

Extension. Une *expression symbolique* est un atome ou une paire formée d'expressions symboliques.

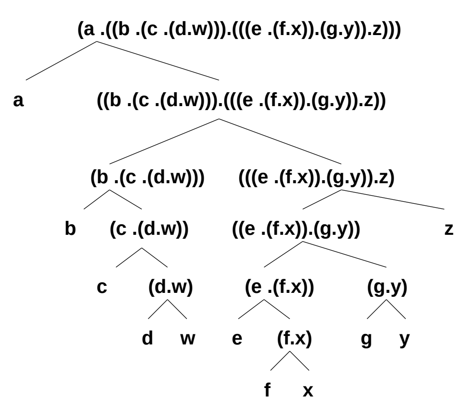
Notation pointée. Le point (entouré d'espaces) et les parenthèses représentent l'appariement.

```
'a              -->    a
(cons 'a 'b)    -->    (a . b)
(cons 'a '())   -->    (a . ())
(list a)        -->    (a . ())
(list a b)      -->    (a . (b . ()))
'(a b c d)      -->    (a . (b . (c . (d . ()))))
'((a b) (c))    -->    ((a . (b . ())) . ((c . ()) . ()))
```


Expressions symboliques représentées par des arbres binaires

Une expression symbolique est un arbre binaire dont les feuilles sont des atomes.

```
(a . ((b . (c . (d . w))) . (((e . (f . x)) . (g . y)) . z)))


(cons a
      (cons (cons b (cons c (cons d w)))
            (cons (cons (cons e (cons f x))
                        (cons g y))
                  z)))
```



Notation pointée et notation usuelle

La notation pointée met en évidence la structure d'*arbre binaire décoré* des expressions symboliques.
Chaque nœud a 0 (feuille) ou 2 (nœud interne) fils, auxquels on accède par `car` et `cdr`.
Chaque feuille est (étiquetée par) un atome.

```
a    ()    (b . 3)    ((a . b) . c)    ((7 . g) . (#f . (y . (z . ()))))
```

Les listes sont des expressions symboliques particulières ; chaque point est suivi d'une parenthèse ouverte :

```
()              ()
(0)             (0 . ())
(0 1)           (0 . (1 . ()))
(0 1 2)         (0 . (1 . (2 . ())))
((0 1) 2)       ((0 . (1 . ())) . (2 . ()))
```

Les constructeur et accesseurs et reconnaisseur sont `cons`, `car`, `cdr`, `pair?`.

De la notation pointée à la notation usuelle

Tout point suivi d'une parenthèse ouverte est supprimé, ainsi que la parenthèse ouverte et la parenthèse fermée correspondante.
L'ordre des suppressions est quelconque.
Un point non suivi d'une parenthèse ouverte n'est pas supprimable !

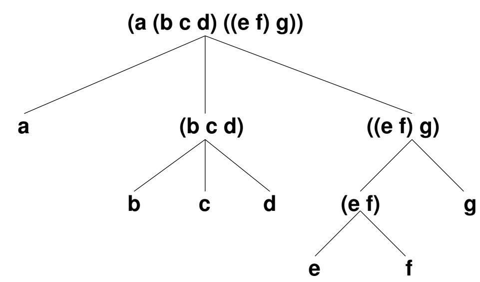
```
((0 . (1 . ())) . (2 . ()))
((0 . (1 . ())) . (2     ))
((0    1 . () ) . (2     ))
((0    1 . () )    2      )
((0    1      )    2      )

((a . (b . ())) . ((c . (d . ())) . ()))
((a . (b . ())) . ((c . (d . ()))))
((a . (b . ())) . ((c . (d))))
((a . (b . ())) . ((c d)))
((a . (b . ())) (c d))
((a . (b)) (c d))
((a b) (c d))

((a . b) . (c . d))
((a . b) c . d)
```

