ANNÉE UNIVERSITAIRE 2010/2011
1ÈRE SESSION D'AUTOMNE

Parcours : IN501
Code UE : IN5011
Épreuve : Programmation 3
Date : Vendredi 16 décembre 2011
Heure : 14h00
Durée : 1h30
Documents : autorisés
Épreuve de Mme Irène Durand

Le barème est donné à titre **indicatif**. Le sujet comporte 2 pages plus deux annexes.

Exercice 1 *(5pts)*

On manipule des expressions arithmétiques en notation préfixe. Une *expression arithmétique* est

- soit une *variable* (représentée par un symbole `Lisp`),
- soit un *nombre* (représenté par un nombre),
- soit une *expression composée* représentée par une liste `(op ex1 ex2 ... exn)` telle que `op` est un symbole d'opération et chaque `exi`, une expression.

Les opérations possibles sont l'addition et la multiplication (n-aires, associatives et commutatives), l'exponentiation (binaire) représentés respectivement par les symboles `Lisp +, *` et `exp`.

Exemples d'expressions :

```
2     (+ x y)       (+ x 3 (* y 4) 2 z (+ x 1))
x     (exp x 2)     (+ 2 (exp x 3) (* x y z))
```

Soit la fonction `flatten-op (op expr)` donnée dans l'Annexe 1 page 3.

1. Compléter le scénario suivant :

```
CL-USER> (flatten-op '+ '(+ (+ x 3) (+ y z)))
                                                ;; Réponse 1
CL-USER> (flatten-op '* '(* (exp x 3) (* y z)))
                                                ;; Réponse 2
```

2. Que fait la fonction `flatten-op` ?
3. Généraliser la fonction `flatten-op` de manière à ce qu'elle puisse traiter plusieurs opérateurs associatifs en parallèle. Exemples :

```
CL-USER> (flatten-ops '(+ *) '(+ (+ x 3) (+ (* (* 3 y) (* 4 z) (* 5 u)))))
(+ X 3 (* 3 Y 4 Z 5 U))
CL-USER> (flatten-ops '(+) '(+ (+ x 3) (+ (* (* 3 y) (* 4 z) (* 5 u)))))
(+ X 3 (* (* 3 Y) (* 4 Z) (* 5 U)))
CL-USER> (flatten-ops '(*) '(+ (+ x 3) (+ (* (* 3 y) (* 4 z) (* 5 u)))))
(+ (+ X 3) (+ (* 3 Y 4 Z 5 U)))
```

Exercice 2 *(6pts)*
Soit le début d'implémentation de la théorie des langages formels donnée dans l'Annexe 2 page 4. Une *lettre* est une instance de la classe `letter`. Un *alphabet* est un ensemble de lettres et une instance de la classe `alphabet`. Un *mot* est une suite de lettres et une instance de la classe `word`. L'opération `alphabet (objet)` s'applique à n'importe quel objet contenant des lettres et retourne l'alphabet des lettres contenues dans l'objet.

1. Définir la classe `word` pour les mots.
2. Implémenter l'opération `make-word (letters)`.
3. Implémenter l'opération `word= (word1 word2)`.
4. Implémenter l'opération `alphabet` pour les mots.
5. Avant de créer un mot avec l'opération `make-word (letters)`, on voudrait vérifier que toutes les lettres de la liste `letters` sont bien de type `letter`. Proposer une solution qui ne modifie pas le code existant.

Exercice 3 *(4pts)*
Soit la macro `numerical-if` définie ci-dessous :

```
(defmacro numerical-if (expr pos zero neg)
  `(let ((g ,expr))
     (cond ((plusp g) ,pos)
       ((zerop g) ,zero)
       (t ,neg)))))
```

1. Que retourne l'expression suivante ? `(macroexpand-1 '(numerical-if n e1 e2 e3))`
2. Quel problème peut poser cette macro ?
3. Donner une nouvelle version qui résout ce problème.

