Programmation fonctionnelle - PG104 -

COURS 9

Myriam Desainte-Catherine et David Renault

February 18, 2020

Itérations générales : formes foldl et foldr

Comme la forme **map**, les formes fold appliquent une fonction aux éléments d'une ou plusieurs listes. Alors que **map** combine les résultats obtenus dans une liste, les formes fold les combinent d'une façon déterminée par leur paramètre fonctionnel `f`. Elles appliquent `f` aux éléments des listes de gauche à droite ou bien de droite à gauche. L'argument `init` est utilisé pour terminer la combinaison récursive du résultat.

- **Calcul de gauche à droite** : `(foldl f init list)` = (**f** e_n(**f** e_{n-1}(**f**...(**f** e_1 *init*))))
  avec list = (e_1 e_2...e_n)

```
> (foldl cons '() '(1 2 3))
(cons 3 (cons 2 (cons 1 '())))
'(3 2 1)
> (foldl * 1 '(1 2 3))
(* 3 (* 2 (* 1 1)))
6
> (foldl (lambda (x y) (+ (sqr x) y)) 0 '(1 2 3))
14
```

- **Calcul de droite à gauche** : `(foldr f init list)` = (**f** e_1(**f** e_2(**f**...(**f** e_n *init*))))
  avec list = (e_1 e_2...e_n)

```
>  (foldr cons '() '(1 2 3))
(cons 1 (cons 2 (cons 3 '())))
'(1 2 3)
>  (foldr cons '(1 2 3) '(3 4 5))
(cons 3 (cons 4 (cons 5 '(1 2 3))))
'(3 4 5 1 2 3)
```

Fonctionnement des formes fold

Quels sont les résultats des expressions suivantes?

```
> (foldl * 0 '(1 2 3))
```

6 ou 0 ou erreur 0

```
> (foldl cons '() (list 1 2 3))
```

'(3 2 1)

'(1 2 3)

'() '(3 2 1)

```
> (foldl cons '(1 2) (list 1 2 3))
```

'(3 2 1 1 2)

'(1 2)

'(3 2 1 2 1)

'(1 2 3 1 2) '(3 2 1 1 2)

```
> (foldr cons '() (list 1 2 3))
```

'(3 2 1)

'(1 2 3)

'() '(1 2 3)

```
> (foldr cons '(1 2) (list 1 2 3))
```

'(3 2 1 1 2)

'(1 2)

'(3 2 1 2 1)

'(1 2 3 1 2) '(1 2 3 1 2)

Formes map, fold, andmap et formes spéciales

- Les formes **map**, fold et andmap sont-elles des formes spéciales? Oui ou Non Non
- Les formes **and** et **or** sont-elles des formes spéciales? Oui ou Non Oui
- Soit l'expression suivante :

```
> (andmap + '(1 2 3) '(4 5 6))
9
```

- Obtient-on le même résultat avec l'expression suivante? Oui ou Non Non

```
> (foldl and #t (map + '(1 2 3) '(4 5 6)))
```

- Soit les fonctions

```
(define (et1 a b) (if a b #f))
(define (et2 a b) (if b a #f))
```

Pour avoir le même résultat que l'expression `(andmap + '(1 2 3) '(4 5 6))`, faut-il utiliser et1 ou et2 à la place de et dans l'expression suivante : et2

```
(foldl et #t (map + '(1 2 3) '(4 5 6)))
```

Évaluation applicative avec **eval** et **apply**

Évaluation : (eval sexpr)

- **sexpr** : expression symbolique
- Si **sexpr** est autoévaluante, renvoyer **sexpr**
- Si **sexpr** est un symbole, alors
  - Rechercher une liaison définissant **sexpr** dans l'environnement courant et renvoyer la valeur associée.
- Si **sexpr** est une liste
  - Calculer (eval (car **sexpr**)). Soit **f** la fonction résultat.
  - Calculer (eval **ei**), pour tout élément **ei** de (cdr **sexpr**). Soit **v** la liste des résultats.
- Calculer (apply **f** **v**)

Application : (apply f v)

- **f** : fonction à appliquer
- **v** : liste des valeurs des arguments
- Soient **e** l'environnement lexical de **f**, **lf** la liste des paramètres formels, et **expr** le corps de la fonction.
- Construire l'environnement local **e-local** constitué des liaisons entre les paramètres formels de **lf** et les valeurs correspondantes dans **v**.
- Calculer : (eval **expr** (cons **e-local** **e**)).

