Programmation fonctionnelle

Impératif vs. fonctionnel

Parmi les différents paradigmes de programmation existant, certains peuvent se combiner avec n'importe quel autre, comme le paradigme objet, mais d'autres, au contraire, sont en opposition. Ainsi, les paradigmes impératif et fonctionnel ont des approches radicalement opposées.

Dans la **programmation impérative**, un programme est une suite d'instructions modifiant l'état de la mémoire. Ainsi, une variable peut même recevoir plusieurs valeurs durant l'exécution d'un programme. Certaines instructions peuvent changer l'état global du programme. Par exemple `liste.append(0)` modifie l'état de `liste`. On appelle cela un **effet de bord**. C'est utile, par exemple, pour trier une liste, sans avoir à créer à en créer une nouvelle.

Au contraire, la **programmation fonctionnelle** refuse tout effet de bord. Un programme est un enchaînement d'application de fonctions, comme $f(g(x))$. Les affectations ne sont pas utilisées, sauf pour s'implifier l'écriture. L'affectation doit alors être vue comme le fait de donner un nom à un résultat ou à une expression. Ce nom ne sera utilisé pour aucun autre résultat dans la fonction considérée. Cela veut dire qu'il n'est pas possible d'écrire de boucles. On utilise la récursivité à la place.

L'autre grande particularité de la programmation fonctionnelle, comme son nom l'indique, c'est l'utilisation des fonctions. Ces fonctions sont traitées comme des données et peuvent être utilisées aussi bien comme paramètre que comme résultat d'une autre fonction.

Voici un exemple pour la recherche d'un minimum d'une liste Python non vide avec une approche itérative et avec une approche fonctionnelle :

```python
def minimum(liste):
    m = liste[0]
    for v in liste:
        m = min(v, m)
    return m
```

```python
def minimum(liste, i=0):
    if len(liste) == i+1: # On est à la fin
        return liste[i]
    else:
        return min(liste[i], minimum(liste, i+1))
```

On remarque que la version fonctionnelle a l'air plus compliquée. C'est parce que les listes Python sont pensées pour être utilisées de façon itérative. Si on prend les listes chaînées utilisées dans les langages fonctionnels, comme Lisp, la fonction devient plus simple :

```python
def minimum(liste):
    if est_vide(queue(liste)):
        return tete(liste)
    else:
        return min(tete(liste), minimum(queue(liste)))
```

Il faut donc choisir les implémentations des structures en fonction du paradigme utilisé.

C'est qui le patron ?

Alors, au final, qu'est-ce qui est le mieux ? C'est difficile à dire. L'approche fonctionnelle permet d'avoir des programmes plus simples à vérifier, puisqu'il n'y a pas d'effet de bord. Ils sont aussi plus simples à écrire à partir des définitions mathématiques. On décrit ce que doit faire la fonction, pas comment elle doit le faire. On a un niveau d'abstraction supplémentaire. Avec l'approche impérative, on est plus proche de la façon dont fonctionne l'ordinateur. Cela donne un plus grand contrôle, au prix d'une programmation parfois plus complexe.

Les deux approches ont la même "puissance". Il n'y en a pas une qui peut faire plus que l'autre. Il peut juste y avoir une approche plus adaptée en fonction du problème, des contraintes, du programmeur et du langage utilisé.
Certains langages comme OCaml, Scheme ou Lisp sont conçus pour utiliser la programmation fonctionnelle et permettent de tirer la pleine puissance de ce paradigme. Le langage Python, comme beaucoup de langages modernes, permet d'utiliser à la fois des aspects impératifs et fonctionnels, selon les besoins ou les envies.

