DEVUIP Service Scolarité	ANNÉE UNIVERSITAIRE 2012/2013 1ÈRE SESSION D'AUTOMNE **Parcours :** IN501 **Code UE :** IN5011 **Épreuve :** Programmation 3 **Date :** Lundi 17 décembre 2012 **Heure :** 08h30 **Durée :** 1h30 Documents : autorisés Épreuve de Mme Irène Durand	

Le barème est donné à titre **indicatif**. Le sujet comporte 4 pages plus une annexe.

Exercice 1 *(4pts)*

Écrire une fonction `vector-sum (v1 v2)` qui fait la somme de deux vecteurs [1] d'entiers. Si l'un des tableaux est plus court que l'autre les éléments manquants sont considérés comme égaux à `0`. Exemples :

```
CL-USER> (vector-sum #(1 2 3) #(1 1 2))
#(2 3 5)
CL-USER> (vector-sum #(1 2 3) #(1 1 2 2))
#(2 3 5 2)
CL-USER> (vector-sum #(1 2 3) #(1 3))
#(2 5 3)
```

Exercice 2 *(4pts)*

On considère des listes triées selon un prédicat d'ordre total `order-fun`. L'ordre étant total, on peut déduire l'égalité de deux éléments à partir de l'ordre. Si l'ordre est donné par $<$, un élément e_1 est égal à un élément e_2 si et seulement si $\neg(e_1 < e_2) \wedge \neg(e_2 < e_1)$ ou encore $\neg(e_1 < e_2 \vee e_2 < e_1)$

1. Écrire une fonction `equality-fun (order-fun)` qui retourne le prédicat d'égalité associé au prédicat d'ordre `order-fun`.

Exemples :

```
CL-USER> (equality-fun #'<)
#<CLOSURE (LAMBDA (E1 E2) :IN EQUALITY-FUN) {1006702AFB}>
CL-USER> (funcall (equality-fun #'<) 1 2)
NIL
CL-USER> (funcall (equality-fun #'<) 2 2)
T
CL-USER> (funcall (equality-fun #'<) 2 1)
NIL
```

2. Écrire une fonction

1. Un vecteur est un tableau à une dimension.

`member-sort (e ol &key (order-fun #'<))` similaire à la fonction Lisp `member` et qui retourne `NIL` si `e` n'appartient pas à la liste triée `ol`, le reste de la liste à partir de `e` sinon.

L'implémentation **doit** exploiter le fait que la liste est triée pour être plus efficace que la fonction `member`. Le prédicat d'ordre de deux éléments est transmis par le paramètre `order-fun`. Exemples :

```
CL-USER> (member-sort 4 '(1 2 4 6 7))
(4 6 7)
CL-USER> (member-sort 5 '(1 2 4 6 7))
NIL
CL-USER> (member-sort "titi" '("tete" "titi" "toto" "tutu") :order-fun #'string<)
("titi" "toto" "tutu")
CL-USER> (member-sort "tata" '("tete" "titi" "toto" "tutu") :order-fun #'string<)
NIL
```

Exercice 3 *(8pts)*

Soit le début d'implémentation de la notion d'**ensemble** donnée en annexe page 5.

1. Écrire une version spécialisée de l'opération `print-object` de manière à ce qu'un ensemble soit écrit par le `Printer` sous la forme : `#S(e1 ... en)` où `e1, ..., en` sont les éléments de l'ensemble.

Exemples :

```
CL-USER> (setf *s* (make-set-from-list '(1 2 3 5 1 4 2 5 2 4)))
#S(3 1 5 2 4)
CL-USER> (setf *s* (adjoin-set 6 *s*))
#S(6 3 1 5 2 4)
```

Dans une phase de débogage, on souhaite vérifier que la liste des éléments fournis à l'opération `make-set` ne contient pas de doublons.

2. Sans modifier le code existant, proposer une solution permettant de lever une exception dans le cas où on appellerait `make-set` avec une liste d'éléments contenant des doublons.

Exemples :

```
CL-USER> (setf *s* (make-set '(1 5 4 2 6)))
#S(1 5 4 2 6)
CL-USER> (setf *s* (make-set '(1 2 4 2 6)))
=> assertion ... failed
```

Le code des méthodes `intersection-set` et `union-set` est très ressemblant. Pour éviter la duplication de code, on souhaite une opération plus générale
`op-bin-set (set1 set2 bin-op)` permettant de réaliser une opération binaire sur deux ensembles `set1` et `set2`. L'opération à effectuer est transmise par le paramètre `bin-op`. Exemples :

3. Définir l'opération `op-bin-set`.
4. Implémenter l'opération `op-bin-set`.

5. Implémenter l'opération `difference-set` en utilisant `op-bin-set`.

On souhaite manipuler des ensembles d'élements sur lesquels il existe un ordre et tirer parti de ce fait pour rendre la représentation plus efficace en représentant les éléments de l'ensemble par une liste ordonnée.

