# Programmation fonctionnelle - PG104 -

# COURS 10

Myriam Desainte-Catherine et David Renault

February 20, 2020

# Qu'est-ce qu'un ramasse-miette?

La notion de ramasse-miettes a été inventée par John McCarthy (1960) pour le premier langage lisp.

## Récupération automatique de la mémoire

- Déterminer quels objets ne peuvent plus être utilisés par le programme
- Récupérer l'espace utilisé par ces objets.

## Avantages et inconvénients

- **Avantages** : programmes plus simples et plus sûrs, erreurs classiques évitées :
  - Accès à une zone non allouée, ou qui a été libérée
  - Libération d'une zone déjà libérée
  - Non-libération de la mémoire inutilisée (fuites mémoire)
- **Inconvénients**
  - Temps d'exécution de la phase de collectage des objets inaccessibles non borné.
  - Non adapté au temps réel.

Depuis les années 90, beaucoup de langages intègrent un ramasse-miette : Common Lisp, Java, C#, Go, Lisaac, Objective-C, Perl, JavaScript, Ruby, ActionScript, PHP, Lua ML, Visual Basic, Python, Scheme, Haskell, Smalltalk, Prolog, Bash, Csh, ...

# Qu'est-ce qu'un ramasse-miette?

## Problème

- **Un tas** : graphe orienté ayant une racine et dont les sommets sont les objets et les arêtes les références entre les objets.
- **Critère d'accessibilité** :
  - Les racines : pile, environnement global
  - Tout objet référencé depuis un objet accessible est lui-même accessible.

## Algorithmes

- **Algorithmes traversants** bloquants (Mark and Sweep - John McCarthy) : "stop-the-word" collector. Fonctionnent en deux passes :
  - Le marquage : chaque objet atteignable depuis la racine est marqué
  - Le nettoyage de la mémoire : les objets non marqués sont libérés.
- **Algorithmes incrémentaux** : utilisation de compteurs de références - problème avec les structures circulaires.
- **Algorithmes à générations** : selon la durée de vie des objets.

# Emacs-lisp

- Emacs-lisp n'a pas de fermetures, les fonctions ne sont représentées que par leur code (sans l'environnement lexical). Soient les fonctions :

```
(defun o(g f) (lambda(x) (funcall g (funcall f x))))
(defun carre (x) (* x x))
```

- Lors de l'application :

```
(funcall (o #'carre #'1-) 2)
```

- On obtient :

```
Debugger entered--Lisp error: (void-variable g)
  (funcall g (funcall f x))
  (lambda (x) (funcall g (funcall f x)))(2)
  funcall((lambda (x) (funcall g (funcall f x))) 2)
  eval((funcall (o (function carre) (function 1-)) 2)
```

# Programmation de fonctionnelles récursives

Première ébauche d'écriture d'une fonction de composition d'une liste de fonctions : ici, tout le calcul est gelé par la $\lambda$-expression. Il s'éxecutera entièrement au moment de l'application de la fonction résultat.

## Exemple

```
(define (bad1 . l)
  (lambda(x)
    (if (null? l)
        x
        ((car l) ((apply bad1 (cdr l)) x)))))

> (define add2 (bad1 add1 add1))
(lambda(x) ; le resultat est la lambda-expression
  (if (null? '(add1 add1))
      x
      (add1 ((bad1 add1) x))))
```

# Application du résultat de bad1

Soit l'application suivante

```
> (add2 1)
((lambda(x)
    (if (null? '(add1 add1))
        x
        (add1 ((bad1 add1) x))))
 1)
```

Choisir une réduction parmi :

- Application de la lambda
- Calcul du if
- Appel récursif
- Forme normale (pas de réduction possible)

# Application du résultat de bad1 (1)

```
( (lambda(x)
    (if (null? '(add1 add1))
        x
        (add1 ((bad1 add1) x))))
1)
```

$\xrightarrow{\beta}$

```
(if (null? '(add1 add1))
    1
    (add1 ((bad1 add1) 1)))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application du résultat de bad1 (2)

```
(if (null? '(add1 add1))
    1
    (add1 ((bad1 add1) 1)))
```

$\xrightarrow{\beta}$