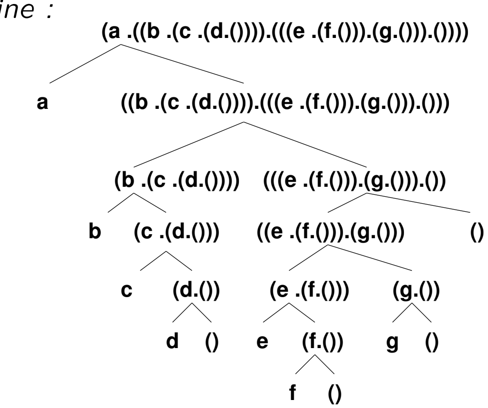

Représentations des listes

Une liste se représente conceptuellement par un arbre (quelconque). En machine, on représente plutôt (par un arbre binaire) l'expression symbolique équivalente (en fait, égale).

Représentations de la liste `(a (b c d) ((e f) g))`

conceptuelle : *en machine :*





Remarque. L'information attachée à un nœud interne se déduit de celle attachée à ses descendants ; seule l'information attachée aux feuilles est explicitement représentée.

Récursivité structurelle : les expressions symboliques

Schéma de base

```
(define F
  (lambda (s u)
    (if (atom? s)
        (G s u)
        (H (F (car s) (Ka s u))
           (F (cdr s) (Kd s u))
           s
           u))))
```

Schéma simplifié

```
(define F
  (lambda (s)
    (if (atom? s)
        (G s)
        (H (F (car s))
           (F (cdr s))
           s))))
```


Récursivité structurelle : exemples symboliques

```
(define size
  (lambda (s)
    (if (atom? s) 1 (+ (size (car s)) (size (cdr s))))))

(define flatten
  (lambda (s)
    (if (atom? s) (list s) (append (flatten (car s)) (flatten (cdr s))))))

(define flatten-a
  (lambda (s a)
    (if (atom? s) (cons s a) (flatten-a (car s) (flatten-a (cdr s) a)))))

(flatten-a '((a .(b .())).((c .(d .())).())) '(1 2)) (a b () c d () () 1 2)

(flatten-a '((a b) (c d)) '(1 2))                    (a b () c d () () 1 2)

(flat-a '((a b) (c d)) '(1 2))                       (a b c d 1 2)
```

Récursivité structurelle : listes et expressions symboliques

Une expression symbolique peut-elle s'écrire sans point ?

```
(define point-free?
  (lambda (s)
    (or (atom? s)
        (and (point-free? (car s))
             (list? (cdr s))
             (point-free? (cdr s))))))

(point-free? 'a)                    #t
(point-free? '((a . b)))            #f
(point-free? '(a . (b . ())))       #t
(point-free? '(a b . c))            #f
```


Egalité, identité

Egalité : `(equal? x y)` est

```
  (cond ((pair? x)
          (and (pair? y) (equal? (car x) (car y)) (equal? (cdr x) (cdr y))))
         ((pair? y) #f)
         (else (eqv? x y))))

(equal? '(a . (b . c)) '(a b . c))   #t
  eq?          =              eqv?                    equal?
----------------------------------------------------------------
symbol?    number?    symbol? number?     symbol? number? pair?
```

Identité. `(eq? x y)` : les valeurs de `x` et `y` sont le même objet en mémoire.

```
(define x '(a . b))   (define y '(a . b))   (define z x)
(eq? 'a 'a)                   #t
(eq? x y)                     #f
(eq? x z)                     #t
```


"Déconstruction"

```
(define describe
  (lambda (s)
    (cond ((null? s) (quote '()))
          ((number? s) s)
          ((symbol? s) (list 'quote s))
          ((pair? s) (list 'cons (describe (car s)) (describe (cdr s))))
          (else s))))

(describe '(1 ((a) 2)))
(cons 1 (cons (cons (cons 'a '())
                (cons 2 '())) '()))

(cons 1 (cons (cons (cons 'a '())
            (cons 2 '())) '()))
(1 ((a) 2))

(describe '((a . b) . (c . d)))
(cons (cons 'a 'b) (cons 'c 'd))

(cons (cons 'a 'b) (cons 'c 'd))
((a . b) c . d)
```