6. Définir la classe `ordered-set` pour les ensembles ordonnés.
7. Implémenter [2] l'opération `member-set` pour les ensembles ordonnés.

Exercice 4 *(4pts)*
On dispose d'objets complexes (graphes, automates, ...) composés en partie d'éléments (noeuds, états, ...). Un schéma de *saturation* revient souvent dans lequel on parcourt l'objet pour récupérer un sous-ensemble de ses éléments.
L'objet complexe est parcouru à partir d'un sous-ensemble de ses éléments (noeuds initiaux, états initiaux, ...) jusqu'à ce qu'il ait été complètement exploré.

```
(defun accessible-elements (elements new-elements object)
  (let* ((next-elements (append elements new-elements))
         (next-new-elements (set-difference (next-elements new-elements object) next-elements)))
    (if (endp next-new-elements)
        next-elements
        (accessible-elements next-elements next-new-elements object))))
```

Ce schéma est paramétrable par la fonction `next-elements`. Par exemple, pour les graphes on pourrait avoir :

```
(defun accessible-nodes (nodes new-nodes graph)
  (let* ((next-nodes (append nodes new-nodes))
         (next-new-nodes (set-difference (next-nodes new-nodes graph) next-nodes)))
    (if (endp next-new-nodes)
        next-nodes
        (accessible-nodes next-nodes next-new-nodes graph))))
```

et pour des automates :

```
(defun accessible-states (states new-states transitions)
  (let* ((next-states (append states new-states))
         (next-new-states (set-difference (next-states new-states transitions) next-states)))
    (if (endp next-new-states)
        next-states
        (accessible-states next-states next-new-states transitions))))
```

Définir une macro `def-saturation (name next-fun)` telle que les appels
– `(def-saturation accessible-nodes next-nodes)` et
– `(def-saturation accessible-states next-states)`
produisent du code équivalent aux deux `defun` ci-dessus.

Exemple : un graphe est représenté par une liste de ses noeuds ; chaque noeud est représenté par une liste contenant son numéro et la liste des sommets qui lui sont adjacents.

2. On pourra utiliser un appel à la fonction `member-sort` de la page 1.

```
CL-USER> (defvar *graph* '((1 (2 3)) (2 (3 4)) (3 (4)) (4 ()))
   "a graph with 4 nodes and 5 edges")
*GRAPH*
CL-USER> (defun next-nodes (nodes graph)
           "list of adjacent nodes of NODES in the GRAPH"
           (reduce #'union
                   (mapcar
                    (lambda (node) (cadr (assoc node graph)))
                    nodes)))
NEXT-NODES
CL-USER> (accessible-nodes '() '(1) *graph*)
(1 3 2 4)
CL-USER> (accessible-nodes '() '(3) *graph*)
(3 4)
CL-USER> (accessible-nodes '() '(4) *graph*)
(4)
```

FIN

Annexe

```
(defgeneric equality-fun (set) (:documentation "test to compare elements of SET"))
(defgeneric elements (set) (:documentation "list of the elements of SET"))
(defgeneric member-set (e set) (:documentation "does E belong to SET"))
(defgeneric make-set (elements &optional equality-fun) (:documentation "set creation"))
(defgeneric make-set-from-list (elements &optional equality-fun)
  (:documentation "set creation from the elements of the list ELEMENTS,
                   comparable with EQUALITY-FUN"))
(defgeneric adjoin-set (e set)
  (:documentation "set with additional element E if E was not present in SET"))
(defgeneric union-set (set1 set2) (:documentation "SET1 union SET2"))
(defgeneric intersection-set (set1 set2) (:documentation "SET1 intersection SET2"))
(defgeneric difference-set (set1 set2) (:documentation "SET1 difference SET2"))

(defclass my-set ()
  ((equality-fun :initarg :equality-fun :reader equality-fun)
   (elements :type 'list :initarg :elements :reader elements))
  (:documentation "class for sets"))

(defmethod make-set (elements &optional (equality-fun #'eql))
  (make-instance 'my-set :elements elements :equality-fun equality-fun))

(defmethod make-set-from-list (elements &key (equality-fun #'eql))
  (make-set (remove-duplicates elements :test equality-fun) equality-fun))

(defmethod member-set (element (set my-set))
  (member element (elements set) :test (equality-fun set)))

(defmethod adjoin-set (element (set my-set))
  (let ((equality-fun (equality-fun set)))
    (make-set (adjoin element (elements set) :test equality-fun) equality-fun)))

(defmethod union-set ((set1 my-set) (set2 my-set))
  (let ((equality-fun (equality-fun set1)))
    (make-set (union set1 set2 :test equality-fun) equality-fun)))

(defmethod intersection-set ((set1 my-set) (set2 my-set))
  (let ((equality-fun (equality-fun set1)))
    (make-set (intersection (elements set1) (elements set2)
                            :test equality-fun) equality-fun)))
```