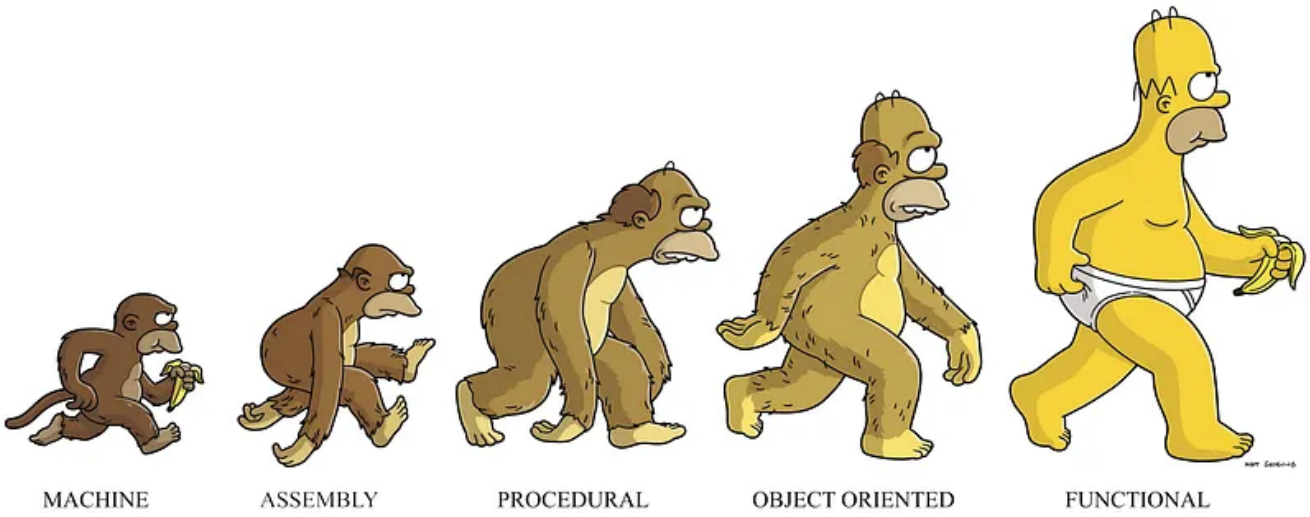
Programmation fonctionnelle



Basée sur l'utilisation des fonctions, à la condition de respecter certains principes.

La prog fonctionnelle n'admet pas le changement d'états et la mutation des données(contrairement à la prog impérative).

Les langages fonctionnels sont ceux vouent un culte à ces principes ou sont basés sur eux :

> Lisp (1958), Scheme (1975), Common Lisp (1984), Haskell (1987), OCaml (1996), Scala (2003), PureScript (2013)...

Les langages de programmation impératifs acceptant le passage de fonctions en paramètres peuvent être utilisés dans le cadre d'une approche fonctionnelle :

> ECMAScript, Java, C#, PHP, Perl, Python, Ruby, Kotlin...

Effets de bord, changement d'états

La programmation fonctionnelle s'affranchit de façon radicale des effets secondaires (ou effets de bord) en interdisant les opérations d'affectation (immutabilité).

Le paradigme fonctionnel n'utilise pas de machine à états pour décrire un programme, mais un emboîtement de fonctions qui agissent comme des “boîtes noires” que l'on peut imbriquer les unes dans les autres.

Chaque boîte possédant plusieurs paramètres en entrée mais une seule sortie, elle ne peut sortir qu'une seule valeur possible pour chaque n-uplet de valeurs présentées en entrée. De cette façon, les fonctions n'introduisent pas d'effets de bord.

Exemples d'effets de bord (side effects) :

- Modifier une variable globale ou la propriété d'un objet
- Ecrire dans la console ou à l'écran
- Ecrire ou lire dans un fichier
- Communiquer avec un réseau
- Communiquer avec un processus externe
- Appeler une fonction qui a des effets de bord

Principes de la pf

Tous les exemples ci-dessous sont traités en javascript (ECMAScript).

Si vous ne connaissez pas Javascript : Learn JS in 5 minutes

1- fonctions pures

Une fonction est dite pure à la double condition :

- Qu'une même série de paramètres d'entrée produise toujours le même résultat
- Qu'elle ne produise pas d'effets de bord (side effects)

a- Utilisation de variables globales

Fonction impure :

```
let pi = 3.14 ;
const calculateArea = (radius) => radius * radius * pi;
calculateArea(10);// returns 314.0, ou autre chose si pi est modifié
```

Suppression de la variable globale **pi** pour la rendre pure :

```
let pi = 3.14;
const calculateArea = (radius, c) => radius * radius * c;
calculateArea(10, pi);// returns 314.0
```

Ou (plus logique), passage de **pi** en constante :

```
const PI = 3.14;
const calculateArea = (radius) => radius * radius * PI;
calculateArea(10);// returns 314.0
```

b- Influence du contexte

Exemples de fonctions impures :

Lecture de fichiers :

```
const charactersCounter = (text) => `Character count: ${text.length}`;

function analyzeFile(filename) {
    let fileContent = open(filename);
    return charactersCounter(fileContent);
}
```

Génération d'aléatoires :

```
function getRandomArbitrary(min, max) {
    return Math.random() * (max - min) + min;
}
```

c- Mutabilité

Exemple de fonction impure :

Modification de globale :

```
let counter = 1;
function increaseCounter(value) {
    counter = value + 1;
}
increaseCounter(counter);
console.log(counter);// 2
```

Rendue pure :

```
let counter = 1;
const increaseCounter = (value) => value + 1;
increaseCounter(counter);// 2
console.log(counter);// 1
```

Avantages :

- Stabilité, prévisibilité
- Testabilité (unitaire)
- Réutilisation (sans effets de bord)

2- Immutabilité

Les données doivent être immutables, c'est-à-dire que leur valeur ne peut changer après initialisation.

Exemple d'itération avec variables mutables :

```
let values = [1, 2, 3, 4, 5];
let sumOfValues = 0;
for (let i = 0; i < values.length; i++) {
    sumOfValues += values[i];
}
sumOfValues // 15
```

Version immutable obtenue par récursion :

```
let list = [1, 2, 3, 4, 5];
let accumulator = 0;
function sum(list, accumulator) {
    if (list.length == 0) {
        return accumulator;
    }
    return sum(list.slice(1), accumulator + list[0]);
}
sum(list, accumulator); // 15
list; // [1, 2, 3, 4, 5]
accumulator; // 0
```

Avantages :

- L'exécution d'un prog n'est plus dépendant des changements d'état

3- Transparence référentielle

Referential transparency

La transparence référentielle est obtenue grâce aux fonctions pures, et elle consiste à permettre le remplacement d'un appel d'une fonction pure, par la valeur qu'elle retourne,sans perturber le fonctionnement du programme.

Exemple :

Soit la fonction carré suivante :

```
const square = (n) => n * n;
```

Pour la même valeur en entrée, cette fonction pure retournera toujours la même valeur :

```
square(2);// 4
square(2);// 4
square(2);// 4
// ...
```

Avec le paramètre 2, la fonction `square` retourne toujours 4. Il est donc possible deremplacer `square(2)` par 4 : Notre fonction est donc référentiellement transparente.

`pure functions + immutable data = referential transparency`

Grâce à ce concept, il est possible de memoïzer (https://en.wikipedia.org/wiki/Memoization) la fonction, pour accroître les performances, en évitant son appel.