Programmation fonctionnelle - PG104 -

COURS 4

Myriam Desainte-Catherine et David Renault

January 29, 2020

Récursivité

Spécification récursive sous forme d'équations fonctionnelles

$$fact(0) = 1$$
$$fact(n) = n * fact(n-1)$$

Programme Scheme récursif

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

Récursivité

Evaluation

$>$ (fact 4)

$\longrightarrow$ (* 4 (fact 3))

$\longrightarrow$ (* 4 (* 3 (fact 2)))

$\longrightarrow$ (* 4 (* 3 (* 2 (fact 1))))

$\longrightarrow$ (* 4 (* 3 (* 2 (* 1 (fact 0)))))

$\longrightarrow$ (* 4 (* 3 (* 2 (* 1 1))))

$\longrightarrow$ (* 4 (* 3 (* 2 1)))

$\longrightarrow$ (* 4 (* 3 2))

$\longrightarrow$ (* 4 6)

$\longrightarrow$ 24

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

Pour évaluer l'application de cette fonction, une pile est nécessaire pour stocker les valeurs successives de n qui sont utilisées dans le calcul lors du retour des appels récursifs.

Récursivité et pile d'appels

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

Evaluation

> (fact 4)

⟶ (* n (fact (sub1 n)))

⟶ (* n (fact (sub1 n)))

⟶ (* n (fact (sub1 n)))

⟶ (* n (fact (sub1 n)))

Pile d'appels

n ⟶ 4

n ⟶ 3

n ⟶ 2

n ⟶ 1

Résultats

- 24
- 6
- 2
- 1

Est-il nécessaire d'empiler toutes les valeurs de n pour calculer la factorielle?

Oui | *Non* | *Je ne sais pas*

Récursivité et itération

Version impérative itérative

```
int
fact(int n)
{
  for (int r=1; n>0; n--)
    r*=n;
  return r;
}
```

Version fonctionnelle

```
(define (fact-t n r)
  (if (zero? n)
      r
      (fact-t (sub1 n)
              (* n r))))
```

Spécification fonctionnelle

$$\textit{fact-t}(0, r) = r$$
$$\textit{fact-t}(n, r) = \textit{fact-t}(n-1, n*r)$$

Récursivité terminale

Evaluation

> (fact-t 4 1)

⟶ (fact-t 3 4)

⟶ (fact-t 2 12)

⟶ (fact-t 1 24)

⟶ (fact-t 0 24)

⟶ 24

Version fonctionnelle

```
(define (fact-t n r)
  (if (zero? n)
      r
      (fact-t (sub1 n)
              (* n r))))
```

Les valeurs successives de n sont utilisées dans les calculs qui sont effectués avant les appels récursifs. Il est inutile de les conserver dans une pile.

Les appels récursifs sont dits terminaux car aucun calcul n'est effectué après leur retour.

Reconnaître des fonctions récursives terminales

Cette fonction est-elle récursive terminale?

```
(define (sum1 n)
   (if (zero? n)
       0
       (add1 (sum1 (sub1 n)))))
```

OUI ou NON

> (sum1 3)

⟶ (add1 (sum1 2))

⟶ (add1 (add1 (sum1 1)))

⟶ (add1 (add1 (add1 (sum1 0))))

⟶ (add1 (add1 (add1 0)))

⟶ (add1 (add1 1))

⟶ (add1 2)

⟶ 3

Reconnaître des fonctions récursives terminales

Cette fonction est-elle récursive terminale?

```
(define (sum2 n r)
   (if (zero? n)
       r
       (sum2 (sub1 n) (+ n r))))
```

OUI ou NON

> (sum2 3 0)

⟶ (sum2 2 3)

⟶ (sum2 1 5)

⟶ (sum2 0 6)

⟶ 6

Reconnaître des fonctions récursives terminales

Cette fonction est-elle récursive terminale?

```
(define (f n)
  (cond ((zero? n) 0)
        ((even? n) (add1 (f (sub1 n))))
        (else (f (sub1 n)))))
```

OUI ou NON

$>$ (f 3)

$\longrightarrow$ (f 2)

$\longrightarrow$ (add1 (f 1))

$\longrightarrow$ (f 0)

$\longrightarrow$ 1

Récursivité terminale – Ordre des calculs

Pour rendre une fonction récursive terminale, on déplace le calcul effectué après l'appel récursif pour le faire avant. Ceci modifie l'ordre des calculs.