Exercice 4 *(5pts)*

1. Écrire une Macro `condv (var defaut &rest lexpr)` où `var` est un symbole, `defaut` une expression quelconque, `lexpr` un nombre quelconque d'expressions `e1 e2 ... en` et qui produit une instruction `cond` contenant `n` clauses telles que la ième est de la forme `((= var i) ei)` et la clause par défaut `(t defaut)`. Exemple :

   ```
   CL-USER> (macroexpand-1 `(condv k def e1 e2 e3 e4))
   (COND ((= K 0) E1) ((= K 1) E2) ((= K 2) E3) ((= K 3) E4) (T DEF))
   T
   ```

2. Compléter le scénario suivant :

   ```
   CL-USER> (setf *x* 1)
                                                ;; Réponse 1
   CL-USER> (condv *x* 'defaut (+ *x* 1) 12 'a)
                                                ;; Réponse 2
   CL-USER> (setf *x* 4)
                                                ;; Réponse 3
   CL-USER> (condv *x* 'defaut (+ *x* 1) 12 'a)
                                                ;; réponse 4
   ```

FIN

Annexe 1

```
(defun flatten-op (op expr)
  (if (atom expr)
      expr
      (let ((args (mapcar (lambda (arg) (flatten-op op arg)) (cdr expr)))
            (op-expr (car expr)))
        (cons op-expr
              (if (eq op-expr op)
                  (let ((newargs '()))
                    (dolist (arg args (nreverse newargs))
                      (if (or (atom arg) (not (eq (car arg) op)))
                          (push arg newargs)
                          (dolist (e (cdr arg))
                            (push e newargs)))))
                  args)))))
```

Annexe 2

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Interface
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defgeneric alphabet (object)
  (:documentation "alphabet of letters appearing in OBJECT"))
(defgeneric make-letter (char)
  (:documentation "new letter named according to CHAR"))
(defgeneric make-word (letters)
  (:documentation "new word with letters LETTERS"))
(defgeneric letterp (object) (:documentation "type predicate por letter"))
(defgeneric letter= (letter1 letter2) (:documentation "equality of letters"))
(defgeneric letter< (letter1 letter2) (:documentation "ordering of letters"))
(defgeneric word= (word1 word2) (:documentation "equality of words"))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Implémentation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defclass letter () ((lchar :reader lchar :initarg :lchar)))

(defmethod make-letter ((lchar character))
  (make-instance 'letter :lchar lchar))

(defmethod print-object ((letter letter) stream)
  (format stream "~C" (lchar letter)))

(defmethod letterp (object) (typep object 'letter))

(defmethod letter= ((letter1 letter) (letter2 letter))
  (char= (lchar letter1) (lchar letter2)))

(defmethod letter< ((letter1 letter) (letter2 letter))
  (char< (lchar letter1) (lchar letter2)))

(defclass alphabet ()
  ((letters :initform '() :initarg :letters :reader letters)))

(defmethod print-object ((alphabet alphabet) stream)
  (format stream "<")
  (let ((letters (letters alphabet)))
    (when letters
      (dolist (letter (butlast letters))
        (format stream "~A," letter))
      (format stream "~A" (car (last letters)))))
  (format stream ">"))
```

```
(defmethod make-alphabet ((letters list) &key (clean t) (sort t))
  (when clean
    (setf letters (remove-duplicates letters :test #'letter=)))
  (when sort
    (setf letters (sort (if clean letters (copy-list letters))
                        #'letter<)))
  (make-instance 'alphabet :letters letters))

(defmethod alphabet ((letter letter))
  (make-alphabet (list letter) :clean nil :sort nil))

(defmethod alphabet ((alphabet alphabet))
  alphabet)

(defclass word ...

(defmethod print-object ((word word) stream)
  (format stream "[")
  (dolist (letter (word-letters word))
    (format stream "~A" letter))
  (format stream "]"))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Exemples
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
CL-USER> (setf *la* (make-letter #\a))
a
CL-USER> (setf *lb* (make-letter #\b))
b
CL-USER> (setf *lc* (make-letter #\c))
c
CL-USER> (setf *waaba* (make-word (list *la* *la* *lb* *la*)))
[aaba]
CL-USER> (setf *wcabb* (make-word (list *lc* *la* *lb* *lb*)))
[cabb]
CL-USER> (alphabet *la*)
<a>
CL-USER> (alphabet *waaba*)
<a,b,c>
CL-USER> (word= *waaba* (make-word (list *la* *la* *lb* *la*)))
T
```