Functional programming

Introduction to functional programming

Functional programming is a programming paradigm (this is, a way of implementing programs) that focuses on the evaluation of mathematical functions. It is based on *lambda calculus*, a system born in 1930 to evaluate function definition, application and recursion.

Functional paradigm is declarative, this is, code is made with declarations instead of sentences or statements. In other words, we tell the program how things are, instead of how to solve a problem. On the opposite side we have imperative languages, which are based on commands in the source code, such as assignments, that tell the program how to solve the problem.

1. Some functional programming features

Functional programming relies on some fundamental concepts:

- **Referential transparency**: the output of a function depends only on its arguments, so that if we call it many times with the same arguments, we will always get the same result. So, functional programming has no side effects that can modify something out of the function
- **Data immutability**: in order to make sure that no side effects will be produced when calling functions, one of the most important rules of functional programming paradigm is that data can't be mutable.
- **Function composition**: functional languages treat functions as data, so we can apply function composition, this is, we can chain the output of a function with the input of next function.
- **First order functions**: these functions are higher level functions, which allow other functions as parameters. They are very popular in languages such as Javascript, when we can define *callbacks* to respond to some events or asynchronous tasks, but they are also an important aspect of functional languages.

Imperative languages can produce side effects in external elements. They also have functions, just like functional programming has, but in this case functions are not mathematical definitions, but a group of sentences that can be called from different parts of the program. Taking into account possible side effects, we might have different results when calling a function many times with the same arguments. Let's have a look at the following code written in C:

```
int externalValue = 1;
int aFunction(int param)
{
    externalValue++;
    return externalValue + param;
}
```

If we call the function as `aFunction(1)`, it will return 3. But if we call it again with the same arguments (`aFunction(1)`) it will return 4... and so on. This kind of situation is forbidden in a functional paradigm, although some functional languages let us commit this "crime".

2. Functional languages

If we are looking for a functional language, we must distinguish between:

- **Pure** functional languages, this is, languages that were created to follow this paradigm, and no other. In this group we can find languages like **Haskell** or **Miranda**. The first one is used by Facebook to deal with big data and analysis.
- **Hybrid** languages, this is, languages that accept several paradigms, although they do well with functional approaches. In this group we can talk about **Scala**, **Clojure**, or **Lisp**.
- Other languages, that were initially imperative, but have modified their syntax and structure to accept some functional features. This is the case of some of the most important languages of these years, such as **Java**, **C#**, **Javascript**...

3. An introductory example

Just to have a first experience with the functional paradigm, let's see how to solve a typical problem. We are going to use a popular language, such as Java, to illustrate it. We have a list of `Person` objects, each one with its own name and age. If we want to get the people that are adults (i.e. their age is greater or equal than 18), we would do it this way with traditional, imperative programming:

```
List<Person> adultPeople = new ArrayList<>();
for (int i = 0; i < people.size(); i++)
{
    if (people.get(i).getAge() >= 18)
        adultPeople.add(people.get(i));
}
```

What we do is explore the original list, and add to a new one every person older than 18. But, if we take advantage of some of the new features provided with Java 8 regarding functional programming, we can solve the same problem like this:

```
List<Person> adultPeople = people.stream()
                                 .filter(p -> p.getAge() >= 18)
                                 .collect(Collectors.toList());
```

As we can see, code is more compact, less error-prone and (once we get used to the syntax), more understandable. You can also see how function composition works: the output of `stream` method is the input for `filter` method, and the output of this method is the input for `collect`.

4. Functional programming in Java

Regarding Java, it has added some functional programming features, specially since version 8, that let us operate with (almost) immutable data, and chain operations through function composition. We are mainly talking about lambda expressions and streams.