Université Nice Sophia Antipolis

Un langage d'expressions préfixées

PCPS, chap. 1

(PCPS)
Editions Ellipses, 2010

Tous les exercices sont corrigés sur ma page Web :

deptinfo.unice.fr/~roy

Une notation préfixée parenthésée

PCPS p. 16

- On écrit `(f x y z)` au lieu de `f(x,y,z)`.

$a + b + 2$	→	`(+ a b 2)`
$a + 3b + 5$	→	`(+ a (* 3 b) 5)`
$sin(\omega t+\varphi)$	→	`(sin (+ (* omega t) phi))`
$(x,y) \longmapsto x + sin\, y$	→	`(lambda (x y) (+ x (sin y)))`
si x>0 alors 3 sinon y+1	→	`(if (> x 0) 3 (+ y 1))`
$f \circ g$	→	`(compose f g)`

fonctions

- Avantage : **aucune ambiguïté**, facile à analyser.
- Inconvénient : il faut s'y habituer...

`(* 2 pi (sqrt (/ L g)))` ⟷ $2\pi\sqrt{\frac{L}{g}}$

Calculer avec des Fonctions

- Un **algorithme** est une méthode systématique de calcul d'une certaine quantité q à partir d'autres quantités a, b, c,...
- On dit que q s'exprime **en fonction de** a, b, c,...
- Exemple : calculer l'aire A d'un disque à partir de son rayon r.

$$A = \pi r^2$$

- Notre **langage de programmation Scheme** va nous permettre d'exprimer ce calcul par une **notation fonctionnelle** :

```
(define (aire r)
  (* pi r r))
```

$aire : \mathbb{R} \longrightarrow \mathbb{R}$

N.B. Racket, langage : *Etudiant niveau avancé (avec write + fractions mêlées)*

La Programmation Fonctionnelle

- Une fonction **reçoit** des valeurs, et **produit** une valeur.
- Le fait de produire une seule valeur n'est pas restrictif, puisque nous disposerons de **données structurées** [une *liste de valeurs* par exemple].

> Le but fondamental de ce cours est d'**apprendre à construire une valeur à partir d'autres valeurs**.

- Le mot important est CONSTRUIRE, et non MODIFIER ce qui est déjà construit !
- Donc jamais de phrases du style `x = x + 1`. Le signe = est réservé à la comparaison uniquement ! Aucune affectation, aucune mutation !

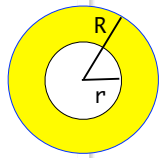
Qu'est-ce qu'un **calcul** ?

- Calculer, c'est programmer un certain nombre de fonctions, puis demander le résultat d'une expression utilisant ces fonctions.

```
(define (aire r)              ; aire d'un disque
  (* pi (sqr r)))

(define (aire-anneau r R)     ; aire d'un anneau
  (- (aire R) (aire r)))

(printf "Valeur de (aire 2) : ~a\n" (aire 2))
(printf "Valeur de (aire-anneau 1 2) : ~a\n" (aire-anneau 1 2))
```



Exécuter →

```
Valeur de (aire 2) : #i12.566370614359172
Valeur de (aire-anneau 1 2) : #i9.42477796076938
```

inexact...

- L'**ordre des fonctions** n'a pas d'importance. Mais un test utilisant une fonction doit bien entendu être placé *après* la définition de la fonction !

Entiers et Rationnels exacts

PCPS p. 34-36

- Les **entiers**, comme 56 ou -743 ou 786543455667925103457865432 1

entiers exacts — *précision "infinie"*

```
> (integer? 5)
#true
> (integer? 5.0)
#true
```

```
> (quotient 5 3)
1
> (modulo 5 3)
2
```

```
> (gcd 12 8)
4
> (lcm 12 8)
24
```

- Les **rationnels** [exacts], comme 5/3 ou -7/11. Attention, 5/3 est une ***notation*** et pas une opération ! Distinguer `(/ 5 3)` et 5/3.

```
> (rational? 5)
#true
> (rational? 5/3)
#true
```

```
> (+ 1/3 1/6)
1/2
> (* 2 3/2)
3
```

```
> (numerator 6/8)
3
> 6/8
3/4
```


Nombres inexacts (ou approchés)

