Département Licence	ANNÉE : 2005/2006 SESSION DE SEPTEMBRE 2006 **Parcours :** INF6 **Code UE :** INF207 **Épreuve :** Programmation Fonctionnelle et Symbolique **Date :** Septembre 2006 **Heure :** 11h **Durée :** 1h30 Documents : autorisés Épreuve de Mme Irène Durand et Mr Pierre Castéran

Le barême est donné à titre **indicatif**

Exercice 1 *(3pts)*

Écrire une fonction `display-list (l stream &key sep)` qui écrit dans le flot `stream` les objets de la liste `l` séparés par le séparateur `sep`.

```
CL-USER> (display-list '(1 2 3 4) t :sep ",")
1,2,3,4
NIL
```

Rappel : pour écrire dans un flot `stream`, on peut utiliser une instruction du type
`(format stream "... ~A ...~A..." arg1 arg2 ...)`.

Exercice 2 *(5pts)*

1. Écrire une fonction `split (l pred)` qui partitionne la liste `l` en deux sous-listes : la première contient les éléments qui vérifient le prédicat `pred`, la deuxième ceux qui ne le vérifient pas. La valeur retournée est la liste contenant les deux sous-listes.

   ```
   CL-USER> (split '(1 2 3 4 5) #'evenp)  ;; even en anglais signifie "pair"
   ((2 4) (1 3 5))
   ```

2. Modifier votre fonction pour qu'elle retourne deux valeurs : les deux sous-listes.

   ```
   CL-USER> (split '(1 2 3 4 5) #'evenp)
   (2 4)
   (1 3 5)
   ```

3. En utilisant la deuxième version de `split`, écrire une expression dont l'exécution donne :

   ```
   CL-USER> <expression à trouver>
   oui: (2 4)
   non: (1 3 5)
   NIL
   ```

Exercice 3 *(4pts)*

On rappelle que l'opérateur `OR` est une macro.
On souhaite écrire un opérateur `none (&rest l)` tel que `(none e1 ... en)` retourne VRAI si aucune des expressions `ei` ne s'évalue en VRAI et FAUX (NIL) sinon. Dans le cas où au

moins une expression s'évalue en VRAI, on souhaite retourner nil **dès qu'**on rencontre un `ei` qui s'évalue en vrai sans évaluer les expressions qui restent.

1. Expliquer pourquoi cela ne peut se faire qu'en implémentant `none` sous forme de macro.
2. Écrire cette macro.

Exercice 4 *(8pts)*
On considère le début d'une implémentation d'un paquetage destiné à manipuler des *langages* (ensembles de *mots* formés de *lettres*).
Les lettres sont des instances de la classe `lettre` et les mots des instances de la classe `mot`.
On suppose implémentées les fonctions et fonctions génériques suivantes pour les lettres et les mots.

```
(defun make-lettre (nom) :documentation "lettre de nom NOM")
(defun make-mot (lettres):documentation "mot composé des lettres LETTRES")

(defgeneric lettres (mot)
  (:documentation "liste des lettres du MOT"))

(defgeneric print-object (object stream)
  (:documentation "writes the printed representation of OBJECT to STREAM"))

CL-USER> (setf *a* (make-lettre "a"))
a
CL-USER> (setf *b* (make-lettre "b"))
b
CL-USER> (setf *c* (make-lettre "c"))
c
CL-USER> (setf *waac* (make-mot (list *a* *a* *c*)))
[aac]
CL-USER> (setf *wbac* (make-mot (list *b* *a* *c*)))
[bac]
CL-USER> (lettres *waac*)
(a a c)
```

Remarque : `print-object` pour une lettre imprime le nom de la lettre et pour un mot les lettres de celui-ci entre crochets (`[...]`).

Soit la fonction générique `mots` :

```
(defgeneric mots (langage)
  (:documentation "liste des mots du LANGAGE"))

CL-USER> (setf *l1* (make-langage (list *waac* *wbac*)))
{[aac],[bac]}
CL-USER> (mots *l1*)
([aac] [bac])
```

1. Définir la classe `langage` avec sa méthode `mots`.
2. Écrire la fonction `make-langage` qui construit un langage à partir d'une liste de mots.

Soit l'opération `concatenation` définie par la fonction générique suivante :

```
(defgeneric concatenation (objet1 objet2)
  (:documentation "concaténation des deux objets"))
```

```
CL-USER> *waac*
[aac]
CL-USER> *wbac*
[bac]
CL-USER> (concatenation *waac* *wbac*)
[aacbac]
```

3. Implémenter l'opération `concatenation` de deux mots.

Soit l'opération `alphabet` définie par la fonction générique suivante :

```
(defgeneric alphabet (objet)
  (:documentation "retourne l'ENSEMBLE des lettres de l'OBJET"))
```

```
CL-USER> (alphabet *l1*)
(b a c)
CL-USER> (alphabet *waac*)
(a c)
```

Remarque : l'**ordre** des lettres de l'alphabet n'est **pas important**.

On suppose que les lettres se comparent avec la fonction Common Lisp `eq`.

4. Implémenter l'opération `alphabet` pour les mots.
5. Implémenter l'opération `alphabet` pour les langages.

Soit l'opération `lunion` définie par la fonction générique suivante :

```
(defgeneric lunion (langage1 langage2)
  (:documentation "union des langages LANGAGE1 et LANGAGE2"))
```

```
CL-USER> *l1*
{[aac],[bc]}
CL-USER> *l2*
{[a],[bc]}
CL-USER> (lunion *l1* *l2*)
{[aac],[a],[bc]}
```

6. Implémenter l'opération `lunion`.