Programmation fonctionnelle - PG104 -

COURS 11

Myriam Desainte-Catherine et David Renault

February 24, 2020

Listes mutables circulaires

Rappel sur la concaténation

```
> (define l '(1 2 3))
> (append l l)
'(1 2 3 1 2 3)
> l
'(1 2 3)
```

Fonction rendant une liste mutable circulaire

```
(define (cirlist ml)
   (letrec ((aux (lambda(l)
                   (if (null? (mcdr l))
                       (set-mcdr! l ml)
                       (aux (mcdr l))))))
     (if (null? ml)
         ml
         (aux ml))))
> (define ml (mcons 1 (mcons 2 '())))
> (cirlist ml)
> ml
#0=(mcons 1 (mcons 2 #0#))
```

Blocs d'expressions

Si x est plus petit que y, on veut échanger x et y et renvoyer x, sinon, on veut renvoyer y. Quelles expressions sont correctes?

```
(let ((tmp 0))
  (if (< x y)
      (set! tmp x)
      (set! x y)
      (set! y tmp)
      x
      y))
```

```
(let ((tmp 0))
  (if (< x y)
      ((set! tmp x)
       (set! x y)
       (set! y tmp)
       x)
      y))
```

Error : if: bad syntax; has 6 parts after keyword in:
(if (< x y) (set! tmp x) (set! x y) (set! y tmp) x y)

Error : application: not a procedure;
expected a procedure that can be applied to arguments
given: #<void>
arguments...:
#<void>
#<void>
1

Blocs d'expressions

Certaines expressions pouvant effectuer des effets de bord, il devient possible de les mettre en séquence. Contrairement aux formes let et lambda, certaines formes, telles le if nécessitent d'utiliser une forme spéciale de mise en séquence.

La forme begin

(begin $\langle e_1 \rangle \langle e_2 \rangle ... \langle e_n \rangle$)

- Chaque expression ei est évaluée selon son ordre d'apparition.
- Le résultat de l'évaluation de la séquence est celui de la dernière.
- Les valeurs des évaluations des expressions précédentes sont perdues.
- Il existe une forme **begin0** qui renvoie le résultat de la première expression de la séquence.

```
(let ((tmp 0))
  (if (< x y)
      (begin (set! tmp x)
             (set! x y)
             (set! y tmp)
             x)
      y))
```

Fermetures et affectations : en Common Lisp

On peut utiliser les fermetures pour modéliser des états.

Générateurs en Common Lisp

```
(let ((i 0))
  (defun gen-entier ()
    (setf i (1+ i))))
```

Exemple

```
* (gen-entier)
1
* (gen-entier)
2
* (gen-entier)
3
```

Fermetures et affectations : et en Scheme?

- Quel est le résultat de l'expression `(f)`? 1 ou Erreur

  Erreur Error : f: undefined; cannot reference undefined identifier

```
(let ((x 1))
  (define (f)
    x)
  (f))
1
```

- Quel est le résultat de l'expression `(e)`? 1 ou 0 ou erreur

  1

```
(define (make-f)
  (let ((x 1))
    (lambda () x)))
> (define e (make-f))
> (define x 0)
```

Fermetures et affectations : générateurs en Scheme

Exemple

```
(define (make-int-gen n)
  (let ((i n))
    (lambda ()
      (set! i (add1 i))
      i)))
```

```
> (define gen0 (make-int-gen -1))
> (define gen10 (make-int-gen 9))
```

```
> (gen0)
0
> (gen0)
1
> (gen0)
2
```

```
> (gen10)
10
> (gen10)
11
> (gen0)
3
```

Les mémo-fonctions (memo functions, memoization)

La technique des mémo-fonctions est utilisée pour optimiser le calcul des fonctions, en mémorisant des résultats d'appels coûteux.

Suite de Fibonacci

```
(define (make-memo-fib); Creation de la fermeture
  ; Initialisation de la table dans l'environnement lexical
  (let ((memo-table '((1 . 1) (2 . 1))))
    (define (memo-fib n); definition de la fonction
      ; Recherche dans la table
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value); la valeur est trouvee
            ;; La valeur est calculee et stockee
            (let ((new-value (+ (memo-fib (sub1 n)); calcul
                                (memo-fib (- n 2)))))
              (set! memo-table; stockage
                    (cons (cons n new-value)
                          memo-table))
              new-value))); retour de la valeur
    memo-fib)); retour de la fonction
```

Les mémo-fonctions : utilisation

Comme pour les générateurs, il faut créer la fermeture par une première application de la fonctionnelle.

