UNIVERSITÉ
CÔTE D'AZUR

Paradigmes et Interprétation

Introduction

Julien Provillard
julien.provillard@univ-cotedazur.fr

UNIVERSITÉ
CÔTE D'AZUR

ORGANISATION

UNIVERSITÉ CÔTE D'AZUR

Organisation du cours

- Volume horaire
  - 12 x 2h00 de CM
  - 12 x 2h00 de TP
  - 3 x 2h00 de TD

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

UNIVERSITÉ CÔTE D'AZUR

Rendu des TP

- Tous les TP à partir du 2ème sont notés.
- Sauf indication contraire, la deadline est fixée au dimanche qui suit le cours à 23h59.
- Pour chaque TP, vous devez déposer sur moodle un unique fichier Racket nommé `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.

UNIVERSITÉ CÔTE D'AZUR

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 partagez 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.

UNIVERSITÉ
CÔTE D'AZUR

PRÉSENTATION ET MÉTHODOLOGIE

UNIVERSITÉ CÔTE D'AZUR

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.

UNIVERSITÉ CÔTE D'AZUR

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.

UNIVERSITÉ CÔTE D'AZUR

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…

UNIVERSITÉ CÔTE D'AZUR

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.

UNIVERSITÉ CÔTE D'AZUR

Méthodologie

- Présentation d'un concept.
- Analyse au travers d'exemples.
- Définition d'un langage minimaliste où le concept apparaît.
- Implémentation d'un interpréteur pour le langage.

UNIVERSITÉ CÔTE D'AZUR

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 !

UNIVERSITÉ CÔTE D'AZUR

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.

UNIVERSITÉ
CÔTE D'AZUR

LE LANGAGE RACKET

UNIVERSITÉ CÔTE D'AZUR

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 (des types produits et sommes comme en OCaml).
- Un système de typage.

UNIVERSITÉ CÔTE D'AZUR

Racket, généralité

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

UNIVERSITÉ CÔTE D'AZUR

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.

UNIVERSITÉ CÔTE D'AZUR

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
```

UNIVERSITÉ CÔTE D'AZUR

Fonctions et lambda-expressions

```
(define mult (lambda (x y) (* x y))) ; définition d'une fonction à la volée
; <=> (define (mult x y) (* x y))
mult
--> #<procedure:...>

(lambda (x y) (* x y)) ; la fonction peut rester anonyme
--> #<procedure:...>

((lambda (x y) (* x y)) 2 3) ; et même être appelée directement
--> 6
```

UNIVERSITÉ CÔTE D'AZUR

Variables locales

```
(define (quadruple x)
  (let ([double (+ x x)]) ; let permet d'introduire une variable locale
    (+ double double)))
(quadruple 5)
--> 20

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

UNIVERSITÉ CÔTE D'AZUR

Variables locales

```
(define (sommeCarre x y)
  (let ([carreX (* x x)]
        [carreY (* y y)]
        [somme (+ carreX carreY)])
        ; simultanément implique qu'un définition ne peut dépendre des précédentes
    somme))
(sommeCarre 2 3)
--> 13
```

UNIVERSITÉ CÔTE D'AZUR

Variables locales

```
(define (sommeCarre x y)
  (let* ([carreX (* x x)] ; la variante let* réalise les définitions successivement
         [carreY (* y y)]
         [somme (+ carreX carreY)])
    somme))
(sommeCarre 2 3)
--> 13

(let* ([id1 def1] [id2 def2] ... [idN defN]) body)
<=> (let ([id1 def1])
      (let ([id2 def2])
        ...
          (let ([idN defN]))
            body) ... ))
```

UNIVERSITÉ CÔTE D'AZUR

Variables locales

```
(define (quadruple x)
  (local [(define double (+ x x))] ; syntaxe alternative avec local-define
    (+ double double)))
(quadruple 5)
--> 20

(define (sommeCarre x y)
  (local [(define carreX (* x x)) ; qui permet aussi de définir plusieurs variables
          (define carreY (* y y))]
    (+ carreX carreY)))
(sommeCarre 2 3)
--> 13
```

UNIVERSITÉ CÔTE D'AZUR

Variables locales

```
(define (compose f g)
  (let ([h (lambda (x) (f (g x)))])
    h))
; introduire une fonction locale avec let impose d'utiliser un lambda

(define (compose f g)
  (local [(define (h x) (f (g x)))]
    h))
; dans ce cas la syntaxe local-define est plus simple
```

UNIVERSITÉ CÔTE D'AZUR

Variables locales

Les fonctions introduites par un let ne peuvent pas être récursives.

```
(define (is-prime? n)
  (let ([has-divisor?
         (lambda (k)
           (if (= k n)
               #f
               (or (= (modulo n k) 0)
                   (has-divisor? (+ k 1)))))])
    (and (>= n 2) (not (has-divisor? 2)))))
```

UNIVERSITÉ CÔTE D'AZUR

Variables locales

Il faut utiliser explicitement la variante letrec.

```
(define (is-prime? n)
  (letrec ([has-divisor?
            (lambda (k)
              (if (= k n)
                  #f
                  (or (= (modulo n k) 0)
                      (has-divisor? (+ k 1)))))])
    (and (>= n 2) (not (has-divisor? 2)))))
```

