Macros

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.

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)); }
```

Macros

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.

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; }
```

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
```

Emacs Lisp (Elisp) primer

$\lambda x \to e$	$(\texttt{lambda}\ (x)\ e)$
$e_1\ e_2$	$(e_1\ e_2)$
$\texttt{if}\ e\ \texttt{then}\ e_t\ \texttt{else}\ e_f$	$(\texttt{if}\ e\ e_t\ e_f)$
(e_1, e_2) et $e_1 : e_2$	$(\texttt{cons}\ e_1\ e_2)$
fst x et *head* x	$(\texttt{car}\ x)$
snd x et *tail* x	$(\texttt{cdr}\ x)$
$\texttt{let}\ x_1 = e_1$	$(\texttt{let}\ ((x_1\ e_1)$
$\quad x_2 = e_2$	$\quad\quad (x_2\ e_2))$
$\texttt{in}\ e$	$\quad e)$

Macros en Lisp

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

```
(defmacro myand (x y) (list 'if x y nil))
```

Une macro en Elisp peut *analyser* ses arguments

```
(defmacro myand (x y)
  (if (and (consp x) (eq (car x) 'myand))
      (list 'myand (cadr x) (list 'myand (caddr x) y))
      (list 'if x y nil)))
```

Un exemple de macro

$$(\textit{dotimes}\ (x\ e_1)\ e_2)$$

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

```
(defmacro 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 'progn b (list 'funcall 'loop
                                                             (list '+ x 1)))))))
          (list 'funcall 'loop 0))))
```

Backquotes

Lisp offre une syntaxe spéciale pour construire du code.

```
(defmacro dotimes (xl b)
  (let ((x (car xl)) (l (cadr xl)))
    `(letrec ((loop (lambda (,x)
                      (if (≤ ,x ,l)
                          (progn ,b (funcall loop (+ ,x 1)))))))
       (funcall 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

```
(defmacro dotimes (xl b)
  (let ((x (car xl)) (l (cadr xl)))
    `(let ((end ,l))
       (letrec ((loop (lambda (,x)
                        (if (≤ ,x end)
                            (progn ,b (funcall loop (+ ,x 1)))))))
         (funcall 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)
    (message
     "%s%d%s"
     start y end)))
```

⇒

```
(let ((start "--  ") (end "  --"))
  (let ((end 10))
    (letrec
        ((loop (lambda (y)
                 (when (≤ y end)
                   (message "%s%d%s"
                            start y end)
                   (funcall loop (+ y 1))))))
      (funcall loop 0))))
```

Gensym

Utilise des identificateurs tout *frais* pour éviter la capture de noms

```
(defmacro dotimes (xl b)
  (let ((x (car xl)) (l (cadr xl))
        (endsym (gensym)) (loopsym (gensym)))
    `(let ((,endsym ,l))
       (letrec ((,loopsym
                 (lambda (,x)
                   (if (≤ ,x ,endsym)
                       (progn ,b (funcall ,loopsym (+ ,x 1)))))))
         (funcall ,loopsym 0)))))
```

Fonctions d'ordre supérieur

Déplacer le code pour contrôler la portée

```
(dotimes (⟨x⟩ ⟨l⟩) ⟨b⟩)
```

devient

```
(let ((body (lambda (⟨x⟩) ⟨b⟩)))
     (end ⟨l⟩)))
  (letrec ((loop (lambda (i)
                   (when (≤ i end)
                     (funcall body i)
                     (funcall loop (+ i 1))))))
    (funcall loop 0)))
```

Fonctions d'ordre supérieur (suite)

```
(defmacro dotimes (xl b)
  (let ((x (car xl)) (l (cadr xl)))
    `(let ((body (lambda (,x) ,b)) (end ,l))
       (letrec ((loop (lambda (i)
                        (when (≤ i end)
                          (funcall body i)
                          (funcall loop (+ i 1))))))
         (funcall loop 0)))))
```

Fonctions d'ordre supérieur (fin)

```
(defmacro dotimes (xl b)
  (let ((x (car xl)) (l (cadr xl)))
    `(dotimes-f (lambda (,x) ,b) ,l)))
(defun dotimes-f (body end)
  (letrec ((loop (lambda (i)
                   (when (≤ i end)
                     (funcall body i)
                     (funcall loop (+ i 1))))))
    (funcall loop 0))))
```

Haskell

La paresse de Haskell lui permet de faire une partie de ce que font les macros

```
if3 x e1 e2 e3
   = case x of { 0 => e1; 1 => e2; 2 => e3 }
```

De plus, l'usage de monades pour les effets de bord offre une opportunité supplémentaire

```
while ct ce = loop
   where loop = do
     t <- ct
     if t
       then do { ce; loop }
       else return ()
```