Quelles propriétés doivent être vérifiées par les opérateurs concernés pour que le résultat n'en soit pas affecté?

Commutativité | Associativité

Factorielle

- Version non récursive terminale
- (* 4 (* 3 (* 2 (* 1 1))))

- Version récursive terminale
- (* 1 (* 2 (* 3 (* 4 1))))

Exemple

> (- 4 (- 3 (- 2 (- 1 1))))

- ??

> (- 1 (- 2 (- 3 (- 4 1))))

- ??

> (- 1 2 3 4 1)

- -9

> (- (- (- (- 1 2) 3) 4) 1)

- -9

Récursivité terminale – Pb de paramètres

Notre fonction fact-t demande un paramètre supplémentaire qui doit être initialisé à 1. Pour garantir son bon fonctionnement, il faut définir deux fonctions, l'une faisant l'appel initial et l'autre le calcul récursif terminal. Il y a deux solutions possibles.

Deux fonctions globales

```
(define (factorielle n)
   (fact-t n 1))
```

```
(define (fact-t n r)
  (if (zero? n)
      r
      (fact-t (sub1 n)
              (* n r))))
```

Une fonction globale avec une fonction locale

```
(define (factorielle n)
  (letrec ((fact-t
            (lambda (n r)
               (if (zero? n)
                   r
                   (fact-t (sub1 n)
                           (* n r))))))
    (fact-t n 1)))
```

Quelle version préférez-vous? la 1ère La 2ième

Récursivité et continuation fonctionnelle

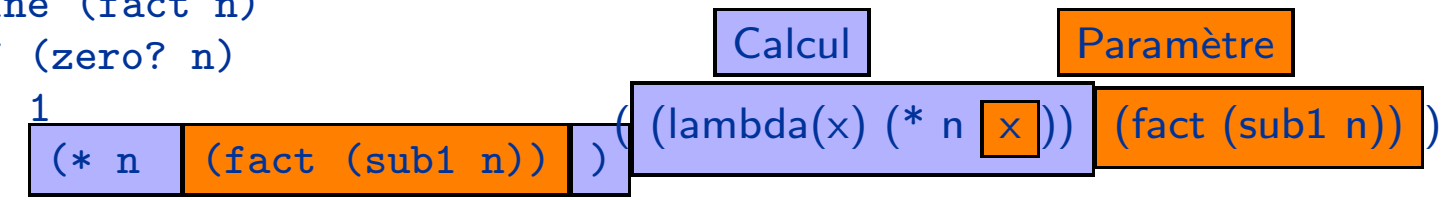
Notion de continuation

- Une continuation représente le futur d'un calcul, c'est-à dire le calcul à faire après le calcul d'une expression
- Une continuation est modélisée par une fonction qui s'applique au résultat d'une expression
- Elle représente un Goto fonctionnel

Comment construire la continuation d'un appel récursif?

- On effectue une abstraction de l'expression contenant le calcul récursif par une application : (k a)
- Le paramètre a de l'application est le résultat de l'appel récursif
- La fonction k est la continuation

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```



Récursivité et continuation fonctionnelle – Exemple

Continuations des appels récursifs de la fonction fact

> (fact 4) ———- (lambda(x) x) – **identity**

⟶ (∗ 4 (fact 3)) ——— (lambda(x) (∗ 4 x))

⟶ (∗ 4 (∗ 3 (fact 2))) ——— (lambda(x) (∗ 4 (∗ 3 x)))

⟶ (∗ 4 (∗ 3 (∗ 2 (fact 1)))) ———- (lambda(x) (∗ 4 (∗ 3 (∗ 2 x))))

⟶ (∗ 4 (∗ 3 (∗ 2 (∗ 1 (fact 0))))) —— (lambda(x) (∗ 4 (∗ 3 (∗ 2 (∗ 1 x)))))

