Langages Enchâssés

Robert Strandh
professeur à l'Université Bordeaux 1

Décembre 2001

Contenu

Chapitre 1

Introduction

On considère une application ayant besoin de dessiner des figures construites d'une façon similaire à celle utilisée par PostScript. L'essence de la façon PostScript consiste à maintenir un environnement graphique contenant une transformation actuelle, un point de dessin actuel et un chemin actuel.

Le dessin se fait en deux phase. D'abord un chemin est construit. Puis le chemin est tracé. Les commandes de construction de chemin sont `moveto`, `lineto`, `arcto` et `curveto`. La commande `moveto` créé un nouveau sous chemin alors que les autres rajoutent un segment au sous chemin actuel.

Le chemin actuel est exprimé avec les coordonnés du système de repères physique (de l'imprimante ou de l'écran), alors que les commandes de construction de chemin sont exprimées avec les coordonnées du système de repères utilisateur actuel. C'est la transformation actuelle qui est chargée de la traduction entre les deux.

Certaines commandes ont pour but de modifier la transformation actuelle. C'est le cas de `translate`, `scale` et `rotate`.

Le traçage du chemin actuel se fait grâce à la commande `stroke`.

Notre petit langage sera appelé *prescript* car la notation sera préfixe alors que la notation de PostScript est postfixe.

Chapitre 2

Définition du petit langage

Voici une spécification assez informelle du langage *prescript*. Notre implémentation ne contiendra pas l'ensemble des commandes de construction de chemin : seul les commandes `moveto` et `lineto` seront implémentées.

Le langage manipule une *pile d'environnements*. Chaque environnement contient un *chemin actuel*, une *transformation actuelle* et un *point actuel*.

2.1 Construction du chemin

Le chemin est initialement vide. En général, il contient un nombre arbitraire de sous chemins. Chaque sous chemin est tracé séparément. Un sous chemin est une liste d'éléments. Le premier élément du sous chemin est un point indiquant le début du chemin. Les autres éléments sont des segments de droites, d'arcs ou de courbes de Bézier (seul des segments de droites seront implémentés). Un segment commence implicitement dans le point de terminaison du segment précédent ou du point initial s'il s'agit du premier segment.

La commande `moveto` commence un nouveau sous chemin, qui sera initialisé avec le point donné en argument de la commande.

La commande `lineto` rajoute un segment de droite à la fin du sous chemin actuel. L'argument à la commande est le point de fin du segment, le point du début étant le point actuel avant l'exécution de la commande.

2.2 Traçage du chemin

Le chemin actuel est tracé grâce à la commande `stroke`. Cette commande ne prend pas d'arguments.

2.3 La transformation actuelle

Le chemin actuel contient des coordonnés physiques. Les commandes de construction du chemin prennent comme arguments des coordonnés dans un système de repères différent et modifiable par l'utilisateur. C'est la transformation actuelle qui est chargé de la traduction entre les deux systèmes de repères.

La transformation actuelle est une transformation *affine* permettant une combinaison arbitraire de rotations, translations et modifications d'échelle. Elle est exprimée par une matrice agissant sur les coordonnés acceptés par les commandes de construction de chemin.

La transformation actuelle peut être modifiée grâce aux commandes `rotate` (rotation du système actuel d'un angle donné), `scale` (transformation de l'échelle du système actuel de facteurs donnés) et `translate` (translation du système actuel d'un vecteur donné).

Une modification de la transformation actuelle influence les commandes de construction de chemin exécutées *après* la modification uniquement.

2.4 Modification de la pile d'environnements

La pile d'environnements peut être manipulée grâce à la commande `gsave`. Cette commande accepte un nombre arbitraire d'autres commandes. Ces commandes seront exécutées avec une pile modifiée. En fait, `gsave` duplique le sommet de la pile, exécute les commandes, puis dépile le sommet. De cette manière, la transformation actuelle, le chemin actuel et le point actuel peuvent être localement modifiés sans effet global.

2.5 Exemples

Voici quelques exemples d'utilisation du petit langage *prescript*.

