Programmation fonctionnelle

Dr. Mounir El Araki Tantaoui

Avec l’aimable autorisation du Professeur Jean Paul Roy

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

Programmation fonctionnelle

- 11 séances de 2 heures
- 1 contrôle continu (30%)
- Des travaux à faire (20 %)
- Examen final (50 %)
- Pré-requis : Aucun

Agenda

- Langage d’expressions préfixées
- Les fonctions
- Programmer avec des images / Animations
- Programmer par récurrence
- Les listes (chainées)
- Les calculs itératifs
  - Compléments sur la récurrence
- Type abstraits et généralisation
- Les arbres binaires
  - Parcours récursif
  - Parcours itératif et calcul formel

Paradigme des langages fonctionnels

- Le principe général de la **programmation fonctionnelle** est de concevoir des programmes comme des fonctions mathématiques que l'on compose entre elles.
- A la différence des **programmes impératifs** organisés en instructions produisent des effets de bords.
- Les programmes fonctionnels sont bâtis sur des expressions dont la valeur est le résultat du programme.
- En particulier dans un langage fonctionnel, il n'existe pas d'effet de bord.

Exemple d’un effet de bord

- ```
  #include <iostream>
  using namespace std;
  int a;
  void f() {
      a = 2;
  }

  int main () {
      a = 1;
      cout << a << endl;
      f();
      cout << a << endl;
      return 0;
  }
  ```
  ```

Programme Fonctionnel

- Un **programme fonctionnel** consiste en une expression **E** (représentant l'algorithme et les entrées).
- L’expression E est sujette à des règles de réécriture : **la réduction consiste** en un remplacement d'une partie de programme fonctionnel par une autre partie de programme selon une règle de réécriture bien définie.
- Ce processus de réduction sera répété jusqu'a l'obtention d'une expression irréductible *(aucune partie ne peut être réécrite)*.
- L'expression **E*** ainsi obtenue est appelée forme normale (FN) de E et constitue la sortie du programme.

Exemple

- SQR 3+2 représente l'application de la fonction calculant le carré d'un nombre à l'expression 3+2

- SQR 3+2 -> SQR 5
- -> 5*5
- -> 25

- SQR 3+2 -> (3+2)*(3+2)
- -> 5*(3+2)
- -> 5*5
- -> 25

Langages Fonctionnels

- Certains langages fonctionnels sont purs :
  - LISP, ML, MIRANDA.
  - D’autres comme les différents dialectes de LISP (SCHEME, …) contiennent des constructions impératives (sauts, effets de bord, …).
- Le **λ-calcul** est la base théorique commune à tous les langages fonctionnels
  - introduit par Church vers 1930.
- Le **λ-calcul** est basé sur trois concepts :
  - les variables,
  - l'abstraction fonctionnelle, permettant de construire des fonctions et
  - l'application de fonctions.
- Il est possible d'exprimer des fonctions d'ordre supérieur dont le résultat lui même est une fonction, et par conséquent applicable à d'autres fonctions.

λ-calcul

- Le λ-calcul est le paradigme de programmation fonctionnelle.
  - Souvent utilisé comme langage élémentaire de très bas niveau et permet de mettre en évidence des problèmes des langages de programmation sous forme très simple.
  - Aussi puissant que les autres formalismes du calcul :
    - machine de Turing,
    - fonctions récursives,
    - primitives, et fonctions récursives générales.
  - Le λ-calcul traite le calcul à partir du concept de fonctions. C'est l'aspect règle de **calcul de la fonction** (plus informatique) qui intervient ici et non l'approche **graphe**, c'est à dire ensemble de couples argument-valeur (plus mathématique).
