Langages Enchâssés

Enseignant : Robert Strandh

Planning : 8x3h de cours/TD/TP + un examen

Horaires : mercredi 13h50-16h40

Lieu : Salle Amphi D (semaines impaires) ou I011 (semaines paires)

Web : `http://dept-info.labri.fr/~strandh/`
puis suivre lien

Support de cours

Robert Strandh et Irène Durand :
Langages Enchâssés (polycopié)

Robert Strandh et Irène Durand :
Traité de programmation en Common Lisp

Paul Graham :
On Lisp : Advanced Techniques for Common Lisp

Transparents

Bibliographie

Paul Graham : ANSI Common Lisp
Prentice Hall

Sonya Keene : Object-Oriented Programming in Common Lisp
A programmer's guide to CLOS
Addison Wesley

Autres documents

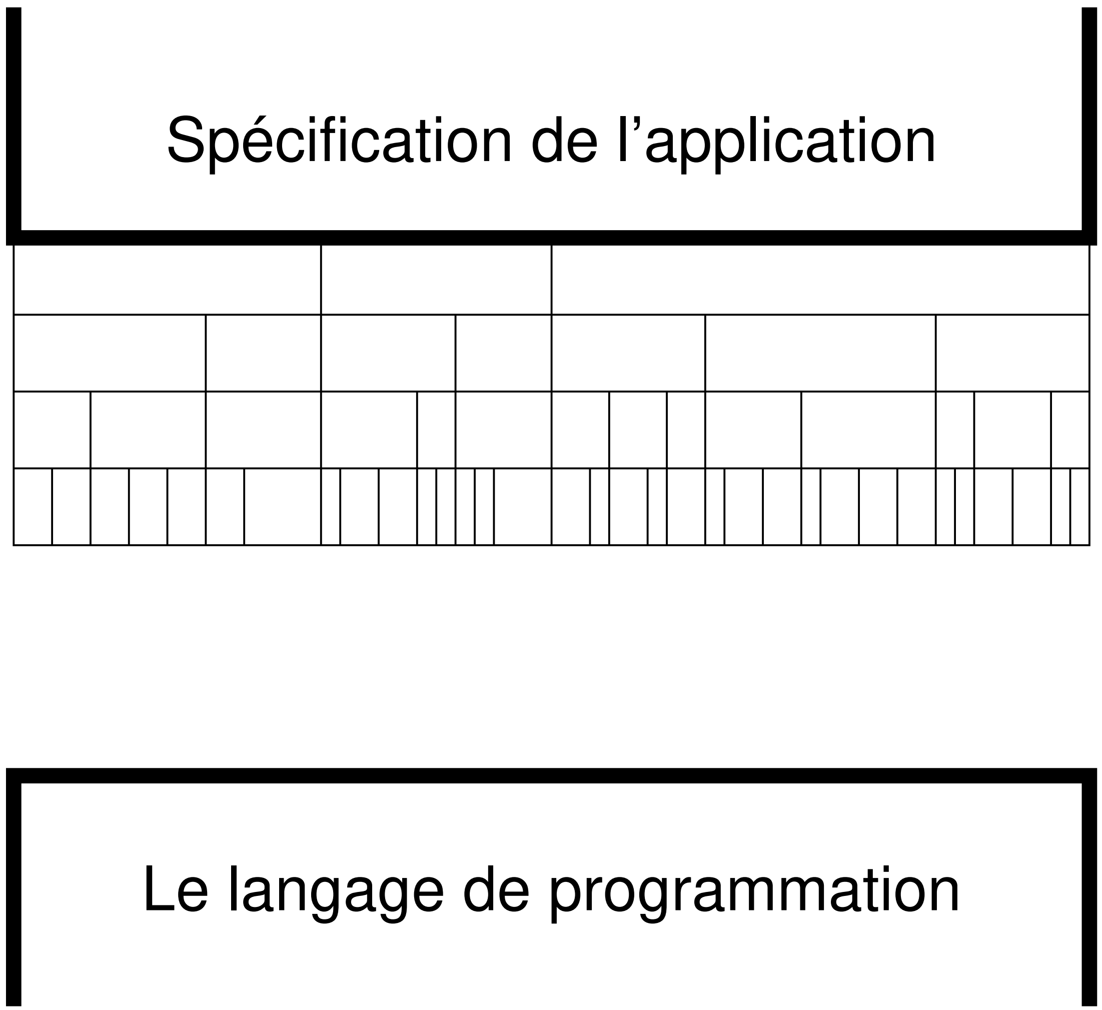
Guy Steele :	Common Lisp, the Language, second edition Digital Press (disponible sur WWW en HTML)
David Lamkins :	Successful Lisp (Tutorial en-ligne)