Le premier exemple n'utilise que des arguments constants. L'interprète développé plus tard sera capable de traiter ce type de programmes :

```
...
(translate 100 20)
(moveto 0 0)
(gsave (lineto 0 200) (stroke))
(lineto 100 100)
(stroke)
...
```

L'interprète aura déjà du mal à interpréter des programmes contenant des expressions arithmétiques comme celle-ci :

```
...
(translate 100 20)
(moveto 0 0)
(gsave (lineto 0 200) (stroke))
(lineto 100 (+ 50 50))
(stroke)
...
```

En utilisant `eval` dans l'interprète, cette situation simple peut être réglée, mais il sera quand-même incapable de traiter des programmes contenant des références à des variables locales comme dans ce programme :

```
(let ((x 10) (y 20))
  ...
  (translate 100 y)
  (moveto 0 0)
  (gsave (lineto 0 200) (stroke))
  (lineto 100 (* x 10))
  (stroke))
```

Dans le programme précédent, le petit langage permet non seulement l'intégration d'expression Common Lisp dans des arguments de fonctions, mais le petit langage s'intègre aussi dans des expressions arbitraires Common Lisp.

L'exemple final montre la possibilité de mélanger librement plusieurs petits langages, ici *prescript* et `loop` :

```
...
(translate 100 100)
(moveto 400 400)
(lineto 430 460)
(stroke))
(moveto 0 0)
  (loop for i from 0 to 100 by 10
        do (gsave (lineto i (- 200 i)) (stroke)))
...
```

Chapitre 3

Le support commun

Afin de pouvoir manipuler les objets nécessaires pour notre petit langage, nous avons besoin d'un support commun. Le voici :

3.1 Le support X11

Ce code permet l'ouverture d'une fenêtre X11 pour que le résultat de la commande `stroke` puisse être vérifié.

Le code est exprimé en utilisant la bibliothèque CLX, qui est une implémentation en Common Lisp du protocole X11. Elle est donc au même niveau d'abstraction que la Xlib pour le langage C.

Remarque : CLX n'est pas une interface à la Xlib. Le protocole X11 est déjà bien adapté au langage Common Lisp et relativement mal adapté au langage C (arguments facultatifs, arguments nommés avec mots clés, etc). Par conséquent, CLX est entièrement écrit en Common Lisp et communique directement avec le protocole X11.

Le code donné ici n'est pas complet dans le sens qu'il ne permet pas la fermeture de la fenêtre, ni la vérification d'erreurs. Il existe uniquement comme support du reste.

```
(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)))
         (display-number (parse-integer s :start (1+ colon) :end dot)))
    (values host-name display-number)))

(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)))
      (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*))
```

3.2 Point

Le petit langage (*prescript*) a besoin de manipuler des points en deux dimensions. Nous avons choisi d'implémenter cette abstraction sous la forme d'une *classe*. Ici, c'est un peu exagéré, car une abstraction plus légère (une *structure*) pourrait être suffisante. La raison principale d'utiliser une classe est pour illustrer la programmation orientée objets en Common Lisp.

Les objets de type `point` sont immuables. Cela évite la nécessité de copier le point actuel lors de l'exécution d'une commande `gsave`. Une modification du point actuel passe alors forcement par le remplacement du point actuel dans le sommet de la pile, par un nouveau point obtenu à partir de l'ancien point. On appelle ce style de programmation *programmation fonctionnelle*, car les sous programmes se comportent presque comme des fonctions mathématiques (sans effets de bord).

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

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

On remarque la convention de créer un *constructeur* qui appelle `make-instance`. Le client doit éviter l'utilisation directe de *make-instance*. La raison est que *make-instance* rend visible un certain nombre de détails d'implémentation de la classe.

3.3 Transformation

Le code contenu de cette section permet de créer des transformation affines, de composer deux transformations, et de transformer un point par une transformation afin d'obtenir un autre point.

On remarque l'utilisation de la construction '#.' permettant l'évaluation d'une expression pendant la phase de lecture par `read`, ainsi que l'utilisation de valeurs multiple pour récupérer les composants d'une transformation.