Application : la forme **apply**

Cette fonction réalise l'application d'une fonction à une liste d'arguments. Ce mécanisme est utile pour l'écriture de fonctions à nombre d'arguments variable.

$$(\texttt{apply}\ \langle f \rangle \langle l \rangle) = (\langle f \rangle\ \langle e_1 \rangle\ \langle e_2 \rangle \ldots \langle e_n \rangle)$$

avec $\langle l \rangle = (\langle e_1 \rangle\ \langle e_2 \rangle \ldots \langle e_n \rangle)$

Exemple

```
> (apply + '(1 2 3)) ; -> (+ 1 2 3)
6
> (apply + '()) ; -> (+)
0
> (apply + 1 2 '(3 4)) ; -> (+ 1 2 3 4)
10
```

Cas d'utilisation : fonctions à nb d'arguments variable

```
(define (moyenne-carre . l)
  (if (null? l)
      0
      (/ (apply + (map sqr l)) (length l))))
> (moyenne-carre)
0
```

```
(define (iota n . l)
  (if (zero? n)
      (cons n l)
      (apply iota (sub1 n) (cons n l))))
> (iota 4)
'(0 1 2 3 4)
```

Fonctionnement de la forme apply

Quels sont les résultats des expressions suivantes?

```
> (apply * '(1 2 3) '(1 2))
```
12 ou erreur erreur

```
> (apply * 1 2 3 '(1 2))
```
12 ou erreur 12

```
> (apply * 1 2 3 1 2)
```
12 ou erreur erreur

```
> (apply * 1 2 3 1 2 '())
```
12 ou erreur 12

```
> (apply and '(#t #f #t))
```
#f ou erreur erreur

Réponses

Quels sont les résultats des expressions suivantes?

```
> (apply * '(1 2 3) '(1 2))
```
erreur

```
> (apply * 1 2 3 '(1 2))
```
12

```
> (apply * 1 2 3 1 2)
```
erreur

```
> (apply * 1 2 3 1 2 '())
```
12

```
> (apply and '(\#t \#f \#t))
```
erreur

```
> (apply map list ''(1 2 3) (2 3 4))
```
'((1 2) (2 3) (3

Exemple de fonction n-aire avec apply

La fonction ou n-aire

```
(define (ou . l)
   (cond ((null? l) #f)
         ((car l))
         (else (apply ou (cdr l)))))
```

Questions

- (apply ou '(#f #t)) : #t ou erreur #t
- (ou #f #t #f (print #f)) : #t ou #f#t ou #t#f #f#t

Remarque : Même si la fonction a un nombre d'arguments variable, et même si elle ne parcourt pas toute la liste (elle s'arrête au premier argument valant vrai), les arguments sont tout de même TOUS évalués au moment de l'application.

Identifier et différencier les formes map, fold et apply

```
(define (moyenne . l)
  (if (null? l)
      0
      (/ (apply + l) (length l))))
```

```
> (moyenne 1 2 3 4 5)
3
> (define l '(1 2 3 4 5))
```

Quels sont les résultats des expressions suivantes?

`> (foldl * 1 l)` 1 ou 120 ou erreur 120

`> (apply * l)` 120 ou erreur 120

`> (apply moyenne l)` 3 ou $4\frac{1}{32}$ ou erreur 3

`> (foldl moyenne 0 l)` 3 ou $4\frac{1}{32}$ ou erreur $4\frac{1}{32}$

`> (apply map list '((1 2 3) (2 3 4)))` '((1 2) (2 3) (3 4)) ou '((1 2 3) (2 3 4)) ou erreur '((1 2) (2 3) (3 4))