```
(add1 ((bad1 add1) 1))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application du résultat de bad1 (3)

```
(add1 ( (bad1 add1) 1))
```

$\xrightarrow{\beta}$

```
(add1 ((lambda(x)
         (if (null? '(add1))
             x
             (add1 ((bad1) x))))
       1))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application du résultat de bad1 (4)

```
(add1 ( (lambda(x)
          (if (null? '(add1))
              x
              (add1 ((bad1) x))))
        1))
```

$\xrightarrow{\beta}$

```
(add1 (if (null? '(add1))
          1
          (add1 ((bad1) 1))))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application du résultat de bad1 (5)

```
(add1 (if (null? '(add1))
          1
          (add1 ((bad1) 1))))
```

$\xrightarrow{\beta}$

```
(add1 (add1 (bad1) 1))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application du résultat de bad1 (6)

```
(add1 (add1 ( (bad1)  1)))
```

$\xrightarrow{\beta}$

```
(add1 (add1 (if (null? '())
                1
                (add1 ((bad1) 1)))))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application du résultat de bad1 (7)

```
(add1 (add1 (if (null? '())
                1
                (add1 ((bad1) 1)))))
```

$\xrightarrow{\beta}$

```
(add1 (add1 1))
(add1 2)
3
```

Forme normale

# Deuxième ébauche d'écriture : bad2

- On sort le if de la $\lambda$-expression afin qu'il s'éxécute lors de l'application de la fonctionnelle.

```
(define (bad2 . l)
  (if (null? l)
      identity
      (lambda (x)
        ((car l) ((apply bad2 (cdr l)) x)))))
```

- Application de bad2 :

```
> (define add2 (bad2 add1 add1))
```

```
(if (null? (list add1 add1))
    identity
    (lambda(x)
      (add1 ((bad2 add1) x))))
```

- **Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de bad2 (1)

```
(if (null? (list add1 add1))
    identity
    (lambda(x)
      (add1 ((bad2 add1) x))))
```

$\xrightarrow{\beta}$

```
(lambda(x)
   (add1 ((bad2 add1) x)))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de bad2 (1)

```
(lambda(x)
   (add1 ((bad2 add1) x)))
```

Les calculs restants se déroulont lors de l'application de la fonction résultat `add2` : `(add2 1)`

```
((lambda(x)
   (add1 ((bad2 add1) x)))
 1)
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de add2 (1)

```
((lambda(x)
    (add1 ((bad2 add1) x)))
 1)
```

$\xrightarrow{\beta}$

```
(add1 ((bad2 add1) 1))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de add2 (2)

```
(add1 ((bad2 add1) 1))
```

$\xrightarrow{\beta}$

```
(add1 ((if (null? (list add1))
           identity
           (lambda(x)
               (add1 ((bad2) x))))
       1))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de add2 (3)

```
(add1 (
```

```
(if (null? (list add1))
    identity
    (lambda(x)
        (add1 ((bad2) x))))
```

```
1))
```

$\xrightarrow{\beta}$

```
(add1 ((lambda(x)
          (add1 ((bad2) x)))
       1))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de add2 (4)

```
(add1 ( (lambda(x)
          (add1 ((bad2) x)))
  1))
```

$\xrightarrow{\beta}$

```
(add1 (add1 ((bad2) 1)))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de add2 (5)

```
(add1 (add1 ( (bad2)  1)))
```

$\xrightarrow{\beta}$

```
(add1 (add1 ((if (null? '())
                 identity
                 (lambda(x) (add1 ((bad2) x))))
             1)))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de add2 (6)

```
(add1 (add1 ((if (null? '())
                 identity
                 (lambda(x) (add1 ((bad2) x))))
             1)))
```

$\xrightarrow{\beta}$

```
(add1 (add1 (identity 1)))
(add1 (add1 1))
(add1 2)
3
```

Forme normale

# Solution récursive terminale

Pour être sûr de mettre l'appel récursif dans la fonctionnelle, il faut rendre celle-ci récursive terminale.

