Updated: 2021-09-11

2.1 What's the big deal? About functional programming?

Northeastern University

What’s the big deal about functional programming?

What does functional programming even mean?

What does “functional programming” mean?

- "What's in a name? That which we call a rose
  By any other name would smell as sweet.”

  Juliet from *Romeo and Juliet*, William Shakespeare.
- It’s not just about functions, despite the name.
- There used to be a type of programming called “symbolic programming” (You could argue that Perl is symbolic programming):
  - The tokens in such a language aren’t just thrown away as code is generated: the tokens are a living part of the language.
- To me, that’s the essence of “functional programming.”

Let's just look at one tiny (but significant) detail

- In a language like Java or C, an assignment is just what it sounds like: a value is "assigned" to a variable:

  ```
  double x = Math.PI;
  ```

  - This "statement" combines a "declaration" of *x* and an assignment (in Java, you can split those two parts up if you like).
  - The constant value *pi* has been given to *x* and that will be its value until changed.
  - In Java, if you never want to allow x's value to be changed, then you write *final* in front of the statement:

    ```
    final double x = Math.PI;
    ```

- But, even in the second form, you would never really say that *x* and *Math.PI* were actually the same thing, would you? It's essentially a <u>temporary</u> relationship.

But in Scala, things are a bit different:

- In Scala, we can write the following:

```
val x = math.Pi
```

  - This declaration essentially says that, from here on, we treat *x* and *math.Pi* identically. We could, at any point, choose either one. In other words, *x* has simply become an *alias* for *math.Pi*.
- And here's the really important point:
  - At any future time, we can replace *x* by *math.Pi* (or we can replace *math.Pi* with *x*) and our program will be absolutely identical: it will in every respect behave the same!
- So, what's the point of this alias?

Declarations as aliases

- Here are some of the reasons why this is a "good thing:"
  - Let's say that an important piece of information in our program is the circumference of a circle.
  - And let's say that we write:

```
val circumference = d * math.Pi
```

  - … and then, in several places, we write *circumference* instead of *d * math.Pi*.

1. We have avoided repeating ourselves (the DRY principle of programming);
2. We have identified the concept of circumference with an appropriate name;
3. We've (very slightly) improved the efficiency of our program;
4. We've (hopefully) made our program more understandable and, at the same time, more *mathematical*.

What about functions, though?

- Let’s say you have developed a matrix manipulation framework that is designed to run on a cluster of 1000 nodes.
- It can multiply matrices together, transpose them, invert them, transform their elements from one domain to another, all that stuff.
- You will need a driver which allows the programmer to set up the matrices and define operations such as multiplication, etc.
- All of those are easy to implement except *transform*. How exactly are you going to let your programmer specify what operation should be performed on an element of a matrix *when he doesn’t have direct access to an element of a matrix?*
- This is a case of “pushdown” logic. We want the system to push our function down into the depths of its data structures.
- Therefore, we need to be able to treat functions just like other objects.

Functions are first-class objects

- If functions are objects, just like ”Hello, World!” or *math.Pi*, then we can operate on functions and apply them to (sets of) parameters.
- How can we operate on functions, though?
- Operations on functions are not well-known obvious stuff like “*”, “+”, etc. which are suitable for arithmetic objects;
- Instead, we can define our own functions and *compose* them.
- Why do we need to compose functions?
    - Because composition is the chief mechanism that allows ***re-use***.
    - And code re-use is the chief contributor to writing robust, efficient code with consistent behavior.

Functional composition and higher-order functions

- A higher order function is a function, at least one of whose parameters is a function (where *xs* is a collection, *f* is a function).
  - `xs map f`
  - `xs reduce f`

- Functional composition is where the result of a higher order function is itself a function.
  - `g andThen h`

But how do we define the functions we need?

- Lambda calculus (lambdas for short).
- A lambda is an anonymous function (or “functional literal”) which defines how its parameters “bound variables” are transformed into its result.
- A lambda also “closes” on any free variables in scope.
- You can find lambdas in the following languages…
  - Java8 (actually there were special cases of lambdas all the way back in Java 1);
  - Python
  - Scala
  - Haskell
  - Pharo
  - All functional programming languages

But that’s not all…

- There are many other aspects of functional programming, many of which we can’t find in Java8 or Python:
    - Lazy (deferred) evaluation;
    - Type inference and shape preservation;
    - Pattern-matching;
    - Referential transparency:
        - Immutability by default;
        - Pure functions (lack of side-effects);
    - Tail recursion;
    - Tuples;
    - Monads, etc.
    - Higher-kinded types.

More later…

See 2.3 Functional Programming in Scala