

Paradigmes et Interprétation

Introduction

Julien Provillard

julien.provillard@univ-cotedazur.fr

PRÉSENTATION GÉNÉRALE

Organisation du cours

□ Volume horaire

- 12 x 1h30 de CM
- 12 x 1h30 de TP

□ Evaluation

- TP notés : 30%
- Partiel : 30%
- Examen terminal : 40%

Rendu des TP

- ☐ Tous les TP à partir du 2^{ème} sont notés.
- ☐ Sauf indication contraire, vous avez jusqu'au dimanche de la semaine qui suit le cours. La deadline est à 23h59.
- ☐ Pour chaque TP, vous devez déposer sur moodle un unique fichier Racket `tpxx_yyyyyyyy.rkt` où `xx` est le numéro du tp et `yyyyyyyy` est votre identifiant étudiant (2 lettres, 6 chiffres).
- ☐ Exemple pour le TP n°2 : `tp02_pj921309.rkt`
- ☐ Respectez **strictement** toutes les consignes : la correction est semi-automatisée.

Rendu des TP

- ❑ Toute intervention de ma part dans votre code sera sanctionnée. Les raisons pour lesquels je peux intervenir :
 - **Nom incorrect par rapport au sujet.** Les algorithmes de test ne peuvent pas fonctionner.
 - **Erreur de syntaxe ou de typage.** Le code ne peut pas être interprété. Si vous bloquez sur un point, gardez le reste des fonctionnalités opérationnelles.
 - **Bugs mineurs.**

- ❑ La triche est détectée et sanctionnée par un 0. Vous pouvez partager des idées mais pas du code. Aucune distinction n'est faite entre ceux qui fournissent du code et ceux qui utilisent le code d'un tiers.

Qu'est-ce qu'un paradigme?

- ☐ Ensemble de concepts et de méthodes de programmation autorisés ou interdits.
- ☐ Ne change pas ce que l'on peut exprimer mais la manière dont on l'exprime.

Exemples

- ❑ **Programmation fonctionnelle.** Un programme est une fonction (au sens mathématique du terme) et son exécution est l'évaluation de cette fonction.
- ❑ **Programmation impérative.** Un programme est une suite d'instructions qui modifient l'état du système.
- ❑ **Programmation structurée.** Variante de la programmation impérative, le flot d'exécution du programme est dicté par des structures de contrôle.
- ❑ **Programmation orientée objet.** Un programme est un ensemble de briques logicielles qui interagissent entre elles.

Exemples

- ❑ **Programmation événementielle.** Le programme réagit à des événements internes ou externes.
- ❑ **Programmation concurrentielle.** Plusieurs sous-programmes (threads) agissent simultanément sur des ressources communes.
- ❑ **Programmation logique.** Un programme est la description d'un problème dans un formalisme logique. Son exécution revient à l'application de règles de déduction.
- ❑ Et bien d'autres...

Objectifs du cours

- ☐ Appréhender les différents concepts qui apparaissent dans les paradigmes classiques.
- ☐ Être capable d'implémenter ces concepts dans un cadre minimaliste afin de comprendre leur fonctionnement.
- ☐ Réaliser l'impact des paradigmes sur la manière de programmer.

Méthodologie

- ☐ Présentation d'un concept.
- ☐ Analyse du concept.
- ☐ Définition d'un langage minimaliste où le concept apparaît.
- ☐ Implémentation d'un interpréteur pour le langage.

Concrètement

- ☐ Les langages définis auront une syntaxe proche de Racket.
- ☐ L'interpréteur sera lui-même écrit dans un dialecte de Racket.
- ☐ On utilisera l'IDE DrRacket.
- ☐ Référence : [Programming Languages: Application and Interprétation.](#)



Vous allez beaucoup programmer dans ce cours, mais ce n'est pas un cours de programmation!

Pourquoi Racket?