CMUCL user's guide

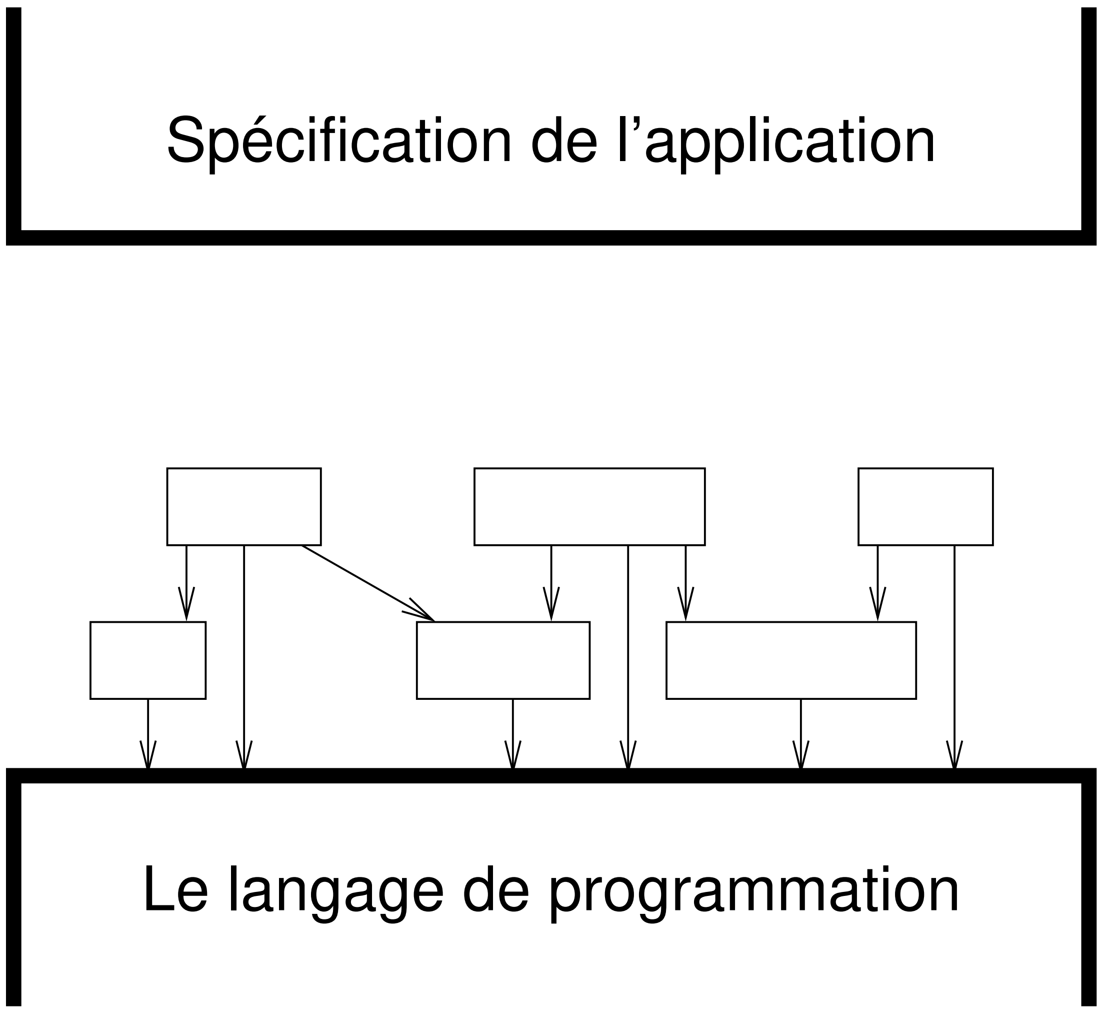
CLX reference manual

The HyperSpec (la norme ANSI complète de Common Lisp, en HTML)

Programmation descendante



Programmation ascendante



Langage dédié

C'est un langage spécifique à un domaine d'application.

