

Programmation *style* fonctionnelle

Nicolas GLADE
2024

Problématique

Programmation impérative vs fonctionnelle

Programmation impérative / Programmation fonctionnelle

- Programmation impérative

- ▶ Exemples de langages :

- ➔ Impératifs purs : Assembler, Basic, C, Pascal, ...
 - ➔ Multiparadigmes (principalement impératifs) : Python, C++, Java, C#, JS, ...

Programmation impérative / Programmation fonctionnelle

- Programmation impérative

- ▶ Exemples de langages :

- ➔ Impératifs purs : Assembler, Basic, C, Pascal, ...

- ➔ Multiparadigmes (principalement impératifs) : Python, C++, Java, C#, JS, ...

- ▶ Un programme impératif est une **machine à états**. On décrit la succession d'opérations composées de séquences d'instructions modifiant les états (**variables mutables**).

Programmation impérative / Programmation fonctionnelle

- Programmation impérative

- ▶ Exemples de langages :
 - ➔ Impératifs purs : Assembler, Basic, C, Pascal, ...
 - ➔ Multiparadigmes (principalement impératifs) : Python, C++, Java, C#, JS, ...
- ▶ Un programme impératif est une **machine à états**. On décrit la succession d'opérations composées de séquences d'instructions modifiant les états (**variables mutables**).
- ▶ On dit à l'ordinateur quoi faire (**what to do**)

Programmation impérative / Programmation fonctionnelle

```
import numpy as np # pour créer un tableau 2D (espace / temps)
import matplotlib.pyplot as plt # pour l'affichage graphique
nb_x = 300 # nombre de colonnes = x l'espace
nb_t = 100 # nombre de lignes = t le temps
x_t = np.zeros(shape=(nb_t,nb_x),dtype=float)
x_t[0,49] = 1 # on ne fait qu'une seule modification de la ligne de 0 : un 1 au centre
x_t[0,99] = 1 # on ne fait qu'une seule modification de la ligne de 0 : un 1 au centre

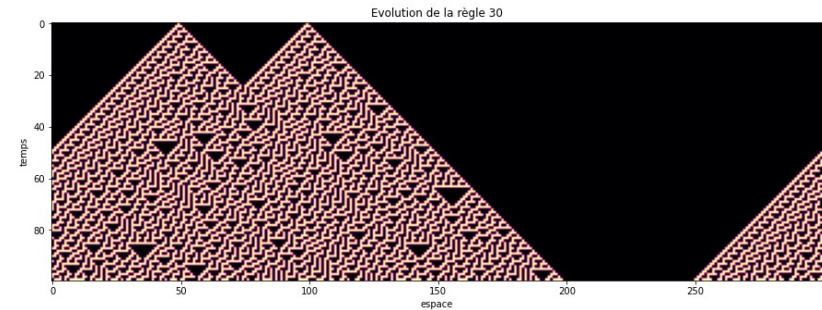
def indice(i):
    return i%nb_x

def evol_regle30(xt_t1, xt_t2) :
    for i in range(nb_x) :
        if xt_t1[i]==1 :
            if xt_t1[indice(i-1)]==1 :
                xt_t2[i]=0
            else :# si la valeur à droite est égale à 0
                xt_t2[i]=1
        else :# si la cellule i est à 0
            if xt_t1[indice(i-1)]+xt_t1[indice(i+1)]==1 :
                xt_t2[i]=1
            else : # si la somme des valeurs à droite et à gauche est différente de 1
                xt_t2[i]=0

for t in range(0,nb_t-1) : # calcule pour tous les temps
    evol_regle30(x_t[t,:], x_t[t+1,:])

fig = plt.figure(figsize=(15, 10))
plt.imshow( x_t , cmap = 'magma' )
plt.title( "Evolution de la règle 7" )
plt.xlabel('espace')
plt.ylabel('temps')
```

Un exemple de programme impératif en Python. Ce programme contient (volontairement) des effets de bord.



Il calcule et affiche un automate cellulaire 1D. Il implémente la règle 30.

Programmation impérative / Programmation fonctionnelle

- Programmation impérative
 - ▶ Un programme impératif est une **machine à états**. On décrit la succession d'opérations composées de séquences d'instructions modifiant les états (**variables mutables**).
 - ▶ On dit à l'ordinateur quoi faire (**what to do**)
- Programmation (déclarative) fonctionnelle
 - ▶ Exemples de langages :
 - ➔ Fonctionnels purs : Lisp, Haskell, Ocaml, Erlang, Purescript, F# ...
 - ➔ Multiparadigmes (rendant possible le style fonctionnel) : C++, Java, C#, Python, Julia, ...

Programmation impérative / Programmation fonctionnelle

- Programmation impérative

- ▶ Un programme impératif est une **machine à états**. On décrit la succession d'opérations composées de séquences d'instructions modifiant les états (**variables mutables**).
- ▶ On dit à l'ordinateur quoi faire (**what to do**)

- Programmation (déclarative) fonctionnelle

- ▶ Exemples de langages :
 - Fonctionnels purs : Lisp, Haskell, Ocaml, Erlang, Purescript, F# ...
 - Multiparadigmes (rendant possible le style fonctionnel) : C++, Java, C#, Python, Julia, ...
- ▶ Un programme fonctionnel est dans l'idée une énorme **fonction (un calcul) mappant** un état initial (entrée) en un état de sortie (output). Il n'y a **pas de changements d'états (immuabilité)**. Le programme décrit un calcul qui est vu comme une **évaluation de fonctions**.

Programmation impérative / Programmation fonctionnelle

- Programmation impérative

- ▶ Un programme impératif est une **machine à états**. On décrit la succession d'opérations composées de séquences d'instructions modifiant les états (**variables mutables**).
- ▶ On dit à l'ordinateur quoi faire (**what to do**)

- Programmation (déclarative) fonctionnelle

- ▶ Exemples de langages :
 - Fonctionnels purs : Lisp, Haskell, Ocaml, Erlang, Purescript, F# ...
 - Multiparadigmes (rendant possible le style fonctionnel) : C++, Java, C#, Python, Julia, ...
- ▶ Un programme fonctionnel est dans l'idée une énorme **fonction (un calcul) mappant** un état initial (entrée) en un état de sortie (output). Il n'y a **pas de changements d'états (immuabilité)**. Le programme décrit un calcul qui est vu comme une **évaluation de fonctions**.
- ▶ Si le programme entier est une énorme fonction (mapping), alors l'ensemble du programme est évaluable.
- ▶ On dit à l'ordinateur comment faire (**How to do**)

Programmation impérative / Programmation fonctionnelle

Ce programme est écrit en Racket (une forme de Lisp), en langage fonctionnel pur.

```
#lang racket
```

```
;; Nombre de colonnes (x l'espace) et de lignes (t le temps)
(define nb-x 30)
(define nb-t 10)

;; Fonction pour afficher la grille (mode texte)
(define (print-grid grid)
  (for-each
   (lambda (row)
     (for-each (lambda (cell) (display cell) (display " ")) (vector->list row))
     (newline)))
  (vector->list grid)))

;; Fonction pour créer une matrice 2D initialisée à zéro
(define (create-matrix rows cols)
  (make-vector rows (make-vector cols 0)))

;; Initialiser la grille avec des valeurs spécifiques
(define (init-grid)
  (let ((grid (create-matrix nb-t nb-x)))
    (vector-set! (vector-ref grid 0) 19 1)
    (vector-set! (vector-ref grid 0) 9 1)
    grid))

;; Calcul de l'indice modulo
(define (indice i n)
  (modulo (+ i n) n))
```

```
;; Fonction pure pour calculer la prochaine ligne
;; en fonction de la règle 30
(define (evol-regle30 xt-t1)
  (define (rule30 i)
    (let ((left (vector-ref xt-t1 (indice (- i 1) nb-x)))
          (center (vector-ref xt-t1 i))
          (right (vector-ref xt-t1 (indice (+ i 1) nb-x))))
      (cond
       ((and (= center 1) (= left 1)) 0)
       ((= center 1) 1)
       ((= (+ left right) 1) 1)
       (else 0))))
  (list->vector (map rule30 (build-list nb-x values)))))

;; Fonction pour calculer toutes les lignes
(define (calcule-toutes-lignes grid)
  (define (evolve grid t)
    (let ((new-line (evol-regle30 (vector-ref grid (- t 1)))))
      (vector-set! grid t new-line)
      grid))
  (foldl (lambda (t g) (evolve g t))
    grid (build-list (- nb-t 1) (lambda (x) (+ x 1)))))

;; Programme principal
(let ((x-t (init-grid)))
  (define result-grid (calcule-toutes-lignes x-t))
  (print-grid result-grid))
```

Programmation impérative / Programmation fonctionnelle

Et en F#, un langage de programmation fonctionnelle

```
module automate

open FSharp.Plotly

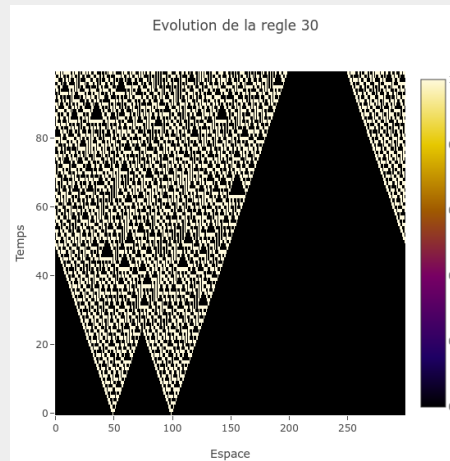
let nb_x = 300 // Nombre de colonnes (espace)
let nb_t = 100 // Nombre de lignes (temps)

// Création d'une matrice 2D initialisée à 0.0
let x_t = Array.init nb_t (fun _ -> Array.create nb_x 0.0)

// Modification de la première ligne
x_t.[0].[49] <- 1.0
x_t.[0].[99] <- 1.0

// Fonction pour gérer les indices circulaires
let indice i = (i + nb_x) % nb_x

// Fonction pour évoluer selon la règle 30
let evolRegle30 (xt_t1: float array) (xt_t2: float array) =
    for i = 0 to nb_x - 1 do
        if xt_t1.[i] = 1.0 then
            if xt_t1.[indice (i - 1)] = 1.0 then
                xt_t2.[i] <- 0.0
            else
                xt_t2.[i] <- 1.0
        else
            if xt_t1.[indice (i - 1)] + xt_t1.[indice (i + 1)] = 1.0 then
                xt_t2.[i] <- 1.0
            else
                xt_t2.[i] <- 0.0
```



