2 – Programmation fonctionnelle

1 – La programmation fonctionnelle

La programmation fonctionnelle (lire cette page avant de continuer la lecture du cours) est née en même temps que la programmation impérative (Lisp dans les années 1950) mais est longtemps restée marginale, plus académique qu'industrielle (langage ML et dérivés, Haskell).
Cependant, la plupart des langage apparus au 21° siècle sont fortement influencés par un *style fonctionnel*.
Python, par exemple, dans sa version 3.10 (PEP 622) propose le filtrage par motif structurel (à lire pour une meilleur compréhension)qui est un des points fort de la programmation fonctionnelle.
Après d'autres introductions fonctionnelles, c'est le signe que les frontières s'estompent entre paradigmes de programmation.

Des entreprises comme :

- Google (avec Go)
- Mozilla (avec Rust)
- Android (avec Kotlin)
- Apple (avec Swift)
- Whatsapp (avec Erlang)
- Tweeter (avec Scala)

entre autres, ont lancé ou utilisent de nouveaux langages qui ont intégré de nombreux aspects de la programmation fonctionnelle, notamment pour faciliter les traitements de gros flux de données et la programmation concurrente, car ce n'est plus tant la puissance des processeurs qui évolue, mais leur nombre, et la possibilité de travailler avec des logiciels en ligne (SaaS : *software as a service*).

2 – Caractéristiques de la programmation fonctionnelle

1 - Non pas "comment ?" mais "quoi ?"

En programmation fonctionnelle, on déclare ce que l'on veut faire mais pas comment le faire cela va être fait : un programme consiste essentiellement à décrire le rapport qui existe entre les données et les résultats que l'on veut obtenir, plutôt que la séquence de traitements qui mène des unes aux autres.

Exemple :
Quelle est la somme des 100 premiers entiers naturels pairs tels que la carré de la somme d'un de ces nombres additionné à 2 est divisible par 137 ?
Voici comment cela peut être programmé en Haskell :

```
sum $ take 100 $ filter (\x -> (x + 2)^2 'mod' 137 == 0) [0, 2..]
```

Dans cet exemple, on compose des fonctions dont le nom permet de comprendre le rôle.

2 – La transparence référentielle

En programmation fonctionnelle, il n'y a pas d'affectation.
Lorsqu'on écrit *a = 1*, a peut être remplacé par 1 tout au long du programme.

3 – Idempotence

(Mathématiques) (Programmation informatique) Se dit d'une fonction qui a exactement le même résultat qu'on l'appelle une fois ou plusieurs.
Vous obtenez toujours le même résultat pour chaque appel d'une fonction.
Il faut donc éviter au maximum les variables globales et les fonctions à effets secondaires.

4 – Concurrence et/ou parallélisation

L'idempotence et la transparence référentielle permettent aux langages modernes de gérer efficacement la concurrence si importante actuellement.
On peut en effet se libérer de contraintes de chronologie.
Par exemple, si :

res = f1(a, b) + f2(a, c)

on peut effectuer les appels à f1 et f2 à n'importe quel moment car a ne sera jamais modifié.

5 – Pas d'affectation : des liens

C'est une conséquence de la transparence référentielle.
Une expression est liée à une variable.
Si on veut changer cette valeur, il faut créer une autre expression.
On pourrait résumer par : ***Ne modifiez pas ! Créez !***
On peut s'en inspirer en Python aussi.

Ainsi, au lieu de :

```
nom = "James"
nom = nom + "Bond"
```

On préférera :

```
prenom = "James"
nom = "Bond"
appellation = prenom + nom
```

6 – Récursion

Sans affectations ni boucles, la programmation fonctionnelle passe souvent par des définitions récursives de fonctions, c'est-à-dire, des fonctions qui s'appellent elles-mêmes.

7 – Les fonctions sont des objets comme les autres

Les fonctions peuvent prendre d'autres fonctions en argument.
Supposons que nous voulions filtrer des listes selon des critères changeants.
On peut utiliser l'opérateur lambda qui permet de créer des fonctions selon la syntaxe suivante :

lambda arguments: image

par exemple, pour créer "à la volée" la fonction x |-> 2x + 1 :

lambda x: 2*x + 1

On peut le lire ainsi : "Une fonction qui prend x en paramètre et renvoie 2*x + 1."

dans le cas d'un filtre :

```
def filtre(critère):
        return lambda liste: [élément for élément in liste if critère(élément)]
```

Par exemple, pour obtenir les nombres pairs et positifs d'une liste d'entiers :

```
pair_pos = filtre(lambda n: n%2 == 0 and n >= 0)
pair_pos([-4, -3, -2, -1, 0, 1, 2, 3, 4])
[0, 2, 4]
```