⟶ (∗ 4 (∗ 3 (∗ 2 (∗ 1 1)))) —— Application de la continuation à 1

⟶ (∗ 4 (∗ 3 (∗ 2 1)))

⟶ (∗ 4 (∗ 3 2))

⟶ (∗ 4 6)

⟶ 24

Récursivité et continuation fonctionnelle – Utilisation

On ajoute un paramètre pour représenter la continuation, qui doit être initialisé à l'identité

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

```
(define (fact-c n k)
  (if (zero? n)
      (k 1)
      (fact-c (sub1 n)
              (lambda (x) (* n (k x))))))
```

> (fact-c 3 identity)

⟶ (fact-c 2 (lambda(x) (* 3 (**identity** x))))

⟶ (fact-c 1 (lambda(x) (* 2 ((**lambda**(x) (*3 (**identity** x))) x))))

⟶ (fact-c 0 (lambda(x) (* 1 ((**lambda**(x) (*2 ((**lambda**(x) (*3 (**identity** x))) x))) x))))

⟶ ((lambda(x) (* 1 ((lambda(x) (* 2 ((lambda(x) (* 3 (identity x))) x))) x))) 1)

-
```
((lambda(x) (* 1
               ((lambda(x) (* 2
                              ((lambda(x) (* 3
                                             (identity x)))
                               x)))
                x)))
 1)
```

Cette fonction est-elle récursive terminale? oui ou non

Récursivité terminale et continuation fonctionnelle

À votre avis, quand une fonction est déjà récursive terminale, la continuation a-t-elle une propriété particulière? oui non

La fonction fact-t

```
(define (fact-t n r)
  (if (zero? n)
      r
      (fact-t (sub1 n)
              (* n r))))
```

```
(define (fact-t-c n r k)
  (if (zero? n)
      (k r)
      (fact-t (sub1 n)
              (* n r)
              k)))
```

> (fact-t 4 1)

⟶ (fact-t 3 4) —— identity

⟶ (fact-t 2 12) —— identity

⟶ (fact-t 1 24) —— identity

⟶ (fact-t 0 24) —— identity

⟶ 24

Qu'est-ce qu'une liste?

