

11X001 – Introduction à la programmation des algorithmes

2.3a Fonctions et Branchements

François Fleuret

<https://fleuret.org/francois>

28 Septembre, 2020

*Le contenu de ce document a été en grande partie
repris du cours de Jean-Luc Falcone.*



**UNIVERSITÉ
DE GENÈVE**

FACULTY OF SCIENCE

Fonctions

Une fonction mathématique est une **relation** entre deux **ensembles**.

Pour chaque élément de l'ensemble de départ correspond **un seul** élément de l'ensemble d'arrivée.

Soit f une fonction d'un ensemble \mathcal{X} vers un ensemble \mathcal{Y} , on peut la définir entièrement par un ensemble $\mathcal{G} \subset \mathcal{X} \times \mathcal{Y}$ de paires (x, y) tel que:

- $x \in \mathcal{X}$ et $y \in \mathcal{Y}$.
- Ces paires ont toutes un premier élément différent, e.g.

$$\forall x \in \mathcal{X}, \exists ! y \in \mathcal{Y}, (x, y) \in \mathcal{G}.$$

Exemples

$$\text{id} : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \mapsto x$$

$$E.g. \text{id}(42) = 42$$

Exemples

$$\text{id} : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \mapsto x$$

$$E.g. \text{id}(42) = 42$$

$$\sqrt{\cdot} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$$

$$x \mapsto y \text{ t.q. } y^2 = x$$

$$E.g. \sqrt{2} = 1.41\dots$$

Exemples

$$\text{id} : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \mapsto x$$

$$\text{E.g. id}(42) = 42$$

$$m : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$$

$$(x, y) \mapsto \frac{x+y}{2}$$

$$\text{E.g. } m(5, 12) = \frac{17}{2}$$

$$\sqrt{\cdot} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$$

$$x \mapsto y \text{ t.q. } y^2 = x$$

$$\text{E.g. } \sqrt{2} = 1.41\dots$$

Exemples

$$\text{id} : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \mapsto x$$

$$E.g. \text{id}(42) = 42$$

$$\sqrt{\cdot} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$$

$$x \mapsto y \text{ t.q. } y^2 = x$$

$$E.g. \sqrt{2} = 1.41\dots$$

$$m : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$$

$$(x, y) \mapsto \frac{x+y}{2}$$

$$E.g. m(5, 12) = \frac{17}{2}$$

$$n : \mathbb{Z} \rightarrow \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$$

$$i \mapsto (i-1, i, i+1)$$

$$E.g. n(8) = (7, 8, 9).$$

Quelles expressions ci-dessous sont des fonctions ? Pour celles qui le sont, quels sont leurs ensembles de départ et d'arrivée?

- La couleur des pixels d'une image donnée ?
- La traduction d'un mot français en anglais ?
- L'altitude d'un point sur terre identifié par des coordonnées géographiques ?
- L'âge en années auquel une personne mesure une taille donnée ?
- L'heure donnée par une pendule ?

Dans la programmation en général, une fonction est juste un **sous-programme**.

Dans la programmation en général, une fonction est juste un **sous-programme**.

En **programmation fonctionnelle** le concept de fonction est très similaire à ce même concept en mathématiques.

On parle alors de **fonctions pures**.

Une fonction pure:

- est un **sous-programme** se comportant comme une **fonction mathématique**,
- accepte des **arguments** (entrée) et **retourne** une valeur (sortie),
- retourne **toujours** la **même valeur** pour des arguments donnés.

Une fonction pure:

- est un **sous-programme** se comportant comme une **fonction mathématique**,
- accepte des **arguments** (entrée) et **retourne** une valeur (sortie),
- retourne **toujours** la **même valeur** pour des arguments donnés.

Une fonction pure ne peut pas:

- modifier ses arguments,
- modifier l'état du programme,
- accéder à d'autres valeurs que ses arguments,
- avoir des "effets de bords".