Utiliser map, fold et apply

- On veut construire la liste des carrés des éléments de la liste '(1 2 3) et obtenir pour résultat '(1 4 9). Quelle est l'expression qui convient?
  (map sqr '(1 2 3)) ou (foldr sqr '(1 2 3)) ou (apply sqr '(1 2 3))
  (map sqr '(1 2 3))
- On veut faire la somme des éléments d'une liste '(1 2 3) et obtenir pour résultat 6. Quelle est l'expression qui convient?
  (map + '(1 2 3)) ou (foldl + 1 '(1 2 3)) ou (apply + '(1 2 3))
  (foldl + 0 '(1 2 3)) et (apply + '(1 2 3))
- On veut transformer une liste à 2 éléments '(1 2) en une paire pointée et obtenir pour résultat '(1 . 2). Quelle est l'expression qui convient?
  (map cons '(1 2)) ou (foldl cons '(1 2)) ou (apply cons '(1 2))
  (apply cons '(1 2))

Fonctions en retour de fonctions : composition

En scheme, il est possible de manipuler et de créer des fonctions dynamiquement au moyen d'expressions.

Composition de fonctions

$$f : A \longrightarrow B$$

$$g : B \longrightarrow C$$

La composée de f par g est la fonction $h : A \longrightarrow C$ telle que $h(x) = g(f(x))$.
Elle est notée $h = gof$.

Opérations de composition

- **Fonctions unaires** : (compose1 $\langle proc_1 \rangle$ $\langle proc_2 \rangle$ ... $\langle proc_n \rangle$)
- **Arité quelconque** : (compose $\langle proc_1 \rangle$ $\langle proc_2 \rangle$ ...$\langle proc_n \rangle$)
  Le nombre de résultats de $\langle proc_i \rangle$ doit correspondre à l'arité de $\langle proc_{i-1} \rangle$

Exemple

```
> ((compose1 sqrt add1) 1) ; (sqrt (add1 1))
1.414213562
> (define (2r x y) (values (+ x y) (- x y)))
> ((compose list 2r) 1 2) ; (list (2r 1 2))
'(3 -1)
```

Fonctions en retour de fonctions : composition

```
(define (2r x y)
   (values (+ x y) (- x y)))
```

- ((compose - sqr sub1) 3) : 8 ou -4 -4
- ((compose + 2r) 1 2) : 2 ou 3 2
- ```
  ((compose (lambda(x) (apply * x))
            (lambda(x) (map sub1 x)))
   '(1 2 3))
  ```

  0 ou erreur 0
  ```

Fonctions en retour de fonctions : curryfication

La fonction curry curryfie son argument. Soit une fonction

$$f : A \times B \longrightarrow C$$

la curryfication lui associe la fonction suivante :

$$f^c : A \longrightarrow (B \longrightarrow C)$$

qui a pour résultat une fonction allant de B dans C et telle que $\forall x \in A, f(x, y) = (f^c(x))(y)$

Haskell B. Cu

Définition de fonctions currifiées

```
> (define *2 ((curry *) 2))
> (*2 3)
6
> (define *curried (curry *))
> (*curried 2)
#<procedure:curried>
> ((*curried 2) 3)
6
```

Construction de fonctions curryfiées à la volée

```
> (((curry list) 1) 2)
'(1 2)
> (((curry list) 1 2) 2)
'(1 2 2)
```

Curry, map et compose

- (map ((curry cons) 1) '(a b c d)) :
  '(1 a b c d) ou '((1 . a) (1 . b) (1 . c) (1 . d)) ou erreur
  '((1 . a) (1 . b) (1 . c) (1 . d))
- (((curry apply) *) '(1 2 3)) : 6 ou '(1 2 3) ou erreur 6
- (((curry map) *) '(1 2 3)) : 6 ou '(1 2 3) ou erreur
  '(1 2 3)
- ((compose ((curry apply) *) ((curry map) sub1)) '(1 2 3)) :
  0 ou erreur 0

Cas d'utilisation de la curryfication

On va curryfier la fonction filter de la bibliothèque pour la spécialiser sur le prédicat even?

```
> (filter even? '(1 2 3 4))
'(2 4)
> (define filter-even ((curry filter) even?))
> (filter-even '(1 2 3 4))
'(2 4)
```

Cela peut permettre de l'utiliser dans une fonctionnelle de liste comme map par exemple.

Exemple

```
> (map filter-even '((1 2 3 4) (3 6 2 4 23 1)))
'((2 4) (6 2 4))
```

Fonctionnement de compose, map, curry et apply

Quels sont les résultats des expressions suivantes?

