Auteur : R. Namyst
Enseignants : L. Bougé A. Bonhomme
Email : `luc.bouge,alice.bonhomme@ens-lyon.fr`

le 11/12/00

TD10 : Le langage LISP

L'objectif de ce TP est d'aborder le fonctionnement de l'interprète du langage LISP (`LISt Processing` Package), en particulier sa capacité â batir des programmes qui se modifient eux-même. LISP est un langage extrêmement répandu dans la communauté des chercheurs en Intelligence artificielle. L'interpréteur LISP s'invoque en tapant `lelisp`.

Pour la premièr partie, il est demandé de taper les exemples afin de voir les résultats ou erreurs retournés par l'interprète LISP (qui ne figure pas dans l'énoncé).

1 Prise en main

Comme son nom l'indique, le langage LISP a été principalement conçu dans le but de manipuler aisément (?) des listes. En fait, LISP manipule surtout des listes mais aussi d'autres d'objets. En première approximation, on peut considérer que LISP manipule deux types d'objets : les atomes et les listes. Un atome peut être un nombre ou un identificateur (au sens classique). Une liste est une suite parenthésée (éventuellement vide) d'objets LISP.

1.1 Évaluations simples

Un peu comme dans l'interprète CAML, chaque entrée clavier est évaluée et le résultat (sans indication de type toutefois) est affiché en réponse :

```
machine% lelisp
?; L'invite du LISP est le caractère '?'. Les commentaires
?; débutent par ';' et terminent par la fin de ligne
? 12; commencons par une évaluation simple
? 12 3; il est possible de lancer une séquence d'évaluations
? toto; essayons un identificateur
** eval: undefined variable: toto
```

Les objets les plus intéressants en LISP sont évidemment les listes, qui deviennent incontournables dès que l'on manipule des fonctions ou des programmes. L'*évaluation* d'une liste considère le premier élément comme une fonction qu'il s'agit d'appliquer à la suite de la liste. C'est de cette manière que l'on effectue un appel de fonction en LISP. Voici quelques exemples :

```
? (+ 1 2)
? (+ 1 2 3 4)
? (- 1 2 3 4)
```

On s'aperçoit ainsi que le langage LISP n'est pas fortement typé et que les listes peuvent contenir des éléments de nature differente.

Notons aussi que la règle précédente empêche d'évaluer des listes dont le premier élément n'est pas une fonction :

```
? (1 2 3); on ne peut pas "évaluer" une telle liste
** eval: undefined function: 1
? (); Il existe une exception pour la liste vide qui est considérée
; comme un atome
? nil; Il existe d'ailleurs une constante prédéfinie désignant ()
```

La plupart des fonctions LISP provoquent également le déclenchement du processus d'évaluation sur leurs arguments avant leur exécution. Cela n'était pas très visible sur les exemples précédents (l'évaluation de `1` retourne `1`...). En voici un autre faisant intervenir une liste contenant une sous-liste :

```
? (+ 1 (- 2 3))
```

1.2 Fonctions de base

Vous connaissez maintenant par cœur la sémantique des fonctions `cons`, `car` et `cdr` :

```
? (cons 1 nil)
? (cons 1 (cons 2 nil))
? (car (cons 1 nil))
? (cdr (cons 1 (cons 2 nil))); Essayez aussi: (cdr (1 2))
```

Il serait néanmoins souvent pratique d'utiliser des listes définies de manière littérale. Il nous faut alors éviter l'évaluation "automatique" de certains paramètres. Une fonction prédéfinie `quote` va nous y aider : elle a pour propriété de ne pas évaluer son paramètre (et donc de le retourner tel quel). Illustration :

```
? (quote (1 2))
? (quote (+ 1 2))
? '(1 2 3); "quote" est si utile qu'il existe une abbréviation: '
? (cons 1 '(2 (3 4) 5))
? (cons 1 2)
; on appelle cela une ``paire pointée''.
? '(1 . ())
? '(1 . (2 . 3))
? '((1 . (2 . ())) . ())
; Qu'en déduisez vous sur les listes et les paires pointées?
```

A votre avis, les fonctions `eval` et `quote` sont-elles commutatives ?

```
? (eval (quote (+ 1 2)))
? (quote (eval(+ 1 2)))
```

1.3 E/S

Dans ce TP, nous nous contenterons d'opérations d'entrée/sortie très simples :

```
? (print 'toto)
? (print '(1 2 3))
? (read); Lit sur l'entree standard un objet LISP, qui est retourné
; sans être évalué.
```