Avantages par rapport à un langage général :

- moins de code à écrire
- moins de code à maintenir
- code plus facile à maintenir grace aux idiomes explicites
- potentiellement plus efficace

Langage dédié

Par rapport à un langage dédié, une bibloithèque :

- ne permet pas de nouvelle syntaxe
- ne permet pas aux compilateur de véifier le respect des idiomes
- est souvent moins efficace

Exemple : OpenGL

Implémentation traditionnelle

L'implémentation d'un langage dédié se fait traditionnellement d'une des manières suivantes :

- préprocesseur,
- interpréteur,
- compilateur.

Implémentation traditionnelle

Difficultés :

- préprocesseur : ordre et compatibilité entre plusieurs pré-processeurs,
- interpréteur : performance + réimplémentation de fonctions de base,
- compilateur : difficulté d'implémentation + réimplémentation de fonctions de base.

Définition de Langage Enchâssé

C'est une technique du Génie Logiciel pour implémenter des langages dédiés.

L'idée est d'augmenter un langage existant (le langage hôte) avec les fonctionnalités nécessaires pour le langage dédié.

Le langage dédié sera donc enchâssé dans le langage hôte.

Abstraction

Il y a trois mécanismes d'abstraction nécessaires :

- abstraction de données (structures, classes),
- abstraction de contrôle (procédures, méthodes),
- abstraction syntaxique (formes spéciales).

Les langages habituels ne sont pas capables d'exprimer l'abstraction syntaxique.

Pourtant elle est nécessaire pour la construction d'un langage enchâssé.

Objectif du module

Présentation des techniques d'implémentation des langages enchâssés grâce aux macros Common Lisp.

Avantage :

- imbrication de plusieurs langages possible,
- compilation si le langage hôte est compilé,
- importation en masse des fonctions de base du langage hôte.

Contenu du module

Cours :

- les macros Common Lisp,
- présentation d'un langage enchâssé.

TD/TP :

- Emacs pour la programmation en Common Lisp,
- écriture et mise au point de macros.

Standards de codage

Il faut s'habituer aux standards de codage

Pour Common Lisp, suivre les exemples dans la littérature

En particulier pour l'indentation de programmes qui est très standardisée.

Il faut pouvoir comprendre le programme sans regarder les ()

Utiliser lisp-mode de GNU Emacs

L'indentation n'est donc pas une question de goût personnel

Voir site web pour les standards de codagae.

Programmation fonctionnelle

Common Lisp est un langage mixte (fonctionnel, impératif, orienté objets)

Pour un bon résultat, utiliser la programmation fonctionnelle le plus possible

Par programmation fonctionnelle, il faut comprendre la programmation sans effets de bord

Les programmes sont plus faciles à tester

Programmation ascendante (bottom-up)

Paramètres sont souvent des fonctions (fermetures)

Langage enchâssé : prescript

- Comme PostScript mais préfixe au lieu de postfixe,
- Construction de chemin (divisé en sous chemins), puis traçage du chemin,
- Environnement actuel contenant un point actuel, une transformation actuelle, et un chemin actuel.
- Environnements organisés sous la forme d'une pile. Le sommet de la pile est l'environnement actuel.

Langage enchâssé : prescript

Le chemin actuel est exprimé en coordonnés *physiques*.

Les commandes de dessin acceptent des coordonnés *utilisateur*.

La transformation actuelle traduit entre les deux.

Langage enchâssé : prescript

Modification de la transformation actuelle : `translate`, `scale`, `rotate`

Commencer un nouveau sous chemin : `moveto`

Rajouter un segment au chemin actuel : `lineto`, `curveto`, `arcto` (seul `lineto` sera implémenté)

Tracer le chemin actuel : `stroke`

Sauver l'environnement actuel : `gsave`

Exemple de prescript

```
(let ((x 100) (y 200))
  ...
  (translate x y)
  (moveto (* 2 x) y)
  (lineto 430 460)
  (stroke))
  (moveto 0 0)
    (loop for i from 0 to 100 by 10
          do (gsave (lineto i (- 200 i)) (stroke)))
  ...)
```

Le support X11

```
(defparameter *dpy* nil)
(defparameter *win* nil)
(defparameter *gctxt* nil)

(defun get-environment-variable (string)
  (cdr (assoc string ext:*environment-list* :test #'string=)))

(defun parse-display-variable (s)
  (let* ((colon (position #\: s))
         (dot (position #\. s :start colon))
         (host-name (if (zerop colon) "localhost" (subseq s 0 colon)))
         (dpy-number (parse-integer s :start (1+ colon) :end dot)))
    (values host-name dpynumber)))
```