```
> (map ((curry cons) 1) '(a b c))
```

- '((1 . a) (1 . b) (1 .
- '(1 a b c)
- erreur

```
> (((curry apply) *) '(1 2 3))
```

- 6
- '(1 2 3)
- erreur

```
> (((curry map) *) '(1 2 3))
```

- 6
- '(1 2 3)
- erreur

```
> ((compose ((curry apply) *)
            ((curry map) sub1))
   '(1 2 3))
```

- 0
- erreur

Programmation de fonctionnelles : composition

Composition de deux fonctions

```
(define (o g f)
  (lambda(x)
    (g (f x))))
```

Exemple

```
> (o sqr sub1)
#<procedure>
> ((o sqr sub1) 2)
1
> (define sqr1- (o sqr sub1))
> (sqr1- 2)
1
```

Remarque : écriture équivalente

```
(define o (lambda (g f) (lambda(x) (g (f x)))))
```

Programmation de fonctionnelles : curryfication

Curryfication d'une fonction binaire

$$curry1 \ : (A \times B \longrightarrow C) \longrightarrow (A \longrightarrow (B \longrightarrow C))$$

telle que pour $f \in (A \times B \longrightarrow C)$ soit $f^c = curry1(f) \in (A \longrightarrow (B \longrightarrow C))$, on a $f(x, y) = (f^c(x))(y)$

```
(define (curry1 f) (lambda (x) (lambda (y) (f x y))))
```

On utilise la fonction map avec deux arguments, c'est-à-dire avec un argument fonctionnel unaire pour premier argument et une liste pour deuxième argument.

```
> (curry1 map)
#<procedure>
> ((curry1 map) list)
#<procedure>
> (define map-list ((curry1 map) list))
> (map-list '(1 2 3 4))
'((1) (2) (3) (4))
```

Remarque : écriture équivalente

```
(define curry1 (lambda (f) (lambda (x) (lambda (y) (f x y)))))
```

Programmation de fonctionnelles : curryfication

Curryfication d'une fonction n-aire

$$curry1n\ : (A_1 \times A_2 \times \ldots \times A_n \longrightarrow C) \longrightarrow (A_1 \longrightarrow (A_2 \times \ldots \times A_n \longrightarrow C))$$

telle que pour $f \in (A_1 \times A_2 \times \ldots A_n \longrightarrow C)$ soit $f^c = curry1n(f) \in (A_1 \longrightarrow (A_2 \times \ldots \times A_n \longrightarrow C))$, on a $f(a_1, a_2, \ldots, a_n) = (f^c(a_1))(a_2, a_3, \ldots, a_n)$

Comment généraliser curry1?

Nombre d'arguments variables? 1ère lambda 2ème lambda

```
(define (curry1 f)
  (lambda (x)
    (lambda (y)
      (f x y))))
```

```
(define (curry1n f)
  (lambda (x)
    (lambda y
      (apply f x y))))
```

On utilise la fonction map avec plusieurs arguments.

```
> (define map-list ((curry1n map) list))
> (map-list '(1 2 3 4))
'((1) (2) (3) (4))
> (map-list '(1 2 3 4) '(5 6 7 8))
'((1 5) (2 6) (3 7) (4 8))
> (map-list '(1 2 3 4) '(5 6 7 8) '(9 10 11 12))
'((1 5 9) (2 6 10) (3 7 11) (4 8 12))
```

Programmation de fonctionnelles : curryfication

Curryfication généralisée d'une fonction n-aire

```
> (((curry list) 1) 2)
'(1 2)
> (((curry list) 1 2) 3 4 5)
'(1 2 3 4 5)
```

Curryfication généralisée d'une fonction n-aire

$$curryin : (A_1 \times A_2 \times \ldots \times A_n \longrightarrow C) \longrightarrow (A_1 \times A_2 \ldots \times A_i \longrightarrow (A_{i+1} \times \ldots A_n \longrightarrow C))$$

telle que pour $f \in (A_1 \times A_2 \times \ldots A_n \longrightarrow C)$
Soit $f^{ci} = curryin(f) \in (A_1 \times A_2 \ldots \times A_i \longrightarrow (A_{i+1} \times \ldots A_n \longrightarrow C))$,
on a $f(a_1, a_2, \ldots, a_n) = (f^{ci}(a_1, a_2, \ldots, a_i))(a_{i+1}, \ldots, a_n)$

Comment généraliser curry1n?

1. Doit-elle devenir récursive?
2. Rendre le nb d'arguments de la première lambda variable?

