Implementing operators in Elixir

Fernando Karpinski

2021

Abstract

In this article, we will implement operators in the Elixir programming language. More specifically, we shall do that for operators that, at this time of writing, are recognized by the language, but are not used by it.

Introduction

The implementation of operators in Elixir can aid in the writing of programs whose syntax is not focused solely on functions. Even though Elixir is a functional programming language, and, thus, the usage of functions in it is something fundamental, the use of operators can help us in some cases. For example, some people might argue that it might be easier to understand the expression `a + b` than the expression `Kernel.+(a, b)`. Indeed, we are, for example, used to the usage of binary operators in mathematics, in such a way that it is common for us to write the operator between the values upon which it will be applied.

Complex numbers

Let us implement some operators for the manipulation of complex numbers. A complex number n contains two parts, the real part, a, and the imaginary part, b:

$$n = a + bi$$

We can define a *struct* for representing a complex number:

```
defmodule ComplexNumber do
    defstruct [:real_part, :imaginary_part]
end
```

Creating complex numbers with the first operator

To create a complex a complex number, we could use the syntax `%ComplexNumber{real_part:r, imaginary_part:i}`, or even delegate such a

work to a function. However, here we are going to take the approach of implementing an operator for that, in this same module:

```
defmodule ComplexNumber do
    defstruct [:real_part, :imaginary_part]

    def r <|> i do
        %ComplexNumber{real_part: r, imaginary_part: i}
    end
end
```

Notice that we use `def`, and that we write the operator and the values upon which it will be applied already in the order they will be used in practice.

Addition

For adding two complex numbers $c1 = m + ni$ and $c2 = x + yi$, we use the following methodology:

$$c1 + c2 = m + ni + x + yi = (m + x) + (n + y)i$$

Let's implement the operator `+++` for said addition:

```
defmodule ComplexNumber do
    defstruct [:real_part, :imaginary_part]

    def r <|> i do
        %ComplexNumber{real_part: r, imaginary_part: i}
    end

    def %ComplexNumber{real_part: r_1, imaginary_part: i_1} +++
        %ComplexNumber{real_part: r_2, imaginary_part: i_2} do
        (r_1 + r_2) <|> (i_1 + i_2)
    end
end
```

Using *pattern matching*, we extract the real and imaginary parts of each complex number and compute the new number, already using the operator for creation we implemented before. We use the parentheses to guarantee that the parts of the new complex number will be calculated before the creation of the data structure itself.

Multiplication

The multiplication of two complex numbers $c1 = m + ni$ and $c2 = x + yi$ can be done as described next:

$$c1 \cdot c2 = (m + ni)(x + yi) = (mx) + (my + nx)i + (ny)i^2 = \\ (mx) + (my + nx)i + (ny)(-1) = (mx - ny) + (my + nx)i$$

Let's designate this implementation to the `&&&` operator:

```
defmodule ComplexNumber do
    defstruct [:real_part, :imaginary_part]

    def r <|> i do
        %ComplexNumber{real_part: r, imaginary_part: i}
    end

    def %ComplexNumber{real_part: r_1, imaginary_part: i_1} +++
        %ComplexNumber{real_part: r_2, imaginary_part: i_2} do
        (r_1 + r_2) <|> (i_1 + i_2)
    end

    def %ComplexNumber{real_part: r_1, imaginary_part: i_1} &&&
        %ComplexNumber{real_part: r_2, imaginary_part: i_2} do
        (r_1 * r_2 - i_1 * i_2) <|> (r_1 * i_2 + r_2 * i_1)
    end
end
```

Conversion to text

Finally, let' choose the `~~~` operator for converting a complex number to its textual version:

```
defmodule ComplexNumber do
    defstruct [:real_part, :imaginary_part]

    def r <|> i do
        %ComplexNumber{real_part: r, imaginary_part: i}
    end

    def %ComplexNumber{real_part: r_1, imaginary_part: i_1} +++
        %ComplexNumber{real_part: r_2, imaginary_part: i_2} do
        (r_1 + r_2) <|> (i_1 + i_2)
    end

    def %ComplexNumber{real_part: r_1, imaginary_part: i_1} &&&
        %ComplexNumber{real_part: r_2, imaginary_part: i_2} do
        (r_1 * r_2 - i_1 * i_2) <|> (r_1 * i_2 + r_2 * i_1)
    end
```

```
    def ~~~ %ComplexNumber{real_part: r, imaginary_part: i} do
        "(#{r}) + (#{i})i"
    end
end
```

Tesiting the implementations

In order to use the implementation of an operator, we must import the corresponding module. Because of that, let's create a small module to test the new operators:

```
defmodule ComplexNumberTest do
    import ComplexNumber

    c_1 = 2 <|> 17
    c_2 = 14 <|> 5
    c_3 = c_1 +++ c_2

    IO.puts("[#{~~~ c_1}] + [#{~~~ c_2}] = #{~~~ c_3}")

    c_1 = 3 <|> 11
    c_2 = 0 <|> 8
    c_3 = c_1 &&& c_2

    IO.puts("[#{~~~ c_1}] * [#{~~~ c_2}] = #{~~~ c_3}")
end
```

Adding the two modules to a file with the `.exs` extension and executing the script, we obtain the following result:

```
[(2) + (17)i] + [(14) + (5)i] = (16) + (22)i
[(3) + (11)i] * [(0) + (8)i] = (-88) + (24)i
```

Conclusion

The implementation of operators in Elixir is an interesting experience and which can also contribute for the creation of domain specific languages (DSLs). On the other hand, it might make the code a little harder to understand. Thus, going with new operators is a decision that involves a tradeoff.