PCPS § 1.6.3

- Le langage Scheme manipule des **nombres inexacts** comme la constante `pi = #i3.141592653589793` dont **la précision est limitée**.
- Dans notre niveau de langage ***(Etudiant Avancé)***, le nombre `1.25` est un nombre **exact** puisque `1.25 = 1 + 0.25 = 1 + 1/4 = 5/4`

```
> (exact? 1.25)
#true
> (exact? #i1.25)
#false
```

```
> (exact? pi)
#false
> pi
#i3.141592653589793
```

- Donc ne confondez pas `1.25` [*exact*] et `#i1.25` [*inexact*]...

```
> (real? 5)
#true
> (real? 4/3)
#true
> (real? pi)
#true
```

```
> (exact? 4/3)
#true
> (exact? pi)
#false
> (= 5.0 5)
#true
```

```
> (sqrt 2)
#i1.4142135623730951
> (sin (/ pi 2))
#i1.0
> (inexact->exact #i3.5)
7/2
```

Complexes : $\mathbb{C} = \mathbb{R} + \mathbb{R}i$

PCPS § 1.6.4

- Les **complexes** sont de la forme a+bi.
- Exemples : 5-2i [*exact*] ou #i5.2+4i [*inexact*].
- Attention, 5-2i est une ***notation*** et pas une opération ! Faites la différence entre (- 5 (* 2 +i)) et 5-2i. Tout seul, le nombre $\sqrt{-1}$ se note en effet +i et non i.

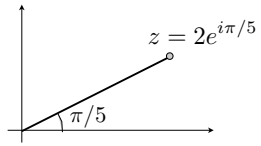
```
> (complex? 5)
#true
> (complex? 5-2i)
#true
```

```
> (+ 5-2i 3)
8-2i
> (* +i +i)
-1
```

```
> (real-part 5-2i)
5
> (angle 5-2i)
#i-0.3805063771123649
```

```
> (define z (* 2 (exp (* 1/5 +i pi))))
> z
#i1.618033988749895+1.1755705045849463i
> (magnitude z)
#i2.0000000000000004
```

calcul inexact !

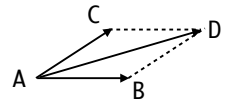


- Les **nombres complexes** sont un extraordinaire outil pour faire de la **géométrie plane** ! En effet un complexe z = x+iy peut être vu comme un point du plan M(x,y) ou bien comme un vecteur $\vec{v}$(x,y).
- Ex : centre de gravité G d'un triangle ABC dont les sommets ont pour affixes zA, zB et zC.

```
(define zG (/ (+ zA zB zC) 3))
```

- Ex : Etant donnés 3 points A, B, C, trouver D tel que ABCD soit un parallélogramme.

```
(define D (- (+ zB zC) zA))
```



- Ex : Un point M aléatoire sur le cercle de centre A et de rayon r.

```
(define zM (+ zC (make-polar 5 (* 2 pi (random))))) ; z_M = z_C + 5e^{iθ}
```

- Le produit scalaire de deux vecteurs $\vec{v_1}$ et $\vec{v_2}$.

```
(real-part (* z1 (conjugate z2)))   etc.
```


number?

complex?	⇐ *real?*	⇐ *rational?*	⇐ *integer?*
(+ z1 z2 ...)	(< x1 x2 ...)	(numerator r)	(quotient a b)
(- z1 z2)	(<= x1 x2 ...)	(denominator r)	(modulo a b)
(* z1 z2 ...)	(> x1 x2 ...)		(gcd a1 a2 ...)
(/ z1 z2)	(>= x1 x2 ...)		(lcm a1 a2 ...)
(= z1 z2 ...)	(abs x)		
(sqr z)	(max x1 x2 ...)		
(sqrt z)	(min x1 x2 ...)		
(expt z1 z2)	(floor x)		
(log z)	(round x)		
(exp z)			
(sin z) ...			
(asin z) ...			
(real-part z) ...			

