ANNÉE UNIVERSITAIRE 2006/2007
1ÈRE SESSION DE PRINTEMPS

Parcours : INF6 / MAI6 / MIAGE6
Code UE : INF207
Épreuve : Programmation Fonctionnelle et Symbolique
Date : Mardi 15 Mai 2007
Heure : 11h
Durée : 1h30
Documents : autorisés
Épreuve de Mme Irène Durand et Mr Robert Strandh

Le barême est donné à titre **indicatif**

Exercice 1 *(2pts)*

Un programmeur Lisp médiocre décide de tester interactivement le fonctionnement des tables de hachage en implémentant un dictionnaire Anglais-Français. Voici le dialogue entre le programmeur et la boucle d'interaction :

```
CL-USER> (defparameter *h* (make-hash-table))
*H*
CL-USER> (setf (gethash "student" *h*) "étudiant")
"étudiant"
CL-USER> (gethash "student" *h*)
NIL
NIL
```

1. Expliquer pourquoi la dernière expression ne permet pas de récupérer la traduction de `"student"` préalablement stockée dans la table. (1p)
2. Comment modifier le dialogue pour permettre cette récupération ? (1p)

Exercice 2 *(4pts)*

Écrire une macro `(num-if expr positif zero negatif)` qui réalise une variante de l'instruction `if`. Le premier argument `expr` doit être une expression numérique tandis que les arguments `positif`, `zero`, `negatif` doivent être chacun une expression.
La sémantique de l'instruction `num-if` est la suivante : si le résultat de l'évaluation de `expr` est strictement positif, l'expression `positif` est évaluée, sinon si le résultat est nul, l'expression `zero` est évaluée, sinon (le résultat est forcément strictement négatif), l'expression `negatif` est évaluée. Exemples :

```
CL-USER> (defparameter *i* -1)
*I*
CL-USER> (num-if (* *i* *i*) 'p 'z 'n)
P
CL-USER> (num-if (+ *i* *i*) 'p 'z 'n)
N
CL-USER> (num-if (- *i* *i*) 'p 'z 'n)
Z
```

Exercice 3 *(5pts)*
Soit la fonction `fun-filter` suivante :

```
(defun fun-filter (l fun &optional (to-remove #'null))
  (if (endp l)
      '()
      (let ((res (funcall fun (car l))))
        (if (funcall to-remove res)
            (fun-filter (cdr l) fun to-remove)
            (cons res (fun-filter (cdr l) fun to-remove))))))
```

Que retournent les appels suivants ?

1. `(fun-filter '((1 4 6) (3) (2 1) (5)) #'cdr)`
2. `(fun-filter '(1 4 3 2 1 5) #'1- #'zerop)`
3. Écrire une version itérative de `fun-filter` en utilisant une boucle explicite au choix (`do, dolist, loop,...`).
4. Écrire une version itérative de `fun-filter` sans utiliser de boucle explicite mais en utilisant des fonctions d'application prédéfinies de `Lisp`.

Exercice 4 *(8pts)*
On souhaite réaliser une bibliothèque pour la manipulation de régions géométriques en deux dimensions. On choisit de représenter un point (ou un vecteur) du plan par un **nombre complexe**.
Le protocole de la bibliothèque définit la classe abstraite `region` sans créneaux qui est la racine de l'arborescence de toutes les classes représentant des régions géométriques,

```
(defclass region () ())
```

ainsi que quatre fonctions génériques :

```
(defgeneric move (region vector))
(defgeneric scale (region scalar))
(defgeneric rotate (region angle))
(defgeneric bounding-box (region))
```

permettant respectivement
– le déplacement (translation de vecteur `vector`)
– la mise à l'échelle (multiplication des coordonnées de la région par le scalaire `scalar`),
– la rotation par rapport à l'origine d'angle `angle` et
– le calcul de la boîte englobante d'une région.
Les fonctions `move, scale` et `rotate` sont destructives et ne retournent pas de valeur utile.
La fonction `bounding-box` retourne **deux nombres complexes** correspondant aux deux coins (inférieur gauche et supérieur droit) du plus petit rectangle englobant la région et dont les côtés sont parallèles aux axes du repère. Des exemples de boîte englobante sont donnés Figure 1.
La bibliothèque encourage les clients à créer leurs propres régions (sous classes de la classe `region`) ; elle fournit néammoins deux régions `circle` et `polygon` ainsi définies :

