ANNÉE UNIVERSITAIRE 2007/2008
2IÈME SESSION D'AUTOMNE

Parcours : CSB6
Code UE : INF353
Épreuve : Programmation 3 : Programmation Fonctionnelle et Symbo
Date : mars 2008
Heure : 8h30
Durée : 1h30
Documents : autorisés
Épreuve de Mme Irène Durand

Le barème est donné à titre **indicatif**. Le sujet comporte 4 pages.

Exercice 1 *(2pts)*

Évaluer les expressions suivantes :

1. `(cons 1 2)`
2. `(cons 1 (cons 2 nil))`
3. `(list 'a (+ 1 2) ())`
4. `(append '(a) (list 1 2) ())`

Exercice 2 *(7pts)*

Soient les fonctions `mystere1 (l pred)` et `mystere2 (l pred)` suivantes :

```
(defun mystere1 (l pred)
  (let ((ok ())
        (not-ok ()))
    (dolist (e l (list (nreverse ok) (nreverse not-ok)))
      (if (funcall pred e)
          (push e ok)
          (push e not-ok)))))

(defun mystere2 (l pred)
  (list
   (remove-if-not pred l)
   (remove-if pred l)))
```

1. Que retourne l'appel `(mystere1 '(1 4 3 2 5 6) #'evenp)` ?
2. En supposant que `pred` soit purement fonctionnel (ne produise aucun effet de bord), `mystere1` et `mystere2` sont-elles équivalentes (même résultat pour mêmes arguments) ?
3. `mystere2` retourne une liste contenant **deux listes** l_1 et l_2. Modifier `mystere2` de manière à ce qu'elle retourne **deux valeurs** l_1 et l_2.
4. Donner un exemple d'appel de cette nouvelle version et les valeurs retournées.
5. Donner une version récursive de `mystere1` (on pourra s'aider d'une fonction auxiliaire `mystere1-aux (l pred ok not-ok)`

Exercice 3 *(4pts)*
Soit la macro suivante :

```
(defmacro with-staff-size (size &body body)
  `(let ((size-var ,size))
    (unless (aref *fonts* size-var)
      (setf (aref *fonts* size-var)
            (make-font size-var)))
    (let ((*font* (aref *fonts* size-var)))
      ,@body)))
```

1. Que retourne l'appel suivant
   `CL-USER> (macroexpand-1 '(with-staff-size 12 (write-line) (write-line)))`
2. Quel problème classique pose le corps de cette macro ?
3. Proposer une solution pour corriger ce problème.

Exercice 4 *(7pts)*
Le programme donné Page **??** est l'ébauche d'une bibliothèque pour manipuler des régions géométriques en deux dimensions. Un point (ou un vecteur) du plan est représenté par un nombre complexe.
Dans un premier temps, deux opérations principales sont à implémenter
– le calcul de la boîte englobante d'une région,
– le calcul de l'aire d'une région.
On commence par traiter des coniques (cercles et ellipses). Pour simplifier on se restreint aux ellipses dont les axes sont parallèles aux axes du repère.

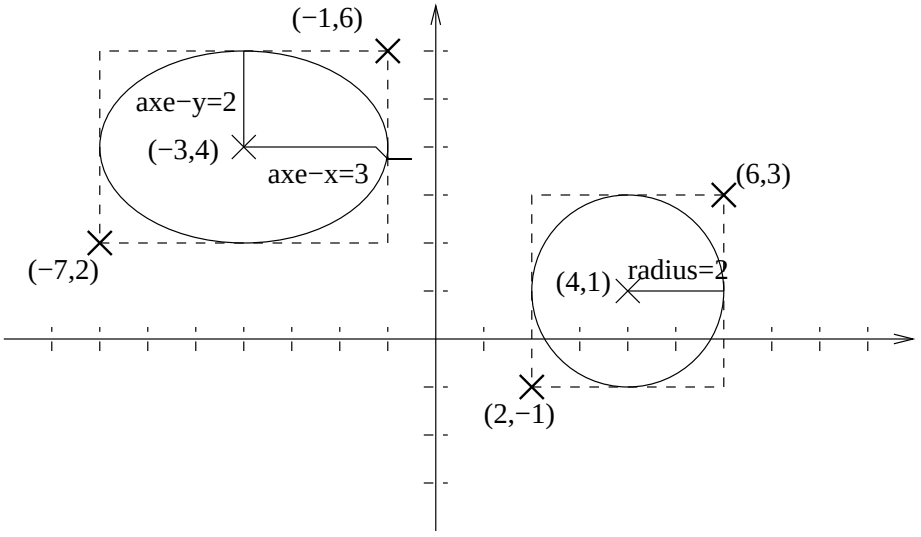

FIG. 1 – Boîtes englobantes

1. Représenter sous forme de schéma la hiérarchie la hiérarchie des classes.
2. Écrire une fonction
   `make-ellipse (axe-x axe-y &optional (center #C(0 0)))`

qui retourne une ellipse de demi-axe horizontal `axe-x`, de demi-axe vertical `axe-y` et de centre `center`.

3. Implémenter la méthode `area` pour la classe `ellipse`. On rappelle que l'aire d'une ellipse de demi-axes x et y peut être calculée par la formule πxy.
4. Implémenter la méthode `bounding-box` pour la classe `ellipse`.
5. Écrire une méthode `print-object` spécialisée pour la classe `ellipse` de telle sorte qu'une ellipse soit imprimée comme le montre l'exemple suivant.

```
CL-USER> (defparameter *e* (make-ellipse 2 3))
*E*
CL-USER> *e*
#<ELLIPSE BEB1239> Center:0 Axe-x:2 Axe-y:3
```

```
(defgeneric bounding-box (region)
  (:documentation
   "computes the bounding-box of REGION
  (2 values: lower-left and upper-right corners of REGION"))

(defgeneric area (region)
  (:documentation
   "computes area of REGION"))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Regions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defclass region () ())

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Conics
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defclass conic (region)
  ((center :initform 0 :accessor center :initarg :center)))

(defmethod print-object ((conic conic) stream)
  (call-next-method)
  (format stream " Center:~A" (center conic)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Circle
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defclass circle (conic)
  ((radius :initform 0 :accessor radius :initarg :radius)))

(defun make-circle (radius &optional (center #C(0 0)))
  (make-instance 'circle :center center :radius radius))

(defmethod print-object ((circle circle) stream)
  (call-next-method)
  (format stream " Radius:~A" (radius circle)))

(defmethod bounding-box ((circle circle))
  (let* ((radius (radius circle))
         (center (center circle))
         (x (realpart center))
         (y (imagpart center)))
    (values
     (complex (- x radius) (- y radius))
     (complex (+ x radius) (+ y radius)))))

(defmethod area ((circle circle))
  (let ((r (radius circle)))
    (* pi r r)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Ellipse
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defclass ellipse (conic)
  ((axe-x :initform 0 :accessor axe-x :initarg :axe-x)
   (axe-y :initform 0 :accessor axe-y :initarg :axe-y))
  (:documentation "Ellipse with axes parallel to coordinates axes"))
```