  - Une règle de calcul est l'expression d'un processus décrivent le passage de l'argument à la valeur.

Lambda Notation

- On doit introduire des formes pour exprimer la notion de fonction et la notion complémentaire d'application

1.Alphabet

  - Alphabet= {atomes} ∪ {λ} ∪ {délimiteurs}
  - W= {atomes} ;
  - W = V ∪ C ;
  - V = {variables} ; C = {constantes} ; délimiteurs = {), (}

2.Langage : défini par la grammaire :

- {{S}, alphabet, P, S} avec P :
- i. S -> v, v ∈W
- ii. S -> (SS) application
- iii. S -> (λx. S) abstraction

- L(S) = {λ-expression}={λ-termes}

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

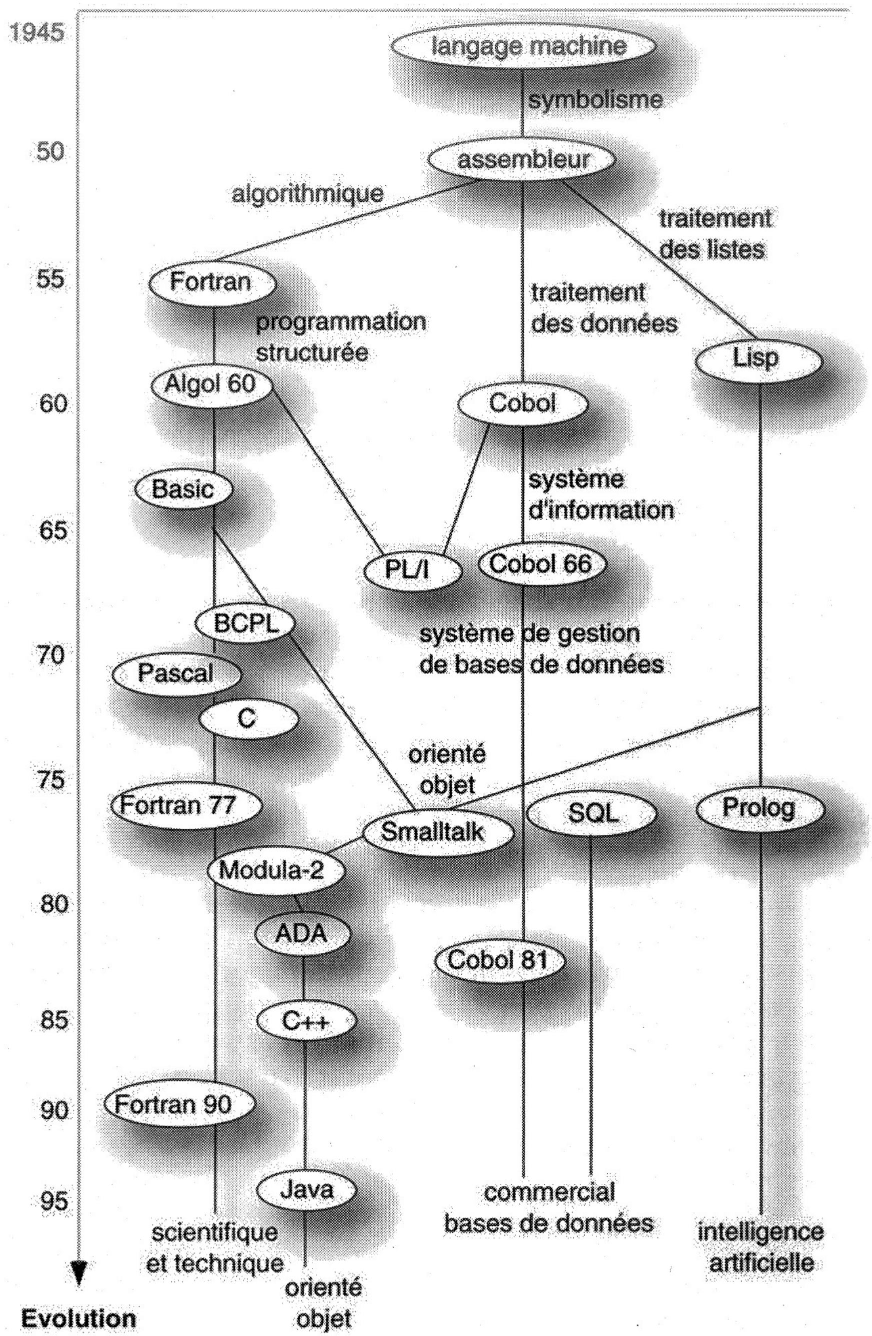
Racket

- Le langage Racket
- La programmation fonctionnelle
- http://racket-lang.org/download/
- http://racket-lang.org/
- http://docs.racket-lang.org/
- http://docs.racket-lang.org/quick/index.html

Quick: An Introduction to Racket with Pictures

by Matthew Flatt

Evolution des langages de programmation



Une notation préfixée parenthésée

- 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(ωt+φ)*** → (sin (+ (* omega t) phi))
  - ***(x,y) → x + sin y*** → (lambda (x y) (+ x (sin y)))
  - ***si x>0 alors 3 sinon y+1*** → (if (> x 0) 3 (+ y 1))
  - ***f o g*** → (compose f g)

- Avantage : **aucune ambigüité**, facile à analyser.
- Inconvénient : il faut s'y ha ...
  - (* 2 pi (sqrt (/ L g))) ↔ $2\pi\sqrt{\frac{L}{g}}$

Les expressions préfixées

- Notation infixe :
  - 2 + 3x/4-5 → (2+ ((3x)/4))-5 priorités des opérateurs
- Notation préfixe :
  - f(x, g(x))
- Notation postfixe : n!
- Les 3 notations : f(x+1, n!)

- Lisp, Scheme, DrRacket → Notation préfixe;
  - 2 + 3 * x; (+ 2 (* 3 x)) → Arbre
  - Données comme des arbres → Langages naturels, bases de connaissances, plan d'action, expressions algébriques, documents XML, compilateurs, etc.
  - + * x log y + z 1 → (+ (* x (log y)) (+ z 1))

Les expressions préfixées

Maths	Lisp/Scheme
f(x,y,z)	(f x y z)
f(x+1, y)	(f (+ x 1) y)
x + f(y)	(+ x (f y))
p et q	(and p q)
si x alor x+1 sinon y	(if (> x 0) (+ x 1) y)

Le Top Level

- Bienvenue dans DrRacket, version 5.3.6 [3m].
- Langage: Etudiant niveau avancé; memory limit: 128 MB.
- > (+ 2 3 4)
- 9
- > (+ (* 2 3)
-     (* 3 4)
-     (* 4 5)) ; ceci est un commentaire (ignoré)
- 38
- > (+2 3 4)
- function call: expected a function after the open parenthesis, but received 2

- La boucle TopLevel est donc un processus qui consiste à :

1. Lire une expression SCHEME grammaticalement correcte (construction d'un objet interne A)
2. Evaluer l'objet A construit pour produire un objet B
3. Afficher une représentation externe de l'objet B sous la forme d'une suite de caractère
4. Retourner au point 1 …

Le Top Level (Dictionnaire globale)

- > pi
- #i3.141592653589793
- > +
- +
- > foo
- foo: this variable is not defined
- > (define degre (/ pi 180))
- > (* 30 degre)
- #i0.5235987755982988
- > log
- log
- > ln
- ln: this variable is not defined

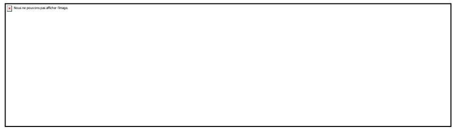
- Define est plus une définition de constante

Calculer avec des Fonctions

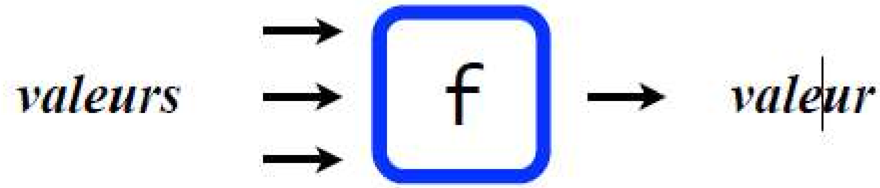
- Un **algorithme** est une méthode systématique de calcul d'une certaine quantité q à partir d'autres quantités a, b, c,...
- Exemple : calculer l'aire A d'un disque à partir de son rayon r.
- On dit que q s'exprime **en fonction de** a, b, c,...

- 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)
        )
```