// Calculer l'évolution pour tous les temps

```
for t = 0 to nb_t - 2 do
    evolRegle30 x_t.[t] x_t.[t + 1]
```

// Conversion de la matrice en une liste de lignes pour Plotly

```
let matrixAsLists =
    x_t
    |> Array.map (fun row -> row |> Array.toList)
    |> Array.toList
```

// Création du graphique avec Plotly

```
let heatmap =
    Chart.Heatmap(matrixAsLists, Colorscale=StyleParam.Colorscale.Electric)
    |> Chart.withTitle "Evolution de la règle 30"
    |> Chart.withX_AxisStyle "Espace"
    |> Chart.withY_AxisStyle "Temps"
```

// Affichage dans le navigateur et sauvegarde en HTML

```
heatmap
|> Chart.Show
heatmap
|> Chart.SaveHtmlAs "evolution_regle30.html"
```

// Instructions pour Linux

```
printfn "Le graphique a été sauvegardé dans 'evolution_regle30.html'. Ouvrez-le avec votre navigateur."
```

Programmation impérative / Programmation fonctionnelle

Les variables sont immuables
et les fonctions sont pures ou
d'ordre supérieur.

Ce programme produit le même
résultat qu'obtenu
précédemment en Python (à part
l'affichage non graphique).

```
44 (define (print-grid grid)
45   (for-each
46     (lambda (row)
47       (for-each (lambda (cell) (display cell) (display " ")) (vector->list row))
48       (newline))
49     (vector->list grid)))
50
51 ;; Programme principal
52 (let ((x-t (init-grid)))
53   (define result-grid (calcule-toutes-lignes x-t))
54   (print-grid result-grid))
```

Welcome to [DrRacket](#), version 8.6 [cs].

Language: [racket](#), with [debugging](#); memory limit: 128 MB.

```
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 0 0
0 0 0 0 0 1 1 0 0 1 0 0 0 1 0 0 0 1 0 1 1 0 0 1 0 0 0 0 0 0
0 0 0 0 1 1 0 1 1 1 1 1 0 1 1 0 1 0 1 1 1 1 1 0 1 1 0 0 0 0
0 0 0 1 1 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 0 0 0
0 0 1 1 0 1 1 1 1 0 0 1 1 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 0 0
0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 0
1 1 0 1 1 1 1 0 1 1 0 0 1 0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 1 0
>
```

Problèmes posés par la programmation impérative classique

Prog impérative : On fait évoluer une machine à états en actualisant des variables

- Cela autorise beaucoup de choses :
 - ▶ Les variables sont ... variables ! Elles peuvent être modifiées \Leftrightarrow mutées \Rightarrow **Mutabilité**
 - on peut écrire $a=f(a)$ ou $a=f(a,b)$ ou encore $\text{pair}\langle a,b \rangle = f(\text{pair}\langle a,b \rangle) \dots$

Problèmes posés par la programmation impérative classique

Prog impérative : On fait évoluer une machine à états en actualisant des variables

- Cela autorise beaucoup de choses :
 - ▶ Les variables sont ... variables ! Elles peuvent être modifiées \Leftrightarrow mutées \Rightarrow **Mutabilité**
 - on peut écrire $a=f(a)$ ou $a=f(a,b)$ ou encore $\text{pair}\langle a,b \rangle = f(\text{pair}\langle a,b \rangle) \dots$
 - ▶ Le programmeur contrôle la mémoire
 - L'**allocation/désallocation** : il peut créer ou détruire des variables à la volée
 - **La portée** : les variables peuvent être locales ou plus globales (jusqu'à la portée du programme entier).

Problèmes posés par la programmation impérative classique

Prog impérative : On fait évoluer une machine à états en actualisant des variables

- Cela autorise beaucoup de choses :
 - ▶ Les variables sont ... variables ! Elles peuvent être modifiées \Leftrightarrow mutées \Rightarrow **Mutabilité**
 - on peut écrire $a=f(a)$ ou $a=f(a,b)$ ou encore $\text{pair}\langle a,b \rangle = f(\text{pair}\langle a,b \rangle) \dots$
 - ▶ Le programmeur contrôle la mémoire
 - L'**allocation/désallocation** : il peut créer ou détruire des variables à la volée
 - La **portée** : les variables peuvent être locales ou plus globales (jusqu'à la portée du programme entier).
- ... et a des conséquences
 - Les **effets secondaires dits « de bord »** : des variables globales peuvent être modifiées dans une fonction \Rightarrow une fonction n'est plus évaluable !
 - Le **plat de spaghettis** : on suit un ensemble d'états (dont les valeurs changent) passant d'une variable à l'autre, variables pouvant être allouées à la volée ou être détruites (libérées de la mémoire) et on doit les amener à faire ce qu'on a prévu !

Problèmes posés par la programmation impérative classique

Prog impérative : On fait évoluer une machine à états en actualisant des variables

- Cela autorise beaucoup de choses :
 - ▶ Les variables sont ... variables ! Elles peuvent être modifiées \Leftrightarrow mutées \Rightarrow **Mutabilité**
 - \rightarrow on peut écrire $a=f(a)$ ou $a=f(a,b)$ ou encore $\text{pair}\langle a,b \rangle = f(\text{pair}\langle a,b \rangle) \dots$
 - ▶ Le programmeur contrôle la mémoire
 - \rightarrow L'**allocation/désallocation** : il peut créer ou détruire des variables à la volée
 - \rightarrow La **portée** : les variables peuvent être locales ou plus globales (jusqu'à la portée du programme entier).
- ... et a des conséquences
 - \rightarrow **Les effets secondaires dits « de bord »** : des variables globales peuvent être modifiées dans une fonction \Rightarrow une fonction n'est plus évaluable !
 - \rightarrow **Le plat de spaghettis** : on suit un ensemble d'états (dont les valeurs changent) passant d'une variable à l'autre, variables pouvant être allouées à la volée ou être détruites (libérées de la mémoire) et on doit les amener à faire ce qu'on a prévu !

... Wow ! C'est difficile !!! ... c'est vraiment difficile de savoir ce que le programme va faire, sauf à l'exécuter ... dans toutes les situations possibles et imaginables !

\Leftrightarrow « Cela nécessite pour le programmeur d'avoir à tout instant un modèle exact de l'état de la mémoire que le programme modifie (Wiki) ».

Problèmes résolus par la programmation fonctionnelle

Prog fonctionnelle : Il n'y a pas d'états à muter ! \Leftrightarrow On ne fait pas évoluer une machine à états en actualisant des variables

- Cela empêche beaucoup de choses :
 - ▶ Les variables **ne sont plus** ... variables ! Elles **ne peuvent plus** être modifiées \Leftrightarrow non mutables \Leftrightarrow **Immuabilité**
 - on **ne peut pas** écrire $a=f(a)$ ou $a=f(a,b)$ ou encore $\text{pair}\langle a,b \rangle = f(\text{pair}\langle a,b \rangle) \dots$

Problèmes résolus par la programmation fonctionnelle

Prog fonctionnelle : Il n'y a pas d'états à muter ! \Leftrightarrow On ne fait pas évoluer une machine à états en actualisant des variables

- Cela empêche beaucoup de choses :
 - ▶ Les variables **ne sont plus** ... variables ! Elles **ne peuvent plus** être modifiées \Leftrightarrow non mutables \Leftrightarrow **Immuabilité**
 - ➔ on **ne peut pas** écrire $a=f(a)$ ou $a=f(a,b)$ ou encore $\text{pair}\langle a,b \rangle = f(\text{pair}\langle a,b \rangle) \dots$
 - ▶ Le programmeur ne se soucie plus vraiment de la mémoire
 - ➔ On ne fait plus d'**allocation/désallocation** : on ne peut plus créer ou détruire des variables à la volée
 - ➔ **La portée** devient locale : les variables ne sont plus globales

Problèmes résolus par la programmation fonctionnelle

Prog fonctionnelle : Il n'y a pas d'états à muter ! \Leftrightarrow On ne fait pas évoluer une machine à états en actualisant des variables

- Cela empêche beaucoup de choses :
 - ▶ Les variables **ne sont plus** ... variables ! Elles **ne peuvent plus** être modifiées \Leftrightarrow non mutables \Leftrightarrow **Immuabilité**
 - on **ne peut pas** écrire $a=f(a)$ ou $a=f(a,b)$ ou encore $\text{pair}\langle a,b \rangle = f(\text{pair}\langle a,b \rangle) \dots$
 - ▶ Le programmeur ne se soucie plus vraiment de la mémoire
 - On ne fait qu'**instancier pour muter**
 - **La portée** devient locale ; les variables ne sont plus globales
- ... et a des conséquences positives
 - **Il n'y a plus d'effets secondaires dits « de bord »** : des variables globales n'existent plus donc toutes les fonctions sont évaluables !
 - **Il n'y a plus de plat de spaghettis** : il n'y a plus d'états !

