ANNÉE UNIVERSITAIRE 2007/2008
1ÈRE SESSION D'AUTOMNE

Parcours : CSB5
Code UE : INF353T
Épreuve : Programmation Fonctionnelle et Symbolique
Date : Lundi 21 avril 2008
Heure : 8h30
Durée : 1h30
Documents : autorisés
Épreuve de Mme Irène Durand

Le barème est donné à titre **indicatif**. Le sujet comporte 2 pages et une annexe.

Exercice 1 *(2pts)*

Évaluer les expressions suivantes :

1. `(cons '(a) '(b c d))`
2. `(list '(a) '(b c d))`
3. `(append '(a) '(b c d))`
4. `(cons nil nil)`

Exercice 2 *(4pts)*

Écrire une fonction `op-vecteur (v1 v2 op)` qui s'applique à deux vecteurs `v1` et `v2` de même longueur n et à une opération binaire `op` pouvant s'appliquer aux éléments des vecteurs et qui retourne un nouveau vecteur v de même longueur et tel que pour chaque `i`, $0 \leq$ `i` $< n$, `v[i] = v1[i] op v2[i]`. Exemples :

```
SMOTS> (op-vecteur #(1 2 3) #(2 3 4) #'+)
#(3 5 7)
SMOTS> (op-vecteur #(1 2 3) #(2 3 4) #'*)
#(2 6 12)
```

On rappelle qu'un vecteur est un tableau à une dimension.

Exercice 3 *(4pts)*

Soit la fonction `leaves` et la variable `*tree*` suivante :

```
(defun leaves (tree &optional (predicate #'(lambda (x) (declare (ignore x)) t)))
  (let ((leaves '()))
    (labels ((aux (tr)
               (if (atom tr)
                   (when (funcall predicate tr)
                     (push tr leaves))
                   (mapc #'aux tr))))
      (aux tree)
      (nreverse leaves))))
```

```
(defparameter *tree* '((1 r 4 f (6 7)) ((2 4) (d 3.4))))
```

Que retournent chacun des deux appels suivants ?

1. `(leaves *tree*)`
2. `(leaves *tree* #'integerp)`

Exercice 4 *(4pts)*
Soit la macro `threaded-length` suivante :

```
(defmacro threaded-length (list next type)
  `(do ((x ,list (,next (the ,type x)))
        (count 0 (index1+ count)))
       ((null x)
        count)))
```

1. Que retourne l'appel suivant ?

   ```
   (macroexpand-1 '(threaded-length queue reply-next reply-buffer))
   ```

2. Quel problème peut se poser avec cette macro ?
3. Proposer une version corrigée de cette macro.

Exercice 5 *(6pts)*
Le programme `region.lisp` donné en annexe contient le début d'une implémentation d'un paquetage destiné à manipuler des *régions* du plan. Un point est représenté par un nombre complexe.

1. Dessiner la hiérarchie des classes.
2. Implémenter la méthode `compute-area` pour les objets de la classe `circle`.
3. Compléter l'implémentation de l'opération `print-object` de manière à obtenir le comportement suivant :

   ```
   CL-USER> (make-polygon 'polygon #C(0 -1) #C(0 1) #C(1 2))
   #<POLYGON {EB71B11}>(corners: (POLYGON #C(0 -1) #C(0 1) #C(1 2)))
   ```

Le calcul de l'aire des polygones s'avère très coûteux. On souhaite donc mémoïser l'aire des polygones, c'est-à-dire, au premier appel de l'opération `area`, calculer l'aire et la sauvegarder et aux appels suivants utiliser la valeur mémorisée au lieu de refaire le calcul.

4. Proposer une solution pour la mémoïsation de l'aire des polygones.