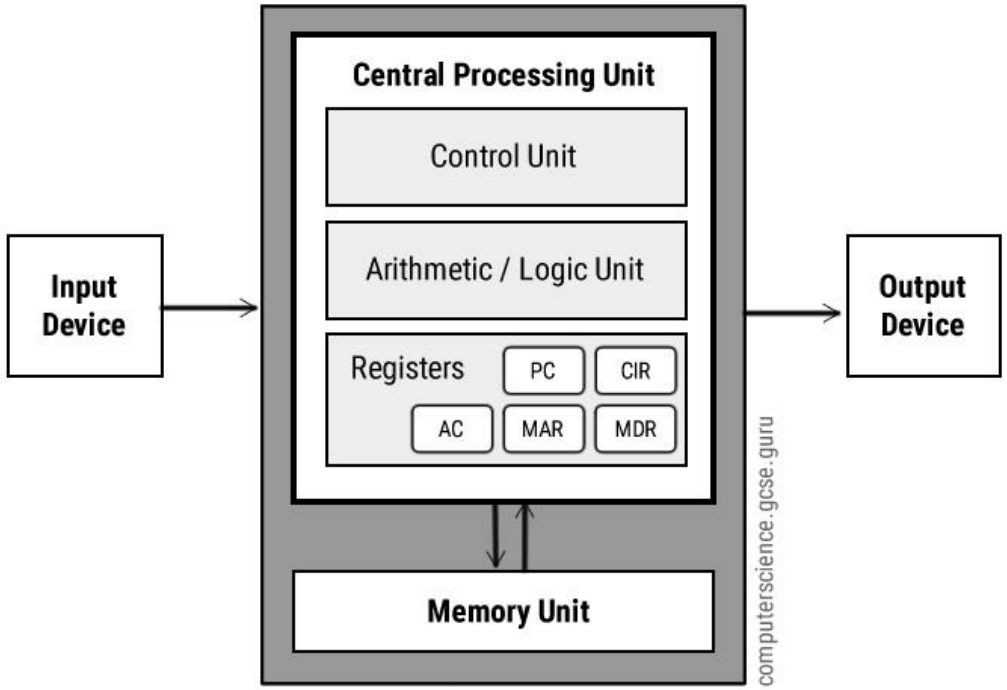
NFP101

Programmation Impérative
vs
Programmation Fonctionnelle

Le paradigme de programmation impérative (1)

- **Basé sur machines à états** (notion de Mémoire) **:**

La Machine de Von Neuman (John Von Neuman 1945)



The control unit controls the operation of the computer's ALU, memory and input/output devices, telling them how to respond to the program instructions it has just read and interpreted from the memory unit.

Arithmetic and Logic Unit (ALU)

The ALU allows arithmetic (add, subtract etc) and logic (AND, OR, NOT etc) operations to be carried out.

MAR	Memory Address Register	Holds the memory location of data that needs to be accessed
MDR	Memory Data Register	Holds data that is being transferred to or from memory
AC	Accumulator	Where intermediate arithmetic and logic results are stored
PC	Program Counter	Contains the address of the next instruction to be executed
CIR	Current Instruction Register	Contains the current instruction during processing

RAM is split into partitions. Each partition consists of an address and its contents (both in binary form).

Le paradigme de programmation impérative (2)

- **Basé sur machines à états** (notion de Mémoire) **:**

La machine de Turing. (Alan Turing 1936)

- Un **ruban infini** divisé en cases consécutives. Chaque case contient 0, 1 ou vide.
- Une **tête de lecture/écriture** qui peut lire et écrire les symboles sur le ruban, et se déplacer vers la gauche ou vers la droite du ruban.
- Un **registre d'état** qui mémorise l'état courant de la machine de Turing. Le nombre d'états possibles est toujours fini, et il existe un état spécial appelé « état de départ » qui est l'état initial de la machine avant son exécution.
- Une **table d'actions** qui indique à la machine quel symbole écrire sur le ruban, comment déplacer la tête de lecture.

