Programmation fonctionnelle

Dr. Mounir El Araki Tantaoui

Avec l'aimable autorisation du Professeur Jean Paul Roy

http://deptinfo.unice.fr/~roy/

Agenda

- Langage d'expressions préfixées
- Les fonctions
- Programmer avec des images / Animations
- Programmer par récurrence
- Les listes (chainées)
- Les calculs itératifs
- **Type abstraits et généralisation**
- Les arbres binaires

- **1. L'abstraction des données**

- ***Maths***. Qu'est-ce qu'un *entier* ? Qu'est-ce qu'un *réel* ?

- Nous en avons une idée intuitive, nous pouvons même en nommer quelques uns : 25, 0.67, π... ***La plupart ne sont même pas calculables !***

- Nous ne savons pas vraiment *de quoi ils sont faits*. Heureusement nous connaissons leurs **propriétés** et leurs **opérations**. Et cela nous suffit !

- Les pros des maths s'occuperont de leur construction en temps utile...

- Quel rapport avec la programmation ?

- Nous avons programmé avec des objets élémentaires [nombres, booléens] et avec des objets composés [structures, images, listes].

- Supposons que nous souhaitions programmer avec des ***matrices***. Notre langage ne les a pas prévues : il ne pouvait pas tout prévoir !

- **IDEE 1 :** Nous allons **définir un nouveau type [abstrait] de donnée *matrice*** en utilisant des types de données déjà existants.

avec des structures, ou bien des listes, ou bien...

- **IDEE 2 :** Nous allons nous empresser d'**oublier la manière dont ils ont été construits** pour nous focaliser sur leur utilisation à travers un ensemble de fonctions *abstraites*.

une boîte à outils "matrice"

- Mise en œuvre du type abstrait « matrice 2×2 »

- Une **matrice 2×2** est la donnée de 4 nombres, organisés en lignes et colonnes. Il faut savoir construire une matrice et accéder à ses éléments.

$$\begin{pmatrix} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{pmatrix}$$

Version 1 : implémentation par des listes

```
(define (mat a b c d)
  (list (list a b) (list c d)))

(define (mat-ref M i j)      ; Mi,j
  (if (= i 1)
    (if (= j 1)
        (first (first M))
        (second (first M)))
    (if (= j 1)
        (first (second M))
        (second (second M)))))
```

```
> (define A (mat 1 2 3 4))
> A
((1 2) (3 4))
> (mat-ref A 2 1)
3
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Version 2 : implémentation par des structures

```
(define-struct matrix (e11 e12 e21 e22))

(define (matrice a b c d)
  (make-matrix a b c d))
```

```
> (define A (matrice 1 2 3 4))
> A
#(struct:matrix 1 2 3 4)
> (matrix-e21 A)
3
```

- Facile. Mais la facilité d'accès aux éléments est trompeuse, car je peux avoir besoin de **calculer les indices !** Une fonction d'accès :

```
(define (matrice-ref M i j)          ; Mi,j
  (if (= i 1)
    (if (= j 1)
        (matrix-e11 M)
        (matrix-e12 M))
    (if (= j 1)
        (matrix-e21 M)
        (matrix-e22 M))))
```

```
> (matrice-ref A 2 1)
3
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

- CONCLUSION : Il n'y a pas unicité de la représentation concrète d'une matrice abstraite. **L'important, c'est la matrice, pas sa représentation !** On *fait abstraction* de sa représentation...

- Quel que soit le choix de la structure concrète d'une matrice, on livre à l'utilisateur un ensemble de fonctions : ***la boîte à outils matrice 2×2*** : (*bibliothèque, module, classe...)*

```
(define (mat a b c d) ...)

(define (mat-ref M i j) ...)
```

- L'utilisateur n'a pas besoin de connaître le détail de l'implémentation.
- Une documentation suffit, avec la *complexité* des fonctions.
- Il va rédiger des algorithmes qui seront **indépendants de la représentation interne d'une matrice**. Exemple :

```
(define (det M)                          ; le déterminant
  (- (* (matrice-ref M 1 1) (matrice-ref M 2 2))
     (* (matrice-ref M 1 2) (matrice-ref M 2 1))))
```

```
> (det A)
-2
```

- Si je change de boîte à outils sur les matrices, la fonction (det M) **n'aura pas du tout à être modifiée** !