"Déconstruction – pretty printing"

```
(define lcons 6) ;; longueur de "(cons "

(define dis      ;; (dis n) ecrit n blancs
  (lambda (n)
    (if (zero? n)
        (display "")
        (begin (display " ") (dis (- n 1))))))

(define disp (lambda (n m) (dis (+ n (* m lcons)))))

(define pretty
  (lambda (d n m)
    (if (not (pair? d))
        (begin (disp n 0)
               (if (symbol? d) (display "'"))
               (display d))
        (begin (display "(cons ")
               (pretty (car d) 0 (+ m 1))
               (newline)
               (disp n (+ m 1))
               (pretty (cdr d) 0 (+ m 1))
               (display ")")))))

(define pr (lambda (d) (begin (newline) (pretty d 0 0))))
```

“Déconstruction” – exemples

```
(pr 'a)
'a

(pr '(a . (b . 1)))
(cons 'a
      (cons 'b
            1))

(pr '(a b c d))
(cons 'a
      (cons 'b
            (cons 'c
                  (cons 'd
                        ()))))

(pr '((a . (b . 1)) . ((c . 2) . ((d . 3) . e))))
(cons (cons 'a
            (cons 'b
                  1))
      (cons (cons 'c
                  2)
            (cons (cons 'd
                        3)
                  'e)))
```


9. Abstraction et blocs

Forme spéciale `let` I

Abstraire (nommer) une sous-expression

Calcul de $2(a+b)^2 + (a+b)(a-c)^2 + (a-c)^3$

Approche naïve :

```
(+ (* 2 (+ a b) (+ a b))
   (* (+ a b) (- a c) (- a c))
   (* (- a c) (- a c) (- a c)))
```

Approche économique et structurée :

```
(let ((x (+ a b)) (y (- a c)))
  (+ (* 2 x x)
     (* x y y)
     (* y y y)))
```


Forme spéciale `let` II

Evaluer $2(a+b)^2 + (a+b)(a-c)^2 + (a-c)^3$
en calculant d'abord $x = a+b$ et $y = a-c$,
c'est appliquer la fonction $(x, y) \mapsto 2x^2 + xy^2 + y^3$
aux arguments $x = a+b$ et $y = a-c$.

Définition

`(let ((x` α`) (y` β`))` γ`)` variante syntaxique de `((lambda (x y)` γ`)` α β`)`

```
(let ((x (+ a b))
      (y (- a c)))
  (+ (* 2 x x) (* x y y) (* y y y)))
```

est donc strictement équivalent à

```
((lambda (x y)
   (+ (* 2 x x) (* x y y) (* y y y)))
 (+ a b)
 (- a c))
```


Procédures locales

```
(define hypo
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))

(define square
  (lambda (x) (* x x)))
```

Comment empêcher l'usage autonome de `square` ?

```
(define hypo1
  (let ((square (lambda (x) (* x x))))
    (lambda (x y)
      (sqrt (+ (square x) (square y))))))
```

Variante

```
(define hypo2
  (lambda (x y)
    (let ((square (lambda (x) (* x x))))
      (sqrt (+ (square x) (square y))))))
```

Forme `let*` : abrège des `let` imbriqués

```
(let* ((x (+ a b)) (y (- a b)) (z (* x y)))
 (+ (* x x) z z (* y y)))

(let ((x (+ a b)) (y (- a b)))
  (let ((z (* x y)))
    (+ (* x x) z z (* y y))))

(let ((x (+ a b)))
  (let ((y (- a b)))
    (let ((z (* x y)))
      (+ (* x x) z z (* y y)))))
```

Ces trois formes sont équivalentes ; la valeur dépend des valeurs de `a` et `b`.

par contre, la valeur de

```
(let ((x (+ a b)) (y (- a b)) (z (* x y)))
  (+ (* x x) z z (* y y))))
```

dépend des valeurs de `a`, `b`, `x` et `y`.