Exemples de fonctions pures:

- Additionner deux nombres.
- Calculer la moyenne d'un tableau de nombres.
- Calculer la luminosité d'une couleur.
- Créer une version en niveau de gris d'une image couleur.

Exemples de fonctions pures:

- Additionner deux nombres.
- Calculer la moyenne d'un tableau de nombres.
- Calculer la luminosité d'une couleur.
- Créer une version en niveau de gris d'une image couleur.

Exemples de fonctions "impures":

- Lire un fichier.
- Rendre une image floue en la modifiant.
- Compléter une base de donnée.
- Uploader un fichier sur le web.

Définition de fonctions en Swift

```
func add( a: Int, b: Int ) -> Int {  
    return a + b  
}
```

Arguments

```
func add( a: Int, b: Int ) -> Int {  
    return a + b  
}
```


Définition de fonctions en Swift

```
func add( Arguments a: Int, b: Int ) -> Résultat Int {  
    return a + b  
}
```

Définition de fonctions en Swift

Signature Arguments Résultat

```
func add( a: Int, b: Int ) -> Int {  
    return a + b  
}
```

Définition de fonctions en Swift

Signature Arguments Résultat

```
func add( a: Int, b: Int ) -> Int {  
    return a + b  
}
```

Corps

Utilisation de fonctions en Swift

```
func add( a: Int, b: Int ) -> Int {  
    return a + b  
}
```

```
let q = add( a: 2, b: 12 )  
print(q) // Affiche 14
```

- Une fonction peut accepter zéro, un ou plusieurs arguments.
- Le type de chaque argument doit être spécifié.
- Le type de retour doit être spécifié.
- Le nom d'appel de chaque argument doit être spécifié.

Exemples

```
func add( a: Int, b: Int ) -> Int
```

```
func favoriteColor( u: User ) -> Color
```

```
func dayOfWeek( d: Date ) -> Day
```

Fonctions pures sans argument

Soient les limitations d'une fonction pure vues ci-dessus, que signifie une fonction sans argument ?

```
func truc() -> Int {  
  // ???  
}
```

Chaque fonction a un type, formé du type de ses arguments et du type de la valeur retournée. Par exemple:

```
func add( a: Int, b: Int ) -> Int {  
  return a + b  
}
```

a le type `(Int, Int) -> Int`.

Chaque fonction a un type, formé du type de ses arguments et du type de la valeur retournée. Par exemple:

```
func add( a: Int, b: Int ) -> Int {  
    return a + b  
}
```

a le type `(Int, Int) -> Int`.

```
func truc() -> Int {  
    // ???  
}
```

a le type `()->Int`.

Chaque fonction a un type, formé du type de ses arguments et du type de la valeur retournée. Par exemple:

```
func add( a: Int, b: Int ) -> Int {  
    return a + b  
}
```

a le type `(Int, Int) -> Int`.

```
func truc() -> Int {  
    // ???  
}
```

a le type `()->Int`.

```
func favoriteColor( u: User ) -> Color {  
    ...  
}
```

a le type `(User)->Color`.

La déclaration `return` termine la fonction et indique la valeur à retourner.

Toute code après cette déclaration est illégal.

```
func add( x: Int, y: Int ) -> Int {  
  print("Avant l'addition")  
  return x + y  
  print("Après l'addition") // Erreur de compilation  
}
```

Swift vérifie statiquement que le typage est correct, e.g.

```
1> func pasTerrible(a: Int) -> Double {  
2.     return a  
3. }  
error: repl.swift:2:12: error: cannot convert return expression  
of type 'Int' to return type 'Double'
```

Exemple: transformer une couleur en niveaux de gris

```
struct Color {
  let red: UInt8
  let green: UInt8
  let blue: UInt8
}

func grayLevels( color: Color ) -> Color {
  let avg = ( UInt16(color.red) +
             UInt16(color.green) +
             UInt16(color.blue) ) / 3
  let level = UInt8(avg)
  return Color( red: level, green: level, blue: level )
}
```

