Cours 3 - Programmation Lisp (II)

Marie-Hélène Abel

Table des matières

Application de fonction

Remarque

Un atome peut à la fois servir à désigner une fonction et avoir une autre valeur .

Exemple

```
>(setq ff (cons 'lambda (list '(yy zz) '(* 5.0 yy zz))))
(lambda (yy zz) (* 5.0 yy zz))
>(ff 8 9)
Erreur : ff n'est pas le symbole d'une fonction
```

Deux formes spéciales permettent d'appliquer des fonctions à des listes d'arguments :

- **Apply**
- **Funcall**

A. La forme Apply

Syntaxe

Apply doit recevoir en dernier argument une liste contenant le reste des arguments.

Exemple

```
>(apply ff 2 3 ())
30.0
>(apply ff 2 '(3))
30.0
>(apply ff '(2 3))
30.0
>
```

B. La forme Funcall

Syntaxe

Funcall reçoit les arguments sur lesquels doit s'appliquer la fonction

Exemple

```
>(funcall ff 2 3)
```

```
30.0
>
```

Scoping, portée de variable, visibilité des variables

II

Lisp distingue les variables selon qu'elles sont lexicales ou spéciales. Les variables peuvent également être locales ou globales.

A. Variables lexicales

Méthode : Contexte de l'évaluation d'un appel de fonction

- Lors de l'évaluation d'un appel de fonction, le corps de la fonction est évalué avec ses paramètres accessibles en tant que variables locales (variables dont l'accès est limité au code où elles sont définies).
- Au début de l'évaluation du corps de la fonction, chacune de ces variables est associée à une valeur initiale, résultat de l'évaluation du paramètre correspondant lors de l'appel de la fonction.

=> La variable est liée (**bound**) à une valeur. Ce lien (**binding**) n'existe que durant l'évaluation de l'appel. Lorsque celle-ci est terminée le lien disparaît.

Attention

Une variable et son lien ne sont pas visibles dans les corps des fonctions qui seraient appelées comme sous fonctions !!!

Exemple

```
>(defun F-en –euros (x)
(test-acces-x)
(/ x 6.55))
F-en-euros
> (defun test-acces-x ()
(print x))
Test-acces-x
>(F-en-euros 100)
Erreur: ...
```

Définition

Une variable lexicale est une variable dont la portée (**scope**) est limitée au texte (portion de code) où elle est définie.

- Elle n'est accessible que dans le texte où elle est définie.
- Dans l'exemple, elle est accessible dans le corps de la définition de la fonction où elle est définie en tant que paramètre.

B. Variables dynamiques ou spéciales

Définition

Variable dont la portée n'est pas limitée à la portion de code où elle est définie, mais s'étend dans les sous fonctions et les sous fonctions des sous fonctions.

Exemple

```
>(defun F-en-euros-bis (x)
(declare (special x))
(test-acces-x)
(/ x 6.55))
F-en-euros-bis
>(F-en-euros-bis 100)
100
15.25
>
```

C. Variables locales

Préambule

Les variables locales se définissent au moyen des opérateurs spéciaux **let** ou **let***

Attention

En Lisp, par défaut, toute variable locale est une variable lexicale.

1. L'opérateur let

Exemple

```
>(defun carre+1 (x)
(* (+ x 1) (+ x 1)))
Carre+1
>(defun carre+1bis (x)
(let ((x1 (+ x 1)))
(* x1 x1)))
Carre+1bis
>
```

Syntaxe

(let ((var-1 forme-1) (var-2 forme-2) ...)
Expression-1
...
Expression-n)

Méthode

let est un opérateur spécial et a donc sa propre règle d'évaluation :

- Chaque variable var-n est initialisée selon la valeur de la forme correspondante (forme-1, etc)
- Les expressions du corps du **let** sont alors évaluées les unes après les autres, dans l'ordre.
- La valeur de la dernière expression évaluée est retournée comme valeur du **let**. Les variables cessent d'exister.
- La portée des variables du **let** est le corps du **let**.

Exemple

>(let ((x 1) (y 2))
(print x)
(setq x y)
(print x)
"fin")
1
2
"fin"
> (let ((x 'a) (y 2))
(print x)
(set x y)
(print x)
"fin")
? ? ? ? ? ? ? ? ? ? ? ? ?

