The Elixir Empire

Praanto Samadder

February 8, 2024

1 Implementing the map function

Now that we have discovered the secrets of the French, we can deploy those secrets to our `map/2` function.

The `map/2` function uses the secret employed by the, previously described, `inc/1` function.

```
defmodule Napoleon do
  def map([head | []], lambda) do
    [lambda.(head)]
  end

  def map([head | tail], lambda) do
    [c | t] = tail
    [lambda.(head) | map([c | t], lambda)]
  end
end
```

Listing 1: Mapping recursively

As mentioned earlier, this is nothing but reusing the logic from our `inc/1` function. Only difference is that our `inc/1` function only took the list of integers as the only parameter and simply added 1 in every recursion step.

In our `map/2` function, we no longer hardcode our arithmetic but instead use a `lambda` function, which is passed as a parameter, to do the numeric operations for us.

2 Filtration

As with the case with `map/2`, the `filter/2` function is a reusing of the logic from `even/1`. Instead of hardcoding our true/false check with the `rem/2` function, we pass a `lambda` function to do the function for us, the `lambda` function returning true/false.

```
defmodule Napoleon do
    def filter([head | []], lambda) do
      case lambda.(head) do
        true -> [head]
        false -> []
```

```
      end
    end

    def filter([head | tail], lambda) do
      case lambda.(head) do
        true -> [head | filter(tail, lambda)]
        false -> filter(tail, lambda)
      end
    end
end
```

Listing 2: Mapping recursively

We again exploit the feature (or bug whatever it may be. Elixir is a weird language no doubt) that `[ 1, 3 | []]` returns `[1, 3]`

3 Reduction

And finally, we implement the `reduce/3` method: a more versatile version of our `length/1` method.

Detailed explanation is unnecessary: the hardcoded add operation has merely been replaced by a lambda function that must take in two parameters.

```
defmodule Napoleon do
    def reduce(src, iac, lambda) do
      do_reduce({iac, src}, lambda)
    end

    def do_reduce({x, [head | []]}, lambda) do
      lambda.(x, head)
    end

    def do_reduce({x, [head | tail]}, lambda) do
      do_reduce({lambda.(x, head), tail}, lambda)
    end
end
```

Listing 3: Mapping recursively

4 But who do we exile?

The functions, `even`, `length`, and `filter` are tail-recursive meaning the last recursive call merely returns the result without doing any additional processing.

Functions `map`, `inc`, and `reduce`, on the other hand are not, as I would argue, tail recursive as they do some additional processing before returning their results and thus shall be exiled.