Examen de langages enchassés

Nicolas Sceaux

1er mars 2001

Exercice 1

```
(defmacro with-buffering (window &body body)
  (let ((window-copy (gensym)))
    `(let ((,window-copy ,window))
       (setf ,window (create-pixmap ,window))
       (and ,@body
    (copy-pixmap ,window ,window-copy))
       (destroy-pixmap ,window)
       (setf ,window ,window-copy))))

(macroexpand '(with-buffering *window*
               (graphical-op1 *window*)
               (graphical-op2 *window*)))
```

En supposant que les fonctions `graphical-opi` retournent `nil` lorsque l'opération échoue : En cas d'échec, on n'exécutera pas le `copy-pixmap`, et par conséquent la variable passée en argument retrouvera finallement sa valeur initiale. Si au contraire tout se passe correctement, `copy-pixmap` est exécuté, la variable passée en argument aura pour valeur celle de la transformation réussie.

Exercice 2

On souhaite intégrer dans Common Lisp un langage enchassé pour SQL. On suppose que l'on dispose de primitives telles que `sql-connect`, `sql-choose-db`, `sql-close`, `sql-query`, etc.

Intéressons-nous tout d'abord aux requêtes de sélection. Elles sont de la forme (simplifiée) : `select fields from tables where conditions`. On introduit la construction : `select &rest selections &key from where`. Les sélections sont écrites ainsi : `field` ou `(table field)` ou `(table field "alias")` ou `(field "alias")`.

Exemple : la requête SQL

```
select aria.name, singer.name "singer_name"
  from aria, singer
  where aria.singer_id = singer.id
```

s'écrirait en lisp :

```
(select '(aria name) '(singer name "singer_name")
  :from '(aria singer)
  :where '(= '(aria singer_id)
             '(singer id)))
```

Un tel appel retournerait :

```
(("Piangero la sorte mia" "Barbara Schlick")
 ("Aure deh per pieta" "Jennifer Larmor"))
```

La difficulté principale réside dans la clause `where`. En effet, il s'agit d'expressions à transformer en expression SQL, où l'on souhaite par ailleurs faire intervenir des variables lisp par exemple.

```
(setf *limit* 25)
(select 'nom
  :from '(eleves)
  :where '(< 'age *limit*))
```

Il s'agira donc de n'évaluer lors de la génération de la requête SQL que les paramêtres non-quotés.

Par ailleurs, on définit les fonctions de mise à jour `insert-records`, `delete-records`, `update-records`, `create-table`, `drop-table`, etc. Exemple :

```
(insert-records :into 'aria
                :attributes '(name, singer_id)
                :values '("Al lempo dell'armi" 1))
```

On a alors besoin de quelques environnements `with-` pour utiliser facilement ces fonctions.

- une macro `with-results (&rest args query &rest body)`. Elle permettrait, pour chaque n-uplet de la réponse, d'exécuter le corps `body` ; le nom de chaque colonne serait précisé dans `args`, ce qui permettrait de faire référence à chaque champ du n-uplet dans le corps.

  Exemple :

  ```
  (with-results ((aria-name singer-name)
                 (select '(aria name) '(singer name "singer_name")
                   :from '(aria singer)
                   :where '(= '(aria singer_id)
                              '(singer id))))
                (format t "~A : ~A~%" singer-name aria-name))
  ```

- une macro `with-transaction (&rest body)`. Elle permettrait de commencer une transaction, effectuer des requêtes de mise à jour par exemple, et de finir par un `commit` en cas de succès, ou d'un `rollback` en cas d'échec.

Il serait plus aisé de ne considérer que des requêtes SQL constantes, et des constructeurs de requêtes a la manière de `format`

```
(with-results ((name age)
               (sql-format ``select nom, age from eleves where age>~A'' 25))
              (format t ``nom: ~A, age: ~A~%'' name age))
```