The Elixir Revolution

Praanto Samadder

February 8, 2024

1 Finding the length

For finding the length recursively, we implement two functions (apart from the function included in the skeleton).

We first use pattern matching to convert our list into a cons cell, the head of with can be discarded as the French used to say during the time of their revolution.

`r_length({x, []})` is called when we have traversed to the end of the list and it is time to return our sum. Otherwise we recursively call the other `r_length` method.

```
defmodule Reduce do
    def length(arg) do
      r_length({0, arg})
    end

    def r_length({x, []}) do
      x
    end

    def r_length({x, [_ | tail]}) do
      r_length({x + 1, tail})
    end
end
```

Listing 1: Length recursively

2 Incrementing each value

While implementing `inc/1` we cannot let our elements meet the same fate as Marie-Antoinette: we must keep all the elements without discarding any heads.

We can achieve this by exploiting a feature in Elixir cons cell: if the tail of a cons cell is another cons cell, then the result is a cons cell where the head of the root cell and the head of the tail of the root are appended in to a list and the tail of the root is the new tail. What on earth does that mean?

```
iex(1)> [1 | [2 | 3]]
[1, 2 | 3]
```

Listing 2: Cons cells exploited

We can thus implement `inc/1` in the following way:

```
defmodule Reduce do
    def inc([head | []]) do
        [head + 1]
    end

    def inc([head | tail]) do
        [c | t] = tail
        [head + 1 | inc([c | t])]
    end
end
```

Listing 3: `inc/1` implemented 'exploitedly'

3 Filtering out even numbers in our list

We can implement our `even/1` method by using logic that is similar to both `inc/1` and `length/1`. We must discard the heads that are not even but keep the ones that are.

Determining if a number is even can be done with the `rem/2` function. Determining whether to keep our can be done by re-establishing the Jacobin club in the form of a `case` statement. And thus as it holds

```
defmodule Reduce do
    def even([head | []]) do
      case rem(head, 2) == 0 do
        true -> [head]
        false -> []
      end
    end

    def even([head | tail]) do
      case rem(head, 2) == 0 do
        true -> [head | even(tail)]
        false -> even(tail)
      end
    end
end
```

Listing 4: Even numbers even-tually

Using these tricks, implementing `div`, `mul`, and `odd` becomes trivial. Hence, code snippets are not included in this report.