

## Programmation 3 : feuille 5

Listes (suite)

### Exercice 5 .1

La fonction `mapcar` retourne une liste dont la longueur est égale à celle de la liste la plus courte des listes passées en arguments. On souhaite réaliser une variante de cette fonction `mapcar-fill` (`f l1 l2`) où `f` est une fonction, `l1` et `l2` des listes et qui retourne la même chose que `mapcar` si les deux listes sont de même longueur et une liste dont les premiers éléments sont les mêmes que ce qui est renvoyé par `mapcar` et le reste des éléments sont les éléments supplémentaires de la liste la plus longue. Exemples :

```
* (mapcar-fill #'(1 5 8) '(3 4 2))
(4 9 10)
```

```
* (mapcar-fill #'(1 5 8) '(3 4 2 6 7 4 2))
(4 9 10 6 7 4 2)
```

```
* (mapcar-fill #'(3 4 2 6 7 4 2) '(1 5 8))
(2 -1 -6 6 7 4 2)
```

Écrire la version de référence de cette fonction. Elle doit donc être récursive et la performance n'est pas un objectif. Vérifier que la fonction donne le même résultat que `mapcar` sur un grand nombre de listes.

### Exercice 5 .2

1. Écrire une fonction `mappend` (`fun list`) qui applique la fonction `fun` à chaque élément de la liste `list` (et dont le résultat doit être une autre liste) puis renvoie la concaténation des résultats ainsi obtenus. Exemple :

```
* (mappend (lambda (x) (list x (1+ x))) '(5 3 9 2 5))
(5 6 3 4 9 10 2 3 5 6)
```

2. Tester votre fonction sur une liste avec un million d'éléments (se servir de `make-list`). Qu'est-ce qui arrive ?
3. Lire la rubrique concernant `CALL-ARGUMENTS-LIMIT` dans la HyperSpec, et expliquer pourquoi la version suivante de la fonction `mappend` n'est pas idéale :

```
(defun mappend2 (fun list)
  (apply #'append (mapcar fun list)))
```

4. Lire la rubrique concernant `reduce` dans la HyperSpec et modifier la version précédente de la fonction `mappend` en utilisant `reduce`.
5. Mesurer le temps d'exécution de la fonction précédente sur une liste de taille 10000 (se servir de `make-list` pour générer une telle liste)
6. (Difficile) Rajouter l'argument mot-clé `:from-end t` à `reduce` dans la version précédente de la fonction `mappend`. Qu'est-ce que vous observez ? Expliquer le résultat !

### Exercice 5 .3

Soit la fonction `next-line` (1) suivante :

```
(defun next-line (l)
  (mapcar #'(lambda (x) (cons 0 x)) (append l (list 0))))
```

1. Tester la fonction `next-line` en l'appliquant  $n$  fois à la liste (1).
2. En utilisant la fonction `next-line`, écrire une fonction `triangle` ( $n$ ) qui construit une liste contenant les listes correspondant aux lignes du triangle de Pascal d'ordre  $n$ .

```
CL-USER> (triangle 0)
((1))
CL-USER> (triangle 1)
((1) (1 1))
CL-USER> (triangle 2)
((1) (1 1) (1 2 1))
CL-USER> (triangle 3)
((1) (1 1) (1 2 1) (1 3 3 1))
```

3. Étudier l'inefficacité possible de votre fonction due à l'utilisation éventuelle des fonctions `append` et `reverse`. Peut-on remplacer l'une et/ou l'autre de ces fonctions par leur version destructive? Si oui, le faire, si non, expliquer pourquoi.

### Exercice 5 .4

1. Écrire une fonction `partition` (`pred`  $l$ ) qui renvoie deux valeurs : la liste des éléments de  $l$  qui vérifient le prédicat `pred` et la liste des éléments de  $l$  qui ne vérifient pas `pred`. L'ordre des éléments dans les listes retournées est respecté par rapport à l'ordre dans la liste  $l$ .
2. Utiliser la fonction `partition` pour écrire une fonction `quicksort` (`less-than`  $l$ )

*Exemples :*

```
CL-USER> (quicksort #'< '(5 8 2 9 56 8 4 9))
(2 4 5 8 8 9 9 56)
CL-USER> (quicksort #'string< '("az" "ab" "cd" "aa" "cd" "ef"))
("aa" "ab" "az" "cd" "cd" "ef")
```

### Exercice 5 .5 *type abstrait*

Écrire la base d'une implémentation de la notion d'ensemble à l'aide de listes (utiliser un argument mot-clé pour l'égalité) :

1. Définir une constante `empty-set` pour représenter l'ensemble vide.
2. Écrire la fonction `set-empty-p` qui teste si un ensemble est vide.
3. Écrire la fonction `set-member` qui teste si un élément appartient à un ensemble.
4. Écrire la fonction `set-adjoin` qui ajoute un élément à un ensemble.
5. Écrire la fonction `set-from-list` qui construit l'ensemble des éléments d'une liste.

Remarque : si on choisit d'implémenter les ensembles par des listes, on pourra utiliser les fonctions `Lisp` `adjoin`, `member`, `remove-duplicates`.

6. Implémenter les fonctionnalités suivantes : union, intersection, différence, tests d'égalité et d'inclusion.

On peut représenter un graphe fini par l'ensemble de couples  $(i, j)$ , correspondant aux arêtes de ce graphe. Cependant, si on considère un graphe non orienté, on doit considérer comme égales les arêtes  $(i, j)$  et  $(j, i)$ .

7. Utiliser le paramètre associé à l'égalité dans les fonctions `set-from-list`, `set-intersection`, etc. pour manipuler des ensembles d'arêtes (et non pas d'arcs!).
8. Étudier les limites de cette représentation des ensembles.

## Exercice 5 .6

1. Écrire une fonction `compress` (`l &key (test #'equal)`) qui compresse la liste `l` sous forme de couples (`element . facteur-de-repetition`).

*Exemples :*

```
CL-USER> (compress '(a a a b b c a))
```

```
((A . 3) (B . 2) (C . 1) (A . 1))
```

```
CL-USER> (compress '("ab" "Ab" "aB" "c" "AB") :test #'string-equal)
```

```
((("ab" . 3) ("c" . 1) ("AB" . 1))
```

2. Écrire la fonction inverse `uncompress` (`l`)

```
CL-USER> (uncompress (compress '("ab" "Ab" "aB" "c" "AB") :test #'string-equal))
```

```
("ab" "ab" "ab" "c" "AB")
```