1.4 Tests

En LISP, certaines fonctions prédéfinies permettent d'effectuer des exécutions conditionnelles (`cond`, `if`, etc.). Par convention, la valeur NIL (ou `()`) est considérée comme "fausse" et toute autre valeur est considérée "vraie". Un atome prédéfini ("`t`") est cependant utilisé par tous les prédicats de base pour renvoyer la valeur "vrai" :

```
? (eq 1 (- 3 2))
? (eq 1 2)
? (eq '1 '1)
? (eq '(1) '(1))
;  Curieux non?
? (atom 1)
? (atom '(1))
? (null ())
```

Avec ces quelques prédicats, il est possible d'écrire des tests :

```
? (setq a 12)
? (if (< a 10) (print 'petit) (print 'grand))
```

Remarque : Pour utiliser plusieurs instructions dans la partie “alors” du `if`, on peut construire une séquence à l’aide de la fonction `progn` :

```
? (progn (print 'a) (print 'b) (print 'c))
```

La forme générale d’un test est donnée par la fonction `cond` :

```
? (cond ((eq a 1) (print 'un)) ((eq a 2) (print 'deux)) (t (print 'grand)))
```

1.5 Variables

La définition d’une variable en LISP s’effectue normalement au moyen de la fonction `set` :

```
? (set x 12)
? (set 'x 12)
? x
```

Pour éviter d’appeler explicitement la fonction “quote”, il existe une version plus pratique : `setq`

```
? (setq x 12)
? (setq l '(+ 1 2 3))
? (eq l l)
;??? Contradiction avec la situation observée auparavant?
```

On voit que l’on peut donc affecter à une variable une liste “évaluable”, et donc du code ! Pour évaluer de telles expressions, il existe une fonction `eval` (duale de `quote`) :

```
? l
? (eval l)
? (eval (cons '+ '(1 2)))
; C'est là une des clés du fonctionnement de l'interprète!
```

Les variables peuvent être, comme en CAML, définies uniquement de manière locale à une suite de déclarations. On utilise pour cela la fonction `let` :

```
? (let ((a 1) (b 2)) (+ a b))
```

2 Définition de fonctions

La définition d’une fonction simple en LISP s’écrit au moyen du mot-clé `DE` :

```
? (de f () (print 'hello))
? (f)
? (de fact (n) (if (eq n 0) 1 (* n (fac (- n 1)))))
? (fac 6)
```

Vous pouvez *tracer* les appels successifs de la fonction `fact` en utilisant la fonction `trace` :

```
? (trace fact)
= (fact)
? (fact 6)
fact ---> n=6
[...]
fact <--- 720
= 720
```

Essayez avec la fonction `fibonacci` ! Pour désactiver la trace, utilisez `untrace`.

3 Exercices sur les entiers

Écrire les fonctions suivantes :

- `square` qui renvoie le carré d’un entier.

- `foo` qui renvoie la liste des `n` premiers entiers dans l'ordre décroissant 0 non compris.
- `fact` qui calcule la factorielle pour un entier `n` supérieur ou égal à 0.
- `fib` qui renvoie le nième terme de la suite de fibonnacci.
- `bar` qui saisit au clavier deux entiers et renvoie le plus grand.

4 Manipulations de listes

Écrire les fonctions suivantes :

- `length` qui renvoie la longueur d'une liste.
- `last` qui renvoie le dernier élément d'une liste.
- `append` qui renvoie la concaténation de deux listes.
- `filter` qui élimine toutes les occurrences d'un atome dans une liste.
- `reverse` qui renvoie la liste dans l'ordre inverse. (NB : proposez plusieurs solutions !)
- `subst` qui remplace toutes les occurrences de `x` par `y` dans une liste.

5 Écriture d'un mini interprète LISP en LISP

5.1 Utilisation de `eval`

Écrivez une fonction `mytoplevel` qui saisisse inlassablement des expressions au clavier et qui affiche le résultat de leur évaluation. On arrêtera l'interprète si l'expression entrée est `(end)`. Que se passe-t-il sis vous modifiez la valeur d'une variable, par exemple par `setq` ?

5.2 Sans `eval`

Écrivez une fonction `myeval` capable (sans utiliser `eval`, bien sûr !) d'évaluer des listes de la forme `(op x1 x2)`. On utilisera les opérations `+`, `-`, `*` et `/`.

- Étendre ensuite au cas n-aire : `(op x1 ... xn)`.
- Ajoutez la gestion de la fonction `quote`.
- Ajouter des fonctions de manupilation de listes : `nil`, `car`, `cdr`, `cons`, etc.
- Ajouter des structures de contrôle : `if`, etc.
- Ajouter `read` et `print`.
- Modifiez `mytoplevel` pour utiliser `myeval` au lieu de `eval`.

5.3 Gestion des variables

Ajoutez la possibilité d'utiliser des variables dans votre interprète (toujours en n'utilisant pas `eval`). Une bonne idée est d'ajouter un paramètre d'*environnement* à votre fonction `myeval`...