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 !

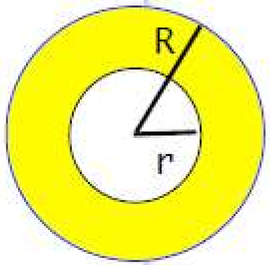
Qu'est-ce qu'un **calcul** ?

- 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

- Les **entiers**, comme 56 ou -743 ou 7865434556679251034578654321

  Entiers exacts

- 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.

```
> (integer? 7)
true
> (integer? 7.0)
true
> (quotient 5 3)
1
> (modulo 5 3)
2
> (gcd 12 8)
4
> (lcm 12 8)
24
>
```

```
> (rational? 5)
true
> (rational? 5/3)
true
> (+ 1/3 1/6)
0.5
> (* 2 3/2)
3
> (numerator 6/8)
3
> (denominator 6/8)
4
> 6/8
0.75
```

Nombres inexacts (ou approchés)

- 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
- Donc on ne confondra pas 1.25 [*exact*] et #i1.25 [*inexact*]...

```
> (exact? 1.25)
true
> (exact? #i1.25)
false
> (exact? pi)
false
> pi
#i3.141592653589793
```

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

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

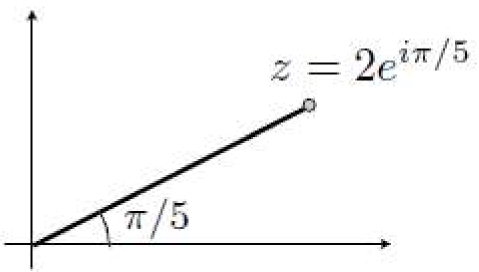
```
> (sqrt 2)
#i1.4142135623730951
> (sin (/ pi 2))
#i1.0
> (inexact->exact #i3.5)
3.5
```

Complexes : C = R + Ri

- 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
```



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) ...			
(real-part z)			

Un langage dynamiquement typé

- Soit f la fonction x ⇥ 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.** En contrepartie, f est polymorphe !