Programmation fonctionnelle

Avantages et inconvénients

Avantages de la programmation fonctionnelle

- Ecrire du code plus contrôlé car limitant les effets de bord
- Ecrire du code plus expressif car son expression est centrée sur l'usage des fonctions (et non d'états)
- Ecrire potentiellement moins de code comparativement à du code impératif mal conçu

Avantages de la programmation fonctionnelle

- Ecrire du code plus contrôlé car limitant les effets de bord
- Ecrire du code plus expressif car son expression est centrée sur l'usage des fonctions (et non d'états)
- Ecrire potentiellement moins de code comparativement à du code impératif mal conçu
- Le programme ou des parties peuvent être vues comme un flux de données traversant un ensemble de fonctions successives :

$result = f1(f2(f3(\dots fn(input)))) = F(input).$

- ▶ Cela présente un avantage dans les modèles d'exécution parallèle.
- ▶ Cela sécurise le code et son exécution.
- ▶ Cela permet aussi la preuve de programme

Inconvénients de la programmation fonctionnelle

- Le principal inconvénient est la difficulté que l'on a à la mettre en œuvre. Elle est liée à notre façon de penser le monde (comme des objets dotés de caractéristiques et fonctionnalités).
 - ▶ Cette manière de penser est renforcée par le fait que l'on pratique en premier la programmation impérative et orientée objet !
 - ▶ Penser fonctions qui transforment des valeurs nous paraît plus difficile que de penser l'évolution de ces mêmes valeurs étape par étape, donc avoir une pensée centrée sur ces états et leurs mutations (changements).

Inconvénients de la programmation fonctionnelle

- Le principal inconvénient est la difficulté que l'on a à la mettre en œuvre. Elle est liée à notre façon de penser le monde (comme des objets dotés de caractéristiques et fonctionnalités).
 - ▶ Cette manière de penser est renforcée par le fait que l'on pratique en premier la programmation impérative et orientée objet !
 - ▶ Penser fonctions qui transforment des valeurs nous paraît plus difficile que de penser l'évolution de ces mêmes valeurs étape par étape, donc avoir une pensée centrée sur ces états et leurs mutations (changements).
- L'autre inconvénient vient de la difficulté de concevoir les structures complexes comme les tableaux, les listes chaînées, les arbres, ou plus complexes encore, comme des objets immuables \Leftrightarrow problème de l'instanciation
 - ▶ En réalité, ces structures sont fortement optimisées : seules les parties changeantes sont réinstanciées

Programmation [style] fonctionnelle

Paradigme – Usages en C#

Ce qu'il faut aborder quand on parle PF

- L'**immutabilité**
- La notion d'**objet de première classe** incluant les fonctions
- La notion d'**objets de seconde classe**
- La notion de **fonction**
- Les **fonctions pures** et la notion d'**effet de bord**
- Les **fonctions anonymes**, **lambda** fonctions, **delagates**
- Les **fonctions d'ordre supérieur** (incluant **prédicats**, **transformations** et **regroupements**)
- Les **foncteurs**
- Les **monades**
- La **mémoïsation**
- La **récurtivité**
- Le **branchless programming** (non spécifique mais lié)

Immuabilité

- Une variable est dite **mutable** lorsqu'on peut modifier son emplacement mémoire à tout moment.

Immuabilité

- Une variable est dite **mutable** lorsqu'on peut modifier son emplacement mémoire à tout moment.
- Elle est dite **immuable** quand, après initialisation (runtime), il n'est plus possible de la modifier.
 - ▶ Si on souhaite la modifier, on doit
 - ➔ créer une nouvelle variable (l'instancier)
 - ➔ utiliser une fonction qui transforme la valeur initiale (variable immuable initiale) en nouvelle valeur
 - ➔ stocker cette valeur dans la nouvelle variable immuable (qui est alors initialisée)

Immuabilité

- Une variable est dite **mutable** lorsqu'on peut modifier son emplacement mémoire à tout moment.
- Elle est dite **immuable** quand, après initialisation (runtime), il n'est plus possible de la modifier.
 - ▶ Si on souhaite la modifier, on doit
 - ➔ créer une nouvelle variable (l'instancier)
 - ➔ utiliser une fonction qui transforme la valeur initiale (variable immuable initiale) en nouvelle valeur
 - ➔ stocker cette valeur dans la nouvelle variable immuable (qui est alors initialisée)
- Les variables immuables ne sont pas des **constantes** :
 - ▶ Une constante n'est pas instanciée : son emplacement mémoire et sa valeur sont fixés à la compilation
 - ▶ Une variable immuable est instanciée : son emplacement mémoire et sa valeur sont fixés en runtime

Immuabilité - importance

- Si on utilise des variables **mutables** :
 - ▶ on centre le fonctionnement du programme sur le stockage de valeurs pouvant changer (économie d'instanciations)

Immuabilité - importance

- Si on utilise des variables **mutables** :
 - ▶ on centre le fonctionnement du programme sur le stockage de valeurs pouvant changer (économie d'instanciations)
- Si on utilise des variables **immuables** :
 - ▶ on considère les variables comme des données (au sens immuable du terme donc)
 - ▶ ces données sont amenées à être transformées par leur traversée à travers des fonctions successives

Immuabilité - importance

- Si on utilise des variables **mutables** :
 - ▶ on centre le fonctionnement du programme sur le stockage de valeurs pouvant changer (économie d'instanciations)
- Si on utilise des variables **immuables** :
 - ▶ on considère les variables comme des données (au sens immuable du terme donc)
 - ▶ ces données sont amenées à être transformées par leur traversée à travers des fonctions successives

Rendre une variable immuable, c'est la transformer en donnée et laisser la responsabilité à la succession de fonctions utilisées sur cette donnée d'entrée de calculer la valeur de sortie attendue.


L'usage des variables immuables prendra tout son sens quand les fonctions pures auront été définies.

Immuabilité - C#

```
public class TestImmuable {  
  
    public readonly int _roX;  
    public int _proX { init => _roX = value;  
        get => _roX; }  
  
    public int _apiX { init; get; } // auto-property  
  
    private int _mX;  
    public int _pmX { init => _mX = value;  
        get => _mX; }  
  
    public readonly struct T {  
        public readonly int _t1;  
        public readonly int _t2;  
        public T(int t1) { _t1 = t1; _t2 = _t1 * 42; }  
    }  
  
    public T _tX;  
  
    public int _cX = 42;  
}
```

```
public TestImmuable() {  
    _roX = 1; _proX = 2; _proX++;  
    _mX = 1; _mX++; _pmX = 2; _pmX++;  
    _apiX = 1; _apiX++;  
    _tX = new T(1);  
}  
  
// public void set_roX(int x) { _roX = x; } // erreur  
// public void set_proX(int x) { _proX = x; } // erreur  
public void set_mX(int x) { _mX = x; } // ok (!)  
// public void set_pmX(int x) { _pmX = x; } // erreur  
public void set_tX(T t) { _tX = t; } // ok (!!)  
  
public void Compute() {  
    // _roX++; _proX++; _apiX++; _pmX++; _tX._t1++; _tX._t2++; // erreur  
    _mX++; // ok (!)  
    set_tX(new T(3));  
}  
  
public override string ToString() {  
    return "_roX:"+_roX+"; _proX:"+_proX+"; _mX: "+_mX+"; _pmX:"+_pmX  
        +" ; _apiX :"+_apiX+"; _tX._t1:"+_tX._t1+"; _tX._t2:"+_tX._t2;  
    }  
}
```

```
internal class Program {  
    public static void Main(string[] args) {  
        TestImmuable testImm = new TestImmuable();  
        testImm.Compute();  
        Console.WriteLine(testImm);  
    }  
}
```



```
_roX:3 ; _proX:3 ;  
_mX: 4 ; _pmX:4 ;  
    _apiX :2 ;  
_tX._t1:3 ; _tX._t2:126
```

Classes d'objets

Les objets informatiques sont classables en 2 classes :

- Les **objets de 1ere classe** (*first class objects* ou *first class citizens*).
 - ▶ Ce sont les objets les plus **fondamentaux**
 - ▶ Sont considérés de première classe les objets qui supportent **toutes les opérations disponibles pour des valeurs**
- Les **objets de 2nd classe** (2nd ...)
 - ▶ Ce sont ceux qui ne sont pas de première classe (... oui)

Classe d'objets - Classification 1ere/2eme classe

Être capable de supporter toutes les opérations disponibles pour les valeurs \Leftrightarrow

- être **construit en runtime** \Leftrightarrow non nécessairement constants
- être **passés comme arguments** (de fonctions) \Leftrightarrow pouvoir subir un calcul
- être **retournés** par des fonctions \Leftrightarrow font office de résultat
- être **assignés** à une variable \Leftrightarrow avoir la même nature qu'une valeur

Classe d'objets - Objets de première classe

C'est une **entité informatique** qui peut :

- être construite en runtime
- être passée comme argument (de fonctions)
- être retournée par des fonctions
- être assignée à une variable

Classe d'objets - Objets de première classe

C'est une **entité informatique** qui peut :

- être construite en runtime
- être passée comme argument (de fonctions)
- être retournée par des fonctions
- être assignée à une variable

En PF, ce sont :

- les valeurs non instanciées (constantes, valeurs passées directement en paramètres)
- les variables immuables (données)
- les fonctions pures

Classe d'objets - Objets de seconde classe

Ces objets :

- peuvent être passés en paramètre de fonctions

mais

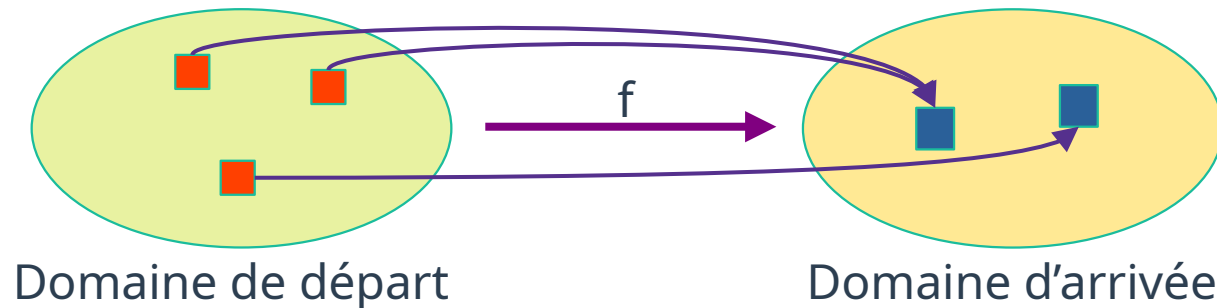
- ne peuvent pas être retournés par une fonction
- ne peuvent pas être assignés à une variable

Classe d'objets - Limitation des langages

- Tous les langages ne peuvent pas traduire dans les faits les propriétés des objets de 1ère classe
- Ex : une fonction pure écrite en C n'est pas un objet de 1ère classe (non assignable)
- Une telle fonction pure qui dans un langage se comporte comme un objet de 2nd classe peut être exprimée en C# comme objet de première classe en utilisant les *lambdas* et les *delegates*.