POUR symbole lu ET état courant → Déplacement, Valeur à écrire, Nouvel état

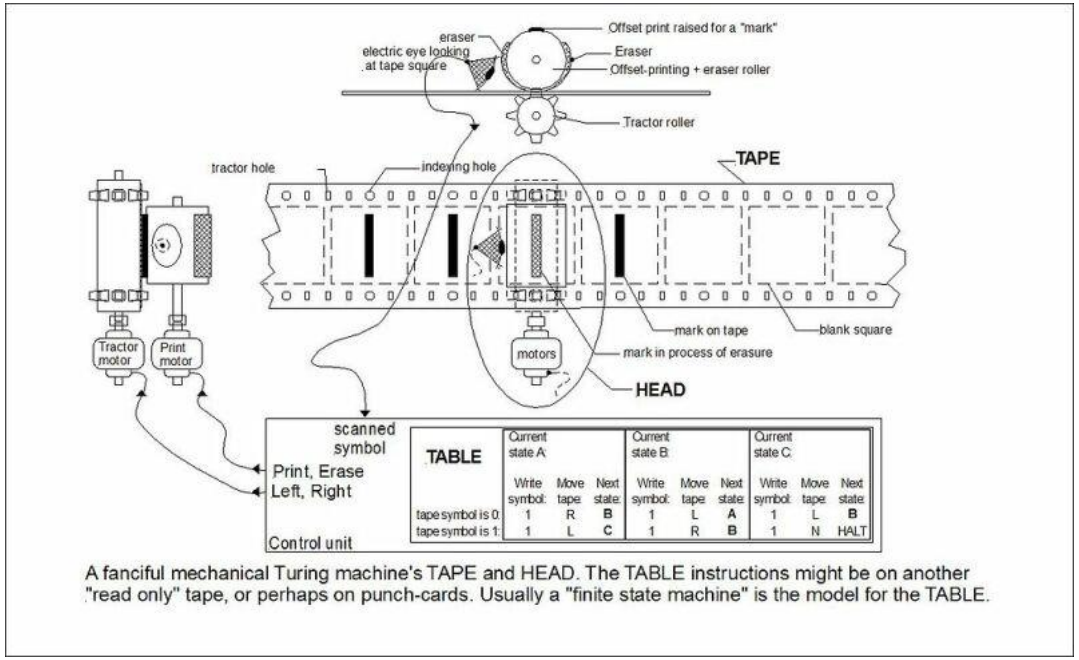
(si aucune action associée pour le symbole lu et l'état courant alors la machine s'arrête)

https://fr.wikipedia.org/wiki/Machine_de_Turing

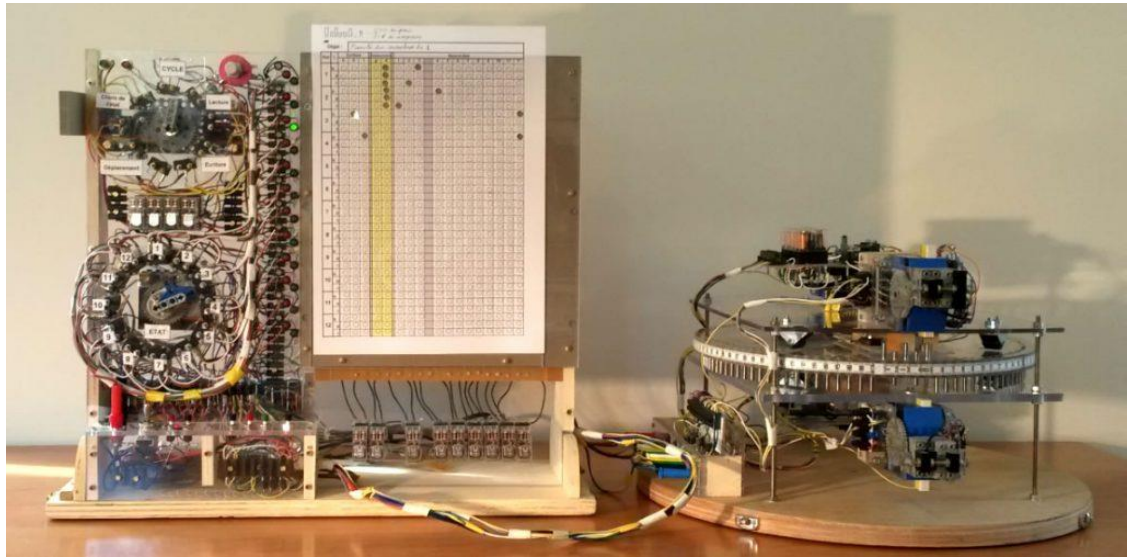
Le paradigme de programmation impérative (2)

- **Basé sur machines à états** (notion de Mémoire) **:**

La machine de Turing. (Alan Turing 1936)



A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.



https://interstices.info/comment-fonctionne-une-machine-de-turing/

Le paradigme de programmation impérative (3)

- **Concepts fondamentaux de la Programmation impérative** :
  - **Séquence / Bloc d'instructions :** liste d'actions élémentaires ordonnées à exécuter par le programme pour changer d'état (inverse de l'exécution parallèle)
  - **Variables :** de portée locales ou globales pour mémoriser des états en mémoire *(Modification / Assignation valeur à variables globales !!! ATTENTION !!!)*