Une fonction arithmétique I

$$f(n) =_{def} \left(\sum_{i=0}^{n-1}([2+f(i)]*[3+f(n-i-1)])\right) \bmod (2n+3)\,.$$

Version naïve, traduction littérale

```
(define f0
  (lambda (n)
    (modulo (apply +
                   (map (lambda (i)
                          (* (+ 2 (f0 i))
                             (+ 3 (f0 (- n i 1)))))
                        (enum 0 (- n 1))))
            (+ n n 3))))
```

```
(time (f0 12))  cpu time:  324        1
(time (f0 13))  cpu time:  964        8
(time (f0 14))  cpu time: 2937       16
```

Catastrophique !

Une fonction arithmétique II

```
(define gib (lambda (n h) (h (gib-a n h '()))))

(define gib-a
  (lambda (n h l) (if (= n 0) l (gib-a (- n 1) h (cons (h l) l)))))

(define harith
  (lambda (l)
    (modulo (apply +
                   (map (lambda (u v) (* (+ 2 u) (+ 3 v)))
                        l
                        (reverse l)))
            (let ((s (length l))) (+ s s 3)))))

(define f1 (lambda (n) (gib n harith)))
```


Une fonction arithmétique III

```
(define f (lambda (n) (fa n 0 '() '())))

(define fa
  (lambda (k i u v)
    (let ((next
           (modulo (apply +
                          (map (lambda (x y)
                                 (* (+ 2 x) (+ 3 y)))
                               u
                               v))
                   (+ i i 3))))
      (if (zero? k)
          next
          (fa (- k 1)
              (+ i 1)
              (cons next u)
              (append v (list next)))))))
```

Une fonction arithmétique IV

> Si `k`, `i`, `u` et `v` ont pour valeurs respectives
> les naturels i, k et les listes $[f(i-1), ..., f(0)]$ et $[f(0), ..., f(i-1)]$,
> alors `(fa k i u v)` a pour valeur $f(k+i)$.

```
(fa 8 5 '(10 3 3 1 0) '(0 1 3 3 10))         8
(f 13)                                       8

(time (f1   14))  cpu time:    0            16
(time (f1  140))  cpu time:    1           195
(time (f1 1400))  cpu time:  206          1477
(time (f  1400))  cpu time:  198          1477
```


Forme `letrec`

`letrec` construction essentielle !

```
(define power4
  (lambda (n)
    (let ((square (lambda (m) (* m m))))
      (square (square n)))))

(define fact
  (lambda (n)
    (letrec ((fact-it
               (lambda (k acc)
                 (if (= k 0) acc (fact-it (- k 1) (* k acc))))))
      (fact-it n 1))))
```

`let` permet des définitions locales non récursives seulement ;
`letrec` permet en plus des définitions locales récursives.

Portée I

Les principes de portée et de renommage relatifs à la forme `lambda` restent valables pour `let`, `let*` et `letrec`.

```
(define a 5)              ...
(add1 a)                  6
(let ((a 3)) (add1 a))    4
(let ((c 3)) (add1 c))    4
(add1 3)                  4
(add1 a)                  6

(define f
  (let ((b 100))
    (lambda (x) (+ x b))))   ...
(let ((b 10)) (f 25))     125
```

La plupart des confusions éventuelles proviennent d'un téléscopage de noms. Il suffit d'appliquer méthodiquement les règles d'évaluation pour éviter les erreurs. On peut aussi renommer (mentalement) certaines variables liées.

Portée II

```
(let ((a 5))
  (let ((fun (lambda (x) (max x a))))
    (let ((a 10) (x 20))
      (fun 1))))                          5

(let ((c 5))
  (let ((fun (lambda (y) (max y c))))
    (let ((a 10) (x 20))
      (fun 1))))                          5

(let ((fun (lambda (x) (max x 5))))
  (let ((a 10) (x 20))
    (fun 1)))                             5

(let ((fun (lambda (x) (max x 5))))
  (fun 1))                                5

((lambda (x) (max x 5)) 1)                5

(max 1 5)                                 5
```