Un langage dynamiquement typé

PCPS p. 34

- Soit f la fonction $x \mapsto 2x - 1$

```
(define (f x)          ; pour tout x, f(x) vaut
  (- (* 2 x) 1))       ; 2x-1
```

- Il est de la **responsabilité du programmeur** d'invoquer f en lui passant un nombre x, sinon gare ! En contrepartie, f est polymorphe !

```
> (f 5)
9
```

```
> (f 3/5)
1/5
```

```
> (f 1+2i)
1+4i
```

```
> (f #i3.4)
#i5.8
```

```
> (f "coucou")
ERROR : * expects type <number> as 2nd argument, given: "coucou"
```

(f "coucou") → (- (* 2 "coucou") 1)

Variables et Types

• Le langage de programmation manipule des **variables**, comme x dans la fonction f de la page précédente.

• Au moment du calcul de (f 5), la variable x prendra comme valeur l'entier 5, puis le calcul de l'expression (- (* 2 x) 1) fournira 9.

• Au moment du calcul de (f 5.3), la variable x prendra comme valeur le réel 5.3, puis le calcul de l'expression (- (* 2 x) 1) fournira 9.6.

CONCLUSION$_1$: les valeurs sont typées. 5 est un entier [integer], 4/3 est un rationel [rational]. Un type est un ensemble de valeurs.

CONCLUSION$_2$: les variables ne sont pas typées. La variable x peut prendre diverses valeurs de types différents. Mais au moment d'utiliser cette variable, sa valeur sera typée.

On dit que **les variables sont dynamiquement typées**

Quelques Types de base

• Les données sur lesquelles nous allons commencer à travailler peuvent avoir pour l'instant comme type :

- un type **numérique** [*nombre*] : integer?, rational?, real?, complex? ⇔ number?
- un type **chaîne de caractères** [*texte*] : string?

Exemple : `"Le résultat de (acos 3) est : ~a\n"`

- un type **valeur de vérité** [*vrai* ou *faux*] : boolean?

`true` ⇔ `#t` ⇔ `#true` `false` ⇔ `#f` ⇔ `#false`

• A chaque type primitif est en effet associée une **fonction à valeur booléenne** [*prédicat*] permettant de savoir si un objet est de ce type :

```
(real? pi)    → #t        (string? "pi") → #t
(integer? 3) → #t         (real? #t)     → #f
(real? 3)     → #t        (real? real?)  → #f
```


Comment tester une condition ?

PCPS § 1.7.1

```
(integer? pi) → #f        (boolean? #false) → #t
```

• Deux constantes #t [*vrai*] et #f [*faux*], notée aussi #true et #false dans notre dialecte Racket *Etudiant Avancé*.

• La conditionnelle de base est `(if <test> <sivrai> <sifaux>)`

```
> (if (inexact? pi) (+ 2 3) (* 2 3))
5
> (if (integer? pi) (+ 2 3) (* 2 3))
6
```

```
(define (valeur-absolue x)          ; real ⟶ real
  (if (>= x 0)
      x
      (- x)))
```

• Erreurs courantes : -x au lieu de (- x), et (- 2) au lieu de -2

• L'expression conditionnelle la plus générale est **cond** qui se lit "*envisageons tous les cas possibles* " :

```
(define (mention note)          ; real ⟶ string
  (cond ((>= note 16) "TB")
        ((>= note 14) "B")
        ((>= note 12) "AB")
        ((>= note 10) "P")
        (else "Echec!")))
```

PCPS § 1.7.2

• Un **cond** est équivalent à une suite de if emboîtés :

```
(define (mention note)          ; real ⟶ string
  (if (>= note 16)
      "TB"
      (if (>= note 14)
          "B"
          (if (>= note 12)
              "AB"
              (if (>= note 10) "P" "Echec!"))))))
```

moins élégant...

- Les conditionnelles **and** et **or** sont aussi des `if` déguisés :

```
(and t1 t2 t3) <==> (if (not t1)
                        #false
                        (if (not t2)
                            #false
                            t3))
```

Résultat : le premier qui est faux, ou bien le dernier.

```
(define (fraction-egyptienne? x)          ; rationnel de la forme 1/n ?
  (and (rational? x) (exact? x) (= (numerator x) 1)))
```

PCPS § 1.7.3

```
(or t1 t2 t3) <==> (if t1
                       #true
                       (if t2
                           #true
                           t3))
```

Résultat : le premier qui est vrai, ou bien le dernier.

```
(define (sur-les-axes? z)                 ; complexe sur les axes ?
  (or (real? z) (= (real-part z) 0)))
```


Eviter les recalculs : les définitions locales

- Comment **éviter de calculer plusieurs fois la même quantité** ?

