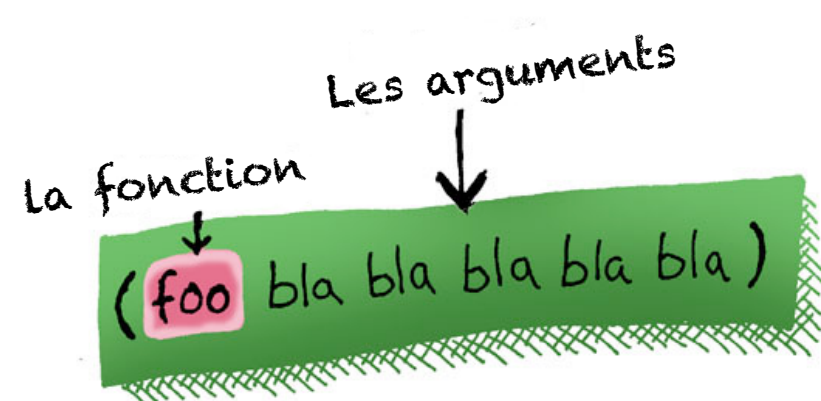


Programmer avec des Fonctions



PCPS, chap. 2 et §5.1

Arité d'une Fonction

- L'**arité** d'une fonction est le **nombre d'arguments** qu'elle attend :

```
(define (loto)
  (random 50))
```

Arité 0

```
(define (perimetre r)
  (* 2 pi r))
```

Arité 1
(unaire)

```
(define (somme-carres x y)
  (+ (* x x) (* y y)))
```

Arité 2
(binaire)

```
(define (distance x1 y1 x2 y2)
  (sqrt (+ (sqr (- x1 x2)) (sqr (- y1 y2)))))
```

Arité 4

La primitive +

```
(+ 2 3)
(+ 2 3 4)
(+ 2 3 4 5)
```

Arité
variable

Paramètres dans une définition de fonction

- Les **paramètres** sont des variables abstraites du texte de la fonction. En principe leur nom n'a pas d'importance : elles sont **muettes**.

```
(define (perimetre r)  
  (* 2 pi r))
```

```
(define (perimetre k)  
  (* 2 pi k))
```

- Pourquoi seulement *en principe* ?

```
(define (périmètre pi)  
  (* 2 pi pi))
```

?

- Idem en maths :

$$\int ax^2 dx = \int ay^2 dy = \int au^2 du$$

```

$$\int aa^2 da$$
 ?
```

Arguments dans un Appel de Fonction

- Il y a deux moments dans l'histoire d'une fonction :

1. le moment où on la **définit**

2. le moment où on l'**utilise**

où on l'*invoque*

où on l'*appelle*

où on l'*exécute*

1. Le moment où on la **définit**

paramètres [variables "muettes"]

```
(define (somme-carres x y)
  (+ (* x x) (* y y)))
```

*l'expression formant le
corps de la fonction*

Soit *somme-carres* la fonction définie par :

$$\text{somme-carres}(x,y) = x^2 + y^2 \quad \forall x \in \mathbb{C}, \forall y \in \mathbb{C}$$

2. Le moment où on l'utilise

- Les **arguments** sont des expressions qui vont être évaluées au moment de l'appel de la fonction, et qui deviendront les valeurs des paramètres.

*appel de
fonction*

arguments

```
(somme-carres (+ 2 3) (* 4 5))
```

*temporaires, juste le temps
de calculer la fonction...*

valeurs des arguments : 5 20

$x = 5$ $y = 20$

$(+ (* x x) (* y y))$

425

Un peu de style SVP...

- On ne choisit pas le **nom des variables** au hasard ! Pensez au lecteur...

```
(define (perimetre x)
  (* 2 pi x))
```

Bad

```
(define (perimetre r)
  (* 2 pi r))
```

Good

```
(define (perimetre rayon)
  (* 2 pi rayon))
```

Very good

- On **indente** le texte de la fonction [distance à la marge], en fonction du nombre de parenthèses ouvrantes non fermées. Les éditeurs de texte le font automatiquement. Sinon : *Ctrl-i* pour ré-indenter tout !
- On **documente** un minimum la fonction, avec des commentaires.

Documentez vos fonctions !