Schémas récursifs avec `let` I

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (H (F (- n 1) (K n u)) n u))))
```

Seule exigence pour `G`, `H` et `K` : la terminaison.
`H` peut être *difficile à découvrir* (c'est rarement le cas pour `G` et `K`).
`H` peut induire un *calcul multiple d'argument* (pas `G` ni `K`).
On réduit significativement le premier problème . . .
et on élimine le second en utilisant `let` :

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (let ((v (F (- n 1) (K n u))))
          (H v n u)))))   ;; utile si plusieurs occurrences de v
```


Schémas récursifs avec `let` II

La variante avec `let` est spécialement utile si `(H v n u)` est une expression complexe, comportant plusieurs occurrences de `v`.

```
(define f
  (lambda (n u)
    (if (zero? n)
        u
        (let ((v (f (- n 1) u)))
          (/ (- 2 v (/ (* v v) u)) 3)))))

(define f
  (lambda (n u)
    (if (zero? n)
        u
        (/ (- 2 (f (- n 1) u)
              (/ (* (f (- n 1) u)
                    (f (- n 1) u)) u))
           3)))))
```

Le premier programme est d'efficacité linéaire en `n`,
tandis que le second, moins lisible, est exponentiel.

Schémas récursifs avec `let` III

Double comptage − rappel.

```
(count '(a b a c d a c) 'a)   (3 . 4)
```

L'utilisation brutale du schéma donne un programme inefficace :

```
(define count1
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        (if (eq? (car l) s)
            (cons (1+ (car (count1 (cdr l) s)))
                  (cdr (count1 (cdr l) s)))
            (cons (car (count1 (cdr l) s))
                  (1+ (cdr (count1 (cdr l) s))))))))
```


Schémas récursifs avec `let` IV

Une solution efficace est possible avec `let` :

```
(define count5
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        (let ((rec (count5 (cdr l) s)))
          (if (eq? (car l) s)
              (cons (1+ (car rec))
                    (cdr rec))
              (cons (car rec)
                    (1+ (cdr rec)))))))))
```

Schémas récursifs avec `let` V

La solution est strictement équivalente à

```
(define count5
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        ((lambda (rec)
           (if (eq? (car l) s)
               (cons (1+ (car rec))
                     (cdr rec))
               (cons (car rec)
                     (1+ (cdr rec)))))
         (count5 (cdr l) s)))))
```

On comparera cette solution aux précédentes
et on notera l'utilité de l'omniprésent `lambda`.

Sous-ensembles I

Les sous-ensembles de $\{a, b, c\}$ sont :

$$\{\,\}\,,\,\{a\}\,,\,\{b\}\,,\,\{c\}\,,\,\{a,b\}\,,\,\{a,c\}\,,\,\{b,c\}\,,\,\{a,b,c\}\,.$$

On pourrait générer séparément les sous-ensembles de 0, de 1, de 2 et de 3 éléments mais l'usage direct d'un schéma récursif est plus simple.

Représentation d'un ensemble : liste sans répétition.

On examine comment la liste des sous-ensembles de $\{a, b, c\}$ se construit au départ de la liste des sous-ensembles de $\{b, c\}$, c'est-à-dire

$$\{\,\}\,,\,\{b\}\,,\,\{c\}\,,\,\{b,c\}\,.$$

On observe d'abord que les sous-ensembles de $\{b, c\}$ sont aussi des sous-ensembles de $\{a, b, c\}$, mais que la réciproque n'est pas vraie ; les sous-ensembles manquants sont

$$\{a\}\,,\,\{a,b\}\,,\,\{a,c\}\,,\,\{a,b,c\}\,.$$

On observe ensuite que les sous-ensembles nouveaux sont les anciens dans lesquels on a inséré l'élément nouveau a.

Sous-ensembles II

```
(define subsets ;; version inefficace !
  (lambda (e)
    (if (null? e)
        '(())
        (append (subsets (cdr e)) (insert-in-all (car e) (subsets (cdr e)))))))
```

La fonction auxiliaire `insert-in-all` prend comme arguments un objet `x` et une liste de listes `ll`. Elle renvoie une liste de listes, dont les éléments sont ceux de `ll` préfixés de `x`.

```
(define insert-in-all
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons x (car ll)) (insert-in-all x (cdr ll))))))
```

Cette "solution" est correcte mais très inefficace : l'appel `(subsets e)`, quand `e` n'est pas vide, provoque deux appels récursifs à `(subsets (cdr e))`.