Fonction

- Mathématiquement :
 - ▶ Une fonction permet de définir un résultat pour chaque élément d'un ensemble appelé domaine.
 - ▶ Ce résultat est obtenu par transformation de valeurs variables passées en paramètres de la fonction.
 - ▶ Elle est l'application d'une transformation d'un domaine de départ en un domaine d'arrivée



Fonction pure

- Une **fonction pure** correspond très exactement à la définition mathématique
- Chaque variable utilisée dans le calcul est passée en paramètre et le résultat est renvoyé en retour.

Fonctions pures / impures

- Aucune variable non-locale (variable membre d'une classe, variable statique) n'est utilisée

```
int x = 3; // une variable non locale à la fonction.
```

```
int f(int y) { // y est locale à la fonction (paramètre)  
    return y*y+x; // le résultat dépend de x !  
}
```

Fonctions pures / impures

- Aucune variable non-locale (variable membre d'une classe, variable statique) n'est utilisée

```
class Test {  
    int _x;  
  
    public Test(int x){ _x = x; }  
    public int g1(int x){ return x*x; }  
    public int g2(int x){  
        _x = x*x;  
        return _x; }  
    public int f1(int y) { return y*y+_x; }  
    public int f2(int y) { return y*y+f(_x); }  
}
```

Fonctions pures / impures

- Aucune variable non-locale (variable membre d'une classe, variable statique) n'est utilisée

```
class Test {  
    int x;
```

```
    public Test(int x){ x = x; }
```

```
    public int g1(int x){ return x*x; }
```

```
    public int g2(int x){
```

```
        x = x*x;
```

```
        return x; }
```

```
    public int f1(int y) { return y*y+x; }
```

```
    public int f2(int y) { return y*y+f(x); }
```

```
}
```

} Ok prog objet

Fonctions pures / impures

- Aucune variable de type référence/pointeur n'est passée par paramètre

```
int f1(int* y) {  
    *y ++;  
    return *y;  
}
```

```
void f2(int *y) {  
    *y ++;  
}
```

Fonctions pures / impures

- Aucune variable de type référence/pointeur n'est passée par paramètre

```
class Object {  
    public:  
        int _x;  
        Object(int x) _x(x) { }  
};  
  
int f1(Object& obj) { return ++obj._x; }  
void f2(Object& obj) { obj._x++; }  
int& f3(Object& obj) { return ++obj._x; }
```

```
int main() {  
    Object* obj = new Object(42);  
    f2(*obj);  
    int x = f1(*obj);  
    std::cout << x << std::endl; //44  
    obj->_x = f1(*obj)+x;  
    f3(*obj)+=f1(*obj);  
    f2(*obj);  
    std::cout << obj->_x << std::endl; //182  
    return 0;  
}
```

Fonctions pures / impures

- Aucun flux d'entrée/sortie (stream) n'est passé par paramètre ou utilisé dans la fonction

```
void f() {  
    Console.WriteLine("Hello");  
}
```

```
String getStringFromStream(StreamReader sr) {  
    String line = String.Empty ;  
    String output = String.Empty ;  
    while ((line = sr.ReadLine()) != null) {  
        output+=line;  
    }  
    return output ;  
}
```

Fonction pure

- Une **fonction pure** correspond très exactement à la définition mathématique
- Chaque variable utilisée dans le calcul est passée en paramètre :
 - ▶ aucune variable non-locale (variable membre d'une classe, variable statique) n'est utilisée
 - ▶ aucune variable de type référence/pointeur n'est passée par paramètre
 - ▶ aucun flux d'entrée/sortie (stream) n'est passé par paramètre
- Elles peuvent (selon le langage) se comporter comme des objets de 1ère classe

Fonctions anonymes

- Une **fonction anonyme** est une fonction ... qui n'est pas nommée.
- Parce que ces fonctions n'ont pas de nom, à l'endroit où l'on voudrait mettre leur nom, on trouve directement les instructions définissant la fonction introduites par une syntaxe particulière. [Wikipedia]
- Tous les langages ne sont pas capables de faire des fonctions anonymes.
- En programmation, les fonctions anonymes sont nommées **lambdas**.
- Les lambdas sont des **objets de première classe**.

Fonctions lambda / Expression lambda

- En C++ :

```
#include <iostream>
#include <functional>
using namespace std;

int main() {
    cout << [](float a, float b) { return (std::abs(a) < std::abs(b)); }(3,5) << endl;
    std::function<bool(float a, float b)> b_sup_a = [](float a, float b) { return (std::abs(a) < std::abs(b)); };
    cout << b_sup_a(3,5) << endl;
    return 0;
}
```

- Et en C# :

```
// En C#
Func<double, double, bool> b_sup_a;
b_sup_a = (double a, double b) => b > a;
Console.WriteLine(b_sup_a(42,1));
Console.WriteLine( ((Func<double,double,bool>)((a,b) => b>a))(3,42) );
```

Expressions lambdas



Fonctions lambda / Expression lambda - delegates

- En C++ :

```
#include <iostream>
#include <functional>
using namespace std;

int main() {
    cout << [](float a, float b) { return (std::abs(a) < std::abs(b)); }(3,5) << endl;
    std::function<bool(float a, float b)> b_sup_a = [](float a, float b) { return (std::abs(a) < std::abs(b)); };
    cout << b_sup_a(3,5) << endl;
    return 0;
}
```

- Et en C# :

```
// En C#
Func<double, double, bool> b_sup_a;
b_sup_a = (double a, double b) => b > a;
Console.WriteLine(b_sup_a(42,1));
Console.WriteLine(((Func<double,double,bool>)((a,b) => b>a))(3,42));
```

Délégués (type référençant une méthode)

Lambdas – des objets de 1^{ère} classe

```
namespace Functional;

public class TestLambdas {

    public static List<double> Generate(int n) {
        List<double> numbers = new List<double>(n);
        for (int i=0; i < n; i++) {
            numbers.Add(i);
        }
        return numbers;
    }
}
```

```
public static List<double> Transform(List<double> numbers, Func<double, double> f) {
    List<double> tnumbers = new List<double>();
    foreach (var v in numbers) {
        tnumbers.Add(f(v));
    }
    return tnumbers;
}
```

Fonction d'ordre supérieur

```
using Functional;

internal class Program {
    foreach (var _c in TestLambdas.Transform(
        TestLambdas.Transform(
            TestLambdas.Generate(10),
            x => x * x ), x => x / 2 ))
        { Console.WriteLine(_c); }
}
```

Expressions lambdas

Délégué

0
0,5
2
4,5
8
12,5
18
24,5
32
40,5

Lambdas et délégués

```
public static void Main(string[] args) {  
    Func<double,double> carre;  
    carre = delegate(double x) { return x * x; };  
    //ou  
    carre = x => x * x;  
    Func<double, double> div = x => x / 2;  
    foreach(var _c in TestLambdas.Transform(  
        TestLambdas.Transform(  
            TestLambdas.Generate(10),  
            carre), div))  
        { Console.Write(_c+" "); }  
  
    Console.WriteLine();  
    // ou  
    foreach(var _c in TestLambdas.Transform(  
        TestLambdas.Transform(  
            TestLambdas.Generate(10),  
            x => x * x), x => x / 2))  
        { Console.Write(_c+" "); }  
    }  
    Console.WriteLine();  
    // ou  
    foreach(var _c in TestLambdas.Transform(  
        TestLambdas.Transform(  
            TestLambdas.Generate(10),  
            delegate(double x) { return x * x; },  
            delegate(double x) { return x/2; } ))  
        { Console.Write(_c+" "); }  
    }  
}
```

Expressions lambdas
Délégués (type Func)
Délégués anonymes



0
0,5
2
4,5
8
12,5
18
24,5
32
40,5

Fonctions lambda / Expression lambda

- En C#, une lambda peut
 - ▶ être exprimée **de façon anonyme** (inline)
 - ▶ Ou bien être récupérée dans une variable de type **Func** ou **Action** (un délégué)
 - ➔ **Action<T...>** si pas de valeur de retour
 - ➔ **Func<T..., ReturnType>** si valeur de retour
- Elles peuvent ainsi être
 - ▶ construites en runtime
 - ▶ être assignée à une variable
 - ▶ passées comme argument (d'autres fonctions)
 - ▶ retournées par des fonctions

Objets de 1^{ère} classe

Délégués

- Les **délégués** (delegates) sont des **références vers des méthodes**
- Un délégué se déclare comme suit :
 - ▶ Soit c'est un délégué générique : `public delegate double OperationMethod(double, double);`
 - ▶ Soit c'est une `Func<T..., Result>` : `public Func<double,double,double> OperationMethod;`
 - ▶ Soit c'est une `Action<T...>` : `public Action<double,double> ComputeMethod;`
- Lorsqu'on instancie un délégué, on peut associer à son instance **n'importe quelle méthode ayant la même signature**, lambdas comprises.
- On peut aussi **retourner un délégué** depuis une fonction.
- Un délégué est un **objet de 1^{ère} classe**