Les objets de type `transformation` sont immuables. Comme avec le point actuel, cela évite la nécessité de copier la transformation actuelle lors de l'exécution de la commande `gsave`.

```
(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)))

(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))

(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)))))

(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)))))
```

3.4 Chemin

Comme le point actuel et la transformation actuelle, le chemin actuel doit être immuable pour que la commande `gsave` soit rapide.

Pour l'implémentation du chemin, nous utilisons le fait que les opérations Common Lisp de base sur les listes sont sans effets de bord. Un chemin est alors implémenté sous la forme de liste de sous chemin. Chaque sous chemin est implémenté sous la forme de liste d'éléments. Le premier élément de chaque sous chemin est un point (le premier point du chemin), et les autres sont des segments (de droites, d'arcs ou de courbes).

Afin de rendre les opérations rapides, nous avons choisi de stocker les sous chemins d'un chemin en sens inverse. Le premier sous chemin est alors le dernier élément de la liste. De même les éléments d'un sous chemin sont stockés en sens inverse. Le point initial est donc le dernier élément de la liste et le dernier élément rajouté est le premier élément de la liste. Cela nous permet d'utiliser `cons` pour la construction de chemins et de sous chemins. Cette fonction étant sans effets de bord, nous n'avons bas besoin de copier le chemin lors de l'exécution de la commande `gsave`.

On remarque que pour tracer un chemin, on le renverse d'abord, ce qui n'est pas strictement nécessaire. Chaque sous chemin doit par contre être renversé.

Nous avons utilisé la fonction Common Lisp `reduce` pour parcourir les éléments d'un sous chemin. Cette fonction permet de travailler sur chaque élément d'une liste avec un valeur accumulée (dans notre cas, le point courant).

On remarque aussi l'utilisation d'une fonction générique `stroke-one`. Cette fonction prend un point et un segment, trace le segment et renvoi comme valeur le nouveau point courant. Cette fonction sera spécialisée par des méthodes spécifiques pour chaque type de segment. Dans notre cas, seul le segment de droite est implémenté.

```
;;; 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))

;;; 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))))
```

3.5 Segments

Seul le segment de droite est implémenté. Il faut imaginer ici une implémentation de chaque type de segment (arc, courbe).

On remarque surtout la méthode sur la fonction générique `stroke-one`. Cette méthode est appelée lorsque le deuxième argument à `stroke-one` est de type `lineseg`.

```
(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))
```

3.6 Environnement

La pile des environnements contient des éléments de type environnement (`env`). Un tel environnement contient tous ce qu'il faut pour la construction et traçage de chemins, à savoir le point actuel, la transformation actuelle et le chemin actuel.

Un objet de type `env` n'est pas immuable. Ces composants sont constamment modifiés par les commande du langage *prescript*.

```
(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)))
```

Chapitre 4

Un interprète pour *prescript*

Dans ce chapitre, nous allons développer un interprète pour le petit langage *prescript*.

Cet interprète nous permet d'écrire des programmes de ce type :

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

*
```

Seul des constantes peuvent être utilisées comme arguments aux fonctions *prescript*.

Voici le code de l'interprète :

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

(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))))))

(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))))))

(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))))
```

```
(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)))))
```

On remarque l'utilisation de la *programmation dirigée par les données* (en Anglais : data-driven programming).

La version ci-dessus contient beaucoup de duplication de code. En particulier, l'idiome :

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

est répété pour chaque commande. Grâce aux macros Common Lisp, il est possible de capturer cet idiome sous la forme de construction syntaxique. Voici le code modifié :

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

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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))))

(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))))

(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))

(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)))))
```

Chapitre 5

Un langage enchâssé pour *prescript*

Dans ce chapitre, nous allons développer un compilateur pour *prescript*. Le code généré sera du code Common Lisp qui sera ensuite compilé par le compilateur Common Lisp.

Voici un exemple de comment se servir de cette version :

