ANNÉE UNIVERSITAIRE 2007/2008
1ÈRE SESSION D'AUTOMNE

Parcours : CSB5
Code UE : INF353
Épreuve : Programmation Fonctionnelle et Symbolique
Date : Mercredi 19 décembre 2007
Heure : 8h30
Durée : 1h30
Documents : autorisés
Épreuve de Mme Irène Durand

Le barème est donné à titre **indicatif**. Le sujet comporte 3 pages et une annexe.

Exercice 1 *(2pts)*

Évaluer les expressions suivantes :

1. `(cons nil '(a b c d))`
2. `(cons '(a b c d) nil)`
3. `(list '(a b c d) nil)`
4. `(append '(a b) nil '(c d))`

Exercice 2 *(6pts)*

Soit la fonction `expand` suivante :

```
(defun expand (paires &optional (concatene nil))
  (if (endp paires)
      paires
      (let ((paire (car paires)))
        (funcall
         (if concatene #'append #'cons)
         (make-list (car paire) :initial-element (cdr paire))
         (expand (cdr paires) concatene)))))
```

1. Que retourne l'appel `(expand '((4 . 1) (3 . 2)))` ?
2. Que retourne l'appel `{(expand '((4 . 1) (3 . 2) (1 . 5)) t)` ?
3. Donnez une version non récursive de `expand`.

Exercice 3 *(4pts)*

Écrire une macro `verifier-propriete (expression propriete &optional stream)` où `expression` est une expression quelconque qui sera évaluée en un objet `objet` et `propriete` est le nom d'un prédicat qui peut s'appliquer à `objet`.
La macro écrit dans le flot `stream`

`objet est propriete`	si `(propriete objet)` retourne vrai
`objet n'est pas propriete`	sinon.

Exemples :

```
;; Propriétes sur les nombres entiers:
(defun pair (n) (evenp n))

(defun premier (n)
  "retourne vrai si N est un nombre premier"
  (when (> n 1)
    (loop
       for facteur from 2 to (isqrt n)
       when (zerop (mod n facteur))
       do (return nil)
       finally (return t))))
```

```
CL-USER> (verifier-propriete (+ 1 3) pair)
4 est PAIR.
NIL
CL-USER> (verifier-propriete (+ 1 3) premier)
4 n'est pas PREMIER.
NIL
```

Exercice 4 *(8pts)*

Le programme `language.lisp` donné en annexe contient le début d'une implémentation d'un paquetage destiné à manipuler des *langages* (ensembles de *mots* formés de *lettres*). Un langage pourra être représenté soit en extension par la suite de ses mots (entourée de `< >`), soit par une expression sur des langages pouvant contenir des opérateurs comme union, intersection, concatenation. Ces expressions sont écrites en notation préfixe :

$(op\ lang_1\ lang_2\ \ldots\ lang_n)$ avec $op \in \{\texttt{Union}, \texttt{Intersection}, \texttt{Concatenation}\}$.

On suppose implémentées les fonctions `make-letter (nom)` et `make-word` de sorte que les lettres aussi bien que les mots sont comparables avec la fonction `eq`. La fonction `make-word` crée un mot à partir d'un nombre quelconque de noms de lettres.

```
CL-USER> (make-letter "a")
{a}
CL-USER> (eq (make-letter "a") (make-letter "a"))
T
CL-USER> (setf *m* (make-word "a" "b"))
[{a}{b}]
CL-USER> (eq *m* (make-word "a" "b"))
T
CL-USER> (setf
           *language1*
           (make-language (make-word) (make-word "a" "a") (make-word "b" "a")))
< [] [{a}{a}] [{b}{a}] >
CL-USER> (setf *language2* (make-language (make-word "b" "a")
                                          (make-word "a" "b" "a")
                                          (make-word "b" "b")))
< [{b}{a}] [{a}{b}{a}] [{b}{b}] >
```

```
CL-USER> (setf *lexpr1* (make-union *language1* *language2* *m*))
(Union < [] [{a}{a}] [{b}{a}] > < [{b}{a}] [{a}{b}{a}] [{b}{b}] > [{a}{b}])
CL-USER> (setf *lexpr2* (make-intersection *language1* *language2*))
(Intersection < [] [{a}{a}] [{b}{a}] > < [{b}{a}] [{a}{b}{a}] [{b}{b}] >)
CL-USER> (setf *lexpr3*
               (make-concatenation (make-word "a" "a") (make-word "a" "b" "a")))
(Concatenation [{a}{a}] [{a}{b}{a}])
CL-USER> (setf *lexpr4* (make-concatenation *lexpr1* *lexpr2* *lexpr3*))
(Concatenation
 (Union < [] [{a}{a}] [{b}{a}] > < [{b}{a}] [{a}{b}{a}] [{b}{b}] > [{a}{b}])
 (Intersection < [] [{a}{a}] [{b}{a}] > < [{b}{a}] [{a}{b}{a}] [{b}{b}] >)
 (Concatenation [{a}{a}] [{a}{b}{a}]))
```

1. Dessiner la hiérarchie des classes définies dans le programme.
2. Implémenter la méthode `cardinality` pour la classe `word`.
3. Implémenter la méthode `cardinality` pour la classe `language`.
4. L'implémentation de l'opération `words` est-elle **complète** pour la classe `alanguage` (autrement dit, pour tout objet d'une classe dérivée de `alanguage`, y-a-t'il une méthode `words` qui s'applique) ? Justifiez la réponse.
5. Quelles méthodes `words` sont appelées lors de l'appel `(words *lexpr1*)` ?
6. Dans les classes `union-lexpr`, `intersection-lexpr` et `concatenation-lexpr`, pourquoi le créneau `operator` a-t'il l'attribut `:allocation :class` ?

Exemples :

```
CL-USER> (cardinality *language1*)
3
CL-USER> (cardinality *m*)
1
CL-USER> (words *m*)
([{a}{b}])
CL-USER> (words *lexpr1*)
([{a}{a}] [] [{b}{a}] [{a}{b}{a}] [{b}{b}])
CL-USER> (words *lexpr2*)
([{b}{a}])
CL-USER> (words *lexpr3*)
([{a}{a}{a}{b}{a}])
CL-USER> (words *lexpr4*)
([{a}{a}{b}{a}{a}{a}{a}{b}{a}] [{b}{a}{a}{a}{a}{b}{a}]
 [{b}{a}{b}{a}{a}{a}{a}{b}{a}] [{a}{b}{a}{b}{a}{a}{a}{a}{b}{a}]
 [{b}{b}{b}{a}{a}{a}{a}{b}{a}])
```

FIN