2. L'opérateur let*

Définition

L'opérateur **let*** est similaire à l'opérateur **let**, à l'exception que les liens des variables sont construits séquentiellement.

Syntaxe

(let* ((var-1 forme-1) (var-2 forme-2) ...)
Expression-1
...
Expression-n)

Remarque

L'expression forme-2 peut utiliser la variable var-1

Exemple

```
>(let* ((x 2) (y (* 4 x)))
(print x)
(setq x y)
(print x)
"fin")
2
8
"fin"
>
```

D. Variables globales

Définition

Une variable globale est une variable qui est visible partout, c'est donc une variable dynamique. Elle peut se définir au moyen des opérateurs spéciaux **defvar** ou **defparameter**.

1. L'opérateur defvar

Syntaxe

```
(defvar var init)
```

Méthode

- Le premier argument est le symbole représentant la variable, il n'est pas évalué
- Le deuxième argument est une expression dont l'évaluation donnera la valeur initiale de la variable

Attention

defvar initialise la variable uniquement si elle n'existe pas déjà.

Exemple

```
> (setq a 3)
3
>
a
3
>(defvar a 4)
A
> a
3
>
```

Remarque

Par convention le symbole représentant la variable commence et finit par une *

2. L'opérateur defparameter

Définition

Idem que **defvar** sauf que **defparameter** réinitialise la variable dans le cas où elle existe déjà

Exemple

```
> (setq a 3)
3
>
a
3
>(defparameter a 4)
A
> a
4
>
```

Blocs

III

Définition

Un bloc est une suite d'expressions jouant le rôle d'une seule expression. Il s'agit d'un regroupement de plusieurs instructions.

Remarque

- Il existe une façon explicite de fabriquer un bloc analogue à une construction "begin-end" des langages procéduraux comme Pascal.
- De nombreuse constructions utilisent un séquencement implicite.

Définition : Contexte

Un contexte est un bloc ayant une ou plusieurs variables locales (**let**, **let***)

A. Blocs explicites

Ils sont définis au moyen des opérateurs spéciaux de type prog : prog1, progn

Définition : progn

Opérateur spécial acceptant un nombre arbitraire d'expressions évaluées séquentiellement. La valeur retournée est celle de la dernière expression évaluée.

Exemple : progn

```
> (progn
(setq w 1)
(print 35)
(+ 3 8))
35
11
> w
1
>
```

Définition : prog1

Idem que **progn** mais retourne la valeur de la première expression évaluée.

Exemple : prog1

```
> (prog1
(setq w 1)
```

```
(print 35)
(+ 3 8))
35
1
> w
1
>
```

B. Blocs implicites

Préambule

Parmi les blocs implicites se trouvent les blocs conditionnels et les blocs itératifs.

1. Blocs conditionnels

Parmi les blocs conditionnels se trouvent les blocs : **cond**, **when**, **unless**

a) Le bloc cond

Définition

Permet de tester plusieurs conditions

Syntaxe

```
(cond
(<test-1> <action11> ...<action1n>)
...
(<test-k> <actionk1> ...<actionkn>))
```

Méthode

L'évaluation du bloc **cond** se fait par itération sur les sous expressions (clauses) :

- Évaluer <test>
- Si résultat <> nil, évaluer les actions associées et sortir du **cond**
- Sinon, évaluer le test suivant
- La valeur retournée par le **cond** est celle de la dernière expression évaluée (cela peut être le dernier test).

b) Le bloc when

Définition

Permet de tester une seule condition.

Syntaxe

```
(when test expr1 ... exprn)
```

Remarque

(when p a b c) <=> (and p (progn a b c)) <=> (cond (p a b c)) <=>(if p (progn a b c)) <=> (unless (not p) a b c)

c) Le bloc unless

Définition

(unless p a b c) <=> (when (not p) a b c)

2. Les blocs itératifs

Parmi les blocs itératifs se trouvent les blocs : **loop**, **dolist**, **dotimes**, **do** ,**do***

a) Le bloc loop

Définition : Boucle indéfinie

Boucle dont les conditions de sortie ne sont pas définies.