Le paradigme de programmation impérative (4)

- **Concepts fondamentaux de la Programmation impérative** :
  - **Fonctions, Procédures, Sous-Programme :** débranchement provisoire de la séquence d'instructions pour exécuter une séquence de code (*normalement à fort taux d'utilisation*).
  - **Très lié au fonctionnement des processeurs:** codes d'opération, lecture/écriture en mémoire. *« Langage Machine »*

Le paradigme de programmation impérative (5)

- **Types d'instructions utilisées en Programmation impérative** :
  - **Conditions:** permet de n'exécuter un bloc d'instructions que lorsque qu'une condition prédéterminée est vrai (*Si … Alors … Sinon … FSI*)
  - **Branchement sans condition / Saut :** permet de déplacer le lecteur d'instructions d'un programme vers un bloc d'instruction particulier (*Aller à, Goto)*

    Dans quels cas les sauts sont utilisés ?
  - **Bouclage / Itération :** permet d'exécuter un bloc d'instructions un certain nombre de fois ou jusqu'à ce qu'une condition soit réalisée (*Tant Que, Pour, etc.)*

    Comment un bloc de bouclage peut être implémenté ?

Langages de programmation impérative

- Les **Langages Structurés** sous-ensemble des Langages impératifs
  - *Fortran, Cobol, Basic, Pascal, C, Ada* **:** La plupart des langages évolués

- Les **Langages à objets** sont ils un sous-ensemble des Langages impératifs ?

  Oui mais Pas Tous !
  - *C++, Java, Eiffel, SmallTalk, C#*
  - *Common Lisp Object System (CLOS), Scheme*
  - *Python, OCaml, Scala*

Le paradigme fonctionnel (1)

- **Concepts fondamentaux de la Programmation fonctionnelle** :
  - **Fonctions :** relation entre deux ensembles A et B tel que tout élément de A soit en relation avec au plus un élément de B. F(x) avec x ∈ A , F(x) est l'image de A et ∈ B.

    F(x) = x / 2

    F(x1, x2) = x1 + x2
  - **Valeurs :** éléments en entrée des fonctions et produits en sortie de celles-ci
  - **Application de fonctions à des valeurs :** production de nouvelles valeurs.

Le paradigme fonctionnel (2)

- **Concepts fondamentaux de la Programmation fonctionnelle** :
  - **La programmation fonctionnelle** consiste donc en l'évaluation de combinaisons de fonctions mathématiques que l'on appelle expression.

    Fonctions: *F(x) = x / 2; G(x1, x2) = x1 + x2*

    Expression: *F(G(3, 5))*

  - **Les valeurs sont non mutables :** Le changement d'état ou la mutation de données ne peuvent pas être produit par l'évaluation d'une fonction (pas de notion d'affectation)

    « Pas d'effet de Bord » : aucune modification de valeurs autres que celles du contexte

Le paradigme fonctionnel (3)

- **Langage fonctionnel** :
  - Langage dont la syntaxe et les caractéristiques encouragent la programmation fonctionnelle.

    Quelques langages fonctionnels: ***Lisp, Scheme, OCaml***

  - **Dans les langages de programmation :** une fonction est l'élément d'un langage qui reçoit un ensemble d' arguments en entrée (la valeur d'entrée), les traite pour produire une valeur de sortie qu'elle retourne.

    Les fonctions sont présentes dans la plupart des langages actuels même non fonctionnels:

    C, C#, Java etc.

  - **Les fonctions récursives sont au cœur de ce paradigme:**

Exemple: *la fonction factorielle Fact(n) = Si n=0 Alors 1 Sinon n * fact(n-1);*

Le paradigme fonctionnel (4)

- **Propriétés des langages fonctionnels** :
  - **Pureté :**

    Un langage fonctionnel est dit pur si il est sans « effet de bord ». Il n'existe aucune instruction d'affectation.

    Les langages fonctionnels purs sont assez rares: ***Lisp, Scheme (Common Lisp*** *n'est pas pur****)***

  - **Transparence référentielle :**

    « Le résultat d'un programme ne change pas si on remplace une expression par une expression de valeur égale »

    Vrai uniquement si les fonctions n'utilisent que leurs valeurs d'entrées (langages fonctionnels purs)

    PAS DE CONTEXTE GLOBAL NI DE REFERENCES *(facilité de test, pas de problème de concurrence)*

Le paradigme fonctionnel (5)

- **Propriétés des langages fonctionnels** :
  - **Les fonctions sont des valeurs comme les autres :** durant son évaluation une fonction peut faire appel à d'autres fonctions. Ces invocations de **fonctions** peuvent être **statiques** elles ont été définies dans le corps de la fonction ou **dynamiques** si la fonction est passées en tant que valeur d'entrée (Callback).

