Université Bordeaux 1 Licence Informatique 2004/2005

UE INF207 - Programmation Fonctionnelle et Symbolique
Devoir surveillé

Tous documents autorisés *Jeudi 14 Avril 2005* *Durée 1h15*

Exercice 1 *Évaluer les expressions suivantes :*

1. `(cons '(A) '(B))`
2. `(append '(A (B)) '(C) '(D))`
3. `(remove '(A) '((A) B C) :test #'equal)`
4. `(mapcar (lambda (e) (if (atom e) 0 (length e))) '((1) 2 (3 4 5)))`

Exercice 2 *Listes*
Ecrire une version récursive de la fonction `map-append(f l)` qui retourne la concaténation des listes obtenues en appliquant la fonction `f` à chacune des sous-listes de la liste `l`.
Ecrire une version itérative de la fonction `append-sublist(l)` qui retourne la concaténation des sous-listes de la liste `l`. Indication : placer chaque élément de chaque sous-liste de `l` dans une liste.
Utiliser la fonction `append-sublist` pour écrire une autre version de `map-append`.
Exemples :

```
> (map-append #'reverse '((1 2 3) (4 (5)) (6)))
(3 2 1 (5) 4 6)
> (map-append #'butlast '((1 2 3) (4 (5)) (6 7)))
(1 2 4 6)
> (append-sublist '((3 2 1) ((5) 4) (6)))
(3 2 1 (5) 4 6)
> (append-sublist '((1 2) (4) (6)))
(1 2 4 6)
```

Exercice 3 *Structures*
On définit la structure `ville` de la façon suivante :

```
(defstruct ville
  nom
  departement
  population
  x
  y)
```

Les créneaux `departement`, `population`, `x` et `y` sont numériques. `departement` et `population` correspondent respectivement au numéro de département et au nombre d'habitants de la ville. `x` et `y` correspondent aux coordonnées (en km) de cette ville dans un plan euclidien.

1. Ecrire une fonction `distance-villes (ville1 ville2)`, où `ville1` et `ville2` sont des structures de type `ville`, et qui retourne la distance entre ces deux villes.

   La distance entre les deux points $A(x_a, y_a)$ et $B(x_b, y_b)$ vaut $\sqrt{|x_a - x_b|^2 + |y_a - y_b|^2}$.

   Note : L'écriture des trois fonctions décrites ci-dessous nécessite l'utilisation de la fonction `sort`. La fonction `adjoin` pourra être utilisée. Une documentation de ces fonctions et des exemples d'utilisation sont donnés en fin de sujet.

2. Ecrire une fonction `tri-population (villes)` qui trie la liste de villes passée en paramètre par nombre décroissant d'habitants. La fonction retourne la liste triée. Cette fonction est destructive (c'est à dire, elle peut modifier la liste passée en paramètre).

3. Ecrire une fonction `tri-population-departement (villes)` qui, à partir d'une liste de villes, retourne la liste contenant la ville la plus peuplée de chaque département, cette liste étant triée par numéro croissant de département. Cette fonction est destructive.

4. Ecrire une fonction `tri-population-distance (villes d)` qui, à partir d'une liste de villes, retourne une liste de villes triée par nombre décroissant d'habitants et telle que la distance entre deux villes quelconques est supérieure à `d` km. Cette liste contient prioritairement les villes les plus peuplées. Cette fonction est destructive.

Exercice 4 *Tables de hachage*

Nous considérons une table de hachage contenant des villes (cf. exercice 3, structure `ville`), les clés étant des symboles représentant les noms des villes.
Ecrire une fonction `liste-villes-distances (table-villes cle-ville)` qui retourne une liste contenant la distance entre chaque ville dans la table de hachage `table-villes` et la ville obtenue à partir de la clé `clé-ville`. Plus présicément, la liste retournée est de la forme : $((nom_1\ d_1)\ (nom_2\ d_2)\ ...)$ où nom_i est le nom d'une ville dans la table et d_i la distance entre cette ville et la ville de clé `clé-ville`. La fonction retourne `nil` si aucun élément n'existe dans la table avec la clé donnée.

Exemples d'utilisation de `sort` *:*

```
>  (sort (list 0 3 2 7 1) #'<)
(0 1 2 3 7)
> (sort (list '(4) '(7 0 9) '(1 6)) #'> :key #'car)
((7 0 9) (4) (1 6))
> (sort (list '(5 "fifi") '(7 "riri") '(9 "loulou")) #'string-lessp :key #'cadr)
((5 "fifi") (9 "loulou") (7 "riri"))
```

Exemples d'utilisation de `adjoin` *:*

```
> (adjoin '(3 2) '((0 2) (4 7)) :key #'car)
((3 2) (0 2) (4 7))
> (adjoin '(4 5) '((0 2) (4 7)) :key #'car)
 ((0 2) (4 7))
> (adjoin '(4 5) '((0 2) (4 7)) :key #'car :test #'<)
((4 5) (0 2) (4 7))
```