Programmation fonctionnelle - PG104 -

COURS 6

Myriam Desainte-Catherine et David Renault

February 3, 2020

Filtrage de listes : fonction **remove**

La fonction **remove** prend en arguments un élément et une liste et elle renvoie la liste privée de la 1ère occurrence de l'élément. Elle admet un 3ième argument optionnel, qui est le prédicat de test de l'égalité. Par défaut c'est **equal?** qui est utilisé.

Exemple avec le prédicat par défaut

```
> (remove 2 '(1 2 3 2 4))
'(1 3 2 4)
> (remove* '(1 2) '(1 2 3 2 4)) ;Enleve toutes les occurrenc
'(3 4)
```

Exemple avec des prédicats passés en argument

```
> (remove 2 '(1 2 3 4) >)
'(2 3 4)
> (remove 2 '(1 2 3 4) (lambda (x y) (not (= x y))))
```

Devinez le résultat : '(2 3 4) ou '(1) ?

Voir aussi remove-dupliquates, remv, remq

Appartenance à une liste : fonction **member**

La fonction **member** prend en arguments un élément e et une liste l et elle renvoie **#f** si e n'appartient pas à l, et la liste l privée de ses éléments jusqu'à l'occurrence de e si celui-ci apparaît dans la liste. Le prédicat d'égalité utilisé est **equal?**.

Exemples

```
> (member 2 '(1 2 3))
'(2 3)
> (member 3 '(1 2 . 3))
'(1 2 . 3) : not a proper list
> (member 2 '(1 2 . 3))
'(2 . 3)
> (member 5 '(1 2 3 4))
#f
```

- Quel est le résultat de cette expression : `(member 2 '(3 2 1 2 4 5 2))` ?

#t ou #f ou '(2 4 5 2) ou '(2 1 2 4 5 2) ou '(2)

Voir aussi memv, memq, memf

Appartenance à une liste : fonction **member**

Exemple

```
> (member 3 '(3 2 1 2 4 5 2) >)
'(2 1 2 4 5 2)
> (member 3 '(3 2 1 2 4 5 2) <)
'(4 5 2)
```

Devinez le résultat de l'expression suivante :

```
> (member 2 '(3 2 1 2 4 5 2) >)
```

#t ou #f ou '(1 2 4 5 2) ou '(2 1 2 4 5 2)

Programmation récursive sur les listes

Définition récursive

- Soit liste vide
- Soit une paire pointée dont le *car* est un élément de la liste et le *cdr* la suite de la liste, soit la liste privée de son premier élément.

Spécification récursive d'un calcul

- Si la liste est vide, calculer le résultat correspondant.
- Sinon, exprimer le calcul en fonction de l'elément courant et du résultat d'un appel récursif sur la liste privée de son premier élément.

Somme des éléments d'une liste

Équations fonctionnelles :

- somme-liste(null) = 0
- somme-liste(l) = car(l) + somme-liste(cdr(l))

```
(define (somme-liste l)
   (if (empty? l)
       0
       (+ (car l)
          (somme-liste (cdr l)))))
```

Programmation récursive sur les listes : append et reverse

Concaténation de deux listes

concat(null, l2) = l2
concat(l1, l2) = cons(car(l1), concat(cdr(l1), l2))]

```
(define (concat l1 l2)
  (if (null? l1)
      l2
      (cons (car l1)
            (concat (cdr l1) l2))))
```

Cette fonction est-elle linéaire? oui ou non

Inversion d'une liste

inversion(null, l) = l
inversion(l1, l1) = inversion(cdr(l1)), cons(car(l1), l2)))

```
(define (inversion l1 l2)
  (if (null? l1)
      l2
      (inversion (cdr l1)) (cons (car l1) l2)))
```

Cette fonction est-elle récursive terminale? oui ou non

Programmation récursive sur les listes : append et reverse

Inversion naïve

inverse-naive(null) = null
inverse-naive(cons(a, l) = append(inverse-naive(l), list(a))

```
(define (inversion-naive l)
  (if (null? l)
      l
      (append (inversion-naive (cdr l))
              (list (car l)))))
```

Cette fonction est-elle linéaire? oui ou non

Cette fonction est-elle récursive terminale? oui ou non

Vers une version récursive terminale

```
(define (inversion-naive-r l1 l2)
  (if (null? l1)
      l2
      (inversion-naive-r (cdr l1)
                         (append l2 (list (car l1))))))
```

