Université Montpellier II — Master d'Informatique
UMINM 121 Evaluation des Langages Applicatifs — 2 heures

R. Ducournau – M. Lafourcade

9 janvier 2007

Documents autorisés : notes de cours, polys, pas de livre.
Les 3 parties se suivent. Notation globale sur 28.

1 Passage des paramètres en COMMON LISP

Le principe du passage des paramètres en COMMON LISP repose sur la notion de *paramètre positionnel obligatoire*. Chaque paramètre est lié à la valeur d'argument correspondant. L'ensemble des ces *liaisons* constitue un *environnement*. C'est une erreur si le nombre de paramètres diffère de celui des arguments.

Pour changer, on adoptera pour les environnements une structure de liste alternée propriété-valeur (à la place de la structure de liste d'association vue en cours).

Exemple : étant donnée la fonction `(defun foo (x y z) ...)`, l'appel `(foo '1 '2 (+ 1 2))` provoquera la création de l'environnement `(x 1 y 2 z 3)` (à l'ordre près).

Question 1

Avec cette nouvelle structure d'environnement, écrire la fonction `make-env` qui prend 3 paramètres, la liste des paramètres, la liste des valeurs d'arguments et l'environnement courant, et qui
- retourne l'environnement à construire,
- ou signale une exception, dans le cas où il y a trop ou pas assez d'arguments (distinguez les 2 cas, par les exceptions "`trop d'arguments`" et "`pas assez d'arguments`" !).

1. écrire une première version de `make-env` qui ne traite que les arguments positionnels ;
2. écrire une seconde version qui traite aussi le cas du mot clé `&rest`.
   Exemple : la fonction `(defun bar (x y z &rest w) ...)`, l'appel `(bar '1 '2 (+ 1 2) '4 '5)` provoquera la création de l'environnement `(x 1 y 2 z 3 w (4 5))` (à l'ordre près).

2 Passage des paramètres par destructuration

On se propose d'étudier une généralisation du mode habituel de passage des paramètres.

Ce mode habituel consiste à *apparier* la liste des paramètres avec la liste des arguments. La *destructuration* consiste à remplacer cet appariement de *listes plates* par un appariement d'*arbres binaires* :
- l'arbre des paramètres est un arbre binaire quelconque de cellules LISP, dont les feuilles sont soit `NIL`, soit des paramètres (symboles non constants) ;
- l'arbre des valeurs d'arguments est en fait la liste des valeurs d'arguments — chaque argument étant évalué de façon habituelle — mais cette liste plate et propre est interprétée comme un arbre binaire (chaque élément de la liste est lui-même une expression, c'est-à-dire un arbre, quelconque.

NB. Un arbre binaire ainsi considéré est constitué de cellules (les noeuds intérieurs de l'arbre) et d'atomes (les feuilles de l'arbre). Les fils d'un noeud intérieur sont référencés par les `car` et `cdr` de la cellule correspondante.

Dans l'arbre de paramètres, les paramètres sont positionnels et obligatoires, comme dans le passage de paramètres normal. Cela se traduit par le fait que
- à chaque cellule de l'arbre des paramètres doit correspondre une cellule de l'arbre des arguments ;
- une feuille `nil` de l'arbre des paramètres doit correspondre à une feuille `nil` de l'arbre des arguments.

– une feuille non `nil` (un paramètre) peut correspondre à n'importe quel sous-arbre.

La destructuration va donc permettre de définir la fonction `foo` qui précède, avec les résultats habituels.

Exemple : Soit la fonction `oof` suivante :

```
(defun oof (cle ((clex . valx) . al))
       ...)
```

Lors de l'appel `(oof 'x '((y . 1) (x . 2)))`, l'appariement entre l'arbre de paramètres `(cle ((clex . valx) . al))` et la liste de valeurs `(x ((y . 1) (x . 2)))` retourne `(cle x clex y valx 1 al ((x . 2)))` — à l'ordre près.

Les expressions `(oof 'x ())`, `(oof 'x 1)`, `(oof 'x '(1))` provoqueraient toutes une erreur car il n'est pas possible d'apparier `((clex . valx) . al)` avec la valeur de leur deuxième argument — `()`, `1` ou `(1)`.

Question 2

En dessinant les arbres à apparier et les appariements à réaliser

1. Montrer, sur l'exemple de la fonction `bar` ci-dessus, comment exprimer avec la destructuration le même passage de paramètres qu'avec le mot-clé `&rest`.
2. montrer comment se traduit, dans la destructuration, les 2 erreurs habituelles (mauvais nombre d'arguments), par exemple lors des appels `(foo '1 '2)` ou `(foo '1 '2 '3 '4)`.

Question 3

La fonction `makenv-destruct` prend, comme `make-env`, 3 arguments, l'arbre des paramètres, l'arbre des arguments, et l'environnement courant, et soit retourne l'environnement augmenté de l'appariement des nouveaux paramètres, soit signale une erreur.