Exemple: transformer une couleur en niveaux de gris

```
struct Color {
  let red: UInt8
  let green: UInt8
  let blue: UInt8
}

func grayLevels( color: Color ) -> Color {
  let avg = ( UInt16(color.red) +
             UInt16(color.green) +
             UInt16(color.blue) ) / 3
  let level = UInt8(avg)
  return Color( red: level, green: level, blue: level )
}

let orange = Color( red: 255, green: 127, blue: 0 )
let gray = grayLevels(color: orange)
```

Exemple: calcul de prix

```
struct Item {  
  let description: String  
  let amount: UInt  
  let price: UInt  
}  
  
func total( item: Item, tax: Double ) -> UInt {  
  let beforeTax = item.amount * item.price  
  let afterTax = (1 + tax) * Double(beforeTax)  
  return UInt(afterTax)  
}
```

Exemple: calcul de prix (2)

```
let apples = Item(description: "Pommes", amount: 12, price: 75)
let t = total( item: apples, tax: 0.045 )
print(t, "cts") // Affiche 940 centimes
```

Portée lexicale (*scope*)

Exercice: quel est l'affichage ?

```
let a = 100
let b = 20
let c = 3

func truc( x: Int ) -> Int {
  let b = -x
  return b * c
}

let w = truc(x: a)

print(a)
print(b)
print(w)
print(x)
```

La portée lexicale d'un identifiant (fonctions, constante, variable) est la portion du programme dans laquelle cet identifiant est défini

- Les déclarations au niveau supérieur ont une portée **globale**.
- Les déclarations entre accolades ont une portée **locale**.
- En cas de redéfinition, la portée la plus locale l'emporte.
- Les arguments d'une fonctions ont une portée **locale** à celle-ci.

Exemple

```
let a = 100 // Global
let b = 20  // Global
let c = 3   // Global

func truc( x: Int ) -> Int { // truc globale, mais x local
  let b = -x // b local, cache b global
  return b * c
}

let w = truc(x: a) // w global
```

On peut définir des fonctions dans une fonctions, celles-ci sont locales à cette dernière.

Par exemple:

```
func f( x: Double, y: Double ) -> Double {  
  func sq( a: Double ) -> Double {  
    return a * a  
  }  
  return sq(a: x) + sq(a: y)  
}
```

Fonctions imbriquées (2)

Quel sera l'affichage ?

```
func f( x: Double, y: Double ) -> Double {  
  func sq( a: Double ) -> Double {  
    print("SQ:", a)  
    return a * a  
  }  
  print("F:", x, y)  
  return sq(a: x) + sq(a: y)  
}  
  
print( "RES:", f(x: 1, y: 2 ) )
```

Fonctions imbriquées (2)

Quel sera l'affichage ?

```
func f( x: Double, y: Double ) -> Double {  
  func sq( a: Double ) -> Double {  
    print("SQ:", a)  
    return a * a  
  }  
  print("F:", x, y)  
  return sq(a: x) + sq(a: y)  
}  
  
print( "RES:", f(x: 1, y: 2 ) )
```

```
F: 1.0 2.0  
SQ: 1.0  
SQ: 2.0  
RES: 5.0
```

Fonctions imbriquées (3)

Quel sera l'affichage ?

```
func f( x: Double, y: Double ) -> Double {  
  func sq( a: Double ) -> Double {  
    return x * x  
  }  
  return sq(a: x) + sq(a: y)  
}  
  
print( "RES:", f( x: 10, y: 5 ) )
```

Fonctions imbriquées (3)

Quel sera l'affichage ?

```
func f( x: Double, y: Double ) -> Double {  
  func sq( a: Double ) -> Double {  
    return x * x  
  }  
  return sq(a: x) + sq(a: y)  
}  
  
print( "RES:", f( x: 10, y: 5 ) )
```

Attention: on a utilisé `x` au lieu de `a` dans `sq` !

