NSI Terminale - Programmation

Paradigmes - Fonctionnel

Florian Kauder

2020/05/04

Le Paradigme fonctionnel

Faire des fonctions, c'est le quotidien de beaucoup de développeurs. Nous sommes tous là, à construire ces fameuses « briques magiques » qui composent les logiciels de tous les jours. Mais une brique mal conçue fragilise tout l'édifice, et je suis persuadé que vous n'avez pas envie de voir votre bâtisse finir en champ de ruines.

La conception de ces briques correspond à un ensemble de techniques rassemblées sous le nom de **programmation fonctionnelle**. Ce paradigme de programmation, utilisé par OCaml, Erlang et autres F#, a été très fortement délaissé par les professionnels durant de très longues années. Mais cette ère est révolue : la **programmation fonctionnelle est sûrement disponible dans votre langage favori** ! Comment ça marche ? Qu'est-ce que cela implique ? Pourquoi devriez-vous en parler à toute votre équipe et l'adopter rapidement ? Tout ceci, c'est ce que je propose de vous expliquer dans cet article.

Vers l'infini et les fonctions

La première chose que vous devez vous dire est que, finalement, n'importe quel langage doit être fonctionnel : nous faisons des fonctions dans quasiment tous les langages, et ce depuis les débuts de l'informatique. En fait, qualifier un langage de *fonctionnel* implique qu'il réponde correctement aux définitions suivantes :

- il est possible d'utiliser une fonction comme paramètre d'une autre fonction.
- il est possible de retourner une fonction comme résultat d'une autre fonction.
- il est possible d'assigner une fonction à une variable.
- il est possible de stocker une fonction dans une structure de données.

Si ces conditions sont respectées, on dit alors que les fonctions dans ce langage sont des **fonctions de première classe**, et que le langage est **fonctionnel**.

Ces définitions nous permettent donc, par exemple, d'appliquer n'importe quelle fonction sur tous les éléments d'un tableau. Pour cet exemple, nous souhaitons mettre au carré tous les nombres d'un tableau d'entier. De façon classique, la fonction ressemblerait à ça :

```
fonction tableauAuCarré(t: tableau d'entiers)
    pour i de 0 à taille(t)
        t[i] = t[i] * t[i]
```

Le problème, c'est que si on veut maintenant mettre au cube les nombres du tableau, on est obligé de créer une nouvelle fonction. Et on sera obligé de créer une nouvelle fonction pour chaque nouveau traitement que l'on veut faire sur le tableau.

La programmation fonctionnelle permet de corriger ce problème : nous pouvons créer une fonction qui itère sur un tableau, et appliquer cette fonction sur tous les éléments du tableau. Cette technique consistant à amener un traitement en paramètre pour pouvoir être changé s'appelle la **généralisation**.

```
fonction appliquerFonctionSurTableau(t : tableau d'entiers, fn : fonction(entier))
    pour i de 0 à taille(t)
        t[i] = fn(t[i])

fonction miseAuCarré(n : entier)
    retourner n * n

appliquerFonctionSurTableau([0, 1, 2, 3], miseAuCarré)
=> [0, 1, 4, 9]
```

Plus de productivité, moins de lignes à coder

Ce qui est génial avec la programmation fonctionnelle, c'est qu'elle est livrée avec plein de fonctions permettant d'appliquer des fonctions. Je vais me concentrer sur les fonctions agissant sur des tableaux, car je trouve qu'il s'agit des exemples les plus intéressants d'application, en plus d'être les plus simples. Ces fonctions que je présente sont (normalement) déjà inclues dans les bibliothèques natives de vos langages. Je donne simplement les algorithmes pour mieux comprendre ce qu'elles font.

map()

Si vous avez déjà entendu parler du *pattern Map/Reduce* souvent nommé quand on parlait du *Big Data*, et bien c'est de ce fameux *map* que nous parlons ici. Il s'agit d'une des opérations de base sur un tableau, permettant de créer un nouveau tableau, auquel on applique une fonction passée en paramètre. Avec l'exemple de tout à l'heure :

```
fonction map(t : tableau d'entiers, fn : fonction(entier))
    res : tableau d'entier de taille=taille(t)

    pour i de 0 à taille(t)
        res[i] = fn(t[i])

    retourner res

map([0, 1, 2, 3], miseAuCarré)
=> [0, 1, 4, 9]
```

