

macro = une fonction qui prend des fragments de code comme arguments et renvoie un nouveau fragment de code.

Les macros sont exécutées lors de la compilation.

Les macros sont généralement utilisées pour créer de nouvelles structures de contrôle, ou pour forcer une copie *inline* et éviter le coût d'un appel de fonction

```
#define MIN(x,y) (x<y)?x:y
int exp_min (exp *a, exp *b)
{ return MIN (exp_size (a), exp_size (b)); }

int exp_min (exp *a, exp *b)
{ return (exp_size (a) < exp_size (b))
        ? exp_size (a) : exp_size (b); }
```

Macro problèmes

L'expansion textuelle peut considérablement accroître la taille des programmes

Il est préférable que l'expansion des macros se termine

La substitution textuelle des paramètres cause un mélange de portée dynamique et de passage d'arguments par nom

```
#define swap(x,y) { int t = y; y = x; x = t; }  
...  
void proc (int a, int b, int t, int t2)  
{ swap(a,b); /* {int t = b; b = a; a = t; } */  
  swap(t,t2); /* {int t = t2; t2 = t; t = t; } */  
  ...  
}
```

Macro problèmes textuels

Certains types de macros manipulent le texte

La catégorie syntactique peut alors être autre que prévue

Ceci, à l'appel et dans les paramètres

```
#define haha(x) {x
#define abs(x) (x<0) ? -x : x
#define swap(x,y) { int t = y; y = x; x = t; }
...
abs(a+1)*2    =>    (a+1<0) ? -a+1 : a+1*2
...
if (a < b)    if (a < b)
    swap(a,b); =>    { int t = b; b = a; a = t; };
else          else
```

Macros en Scheme

Contrairement à C où les macros opèrent sur le texte, en Scheme elles opèrent sur l'arbre de syntaxe

```
(define-macro myand (lambda (x y) (list 'if x y #f)))
```

Une macro en Scheme peut *analyser* ses arguments

```
(define-macro (myand x y)
  (if (and (cons? x) (eq? (car x) 'myand))
      (list 'myand (cadr x) (list 'myand (caddr x) y))
      (list 'if x y #f)))
```

Un exemple de macro

`(dotimes (x e1) e2)`

Exécute e_2 pour chaque valeur de x de 0 à e_1 :

`(define-macro (dotimes xl b)`

`(let ((x (car xl)) (l (cadr xl)))`

`(list 'letrec (list (list 'loop (list 'lambda (list x)`

`(list 'if (list '≤ x l)`

`(list 'begin b (list 'loop (list '+ x 1))))`

`(list 'loop 0))))))`

Backquotes

Scheme offre une syntaxe spéciale pour construire du code.

```
(define-macro (dotimes xl b)
  (let ((x (car xl)) (l (cadr xl)))
    \letrec ((loop (lambda (,x)
                     (if (<= ,x ,l)
                         (begin ,b (loop (+ ,x 1)))))))
      (loop 0))))
```

Passage par nom

La borne est ré-évaluée à chaque fois, comme dans de l'appel par nom

Nous voulons de l'appel par valeur

```
(define-macro (dotimes xl b)
  (let ((x (car xl)) (l (cadr xl)))
    `(let ((end ,l)
          (letrec ((loop (lambda (,x)
                          (if (<= ,x end)
                              (begin ,b (loop (+ ,x 1)))))))
            (loop 0))))))
```

Capture de nom

capture de nom = quand une manipulation du code change l'interprétation d'un identificateur, qui fait alors référence à une autre variable

```
(let ((start "-- ")
      (end " --"))
  (dotimes (y 10)
    (print start y end)))

⇒

(let ((start "-- ")
      (end " --"))
  (letrec ((loop (lambda (y)
                   (if (<= y end)
                       (begin (print start y end)
                              (loop (+ y 1)))))))
    (loop 0))))
```

Utilise des identificateurs tous frais pour éviter la capture de noms

```
(define-macro (dotimes xl b)
  (let ((x (car xl)) (l (cadr xl))
        (endsym (gensym)) (loopsym (gensym)))
    `(let ((,endsym ,l))
      (letrec ((,loopsym (lambda (,x)
                          (if (<= ,x ,endsym)
                              (begin ,b (,loopsym (+ ,x 1))))))
        (,loopsym 0))))))
```

Fonctions d'ordre supérieur

Utilise la flexibilité des fonctions d'ordre supérieur pour contrôler l'environnement dans lequel le code est exécuté

```
(dotimes (<x> <l>) <b>)
```

devient

```
(let ((body (lambda (<x>) <b>)))  
  (end <l>))  
(letrec ((loop (lambda (i)  
                  (if (<= i end)  
                      (begin (body i) (loop (+ i 1))))))  
  (loop 0)))
```

Fonctions d'ordre supérieur (suite)

```
(define-macro (dotimes xl b)  
  (let ((x (car xl)) (l (cadr xl)))  
    `(let ((body (lambda (,x) ,b)) (end ,l))  
      (letrec ((loop (lambda (i)  
                    (if ( $\leq$  i end)  
                        (begin (body i) (loop (+ i 1)))))))  
        (loop 0))))))
```