LI101 : Programmation Récursive

©Equipe enseignante Li101

Université Pierre et Marie Curie

Semestre : Automne 2013

Cours 7 : Fonctionnelles

Plan du cours

1. Les fonctions comme des valeurs
   - Comme paramètre
   - Comme valeur de retour
   - Composition de fonctions
2. Les fonctionnelles :
   - Itération
   - `map`
   - `filter`
   - `reduce`

Valeurs fonctionnelles

Une fonction est aussi une valeur

Par exemple, on peut construire des listes de fonctions :

```
(list + *) ⟶ (#<primitive:+> #<primitive:*>)
```

On peut appliquer une fonction extraite d'une telle liste :

```
((car (list + *)) 5 8) ⟶13
((cadr (list + *)) 5 8) ⟶40
```

Résultat fonctionnel

Une fonction peut être une valeur de retour d'une fonction

```
;;; mul-ou-div: bool -> (Nombre * Nombre -> Nombre)
;;; (mul-ou-div b) rend soit la multiplication, soit
;;; la division selon que b est vrai ou faux.
(define (mul-ou-div b)
   (if b * /))
```

Le résultat de `mul-ou-div` **est une fonction**

$\Rightarrow$ il peut être appliqué :

```
((mul-ou-div #t) 10 2) ⟶20
((mul-ou-div #f) 10 2) ⟶5
```

Application

Rappel de la syntaxe :

$$(\underbrace{e_1}_{\textit{fonction}} \quad \underbrace{e_2 \ldots e_n}_{\textit{arguments}})$$

Rappel de la règle d'évaluation (fonctions définies)

1. Évaluer l'expression en position de fonction,
   $\rightarrow$ on obtient une valeur fonctionnelle :
   paramètres (variables) + *corps* (expression)
2. Évaluer les expressions en positions d'arguments,
   $\rightarrow$ on obtient des valeurs
3. Dans le corps, remplacer les paramètres par les valeurs obtenues,
4. Évaluer l'expression ainsi obtenue.

Application : exemples

Évaluation de `((mul-ou-div #f) (* 10 2) (+ 2 3))`

```
⟶((if #f * /) (* 10 2) (+ 2 3))
⟶(/ (* 10 2) (+ 2 3))
⟶(/ 20 (+ 2 3))
⟶(/ 20 5)
⟶4
```

Évaluation de `(= 12 ((mul-ou-div #t) 4 3))`

```
⟶(= 12 ((if #t * /) 4 3))
⟶(= 12 (* 4 3))
⟶(= 12 12)
⟶#t
```

Fonctions en arguments

Fonctions comme arguments d'une fonction

Exemple : **la composition de fonctions**

```
;;; compose: (alpha -> beta) * (gamma -> alpha) * gamma
;;;          -> beta
;;; (compose f g x) rend le résultat de la fonction
;;; composée (f ∘ g) appliquée à x.
(define (compose f g x)
   (f (g x)))
```

Notez la signature

Composition de fonctions : exemple

```
;;; plus-8: Nombre -> Nombre
;;; (plus-8 n) ajoute 8 à n
(define (plus-8 n)
   (+ n 8))
;;; fois-2: Nombre -> Nombre
;;; (fois-2 n) multiplie son argument par 2
(define (fois-2 n)
   (* n 2))
```

Évaluation de `(compose plus-8 fois-2 5)`

⟶`(plus-8 (fois-2 5))`
⟶`(plus-8 (* 5 2))`
⟶`(plus-8 10)`
⟶`(+ 10 8)`
⟶`18`

Fonctionnelles

Les fonctions qui manipulent des fonctions (comme argument ou comme résultat) sont appelées fonctionnelles

On dit aussi : *fonctions d'ordre supérieur*

- La fonction `mul-ou-div` est une fonctionnelle.
- La fonction `compose` est une fonctionnelle.
- On peut aussi définir des fonctionnelles par **récurrence**
  - sur les entiers ;
  - sur les listes ;
  - etc.

Itération

Répéter l'application d'une fonction : f^n

En mathématique :

$$\begin{cases} f^0(x) & = & x \\ f^{n+1}(x) & = & f(f^n(x)) \end{cases}$$

En Scheme :

```
;;; iter: (alpha -> alpha) * nat * alpha -> alpha
;;; (iter f n x) donne la valeur de f appliquée
;;; n fois à x.
(define (iter f n x)
   (if (= n 0)
       x
       (f (iter f (- n 1) x))))
```

Itération

Exemple

Évaluation de `(iter fois-2 3 1)`

⟶`(fois-2 (iter fois-2 2 1))`
⟶`(fois-2 (fois-2 (iter fois-2 1 1)))`
⟶`(fois-2 (fois-2 (fois-2 (iter fois-2 0 1))))`
⟶`(fois-2 (fois-2 (fois-2 1)))`
⟶`(fois-2 (fois-2 2))`
⟶`(fois-2 4)`
⟶`8`

Itération

Autre définition possible

En mathématique, on a

$$f(f^n(x)) = f^n(f(x))$$

On peut donc aussi définir en Scheme :

```
(define (iter f n x)
   (if (= n 0)
       x
       (iter f (- n 1) (f x))))
```

Itération

On obtient une suite d'évaluations différente mais

avec le même résultat

Exemple : Évaluation de `(iter fois-2 3 1)`

```
⟶(iter fois-2 2 (fois-2 1))
⟶(iter fois-2 2 2)
⟶(iter fois-2 1 (fois-2 2))
⟶(iter fois-2 1 4))))
⟶(iter fois-2 0 (fois-2 4)))
⟶(iter fois-2 0 8)
⟶8
```

Fonctionnelles

Itération avec des listes

Cas particuliers du **schéma général** de récurrence sur les listes

- **Schéma d'application :** itérateur `map`

  transformer une liste en une autre par application d'une fonction à chaque élément.