Délégués

```
using System.Numerics;
namespace Functional;
```

Signatures identiques

```
public class Moves {
    public static Vector3 Up(Vector3 pos, float move) { return new Vector3(pos.X, pos.Y+move, pos.Z); }
    public static Vector3 Down(Vector3 pos, float move) { return new Vector3(pos.X, pos.Y-move, pos.Z); }
    public static Vector3 Left(Vector3 pos, float move) { return new Vector3(pos.X-move, pos.Y, pos.Z); }
    public static Vector3 Right(Vector3 pos, float move) { return new Vector3(pos.X+move, pos.Y, pos.Z); }
    public static Vector3 Far(Vector3 pos, float move) { return new Vector3(pos.X, pos.Y, pos.Z+move); }
    public static Vector3 Near(Vector3 pos, float move) { return new Vector3(pos.X, pos.Y, pos.Z-move); }
}
```

```
public class Creature {
    public enum MoveType { North, South, West, East, Upstairs, Downstairs};
    public delegate Vector3 MoveMethod(Vector3 pos, float move);
    private List<MoveMethod> _Moves;
    private Vector3 _pos;
```

Délégué (déclaration)

```
    public Creature(Vector3 pos) {
        _pos = pos;
        _Moves = new List<MoveMethod>(20);
    }
    // ...
```

```
//...
```

```
public void AddMove(MoveType m) {
    switch (m) {
        case MoveType.North : _Moves.Add(Moves.Up); break;
        case MoveType.South : _Moves.Add(Moves.Down); break;
        case MoveType.West : _Moves.Add(Moves.Left); break;
        case MoveType.East : _Moves.Add(Moves.Right); break;
        case MoveType.Upstairs : _Moves.Add(Moves.Far); break;
        case MoveType.Downstairs : _Moves.Add(Moves.Near); break;
    }
}

public MoveMethod? GetMove(int index) {
    if (index >= 0 && index < _Moves.Count)
        return _Moves[index];
    return null;
}

public void MoveAlongPath() {
    foreach (var moveFunc in _Moves) {
        _pos = moveFunc(_pos, 1);
        Console.WriteLine("Position : "+_pos);
    }
}
}
```


Délégués

```
using System.Numerics;
namespace Functional;
```

Signatures identiques

```
public class Moves {
```

```
    public static Vector3 Up(Vector3 pos, float move)
    public static Vector3 Down(Vector3 pos, float move)
    public static Vector3 Left(Vector3 pos, float move)
    public static Vector3 Right(Vector3 pos, float move)
    public static Vector3 Far(Vector3 pos, float move)
    public static Vector3 Near(Vector3 pos, float move)
}
```

```
{ return new Vector3(pos.X, pos.Y+move, pos.Z); }
{ return new Vector3(pos.X, pos.Y-move, pos.Z); }
{ return new Vector3(pos.X-move, pos.Y, pos.Z); }
{ return new Vector3(pos.X+move, pos.Y, pos.Z); }
{ return new Vector3(pos.X, pos.Y, pos.Z+move); }
{ return new Vector3(pos.X, pos.Y, pos.Z-move); }
```

```
public class Creature {
```

```
    public enum MoveType { North, South, West, East, Upstairs, Downstairs};
    public delegate Vector3 MoveMethod(Vector3 pos, float move);
    private List<MoveMethod> _Moves;
    private Vector3 _pos;
```

```
    public Creature(Vector3 pos) {
        _pos = pos;
        _Moves = new List<MoveMethod>(20);
    }
```

```
// ...
```

Délégué (déclaration)

Instanciation / Affectation méthodes

```
//...
```

```
public void AddMove(MoveType m) {
    switch (m) {
        case MoveType.North : _Moves.Add(Moves.Up); break;
        case MoveType.South : _Moves.Add(Moves.Down); break;
        case MoveType.West : _Moves.Add(Moves.Left); break;
        case MoveType.East : _Moves.Add(Moves.Right); break;
        case MoveType.Upstairs : _Moves.Add(Moves.Far); break;
        case MoveType.Downstairs : _Moves.Add(Moves.Near); break;
    }
}
```

```
public MoveMethod? GetMove(int index) {
    if (index >= 0 && index < _Moves.Count)
        return _Moves[index];
    return null;
}
```

```
public void MoveAlongPath() {
    foreach (var moveFunc in _Moves) {
        _pos = moveFunc(_pos, 1);
        Console.WriteLine("Position : "+_pos);
    }
}
```

```
}
```

Délégués

```
using System.Numerics;
namespace Functional;
```

Signatures identiques

```
public class Moves {
```

```
    public static Vector3 Up(Vector3 pos, float move)
    public static Vector3 Down(Vector3 pos, float move)
    public static Vector3 Left(Vector3 pos, float move)
    public static Vector3 Right(Vector3 pos, float move)
    public static Vector3 Far(Vector3 pos, float move)
    public static Vector3 Near(Vector3 pos, float move)
}
```

```
{ return new Vector3(pos.X, pos.Y+move, pos.Z); }
{ return new Vector3(pos.X, pos.Y-move, pos.Z); }
{ return new Vector3(pos.X-move, pos.Y, pos.Z); }
{ return new Vector3(pos.X+move, pos.Y, pos.Z); }
{ return new Vector3(pos.X, pos.Y, pos.Z+move); }
{ return new Vector3(pos.X, pos.Y, pos.Z-move); }
```

```
public class Creature {
```

```
    public enum MoveType { North, South, West, East, Upstairs, Downstairs};
    public delegate Vector3 MoveMethod(Vector3 pos, float move);
    private List<MoveMethod> _Moves;
    private Vector3 _pos;
```

Délégué (déclaration)

```
    public Creature(Vector3 pos) {
```

```
        _pos = pos;
```

```
        _Moves = new List<MoveMethod>(20);
```

Instanciation / Affectation méthodes

```
// ...
```

Usage des fonctions stockées

```
//...
```

```
public void AddMove(MoveType m) {
```

```
    switch (m) {
```

```
        case MoveType.North : _Moves.Add(Moves.Up); break;
        case MoveType.South : _Moves.Add(Moves.Down); break;
        case MoveType.West : _Moves.Add(Moves.Left); break;
        case MoveType.East : _Moves.Add(Moves.Right); break;
        case MoveType.Upstairs : _Moves.Add(Moves.Far); break;
        case MoveType.Downstairs : _Moves.Add(Moves.Near); break;
    }
```

```
public MoveMethod? GetMove(int index) {
```

```
    if (index >= 0 && index < _Moves.Count)
        return _Moves[index];
    return null;
}
```

```
public void MoveAlongPath() {
```

```
    foreach (var moveFunc in _Moves) {
        _pos = moveFunc(_pos, 1);
        Console.WriteLine("Position : "+_pos);
    }
```

```
}
```

Délégués

```
using System.Numerics;
namespace Functional;
```

```
public class Moves {
```

```
    public static Vector3 Up(Vector3 pos, float move)
    public static Vector3 Down(Vector3 pos, float move)
    public static Vector3 Left(Vector3 pos, float move)
    public static Vector3 Right(Vector3 pos, float move)
    public static Vector3 Far(Vector3 pos, float move)
    public static Vector3 Near(Vector3 pos, float move)
}
```

Signatures identiques

```
{ return new Vector3(pos.X, pos.Y+move, pos.Z); }
{ return new Vector3(pos.X, pos.Y-move, pos.Z); }
{ return new Vector3(pos.X-move, pos.Y, pos.Z); }
{ return new Vector3(pos.X+move, pos.Y, pos.Z); }
{ return new Vector3(pos.X, pos.Y, pos.Z+move); }
{ return new Vector3(pos.X, pos.Y, pos.Z-move); }
```

```
public class Creature {
```

```
    public enum MoveType { North, South, West, East, Upstairs, Downstairs};
    public delegate Vector3 MoveMethod(Vector3 pos, float move);
    private List<MoveMethod> _Moves;
    private Vector3 _pos;
```

```
    public Creature(Vector3 pos) {
        _pos = pos;
        _Moves = new List<MoveMethod>(20);
    }
```

```
// ...
```

Délégué (déclaration)

Instanciation / Affectation méthodes

Retour d'un délégué

Usage des fonctions stockées

```
//...
```

```
public void AddMove(MoveType m) {
    switch (m) {
        case MoveType.North : _Moves.Add(Moves.Up); break;
        case MoveType.South : _Moves.Add(Moves.Down); break;
        case MoveType.West : _Moves.Add(Moves.Left); break;
        case MoveType.East : _Moves.Add(Moves.Right); break;
        case MoveType.Upstairs : _Moves.Add(Moves.Far); break;
        case MoveType.Downstairs : _Moves.Add(Moves.Near); break;
    }
}
```

```
public MoveMethod? GetMove(int index) {
    if (index >= 0 && index < _Moves.Count)
        return _Moves[index];
    return null;
}
```

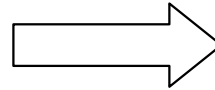
```
public void MoveAlongPath() {
    foreach (var moveFunc in _Moves) {
        _pos = moveFunc(_pos, 1);
        Console.WriteLine("Position : "+_pos);
    }
}
```

Délégués

```
using System.Numerics;
using Functional;

internal class Program_Creatures {

    public static void Main(string[] args) {
        Creature creature = new Creature(Vector3.Zero);
        creature.AddMove(Creature.MoveType.North);
        creature.AddMove(Creature.MoveType.North);
        creature.AddMove(Creature.MoveType.East);
        creature.AddMove(Creature.MoveType.North);
        creature.AddMove(Creature.MoveType.East);
        creature.AddMove(Creature.MoveType.West);
        creature.AddMove(Creature.MoveType.Downstairs);
        creature.AddMove(Creature.MoveType.North);
        creature.MoveAlongPath();
        Console.WriteLine("Utilisation de GetMove : "+creature.GetMove(3)(Vector3.Zero,2));
    }
}
```