Le support X11

```
(defun open-window ()
  (multiple-value-bind (host display)
      (parse-display-variable (get-environment-variable "DISPLAY"))
    (let* ((dpy (xlib:open-display host :display display))
           (screen (xlib:display-default-screen dpy))
           (black (xlib:screen-black-pixel screen))
           (white (xlib:screen-white-pixel screen))
           (win (xlib:create-window :parent (xlib:screen-root screen)
                                    :background white
                                    :x 0 :y 0 :width 500 :height 500))
           (gcontext (xlib:create-gcontext :drawable win
                                           :background white
                                           :foreground black)))
```

Le support X11

```
(xlib:map-window win)
(xlib:display-force-output dpy)
(setf *dpy* dpy
      *win* win
      *gctxt* gcontext))))

(defun draw-line (x1 y1 x2 y2)
  (xlib:draw-line *win* *gctxt*
                  (round x1) (round y1) (round x2) (round y2))
  (xlib:display-force-output *dpy*))
```

Point

```
(defclass point ()
  ((x :initarg :x)
   (y :initarg :y)))

(defun make-point (x y)
  (make-instance 'point :x x :y y))
```

Transformation

```
(defclass transformation ()
  ((mxx :initarg :mxx)
   (mxy :initarg :mxy)
   (myx :initarg :myx)
   (myy :initarg :myy)
   (tx :initarg :tx)
   (ty :initarg :ty)))

(defun get-transformation (transformation)
  (with-slots (mxx mxy myx myy tx ty) transformation
    (values mxx mxy myx myy tx ty)))
```

Transformation

```
(defun make-transformation (mxx mxy myx myy tx ty)
  (make-instance 'transformation
    :mxx mxx :mxy mxy :myx myx :myy myy :tx tx :ty ty))

(defun make-translation-transformation (tx ty)
  (make-transformation 1 0 0 1 tx ty))

(defun make-rotation-transformation (angle)
  (let ((radians (* angle #.(/ pi 180))))
    (let ((s (sin radians))
          (c (cos radians)))
      (make-transformation c (- s) s c 0 0))))

(defun make-scaling-transformation (sx sy)
  (make-transformation sx 0 0 sy 0 0))
```

Transformation

```
(defun compose-transformations (t1 t2)
  (multiple-value-bind (mxx1 mxy1 myx1 myy1 tx1 ty1)
      (get-transformation t1)
    (multiple-value-bind (mxx2 mxy2 myx2 myy2 tx2 ty2)
        (get-transformation t2)
      (make-transformation (+ (* mxx1 mxx2) (* mxy1 myx2))
                           (+ (* mxx1 mxy2) (* mxy1 myy2))
                           (+ (* myx1 mxx2) (* myy1 myx2))
                           (+ (* myx1 mxy2) (* myy1 myy2))
                           (+ (* mxx1 tx2) (* mxy1 ty2) tx1)
                           (+ (* myx1 tx2) (* myy1 ty2) ty1)))))
```

Transformation

```
(defun transform-point (point transformation)
  (with-slots (mxx mxy myx myy tx ty) transformation
    (with-slots (x y) point
      (make-point (+ (* mxx x) (* mxy y) tx)
                  (+ (* myx x) (* myy y) ty)))))
```

Chemin

```
;;; A path is a list of subpaths.  Each valid subpath contains
;;; a list of elements (line- , curve-, and arc segments) with the
;;; last element being a point, the origin of the subpath.

;;; create an uninitialized path
(defun new-path ()
  '())

;;; initialize a path with a point
(defun initialize-path (path point)
  (cons (list point) path))
```

Chemin

```
;;; add an element to an initialized path
(defun add-element (path element)
  (assert (car path))
  (cons (cons element (car path)) (cdr path)))

(defgeneric stroke-one (point segment))

(defun stroke-subpath (subpath)
  (reduce #'stroke-one (cdr subpath) :initial-value (car subpath)))

(defun stroke-path (path)
  (mapc #'stroke-subpath (reverse (mapcar #'reverse path))))
```

Segment

```
(defclass lineseg ()
  ((point :initarg :point)))

(defun make-lineseg (point)
  (make-instance 'lineseg :point point))

(defmethod stroke-one (point (seg lineseg))
  (let ((p2 (slot-value seg 'point)))
    (draw-line (slot-value point 'x)
               (slot-value point 'y)
               (slot-value p2 'x)
               (slot-value p2 'y))
    p2))
```