- ❑ Pour interpréter un programme, il faut :
 - Le découper en lexèmes (analyse lexicale)
 - En extraire la structure dans un arbre syntaxique (analyse syntaxique)
 - Ces phases vous seront expliquées en cours de Compilation
 - En Racket, elles sont gratuites (un programme est déjà sous la forme d'arbre syntaxique)
- ❑ Racket est un langage multi-paradigmes où on peut trouver la plupart des concepts exposés dans le cadre du cours.

LE LANGAGE RACKET

Racket, vraiment?

- ☐ En fait un dialecte : `plait`. Les fichiers sources devront débuter par `#lang plait`.
- ☐ Utilisation d'un noyau restreint de Racket, la maîtrise du langage **n'est pas** un prérequis!
- ☐ De nouvelles primitives pour la définition et l'utilisation de structures de données.
- ☐ Un système de typage.

Racket, généralité

- ❑ Une variante de LISP
- ❑ Langage fonctionnel en notation parenthésée préfixe
 - $1 + 2 * 3 \Rightarrow (+\ 1\ (*\ 2\ 3))$
 - $f(x, 3, z) \Rightarrow (f\ x\ 3\ z)$
- ❑ Pas ou peu de modification de la mémoire

Types de bases

- ❑ Les nombres : `1`, `3.1415927`, ...
 - ❑ Les chaînes de caractères : `"Hello world !"`, `"invalid input"`, ...
 - ❑ Les booléens : `#t` et `#f`
 - ❑ Les symboles : `'hello`, `'+`, ...
 - ❑ Les listes (littérales) : `'(1 2 3)`, `'(a b c d)`, `'((1 2) (3) (4 5 6))`, ...
- Le *quote* permet de définir des listes d'un type uniforme.
- ❑ Les s-expressions : ``(+ 1 (* 2 3))`, ``(f x (g (+ y 1)))`

La *backquote* permet de définir des s-expressions : ce sont des arbres permettant de représenter du code brut.

Définitions globales

```
(define one 1) ; une constante
```

```
one
```

```
--> 1
```

```
(define (add x y) (+ x y)) ; une fonction à deux paramètres
```

```
(add 1 2)
```

```
--> 3
```

```
(define (funOne) 1) ; une fonction sans paramètre
```

```
(funOne)
```

```
--> 1
```

Fonctions et lambda-expressions

```
(define mult (lambda (x y) (* x y)))
```

```
; <=> (define (mult x y) (* x y))
```

```
mult
```

```
--> #<procedure:....>
```

```
(lambda (x y) (* x y))
```

```
--> #<procedure:....>
```

```
((lambda (x y) (* x y)) 2 3)
```

```
--> 6
```

Variables locales

```
(define (quadruple x)
  (let ([double (+ x x)])
    (+ double double)))
(quadruple 5)
--> 20
```

```
(define (sommeCarre x y)
  (let ([carreX (* x x)]
        [carreY (* y y)])
    (+ carreX carreY)))
(sommeCarre 2 3)
--> 13
```

Variables locales

```
(define (quadruple x)
  (local [(define double (+ x x))])
    (+ double double)))
(quadruple 5)
--> 20
```

```
(define (sommeCarre x y)
  (local [(define carreX (* x x))
          (define carreY (* y y))])
    (+ carreX carreY)))
(sommeCarre 2 3)
--> 13
```

Listes

□ Création

empty

--> '()

(cons 0 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))

--> '(0 1 2 3 4)

'(0 1 2 3 4)

--> '(0 1 2 3 4)

(list 0 (+ 1 2) (* 3 4))

--> '(0 3 12)

Listes

□ Accès

```
(define L '(0 1 2 3 4))
```

```
(first L)
```

```
--> 0
```

```
(second L)
```

```
--> 1
```

```
(third L)
```

```
--> 2
```

```
(fourth L)
```

```
--> 3
```

```
(list-ref L 2)
```

```
--> 2
```

```
(rest L)
```

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

```
(length L)
```

```
--> 5
```