```
(define (o f . l)
  (if (null? l)
      f
      (apply o (lambda(x) (f ((car l) x)))
             (cdr l)))))
```

Lors de l'application de la fonctionnelle **o**, tous les calculs s'effectuent, sauf ceux qui nécessitent de connaître l'argument de la fonction résultat.

# Application de **o**

```
> (define sqr1+ (o add1 sqr))
```

```
(if (null? (list sqr))
    add1
    (o (lambda(x)
         (add1 (sqr x)))))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de **o** (1)

```
(if (null? (list sqr))
    add1
    (o (lambda(x)
         (add1 (sqr x)))))
```

$\xrightarrow{\beta}$

```
(o (lambda(x)
     (add1 (sqr x))))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de **o** (2)

```
(o (lambda(x)
     (add1 (sqr x))))
```

$\xrightarrow{\beta}$

```
(if (null? '())
    (lambda(x)
      (add1 (sqr x)))
    (o (lambda(x) ...)))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

# Application de **o** (3)

```
(if (null? '())
    (lambda(x)
      (add1 (sqr x)))
    (o (lambda(x) ...)))
```

$\xrightarrow{\beta}$

```
(lambda(x)
   (add1 (sqr x)))
```

**Étape suivante** : $\lambda$ ou if ou récursion ou forme normale

forme normale

# Application de **sqr1+**

Les calculs suivants sont réalisés lors de l'application de la fonction `sqr1+`. Ils ne concernent que ceux qui dépendent de son argument.

```
> (sqr1+ 1)
```

```
((lambda(x)
    (add1 (sqr x)))
  1)
```

$\xrightarrow{\beta}$

```
(add1 (sqr 1))
```

# Composition de fonctions avec une fonctionnelle de liste

```
(define (o f . l)
  (if (null? l)
      f
      (apply o (lambda(x) (f ((car l) x)))
             (cdr l)))))
```

# Composition de fonctions avec une fonctionnelle de liste

```
(define (o f . l)
  (if (null? l)
      f
      (apply o (lambda(x) (f ((car l) x)))
               (cdr l)))))
```

Peut-on éviter d'écrire la récursivité à la main en utilisant une fonctionnelle telle que : map ou apply ou fold ou aucune fold

# Composition de fonctions avec une fonctionnelle de liste

Quelle forme `fold` faut-il utiliser pour composer les fonctions? o1 ou o2 o2

```
(define (o1 . l)
  (foldl (lambda (f g)
           (lambda (x)
             (f (g x))))
         identity
         l))
```

```
(define (o2 . l)
  (foldr (lambda (f g)
           (lambda (x)
             (f (g x))))
         identity
         l))
```

# Composition de fonctions avec une fonctionnelle de liste

Quelle fonction est préférable? o1 ou o2 o1

Quelle fonction est récursive terminale?

```
(define (o1 . l)
  (foldl (lambda (f g)
           (lambda (x)
             (g (f x))))
         identity
         l))
```

```
(define (o2 . l)
  (foldr (lambda (f g)
           (lambda (x)
             (f (g x))))
         identity
         l))
```

# Conclusion

## Règles d'écriture

- L'appel récursif doit être effectué par la fonctionnelle plutôt que par la fonction résultat. De cette façon, on effectue la boucle une seule fois au moment de la construction, sinon, la boucle est effectuée à chaque application de la fonction résultat.
- Pour ne pas geler l'appel récursif de la fonctionnelle, il faut qu'il soit extérieur à toute fermeture ($\lambda$-expression), ce qui implique de constuire les fermetures en arguments plutôt qu'en valeurs de retour d'appels récursifs, et donc de rendre les fonctionnelles récursives terminales.

