Université Bordeaux 1 Licence S&T Mention Informatique S6 2007/2008

UE INF356 **Projet de Programmation 3**

Devoir surveillé

Tous documents autorisés. Mardi 25 Mars 2008
Durée : 1h20.
Le barème est donné à titre **indicatif**.

Exercice 1 *(4pts)*

L'archive d'un système Lisp nommé `clones` et décrit par le fichier `clones.asd` se trouve à l'url `http://www.ltn.lv/~jonis/clones-clim.tgz`. On dispose du système `ASDF` mais pas du système `asdf-install`.

1. Donner la liste des opérations permettant d'installer `clones` localement sur son propre compte.
2. Quelle expression `SBCL` permet de charger (le cas échéant compiler) le système `clones` dans votre session Lisp.

Si on dispose du système `asdf-install`,

3. quelle expresions peut-on taper dans une session Lisp pour installer automatiquement le système `clones`.

Exercice 2 *(7pts)*

On considère un jeu qui se déroule dans les salles d'un château et dont un début d'implémentation est donné en page 4.

1. Quel est l'effet et la valeur retournée par l'appel suivant ?

```
CL-USER> (ajouter-salle "Arsenal" 'salle)
```

Supposons qu'on ait aussi effectué l'appel

```
CL-USER> (ajouter-salle "Vestibule" 'salle)
```

2. Quel est l'effet et la valeur retournée par l'appel suivant ?

```
CL-USER> (joindre-salles "Vestibule" "Arsenal")
```

3. puis par l'appel

```
CL-USER> (joindre-salles "Arsenal" "Vestibule")
```

Les questions suivantes sont indépendantes des précédentes.
Des acteurs vont maintenant évoluer dans le château.

4. Définir une classe `acteur` : un acteur est défini par son *nom* et la salle dans laquelle il se trouve ou nil s'il n'est dans aucune salle.
5. Implémenter l'opération `print-object` pour les acteurs ; on imprimera le nom de l'acteur et le cas échéant le nom de la salle dans laquelle il se trouve. Exemple :

```
CL-USER> (defparameter *heros* (make-instance 'acteur :nom "Fergus"))
*HEROS*
CL-USER> *heros*
Acteur: Fergus
```

Voir plus loin un exemple quand l'acteur est situé dans une salle.
Soit l'opération `bouger-de-a` définie par la fonction générique `(defgeneric bouger-de-a (acteur source des`
qui déplace l'acteur de la salle source vers la salle destination. `source` ou `destination` peuvent valoir `nil` cela signifie que l'acteur rentre ou sort du château. Exemples :

```
CL-USER> (bouger-de-a *heros* nil (obtenir-salle "Vestibule"))
Salle Vestibule occupee par (Fergus)
CL-USER> (bouger-de-a *heros*
                      (obtenir-salle "Vestibule")
                      (obtenir-salle "Arsenal"))
Salle Arsenal occupee par (Fergus)
CL-USER> (obtenir-salle "Vestibule")
Salle Vestibule
```

6. À l'aide de méthodes secondaires `:after`, compléter l'implémentation de `bouger-de-a` de manière à ce que l'acteur soit supprimé de sa source (quand c'est bien une salle et pas `nil`) et rajouté à sa destination (quand c'est bien une salle et pas `nil`).

Exercice 3 *(5pts)*
On se place dans le cadre du programme de l'exercice précédent mais cet exercice est indépendant du précédent.
Pour joindre deux salles par une sortie, on voudrait pouvoir écrire

```
CL-USER> ["Vestibule" "Arsenal"]
Sortie 1: vers Salle Arsenal
```

au lieu de

```
CL-USER> (joindre-salles "Vestibule" "Arsenal")
Sortie 1: vers Salle Arsenal
```

Proposer une solution utilisant les *read-macros*.

Exercice 4 *(4pts)*
Vous avez un fichier dont le nom est "`donnees`" contenant le texte suivant (commençant dans la première colonne) :

```
ceci est
(un text (je pense)
) sans doute
```

Il y a donc trois lignes de texte, et aucune ligne ne contient des caractères blancs après le dernier caractère non blanc.

1. Montrer comment créer un flot permettant la lecture de ce fichier.
2. Quel est le résultat (valeur de retour, effet) de chaque opération dans la suite suivante :

```
(read-line flot)
(read-char flot)
(read flot)
(read flot)
(read flot)
(read flot)
```

où 'flot' est une variable contenant le flot crée dans la partie 1 ?

```
(defvar *salles*
  (make-hash-table :test #'equal)
  "Table contenant les salles du chateau")

(defvar *compteur-de-sorties* 0
  "Pour numeroter automatiquement les sorties")

(defclass entite-nommee ()
  ((nom :initarg :nom :type string :reader nom :initform "sans nom")))

;; pour simplifier une sortie est unidirectionnelle.
(defclass sortie ()
  ((numero :reader numero :initform (incf *compteur-de-sorties*))
   (destination :reader destination :initarg :destination))
  (:documentation "sortie d'une salle vers la salle DESTINATION"))

(defmethod print-object ((sortie sortie) stream)
  (format stream "Sortie ~A: vers ~A" (numero sortie) (destination sortie)))

(defclass salle-de-base (entite-nommee)
  ((sorties :accessor sorties :initarg :sorties :initform nil)
   (occupants :accessor occupants :initarg :occupants :initform nil))
  (:documentation
   "salle de base de laquelle vont deriver tous les types de salles"))

(defmethod print-object ((salle salle-de-base) stream)
  (format stream "Salle ~A" (nom salle))
  (when (occupants salle)
    (format stream " occupee par ~A" (mapcar #'nom (occupants salle)))))

;; pour l'instant une salle ressemble a une salle de base.
(defclass salle (salle-de-base) ())

(defun ajouter-salle (nom classe &rest initargs)
  "ajouter une salle a la table des salles"
  (let ((salle (apply #'make-instance classe :nom nom initargs)))
    (setf (gethash nom *salles*) salle)
    salle))

(defun obtenir-salle (nom-ou-salle)
  "recuperer une salle de la table a partir de son nom"
  (if (typep nom-ou-salle 'salle-de-base)
      nom-ou-salle
      (let ((salle (gethash nom-ou-salle *salles*)))
        salle)))

(defun joindre-salles (source destination)
  "creer une sortie dans SOURCE vers DESTINATION"
  (let ((sortie
         (make-instance 'sortie :destination (obtenir-salle destination))))
    (push sortie
          (sorties (obtenir-salle source)))
    sortie))
```