Listes

□ Prédicats

```
(define L '(0 1 2 3 4))
```

```
(empty? L)
```

```
--> #f
```

```
(cons? L)
```

```
--> #t
```

```
(equal? L (cons 0 '(1 2 3 4)))
```

```
--> #t
```

```
(member 5 L)
```

```
--> #f
```

Listes

□ Fonctions avancées

```
(build-list 5 add1)
--> '(1 2 3 4 5)
; (build-list n f)
; <=> (list (f 0) (f 1) ... (f (- n 1)))
```

```
(map add1 '(0 1 2 3 4))
--> '(1 2 3 4 5)
; (map f (list e1 e2 ... eN))
; <=> (list (f e1) (f e2) ... (f eN))
```

```
(map2 * '(0 1 2 3) '(4 5 6 7))
--> '(0 5 12 21)
; (map2 f (list e1 e2 ... eN) (list e1' e2' ... eN'))
; <=> (list (f e1 e1') (f e2 e2') ... (f eN eN'))
```


Listes

☐ Fonctions avancées

```
(foldl cons empty '(0 1 2 3 4))
--> '(4 3 2 1 0)
; (foldl f acc (list e1 e2 ... eN))
; <=> (f eN (f ... (f e2 (f e1 acc))...))
```

```
(foldr (lambda (x y) (cons x (cons x y)))
       empty '(0 1 2 3 4))
--> '(0 0 1 1 2 2 3 3 4 4)
; (foldr f acc (list e1 e2 ... eN))
; <=> (f e1 (f e2 (f ... (f eN acc))...))
```

Ces deux fonctions vous posent généralement problème. Il est toujours possible de s'en passer en les remplaçant par des fonctions intermédiaires récursives.

Branchements

```
(define (abs x)
  (if (< x 0)
      (- 0 x)
      x))
(abs -3)
--> 3
```

```
(define (sign x)
  (cond [(= x 0) 0]
        [< x 0] -1]
        [else 1]))
(sign -3)
--> -1
```

□ On dispose des tests et connecteurs classiques

- Pour les nombres : `=`, `>`, `<`, `<=`, `>=`
- Pour les autres types : `equal?`
- Les connecteurs : `and`, `or`, `not`

Instructions impératives

❑ Assignation (à éviter)

```
(define x 0)
```

```
(set! x 1)
```

❑ Affichage

```
(display "valeur de x : ")
```

```
(display x)
```

```
(display "\n")
```

❑ Chaînage d'instruction

```
(begin (set! x 2) x)
```

❑ Tests et erreurs

```
(test x 2)
```

```
(test/exn (error 'test "Exemple d'erreur...") "erreur")
```

```
(print-only-errors #t)
```

for, while?

```
void print_ex() {
    for(i = 0; i < 10; i++)
        printf("%d\n", i);
}
```

```
(define (print_ex)
  (local ((define (iter i)
    (when (< i 10)
      (display i)
      (display "\n")
      (iter (+ i 1))))))
  (iter 0)))
```

```
int log(int i) {
    int res = 0;
    while(i > 1) {
        i = i / 2;
        res++;
    }
    return res;
}
```

```
(define (log i)
  (local ((define (iter i res)
    (if (> i 1)
      (iter (/ i 2) (+ res 1))
      res))))
  (iter i 0)))
```

Et plait ?

- ❑ Le principal apport est un système de typage.
- ❑ Toute valeur est associée à un type
 - `3` => `Number`
 - `not` => `(Boolean -> Boolean)`
 - `+` => `(Number Number -> Number)`
 - `(list 1 2 3)` => `(Listof Number)`
 - `"hello world !"` => `String`
 - `'x` => `Symbol`
- ❑ On peut spécifier le type lors d'une définition, il est alors vérifié.
 - `(define x : Number 3) ; ok`
 - `(define y : Number #t) ; erreur`
- ❑ Dans le cas contraire, il est inféré.