```
(define (curry1n f)
 (lambda (x)
   (lambda y
     (apply f (cons x y)))))
```

```
(define (curryin f)
 (lambda x
   (lambda y
     (apply f
            (append x y)))))
```

Ecriture de fonctionnelles

Quelle version vous paraît plus efficace?

- (define (carre-1a x) ((compose sqr sub1) x))
- (define carre-1b (compose sqr sub1))

Valeurs de carre-1a et carre1b

- La valeur de carre-1a est : (lambda (x) ((compose sqr sub1) x)) – Forme normale
- La valeur de carre-1b est la réduction de (compose sqr sub1)
  - ((lambda (g f) (lambda (x) (g (f x)))) sqr sub1)

  $\longrightarrow$ (lambda (x) (sqr (sub1 x))) – Forme normale

Réductions des applications

- (carre-1a 1)

$\longrightarrow$ ((lambda (x) ((compose sqr sub1) x)) 1)

$\longrightarrow$ (((lambda (g f) (lambda (x) (g (f x)))) sqr sub1) 1)

$\longrightarrow$ ((lambda (x) (sqr (sub1 x))) 1)

$\longrightarrow$ (sqr (sub1 1))

$\longrightarrow$ (sqr 0)

- (carre-1b 1)

$\longrightarrow$ ((lambda (x) (sqr (sub1 x))) 1)

$\longrightarrow$ (sqr (sub1 1))

$\longrightarrow$ (sqr 0)

$\longrightarrow$ 0

Fonctionnelles et C : rappel

```
void* ajouteur (int nombre)
{
  int ajoute (int valeur) { return valeur + nombre; }
  return ajoute;
}

int main(void)
{
    int (*ajoute10)(int) = ajouteur(10);
    ajouteur(20);
    printf("%d\n", ajoute10(1));
    return 0;
}
```

Qu'affiche cette fonction : 11 ou 21 ? 21

Fonctionnelles et C : deuxième version

```
void* ajouteur (int nombre)
{
  int* n = malloc(sizeof(int));
  *n = nombre;
  int ajoute (int valeur) { return valeur + *n; }
  return ajoute;
}

int main(void)
{
    int (*ajoute10)(int) = ajouteur(10);
    ajouteur(20);
    printf("%d\n", ajoute10(1));

    return 0;
}
```

Qu'affiche cette fonction : 11 ou 21 ? 21

Notion de fermeture (closure)

Définitions

- L'environnement lexical d'une fonction est l'environnement dans lequel elle est définie.
- Une fermeture (closure en anglais) est la représentation d'une fonction sous forme d'un couple associant l'environnement lexical et le code de la fonction.
- En Scheme les fonctions sont représentées par des fermetures pour conserver leur environnement de définition contenant des références éventuelles vers des variables libres (ce n'est pas le cas par exemple du langage emacs-lisp).
- Les fonctions créées dynamiquement sont alors utilisables à l'extérieur de la fonction qui les a créées grâce au mécanisme de gestion automatique de la mémoire (ramasse-miettes, garbage collector).
- Les fermetures peuvent être utilisées pour représenter des états, par modification de l'environnement (voir chapitre suivant).

Par exemple pour l'application suivante de la fonction de composition :

```
> (define carre-1 (o sqr sub1))
```

La fermeture représentant carre-1 est la suivante :

[((f . sub1) (g . sqr)) (lambda(x) (g (f x)))]

Les liaisons définissant les variables libres f et g de carre-1 dans l'environnement lexical de la fermeture permettent de conserver les valeurs qui ont été données lors de l'application de la fonctionnelle.