Sous-ensembles III

Version efficace :

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        ((lambda (le) (append le (insert-in-all (car e) le)))
         (subsets (cdr e))))))
```

Variante équivalente, plus lisible :

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        (let ((rec (subsets (cdr e))))
          (append rec (insert-in-all (car e) rec))))))
```

Partitions I

La structure de l'ensemble des partitions d'un ensemble donné est peu apparente, mais cela n'empêche pas l'usage de la récursion. Il est naturel de considérer d'abord un exemple. Les cinq partitions de $\{a,b,c\}$ sont

$$\begin{array}{l}\{\{a\},\{b\},\{c\}\}\,,\\ \{\{a\},\{b,c\}\}\,,\ \{\{b\},\{a,c\}\}\,,\ \{\{c\},\{a,b\}\}\,,\\ \{\{a,b,c\}\}\,.\end{array}$$

Les partitions de $\{b,c\}$ sont

$$\begin{array}{l}\{\{b\},\{c\}\}\,,\\ \{\{b,c\}\}\,.\end{array}$$

On observe qu'une partition de $\{a,b,c\}$ est obtenue au départ d'une partition de $\{b,c\}$ selon deux techniques :

1. En insérant le singleton $\{a\}$ comme partie supplémentaire ;
   $\{\{b\},\{c\}\}$ donne $\{\{a\},\{b\},\{c\}\}$;
   $\{\{b,c\}\}$ donne $\{\{a\},\{b,c\}\}$.
2. En insérant l'élément a dans une partie existante ;
   $\{\{b\},\{c\}\}$ donne $\{\{a,b\},\{c\}\}$ et $\{\{b\},\{a,c\}\}$;
   $\{\{b,c\}\}$ donne $\{\{a,b,c\}\}$.

Partitions II

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        (append (procede-1 (car e) (partitions (cdr e)))
                (procede-2 (car e) (partitions (cdr e)))))))
```

Attention à l'(in)efficacité !

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        (let ((rec (partitions (cdr e))))
          (append (procede-1 (car e) rec)
                  (procede-2 (car e) rec))))))
```


Partitions III

```
(define procede-1
  (lambda (x lp) (insert-in-all (list x) lp)))

(define insert-in-all
  (lambda (x le)
    (if (null? le)
        '()
        (cons (cons x (car le)) (insert-in-all x (cdr le))))))

(define procede-2
  (lambda (x lp) (append-map (lambda (p) (split x p)) lp)))

(define append-map
  (lambda (f l)
    (if (null? l)
        '()
        (append (f (car l)) (append-map f (cdr l))))))

(define append-map (lambda (f l) (apply append (map f l))))
```


Partitions IV

La fonction `split` réalise le procédé 2 proprement dit, pour une partition. Comme toujours lorsque l'on spécifie une fonction auxiliaire, il convient de le faire de la manière la plus générale possible. Le second argument ne sera donc pas nécessairement une partition, mais une quelconque liste de listes.

```
(split 'a '((b) (c)))      (((a b) (c)) ((b) (a c)))
(split 'a '((b c)))        (((a b c)))
(split '2 '((1 2) () (3))) (((2 1 2) () (3)) ((1 2) (2) (3)) ((1 2) () (2 3)
```

Construction de `split` : facile par la tactique habituelle ;
comment obtient-on `(split x ll)` à partir de `(split x (cdr ll))` ?
Un exemple est toujours éclairant :

```
(split '0 '(() (3)))       ;; (split x (cdr ll))
           (((0) (3)) (() (0 3)))

(split '0 '((1 2) () (3))) ;; (split x ll)
           (((0 1 2) () (3)) ((1 2) (0) (3)) ((1 2) () (0 3)))
```