1. Définir la fonction `makenv-destruct`. Il s'agit d'une récursion d'arbre normale (de même structure que `size-tree` ou `subst`, pour reprendre des exemples du cours) mais en parallèle sur les 2 arbres. Une récursion partiellement terminale est préférable car plus simple.
2. Quelle modification faut-il faire au méta-évaluateur du cours pour remplacer le passage de paramètres habituel par la destructuration ?

3 Matching

Le *matching* est une généralisation de la destructuration. Quatre points font la différence :

– l'échec du matching n'engendre pas une exception ;

– lorsqu'une feuille de l'arbre des paramètres est une constante autre que `nil`, l'appariement réussit si la valeur correspondante dans l'arbre des arguments est égale à la constante ;

– le même paramètre peut apparaître plusieurs fois dans l'arbre des paramètres : pour que le matching réussisse, il faut que le paramètre soit chaque fois associé à la même valeur ;

– le symbole _ représente une position vide dont l'appariement est indifférent.

Le *matching* de `((x . 1)(y (x)))` avec `((a . 1)(b (a)))` réussit et retourne `(x a y b)`.

Question 4

1. Par contre, si l'on remplace l'arbre des valeurs par `((a . 2)(b (a)))` ou par `((a . 1)(b (c)))`, l'appariement échoue : dessinez les arbres à apparier, montrer les appariements partiels qui réussissent et les causes de l'échec.
2. Définir la fonction `match`, similaire à `makenv-destruct` sauf qu'elle retourne le symbole `:fail` en cas d'échec et qu'elle applique les 2 contraintes précitées d'égalité avec les constantes ou entre plusieurs occurrences d'un même paramètre, ainsi que le traitement de _.

4 Filtrage

Le filtrage est une structure de contrôle basée sur le *matching* : la forme qui l'implémente est nommée `case-match`. Elle est similaire à un `case` (variante du `cond` dans laquelle la valeur d'une expression est comparée à des constantes), mais la comparaison s'effectue par *matching* et, lorsqu'elle réussit, les paramètres correspondants sont considérés comme liés, comme dans un `let` ou une lambda-expression. La syntaxe générale est :

```
(case-match expr
     (filtre-1 . progn-1)
     (filtre-2 . progn-2)
     ...
     (filtre-k . progn-k))
```

Chaque clause est constituée d'une expression (le filtre), qui n'est pas évaluée, et d'un `progn` implicite. La valeur de `expr` est appariée successivement avec les `filtre-i` : au premier succès, le `progn-i` correspondant est évalué, dans l'environnement résultant de l'appariement : l'usage des paramètres présents dans `filtre-i` (à l'exception de _) est donc possible dans `progn-i`. Le filtre _ (placé en dernier) récupère tous les cas, comme un `T` dans un `cond`. Si aucun filtre ne réussit, `case-match` retourne `nil`.

Question 5

Le filtrage permet de se passer de la plupart des conditionnelles.

1. Écrire en utilisant `case-match` la fonction `assoc` qui prend 2 arguments, une clé et une liste d'association, et recherche la clé dans la liste, en retournant la paire dont le `car` est la clé, ou `nil`.

   NB. Le code résultant ne doit pas contenir d'autres structures de contrôle que `case-match`.

2. Dans le cadre du méta-évaluateur, implémenter `case-match` comme une forme spéciale, en l'écrivant en LISP de base (avec `if` et `cond`). On définira une fonction `meval-case-match` avec les paramètres adéquats.

   NB. C'est la façon la plus simple d'implémenter le filtrage.

3. Redéfinir `meval-case-match` en l'écrivant avec `case-match`, sans autre structure de contrôle si c'est possible.

   Le code résultant est-il évaluable ?

Question 6

Une alternative consiste en une transformation source à source, donc une macro. Une clause de filtrage doit alors se traduire par des tests, pour vérifier l'appariement de la valeur avec le filtre, et des `let`, pour introduire les nouvelles variables (l'environnement n'est plus créé explicitement, ce sont les `let` qui le feront, à l'évaluation).

1. Montrer sur l'exemple de la fonction `assoc` comment devrait s'expanser une macro `case-match`.
2. Définir la macro `case-match` (on pourra se restreindre à une esquisse et se contenter de spécifier des fonctions auxiliaires).
3. Montrer quelles optimisations seraient nécessaires pour que l'appariement sur un filtre ne refasse pas les tests qui ont déjà été faits pour les filtres précédents (prendre l'exemple de `assoc`).

En savoir plus

Le filtrage est à la base de plusieurs langages de programmation, fonctionnels (HASKELL) ou logiques (PROLOG). Christian Queinnec lui a consacré un joli petit livre, malheureusement épuisé (chez InterEditions), mais disponible sur sa page Web. Le *matching* est à la base de tous les systèmes de règles (logique, réécriture, systèmes experts, etc.) Enfin, la destructuration est utilisée dans de nombreux dialectes LISP : en COMMON LISP, elle n'existe que dans la forme spéciale `destructuring-let`.