REPONSE : Utiliser des **définitions locales à un calcul**.

```
> (somme-carrés (+ 2 3) (+ 2 3))          ; Bad...
50
> (local [(define v (+ 2 3))]             ; Good !
    (somme-carrés v v))
50
```

PCPS §2.6

- *Sachant que* `v` *vaut* `(+ 2 3)`, *calculer ce qui suit...*
- Forme générale :

```
(local [(define ...)
        (define ...)
        ...]
  expr)
```

*L'expression **expr** utilise les définitions locales.*

où chaque **définition locale** est **temporaire**, juste le temps de calculer l'expression *expr*. Les définitions sont évaluées *en séquence*.

Exemple : racine d'un trinôme du 2nd degré

- Proposons-nous de trouver une racine d'un trinôme $ax^2 + bx + c$, où l'on suppose que $a \neq 0$. Le trinôme est donné par la suite a,b,c de ses coefficients **réels**, la lettre x est muette... Calcul classique ($\Delta \geq 0$) :

$$ax^2+bx+c = a\left[x^2+\frac{b}{a}x+\frac{c}{a}\right] = a\left[\left(x+\frac{b}{2a}\right)^2-\left(\frac{\sqrt{\Delta}}{2a}\right)^2\right] \quad \text{avec} \quad \Delta = b^2-4ac$$
$$= a\,(x-x_1)\,(x-x_2) \quad \text{avec} \quad x_i = \frac{-b \pm \sqrt{\Delta}}{2a}$$

- Rendons l'une des deux racines :

```
(define (une-racine a b c)      ; une racine de ax² + bx + c, ou bien #f
  (if (= a 0)
      (error "Pas un trinome !")
      (local [(define Δ (- (sqr b) (* 4 a c)))]    ; le discriminant Δ
        (if (< d 0)
            #f
            (/ (- (sqrt Δ) b) (* 2 a))))))
```


Tester son programme *(check)*

PCPS p. 48

- Le test d'un programme occupe une grande partie du temps de programmation ! Il faut **tester tous les cas** [génériques] possibles !
- Le niveau ***Etudiant avancé*** offre trois primitives pour vérifier :

`check-expect` `check-within` `check-error`

- Quelle est la **spécification** de `x = (une-racine a b c)` ? Il faut que x soit une racine de $ax^2 + bx + c$, disons à 10^{-5} près dans les réels :

```
(define (racine? x a b c)       ; x est-elle racine de ax² + bx + c ?
  (< (abs (+ (* a x x) (* b x) c)) #i0.00001))
```

All tests passed!

```
(check-expect (une-racine 1 1 -6) 2)   ; exact
(check-within (une-racine 1 (sqrt 2) -1) #i0.5 #i0.1)    ; approché
(check-error (une-racine 0 1 2) "Pas un trinome !")      ; erreur prévue
(check-expect (racine? (une-racine 1 (sqrt 2) -1) 1 (sqrt 2) -1) #t)
```

Tester son programme *(show, printf)*

PCPS p. 49

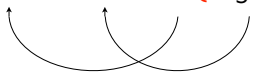
• Dans l'éditeur, la fonction (show expr) du teachpack valrose.rkt permet de faire l'écho de la demande de calcul de expr au toplevel.

```
(show (une-racine 3 -1 -2))
```

• Plus classique, on peut **afficher le résultat d'un calcul** au sein d'un message explicatif (printf <str> <expr> ...) qui affiche la chaîne de caractères <str>, pouvant contenir des **jokers** ~a. Les expressions <expr> ... fournissent les valeurs des jokers :

```
> (printf "Hello World !\n")
Hello World !
> (define n 2015)
> (printf "Le logarithme de ~a est ~a\n" n (log n))
Le logarithme de 2015 est 7.608374474380783
```

```
(printf "Le logarithme de ~a est ~a\n" n (log n))
```



N.B. On utilise \n *pour aller à la ligne.*