

Programmation applicative – L2

TD 3 : Variables, liaisons, portée

G.Artignan {artignan@lirmm.fr} M.Lafourcade {lafourcade@lirmm.fr}
S.Daudé {sylvain.daude@univ-montp2.fr} A.Chateau {chateau@lirmm.fr}
B.Paiva Lima da Silva {bplsilva@gmail.com} C.Dony {dony@lirmm.fr}

1 La forme spéciale `let`

La forme spéciale `let` est très pratique car elle permet de définir des **variables locales**. En effet, il est souvent utile de définir localement à une fonction (ou à une expression) des variables qui sont utilisées seulement dans le corps de cette fonction (ou expression).

<pre>(let ((⟨var₁⟩⟨exp₁⟩) (⟨var₂⟩⟨exp₂⟩) ... (⟨var_n⟩⟨exp_n⟩)) ⟨corps⟩)</pre>	Par exemple : > (let ((a 1)(b 2)(c (* 2 3))) (+ (* b c) a)) ⇒ 13
---	---

Remarque — Les variables locales définies par un `let` peuvent être des données comme des fonctions. Il est parfois utile de définir des fonctions simplement localement. Par exemple :

```
> (let ((a 1)
        (foo (lambda (x) (+ 2 x))))
      (+ (foo 2) a))
⇒ 5
```

Évaluation de la forme `let` Pour chaque définition (couple *symbole/expression*) le symbole est associé à l'évaluation de son expression. Le corps du `let` est évalué après substitution des variables définies par leur valeur.

Remarque — La substitution variable/valeur est effectuée uniquement dans le corps et pas dans le bloc de définition.

Le corps du `let` est une séquence d'expression. De la même manière qu'avec la forme `begin`, toutes les expressions de la séquence sont évaluées et la valeur de la dernière expression est le résultat renvoyé par le `let`.

Exercice 1 Quelle est la valeur des expressions suivantes :

<pre>a) (let ((a 2) (b (* 2 2)) (c 10)) (* c (- a b)))</pre>	<pre>b) (let ((x 5)) (* x x) (let ((x 10)) (* x x)))</pre>	<pre>c) (let ((a 2) (b (let ((a 3) (b 4)) (- a b)(+ a b)))) (* a b))</pre>
--	--	--

2 let et lambda, même combat !

Lorsque un `let` est évalué il y a substitution des variables locales var_i dans le corps du `let`, par la valeur de l' exp_i qui leur est associé dans la définition du `let`. Ce qui est exactement le comportement de l'évaluation d'une lambda expression. Ainsi :

$$\begin{array}{l} (\text{let } ((\langle var_1 \rangle \langle exp_1 \rangle) \\ \quad (\langle var_2 \rangle \langle exp_2 \rangle) \\ \quad \dots \\ \quad (\langle var_n \rangle \langle exp_n \rangle)) \\ \quad \langle corps \rangle) \end{array} \Leftrightarrow \begin{array}{l} ((\text{lambda } (\langle var_1 \rangle \langle var_2 \rangle \dots \langle var_n \rangle) \\ \quad \langle corps \rangle) \\ \quad \langle exp_1 \rangle \langle exp_2 \rangle \dots \langle exp_n \rangle) \end{array}$$

Exercice 2 Ré-écrire les expressions de l'exercice 1 en utilisant `lambda`.

3 Liaison et portée d'une variable

Les notions de liaison et de portée d'une **variable** sont fondamentales dans tous les langages informatiques. On appelle **liaison** le lien qui est effectué entre un symbole var_i et une valeur correspondant au résultat de l'évaluation de l'expression exp_i . Ce qui signifie que le symbole var_i désigne la valeur de exp_i . On dit que ce lien a un espace de validité que l'on appelle **portée**.

Par exemple, l'expression `(define a (+ 2 3))` lie le symbole `a` à la valeur de `(+ 2 3)` c'est à dire 5. La portée d'un `define` au *top-level* (c'est à dire au niveau du prompteur `">"`) est toute la session Scheme, on parle de **variable globale**. Par contre, dans l'expression `(let ((a (+ 2 2))) a)` le symbole `a` est lié à la valeur 4 seulement dans le corps du `let`, on parle alors de **variable locale**. Ainsi :

```
> (define a (+ 2 3))           ⇒ rien
> a                             ⇒ 5
> (let ((a (+ 1 1)) (b 2)) (+ a b)) ⇒ 4
> a                             ⇒ 5
> b                             ⇒ erreur - b non défini
```

Remarque — On note que dans l'expression `let` le `a` global n'est pas visible car il est caché par le `a` local. De façon générale, les variables définies par les liaisons d'un `let` cachent les autres définitions seulement le temps de l'évaluation du corps du `let`.