Environnement

```
(defclass env ()
  ((position :initarg :position)
   (transformation :initarg :transformation)
   (path :initform (new-path) :initarg :path)))

(defun make-initial-env ()
  (make-instance 'env
    :position (make-point 0 0)
    :transformation (make-transformation 1 0 0 -1 0 500)))

(defun copy-env (env)
  (with-slots (position transformation path) env
    (make-instance 'env
      :position position :transformation transformation :path path)))
```

Un interprète pour prescript

```
* (prescript-interpreter
    '((moveto 2 4)
      (lineto 20 10)
      (gsave (lineto 34 23) (stroke))
      (lineto 44 55)))

*
```

Un interprète pour prescript

```
(setf (get 'moveto 'prescript)
      #'(lambda (envstack x y)
          (with-slots (position transformation path) (car envstack)
            (let ((p (transform-point (make-point x y)
                                      transformation)))
              (setf position p)
              (setf path (cons (list p) path))))))

(setf (get 'scale 'prescript)
      #'(lambda (envstack sx sy)
          (with-slots (transformation) (car envstack)
            (setf transformation
                  (compose-transformations
                   transformation
                   (make-scaling-transformation sx sy))))))
```

Un interprète pour prescript

```
(setf (get 'translate 'prescript)
      #'(lambda (envstack tx ty)
          (with-slots (transformation) (car envstack)
            (setf transformation
                  (compose-transformations
                   transformation
                   (make-translation-transformation tx ty))))))

(setf (get 'rotate 'prescript)
      #'(lambda (envstack angle)
          (with-slots (transformation) (car envstack)
            (setf transformation
                  (compose-transformations
                   transformation
                   (make-rotation-transformation angle))))))
```

Un interprète pour prescript

```
(setf (get 'newpath 'prescript)
      #'(lambda (envstack)
          (with-slots (path) (car envstack)
            (setf path (new-path)))))

(setf (get 'lineto 'prescript)
      #'(lambda (envstack x y)
          (with-slots (position transformation path) (car envstack)
            (let ((p (transform-point (make-point x y)
                                      transformation)))
              (setf path (add-element path (make-lineseg p)))
              (setf position p)))))

(setf (get 'stroke 'prescript)
      #'(lambda (envstack)
          (with-slots (path) (car envstack)
            (stroke-path path))))
```

Un interprète pour prescript

```
(setf (get 'gsave 'prescript)
      #'(lambda (envstack &rest commands)
          (let ((newenvstack (cons (copy-env (car envstack)) envstack)))
            (loop for command in commands
                  do (apply (get (car command) 'prescript)
                            newenvstack
                            (cdr command))))))

(defun prescript-interpreter (commands)
  (let ((envstack (list (make-initial-env))))
    (loop for command in commands
          do (apply (get (car command) 'prescript)
                    envstack
                    (cdr command)))))
```

Un interprète pour prescript

```
(setf (get 'machin 'prescript)
      #'(lambda (...)
          ...))
```

Un interprète pour prescript

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Additional support

(defmacro defprescript (name arglist &body body)
  `(setf (get ',name 'prescript)
    #'(lambda (envstack ,@arglist)
        (with-slots (position transformation path) (car envstack)
          ,@body))))
```

Un interprète pour prescript

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The interpreter

(defprescript moveto (x y)
  (let ((p (transform-point (make-point x y) transformation)))
    (setf position p)
    (setf path (cons (list p) path))))

(defprescript scale (sx sy)
  (setf transformation
        (compose-transformations
         transformation (make-scaling-transformation sx sy))))
```

Un interprète pour prescript

```
(defprescript translate (tx ty)
  (setf transformation
        (compose-transformations
         transformation (make-translation-transformation tx ty))))

(defprescript rotate (angle)
  (setf transformation
        (compose-transformations
         transformation (make-rotation-transformation angle))))
```