Programmation récursive sur les listes : append et reverse

Vers une version récursive terminale

```
(define (inversion-naive-r l1 l2)
  (if (null? l1)
      l2
      (inversion-naive-r (cdr l1)
                         (append l2 (list (car l1))))))
```

```
> (inversion-naive-r '(1 2 3 4) '())
```

`'(4 3 2 1)` ou `'(1 2 3 4)`

En rendant récursive terminale la fonction on a inversé l'ordre de la liste résultat : il faut ajouter au début et non à la fin

- (inversion-naive-r '(1 2 3 4) '())
- (inversion-naive-r '(2 3 4) (append '() (list 1)))
- (inversion-naive-r '(2 3 4) '(1))
- (inversion-naive-r '(3 4) (append '(1) (list 2)))
- (inversion-naive-r '(3 4) '(1 2))
- (inversion-naive-r '(4) (append '(1 2) (list 3)))
- (inversion-naive-r '(4) '(1 2 3))
- (inversion-naive-r '() (append '(1 2 3) (list 4)))

Programmation récursive sur les listes : **cons** et récursion terminale

iota1 récursive

iota1(0) = '()
iota1(n) = cons(n, iota1(n-1))

```
(define (iota1 n)
  (if (zero? n)
      '()
      (cons n
            (iota1 (sub1 n)))))
```

```
> (iota1 3)
```

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

iota2 récursive terminale

iota2(0, l) = l
iota2(n, l) = iota2(n-1, cons(n, l))

```
(define (iota2 n l)
    (if (zero? n)
      l
      (iota2 (sub1 n)
             (cons n l))))
```

```
> (iota2 3 '())
```

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

En inversant l'ordre d'un **cons** *et d'un appel récursif, on inverse l'ordre de la liste résultat – l'opération cons n'est ni associative ni commutative*

Les listes d'association : a-listes

Définition

C'est une liste de paires pointées. Le car de chaque paire est généralement une clef et ces listes servent à représenter des tables (indexées), des dictionnaires, des environnements.

Fonctions

- La fonction **assoc** admet deux paramètres : une clef et une a-liste. Elle parcourt la liste et renvoie la première paire pointée dont le car est égal au sens de *equal*? à la clef, et $\#f$ sinon.
- La fonction assq réalise le même travail avec *eq*? (assv pour *eqv*)

Les listes d'association : a-listes

Fonctions

- La fonction **assoc** admet deux paramètres : une clef et une a-liste. Elle parcourt la liste et renvoie la première paire pointée dont le car est égal au sens de *equal*? à la clef, et $\#f$ sinon.
- La fonction assq réalise le même travail avec *eq*? (assv pour *eqv*)

```
(define (add-alist1 a l)
   (cond ((assoc a l) l)
         (else (cons a l))))
```

```
>(define *carnet*
  '((Pierre "pierre@labri.fr")
    (Laure "Durand@laposte.net")
    (Laura "laura@twinpeaks.fr")))
> (assoc 'Laure *carnet*)
'(LAURE "Durand@laposte.net")
> (cdr (assoc 'Laure *carnet*))
'("Durand@laposte.net")
> (cadr (assoc 'Laure *carnet*))
"Durand@laposte.net"
```

> (add-alist1 '(Marc "phalippou@enseirb.fr") *carnet*)

'((Marc "phalippou@enseirb.fr") (Pierre "pierre@labri.fr") (Laure "Durand@laposte.net") (Laura "laura@twinpeaks.fr"))

ou '(Marc "phalippou@enseirb.fr") ou #f

> (add-alist1 '(Laura "laura@twinpeaks.fr") *carnet*)

'((Marc "phalippou@enseirb.fr") (Pierre "pierre@labri.fr") (Laure "Durand@laposte.net") (Laura "laura@twinpeaks.fr"))

ou '((Laura "laura@twinpeaks.fr") (Pierre "pierre@labri.fr") (Laure "Durand@laposte.net") (Laura "laura@twinpeaks.fr"))

ou '((Pierre "pierre@labri.fr") (Laure "Durand@laposte.net")