```
* (prescript
   (moveto 3 4)
   (gsave
    (scale 2 3)
    (loop for i from 5 downto 0
          for j from 0 to 5
          do (lineto i j))
    (stroke))
   (lineto 1 1)
   (stroke))

*
```

Comme le montre l'existence d'une boucle `loop` à l'intérieur du bloc `prescript`, le nouveau langage est complètement intégré dans Common Lisp.

Ici, `prescript` est clairement une macro, car elle doit créer un contexte contenant la transformation actuelle et le point de dessin actuel. Notre tâche est d'écrire la macro `prescript`.

À l'intérieur du bloc `prescript`, `moveto`, `scale`, `newpath` et `lineto` doivent de préférence être des fonctions pour que le programmeur puisse faire quelque chose comme ceci :

```
* (prescript
   (moveto 0 0)
   (mapc #'lineto '(1 2 3) '(5 4 3))
   (stroke))

*
```

Dans un premier temps, nous ne traitons pas le cas de `gsave`. La stratégie sera donc de créer, grâce à `labels`, des fonctions locales, disponibles à l'intérieur du bloc :

```
(defmacro prescript (&body body)
  `(labels
       ((moveto (x y)
          ...)
        (scale (sx sy)
          ...)
        ...
    ,@body
    (values)))
```

La construction `(values)` à la fin est pour marquer le fait que le bloc `prescript` ne génère pas de valeur utile.

Maintenant, il faut introduire la pile d'environnements.

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

```
        (translate (tx ty)
          ...)
        (rotate (angle)
          ...)
        (newpath ()
          ...)
        (lineto (x y)
          ...)
        (stroke ()
          ...)))
       ,@body
       (values))))))
```

On remarque l'utilisation de `gensym` pour éviter des problèmes de capture de variables.

Afin de simplifier le code pour les commandes *prescript*, on introduit deux macros utilitaires :

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

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

La première permet d'exécuter une suite d'expressions dans un contexte où la variable `e` contient l'environnement actuel, la variable `position` contient le point actuel, la variable `transformation` contient la transformation actuelle et la variable `path` contient le chemin actuel.

La deuxième permet d'exécuter une suite d'expressions dans un contexte dans lequel un point a été transformé par une transformation dont le nom est `transformation`. Elle est donc destinée à être utilisée dans le corps d'un appel à la macro `with-current-env`.

Ces deux macros capture de manière intentionnelle des variables.

Avec ces macros supplémentaires, voici maintenant le code de la macro `prescript` :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The macro prescript

(defmacro prescript (&body body)
  (let ((envstack (gensym)))
    `(let ((,envstack (list (make-initial-env))))
      (labels
          ((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)))))
           (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)))))
           (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))))
        ,@body
        (values)))))
```

Nous allons maintenant attaquer la partie la plus difficile, à savoir `gsave`. Il est impossible d'écrire `gsave` sous la forme de fonction. La raison est que l'environnement actuel doit être modifié *avant* l'exécution de la première expression du corps de `gsave`. Par conséquent, il faut utiliser une macro.

La difficulté principale est que la macro `gsave` est imbriquée dans une autre macro, `prescript`. On peut dire que la macro `with-transformed-point` est imbriquée dans la macro `with-current-env` ci-dessus, mais c'est une imbrication d'appels uniquement. Dans le cas de `gsave`, la macro `prescript` doit *définir* la macro `gsave` et ceci d'une manière locale, analogue à `labels` pour les fonctions.

La solution est d'utiliser `macrolet`. Voici le code qui en résulte :

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The macro prescript

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

Maintenant, imaginons une évolution possible de la macro `prescript`. Un utilisateur sophistiqué souhaite peut-être pouvoir rajouter de nouvelles fonctions comme `curveto` ou `arcto`. Il convient alors de définir un langage enchâssé pour ce type d'extensions. Voici une possibilité :

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

(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))))

(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*)))

(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)))

(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)))

(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))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; The macro prescript

(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)
                    `(progn (push (copy-env (car ,',envstack))
                                  ,',envstack)
                      (unwind-protect (progn ,@body)
                        (pop ,',envstack)))))
          ,@body
          (values))))))
```