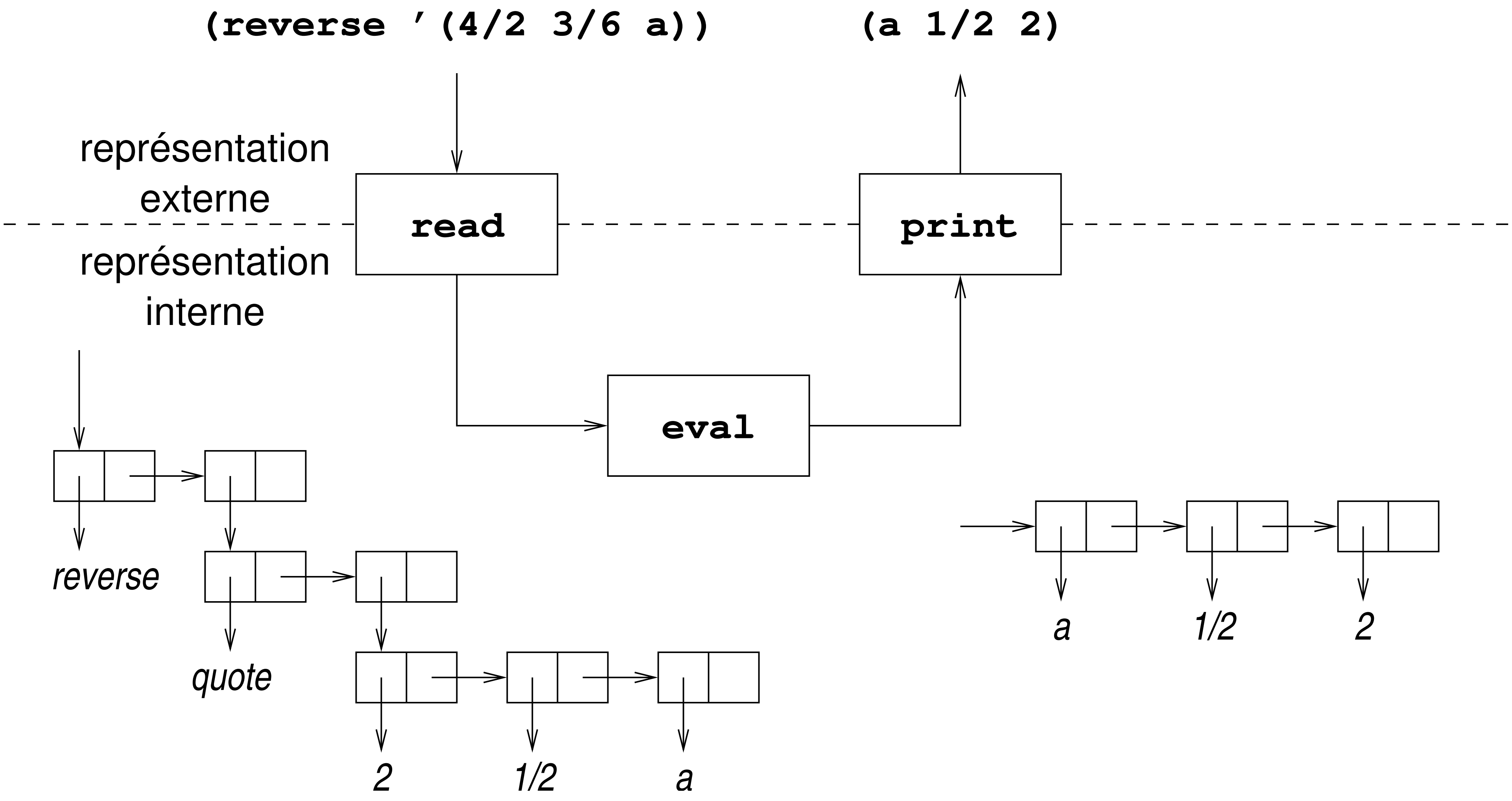
Un interprète pour prescript

```
(defprescript newpath ()
  (setf path (new-path)))

(defprescript lineto (x y)
  (let ((p (transform-point (make-point x y) transformation)))
    (setf path (add-element path (make-lineseg p)))
    (setf position p)))

(defprescript stroke ()
  (stroke-path path))
```

Un interprète pour prescript

```
(defprescript gsave (&rest commands)
  (let ((newenvstack (cons (copy-env (car envstack)) envstack)))
    (loop for command in commands
          do (apply (get (car command) 'prescript)
                    newenvstack
                    (cdr command)))))

(defun prescript-interpreter (commands)
  (let ((envstack (list (make-initial-env))))
    (loop for command in commands
          do (apply (get (car command) 'prescript)
                    envstack
                    (cdr command)))))
```

Évaluation dans Common Lisp



Macros

```
(defmacro nil! (var)
  (list 'setf var nil))
```

Evaluation en deux phases: macro-expansion puis évaluation

Important: la macro-expansion est faite par le compilateur.
L'expression :

(nil! x)

est transformée en l'expression :

(setf x nil)

qui sera compilée à la place de l'expression initiale Les macros permettent donc de programmer le compilateur.