Oui, c'est la même chose (comme par hasard), à **une différence près** : *map* crée un nouveau tableau, ne modifiant donc pas l'ancien.

reduce()

Toujours dans le fameux couple *Map/Reduce*, nous nous occupons maintenant du second. *reduce* est une fonction formidable qui permet de **réduire** un tableau à une donnée. Voici un exemple avant d'aller plus loin :

```
fonction reduce(t : tableau d'entiers, fn : fonction(?, entier))
    resultat : ?

    pour i de 0 à taille(t)
        resultat = fn(resultat, t[i])
```

```
    retourner res

fonction somme(précédentRésultat : entier, élémentActuel : entier)
    retourner précédentRésultat + élémentActuel

reduce([0, 1, 2, 3], somme)
=> 6
```

La fonction *reduce* va appliquer une fonction à tous les éléments du tableau, avec comme paramètre le résultat calculé jusqu'à l'élément actuel (**l'accumulateur**) et l'élément actuel. Le but de la fonction est d'obtenir un élément simple, comme la somme, qui nécessite d'utiliser tous les éléments du tableau. Il s'agit de la même chose que quand vous calculez vous-même la somme d'un tableau : prenons le tableau [4, 1, 6]. Pour faire la somme, nous suivons la démarche suivante :

- Vous retenez 4.
- Vous ajoutez 1, ce qui fait 5. Vous retenez 5.
- Vous ajoutez 6, ce qui fait 11. Vous retenez 11.

Ce comportement est le même que celui de *reduce* : nous retenons quelque chose, nous effectuons un traitement entre le résultat retenu et le nouvel élément, puis nous le retenons, et ainsi de suite.

Note : j'ai volontairement mis **?** à la place du type de l'accumulateur, car il est normalement du type que l'on souhaite.

filter()

Très souvent, vous avez besoin de conserver uniquement les éléments répondant à une certaine condition. Il s'agit du principe même de la fonction *filter*, qui va **filtrer** les éléments d'un tableau pour conserver uniquement ceux répondant à une condition, que l'on nomme le **postulat**. Au final, cette fonction nous permet de récupérer un nouveau tableau avec uniquement les éléments voulus.

```
fonction filter(t : tableau d'entiers, fn : fonction(entier))
    resultat : tableau de taille dynamique

    pour i de 0 à taille(t)
        si (fn(t[i]) est vrai)
            alors resultat.ajouter(t[i])

    retourner resultat

fonction positif(n: entier)
    retourner n >= 0

filter([0, -4, 6, -2, 1], positif)
=> [0, 4, 1]
```

... et ce ne sont pas les seules.

Vous pouvez trouver d'autres fonctions comme :

- `sort(tableau, fonction(e1, e2))` : permet de trier un tableau (une fonction comparant les éléments un à un est passée le long du tableau).

- `find(tableau, fonction(e))` : permet de chercher un élément dans un tableau (la fonction donnée en paramètre renvoie *vrai* quand l'élément est le bon).
- `some(tableau, fonction(e))` : renvoie *vrai* si un élément existe dans un tableau (même fonctionnement que **find**).
- `forEach(tableau, fonction(e))` : applique la fonction à tous les éléments du tableau.
- etc.

Anonymes et chaînables

Jusqu'ici, nous déclarions chacune des fonctions que nous souhaitions utiliser. Mais il est possible de définir une fonction *à la volée* sans lui donner de nom ou autre : ces fonctions sont dites **anonymes** (on parle aussi de **lambda-fonctions**, en référence au lambda-calcul). Cette fonctionnalité va nous permettre d'écrire des fonctions très rapidement, le plus souvent avec **une syntaxe spécifique et minimaliste** (mais cependant propre à chaque langage).

```
map([0, 1, 2, 3], (n: entier) => n * n)
=> [0, 1, 4, 9]
```

Très souvent, dans les notations proposées par les langages, le *return* est implicite, pour justement simplifier au maximum les fonctions.