Exemple

```
> (define memo-fib (make-memo-fib))
> (memo-fib 5)
5
> (memo-fib 8)
21
>
> (time (memo-fib 100))
cpu time: 0 real time: 0 gc time: 0
354224848179261915075
```

Les mémo-fonctions : généralisation de fib (1)

; Combien de paramètres 1 ou 2 ?

```
(define (make-memo ? ?)

; Quelle valeur initiale? Un des paramètres? oui ou non

  (let ((memo-table ?)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)

; Quel calcul? Une fonction donnée en paramètre? oui ou non

            (let ((new-value ?)) ; (+ (memo (sub1 n))
                                 ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

Les mémo-fonctions : généralisation de fib (1)

; Combien de paramètres 1 ou 2 ?

```
(define (make-memo ? ?)

; Quelle valeur initiale? Un des paramètres? oui ou non

  (let ((memo-table ?)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)

; Quel calcul? Une fonction donnée en paramètre? oui ou non

            (let ((new-value ?)) ; (+ (memo (sub1 n))
                                 ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

Les mémo-fonctions : généralisation de fib (2)

; Combien de paramètres 2 ?

```
(define (make-memo f init)
```

; Quelle valeur initiale? Un des paramètres? oui ou non

```
  (let ((memo-table ?)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)
```

; Quel calcul? Une fonction donnée en paramètre? oui ou non

```
            (let ((new-value ?)) ; (+ (memo (sub1 n))
                                 ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

Les mémo-fonctions : généralisation de fib (3)

; Combien de paramètres 2 ?

```
(define (make-memo f init)
```

; Quelle valeur initiale? Un des paramètres? oui

```
  (let ((memo-table init)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)
```

; Quel calcul? Une fonction donnée en paramètre? oui ou non

```
            (let ((new-value ?)) ; (+ (memo (sub1 n))
                                 ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

Les mémo-fonctions : généralisation de fib (4)

; Combien de paramètres 2 ?

```
(define (make-memo f init)
```

; Quelle valeur initiale? Un des paramètres? oui

```
  (let ((memo-table init)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)
```

; Quel calcul? Donné en paramètre? oui

```
            (let ((new-value (f ? ?))) ; (+ (memo (sub1 n))
                                       ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

Les mémo-fonctions : généralisation de fib (5)

```
; Combien de paramètres 2 ?

(define (make-memo f init)

; Quelle valeur initiale? Un des paramètres? oui

  (let ((memo-table init)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)

; Combien de paramètres à f? 1 ou 2 ?

            (let ((new-value (f ? ?))) ; (+ (memo (sub1 n))
                                       ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

Les mémo-fonctions : généralisation de fib (6)

```
(define (make-memo f init)
  (let ((memo-table init)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)

; Combien de paramètres à f? 2

            (let ((new-value (f n memo))) ; (+ (memo (sub1 n))
                                          ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

```
(define fib (make-memo (lambda (n memo)
                         (+ (memo (sub1 n))
                            (memo (- n 2))))
                       '((1 . 1) (2 . 1))))
```

Les mémo-fonctions : généralisation de fib (7)

```
(define (make-memo f init)
  (let ((memo-table init)) ; '((1 . 1) (2 . 1))
    (define (memo n)
      (let ((computed-value (assoc n memo-table)))
        (if computed-value
            (cdr computed-value)
            (let ((new-value (f n))) ; (+ (memo (sub1 n))
                                     ;    (memo (- n 2)))
              (set! memo-table
                    (cons (cons n new-value)
                          memo-table))
              new-value))))
    memo))
```

Quelle construction avec un paramètre?

```
(define fib
   (make-memo
     (lambda(n)
        (+ ( fib (sub1 n))
           ( fib (- n 2))))
        '((1 . 1) (2 . 1)))))
```

```
(define fib
   (make-memo
      (lambda(n)
         (+ ( memo (sub1 n))
            ( memo (- n 2))))
      '((1 . 1) (2 . 1)))))
```

Les mémo-fonctions : généralisation de fib (8)

Construction de la fonction fact

Quelle expression faut-il utiliser pour construire la fonction fact?