Lorsque Scheme évalue une variable, il regarde à quoi celle-ci est liée avant tout de façon locale, si une liaison est trouvée alors il l'utilise, sinon il regarde au niveau local supérieur. Ainsi :

```
> (let ((b 2)) (+ a b)) ⇒ 7
```

Exercice 3 Quelle est la valeur des expressions suivantes tapées au *top-level* :

```
(define a 10)                               (let ((a (* 2 a))) (* 2 a))

(let ((x a)) (* 2 x))                       (let ((a (* 2 a)) (b 5) (c (+ 1 a)))
(let ((a a)) (* 2 a))                       (+ a b c))
```

Exercice 4 Quel est l'affichage engendré par l'expression suivante ?

```
(let ((a 'b) (b 3))
  (display 'a)(display " : ")(display a)(newline)
  (display 'b)(display " : ")(display b)(newline)
  (let ((a 4) (b a))
    (display 'a)(display " : ")(display a)(newline)
    (display 'b)(display " : ")(display b)))
```

3.1 Les autres let : let*, letrec

Pour faire des liaisons en séquence dans un `let` on doit utiliser la forme spéciale `let*`. Elle permet d'utiliser dans le calcul de exp_j la valeur de var_i si $i < j$. Elle permet de voir des variables liées préalablement dans le `let*`. On a donc l'équivalence :

$$\begin{array}{ccc}
 (\text{let}^* ((\langle var_1 \rangle \langle exp_1 \rangle) & & (\text{let} ((\langle var_1 \rangle \langle exp_1 \rangle)) \\
 (\langle var_2 \rangle \langle exp_2 \rangle) & \Leftrightarrow & (\text{let} ((\langle var_2 \rangle \langle exp_2 \rangle)) \\
 \dots & & \dots \\
 (\langle var_n \rangle \langle exp_n \rangle)) & & (\text{let} ((\langle var_2 \rangle \langle exp_2 \rangle)) \\
 \langle corps \rangle) & & \langle corps \rangle) \dots)
 \end{array}$$

Par exemple pour la fonction `numjour` de l'exercice 5 de la feuille TD2, qui renvoyait le nombre de jour pour la date de Pâques, il est pratique d'utiliser un `let*` :

```
(define (numjour-let annee)
  (let* ((a (modulo annee 19))
        (b (modulo annee 4))
        (c (modulo annee 7))
        (d (modulo (+ (* 19 a) 24) 30))
        (e (modulo (+ (* 2 b) (* 4 c) (* 6 d) 5) 7)))
    (+ 22 d e)))
```

Il existe également un `let` qui permet de gérer les appels récursifs. C'est la forme spéciale `letrec` que nous détaillerons plus tard. Par exemple :

```
(letrec ((fact (lambda (x) (if (= 0 x) 1 (* x (fact (- x 1))))))
  (a 5))
  (fact a))
```

Exercice 5 *Quelle est la valeur des expressions suivantes tapées au top-level :*

<pre>(define a 10) (let ((a 2)(b a)) (* a b)) (let* ((a 2)(b a)) (* a b))</pre>	<pre>(let* ((a (* 2 a)) (b 5) (c (+ 1 a))) (+ a b c)) (letrec ((a (* 2 a)) (b 5) (c (+ 1 a))) (+ a b c)) (let ((x 5)) (let* ((y (+ x 10)) (z (* x y))) (+ x y z)))</pre>
---	--

Remarque — Notez que dans les liaisons en séquences, `letrec` se comporte comme `let*`.

Exercice 6 *Quelle est la valeur des expressions suivantes :*

```
(define a 10)                                     (let ((a 0)) (foo 5))

(define (foo x) (* a x))                          ((lambda (a)
  (let ((a 1)
        (foo (lambda (x) (+ a x))))
    (foo a))) 5)

(let ((a 0)) (foo 5))                              ((lambda (a)
  (let* ((a 1)
         (foo (lambda (x) (+ a x))))
    (foo a))) 5)

(define a 100)
```