RES: 200.0

Fonctions imbriquées (4)

Quel sera l'affichage ?

```
func f( x: Double, y: Double ) -> Double {  
  func sq( a: Double ) -> Double {  
    return x * x  
  }  
  return sq(a: x) + sq(a: y)  
}  
  
print( "RES:", sq( a: 10 ) )
```

Fonctions imbriquées (4)

Quel sera l'affichage ?

```
func f( x: Double, y: Double ) -> Double {  
    func sq( a: Double ) -> Double {  
        return x * x  
    }  
    return sq(a: x) + sq(a: y)  
}  
  
print( "RES:", sq( a: 10 ) )
```

Erreur: la fonction `sq` n'existe pas en dehors de `f`.

```
error: repl.swift:23:16: error: use of unresolved identifier 'sq'  
print( "RES:", sq( a: 10 ) )
```

On peut définir des noms d'appel différents des noms des argument dans la fonction:

```
func grayLevels( color c: Color ) -> Color {  
  let avg = ( c.red + c.green + c.blue ) /3  
  return Color( red: avg, green: avg, blue: avg )  
}  
  
let orange = Color( red: 255, green: 127, blue: 0 )  
let gray = grayLevels(color: orange)
```

On peut supprimer le nom d'appel en utilisant un underscore:

```
func grayLevels( _ c: Color ) -> Color {  
    let avg = ( c.red + c.green + c.blue ) /3  
    return Color( red: avg, green: avg, blue: avg )  
}  
  
let orange = Color( red: 255, green: 127, blue: 0 )  
let gray = grayLevels(orange)
```

Sans nom d'appel (2)

```
func f( _ x: Double, _ y: Double ) -> Double {  
  func sq( _ x: Double ) -> Double {  
    return x * x  
  }  
  return sq(x) + sq(y)  
}
```

```
f(12, -13)
```

Distance entre deux points

```
struct Point {
  let x, y: Double
}

func distance(between p: Point, and q: Point) -> Double{
  func sq( _ x: Double ) -> Double { return x * x }
  return sqrt( sq(p.x-q.x) + sq(p.y-q.y) )
}

let d = distance( between: Point(x: 0, y: 0),
                  and: Point(x: 2.5, y: 3) )
```

Surcharge de fonctions

On peut définir plusieurs fonctions qui diffèrent par le **type des arguments**:

```
func add( x: Int, y: Int ) -> Int {  
  return x + y  
}
```

```
func add( x: Double, y: Double ) -> Double {  
  return x + y  
}
```

```
let a = add( x: 1, y: 2 )  
let b = add( x: 1.5, y: -2.4 )  
let c = add( x: 0.5, y: 7 )
```


On peut définir plusieurs fonctions qui diffèrent par **le nombre d'arguments**:

```
func add( x: Int ) -> Int {  
  return x  
}
```

```
func add( x: Int, y: Int ) -> Int {  
  return x + y  
}
```

```
func add( x: Int, y: Int, z: Int ) -> Int {  
  return x + y + z  
}
```

```
let a = add( x: 1 )  
let b = add( x: 1, y: 2, z: 3 )
```

On peut définir plusieurs fonctions qui diffèrent par **les noms d'appel des arguments**:

```
func add( x: Int, y: Int ) -> Int {  
  return x + y  
}
```

```
func add( from: Int, to: Int ) -> Int {  
  func tri( _ n: Int ) -> Int {  
    return n * (n+1) / 2  
  }  
  return tri(to) - tri(from)  
}
```

```
let a = add(x: 1, y: 3)  
let b = add(from: 2, to: 5)
```

On peut définir plusieurs fonctions qui diffèrent par le **type de retour**:

```
func add( x: Int, y: Int ) -> Int {  
  return x + y  
}
```

```
func add( x: Int, y: Int ) -> Double {  
  return Double(x + y)  
}
```

```
let a: Int = add(x: 1, y: 2)  
let b: Double = add(x: 1, y: 2)
```