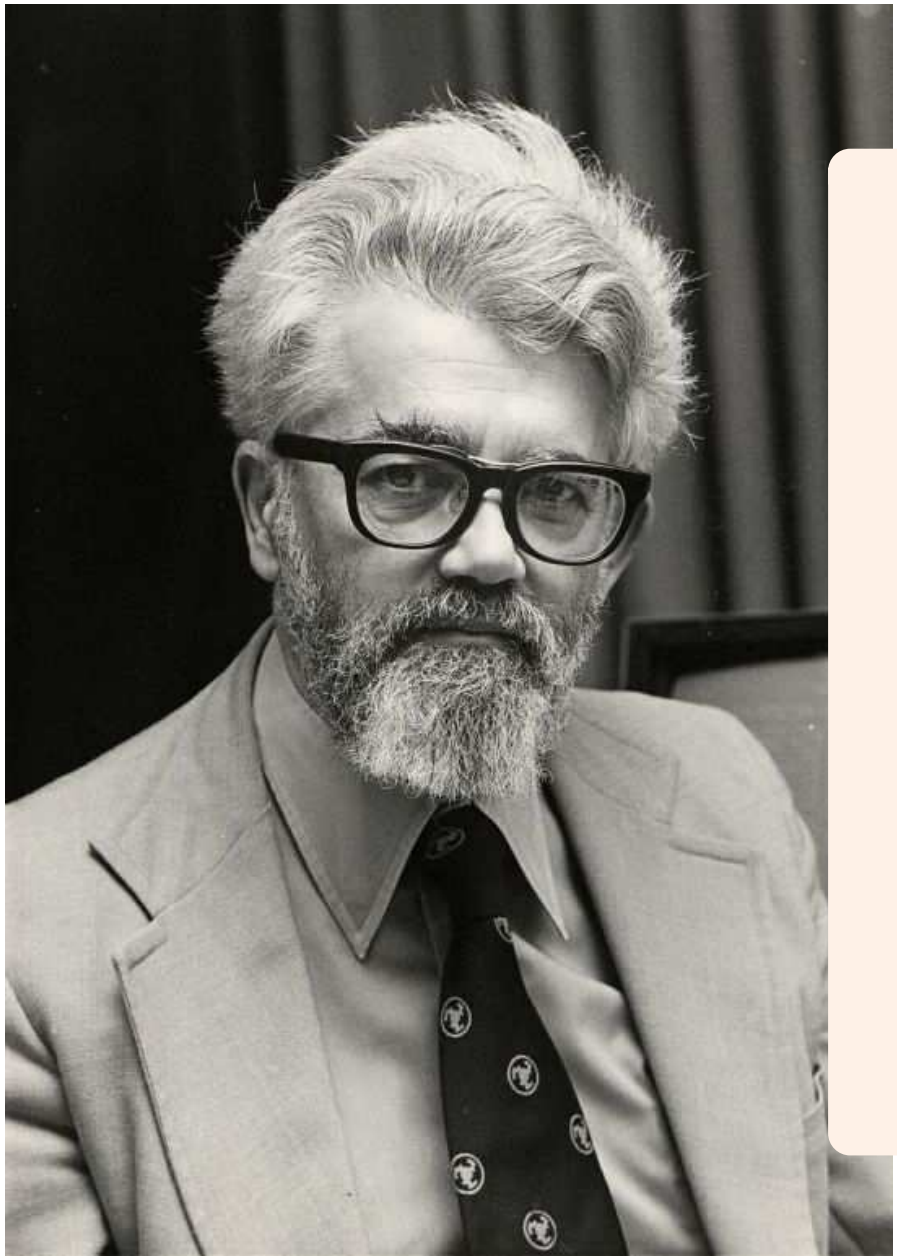
Caractéristiques

- Implémentation : un tableau ou une liste chaînée ?
- La taille d'une liste est-elle limitée lors de sa définition? oui ou non
- Une liste doit-elle être homogène ou bien peut-elle être hétérogène ?

Liste simplement chaînée

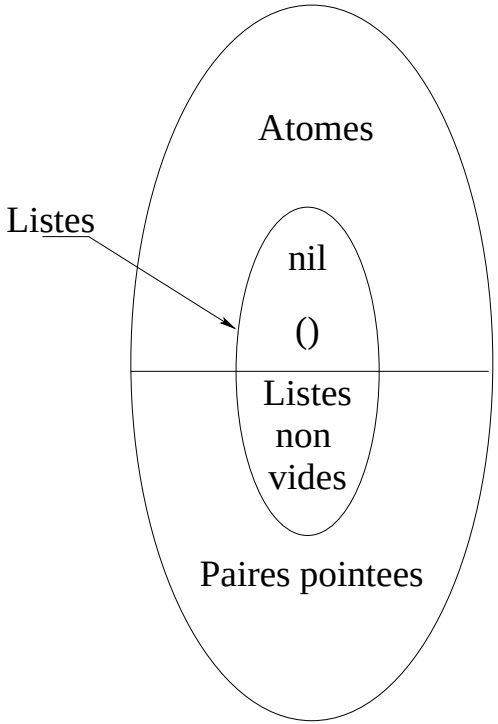
- Primitives : vide? longueur premier ième suivant ajouter