- Il est en effet de bon ton de faire précéder le texte d'une fonction non triviale de **commentaires** décrivant ses paramètres, et expliquant ce qu'elle calcule, avec les astuces algorithmiques si besoin.

```
; (fac n) retourne la factorielle n!  
; On suppose n entier ≥ 0  
  
(define (fac n)      ;  $\mathbb{N} \rightarrow \mathbb{N}$   
  (if (= n 0)  
      1  
      (* n (fac (- n 1)))))
```

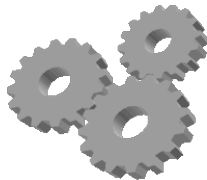
← *la spécification
en commentaire*

```
(printf "(fac 10) = ~a\n" (fac 10))  
(check-expect (fac 10) 3628800)
```

← *tests*

Comment ça marche ?...

PCPS § 1.8



- Soit la FONCTION suivante :

```
(define (somme-carrés x y) ; number × number → number  
  (+ (sqr x) (sqr y)))
```

- Calcul de l'expression (somme-carrés (+ 1 2) (* 2 3)). Que va-t-il se passer exactement ?...

ORDRE APPLICATIF D'EVALUATION

- ① Tous les éléments de l'expression sont évalués :
somme-carrés → #<procedure:somme-carrés>
(+ 1 2) → 3
(* 2 3) → 6
- ② Les variables paramètres sont "liées" aux valeurs obtenues :
x → 3, y → 6
- ③ Le corps de la fonction est évalué, et produit le résultat 45
- ④ Les liaisons temporaires effectuées en ② sont détruites.

☞ Ce n'est pas forcément *notre manière intuitive de calculer* !

• **Contre-exemple 1.** J'ai programmé une fonction (fac n) calculant n! et je demande le calcul de (* (fac 1000) 0). Je sais que le résultat est 0 mais Scheme va **calculer inutilement** (fac 1000).

Ce phénomène est identique pour (presque) tous les langages de prog...

➡ *Parade : tâcher d'être soi-même **intelligent**...*

• **Contre-exemple 2.** Supposons que je veuille calculer l'expression :

(somme-carrés (* 2 3) (* 2 3))

La stratégie énoncée à la page précédente implique l'**évaluation deux fois de la même expression** (* 2 3) ce que l'on ne ferait pas à la main.

Ce phénomène est identique pour tous les langages de prog...

➡ *Parade : ne faire le calcul qu'une seule fois (local).
Eviter les recalculs qui consomment du temps !*

Les Formes Spéciales

PCPS p. 40

- Soit à prouver que **if** n'est pas une fonction !

Preuve : si c'était le cas, (if p q r) commencerait par évaluer ses trois arguments p, q et r [page 18], ce qui n'est pas le cas. Seuls sont évalués p, q, ou bien p, r. CQFD

- Idem pour **define**, **cond**, **and**, **or**, **local**.

Ce sont des mots-clés [*keywords*] de formes spéciales.

Mécanique de l'évaluation d'une FORME SPECIALE

- Pas de règle uniforme : pour chacune, c'est **spécial** !!! Voir la doc.
- Par contre, la règle est la même pour tous les **appels de fonctions**.

Cette distinction est très importante !

Décomposer en plusieurs fonctions

- Résister à la tentation de tout écrire en une seule fonction !
- Exemple : calculer une racine de l'équation $ax^2+bx+c = 0$
- Un trinôme est donné par ses trois coefficients a, b, c

```
(define (une-racine a b c)      ; number × number × number → number
  (if (= a 0)
      (error "une-racine : pas un trinôme !")
      (local [(define delta (discriminant a b c))] ; sachant que  $\Delta = \dots$ 
              (/ (- (sqrt delta) b) (* 2 a)))))

(define (discriminant a b c)
  (- (sqr b) (* 4 a c)))
```

(une-racine 1 -1 -6)	→	3	$x^2 - x - 6$
(une-racine 1 -2 5)	→	1+2i	$x^2 - 2x + 5$
(une-racine 2 1 -1)	→	1/2	$2x^2 + x - 1$
(une-racine 1 (sqrt 2) -1)	→	#i0.5176380902050414	$x^2 + x\sqrt{2} - 1$

Quelques fonctions simples en maths...