Backquote

```
* (setf a 1 b 2)

2
* `(a is ,a and b is ,b)

(A IS 1 AND B IS 2)
* (defmacro nil! (var)
    `(setf ,var nil))

NIL!
* (nil! *x*)

NIL
*
```

Backquote

```
* (setf lst '(1 2 3))

(1 2 3)
* `(the list is ,lst I think)

(THE LIST IS (1 2 3) I THINK)
* `(the elements are ,@lst I think)

(THE ELEMENTS ARE 1 2 3 I THINK)
*
```

Exemples

```
(defmacro while (test &rest body)
  `(do ()
       ((not ,test))
     ,@body))

(defmacro for (init test update &rest body)
  `(progn
     ,init
     (while ,test
       ,@body
       ,update)))
```

Difficultés

Capture de variables

```
;;; incorrect
(defmacro ntimes (n &rest body)
  `(do ((x 0 (1+ x)))
       ((>= x ,n))
     ,@body))

* (let ((x 10))
    (ntimes 5
      (setf x (1+ x)))
    x)
10
*
```

Difficultés

Multiple évaluation

```
;;; incorrect
(defmacro ntimes (n &rest body)
  (let ((g (gensym)))
    `(do ((,g 0 (1+ ,g)))
         ((>= ,g ,n))
       ,@body)))
```

```
* (let ((v 10))
    (ntimes (setf v (1- v))
            (format t "*")))
*****
NIL
*
```

Difficultés

```
;;; correct
(defmacro ntimes (n &rest body)
  (let ((g (gensym))
        (h (gensym)))
    `(let ((,h ,n))
       (do ((,g 0 (1+ ,g)))
           ((>= ,g ,h))
         ,@body))))
```

Exemple: condlet

Nous souhaitons pouvoir faire quelque chose comme ceci :

```
* (condlet (((= 1 2) (x (princ 'a)) (y (princ 'b)))
            ((= 1 1) (y (princ 'c)) (x (princ 'd)))
            (t (x (princ 'e)) (z (princ 'f))))
    (list x y z))

CD
(D C NIL)
*
```

Exemple: condlet (suite)

Voici une solution possible :

```
(defmacro condlet (clauses &body body)
  (let ((bodfn (gensym))
        (vars (mapcar #'(lambda (v) (cons v (gensym)))
                      (remove-duplicates
                       (mapcar #'car
                               (mappend #'cdr clauses))))))
    `(labels ((,bodfn ,(mapcar #'car vars)
               ,@body))
       (cond ,@(mapcar #'(lambda (cl)
                           (condlet-clause vars cl bodfn))
                      clauses)))))

(defun condlet-clause (vars cl bodfn)
  `(,(car cl) (let ,(mapcar #'cdr vars)
```

```
            (let ,(condlet-binds vars cl)
              (,bodfn ,@(mapcar #'cdr vars))))))

(defun condlet-binds (vars cl)
  (mapcar #'(lambda (bindform)
              (if (consp bindform)
                  (cons (cdr (assoc (car bindform) vars))
                        (cdr bindform))))
          (cdr cl)))
```

Exemple: with-db

L'idiome suivant est fréquent :

```
(let ((temp *db*))
  (setq *db* db)
  (lock *db*)
  (prog1 (eval-query q)
         (release *db*)
         (setq *db* temp)))
```

`On souhaite pouvoir écrire~:`

```
(with-db db
  (eval-query q))
```

Exemple: with-db (suite)

```
(defmacro with-db (db &body body)
  (let ((temp (gensym)))
    `(let ((,temp *db*))
       (unwind-protect
         (progn
           (setq *db* ,db)
           (lock *db*)
           ,@body)
         (progn
           (release *db*)
           (setq *db* ,temp))))))
```

Exemple: with-db (suite)

```
(defmacro with-db (db &body body)
  (let ((gbod (gensym)))
    `(let ((,gbod #'(lambda () ,@body)))
       (declare (dynamic-extent ,gbod))
       (with-db-fn *db* ,db ,gbod))))

(defun with-db-fn (old-db new-db body)
  (unwind-protect
       (progn
         (setq *db* new-db)
         (lock *db*)
         (funcall body))
    (progn
      (release *db*)
      (setq *db* old-db))))
```

Exemple: macros anaphoriques

```
(let ((it (complicated-calculation)))
  (if it
      (do-something-with it)
      (do-something-else-with it)))

(aif (complicated-calculation)
     (do-something-with it)
     (do-something-else-with it))
```

