ANNÉE UNIVERSITAIRE 2008/2009
2IÈME SESSION DE PRINTEMPS

Parcours : CSB6
Code UE : INF356
Épreuve : Projet de Programmation 3
Date : juin 2009
Heure : h
Durée : 1h30
Documents : autorisés
Épreuve de Mme Irène Durand

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

Exercice 1 *(6pts)*

Soit le programme donné en annexe Page 3. On suppose que

1. deux fichiers `f1.txt` et `f2.txt` se trouvent dans le répertoire courant,
2. le fichier `f1.txt` contient uniquement la ligne `2 23 1 45` ,
3. le fichier `f2.txt` contient uniquement la ligne `12 23 rr 45` .

Qu'imprime et que retourne l'appel `(log-analyzer "f1.txt" "f2.txt")` ?

Exercice 2 *(2pts)*

Écrire une fonction `display-sequence` qui écrit le contenu d'une séquence dans un flot avec un argument mot-clé `:sep` (ayant pour valeur par défaut un espace) permettant de séparer les arguments. Exemples :

```
TERM> (display-sequence '(a b c) *standard-output*)
A B C
NIL
TERM> (display-sequence '(a b c) *standard-output* :sep ", ")
A, B, C
NIL
```

Exercice 3 *(12pts)*

Soit $\mathcal{F}$, un ensemble de *symboles*[1] munis d'une *arité* (aussi appelé *alphabet gradué*). Soit $\mathcal{V}$, un ensemble dénombrable de *variables*. On considère $\mathcal{T}(\mathcal{F},\mathcal{V})$, l'ensemble des *termes* construits à partir de $\mathcal{F}$ et $\mathcal{V}$.
Par exemple, avec $\mathcal{F} = \{0, S, +\}$, $\mathsf{arity}(0) = 0$, $\mathsf{arity}(S) = 1$, $\mathsf{arity}(+) = 2$ et $\mathcal{V} = \{x, y, z, \ldots\}$, on peut contruire les termes $x, 0, S(0), S(S(0)), +(0, x), +(x, S(y))$ appartenant à $\mathcal{T}(\mathcal{F},\mathcal{V})$. Les symboles 0 et S permettent de représenter les entiers naturels (0 représente 0, $S(0)$ représente 1, $S(S(0))$ représente 2, ...) et le symbole binaire + permet de représenter l'addition sur les entiers.
On souhaite implémenter un module dédié aux termes.

[1] Dans cet exercice, le mot *symbole* est utilisé au sens de la théorie des langages et ne désigne donc pas un symbole `Lisp`

1. Définir un paquetage `term` utilisant uniquement le paquetage `:common-lisp`.
2. Définir une classe `named-object` contenant un créneau `name` permettant de définir des objets nommés par une chaîne de caractères.
3. Implémenter une version spécialisée de l'opération préféfinie `print-object` pour les objets de la classe `named-object` de manière à ce que le nom de l'objet soit affiché.

```
TERM> (make-instance 'named-object :name "Juin")
Juin
```

4. Un symbole a un nom et une arité. Définir la classe `arity-symbol` des symboles munis d'une arité.
5. Définir une fonction `make-arity-symbol` qui prend en paramètre un nom de symbole et optionnellement une arité (par défaut 0) et qui fabrique le symbole correspondant.
6. Définir une classe abstraite sans créneau `general-term` dont le rôle est de regrouper tous les types de termes (y compris les variables).
7. Une variable est un terme qui a un nom. Définir une classe concrète `var` permettant de représenter les variables.
8. Un terme (qui n'est pas une variable) a un symbole à sa racine et une liste d'arguments (éventuellement vide si le terme est d'arité nulle). Définir une classe `term` permettant de représenter les termes non variable avec les accesseurs `root` et `args` pour accéder respectivement au symbole racine et à la liste des arguments du terme.
9. Dessiner la hiérarchie des classes obtenue.
10. Proposer une solution pour véfifier automatiquement que les termes créés ont un nombre d'arguments compatible avec l'arité du symbole racine.
11. Écrire une fonction `make-term` prenant un premier argument obligatoire qui est le symbole racine du terme à construire et un second argument optionnel qui correspond à la liste des arguments du terme.
12. Implémenter une version spécialisée de `print-object` pour les objets de la classe `term` de manière à ce qu'un terme s'affiche sous la forme mathématique standard. Exemples :

```
TERM> (setf *s-zero* (make-arity-symbol "0"))
0
TERM> (setf *s-s* (make-arity-symbol "S" 1))
S
TERM> (setf *s-plus* (make-arity-symbol "+" 2))
+
TERM> (setf *v-x* (make-instance 'var :name "x"))
x
TERM> (make-term
       *s-plus*
       (list
        (make-term *s-zero*)
        *v-x*))
+(0,x)
TERM>
```

Remarque : on pourra se servir de la fonction `display-sequence` définie dans l'exercice précédent.

13. La taille d'un terme est le nombre de symboles (non variables) qui le composent. Ainsi, les variables sont de taille 0. L'opération `term-size` calcule la taille d'un terme. Implémenter cette opération pour tous les termes.

Annexe

```
(defun treat-entry (entry)
  (format t "Treating entry ~A~%" entry)
  entry)

(defun entry-ok-p (entry) (integerp entry))

(define-condition bad-entry-condition (condition)
  ((contents :initarg :contents :reader contents)))

(defun parse-entry (entry)
  (if (entry-ok-p entry)
      (treat-entry entry)
      (signal 'bad-entry-condition :contents entry)))

(defun analyze-log (file)
  (with-open-file (in file :direction :input)
    (handler-case (parse-log-stream in)
      (bad-entry-condition (c)
        (declare (ignore c))
        (format t "captured by analyze-log~%")))))

(defun parse-log-stream (stream)
  (loop
     for entry = (read stream nil nil)
     while entry
     for correct-entry = (handler-case (parse-entry entry)
                           (bad-entry-condition (c)
                             (format
                              *error-output*
                              "captured by parse-log-stream ~% bad-entry ~A~%"
                              (contents c))))
     when correct-entry collect correct-entry))

(defun log-analyzer (&rest logs)
  (loop for log in logs
     collect (cons (analyze-log log) log)))
```