Partitions V

Le premier élément du résultat est à créer de toutes pièces ; c'est [[(cons (cons x (car ll)) (cdr ll))]]. Le reste s'obtient en remplaçant dans [[(split x (cdr ll))]] (liste de listes de listes) chaque élément [[ss]] (liste de listes) par [[(cons (car ll) ss)]].

```
(define split
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons (cons x (car ll)) (cdr ll))
              (map (lambda (ss) (cons (car ll) ss))
                   (split x (cdr ll)))))))
```

On peut à présent utiliser la fonction `partition` :

```
(partitions '(a b c))  ==>
(((a) (b) (c)) ((a) (b c)) ((a b) (c)) ((b) (a c)) ((a b c)))
```

Réduire le cas d'une liste non vide `l` au cas de `(cdr l)` est l'essentiel du travail d'application du schéma de récursion, mais résoudre le cas de la liste vide est tout aussi important.

Inversion de fonction I

On résout ici le problème de l'inversion d'une fonction réelle de variables réelles. Inverser la fonction

$$f : \mathbb{R}^3 \to \mathbb{R} : (x_1, x_2, x_3) \mapsto f(x_1, x_2, x_3)$$

par rapport à x_2 consiste à construire une fonction

$$g : \mathbb{R}^3 \to \mathbb{R} : (x_1, u, x_3) \mapsto g(x_1, u, x_3)$$

telle que

$$g(x_1, f(x_1, x_2, x_3), x_3) = x_2 \quad \text{et} \quad f(x_1, g(x_1, u, x_3), x_3) = u\,,$$

ou encore

$$g(x_1, u, x_3) = x_2 \quad \text{et} \quad f(x_1, x_2, x_3) = u\,,$$

pour toutes valeurs adéquates de x_2 et u. Le problème est mathématiquement difficile, puisque l'inverse n'existe pas toujours, mais devient plus simple dans le cas où la fonction à inverser est continue et strictement monotone (croissante ou décroissante) par rapport à l'argument (ici x_2) sur lequel porte l'inversion.

Inversion de fonction II

On ramène le cas général (n arguments, $p$ième argument) au cas particulier d'un premier argument sur lequel porte l'inversion, et d'une liste d'autres arguments. Des opérateurs comme `in` et `out` font la conversion, ici dans le cas $n = 3$, $p = 2$:

```
(define in
  (lambda (f)
    (lambda (x l) (f (car l) x (cadr l)))))

(define out
  (lambda (h)
    (lambda (x1 x2 x3) (h x2 (list x1 x3)))))

(define f (lambda (a b c) (+ a (* b c))))

(f 34 56 23)              1322
((in f) 56 '(34 23))      1322
((out (in f)) 34 56 23)   1322
```


Inversion de fonction III

Fonction $(x, \ell) \mapsto f(x, \ell)$, fonction inverse $(u, \ell) \mapsto g(u, \ell)$
Les deux fonctions sont croissantes en leur premier argument.

Approximations successives : soit $(x_0, x_1, \ldots) \longrightarrow g(u, \ell)$
Si $f(x_n, \ell) > u$, alors $x_n > g(u, \ell)$ et on choisit $x_{n+1} < x_n$;
si $u > f(x_n, \ell)$, on choisit $x_{n+1} > x_n$.

Deux variantes : $\mathbb{R} \to \mathbb{R}$ et $\mathbb{R}^+ \to \mathbb{R}^+$. On passe de l'une à l'autre par transformation exponentielle ou logarithmique. La seconde variante est développée ici.

Technique de la bissection. On maintient un intervalle $[m, M]$ dans lequel la valeur recherchée se trouve, on le rétrécit à chaque itération. Au départ, l'intervalle est très grand, par exemple $m = 10^{-6}$ et $M = 10^7$. A chaque étape, on calcule la moyenne (géométrique) μ des bornes de l'intervalle puis la valeur $f(\mu, \ell)$. En fonction de cette valeur, on décide de s'arrêter, si l'écart entre $f(\mu, \ell)$ et u n'excède pas une certaine quantité ε_y ou de continuer soit avec l'intervalle $[m, \mu]$, soit avec l'intervalle $[\mu, M]$. Chaque étape a pour effet de réduire la longueur de l'intervalle (de moitié, pour la première variante) et on peut s'arrêter dès que cette longueur devient moindre qu'une certaine quantité ε_x.