Position : <0 1 0>
Position : <0 2 0>
Position : <1 2 0>
Position : <1 3 0>
Position : <2 3 0>
Position : <1 3 0>
Position : <1 3 -1>
Position : <1 4 -1>
Utilisation de GetMove : <0 2 0>

Fonction d'ordre supérieur

- Une **fonction d'ordre supérieur** ont au moins une des propriétés suivantes :
 - ▶ Prend au moins une à plusieurs **fonctions en paramètre**
 - ▶ **Retourne une fonction** (on dit dans ce cas que la fonction est **curryfiée**)

Fonction d'ordre supérieur

- Une **fonction d'ordre supérieur** ont au moins une des propriétés suivantes :
 - ▶ Prend au moins une à plusieurs **fonctions en paramètre**
 - ▶ **Retourne une fonction** (on dit dans ce cas que la fonction est **curryfiée**)
- Exemples :
 - ▶ les fonctions de type « **map** » :
 - ➔ prennent une fonction f et un type énumérable (ex : une liste) en paramètre
 - ➔ appliquent la fonction f sur les valeurs énumérables
 - ➔ retournent les nouvelles valeurs énumérables
 - ▶ Les fonctions de type « **filter** » :
 - ➔ idem mais renvoient les éléments sélectionnés selon une condition
 - ▶ Les fonctions de type « **reduce** » :
 - ➔ idem mais renvoient une simple valeur

Fonction d'ordre supérieur

- Exemple déjà vu :

```
public class TestLambdas {  
  
    public static List<double> Generate(int n) {  
        List<double> numbers = new List<double>(n);  
        for (int i=0; i < n; i++) {  
            numbers.Add(i);  
        }  
        return numbers;  
    }  
  
    public static List<double> Transform(List<double> numbers, Func<double, double> f) {  
        List<double> tnumbers = new List<double>();  
        foreach (var v in numbers) {  
            tnumbers.Add(f(v));  
        }  
        return tnumbers;  
    }  
}
```

Notre fonction « Transform » :

- Prend une fonction en paramètre
 - Prend un énumérable en paramètre
 - Retourne un énumérable
- C'est une fonction **Map**

```
public static void Main(string[] args) {  
    Func<double, double> carre;  
    carre = delegate(double x) { return x * x; };  
    //ou  
    carre = x => x * x;  
    Func<double, double> div = x => x / 2;  
    foreach (var _c in TestLambdas.Transform(  
        TestLambdas.Transform(  
            TestLambdas.Generate(10),  
            carre), div))  
        { Console.Write(_c + " "); }  
}
```

Fonction d'ordre supérieur

- Exemple : fonction **Map** - Sur des **IEnumerable<T>** (List / Array ...)

```
public static class FunctionalIEnumerable {  
  
    public static IEnumerable<U> Map<T, U>(this IEnumerable<T> s, Func<T, U> f) {  
        foreach (var item in s)  
            yield return f(item);  
    }  
  
    public static U Reduce<U>(IEnumerable<U> s, Func<U, U, U> f) {  
        U result = default;  
        foreach (var e in s) {  
            result = f(result, e);  
        }  
        return result;  
    }  
  
    // ...  
}
```

```
namespace Functional;  
  
internal class Program_Functional {  
  
    public static List<int> Generate(int n) {  
        List<int> numbers = new List<int>(n);  
        for (int i = 0; i < n; i++) { numbers.Add(i); }  
        return numbers;  
    }  
  
    public static void Main(string[] args) {  
        List<int> numbers = Generate(10);  
        numbers = FunctionalIEnumerable.Map<int, int>(numbers, x => x * x).ToList();  
        int sum = FunctionalIEnumerable.Reduce<int>(  
            numbers,  
            delegate(int result, int x) { return result+x; }  
        );  
        Console.WriteLine(sum);  
    }  
}
```

285

Fonction d'ordre supérieur

- Exemple : Sur un arbre

```
namespace Functional;
public class Tree<T> {
    public class Node {
        public T data;
        public Node leftnode, rightnode;

        public static Node NewInstance(T val) {
            Node y = new Node();
            y.data = val;
            y.leftnode = null;
            y.rightnode = null;
            return y;
        }
    };

    private Node _RootNode = null;
    public Node RootNode { init => _RootNode = value; get => _RootNode; }
    private List<Node> _NodeList = new List<Node>();
    public List<Node> NodeList { init => _NodeList = value; get => _NodeList; }
```

//...

// ...

```
public void Add(T newval) {
    Node node = Node.NewInstance(newval);
    if (_RootNode == null) { _RootNode = node; }
    else if (_NodeList[0].leftnode == null) { _NodeList[0].leftnode = node; }
    else {
        _NodeList[0].rightnode = node;
        _NodeList.RemoveAt(0);
    }
    _NodeList.Add(node);
}

public void Build(T[] ar) {
    foreach (var a in ar) { Add(a); }
}
}
```

Fonction d'ordre supérieur

- Exemple : Sur un arbre

```
public static class FunctionalTree {  
  
    public static Tree<U> Map<T, U>(this Tree<T> s, Func<T, U> f) {  
        Tree<U> tree = new Tree<U>();  
        foreach (var item in s.NodeList)  
            tree.Add((U)f(item.data));  
        return tree;  
    }  
  
    public static U Reduce<U>(Tree<U> s, Func<U, U, U> f) {  
        U result = default;  
        foreach (var e in s.NodeList) {  
            result = f(result, e.data);  
        }  
        return result;  
    }  
  
    // ...  
}
```

```
namespace Functional;  
  
internal class Program_Functional {  
  
    public static List<int> Generate(int n) {  
        List<int> numbers = new List<int>(n);  
        for (int i = 0; i < n; i++) { numbers.Add(i); }  
        return numbers;  
    }  
  
    public static void Main(string[] args) {  
        Tree<int> tree = new Tree<int>();  
        tree.Add(1);  
        tree.Build([1,2,3]);  
        tree = FunctionalTree.Map<int, int>(tree, x => x / 2);  
        Func<int, int, int> sumFunc = delegate(int result, int x) { return result + x; };  
        sum = FunctionalTree.Reduce<int>(tree, sumFunc);  
        Console.WriteLine(sum);  
    }  
}
```

2



Notion de Foncteur

- Un **foncteur** est un mapping entre catégories (vient de la théorie des catégories)

Notion de Foncteur

- Un **foncteur** est un mapping entre catégories (vient de la théorie des catégories)
- Une **catégorie** est un objet mathématique qui est défini par :
 - ▶ des **objets** A (ex : des valeurs)
 - ▶ des **morphismes** m (\Leftrightarrow des transformations entre objets)
 - ▶ des opérations \circ (ex : composition $h=f\circ g$) entre morphismes
 - ▶ des morphismes particuliers (ex : identité $Id : A \rightarrow A$)

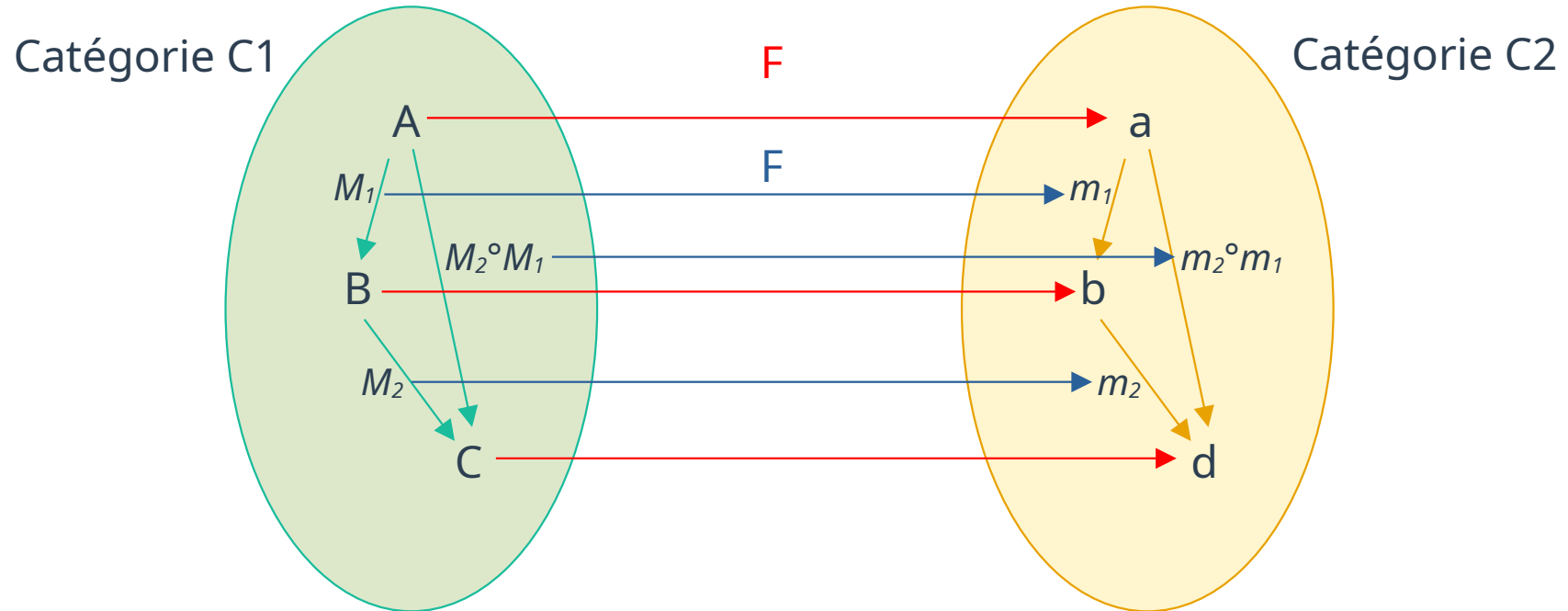
Notion de Foncteur

- Un **foncteur** est un mapping entre catégories (vient de la théorie des catégories)
- Une **catégorie** est un objet mathématique qui est défini par :
 - ▶ des **objets** A (ex : des valeurs)
 - ▶ des **morphismes** m (\Leftrightarrow des transformations entre objets)
 - ▶ des opérations \circ (ex : composition $h=f\circ g$) entre morphismes
 - ▶ des morphismes particuliers (ex : identité $Id : A \rightarrow A$)
- Exemples de catégories (info) : les réels, les entiers, les booléens, les strings
 - ▶ Chaque catégorie est peuplée d'objets (valeurs) et de morphismes (fonctions possibles)