- PROBLEME : l'utilisateur verra peut-être la nature concrète d'une matrice s'il demande son affichage.

```
> A
((1 2) (3 4))
```

```
> A
#(struct:matrix 1 2 3 4)
```

- SOLUTION : une fonction mat->string calculant une représentation externe de la matrice sous forme de chaîne de caractères.

```
(define (mat->string M)     ; matrice → string
  (format "(~a\t~a)\n(~a\t~a)\n"
          (matrice-ref M 1 1)
          (matrice-ref M 1 2)
          (matrice-ref M 2 1)
          (matrice-ref M 2 2)))
```

```
> (printf "Matrice A:\n~a"
          (mat->string A))
Matrice A :
(1    2)
(3    4)
```

- **Un autre exemple : les nombres rationnels**

- Supposons que notre langage de programmation n'offre pas les **nombres rationnels exacts** comme 3/7, mais nous en avons besoin !
- On représentera *mathématiquement* un rationnel ≠ 0 par une **fraction irréductible** unique p/q avec q > 0 et pgcd(p,q) = 1.

- On choisit de représenter *informatiquement* un rationnel par une **structure à deux champs** : **numérateur** et **dénominateur**, mêmes conditions.
- La boîte à outils ***nombres rationnels*** doit contenir le minimum de fonctions dépendant du choix du type de donnée concrète, ici la structure. Les autres fonctions sur les nombres rationnels n'auront pas à savoir qu'un rationnel est représenté par une structure !

;;; Un type abstrait "nombre rationnel" en Scheme
;;; Implémentation par des structures.
;;; Fractions irréductibles à dénominateur > 0

```
(define-struct rat (numer denom))

(define (rationnel p q)                 ; le rationnel p/q avec p et q entiers
  (local [(define (fraction p q)        ; ici q est > 0
            (local [(define g (gcd p q))]
              (make-rat (/ p g) (/ q g))))]
    (cond ((= q 0) (error 'rationnel "dénominateur nul !"))
          ((> q 0) (fraction p q))
          (else (fraction (- p) (- q))))))

(define (numerateur r)                  ; le numérateur de r
  (rat-numer r))

(define (denominateur r)                ; le dénominateur de r
  (rat-denom r))

(define (rationnel? obj)
  (rat? obj))

(define (rat->string r)                 ; la représentation externe
  (cond ((= 0 (numerateur r)) "0")
        ((= 1 (denominateur r)) (format "~a" (numerateur r)))
        (else (format "~a/~a" (numerateur r) (denominateur r)))))
```

- Exemple d'utilisation, l'algorithme abstrait d'**addition dans Q** :

```
(define (rat+ r1 r2)                        ; j'utilise le Type Abstrait !
  (rat (+ (* (numerateur r1) (denominateur r2))
          (* (numerateur r2) (denominateur r1)))
       (* (denominateur r1) (denominateur r2))))
```

```
> (define r1 (rat 6 -4))
> (define r2 (rat 1 3))
> (define r3 (rat 0 -5))
> (printf "r1 = ~a\n" (rat->string r1))
r1 = -3/2
> (printf "r2 = ~a\n" (rat->string r2))
r2 = 1/3
> (printf "r3 = ~a\n" (rat->string r3))
r3 = 0
> (printf "r1 + r2 = ~a\n" (rat->string (rat+ r1 r2)))
r1 + r2 = -7/6
```

- Notez dans l'addition r1 + r2 l'obtention automatique d'une fraction irréductible !
- **Mais** : puis-je mélanger mes rationnels avec les nombres de Scheme ?

- **Vers une arithmétique générique**
- On a développé deux boîtes à outils : ***matrice 2×2*** et ***rationnel*** avec des fonctions mat->string et rat->string.
- On peut programmer une ***fonction générique*** qui va tester le type :

```
(define (toString num)
  ; num est un nombre Scheme ou une 'matrice' ou un 'rationnel'
  (cond ((number? num) (number->string num))
        ((mat? num) (mat->string num))
        ((rat? num) (rat->string num))
        (else (error 'toString "Type de nombre non reconnu !" num))))
```

```
> (printf "~a\n" (toString r1))
-3/2
> (printf "~a" (toString B))
(1    2)
(3    4)
> (printf "~a\n" (toString pi))
3.141592653589793
```