Autre concept intéressant : dans les langages objets où les tableaux peuvent avoir des propriétés, il est possible de faire un **chaînage** avec plusieurs fonctions. Chaque fonction renvoyant un tableau, nous pouvons enchaîner les appels fonctionnels pour obtenir des algorithmes très puissants en peu de lignes.

Par exemple : "ajouter tous les nombres qui suivent la chaîne `"X:"` dans le tableau

```
1  let array: tableau de chaînes de caractères = ["X:32", "X:41", "Y:28", "Y:24", "X:22"];
2
3  array.filter((element: chaîne de caractères) => element.exists("X:"))
4       .map((element: chaîne de caractères) => element.substring(taille("X:")).versEntier())
5       .reduce((accumulator: entier, element: entier) => accumulator + element)
```

- Ligne 1: `array = ["X:32", "X:41", "Y:28", "Y:24", "X:22"]`
- Ligne 3: ne garde que ceux qui contiennent "`"X:`". On a donc `["X:32", "X:41", "X:22"]`
- Ligne 4: ne garde que ce qui est après `"X:"` et le transforme en entier : `[32, 41, 22]`
- Ligne 5: on accumule ces entiers : `32 + 41 + 22` et on a `95`

Impact sur la qualité de code

Posons directement les choses : la programmation fonctionnelle est bénéfique pour la qualité d'un code, mais seulement si elle respecte certaines règles. Par exemple :

- 1 – toute fonction complexe (ici, dont le nombre de lignes est supérieur ou égal à 2) doit être déclarée et nommée.
- 2 – si une fonction contient un enchaînement de méthodes (comme au-dessus dans le *map*), il est préférable de déclarer et nommer la fonction.
- 3 – il est préférable d'éviter les effets de bord sur les fonctions utilisées dans les *map*, *reduce*, etc., c'est-à-dire d'éviter de modifier la structure et les données manipulées par les fonctions (on parle alors de **fonctions pures**).
- 4 – bien évidemment, un chaînage trop long (à partir de 5 à 6 fonctions) doit être divisé.

Ces différentes règles permettent de tirer le meilleur de la programmation fonctionnelle et de conserver un code propre.

Lisibilité

L'un des principaux avantages de la programmation fonctionnelle est bien évidemment la lisibilité. Elle nécessite un prérequis (la connaissance des *fonctions de base*), mais une fois cette connaissance acquise de toute l'équipe, elle permet d'**avoir un code plus simple et plus court**. On va notamment réduire considérablement le nombre de boucles dans le programme (remplacées par des appels de fonctions), ainsi qu'ajouter la possibilité de traiter les éléments d'un tableau directement sans passer par la syntaxe *array[i]*.

Pour remplacer l'imbrication des boucles dans des boucles, on peut envisager de :

- créer une fonction permettant de traiter notre structure sur toutes les dimensions qu'elle nécessite (nécessite une fonction par type d'itération, donc rapidement lourd).
- déclarer plusieurs fonctions différentes (une par boucle), et chaque fonction va appeler une autre fonction au moment où la boucle est nécessaire (permet de séparer l'enchaînement de boucles, mais peut rendre la lecture un peu plus compliquée).

Maintenabilité – Evolutivité

Une des conséquences directes d'une meilleure lisibilité est d'avoir un code beaucoup plus maintenable. En fait, cette maintenabilité vient aussi du fait qu'il est possible de déboguer chaque *étape* d'un **chaînage** assez facilement, contrairement à une écriture plus impérative dont les différents traitements seraient mélangés. La séparation du traitement et de l'itération permet aussi de **faciliter l'identification des bogues**, en proposant notamment de tester la fonction de traitement indépendamment du reste.

L'évolutivité est aussi plutôt avantagée : il est beaucoup plus simple de rajouter une nouvelle étape dans un **chaînage** que dans une boucle classique. Il est cependant nécessaire de penser à la règle (4) sur le nombre de fonctions d'un chaînage qui peut entraîner un peu de refactorisation de code.

Testabilité

Sur ce point, la programmation fonctionnelle est encore plus intéressante : chacun de nos traitements importants peut être transformé en une suite de fonctions. Il devient donc possible de **tester unitairement chacune de ces fonctions complètement indépendamment**.