Exemple: macros anaphoriques (suite)

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defmacro awhen (test-form &body body)
  `(aif ,test-form
    (progn ,@body)))

(defmacro awhile (expr &body body)
  `(do ((it ,expr ,expr))
       ((not it))
    ,@body))
```

Exemple: macros anaphoriques (suite)

```
(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))

(defmacro acond (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (sym (gensym)))
        `(let ((,sym ,(car cl1)))
           (if ,sym
               (let ((it ,sym)) ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

Un compilateur pour prescript

```
(defmacro with-transformed-point ((x y p) &body body)
  `(let ((,p (transform-point (make-point ,x ,y) transformation)))
    ,@body))

(defmacro with-current-env (envstack &body body)
  (let ((e (gensym)))
    `(let ((,e (car ,envstack)))
      (with-slots (position transformation path) ,e
        ,@body))))
```

Un compilateur pour prescript

```
(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
       (labels
          ((moveto (x y)
             ...)
           (scale (sx sy)
             ...)
           (translate (tx ty)
             ...)
           ...)
        ,@body
        (values)))))
```

Un compilateur pour prescript

```
(moveto (x y)
  (with-current-env ,envstack
    (with-transformed-point (x y p)
      (setf position p)
      (setf path (initialize-path path p)))))
(scale (sx sy)
  (with-current-env ,envstack
    (setf transformation
          (compose-transformations
           transformation (make-scaling-transformation sx sy)))))
```

Un compilateur pour prescript

```
(translate (tx ty)
  (with-current-env ,envstack
    (setf transformation
          (compose-transformations
           transformation (make-translation-transformation tx ty)))))
(rotate (angle)
  (with-current-env ,envstack
    (setf transformation
          (compose-transformations
           transformation (make-rotation-transformation angle)))))
```

Un compilateur pour prescript

```
(newpath ()
  (with-current-env ,envstack
    (setf path (new-path))))
(lineto (x y)
  (with-current-env ,envstack
    (with-transformed-point (x y p)
      (setf path (add-element path (make-lineseg p)))
      (setf position p))))
(stroke ()
  (with-current-env ,envstack
    (stroke-path path))))
```

Le problème de `gsave`

```
(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
       (labels
          ((moveto (x y)
             ...)
           ...)
         (macrolet ((gsave (&body body)
                          ...))
           ,@body
           (values))))))
```

Le problème de gsave

```
(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
       (labels ...
        (macrolet ((gsave (&body body)
                     `(progn (push (copy-env (car ,',envstack))
                                   ,',envstack)
                      (unwind-protect (progn ,@body)
                        (pop ,',envstack)))))
          ,@body
          (values))))))
```

Un compilateur extensible

```
(defmacro aif (test then &optional else)
  `(let ((it ,test))
     (if it ,then ,else)))

(defparameter *expanders* '())

(defmacro defprescript (name arglist &body body)
  `(aif (assoc ',name *expanders*)
     (setf (cdr it) '(,arglist ,@body))
     (push '(,name ,arglist ,@body) *expanders*)))
```

Un compilateur extensible

```
(defprescript moveto (x y)
  (with-transformed-point (x y p)
    (setf position p)
    (setf path (initialize-path path p))))

(defmacro modify-transformation (arg)
  `(setf transformation
         (compose-transformations
          transformation
          ,arg)))
```

Un compilateur extensible

```
(defprescript scale (sx sy)
  (modify-transformation (make-scaling-transformation sx sy)))

(defprescript translate (tx ty)
  (modify-transformation (make-translation-transformation tx ty)))

(defprescript rotate (angle)
  (modify-transformation (make-rotation-transformation angle)))
```

Un compilateur extensible

```
(defprescript newpath ()
  (setf path (new-path)))

(defprescript lineto (x y)
  (with-transformed-point (x y p)
    (setf path (add-element path (make-lineseg p)))
    (setf position p)))

(defprescript stroke ()
  (stroke-path path))
```

Un compilateur extensible

```
(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
       (labels ,(mapcar #'(lambda (expander)
                            (let ((name (car expander))
                                  (arglist (cadr expander))
                                  (body (cddr expander)))
                              `(,name ,arglist
                                 (with-current-env ,envstack
                                   ,@body))))
                        *expanders*)
         (macrolet ((gsave (&body body)
                      ...))
           ,@body
           (values))))))
```