Exhibit A

```
if (tree) {
  for (iter it = new_iterator (tree, beg, end);
       not_finished (it);
       go_to_next (it))
    {
      tree_node x = get_node (it);
      ... fait qqch ...
    }
}
```

Code répété à plusieurs endroits du programme

Abstraction idéale

```
foreach x in tree from beg to end {
   ... fait qqch ...
}
```

Concis: évite les erreurs

Abstrait: disparition des détails d'utilisation de l'itérateur

Syntaxe spécifique, plus élégante

Plus facile à lire/comprendre

Abstraction par fonction, 1^{er} essai

```
foreach_node_in_tree
   (x, tree, beg, end,
    ... fait qqch ...)
```

Abstraction par fonction, 1^{er} essai

```
foreach_node_in_tree
   (x, tree, beg, end,
    ... fait qqch ...)
```

Erreur à la compilation: ‘x‘ n’existe pas

Erreur à l’exécution:

`... fait qqch ...` est exécuté exactement une fois

Abstraction par fonction, 2^e essai

```
foreach_node_in_tree
  (tree, beg, end,
   λ (tree_node x) { ... fait qqch ... })
```

Ça marche, cette fois

Concis et abstrait!

Syntaxe non spécifique, moins élégante

- Indentation de `...  fait qqch ...`
- éléments “superflus” comme λ, les parenthèses, le type
- Ordre de `x` et `t, beg, end`

Solutions

Utiliser des fonctions

- Code moins élégant
- Potentiellement aussi moins efficace

Étendre le langage

- (Très) coûteux; impossible en général

Métaprogrammation

- Permettre d'introduire de nouvelles syntaxes "localement"
- Augmente la complexité du langage
- Peut rendre le code incompréhensible

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.

But: créer de nouvelles structures de contrôle, rendre le code plus concis, ou 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.

But: créer de nouvelles structures de contrôle, rendre le code plus concis, ou 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); }
```

Iteration avec macros en C

Avec les macros C, on peut obtenir:

```
TREE_FOREACH (x, tree, start, end) {
  ... fait qqch ...
}
```

Toujours pas idéal, mais sans λ (et sans macros), ça ressemblerait à:

```
tree_foreach (tree, start, end, myfun, <qqch>)
```

Avec

```
void myfun (tree_node x, void *args) {
  ... fait qqch (utilisant args) ...
}
```

Expansion des macros

L'expansion des macros peut se faire, sur le texte, les lexèmes, l'ASA, ...

A lieu pendant la compilation: avant de savoir si le code sera exécuté

Expansion et exécution: une autre machine, un autre siècle

Expansion n'a pas accès aux infos de l'exécution (le futur)

Exécution n'a pas toujours accès aux infos de l'expansion (le passé)

Macro problèmes

L'expansion peut considérablement accroître la taille des programmes

Il est préférable que l'expansion des macros se termine

La substitution des paramètres donne une sémantique similaire à:

- portée dynamique: conflits de noms
- passage d'arguments par nom: évaluation excessive

```
#define swap(x,y) { int t = y; y = x; x = t; }
```

Macro problèmes

L'expansion peut considérablement accroître la taille des programmes

Il est préférable que l'expansion des macros se termine

La substitution des paramètres donne une sémantique similaire à:

- portée dynamique: conflits de noms
- passage d'arguments par nom: évaluation excessive

```
#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; } */
  swap(a,b[x++]); /* {int t = b[x++]; b[x++] = a; a =
```

Blame et abstraction

Dans