Syntaxe

(loop {form}*)

Remarque : Instruction de sortie

On peut en sortir par l'instruction : (return-from nil <valeur>)

b) Le bloc dolist

Définition

Boucle permettant de parcourir tous les éléments d'une liste

Syntaxe

(dolist (var listform [resultform]) corps)

Méthode

- Listform est évalué et doit être une liste
- Exécution du corps pour chaque instanciation de var à chaque élément de listform
- Retourne l'évaluation de resultform, nil si celle-ci est absente.

Exemple

(dolist (x '(a z e r t y) 'toto)
(print x))
? ? ? ? ? ? ? ? ? ? ? ?

c) Le bloc dotimes

Définition

Boucle permettant d'exécuter une expression un certain nombre de fois.

Syntaxe

(dotimes (var countform [resulform] corps)

Méthode

- Countform est évalué et doit être entier
- Exécution du corps de 0 inclus à countform exclus

- Retourne l'évaluation de la forme resultform, nil si elle est absente.

Exemple

```
(dotimes (x 2 'toto)
(print x))
? ? ? ? ? ? ? ? ? ? ?
```

d) Le bloc do

Définition

Boucle qui s'exécute jusqu'à la vérification d'un test.

Syntaxe

```
(do ((var1 init1 step1) (var2 init2 step2) ... (varn initn stepn)) (end-test result)
Corps)
```

Méthode

- initialisation des indexes (vari) en parallèle. Si les init sont omis, par défaut ils sont à nil
- Si les stepi sont omis, les vari correspondant restent inchangés par le do, il faut donc les faire évoluer par un **set**.

Exemple

```
>(do ((x 1 (+ x 1)) (y 2 (+ y 2))) ((= y 6) t)
(print x)
(print y))
1
2
2
4
T
>
```

e) Le bloc do*

Syntaxe

Même syntaxe que le do

Méthode

- Idem que la boucle **do** excepté à la différence que les indexes (vari) ne sont plus initialisés en parallèle mais en séquence.

Opération sur les listes : mapping

IV

Définition

La **mapping** est un type d'itération dans lequel une fonction est successivement appliquée aux éléments d'une ou plusieurs séquences.

Le résultat de l'itération est une séquence contenant les résultats des applications de la fonction.

Différents mapping existent en lisp : **mapcar**, **maplist**, **mapc**, **mapl** , **mapcan**, **mapcon**

Syntaxe

- Mapcar function list &rest more-lists
- Maplist function list &rest more-lists
- Mapc function list &rest more-lists
- Mapl function list &rest more-lists
- Mapcan function list &rest more-lists
- Mapcon function list &rest more-lists

Méthode

Pour chaque **mapping**, le premier argument doit être une fonction et le reste doit être des listes. La fonction doit prendre autant d'arguments qu'il y a de liste.

A. mapcar

Méthode : Mapcar opère sur les éléments successifs de la liste.

Tout d'abord la fonction est appliquée au **car** de chaque liste puis au **cadr** de chaque liste, etc.

Attention

Idéalement, les listes sont de même longueur si ce n'est pas le cas, les itérations se terminent une fois la liste la plus courte traitée, les éléments des listes plus longues sont ignorés.

Exemple

```
> (mapcar #'(lambda (l) (+ (car l) (cadr l))) '((3 7)(2 3)(2 9)(5 3)))
(10 5 11 8)
>
```

B. maplist

Méthode

maplist s'apparente à **mapcar** excepté que la fonction est appliquée aux listes et successivement aux **cdr** de ces listes.

Exemple

```
>(maplist #'(lambda (x) (cons 'foo x)) '(a b c d))
((foo a b c d) (foo b c d) (foo c d) (foo d))
>
```

C. mapl et mapc

Méthode

mapl et **mapc** agissent comme **maplist** et **mapcar** à part qu'elles n'accumulent pas les résultats de la fonction appelante comme valeur de retour mais retourne la première liste.

D. Mapcan et mapcon

Méthode

Mapcan et **mapcon** agissent comme **mapcar** et **maplist** sauf qu'elles combinent les résultats de la fonction en utilisant **nconc** à la place de **list**.