ANNÉE UNIVERSITAIRE 2011/2012
2IÈME SESSION D'AUTOMNE

Parcours : IN501
Code UE : IN5011
Épreuve : Programmation 3
Date : Lundi 04 juin 2012
Heure : 8h30
Durée : 1h30
Documents : autorisés
Épreuve de Mme Irène Durand

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

On représente un monôme ax^n par une paire pointée `(a . n)` où `a` est un nombre réel et `n` un entier.

Exercice 1 *(10pts)*

1. Écrire une fonction `coeff(monome)` qui retourne le coefficient `a` du monôme `(a . n)`.
2. Écrire une fonction `degre(monome)` qui retourne le degre `n` du monôme `(a . n)`.
3. Écrire une fonction `monome+(m1 m2)` qui étant donnés deux monômes `m1` et `m2` de même degré, retourne un seul monôme correspondant à la somme des deux monômes.

```
CL-USER> (degre '(2 . 3))
3
CL-USER> (coeff '(2 . 3))
2
CL-USER> (monome+ '(2 . 3) '(5 . 3))
(7 . 3)
```

4. Écrire une fonction `monomes+(monomes)` qui étant donnée une liste de monômes de degrés identiques $(a_1x^n, a_2x^n, \ldots a_kx^n)$ retourne un seul monôme correspondant à la somme des monômes soit $(\sum_{i=1}^{k} a_i)x^n$.

```
CL-USER> (monomes+ '((2 . 3) (4 . 3) (1 . 3)))
(7 . 3)
```

5. Écrire une fonction non destructive `sort-monomes(monomes)` qui retourne la liste des monômes triés dans l'ordre des degrés croissants.

```
CL-USER> (sort-monomes '((7 . 3) (7 . 4) (5 . 1)))
((5 . 1) (7 . 3) (7 . 4))
```

6. Implémenter la fonction `groupe-degres(monomes)` qui prend en paramètre une liste de monômes et retourne la liste des monômes additionnés par degré.

On pourra s'aider (mais ça n'est pas obligatoire) de la fonction `partition(pred l)` donnée dans l'annexe 1.

```
CL-USER> (groupe-degres '((2 . 3) (3 . 2) (4 . 3) (4 . 2) (5 . 1)(1 . 3)))
((7 . 3) (7 . 2) (5 . 1))
```

Exercice 2 *(4pts)*

1. Écrire une fonction `list-monome(monome)` qui étant donnée un monôme `(a . n)` retourne une liste dont le premier élément est le symbole `*`, le deuxième élément est le nombre `a` et les `n` éléments suivants le symbole `X`.

```
CL-USER> (list-monome '(3 . 4))
(* 3 X X X X)
```

2. Écrire une macro `defpoly(nom monomes)` qui étant donnée une liste non vide de monômes définit la fonction de nom `nom` qui à un réel `x` associe la valeur en `x` du polynôme associé à la liste de monômes.

```
CL-USER> (macroexpand-1 '(defpoly poly ((5 . 0) (3 . 2) (4 . 3))))
(DEFUN POLY (X) (+ (* 5) (* 3 X X) (* 4 X X X)))
T
CL-USER> (defpoly poly ((5 . 0) (3 . 2) (4 . 3)))
POLY
CL-USER> (poly 2)
49
```

Exercice 3 *(6pts)*

Un *énumérateur* est un objet servant à énumérer les éléments d'une suite finie ou infinie. Étant donné un énumérateur `e`, chaque appel à `(call-enumerator e)`, retourne deux valeurs : la première valeur est la valeur énumérée (ou `NIL` s'il n'y a plus de valeur) et la deuxième valeur indique si une valeur a été énumérée. On peut considérer plusieurs types d'énumérateurs. Par exemple,

– un énumérateur constant qui retourne toujours la même valeur ;
– un énumérateur des éléments d'une liste ;
– un énumérateur des éléments d'une liste circulaire ;
– un énumérateur des éléments d'une suite définie inductivement ;
– ....

Un début d'implémentation est donné dans l'annexe 2.

1. Compléter le scénario de la figure 1, page 3.
2. Définir la classe `constant-enumerator` pour les énumérateurs constants.
3. Implémenter la méthode `next-element-p` pour les énumérateurs constants.
4. Implémenter la méthode `next-element` pour les énumérateurs constants.

```
CL-USER> (defparameter *c* (make-constant-enumerator 2))
*C*
CL-USER> (call-enumerator *c*)
2
T
CL-USER> (call-enumerator *c*)
2
T
```

5. Dessiner la hiérarchie des classes.

```
CL-USER> (defparameter *e* (make-list-enumerator '(do mi sol)))
                ;; Réponse 1
CL-USER> (call-enumerator *e*)
DO
T
CL-USER> (call-enumerator *e*)
                ;; Réponse 2

CL-USER> (call-enumerator *e*)
SOL
T
CL-USER> (call-enumerator *e*)
NIL
NIL
CL-USER> (call-enumerator *e*)
                ;; Réponse 3

CL-USER>
```

FIG. 1 – Scénario à compléter

6. Définir une fonction `make-circular-list-enumerator(l)` qui crée un énumérateur (infini) qui énumère de manière circulaire les éléments de `l`.

```
CL-USER> (defparameter *ecl* (make-circular-list-enumerator '(do mi sol)))
*ECL*
CL-USER> (call-enumerator *ecl*)
DO
T
CL-USER> (call-enumerator *ecl*)
MI
T
CL-USER> (call-enumerator *ecl*)
SOL
T
CL-USER> (call-enumerator *ecl*)
DO
T
```

FIN

Annexe 1

```
(defun partition (pred l)
  (let ((ok '())
        (nok '()))
    (dolist (e l (values (nreverse ok) (nreverse nok)))
      (if (funcall pred e)
          (push e ok)
          (push e nok)))))
```

Annexe 2

```
(defclass abstract-enumerator () ())

(defgeneric next-element-p (enumerator)
  (:documentation "T s'il existe un element suivant pour ENUMERATOR"))

(defgeneric next-element (enumerator)
  (:documentation "l'élément suivant de ENUMERATOR"))

(defgeneric call-enumerator (enumerator)
  (:documentation "retourne deux valeurs:
     en première valeur la valeur énumérée par l'énumérateur ENUMERATOR,
     en deuxième valeur T si il restait une valeur à énumérer, NIL sinon"))

(defgeneric init-enumerator (enumerator)
  (:documentation "remet l'énumérateur ENUMERATOR dans son état initial"))

(defgeneric copy-enumerator (enumerator))

(defmethod init-enumerator ((e abstract-enumerator)) e)

(defmethod call-enumerator ((e abstract-enumerator))
  (if (next-element-p e)
      (values (next-element e) t)
      (values nil nil)))

(defclass list-enumerator (abstract-enumerator)
  ((initial-list :type list :initarg :initial-list :reader initial-list)
   (current-list :type list :initarg :current-list :accessor current-list))
  (:documentation
   "enumerater des éléments d'une liste"))

(defmethod next-element-p ((le list-enumerator))
  (not (null (current-list le))))

(defmethod next-element ((le list-enumerator))
  (pop (current-list le)))

(defmethod init-enumerator :after ((e list-enumerator))
  (setf (current-list e) (initial-list e)))

(defun make-list-enumerator (l)
  (init-enumerator
   (make-instance 'list-enumerator :initial-list l)))
```