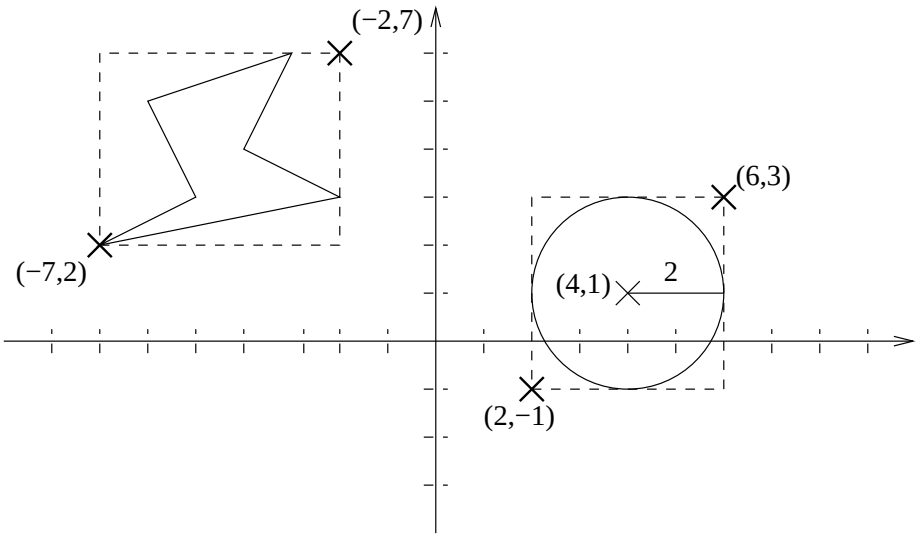

FIG. 1 – Boîtes englobantes

```
(defclass circle (region)
  ((center :initarg :center :accessor center)
   (radius :initarg :radius :accessor radius)))

(defclass polygon (region)
  ((corners :initarg :corners :accessor corners)))
```

On rappelle que si z est l'affixe d'un point p du plan, le transformé de p par la rotation d'angle θ et de centre $(0, 0)$ a pour affixe $e^{i\theta}z$.

1. Compléter l'implémentation des méthodes spécialisées pour le cercle. (2pts)
2. Compléter l'implémentation des méthodes spécialisées pour le polygone. (3pts)

```
(defmethod move ((circle circle) vector)
  ...)

(defmethod scale ((circle circle) scalar)
  ...)

(defmethod rotate ((circle circle) angle)
  ...)

(defmethod bounding-box ((circle circle))
  ...)

(defmethod move ((polygon polygon) vector)
  (setf (corners polygon)
        (mapcar (lambda (corner) (+ corner vector)) (corners polygon))))
```

```
(defmethod scale ((polygon polygon) scalar)
  ...)

(defmethod rotate ((polygon polygon) angle)
  ...)

(defmethod bounding-box ((polygon polygon))
  ...)
```

Quelques mois après la première livraison de la bibliothèque, on s'aperçoit que l'opération `bounding-box` est employée beaucoup plus fréquemment que les autres opérations. Pour améliorer les performances, on souhaite stocker la valeur de la boîte englobante de chaque région pour ne pas devoir la recalculer à chaque appel. Cependant, la boîte englobante d'une région doit être recalculée chaque fois que la région est modifée par l'une des autres opérations.

3. Montrer comment modifier le code pour permettre cette optimisation sans que les clients aient besoin de modifier le code dont ils disposent déjà. (3pts)