- **Schéma de filtrage :** itérateur `filter`

  sélectionner des éléments d'une liste à l'aide d'un prédicat pour construire une sous-liste.

- **Schéma de réduction :** itérateur `reduce`

  calculer une valeur par applications composées d'une fonction sur les éléments d'une liste.

Schéma d'application

Appliquer une fonction à chaque élément

```
(define (liste-negation L)
   (if (pair? L)
       (cons (not (car L)) (liste-negation (cdr L)))
       (list)))
```

Évaluation `(liste-negation (list #t #f)` $\longrightarrow$ `(#f #t)`

```
(define (liste-oppose L)
    (if (pair? L)
       (cons (- (car L)) (liste-oppose (cdr L)))
       (list)))
```

Évaluation `(liste-oppose (list -1 0 1))` $\longrightarrow$ `(1 0 -1)`

Schéma d'application

Forme générale

```
(define (liste-F L)
   (if (pair? L)
       (cons (F (car L)) (liste-F (cdr L)))
       (list)))
```

Généraliser = paramètre de fonction

```
(define (liste-applique F L)
   (if (pair? L)
       (cons (F (car L)) (liste-applique F (cdr L)))
       (list)))
```

On a : `(liste-applique not (list #t #f))` $\longrightarrow$ `(#f #t)`

Itérateur map

Fonction *prédéfinie* en Scheme :

```
;;; map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; (map f L) applique la fonction f aux éléments
;;; de la liste L
```

map et définition locale

Ajouter 1 à tous les éléments d'une liste

La fonction "ajouter 1" n'existe pas ? Je la définis !

```
(define (liste-succ L)
   (define (succ x)
      (+ x 1))
   (map succ L))
```

Schéma de filtrage

Ne garder que les nombres positifs d'une liste :

```
(define (filtre-positifs L)
   (if (pair? L)
       (if (positive? (car L))
           (cons (car L) (filtre-positifs (cdr L)))
           (filtre-positifs (cdr L)))
       (list)))
```

Ne garder que les listes non vides d'une liste de listes :

```
(define (filtre-non-vides L)
   (if (pair? L)
       (if (pair? (car L))
           (cons (car L) (filtre-non-vides (cdr L)))
           (filtre-non-vides (cdr L)))
       (list)))
```

Itérateur `filter`

Fonction prédéfinie en Scheme

```
;;; filter: (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
;;; (filter p? L) filtre la liste L en ne conservant que
;;; les éléments pour lesquels p? est vérifié
```

Exemple 1 :

```
(filter pair? '((1 2) () (3 4 5) () (6) ()))
          ⟶((1 2) (3 4 5) (6))
```

Exemple 2 : en utilisant une fonction interne, ne garder que les chaînes de longueur au moins n

```
(define (filtre-longueur n L)
   (define (au-moins-n s)
           (> (string-length s) n))
   (filter au-moins-n L))
```

Schéma de réduction

Somme des éléments d'une liste :

```
(+ n1 (+ n2 ... (+ nk 0)...))
```

Produit des éléments d'une liste :

```
(* n1 (* n2 ... (* nk 1)...))
```

Superposition des images d'une liste :

```
(overlay im1 (overlay im2 ...
                    (overlay imk (empty-image))...))
```

Format général : étant données une fonction binaire `f` et une valeur finale `a` :

```
(f v1 (f v2 ... (f vk a)...))
```

Itérateur reduce

Fonction prédéfinie en Scheme

```
;;; reduce: (alpha * beta -> beta) * beta * LISTE[alpha]
;;;         -> beta
;;; (reduce f a L) réduit la liste L en appliquant
;;; la fonction binaire f aux éléments de la liste
;;; Avec a comme valeur finale.
```

Exemple 1 :

```
(reduce string-append "" (list "Hello" "... " "world"))
```

$\longrightarrow$ `"Hello... world"`

Itérateur reduce

Exemple 2 : Somme des longueurs de chaînes de caractères

```
(define (sommme-longueurs L)
   (define (add-longueur s n)
           (+ (string-length s) n))
   (reduce add-longueur 0 L))
```

Application :

```
(somme-longueurs (list "Hello" "... " "world")) ⟶14
```

Itérateur `reduce`

Attention à l'ordre des calculs !

On a : `(reduce - 0 (list 30 20 10))` $\longrightarrow$ `20`
Pourquoi ?

```
(reduce - 0 (list 30 20 10))
  ⟶ (- 30 (reduce - 0 (20 10)))
  ⟶ (- 30 (- 20 (reduce - 0 (10))))
  ⟶ (- 30 (- 20 (- 10 (reduce - 0 ()))))
  ⟶ (- 30 (- 20 (- 10 0)))
```

C'est-à-dire :

```
30 - (20 - 10) = 30 - 10 = 20
```

Travail avant le prochain TD/TME

- Points traités :
  - Aspects fonctionnels du langage : les fonctions comme valeurs
  - Les fonctionnelles : itérateurs sur des listes
- Être capable d'écrire, sans l'aide des notes et du cours, la spécification et la définition des fonctions présentées