```
(define fact (make-memo (lambda(n)
                          (* n (fact (sub1 n))))
                        1 ))
```

```
(define fact (make-memo (lambda(n)
                          (* n (fact (sub1 n))))
                        '(1 . 1) ))
```

```
(define fact (make-memo (lambda(n)
                          (* n (fact (sub1 n))))
                        '((1 . 1)) ))
```

Rappels sur l'évaluation et l'application – 1

Évaluation applicative : (eval o env)

Cette forme d'évaluation est utilisée pour toutes les fonctions construites avec des **lambda**, **define**, **let** et **letrec**. C'est celle qui est mise en oeuvre dans la plupart des langages de programmation, en particulier impératifs (C, Java). Soit **env** l'environnement courant.

- Si l'objet **o** est autoévaluant, renvoyer **o**
- Si **o** est un symbole, alors
  - Rechercher une liaison définissant **o** dans **env**, renvoyer la référence associée.
- Si **o** est une liste
  - Calculer (eval (car **o**) **env**). Soit **f** la fermeture résultat.
  - Calculer (eval **a** **env**), pour tout élément **a** de (cdr **o**). Soit **v** la liste des résultats.
- Calculer (**apply f v**)

Rappels sur l'évaluation et l'application – 2

Application : (apply **f** **v**)

Avec :

- **f** : fermeture de la fonction à appliquer
- **v** : liste des valeurs des arguments
- Soient **e** l'environnement lexical de **f**, **lf** la liste des paramètres formels, et **c** le corps de la fermeture.
- Construire l'environnement local **e-local** constitué des liaisons entre les paramètres formels de **lf** et les références des valeurs correspondantes dans **v**.
- Pour la suite d'expressions **expr** du corps **c** de **f**, faire :
  (eval **expr** (cons **e-local** **e**)).
- Renvoyer le résultat de l'évaluation de la dernière expression de **c**.

Rappels sur l'évaluation et l'application – 3

Évaluation paresseuse

L'évaluation paresseuse ou par nécessité consiste à retarder l'évaluation des paramètres jusqu'au moment de leur utilisation. Éventuellement, certains paramètres ne sont pas évalués dans certains cas. Ce mécanisme est nécessaire pour implémenter les conditionnelles et donc les boucles.

Remplacement textuel

Il y a deux niveaux de substitutions : l'appel d'une macro est substitué par la définition de la macro dans laquelle les paramètres formels ont été substitués par les arguments donnés lors de l'appel. Toutes ces substitutions sont textuelles. Ainsi, la structure syntaxique n'est pas prise en compte. En C, les macro-fonctions fonctionnent de cette façon. Pour éviter certains pièges syntaxiques, il faut respecter des règles d'écriture des macros (paramètres entre parenthèses, corps entre parenthèses).

Remarque : Quelques problèmes liés à la non prise en compte de la syntaxe

```
#define CARRE(x) x*x
```

3∗CARRE(x+1) ⟶ `3*x+1*x+1` ou `3*(x+1)*(x+1)`

3∗CARRE(x+1) ⟶ `3*x+1*x+1`

```
#define CARRE(x) ((x)*(x))
```

CARRE(x++) ⟶ `((x++) * (x))` ou `((x++) * (x++))`

Macroexpansions par transformation de source

En lisp et en scheme, les macroexpansions fonctionnent par transformation de source, en tenant compte de la syntaxe. Elles permettent d'écrire ces formes dites spéciales, dont l'évaluation n'est pas applicative. Les arguments sont évalués sur demande (en lisp) ou par nécessité (en scheme).

Définition en scheme

(**define-syntax-rule** ⟨*pattern*⟩ ⟨*template*⟩)

- ⟨*pattern*⟩ : (⟨*nom*⟩-⟨*macro*⟩ ⟨p_1⟩ ...)
- ⟨p_i⟩ : variables de la macro
- ⟨*template*⟩ : expressions
- Remplacement des variables dans le template
- Le résultat est une forme
- Évaluation de la forme dans l'environnement d'appel

Macroexpansions en scheme

Exemple

```
(define-syntax-rule (ifnot test then else)
  (if (not test)
      then
      else))
```

> (define x 1)

> (expand-once #'(ifnot (= 1 2) 0 x)) ⟶ #<syntax:8:17 (if (not (= 1 2)) 0 x)>

> (syntax->datum (expand-once #'(ifnot (= 1 2) 0 x))) ⟶ '(if (not (= 1 2)) 0 x)

> (ifnot (= 1 2) 0 x) ⟶ 0

- On constate que les arguments ne sont pas évalués lors du remplacement. Une modification de paramètres est alors possible dans la macro.
- Il faut borner son utilisation des macros aux calculs qui ne peuvent pas être faits par des fonctions : cas où les paramètres ne doivent pas tous être évalués, ou cas où l'on veut modifier certains paramètres)

Question : modification de paramètres

- Soient les définitions suivantes :

```
(define (incrementer x)
  (set! x (add1 x)))

> (define a 0)
> (incrementer a)
> a
```

Quel est le résultat de la dernière expression? 0 ou 1

- Soient les définitions suivantes :

```
(define-syntax-rule (incrementer! x)
  (set! x (add1 x)))