Notion de Foncteur

- Un **foncteur** est un mapping entre catégories (vient de la théorie des catégories)
- Une **catégorie** est un objet mathématique qui est défini par :
 - ▶ des **objets** A (ex : des valeurs)
 - ▶ des **morphismes** m (\Leftrightarrow des transformations entre objets)
 - ▶ des opérations \circ (ex : composition $h=f\circ g$) entre morphismes
 - ▶ des morphismes particuliers (ex : identité $Id : A \rightarrow A$)
- Exemples de catégories (info) : les réels, les entiers, les booléens, les strings
 - ▶ Chaque catégorie est peuplée d'objets (valeurs) et de morphismes (fonctions possibles)
- Un foncteur « explique » donc comment transformer
 - ▶ des objets d'une catégorie en d'autres objets d'une autre catégorie
 - ▶ des morphismes d'une catégorie en d'autres morphismes d'autres catégories

Notion de Foncteur

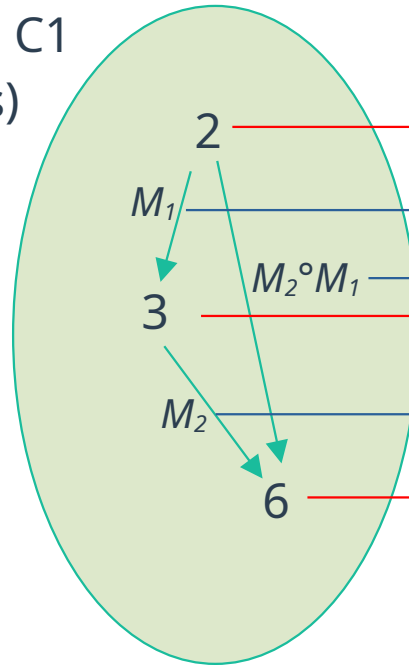


$$\begin{aligned} m_1 &= F(M_1) \\ m_2 &= F(M_2) \\ m_2 \circ m_1 &= F(M_2 \circ M_1) = F(M_2) \circ F(M_1) \end{aligned}$$

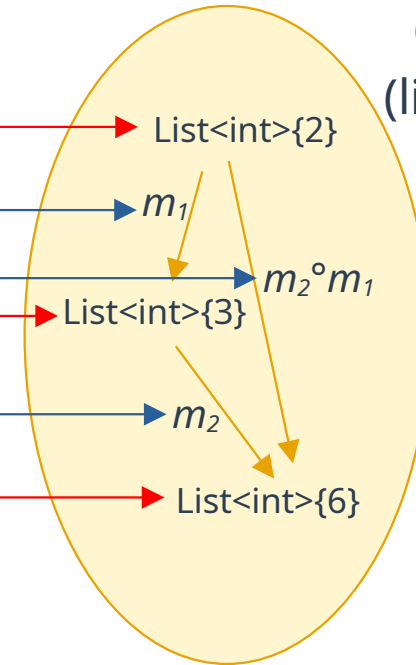
Le diagramme commute
(préservation de la structure)

Notion de Foncteur - Exemple

Catégorie C1
(entiers)



Catégorie C2
(listes d'entiers)



F

F

$$\begin{aligned} m_1 &= F(M_1) \\ m_2 &= F(M_2) \\ m_2 \circ m_1 &= F(M_2 \circ M_1) = F(M_2) \circ F(M_1) \end{aligned}$$

Le diagramme commute
(préservation de la structure)

Notion de Monade

- **Monade** vient du grec *μονάς* : *unité*
- Une monade combine les avantages de la P. Fonctionnelle et de la P. Impérative

Notion de Monade

- **Monade** vient du grec *μονάς* : *unité*
- Une monade combine les avantages de la P. Fonctionnelle et de la P. Impérative
- C'est un type qui encapsule :
 - ▶ des valeurs
 - ▶ des fonctions
- Cela :
 - ▶ évite les effets de bords (tout est encapsulé dans la monade)
 - ▶ facilite la preuve de programme
 - ▶ optimise la parallélisation
 - ▶ ...

Définition formelle de la Monade

- Une **Monade** [Wikipédia] est constituée de :
 - ▶ Un **constructeur monadique** : associée $M : t \rightarrow Mt$
 - ▶ Une fonction **unit** ou **return**, qui construit à partir d'un élément de type sous-jacent **a** un autre objet de type monadique **Ma**. Cette fonction est alors de signature **return**: $t \rightarrow Mt$
 - ▶ Une fonction **bind**, représentée par l'opérateur **infixe** $\gg=$, associant à un type monadique et une fonction d'association à un autre type monadique. Il permet de composer une fonction monadique à partir d'autres fonctions monadiques.

Cet opérateur est de type $\gg= : Mt \rightarrow (t \rightarrow Mu) \rightarrow Mu$

Exemple de Monade (Maybe / Just) en C#

```
namespace Functional;
```

```
//----- IMonade (Maybe) -----
```

```
public interface IMonade<T> {  
    public IMonade<T> unit(Func<T, T, T> f, T v);  
    public IMonade<T> bind(Func<T, IMonade<T>> f);  
    public T? Get();  
}
```

```
//----- Nothing -----
```

```
public class Nothing<T> : IMonade<T> {  
    public IMonade<T> unit(Func<T, T, T> f, T v) => this;  
    public IMonade<T> bind(Func<T, IMonade<T>> f) => this;  
    public T? get() => this.get();  
}
```

```
//----- MonadeImpl (Just) -----
```

```
public class MonadeImpl<T> : IMonade<T> {  
    private T val;  
  
    public MonadeImpl(T val) {this.val = val;}  
  
    public IMonade<T> unit(Func<T, T, T> f, T v) {  
        return (this.val == null || this.val is Nothing<T>)  
            ? new Nothing<T>()  
            : new MonadeImpl<T>(f(this.val, v));  
    }  
  
    public IMonade<T> bind(Func<T, IMonade<T>> f) {  
        return (this.val == null || this.val is Nothing<T>)  
            ? new Nothing<T>()  
            : f(this.val);  
    }  
  
    public T? get() => this.val;  
}
```

```
namespace Functional;
```

```
// ----- MAIN -----
```

```
internal class Program_Functional {  
  
    public static void Main(string[] args) {  
        MonadeImpl<string> a  
            = new MonadeImpl<string>(String.Empty);  
  
        Func<string, string, string> remove  
            = delegate(String S, String subS) {  
                int index = S.IndexOf(subS);  
                if (index >= 0) return S.Remove(index, subS.Length);  
                return S;  
            };  
  
        string s = a  
            .unit((el,s) => el+s, "Hello")  
            .unit((el,s) => el+s, " Nico")  
            .unit(remove, " Nico")  
            .unit((el,s) => el+s, " World")  
            .unit((el,s) => el+s, "!!")  
            .get(); // => Hello World!  
        Console.WriteLine(s);  
    }  
}
```

C# LINQ

- C# fournit un moyen de chaîner des opérations : LINQ = Language Integrated Queries
- Elimine les effets de bord
- Une expression de requête :
 - ▶ Est un objet du 1^{er} ordre
 - ▶ Contient 3 clauses
 - **From** : spécifie la source de données
 - **Where** : applique le filtre
 - **Select** : spécifie le type des éléments retournés

```
using Functional;

internal class Program_Lambdas {


    public static void Main(string[] args) {

        Func<double,double> carre = x => x * x;
        Func<double, double> div = x => x / 2;

        var squares = TestLambdas.Transform(TestLambdas.Generate(10), carre);

        List<double> some_squares =
            (from s in squares
             where ((s % 2) == 0) && s > 10
             select div(s)).ToList();

        foreach (var v in some_squares) {
            Console.WriteLine(v);
        }
    }
}
```



8
18
32

Récurtivité

- La récursivité est un principe par lequel un algorithme s'invoque lui même (auto-référence) :
 - ▶ La fonction (algorithme) f fait un traitement sur des valeurs d'entrée A ...
 - ▶ ... et appelle f en lui fournissant ses valeurs traitées B comme nouvelle entrée A ...
 - ▶ ... et appelle f en lui fournissant ses valeurs traitées B comme nouvelle entrée A ...
 - ▶ ...
 - ▶ jusqu'à ce que le traitement soit terminé et, dans ce cas, f n'appelle pas f
- Donc cela revient à écrire :

$$f(f(f(f(\dots f(A_0)\dots)))) \iff \begin{array}{lcl} A_1 & = & f(A_0) \\ A_2 & = & f(A_1) \\ \dots & = & \dots \\ A_i & = & f(A_{i-1}) \end{array}$$

Récurtivité

- Exemple :

```
static public int Fibonacci(int n) {  
    if (n > 1) return Fibonacci(n - 1) + Fibonacci(n - 2);  
    if (n == 1) return 1;  
    return 0;  
}
```
- L'intérêt de la récursivité est que :
 - ▶ aucune boucle impliquant un compteur n'est impliquée.
 - ▶ la fonction s'appellant elle-même, les effets de bords sont limités

Récurtivité – HowTo ?