- En continuant dans cette direction, il faudra prévoir des **opérations génériques** comme en maths : additionner un entier et un *rationnel*, ou une *matrice* et un réel, etc.
- Pour additionner une matrice et un réel, il faudra savoir convertir un réel en matrice, donc introduire des **opérateurs de conversion de type**.
- Grosso modo, si x est construit dans le type E, il faut forcer x à être de type F le temps d'une opération... si c'est possible ! Exemple :

```
(define (real->matrice x)     ; conversion
  (matrice x 0 0 x))

(define (add num1 num2)       ; addition générique
  (cond ((and (real? num1) (matrice? num2)) (mat+ (real->matrice num1) num2))
        .....))
```

$$x = \begin{pmatrix} x & 0 \\ 0 & x \end{pmatrix}$$

```
> (printf "~a\n" (toString (add pi A)))
(4.141592653589793    2)
(3      7.141592653589793)
```

$real \times matrice \longrightarrow matrice$

- **2. L'abstraction des traitements**
- Un programmeur passe son temps à écrire les mêmes algorithmes !
- Comment lui faire gagner du temps et l'entraîner à raisonner en termes de ***classes*** d'algorithmes, de ***schémas*** algorithmiques ?...
- REPONSE : **en faisant abstraction des traitements particuliers, donc en généralisant : passer le traitement [une fonction] en paramètre !**

Définition : Une fonction $f : E \to F$ est **d'ordre supérieur** si son domaine de départ ou d'arrivée contient des fonctions.

$(Num \to Num) \to Num$

$f \mapsto f(0)$

```
(define (val0 f)
  (f 0))
```

$(Num \to Num) \to (Num \to Num)$

$f \mapsto f'$

```
(define (D f)           ; dérivée approchée
  (lambda (x)
    (/ (- (f (+ x #i0.01) (f x)) #i0.01)))
```

- Soit à calculer la somme des entiers de a jusqu'à b, avec a ≤ b :

```scheme
(define (sigma1 a b)    ; a + (a+1) + (a+2) + ... + b
  (if (> a b)
      0
      (+ a (sigma1 (+ a 1) b))))
```

```
10 + 11 + 12 + ... + 19 + 20
> (sigma1 10 20)
165
```

- Mais cette fonction ne me permet pas de calculer la somme des carrés. Généralisons en prenant les images par une fonction f :

```scheme
(define (sigma2 a b f)            ; f(a) + f(a+1) + f(a+2) + ... + f(b)
  (if (> a b)
      0
      (+ (f a) (sigma2 (+ a 1) b f))))
```

```
10 + 11 + 12 + ... + 19 + 20
> (sigma2 10 20 identity)
165
```

$10^2 + 11^2 + 12^2 + ... + 19^2 + 20^2$

```
> (sigma2 10 20 sqr)
2585
```

- Qui peut le plus peut le moins ...

- Généralisons en prenant une ***fonction de pas***. Le suivant de a ne sera pas forcément (+ a 1) mais (s a) :

```
(define (sigma3 a b f s)              ; f(a) + f(s(a)) + f(s(s(a))) + ...
  (if (> a b)
      0
      (+ (f a) (sigma3 (s a) b f s))))
```

$10 + 11 + 12 + ... + 19 + 20$

```
> (sigma3 10 20 identity add1)
165
```

$10^2 + 12^2 + 14^2 + ... + 20^2$

```
> (sigma3 10 20 sqr (λ (x) (+ x 2)))
1420
```

- Généralisons en prenant un prédicat fini? servant de test d'arrêt :

```
(define (sigma4 a f s fini?)
  (if (fini? a)
      0
      (+ (f a) (sigma4 (s a) f s fini?))))
```

$10^2 + 12^2 + 14^2 + ... + 20^2$

```
(sigma4 10 sqr (λ (x) (+ x 2)) (λ (x) (>
x 20)))
```

- # Ordre supérieur sur les listes : le schéma **map**

- On a souvent besoin d'**appliquer la même fonction sur tous les éléments d'une liste** [***en parallèle***], et de récolter la liste des résultats.

```
> (map sqr (list 1 2 3 4 5))
(1 4 9 16 25)
> (map (lambda (x) (* x 2)) (list 1 2 3 4 5))
(2 4 6 8 10)
```

```
(define ($map f L)
  (if (empty? L)
      L
      (cons (f (first L)) ($map f (rest L)))))
```

- (map f L1 ... Ln) lorsque f accepte n arguments :

```
> (map (lambda (x y) (+ x (* 2 y))) '(1 2 3) '(4 5 6))
(9 12 15)
> (map + '(1 2 3) '(4 5 6))
(5 7 9)
```

Toutes les listes de même longueur !