```
#define swap(x,y) { int t = y; y = x; x = t; }
...
   swap(t,t2);
   swap(a,b[x++]);
   ...
```

À qui la faute?

Pourquoi?

Macro problèmes textuels

Certains types de macros manipulent le texte (e.g. M4, CPP, ...)

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

$$(\texttt{defmacro}\ \textit{et}\ (x\ y)\ (\textit{list}\ '\texttt{if}\ x\ y\ \textit{nil}))$$

Dit de transformer les appels de la forme

$$(\textit{et}\ E_1\ E_2) \implies (\texttt{if}\ E_1\ E_2\ \textit{nil})$$

$(\texttt{if}\ E_1\ E_2\ \textit{nil})$ est à la fois un morceau de code et une liste

Homoiconicité: même syntaxe pour le code et les données

Expansion de `et`

$$(\texttt{defmacro}\ \textit{et}\ (x\ y)\ (\textit{list}\ '\texttt{if}\ x\ y\ \textit{nil}))$$

$$\ldots\ (\textit{et}\ (<\ 0\ x)\ (<\ x\ 10))\ \ldots$$

Le compilateur “appelle” `et`:

$x \mapsto (<\ 0\ x)$

$y \mapsto (<\ x\ 10)$

il faut évaluer:

$(\textit{list}\ '\texttt{if}\ x\ y\ \textit{nil})$

$'\texttt{if} \rightsquigarrow \texttt{if}$ $\quad x \rightsquigarrow (<\ 0\ x)$ $\quad y \rightsquigarrow (<\ x\ 10)$ $\quad \textit{nil} \rightsquigarrow \textit{nil}$

$$(\textit{list}\ldots) \rightsquigarrow (\texttt{if}\ (<\ 0\ x)\ (<\ x\ 10)\ \textit{nil})$$

Analyse de code

Une macro en Elisp peut *analyser* ses arguments

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

Dit de transformer les appels de la forme

$$(\mathit{et}\ (\mathit{et}\ E_1\ E_2)\ E_3) \implies (\mathit{et}\ E_1\ (\mathit{et}\ E_2\ E_3))$$

Note: $(\mathit{car}\ x) \simeq (\mathit{nth}\ 0\ x)$ $(\mathit{cadr}\ x) \simeq (\mathit{nth}\ 1\ x)$ $(\mathit{caddr}\ x) \simeq (\mathit{nth}\ 2\ x)$

Applications des macros en ELisp

Les boucles `dotimes`, `dolist`, `loop`, `named-let`, ...

La définition de macros avec `defmacro`

La notion de *l-value*, avec `setf`, `push`, `incr`, ...

La définition de types avec `defstruct`, `defclass`

Les (multi)méthodes avec `defgeneric`, `defmethod`

Le filtrage avec `pcase`

Génération d'analyseur de syntaxe avec `define-peg-rule`, ...

En/dépaquetage de données binaires avec Bindat

Une boucle en ELisp

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

```
(letrec ((loop (lambda (x)
                 (if (< x e1)
                     (progn e2 (funcall loop (+ x 1)))))))
  (funcall loop 0))
```

On aimerait pouvoir écrire juste:

```
(dotimes (x e1) e2)
```

Une définition laborieuse de `dotimes`

```
(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 des structures partiellement constantes, avec des sortes de patrons:

- $'E$ renvoie E tel quel
- $`E$ renvoie E en y remplaçant les “unquote”
- $,E$ sera remplacé par la valeur de E

$`$ se prononce “backquote” ou “quasiquote” et $,$ se prononce “unquote”

Par exemple:

$$`(1\ (+\ 2\ 2)\ ,(+\ 3\ 4)\ '5) \Rightarrow (1\ (+\ 2\ 2)\ 7\ '5)$$
$$'(1\ 2\ ,(+\ 3\ 4)\ `5) \Rightarrow (1\ 2\ ,(+\ 3\ 4)\ `5)$$

`dotimes` avec backquote

On peut donc avantageusement redéfinir *dotimes* avec le backquote:

```
(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))))
```

Expansion de `dotimes`

$$\ldots\ (\textit{dotimes}\ (x\ 10)\ (\textit{message}\ \text{"}\%S\text{"}\ (+\ x\ l)))\ \ldots$$

Le compilateur “appelle” `dotimes`, il faut évaluer:

$$xl \mapsto (x\ 10)$$

$$b \mapsto (\textit{message}\ \text{"}\%S\text{"}\ (+\ x\ l$$

```
(let ((x (car xl)) (l (cadr xl)))
  `(letrec ((loop (lambda (,x)
                    (if (<= ,x ,l)
                        (progn ,b (funcall loop (+ ,x 1)))))))
     (funcall loop 0)))
```

Expansion de `dotimes`

$$\ldots\ (\textit{dotimes}\ (x\ 10)\ (\textit{message}\ \text{"}\%S\text{"}\ (+\ x\ l)))\ \ldots$$

Le compilateur “appelle” `dotimes`, il faut évaluer:

$$xl \mapsto (x\ 10)$$
$$b \mapsto (\textit{message}\ \text{"}\%S\text{"}\ (+\ x\ l$$
$$x \mapsto x$$
$$l \mapsto 10$$

```
`(letrec ((loop (lambda (,x)
                  (if (<= ,x ,l)
                      (progn ,b (funcall loop (+ ,x 1)))))))
   (funcall loop 0))
```

Expansion de `dotimes`

$$\ldots\ (\textit{dotimes}\ (x\ 10)\ (\textit{message}\ \text{"}\%S\text{"}\ (+\ x\ l)))\ \ldots$$

Le compilateur “appelle” `dotimes`, qui renvoie:

```
(letrec ((loop (lambda (x)
                 (if (<= x 10)
                     (progn (message "%S" (+ x l))
                            (funcall loop (+ x 1)))))))
  (funcall loop 0))
```

Ré-évaluation excessive

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

On voudrait plutôt la calculer une fois pour toute:

```
(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 ()
```