Implementando operadores em Elixir

Fernando Karpinski

23 de junho de 2021

Resumo

Nesse artigo, implementaremos operadores na linguagem de programação Elixir. Mais especificamente, faremos isso para operadores que, no momento de escrita desse texto, são reconhecidos pela linguagem, mas não são usados por ela.

Introdução

A implementação de operadores em Elixir pode auxiliar na escrita de programas cuja sintaxe não esteja focada apenas em funções. Apesar de Elixir ser uma linguagem de programação funcional, e, portanto, o uso de funções nela ser fundamental, o uso de operadores pode nos ajudar em alguns casos. Por exemplo, algumas pessoas podem argumentar que é mais fácil entender a expressão `a + b` do que a expressão `Kernel.+(a, b)`. De fato, estamos, por exemplo, acostumados com o uso de operadores binários na matemática, de modo que é comum escrevermos o operador entre os valores sobre os quais o mesmo será aplicado.

Números complexos

Vamos implementar alguns operadores para a manipulação de números complexos. Um número complexo n é formado por duas partes, a parte real, a, e a parte imaginária, b:

$$n = a + bi$$

Podemos definir uma *struct* para representar um número complexo:

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

Criando números complexos com o primeiro operador

Para criar um número complexo, a princípio, poderíamos usar a sintaxe `%ComplexNumber{real_part:r, imaginary_part:i}`, ou até mesmo delegar esse trabalho para uma função. Porém, aqui usaremos a abordagem de implementar um operador para essa funcionalidade, nesse mesmo módulo:

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

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

Note que usamos `def`, e escrevemos o operador e os valores sobre os quais o mesmo será aplicado já no formato em que os utilizaremos na prática.

Adição

Para somarmos dois números complexos $c1 = m + ni$ e $c2 = x + yi$, usamos a seguinte medologia:

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

Vamos implementar o operador `+++` para essa adição:

```
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
```

Usando *pattern matching*, extraímos as parte real e imaginária de cada número complexo e computamos o novo número, já usando o operador de criação que implementamos anteriormente. Usamos os parênteses para garantir que as partes do novo número complexo serão calculadas antes da criação da estrutura de dados em si.

Multiplicação

A multiplicação de dois números complexos $c1 = m + ni$ e $c2 = x + yi$ pode ser realizada conforme descrito a seguir:

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

Vamos atribuir essa implementação ao operador `&&&`:

```
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
```

Conversão para texto

Por fim, vamos escolher o operator `~~~` para converter um número complexo em sua versão textual:

```
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
```

Testando as implementações

Para usar a implementação de um operador, precisamos importar o módulo correspondente. Por isso, vamos criar um pequeno módulo para testar esses operadores sobre números complexos:

```
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
```

Adicionando os dois módulos a um arquivo de extensão `.exs` e executando-o, obtemos o seguinte resultado:

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

Conclusão

A implementação de operadores em Elixir é uma experiência interessante e que pode contribuir também para linguagens de domínio específico (DSLs - *Domain Specific Languages*). Por outro lado, é possível argumentar também que o código resultante seja um pouco difícil de ler. De todo modo, trata-se de um caminho cuja viabilidade pode precisar ser analisada no caso concreto.