```
> (f 5)
9
> (f 3/5)
0.2
> (f 1+2i)
1+4i
> (f #i3.4)
#i5.8
```

```
> (f "coucou")
*: expects a 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.

- **CONCLUSION1 : les valeurs sont typées. 5 est un entier [integer], 4/3 est un rationel [rational]. Un type est un ensemble de valeurs.**
- **CONCLUSION2 : 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**
  - un type **chaîne de caractères** [*texte*] : **string**
    - *Exemple : "Le résultat de (acos 3) est : »*
  - un type **valeur de vérité** [*true* ou *false*] : **boolean**

- A chaque type est associée une **fonction à valeur booléenne** [***prédicat***] permettant de savoir si un objet est d'un type donné :

```
> (real? 3)
true
> (real? pi)
true
> (real? true)
false
> (real? real?)
false
```

```
> (string? "pi")
true
> (string? pi)
false
```

Comment tester une condition ?

- Deux constantes #t [*vrai*] et #f [*faux*], notée aussi true et false.
- 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
- Formes spéciales : define, lambda, local, if, cond, and, or, case, begin

Le conditionnel

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

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

Real → String

- 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...

And/ Or

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

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

- (or t1 t2 t3) <==> (if (t1) #t
  (if (t2) #t
  t3))

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

L'évaluation d'une expression arithmétique

- Supposons une expression (f a b …) dont nous cherchons une valeur V au TopLevel.
  - Si f est le mot-clé d'une forme spéciale, on procède à un traitement spéciale!
  - Sinon on **évalue** tous les éléments de la forme parenthèsées et on obtient resp. les valeurs F A B ….
    - Si F n'est pas une procédure, erreur et revenir au TopLevel
    - Sinon on **applique** la procédure F aux valeurs trouvées A B …et obtenir la valeur V

- Dans le cas où on applique la fonction F sur tout les éléments évalués → Appel par valeur (la valeur est transmise à la fonction)
  - L'ordre d'évaluation est de l'intérieur vers l'extérieur
  - Exemple f(x,y) = x
  - $f(1+1, 10^{20}) = f(2, 100000000000000000000) = 2$ (appel par valeur)
  - $f(1+1,10^{20}) = 1+1= 2$ (Evaluation paresseuse → retarder l'évaluation des arguments)

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))
50
> (local [(define v (+ 2 3))]
   (somme-carrés v v))
50
```

- 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 ax2 + bx + c, où l'on suppose que a ≠ 0. Le trinôme est donné par la suite a,b,c de ses coefficients **réels**, la lettre x est muette… Calcul classique (Δ ≥ 0) :

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

$$= a\,(x-x_1)\,(x-x_2) \quad \text{avec} \quad x_i = \frac{-b\pm\sqrt{\Delta}}{2a}$$

- Prenons la racine la plus petite :

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

Tester son programme *(check)*

- 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 ax2 + 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)) 0.00001))

All tests passed!