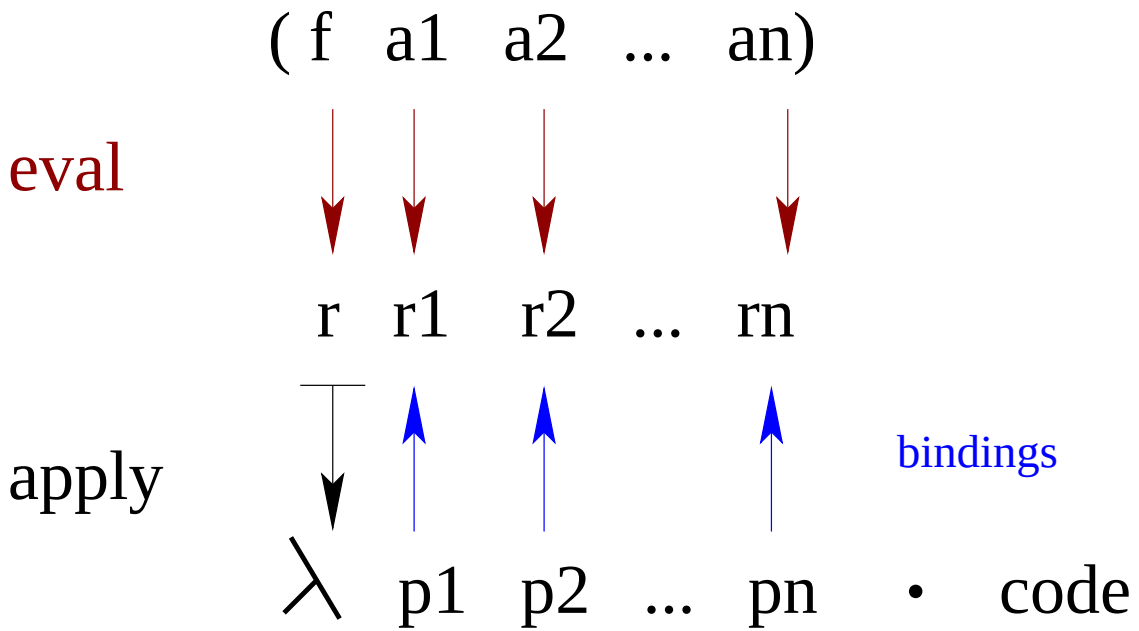
# Formes impératives

## Références

*Une référence est un objet correspondant à une adresse mémoire et dont l'indirection est faite automatiquement dans toute situation où une valeur est requise. L'adresse associée à une référence n'est pas directement manipulable en tant que telle (il n'existe pas d'opérations pour le programmeur sur les références)*

- Un symbole est lié à une référence, correspondant à un atome ou une paire pointée.
- L'évaluation d'un symbole renvoie une référence vers sa valeur.
- La référence est utilisée partout où la valeur n'est pas requise.
- On trouve des références dans d'autres langages : Java, C++.

# Passage d'arguments



# Passage d'arguments

Soit f une fonction, soient p1, p2, ... pn ses paramètres formels. Soit l'application :

$$(f\ a1\ a2\ ...\ an)$$

Soient r1, r2, ..., rn les références vers les résultats des évaluations respectives des arguments a1, a2, ..., an.
Lors de l'application, un environnement local est construit. Il est constitué des liaisons entre les paramètres formels pi de la fonction f et les références ri des arguments de l'application.

$$((p1\ .\ r1)(p2\ .\ r2)...(pn\ .\ rn))$$

Les références r1, r2, ..., rn sont utilisées comme des valeurs à travers les symboles p1, p2, ..., pn, les indirections étant effectuées automatiquement. Ainsi, il est impossible de modifier un paramètre pi, car la modification reste locale à cet environnement.

# L'affectation

## La forme **set!**

(**set!** ⟨*id*⟩ ⟨*e*⟩)

- La référence associée à l'identificateur ⟨*id*⟩ est remplacée par la référence du résultat de l'évaluation de l'expression ⟨*e*⟩.
- La valeur de retour de l'affectation est la valeur $\#<void>$ que la fonction `read` n'affiche pas. La procédure `void` rend ce même résultat en prenant un nombre quelconque d'arguments.

## Modification de paires pointées

On ne peut pas modifier les paires pointées de base dans la norme scheme. En Racket, il faut utiliser le paquetage `mpair`

## Exemple

```
> (define mp (mcons 1 2))
> (set-mcar! mp 2)
> mp
(mcons 2 2)
```

# Question : affectation et let

Soit l'expression suivante, quel est son résultat?

```
> (let ((x 1)
        (y 2))
    (set! x 3)
    (set! y 1)
    (cons x y))
```

'(3 . 1) ou '(1 . 2) ou erreur

'(3 . 1)

# Modification de paramètres

On se donne la session suivante :

```
(define (incrementer x)
  (set! x (add1 x)))
> (incrementer 2)
> (define i 0)
> (incrementer i)
```

Quel est le résultat de cette expression?

0 ou 1

```
> i
```