UNIVERSITÉ CÔTE D'AZUR

Variables locales

Ou la syntaxe local-define particulièrement adaptée.

```
(define (is-prime? n)
  (local [(define (has-divisor? k)
            (if (= k n)
                #f
                (or (= (modulo n k) 0)
                    (has-divisor? (+ k 1)))))]
    (and (>= n 2) (not (has-divisor? 2)))))
```

UNIVERSITÉ CÔTE D'AZUR

Listes

❑ Création

```
empty ; la liste vide
--> '()

(cons 0 (cons 1 (cons 2 (cons 3 (cons 4 empty))))) ; ajouts d'éléments
--> '(0 1 2 3 4)

'(0 1 2 3 4) ; création en bloc d'une liste de valeurs avec un quote
--> '(0 1 2 3 4)

(list 0 (+ 1 2) (* 3 4)) ; ou avec list si des calculs sont nécessaires
--> '(0 3 12)
```

UNIVERSITÉ CÔTE D'AZUR

Listes

- Accès

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

(first L) ; premier élément de la liste
--> 0
(second L) ; deuxième élément de la liste
--> 1
(third L) ; troisième élément de la liste
--> 2
(fourth L) ; quatrième élément de la liste
--> 3
(list-ref L 4) ; n-ième élément de la liste, pas de fonction primitive pour n >= 4
--> 4
(rest L) ; la liste égale à L privée de son premier élément
--> '(1 2 3 4)
(length L) ; le nombre d'éléments dans la liste
--> 5
```

UNIVERSITÉ CÔTE D'AZUR

Listes

Prédicats

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

(empty? L) ; la liste est-elle vide ?
--> #f

(cons? L) ; la liste est-elle non-vide ?
--> #t

(equal? L (cons 0 '(1 2 3 4))) ; égalité de deux listes
--> #t

(member 5 L) ; appartenance d'un élément à une liste
--> #f
```

UNIVERSITÉ CÔTE D'AZUR

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'))
```

UNIVERSITÉ CÔTE D'AZUR

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.

UNIVERSITÉ CÔTE D'AZUR

Branchements

```
(define (abs x)
  (if (< x 0) ; branchement classique à deux branches
      (- 0 x) ; si vrai
      x))     ; si faux
(abs -3)
--> 3

(define (sign x)
  (cond [(= x 0) 0]  ; branchement avec un nombre de branches quelconque
        [(< x 0) -1] ; on évalue la première ligne où la condition est vraie
        [else 1]))   ; une erreur se produit si aucune condition n'est vérifiée
(sign -3)
--> -1
```

- On dispose des tests et connecteurs classiques
  - Pour les nombres : `=`, `>`, `<`, `<=`, `>=`
  - Pour les autres types : `equal?`
  - Les connecteurs : `and`, `or`, `not`

UNIVERSITÉ CÔTE D'AZUR

Instructions impératives

- Assignation (à éviter)

```
(define x 0)
(set! x 1) ; mutation de x
```

- Affichage

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

- Chaînage d'instruction

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

- Branchement sans else

```
(when (< x 0) (display "x est négatif\n"))
(when cond e1 e2 ... eN) <=> (if cond (begin e1 e2 ... eN) (void))
```

UNIVERSITÉ CÔTE D'AZUR

Instructions impératives

Erreurs

```
(error 'symbol "message") ; lancement d'une erreur
; le message complet de l'erreur est "symbol: message"
```

Tests et erreurs

```
(print-only-errors #t) ; résultats de tous les tests ou seulement les échecs

(test expression resultat-attendu) ; test d'égalité de deux expressions
(test (+ 1 1) 2) ; ok
(test (+ x 2) (* x 2)) ; vrai si x = 2

; teste si une expression lance une erreur dont le message contient un certain motif
(test/exn expression "message part")
(test/exn (error 'symbol "un message") "message") ; ok
(test/exn (error 'symbol "un message") "symbol") ; ok
(test/exn (error 'symbol "un message") "un autre message") ; échec
```

UNIVERSITÉ CÔTE D'AZUR

Et les boucles ?

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

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

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

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

UNIVERSITÉ CÔTE D'AZUR

Qu'apporte `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.

UNIVERSITÉ CÔTE D'AZUR

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
```

UNIVERSITÉ CÔTE D'AZUR

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))
```

UNIVERSITÉ CÔTE D'AZUR

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
```

UNIVERSITÉ CÔTE D'AZUR

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))]))
```

UNIVERSITÉ CÔTE D'AZUR

Déconstruire un type utilisateur

Structure générale du `type-case`

```
(type-case type valeur
  [(variante_1 id_1 ... id_k1) expr_1]
  ...
  [(variante_N id_1 ... id_kN) expr_N]
  [else expr]))
```

- `valeur` doit être du type `type` .
- `kI` est le nombre de champs de $\texttt{variante}_I$.
- 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.

UNIVERSITÉ CÔTE D'AZUR

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
```

UNIVERSITÉ CÔTE D'AZUR

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.