Bonus : Le lambda-calcul

Le lambda-calcul est considéré comme le fondement les langages fonctionnels. Il a été créé dans les années 1930 par Alonzo Church. Ce formalisme est entièrement basé sur le principe de fonction et d'application d'une fonction. Tous les éléments manipulés en lambda-calcul sont des fonctions.
En mathématiques, on note $f : x \mapsto \mathrm{E}$, E étant une expression dans laquelle apparaît (ou pas) x, pour indiquer la définition d'une fonction. En lambda-calcul, on note $\lambda x.\ \mathrm{E}$. C'est de là que vient la notation `lambda x: E` en Python.
Ce formalisme est basé sur l'évaluation des lambda-termes, ce qui signifie l'application d'une fonction à un élément. On note $f\ x$ l'application de la fonction f à l'élément x. C'est l'équivalent de $f(x)$ en maths. Afin de réduire un lambda-terme, on utilise deux opérations :

- La α-conversion consiste à remplacer le nom d'une variable libre par un autre. Par exemple on peut passer de $\lambda x.\ \mathrm{A}\ x$ en $\lambda y.\ \mathrm{A}\ y$.
- La β-réduction consiste à appliquer une fonction à un lambda-terme et donc à remplacer toutes les occurrences d'une variable libre par le lambda-terme.

Par exemple : $(\lambda f.\ \lambda g.\ (g\ (f\ x)))\ g\ h \xrightarrow{\alpha} (\lambda f.\ \lambda g_0.\ (g_0\ (f\ x)))\ g\ h \xrightarrow{\beta} \lambda g_0.\ (g_0\ (g\ x))\ h \xrightarrow{\beta} h\ (g\ x)$

À l'aide des fonctions, on peut définir les autres types.
Les booléens sont définis ainsi : vrai = $\lambda x.\ \lambda y.\ x$ et faux = $\lambda x.\ \lambda y.\ y$.
Cette définition peut sembler étonnante mais elle permet d'écrire facilement une instruction conditionnelle, "si T alors A sinon B" de la manière suivante : $\lambda b.\ \lambda\mathrm{A}.\ \lambda\mathrm{B}.\ b$ A B. Ainsi, si b est vrai, alors on obtiendra A. Sinon on obtiendra B.
De la même manière, on peut définir le "et" booléen : et = $\lambda b_1.\ \lambda b_2.\ b_1\ b_2$ faux.

et faux faux	et vrai faux	et vrai vrai
$(\lambda b_1.\ \lambda b_2.\ b_1\ b_2$ faux) faux faux	$(\lambda b_1.\ \lambda b_2.\ b_1\ b_2$ faux) vrai faux	$(\lambda b_1.\ \lambda b_2.\ b_1\ b_2$ faux) vrai vrai
$(\lambda b_2.$ faux b_2 faux) faux	$(\lambda b_2.$ vrai b_2 faux) faux	$(\lambda b_2.$ vrai b_2 faux) vrai
faux faux faux	vrai faux faux	vrai vrai faux
$(\lambda x.\ \lambda y.\ y)$ faux faux	$(\lambda x.\ \lambda y.\ x)$ faux faux	$(\lambda x.\ \lambda y.\ x)$ vrai faux
$(\lambda y.\ y)$ faux	$(\lambda y.$ faux) faux	$(\lambda x.\ \lambda y.$ vrai) faux
faux	faux	vrai

Enfin, le lambda-calcul a été le premier langage à permettre la définir les fonctions récursives. Cela se fait avec le combinateur de point fixe $\mathrm{Y} = \lambda f.\ (\lambda x.\ (f\ (x\ x)))\ (\lambda x.\ (f\ (x\ x)))$.
On a la réduction suivante :

$$\mathrm{Y}\ g \xrightarrow{\beta} (\lambda x.\ (g\ (x\ x)))\ (\lambda x.\ (g\ (x\ x))) \xrightarrow{\beta} g\ ((\lambda x.\ (g\ (x\ x)))\ (\lambda x.\ (g\ (x\ x)))) \rightarrow g\ (\mathrm{Y}\ g)$$

Ainsi, on peut appliquer un nombre arbitraire de fois la fonction g. On peut par exemple définir la taille d'une liste de la manière suivante :

$$\texttt{taille} = \mathrm{Y}\ (\lambda f.\ \lambda l.\ (\texttt{est\_vide}\ l)\ \texttt{zero}\ (\texttt{succ}\ (f\ (\texttt{queue}\ l))))$$