Inversion de fonction IV

Variables globales, fonctions auxiliaires :

```
(define *min* 1.e-6)            (define *eps-x* 1.e-12)
(define *max* 1.e+7)            (define *eps-y* 1.e-12)
(define mu (lambda (a b) (sqrt (* a b))))
(define prox (lambda (a b eps) (< (abs (- a b)) eps)))

(define inv+    ;; Cas où la fonction f est croissante
  (lambda (f)
    (lambda (u l)
      (i+ f u l *min* *max* *eps-x* *eps-y*))))

(define i+
  (lambda (f u l x0 x1 epsx epsy)
    (cond ((prox x0 x1 epsx) (mu x0 x1))
          ((prox (f (mu x0 x1) l) u epsy) (mu x0 x1))
          ((< (f (mu x0 x1) l) u) (i+ f u l (mu x0 x1) x1 epsx epsy))
          (else (i+ f u l x0 (mu x0 x1) epsx epsy)))))
```


Inversion de fonction V

L'opérateur `inv+` prend comme argument une fonction `f` croissante en son premier argument et renvoie la fonction inverse.

Le prédicat `prox` prend comme arguments deux réels a et b et un réel positif ε ; il renvoie vrai si $|a-b| < \varepsilon$.

Fonction `i+`

> *Si la fonction $f : (\mathbb{R} \times \mathbb{R}^k) \to \mathbb{R}$ est croissante en son premier argument et si l'unique solution x de l'équation $f(x,\ell) = u$ appartient à l'intervalle $[x_0 : x_1]$, alors $i^+(f,u,\ell,x_0,x_1,\varepsilon_x,\varepsilon_y)$ est un nombre x' proche de x, au sens que x' ne s'écarte pas de x de plus de ε_x ou que $f(x',\ell)$ ne s'écarte pas de u de plus de ε_y.*

On définit de manière analogue un opérateur `inv-` pour inverser les fonctions décroissantes, qui fera appel à l'opérateur auxiliaire `i-` ; ce dernier ne diffère de `i+` que par la permutation des comparateurs `<` et `>` dans les conditions des clauses contenant les appels récursifs.

Inversion de fonction VI

Un dernier point intéressant consiste à modifier `i+` de manière à éviter l'évaluation multiple des expressions `(mu x0 x1)` et `(f (mu x0 x1) l)`.

```
(define i+       ;; avec évaluation multiple
  (lambda (f u l x0 x1 epsx epsy)
    ((lambda (aux1)
       (cond ((prox x0 x1 epsx) aux1)
             ((prox (f aux1 l) u epsy) aux1)
             ((< (f aux1 l) u) (i+ f u l aux1 x1 epsx epsy))
             (else (i+ f u l x0 aux1 epsx epsy))))
     (mu x0 x1))))

(define i+       ;; sans évaluation multiple de aux1
  (lambda (f u l x0 x1 epsx epsy)
    (let ((aux1 (mu x0 x1)))
       (cond ((prox x0 x1 epsx) aux1)
             ((prox (f aux1 l) u epsy) aux1)
             ((< (f aux1 l) u) (i+ f u l aux1 x1 epsx epsy))
             (else (i+ f u l x0 aux1 epsx epsy))))))
```


Inversion de fonction VII

On peut réutiliser la même technique pour éviter la double évaluation de l'expression `(f aux1 l)` ; on obtient ainsi, sans utiliser `let` :

```
(define i+       ;; sans évaluation multiple de aux1 et aux2
  (lambda (f u l x0 x1 epsx epsy)
    ((lambda (aux1)
       ((lambda (aux2)
          (cond ((prox x0 x1 epsx) aux1)
                ((prox aux2 u epsy) aux1)
                ((< aux2 u) (i+ f u l aux1 x1 epsx epsy))
                (else (i+ f u l x0 aux1 epsx epsy))))
        (f aux1 l)))
     (mu x0 x1))))
```