ANNÉE UNIVERSITAIRE 2006/2007
2IÈME SESSION DE PRINTEMPS

Parcours :	INF6 / MAI6 / MIAGE6
Code UE :	INF207
Épreuve :	Programmation Fonctionnelle et Symbolique
Date :	Juin 2007
Heure :	8h30
Durée :	1h30
Documents :	autorisés
Épreuve de	Mme Irène Durand et Mr Robert Strandh

Le barême est donné à titre **indicatif**. Le sujet comporte 3 pages, les annexes 4 pages.

Exercice 1 *(2pts)*

Soit la spécification de la fonction `find-if` issue de la HyperSpec et donnée en Annexe. Que retournent les appels suivants ?

1. `(find-if #'evenp '(1 2 3 4 5) :from-end t)`
2.
```
(find-if (lambda (s) (> (length s) 3))
         '(("rouge" . "red") ("vert" . "green") ("jaune" . "yellow"))
         :key #'cdr)
```

Remarque : si l'argument mot-clé `:key f` est présent, le prédicat passé en premier argument de `find-if` est appliqué à `(f e)` au lieu de `e` pour chaque `e` considéré et élément de la liste passée en deuxième argument. Un exemple d'utilisation du mot-clé `:key` (fonctionnant dans le même esprit que celui de `find-if`) est présent dans la spécification de la fonction `sort` aussi donnée en Annexe.

Exercice 2 *(4pts)*

Soit les macros `none` et `once` définies ci-dessous :

```
(defmacro none (&body expressions)
   `(not (or ,@expressions)))

(defmacro once (&body expressions)
  (and
   expressions
   `(if ,(car expressions)
        (none ,@(cdr expressions))
        (once ,@(cdr expressions)))))
```

1. Que retourne l'appel `(macroexpand-1 '(none (eq 'a 'a) (zerop 3)))` ?
2. Que retourne l'appel `(none (eq 'a 'a) (zerop 3))` ?
3. Que retourne l'appel `(macroexpand-1 '(once (eq 'a 'a) (zerop 3)))` ?
4. Que retourne l'appel `(once (eq 'a 'a) (zerop 3))` ?

Exercice 3 *(7pts)*

On utilise une liste de paires pointées pour représenter les multiplicités des éléments d'une liste. Par exemple la liste de paires `((2 . 3) (3 . 0) (4 . 1))` représente les multiplicités de la liste `(3 0 0 1 3 1 0 1 1)` mais aussi de la liste `(0 0 0 1 1 1 1 3 3)`. Pour simplifier, on suppose que les éléments sont des entiers.

1. Écrire une fonction `check-multiplicity (pmul l)` qui retourne vrai si la liste de paires d'entiers `pmul` est bien une liste de multiplicités pour la liste d'entiers `l` et faux sinon.
2. Écrire une fonction `multiplicity (l)` qui retourne une liste de multiplicités pour la liste `l`. On ne demande pas que la liste des multiplicités soit ordonnée.
   `((2 . 3) (3 . 0) (4 . 1))` est équivalent à `((2 . 3) (4 . 1) (3 . 0))` ou à `...`
3. Compléter l'appel suivant pour obtenir une expression qui trie une liste de multiplicités en fonction du nombre d'éléments.

```
CL-USER> (... '((3 . 0) (4 . 1) (2 . 3)) ...)
((2 . 3) (3 . 0) (4 . 1))
```

Exercice 4 *(7pts)*

Un éditeur de figures utilise une structure de données complexe entièrement accessible à partir d'un objet de type `buffer`. Un tel objet contient l'ensemble des objets manipulés par l'éditeur.

```
(defclass buffer (...)
  ((modified :initform nil :accessor modified)
   ...))
```

Le créneau `modified` permet de déterminer si le buffer a été modifié depuis la dernière sauvegarde. Il doit prendre la valeur `NIL` après chaque sauvegarde et la valeur `T` dès que la méthode de modification d'un créneau à sauvegarder est appelée.
La variable spéciale `*current-buffer*` contient à tout moment le buffer courant. Lors de la sauvegarde du buffer sur disque, les créneaux contenant des valeurs de base sont sauvegardés tandis que ceux qui peuvent être calculés ne nécessitent pas de sauvegarde.
Par exemple, l'éditeur contient la classe `circle` ainsi définie :

```
(defclass circle (form)
  ((center :initarg :center :accessor center)
   (radius :initarg :radius :accessor radius)
   (area :initform nil))
  (:documentation "class for circles"))
```

Les créneaux `center` et `radius` doivent être sauvegardés, tandis que le créneau `area` qui peut être calculé à partir du créneau `radius` ne nécessite pas de sauvegarde.

1. (3p) Rajouter des méthodes permettant l'affectation du créneau `modified` du buffer courant à `T` à chaque fois que l'un des créneaux `center` ou `radius` d'un cercle est modifié par le biais de son accesseur.
   On rappelle que l'argument `:accessor center` du créneau `center` (par exemple) définit implicitement (entre autres) la méthode :

```
(defmethod (setf center) (val (circle circle))
  (setf (slot-value circle 'center) val))
```

(4p) Si la technique de la question précédente était généralisée à l'ensemble des classes d'un éditeur de figures, il en résulterait un très grand nombre de méthodes très similaires. Pour éviter cette duplication de code, on souhaiterait pouvoir écrire quelque chose du type :

```
(define-saved-class circle (form)
  ((center :initarg :center :accessor center :save)
   (radius :initarg :radius :accessor radius :save)
   (area :initform nil))
  (:documentation "a class for circles"))
```

pour générer à la fois la définition de la classe et les méthodes définies par la question précédente pour les créneaux à sauvegarder.
Remarquer l'utilisation du mot-clé `:save` pour indiquer les créneaux à sauvegarder.
Écrire la macro `define-saved-class` effectuant ces définitions.

Exemple :

```
CL-USER> (macroexpand-1
           '(define-saved-class circle (form)
             ((center :initarg :center :accessor center :save)
              (radius :initarg :radius :accessor radius :save)
              (area :initform nil))
             (:documentation "class for circles")))
(PROGN
  (DEFCLASS CIRCLE (FORM)
    ((CENTER :INITARG :CENTER :ACCESSOR CENTER)
     (RADIUS :INITARG :RADIUS :ACCESSOR RADIUS)
     (AREA :INITFORM NIL))
    (:DOCUMENTATION "class for circles"))
  (DEFMETHOD ... CENTER ...)
  (DEFMETHOD ... RADIUS ...))
T
```

Remarque : le nombre de méthodes produites par la macro doit être égal au nombre de créneaux possédant l'attribut `:save`.