Dans le cas où vous travaillez avec des **fonctions pures** (sans aucun effet de bord), vous avez alors des unités extrêmement simples à tester. Une **fonction pure** va, par définition, **retourner toujours le même résultat pour les mêmes paramètres d'entrée**. Si le test passe donc avec certains paramètres d'entrée, vous avez alors **la garantie que votre fonction marchera toujours avec ces paramètres d'entrée**.

Exemples concrets

Pour illustrer tout ce que nous avons vu jusqu'ici, nous allons nous soumettre à un petit exercice. Nous possédons un fichier de données donnant les températures maximales dans plusieurs villes dans le monde (toutes les informations contenues dans le fichier sont factices). Le but va être **d'extraire la température maximale parmi les villes françaises.** Le fichier de données est le suivant :

```
{
    "date": "26-07-2017",
    "data": [
       {"country": "FR", "city": "Lyon", "temperature": "25.3C"},
       {"country": "USA", "city": "Chicago", "temperature": "36.8C"},
```

```
        {"country": "DE", "city": "Berlin", "temperature": "21.4C"},
        {"country": "FR", "city": "Paris", "temperature": "27.6C"},
        {"country": "IT", "city": "Rome", "temperature": "31.7C"},
        {"country": "ES", "city": "Madrid", "temperature": "35.9C"},
        {"country": "DE", "city": "Munich", "temperature": "26.1C"},
        {"country": "FR", "city": "Bordeaux", "temperature": "27.4C"},
        {"country": "EN", "city": "Londres", "temperature": "25.1C"},
        {"country": "USA", "city": "Los Angeles", "temperature": "22C"},
        {"country": "EN", "city": "Oxford", "temperature": "24.1C"},
        {"country": "USA", "city": "New York", "temperature": "31.2C"}
    ]
}
```

J'écrirai en Python (*l'article d'origine présente aussi d'autres langages*) la fonction en paradigme impératif et la fonction en paradigme fonctionnel pour pouvoir voir la différence entre les deux.

La programmation fonctionnelle marche assez bien en Python. Le langage implémentant le mot-clé **lambda**, il est possible de définir des fonctions anonymes de la façon suivante :

```
lambda a, b: a > b
```

Les fonctions de base des tableaux **n'étant pas chaînables**, il est possible d'émuler ce comportement **en utilisant le résultat d'une fonction comme paramètre de la suivante**. Le résultat est que **le sens de la lecture est inversé** : nous commençons la lecture par la fonction la moins imbriquée, donc la dernière à être exécutée, ce qui peut être troublant lors des premières rencontres avec cette syntaxe.

Néanmoins, cette syntaxe reste beaucoup plus courte que la précédente, et ce tout en conservant une lecture correcte de l'algorithme.

```
import json

with open('./data.json') as dataFile:
    data = json.load(dataFile)["data"]

def parseTemperature(temperatureStr):
    return float(temperatureStr[:temperatureStr.index("C")])

def getMaximum(a, b):
    return a if a > b else b

def computeFranceMaxImp():
    max = 0

    for entry in data:
        if entry["country"] == "FR":
            temperature = parseTemperature(entry["temperature"])

            if temperature > max:
                max = temperature

    return max

def computeFranceMaxFn():
```

```
    return reduce(getMaximum,
    map(lambda e: parseTemperature(e["temperature"]),
    filter(lambda e: e["country"] == "FR", data)))


print computeFranceMaxImp()
print computeFranceMaxFn()
```

Si un engouement se crée actuellement autour de la programmation fonctionnelle, il se justifie par ses différents avantages que l'on peut en tirer, **et ce très rapidement**. Ce paradigme sort de plus en plus de son ancienne bulle académique **pour venir s'installer confortablement dans le monde professionnel**, où les développeurs sont ravis d'utiliser ses principes.

Note finale de votre enseignant

Le paradigme fonctionnel est enseigné dans toutes les universités et école d'informatique. Il n'est pas plus difficile que le paradigme objet ou impératif.

Depuis une quinzaine d'année et l'émergence d'enjeux liés à la massification des données il a fait une entrée remarquée dans les grands groupes informatiques.

Citons Twitter, Netflix (scala) et Facebook (OCaml) qui l'utilisent pour accomplir des travaux sur les données considérables qu'ils génèrent.