- La primitive (`random n`), avec n entier > 0 , retourne un entier aléatoire de l'intervalle $[0, n-1]$. Par exemple $(\text{random } 3) \in \{0, 1, 2\}$.
- Exemple : un **dé truqué** ! Je souhaite tirer 0 ou 1, mais :
 - 0 : avec 1 chance sur 3
 - 1 : avec 2 chances sur 3
- Or `(random 2)` donne 0 ou 1 avec la même probabilité $1/2$
- J'utilise donc un dé à 3 faces marquées 0, 1, 2 [car 3 cas possibles].

```
(define (de-truque) ;  $\emptyset \rightarrow \mathbb{N}$ 
  (local [(define tirage (random 3))] ; dé à 3 faces : 0,1,2
    (if (= tirage 0)
        0 ; proba(0) = 1/3
        1))) ; proba(1) = 2/3
```

- Une **suite convergente** vers $\sqrt{2}$.

$$u_0 = 1, u_n = \frac{1}{2} \left(u_{n-1} + \frac{2}{u_{n-1}} \right) \qquad \lim_{n \rightarrow \infty} u_n = \sqrt{2}$$

```
(define (terme-suivant u)      ; real → real
  (/ (+ u (/ 2 u)) 2))
```

- Une version un peu plus décomposée :

```
(define (terme-suivant u)      ; real → real
  (moyenne u (/ 2 u)))

(define (moyenne a b)         ; real × real ⇒ real
  (/ (+ a b) 2))
```

Deux avantages :

- un peu plus **lisible**
- la fonction (moyenne a b) pourra être **ré-utilisée** ailleurs.

; Je prends comme premier terme a
; un réel quelconque > 0

```
> (define a #i54.0)
> (define u1 (terme-suivant a))
> (define u2 (terme-suivant u1))
> (define u3 (terme-suivant u2))
> (define u4 (terme-suivant u3))
> (define u5 (terme-suivant u4))
> (define u6 (terme-suivant u5))
> (define u7 (terme-suivant u6))
> (define u8 (terme-suivant u7))
```

• Et si je veux calculer plus loin ?

Il faut un moyen

d'exprimer une **répétition** !

On en reparlera...

```
> u1
#i27.01851851851852
> u2
#i13.546270911075572
> u3
#i6.846956509801686
> u4
#i3.569528547256061
> u5
#i2.064913314814075
> u6
#i1.5167384879448198
> u7
#i1.4176786819197515
> u8
#i1.4142177971315157
> etc.
```

On flaire la convergence !

```
> (sqrt 2)
1.4142135623730951
```

Les fonctions anonymes

- Les matheux savent parler d'une fonction sans lui donner de nom :

$$(x, y) \mapsto x^2 + y$$

- Nous aussi : `(lambda (x y) (+ (sqr x) y))`

- Autres exemples :

$x \mapsto x^3$	<code>(lambda (x) (* x x x))</code>
$(x, y, z) \mapsto x - y + z$	<code>(lambda (x y z) (+ x (- y) z))</code>
$\mapsto 3$	<code>(lambda () 3)</code>

- Ce sont des **fonctions anonymes**.

- On peut prendre la **valeur en un point** d'une fonction anonyme :

```
> ((lambda (x) (* x x x)) 2)
8
> ((lambda (x y z) (+ x (- y) z)) 3 4 9)
8
> ((lambda () 3))
3
```

- On peut toujours briser l'anonymat et **donner un nom** à la fonction :

```
(define cube
  (lambda (x)
    (* x x x)))
```



```
(define (cube x)
  (* x x x))
```

```
> (cube 2)
8
> (lambda (x) (* x x x))
#<procedure>
> cube
#<procedure:cube>
```


- Une fonction peut très bien
 - prendre une fonction en **argument**
 - retourner une fonction en **résultat**

Fonction \implies Nombre

La prise de valeur en 0

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

```
> (val0 cos)
```

```
1
```

```
> (val0 (lambda (x) (+ x 3)))
```

```
3
```

Fonction \times Fonction \implies Fonction

La composition de fonctions [la loi rond]

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

```
> (define cos^2 (compose sqr cos))
```

```
> (cos^2 pi)
```

```
#i1.0
```

$x \mapsto \cos^2 x$

Fonction \times Réel \implies Réel

La dérivation approchée $f'(x)$

```
(define (derivee f x)
  (/ (- (f (+ x #i0.001)) (f x)) #i0.001))
```

```
> (derivee log 2)
```

```
#i0.49987504165105445
```

QUIZZ

Quelle est la « signature » de la fonction `fun-const` ci-dessous?

```
(define (fun-const n)
  (lambda () n))
```

- a) nombre->fonction b) nombre->nombre->fonction

Réponse: a

Comment peut-on l'utiliser?

```
(fun-const 3)
```

a)

```
((fun-const) 3)
```

b)

```
((fun-const 3))
```

c)

Réponse: a et c

Des fonctions à plusieurs résultats ?