Les listes d'association : a-listes

Fonctions

- La fonction **assoc** admet deux paramètres : une clef et une a-liste. Elle parcourt la liste et renvoie la première paire pointée dont le car est égal au sens de *equal*? à la clef, et $\#f$ sinon.
- La fonction assq réalise le même travail avec *eq*? (assv pour *eqv*)

```
>(define *carnet*
  '((Pierre "pierre@labri.fr")
   (Laure "Durand@laposte.net")
   (Laura "laura@twinpeaks.fr")))
> (assoc 'Laure *carnet*)
'(LAURE "Durand@laposte.net")
> (cdr (assoc 'Laure *carnet*))
'("Durand@laposte.net")
> (cadr (assoc 'Laure *carnet*))
"Durand@laposte.net"
> (assoc 'hell-boy *carnet*)
#f
```

```
(define (add-alist2 a l)
   (cond ((assoc (car a) l) l)
         (else (cons a l))))
```

Programmation récursive sur les objets scheme

Objet scheme – Expression symbolique

- Soit un atome
- Soit une paire pointée admettant pour *car* et pour *cdr* un objet scheme (un atome ou une paire pointée)

Spécification récursive d'un calcul

- Si l'objet est un atome, calculer le résultat correspondant.
- Sinon, exprimer le calcul en fonction du fils gauche (le *car*), et du fils droit (le *cdr*).

```
(define (what a)
   (cond
     ((not (pair? a)) 1)
     (else
       (+ (what (car a))
          (what (cdr a))))))
```

Quels sont les résultats des appels suivants?

(what 1) 0 1 2

(what '(1 . 2)) 1 2 3

(what '()) 0 1 2

(what '((1 . 2) 3)) 2 3 4

(what '((1 2 (3)) () ((4 5) 6) 7)) 7 8 13

Statut de '() : atome ou liste vide?

Programmation récursive sur les objets scheme

Arbre binaire

- Soit l'arbre vide (null)
- Soit une feuille (atome non null)
- Soit une paire pointée admettant au moins un arbre non vide pour *car* ou pour *cdr*.

Spécification récursive d'un calcul

- Si l'arbre est vide, calculer le résultat correspondant.
- Si l'arbre est une feuille, calculer le résultat correspondant.
- Sinon, exprimer le calcul en fonction du fils gauche (le *car*), et du fils droit (le *cdr*).

```
(define (yawhat a)
   (cond
     ((null? a) 0)
     ((not (pair? a)) 1)
     (else
       (+ (yawhat (car a))
          (yawhat (cdr a))))))
```

Quels sont les résultats des appels suivants?

(yawhat 1)

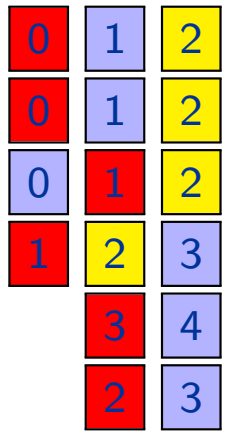
(yawhat '())

(yawhat '(1))

(yawhat '(1 . 2))

(yawhat '(1 2 3 ()))

(what '(1 () . 2))



Exercice : longueur récursive de listes de listes

L'argument est supposé être toujours une liste, donc son cdr est aussi une liste

Equations récursives

- length*('()) = 0
- length*(cons(cons(a , r1), r2)) = length*(cons(a,r1)) + length*(r2)
- length*(cons(a , r)) = 1 + length*(r)

```
(define (length* l)
   (match l
     ('() 0)
     ((cons (cons _ _) r2)
      (+ (length* (car l))
         (length* r2)))
     ((cons _ r)
      (add1 (length* r)))
     (_ (raise
         "length*:list required"))))
```

```
(define (yalength* l)
   (cond
     ((null? l) 0)
     (else
       (+ (if (pair?(car l))
              (yalength* (car l))
              1)
          (yalength* (cdr l))))))
```

Exercice : comparaison des fonctions

```
(define (what a)
   (cond
     ((not (pair? a)) 1)
     (else
       (+ (what (car a))
          (what (cdr a))))))
```

```
(define (yawhat a)
   (cond
     ((null? a) 0)
     ((not (pair? a)) 1)
     (else
       (+ (yawhat (car a))
          (yawhat (cdr a))))))
```

```
(define (yalength* l)
   (cond
     ((null? l) 0)
     (else
       (+ (if (pair?(car l))
              (yalength* (car l))
              1)
          (yalength* (cdr l))))))
```

Est-ce que les fonctions suivantes :

- Comptent la liste vide en car ?

yalength* | what | yawhat

- Comptent la liste vide en cdr ?

yalength* | what | yawhat

- Ne comptent jamais la liste vide ?

yalength* | what | yawhat

Il faut bien considérer le type des paramètres des fonctions, même si le langage ne nous y oblige pas