- Il suffit de se rappeler ce que la fonction ***f*** renvoie comme type de retour et que ce retour est utilisé comme argument ***v*** dans la fonction appelée dans le return.
- On écrit ainsi la fonction normalement sauf que :
 - ▶ On retourne non pas ***v*** mais ***f(v)***
 - ▶ Sauf si la condition d'arrêt est atteinte, auquel cas on renvoie ***v₀***

$$f(f(f(f(\dots f(A_0)\dots)))) \iff \begin{array}{lcl} A_1 & = & f(A_0) \\ A_2 & = & f(A_1) \\ \dots & = & \dots \\ A_i & = & f(A_{i-1}) \end{array}$$

Mémoïzation

- Le principe est de mettre en mémoire (cache) les valeurs de retour des fonctions en fonction des valeurs d'entrée de façon à optimiser les calculs.

Mémoïzation

- Analysons la sortie de :

```
static public int Fibonacci(int n) {  
    if (n > 2) {  
        Console.WriteLine("n="+n+" / n-1="+n-1+" / n-2="+n-2);  
        return Fibonacci(n - 1) + Fibonacci(n - 2);  
    }  
    return 1;  
}
```

Mémoïzation

- Analysons la sortie de :

```
static public int Fibonacci(int n) {
    if (n > 2) {
        Console.WriteLine("n="+n+" / n-1="+n-1+" / n-2="+n-2);
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
    return 1;
}
```

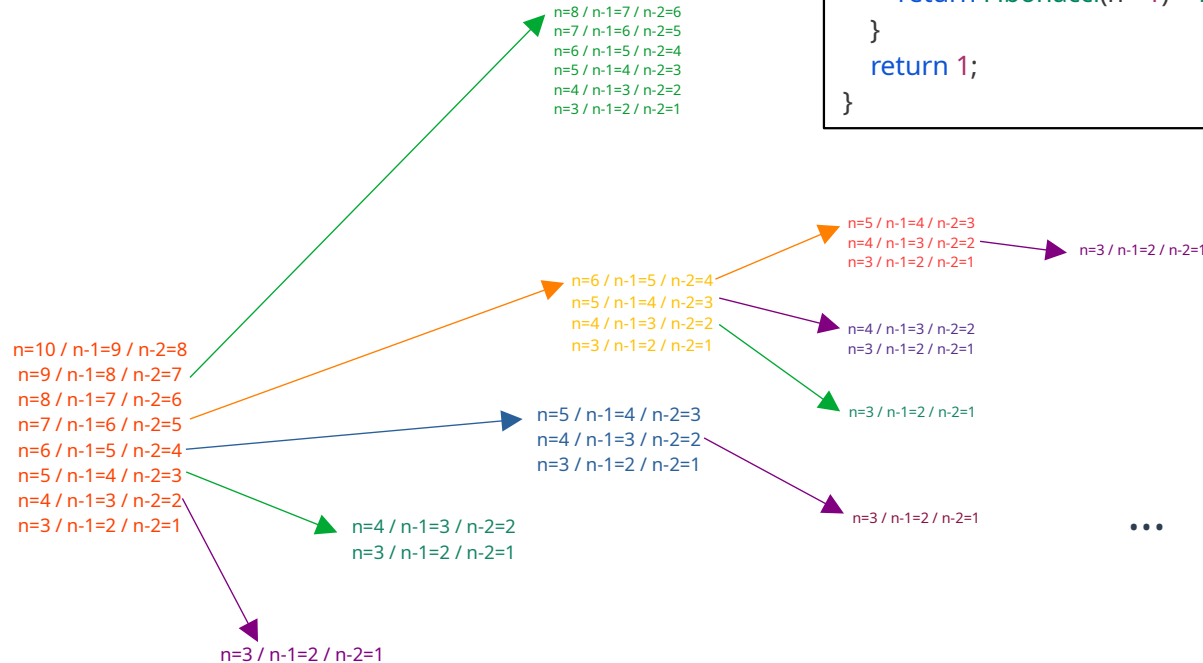


n=10 / n-1=9 / n-2=8
n=9 / n-1=8 / n-2=7
n=8 / n-1=7 / n-2=6
n=7 / n-1=6 / n-2=5
n=6 / n-1=5 / n-2=4
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=6 / n-1=5 / n-2=4
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=7 / n-1=6 / n-2=5
n=6 / n-1=5 / n-2=4
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=8 / n-1=7 / n-2=6
n=7 / n-1=6 / n-2=5
n=6 / n-1=5 / n-2=4
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=6 / n-1=5 / n-2=4
n=5 / n-1=4 / n-2=3
n=4 / n-1=3 / n-2=2
n=3 / n-1=2 / n-2=1
n=3 / n-1=2 / n-2=1
n=4 / n-1=3 / n-2=2

Mémoïsation

- Analysons la sortie de :

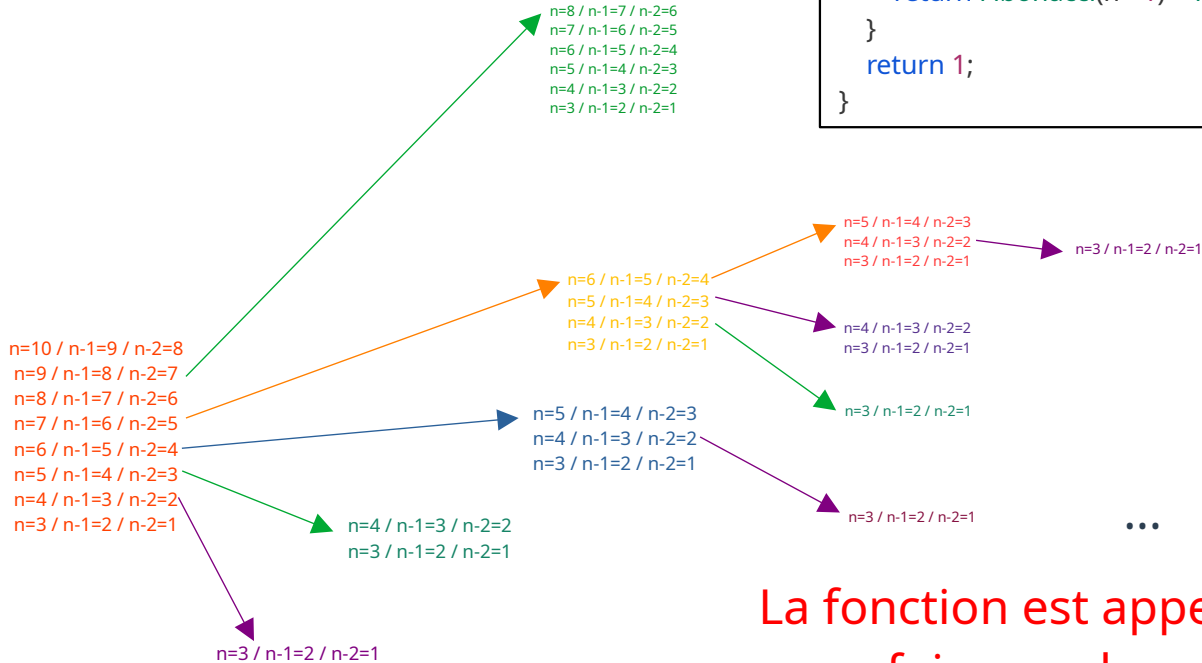
```
static public int Fibonacci(int n) {  
    if (n > 2) {  
        Console.WriteLine("n="+n+" / n-1="+n-1+" / n-2="+n-2);  
        return Fibonacci(n - 1) + Fibonacci(n - 2);  
    }  
    return 1;  
}
```



Mémoïzation

- Analysons la sortie de :

```
static public int Fibonacci(int n) {
    if (n > 2) {
        Console.WriteLine("n="+n+" / n-1="+n-1+" / n-2="+n-2);
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
    return 1;
}
```



La fonction est appelée de nombreuses fois pour les mêmes valeurs

$n=10 / n-1=9 / n-2=8$
 $n=9 / n-1=8 / n-2=7$
 $n=8 / n-1=7 / n-2=6$
 $n=7 / n-1=6 / n-2=5$
 $n=6 / n-1=5 / n-2=4$
 $n=5 / n-1=4 / n-2=3$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=3 / n-1=2 / n-2=1$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=5 / n-1=4 / n-2=3$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=3 / n-1=2 / n-2=1$
 $n=6 / n-1=5 / n-2=4$
 $n=5 / n-1=4 / n-2=3$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=3 / n-1=2 / n-2=1$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=7 / n-1=6 / n-2=5$
 $n=6 / n-1=5 / n-2=4$
 $n=5 / n-1=4 / n-2=3$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=3 / n-1=2 / n-2=1$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=5 / n-1=4 / n-2=3$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=3 / n-1=2 / n-2=1$
 $n=6 / n-1=5 / n-2=4$
 $n=5 / n-1=4 / n-2=3$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$
 $n=3 / n-1=2 / n-2=1$
 $n=4 / n-1=3 / n-2=2$
 $n=3 / n-1=2 / n-2=1$

Mémoïzation

- La solution est de mémoïzer les valeurs déjà calculées dans un dictionnaire. Ainsi ces valeurs ne sont pas recalculées.

```
static public int Fibonacci(int n) {  
    if (n > 2) {  
        Console.WriteLine("n="+n+" / n-1="+n-1+" / n-2="+n-2);  
        return Fibonacci(n - 1) + Fibonacci(n - 2);  
    }  
    return 1;  
}
```



```
static public int Fibonacci_Memoized(int n) {  
    Dictionary<int,int> table = new Dictionary<int, int>();  
    if (!table.ContainsKey(n)) {  
        if (n < 3) // n = 1 or 2  
            table.Add(n, 1);  
        else  
            table.Add(n, Fibonacci_Memoized(n - 1) + Fibonacci_Memoized(n - 2));  
    }  
    return table[n];  
}
```