> (define a 1)
> (incrementer! a)
> a
```

Quel est le résultat de la dernière expression? 2 ou 1

Question : évaluation de paramètres

- Soient les définitions suivantes :

```
(define (ifn-f test alors sinon)
  (if (not test) alors sinon))

> (ifn-f (= 1 2) 1 2)
1
> (ifn-f (= 1 2) (print 1) (print 2))
```

Quel est l'affichage produit par les `print` 1 ou 2 ou 12 ?

- Soient les définitions suivantes :

```
(define (fact-f n)
   (ifn-f (zero? n)
          (* n (fact-f (sub1 n)))
          1))
> (fact-f 3)
```

Quel est le résultat de cette dernière expression? 6 ou erreur

Question : évaluation de paramètres

- Soient les définitions suivantes :

```
(define (fact-f n)
   (ifn-f (zero? n)
          (* n (fact-f (sub1 n)))
          1))
> (fact-f 3)
```

Quel est le résultat de cette dernière expression? 6 ou erreur

- Soient les définitions suivantes :

```
(define (fact-m n)
    (ifnot (zero? n)
           (* n (fact-m (sub1 n)))
           1))
> (fact-m 3)
```

Quel est le résultat de cette dernière expression? 6 ou erreur

Macroexpansions

Mécanisme de citation

- **Quote** : ’
- **Backquote** : ‘
- **Virgule** : ,
- **Arobase** : @

Exemple

> (define l ’(1 2 3))

> ‘(1 ,l) $\longrightarrow$ (1 (1 2 3))

> ‘(+ ,@l) $\longrightarrow$ (+ 1 2 3)

Macroexpansions en lisp

Syntaxe

```
(defmacro nom-macro (p1 p2 ... pn)
   corps)
```

Fonctionnement

- Constitution de l'environnement local : des liaisons sont établies entre les paramètres formels et les paramètres d'appel **non évalués**.
- Macroexpansion : le corps de la macro est évalué dans cet environnement, augmenté de l'environnement lexical de la macro.
- Le résultat de la macroexpansion est évalué dans l'environnement d'appel.

Exemple

```
(defmacro ifn (test e1 e2)
   `(if (not ,test) ,e1 ,e2))
> (macroexpand-1 '(ifn (= 1 2) (+ 2 1) (* 2 2)))
(IF (NOT (= 1 2)) (+ 2 1) (* 2 2))
```

Remarque : On constate que les arguments n'ont pas été évalués.

Problèmes d'évaluations multiples : en scheme, lisp et C

```
(define-syntax-rule (cube x)
   (* x x x))
```

- Problème d'utilisation

```
> (define a 1)
> (define-syntax-rule (++! x)
     (begin (set! x (add1 x))
            x))
> (cube (++! a)); -> (* (++! a) (++! a) (++! a))
24
```

- Pour remédier à ce problème, il faut créer des variables temporaires destinées à recevoir les valeurs des expressions fournies dans les paramètres.

```
(define-syntax-rule (cube x)
   (let ((tmp x))
     (* tmp tmp tmp)))

> (cube (++! a)); ->   (let ((tmp (++! a))
                ;         (* tmp tmp tmp)))

125
```

Question : évaluations multiples

```
(define-syntax-rule (for i start end body)
   (letrec ((loop (lambda (i)
                    (if (> i end) (void)
                        (begin body (loop (+ i 1)))))))
     (loop start)))
```

- Soit la session suivante, quel est le résultat de l'évaluation de a?

  0 ou 1 ou 2 ou 3

  3

```
> (define a 0)
> (for k 1 2 (set! a (+ a k)))
> a
```

i $\longrightarrow$ k start $\longrightarrow$ 1 end $\longrightarrow$ 2
body $\longrightarrow$ (set! a (+ a k))

- Soit la session suivante, combien de fois l'expression (* 1 2) est-elle évaluée dans la macro? 1 ou 2 ou 3 ou 4

  3

```
> (define a 0)
> (for i 1 (* 1 2) (set! a (+ a i)))
```

> (for i 1 (begin (print a) 2) (set! a (+ a i)))
013

Question : évaluations multiples

```
(define-syntax-rule (for i start end body)
   (let ((end-value end))
     (letrec ((loop (lambda (i)
                      (if (> i end-value) (void)
                          (begin body (loop (+ i 1)))))))
       (loop start))))
```

```
> (define a 0)
> (for i 1 (begin (print a) 2)  (set! a (+ a i)))
0
```

- Quel est le résultat de l'expression suivante Stack Overflow ou

  i: Undefined

  i: undefined

> (for i 1 (+ 1 i) (set! a (+ a i)))