Surcharge de fonctions (4)

On peut définir plusieurs fonctions qui diffèrent par le **type de retour**:

```
func add( x: Int, y: Int ) -> Int {  
    return x + y  
}
```

```
func add( x: Int, y: Int ) -> Double {  
    return Double(x + y)  
}
```

```
let a: Int = add(x: 1, y: 2)  
let b: Double = add(x: 1, y: 2)
```

Attention: Il peut y avoir des ambiguïtés:

```
let c = add(x: 1, y: 2)
```

```
./test.swift:11:9: error: ambiguous use of 'add(x:y:)'  
let c = add(x: 1, y: 2)  
                ^
```

- Utilisez des noms clairs et précis
- Évitez la surcharge des fonctions
- Préférez la portée la plus limitée possible

Branchements

Les **structures de contrôle** permettent de modifier le **flux** d'exécution du programme.

Il en existe plusieurs sortes:

- **branchements**,
- boucles,
- erreurs,
- sauts.

Attention

- Seuls les branchements sont compatibles avec la programmation fonctionnelle stricte.
- Les boucles seront vues dans la partie procédurale du cours.

La déclaration `if...else...` permet de choisir entre deux chemins d'exécution possibles:

```
func abs( _ x: Double ) -> Double {
  if x >= 0 {
    return x
  } else {
    return -x
  }
}

print( abs( 100 ) ) // Affiche 100.0
print( abs( -100 ) ) // Affiche 100.0
print( abs( 0 ) ) // Affiche 0.0
```


If... Else...

```
if x >= 0 {  
    return x  
} else {  
    return -x  
}
```

If... Else...

```
if x >= 0 {  
    return x  
} else {  
    return -x  
}
```

Condition

If... Else...

```
if x >= 0 {  
    return x  
} else {  
    return -x  
}
```

Condition
Cas vrai

If... Else...

```
if x >= 0 {  
    return x  
} else {  
    return -x  
}
```

Condition
Cas vrai
Cas faux

Exemple: If...Else...

```
func abs( _ x: Double? ) -> Double? {  
  if x == nil {  
    return nil  
  } else {  
    return abs( x! )  
  }  
}
```

Exemple: If...Else... (2)

```
func inverse( _ x: Int ) -> Double? {  
  if x == 0 {  
    return nil  
  } else {  
    return 1 / Double(x)  
  }  
}
```

Mauvais Exemple: If...Else...

```
func isGray( color c: Color ) -> Bool {  
  if c.red == c.green && c.red == c.blue {  
    return true  
  } else {  
    return false  
  }  
}
```

Mauvais Exemple: If...Else...

```
func isGray( color c: Color ) -> Bool {  
  if c.red == c.green && c.red == c.blue {  
    return true  
  } else {  
    return false  
  }  
}
```

```
// Meilleur code  
func isGray( color c: Color ) -> Bool {  
  return c.red == c.green && c.red == c.blue  
}
```


Enchaîner les cas: else if

```
func sign( _ x: Double ) -> Double {  
  if x > 0 {  
    return 1  
  } else if x < 0 {  
    return -1  
  } else {  
    return 0  
  }  
}
```

Enchaîner les cas: else if (2)

```
func next( light: TrafficLight ) -> TrafficLight {  
  if light == .red {  
    return .green  
  } else if light == .green {  
    return .yellow  
  } else {  
    return .red  
  }  
}
```

Remarque: On verra une meilleure structure de contrôle pour les énumérations.

On pourrait utiliser un `if` sans la clause fausse. Mais en programmation **fonctionnelle** cette construction est inutile !

On le réservera pour l'affichage (par exemple pour le debug):

```
func twice( _ x: Int ) -> Int {  
  if x == 0 {  
    print( "X est nul !" )  
  }  
  return 2 * x  
}
```

FIN