Déclaration de types et fonctions

- Dans une fonction on peut déclarer le type des variables et le type de retour.

```
(define (f [x : Number] [y : Number]) : Number  
  (+ x (* 2 y)))
```

- Un type peut être polymorphe (instancié à l'évaluation), par exemple :

```
cons : ('a (Listof 'a) -> (Listof 'a))  
(cons 3 (list 2 1 0)) ; ok  
(cons #t (list #f)) ; ok  
(cons #t (list 2 1 0)) ; erreur
```

Définir de nouveaux types

```
(define-type FormeGeometrique ; nom du type  
  ; suivi de ses variantes  
  [Carre (cote : Number)]  
  [Rectangle (largeur : Number) (hauteur : Number)]  
  [Cercle (rayon : Number)])
```

; chaque variante définit un constructeur

```
(define carre (Carre 10))  
(define rectangle (Rectangle 5 10))  
(define cercle (Cercle 5))
```

Fonctions générées

☐ Prédicats

```
(Rectangle? carre)
```

```
--> #f
```

```
(Rectangle? rectangle)
```

```
--> #t
```

☐ Accesseurs

```
(Carre-cote carre)
```

```
--> 10
```

```
(Rectangle-hauteur rectangle)
```

```
--> 10
```

```
(Rectangle-largeur carre)
```

```
--> Rectangle-largeur: contract violation
```


Déconstruire un type utilisateur

```
(define (aire [f : FormeGeometrique]) : Number
  (cond
    [(Carre? f) (let ([c (Carre-cote f)])
                  (* c c))]
    [(Rectangle? f) (let ([l (Rectangle-largeur f)]
                          [h (Rectangle-hauteur f)])
                      (* l h))]
    [(Cercle? f) (let ([r (Cercle-rayon f)])
                   (* 3.1415927 (* r r)))]))
```

```
(define (aire [f : FormeGeometrique]) : Number
  (type-case FormeGeometrique f
    [(Carre c) (* c c)]
    [(Rectangle l h) (* l h)]
    [(Cercle r) (* 3.1415927 (* r r))]))
```

Déconstruire un type utilisateur

□ Structure générale du `type-case`

```
(type-case type valeur  
  [(variante1 id1 ... idk1) expr1]  
  ...  
  [(varianteN id1 ... idkN) exprN]  
  [else expr]))
```

- `valeur` doit être du type `type`.
- `kI` est le nombre de champs de `varianteI`.
- On cherche si `valeur` correspond à une des variantes énumérées.
 - Si c'est le cas, ses champs sont liés aux différents identificateurs et on évalue l'expression associée.
 - Sinon c'est l'expression dans la clause `else` qui est évaluée.
- La clause `else` est interdite si l'énumération des variantes est exhaustive, obligatoire dans le cas contraire.

Types utilisateurs avancés

□ Un type utilisateur peut être récursif et/ou polymorphe

```
(define-type (List 'a) ; polymorphe, 'a peut être n'importe quel type
  [Empty] ; pas de champ dans cette variante
  ; récursif, une variante a un champ du type que l'on décrit
  [Cons (head : 'a) (tail : (List 'a))]) ; deux champs (tête et queue de liste)
```

```
(define (Length [l : (List 'a)]) : Number
  (type-case (List 'a) l
    [(Empty) 0] ; pas de champ donc liste d'identificateurs vide
    [(Cons hd tl) (+ 1 (Length tl))]) ; deux identificateurs dans la liste
```

```
(Length (Cons 2 (Cons 1 (Cons 0 (Empty)))))
```

```
--> 3
```

Note sur le parenthésage

- ❑ Les parenthèses, les crochets et les accolades sont entièrement équivalents.
- ❑ Utiliser différentes formes de parenthésage sert **uniquement** à augmenter la lisibilité du code.
- ❑ Dans le cadre du cours, on utilisera la convention suivante :
 - Les parenthèses sont utilisées pour l'interpréteur.
 - Les accolades sont utilisées pour le langage interprété.
 - Les crochets apparaissent indifféremment dans les deux cas précédents.