- Pensez en maths à une fonction comme sqr qui s'applique en même temps à toutes les composantes d'un vecteur...

- Ordre supérieur sur les listes : le schéma **filter**
- On a souvent besoin de ne conserver que les éléments d'une liste vérifiant une condition :

```
(define (impairs L)     ; les nombres impairs d'une liste numérique L
  (cond ((empty? L) L)
        ((odd? (first L)) (cons (first L) (impairs (rest L))))
        (else (impairs (rest L)))))
```

```
> (impairs '(5 2 3 1 6))
(5 3 1)
```

- On utilise plutôt la fonction primitive (filter **pred?** L), où **pred?** Est un prédicat binaire.

```
> (filter odd? '(5 2 3 1 6))
(5 3 1)
> (filter (lambda (x) (> x 10)) '(5 12 10 8 14 19 3))
(12 14 19)
```

- # Ordre supérieur sur les listes : le schéma **foldr**

- Il réalise l'**abstraction du parcours récursif de liste.** Voici deux fonctions ***isomorphes*** :

```
(define (somme L)
  (if (empty? L)
      0
      (+ (first L) (somme (rest L)))))
```

```
(define (map g L)
  (if (empty? L)
      L
      (cons (g (first L)) (map g (rest L)))))
```

- On notera e le résultat pour la liste vide. Et on généralisera les fonctions + ou cons en une fonction binaire **op(x,r)** où x représente le premier élément de L et r le résultat de l'appel récursif sur le reste.

```
(define (foldr f e L)
  (if (empty? L)
      e
      (op (first L) (foldr f e (rest L)))))
```

- Alors:

```
(scarrés L) = (foldr + 0 L)
(map g L) = (foldr (λ (x r) (cons (f x) r)) empty L)
```

- Ordre supérieur sur les listes : la primitive **apply**
- La fonction d'addition + prend un nombre quelconque ≥ 2 d'arguments :

```
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 1 2 3 4 5 6 7 8 9 10)
55
```

- Mais supposons que les nombres à additionner soient dans une liste. Que faire ?

```
> (define L '(2 3 4))
> (+ L)
ERROR
```

- La solution passe par (apply f L) : ***appliquer*** une fonction f à une liste d'arguments L : (apply f (list a b c d)) → (f a b c d)

```
> (apply + L)    ; ⇔ (+ 2 3 4)
9
```

- Combiner les schémas d'ordre supérieur
- Exemple. Soit à calculer une approximation de la somme :

$$S = \frac{1}{1!} + \frac{1}{3!} + \frac{1}{5!} + \cdots + \frac{1}{15!}$$

```
; je commence par construire la liste de tous les entiers de 1 à 15
> (build-list 15 add1)
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
; puis je filtre les impairs
> (filter odd? (build-list 15 add1))
(1 3 5 7 9 11 13 15)
; ensuite j'applique la fonction k → 1/k! à tous les éléments de la liste
> (map (lambda (k) (/ 1 (fac k))) (filter odd? (build-list 15 add1)))
(1 1/6 1/120 1/5040 1/362880 1/39916800 1/6227020800 1/1307674368000)
; je somme le tout
> (define S (apply + (map (lambda (k) (/ 1 (fac k)))
                          (filter odd? (build-list 15 add1)))))
> S
1536780478171
─────────────
1307674368000
; je rend inexact au besoin
> (exact->inexact S)
#i1.1752011936437987
```

```
(define S
  (apply + (map (λ (k) (/ #i1 (fac k)))
                (filter odd? (build-list 15 add1)))))
```

Le style APL...

- Il existe beaucoup de schémas d'ordre supérieur, qui ne sont pas intégrés d'emblée à Racket...
- Exemple : (member x L) coupe la recherche dans la liste L, dès que l'élément x est trouvé.
- Souvent, on cherche un élément vérifiant une condition donnée par un prédicat pred?. **Généralisons** donc member en member-if :

```
(define (member-if pred? L)
  (cond ((empty? L) false)
        ((pred? (first L)) true)
        (else (member-if pred? (rest L)))))
```

```
> (member-if (lambda (n) (>= n 10)) '(8 4 9 15 7 12 3))
true
> (member-if (lambda (y) (= y 9)) '(8 4 9 15 7))
true
> (member-if symbol? '(8 4 9 15 7 12 3))
false
```