- Une fonction Scheme peut prendre plusieurs arguments mais ne retourne qu'**un seul résultat** :

```
(define (somme-carrés x y z) ; Num x Num x Num --> Num
  (+ (sqr x) (sqr y) (sqr z)))
```

- Et si je veux en retourner deux ? C'est impossible ?...
- Non : il suffit de regrouper les deux dans une **structure**.
- Un peu comme le matheux regroupe deux réels dans un vecteur de \mathbf{R}^2 .
- Au lieu de parler de x et y, on parle d'un point p du plan.
- Au lieu de parler de a, de b et de c, on parle du triplet (a,b,c).
- Problème : comment construire une **structure** ?

Un exemple de structure prédéfinie

- Racket fournit la structure `posn` de point du plan. On peut :
 - **fabriquer** un nouveau point avec `make-posn`
 - **accéder** aux coordonnées d'un point avec `posn-x` et `posn-y`
 - **tester** si une valeur quelconque est de type `posn` avec `posn?`

```
> (define a (make-posn 10 20))
> a
#(struct:posn 10 20)
> (posn? a)
#t
> (posn? 2013)
#f
> (posn-x a)
10
> (posn-y a)
20
```

- Il n'est **pas possible** en « programmation fonctionnelle pure » de faire **muter** les coordonnées de `a`. Mais on peut fabriquer un nouveau point :

```
> (define new-a
  (make-posn (posn-x a) (+ 1 (posn-y a))))
> new-a
#(struct:posn 10 21)
> a
#(struct:posn 10 20)
```

Définir son propre type de structure

- Pour définir les 4 fonctions associées à la structure `posn`, il a suffit à Racket d'évaluer la phrase :

```
(define-struct posn (x y))
```

*x et y sont les **champs***

- Je veux définir la structure de cercle. Un cercle est donné par son centre, son rayon et sa couleur :

```
(define-struct cercle (centre rayon couleur))
```

posn number string

```
> (define a (make-posn 10 20))  
> (define C (make-cercle a 8 "red"))  
> C  
#(struct:cercle #(struct:posn 10 20) 8 "red")  
> (cercle? C)  
#t
```

```
> (cercle-centre C)  
#(struct:posn 10 20)  
> (cercle-rayon C)  
8  
> (cercle-couleur C)  
"red"
```

Des fonctions sur des structures...

- La structure de cercle étant définie, je peux programmer des fonctions qui calculeront sur des structures :

```
(define-struct cercle (centre rayon couleur))
(define-struct rect (x y larg haut))
; Convenons qu'une "figure" est un point, un cercle ou un rectangle, ok ?
(define a (make-posn 10 20))           ; une figure
(define C (make-cercle a 8 "red"))     ; une seconde figure
(define R (make-rect 35 10 20 30))    ; une troisième figure

(define (longueur fig)                 ; figure → number
  (cond ((posn? fig) 0)
        (cercle? fig) (* 2 pi (cercle-rayon fig)))
        ((rect? fig) (* 2 (+ (rect-larg fig) (rect-haut fig))))
        (else (error "longueur : on attendait une figure : " fig))))
```

```
> (longueur a)
0
```

```
> (longueur C)
50.26548245743669
```

```
> (longueur R)
100
```

Le lambda-calcul, c'est quoi au juste ?

• Il s'agit d'une **théorie logique** qui propose de fonder les mathématiques non sur le concept d'**ensemble** (Cantor, Bourbaki) mais sur celui de **fonction** (Church). Dans les années 1930-40. Le λ -calcul manipule des expressions pouvant prendre l'une des trois formes suivantes :

- une **variable**

x, y, \dots

x, y, \dots

- une **abstraction**
(ou λ -expression, ou fonction)

$\lambda x. e$

(lambda (x) e)

- une **application** (d'une fonction u
à un argument v)

uv

(u v)

• Ce petit langage (de programmation) a une **puissance universelle** (il peut calculer tout ce qui est *calculable*). Il appartient aux **fondements** de la **théorie de la programmation** en permettant de modéliser tout autre langage de programmation. Secteur théorique de l'Informatique.

<http://www.lsv.ens-cachan.fr/~goubault/Lambda/lambda.pdf>