```
(check-expect (une-racine 1 1 -6) -3)   ; exact
(check-within (une-racine 1 (sqrt 2) -1) #i-1.9 #i0.05)  ; 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) true)
```

Tester son programme ***(show, printf)***

- 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

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

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 :

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

$$\cancel{\int aa^2da}$$

?

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**
     - **On l'invoque**
     - **On l'appelle**
     - **On l'exécute**

- 1. Le moment où on la **définit**

paramètres [variables muettes]

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

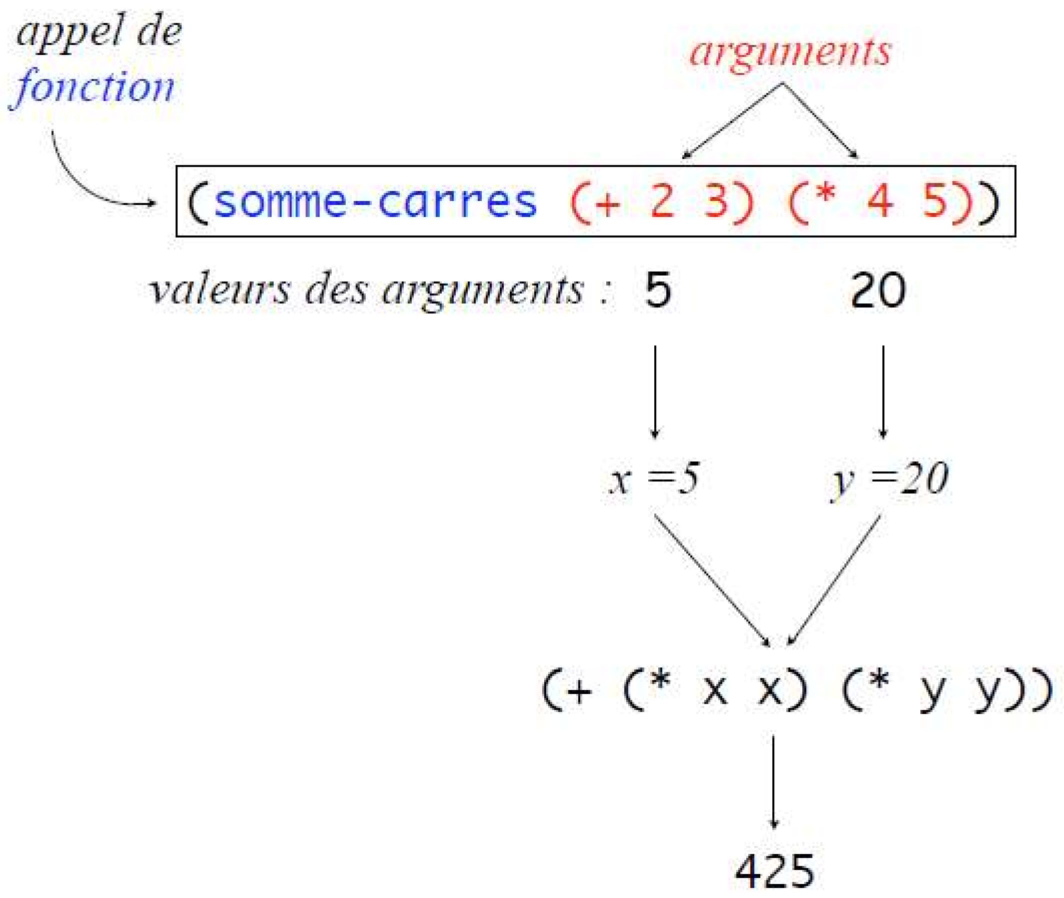
l'expression formant le ***corps*** *de la fonction*

Soit somme-carres *la fonction définie par :*

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.



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)          ; ℕ → 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 ?...

- 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 ?...

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

- 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 ce qui n'est pas le cas. Seuls sont évalués p, q, ou bien p, r.

- 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 trinome !")
      (local [(define delta (discriminant a b c))]   ; sachant que Δ = ...
        (/ (- (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 (random 3) ∈ {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)        ; ∅ → 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 $\sqrt{2}$

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

```
(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 : Prog2.1

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

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 :

$x \mapsto x^3$	`(lambda (x) (* x x x))`
$(x, y, z) \mapsto x - y + z$	`(lambda (x y z) (+ x (- y) z))`
$\mapsto 3$	`(lambda () 3)`

- 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**

Prog2.2

Fonction $\longrightarrow$ *Nombre*

La prise de valeur en 0

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

```
> (val0 cos)
1
> (val0 (lambda (x) (+ x 3)))
3
```

Fonction $\times$ *Fonction* $\longrightarrow$ *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* $\longrightarrow$ *Réel*

La dérivation approchée f'(x)

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

```
> (derivee log 2)
#i0.49987504165105445
```

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 **R2**.
- 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))
;(posn? a)
;(posn? 2013)
;(posn-x a)
;(posn-y a)
```

Prog2.3

- Il n'est pas possible en **programmation fonctionnelle** 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
;a
```

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))

- 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*

Modélisation d'un nombre rationnel

```
• (define-struct rat (n d))

• (define (rationnel p q)   ; retourne le rationnel de p/q simplifié
•   (cond ((= q 0) (error 'rationnel "Dénominateur nul!"))
•         ((< q 0) (rationnel (- p) (- q)))  ; on monte le signe
•         (else (local [(define g (gcd p q))]
•                 (make-rat (quotient p g) (quotient q g))
•                 )
•               )
•         )
•   )
```

Prog2.4

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