Listes lisp : John Mac Carthy

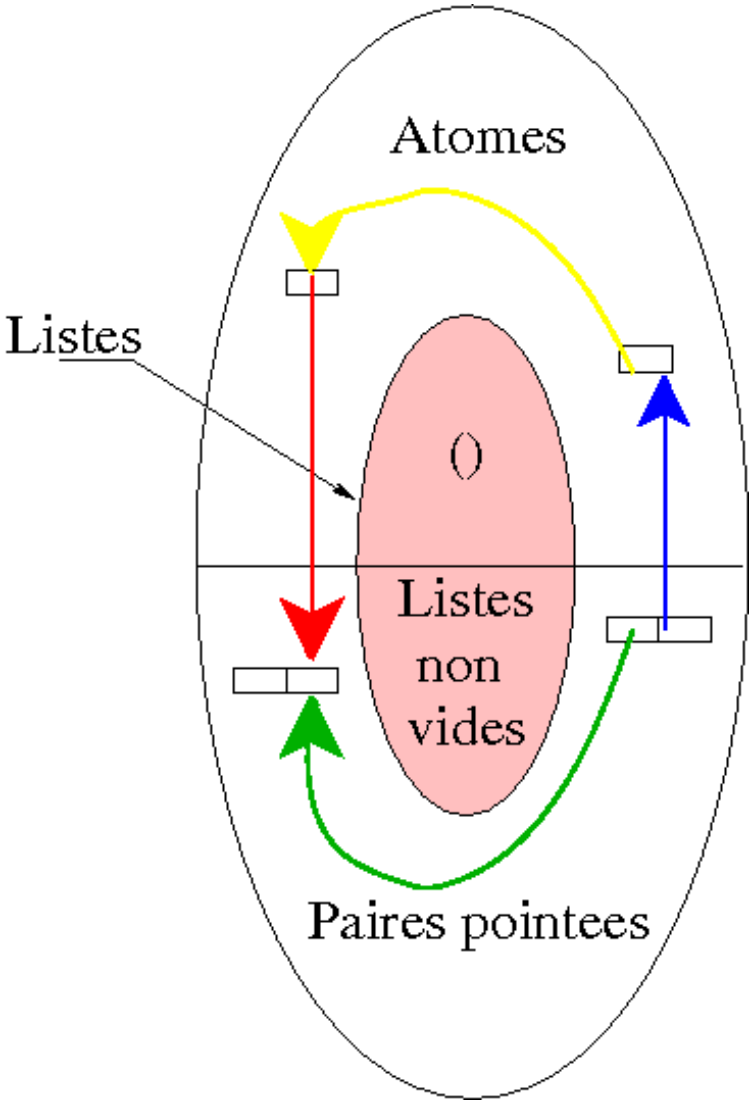


- Implémentation : Liste chaînée
- La taille d'une liste est-elle limitée? non
- Une liste peut être hétérogène
- Liste simplement chaînée
- Primitives : vide? premier suivant ajouter
- Liste récursive : objets primitifs
  - Vide : ()
  - Paire (pointée) : (a . b)
  - Valeurs (atomes)
- Métacircularité syntaxique

Les objets Scheme



Les objets Scheme



Quelles sont les flèches qui sont correctes?

Les objets

Les atomes

- Numériques,
- Booléens,
- Symboles,
- Chaînes de caractères
- Liste vide : ()
- Prédicat : **null**?

Les paires pointées

- Constructeur : **cons**
- Accesseurs : car, cdr
  (argument paire pointée uniquement)
- Prédicats : pair?, **cons**?
- Abbréviation : cddr, caadr, ..., cadddr, ..., list−ref

Forme quote et symboles

```
> (quote Pierre)
'Pierre
>'Pierre
'Pierre
> (define a 'Pierre)
> a
'Pierre
> Pierre
Error : Pierre undefined;
> '(define a 'Pierre)
'(define a 'Pierre)
```

Exemples

```
> (cons 1 2)
'(1 . 2)
> (cons (cons (cons 1 2) 3) 4)
'(((1 . 2) . 3) . 4)
> (cons 1 (cons 2 (cons 3 4)))
'(1 2 3 . 4)
> (car (cons 1 2))
1
> (cdr (cons 3 (cons 1 4))
'(1 . 4)
```

Forme spéciale quote

Quels sont les résultats des expressions suivantes ?

> (quote (+ 1 2))

3 ou '(+ 1 2)

> (quote (+ 1 2))

'(+ 1 2)

> '(+ 1 2)

'(+ 1 2) ou 3

> '(+ 1 2)

'(+ 1 2)

> '(1 (2))

'(1 (2)) ou Erreur

> '(1 (2))

'(1 (2))

> '(1 '(2))

'(1 (2)) ou '(1 '(2)) ou Erreur

> '(1 '(2))

'(1 '(2))

> '(())

'(()) ou Erreur

> '(())

'(())

> '('())

'(()) ou '('()) ou Erreur

> '('())

'('())