    Fonctions: *F(x) = x / 2; G(x1, x2) = x1 + x2; H(x, f) = f(G(x, x));*

    Expression: *H(5, F)*
  - Une fonction peut **retourner une fonction** en tant que résultat
  - Une **fonction qui reçoit une fonction** en paramètre ou **retourne une fonction** est dite **: « Fonction d'ordre Supérieur »**
  - **Peut être apparenté à la programmation déclarative :** *SQL*, *Prolog*

Le paradigme fonctionnel (6)

- **Intérêts des langages fonctionnels purs** :
  - **Pas de problèmes de concurrence**
  - **Traitement en parallèle Facilité**
  - **Mémoisation** Optimisation par mise en cache des résultats

Langages fonctionnels (1)

- **Typage dans les langages fonctionnels** :
  - Les types des arguments passés à une fonction doivent être compatible avec ceux attendus par la fonction
  - En cas d'incompatibilité l'évaluation de la fonction ne pourra être effectuée.
  - La vérification du type ou typage des arguments est nécessaire

- **Typage Dynamique :** Le type des arguments est contrôlé au moment de l'évaluation de la fonction

- **Typage Statique :** Le type des arguments est contrôlé à la compilation ou au moment de l'analyse sémantique

Langages fonctionnels (2)

- **Typage dans les langages fonctionnels** :
  - **Typage explicite :** Le type d'un identificateur ou d'une valeur est donné explicitement par l'auteur du code.
    Déclaration avec typage explicite: *String nom*

  - **Inférence de type :** Découverte du type d'un identificateur ou d'une valeur en fonction de son utilisation ou de la fonction qui la produit.
    Même type que la valeur affectée ou du type résultat:

    Expression: *nombreLettre(toMajuscule(nom))*
    - Le type de ***nom*** *devrait être String* car l'argument de ***toMajuscule*** doit être un String .
    - La fonction ***toMajuscule*** retourne une valeur de type *String* donc l'argument passé à ***nombreLettre*** est du type *String*

Exemple Programmation Impérative Vs fonctionnelle

- Calcul de la factorielle d'un nombre:

  « La factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n. La factorielle de 0 est 1 »

- *Programmation Impérative :*

```
Fact(n) {
  fact ← 1;
  Pour i = 1 à n
       fact ← fact * i;
  Fin Pour
  retourner fact;
}
```

Remarques

- Plusieurs affectations avant d'obtenir le résultat
- La modification de **`fact`** à un effet de bord
- La factorielle utilise des instructions impératives:
  - Affectations
  - Itération / bouclage

Exemple Programmation Impérative Vs fonctionnelle

- *Programmation fonctionnelle de la factorielle :*

```
Fact(n) {
  Si n = 0 Alors
     retourner 1;
  Sinon
     retourner n * Fact(n – 1);
  Fsi
}
```

Remarques

- Aucune affectation, pas d'effet de bord
- La description de la factorielle est implémentée
- Utilisation de la récursion (!! mémoire et pile !!)
- D'un point de vue mathématique: ressemble à une suite par récurrence *U0 = 1, U(n) = U(n-1) * n*

Comparaison Programmation Impérative Vs fonctionnelle

	Prog. impérative	Prog. Fonctionnelle
Approche du programmeur	- Algorithme - Gestion modification des états	- Informations en entrées - Transformations requises
Modification de l'état	IMPORTANTE	AUCUNE
Ordre d'exécution	IMPORTANT	PEU IMPORTANT
Contrôle du flux principal	Boucles, Conditions, Appels de Fonctions et de Méthodes	Peu important (respect priorités des fonctions)
Unité de manipulation principale	Instances de modules ou de classes	Fonctions

Conclusion

- **La programmation impérative :**
  - Impose une séquence d'instruction dont l'ordre est important.
  - Est basée sur la mutabilité des variables.

- **La programmation fonctionnelle :**
  - Remet en cause notre façon de résoudre un problème.
  - Peut s'utiliser même dans des langages fortement impératifs.
  - Promeut la composabilité (assemblage de fonctions)
  - Proscrit les effets de bord et les risques de bugs induits.

En fonctionnel on exprime dans le langage ce qu'est une fonction et on laisse décider la machine comment la réaliser.