Enumerable primes

Programming II

Johan Montelius

Spring Term 2022

Introduction

Your task is to implement a data structure that holds the infinite sequence of primes and can be used by the `Enum` and `Stream` modules. In the end we should be able to do like this:

```
iex(10)> Enum.take( Stream.map(Primes.primes(), fn(x) -> 2*x end), 5)
[4, 6, 10, 14, 22]
```

or, using the pipe notation:

```
iex(11)> Primes.primes() |>
         Stream.map(fn(x) -> 2*x end) |>
         Enum.take(5)
```

The primes should implement the `Enumarable` protocol and so you should have some understanding of `Structs` and `Protocols` but don't be afraid, things will work out.

Infinite sequence of primes

Before implementing the enumerable protocol you should implement a infinite sequence of primes using anonymous functions. We divide this into three tasks and we start by implementing the infinite sequence of natural numbers (starting on 3).

z

The function `z/1` should take one argument, an integer, and return a function that when applied returns a tuple {`n, fun`} where `n` is the integer given and `fun` is a function that takes no argument and when applied returns a tuple {`n+1, fun`}. The function now returned should of course return a tuple with the firste element `n+2` etc.

This is what it should look like:

```
iex(29)> z = Primes.z(3)
z = Primes.z(3)
#Function<4.106663018/0 in Primes.z/1>
iex(30)> {_, z} = z.()
{_, z} = z.()
{4, #Function<4.106663018/0 in Primes.z/1>}
iex(31)> {_, z} = z.()
{_, z} = z.()
{5, #Function<4.106663018/0 in Primes.z/1>}
iex(32)> {_, z} = z.()
{_, z} = z.()
```

filter

The next thing you should implement is a `filter/2` function. This function should take a function that can be used to generate a sequence as above, and a number, `f`, that it should use as a filter. It should return a tuple {`p`, `fun`} where `p` is the first number in the sequence that is not divisable by `f`. The function, the second element in the tuple, is a function that will deliver the next number in the sequnce not divisable by `f`. This is what it should look like:

```
iex(55)> {_n, f} = Primes.filter(Primes.z(3), 2)
{_n, f} = Primes.filter(fn() -> Primes.z(3) end, 2)
{3, #Function<1.106663018/0 in Primes.filter/2>}
iex(56)> {_n, f} = f.()
{_n, f} = f.()
{5, #Function<1.106663018/0 in Primes.filter/2>}
iex(57)> {_n, f} = f.()
{_n, f} = f.()
{7, #Function<1.106663018/0 in Primes.filter/2>}
```

If this works we have way of implementing a sequence of all numbers not divisable by two; why not continue on this path and implement the sieve of Eratosthenes.

sieve

Implement a function `sieve/2` that takes a function and a (prime) number as arguments. The function should be a sequence generator like `z` of `f` in the above examples. We will assume that the generator returns all possible primes greater than the prime given as the second argument.

```
  def sieve(n, p) do
    :
```

```
    :
  end
```

When we say "possible" we mean that the numbers generated have been filtered by all primes less than `p`. How do you generate the next prime greater than `p`? Can you also create a function that returns all possible primes greater than this prime although they might be divisable by the new prime that you generated?

If you can do this, how do you let `sieve/2` return a tuple `{next, fun}` where `next` is the next prime greater than `p` and `fun` is function that returns all possible primes greater than, and filtered by, the prime that you have found?

primes

This final thing is to boot strap everything and provide a function `primes` that returns a function that when applied generates a tuple `{2, fun}` where `2` of course is the first prime and `fun` is a function that will return the remaining primes.

```
def primes() do
  fn() -> {2, fn() -> .......  end} end
end
```

Hmm, what do we write on the dotted line? We could try with `z(3)`; that would give us all numbers greater than 3 but we don't want to have all the even numbers. Should we filter the number from `z(3)` with 2? Close, but that would give us the sequence 2,3,5,6 and we do not want to have 6. Ahh,....

```
iex(4)>p = Primes.primes()
#Function<2.128507103/0 in Primes.primef/0>
iex(5)> {_, p} = p.()
{_, p} = p.()
{2, #Function<7.128507103/0 in Primes.primef/0>}
iex(6)> {_, p} = p.()
{_, p} = p.()
{3, #Function<4.128507103/0 in Primes.sieve/2>}
iex(7)> {_, p} = p.()
{_, p} = p.()
{5, #Function<1.128507103/0 in Primes.filter/2>}
iex(8)> {_, p} = p.()
{_, p} = p.()
{7, #Function<1.128507103/0 in Primes.filter/2>}
iex(9)> {_, p} = p.()
```

```
{_, p} = p.()
{11, #Function<1.128507103/0 in Primes.filter/2>}
iex(10)>
```

Enumerable

So now you can generate an infinite sequence of primes, time to make it work with the `Enum` and `Strem` modules. The thing you now nned to do is implement the `Enumerable` protocol.

A protocol is defined for a particular struct so we have to create a struct associated with the `Primes` module. It's a simple data structure with only one element, `next`. that will be your primes function. The function `primes/0` will now return a `Primes` struct with the initial fucntion.

```
defmodule Primes do

  defstruct [:next]

  def primes() do
    %Primes{next: ... }
  end
```

The protocol can now be defined and we nned to implement four functions. The first three are not applicable since we have an infinite sequence so we simply return {`:error`, `__MODULE__`}. It is in the function `reduce/3` where the magic happens.

```
defimpl Enumerable do

    def count(_) do  {:error, __MODULE__}  end
    def member?(_, _) do {:error, __MODULE__}  end
    def slice(_) do {:error, __MODULE__} end

    def reduce(_,       {:halt, acc}, _fun) do
       {:halted, acc}
    end
    def reduce(primes,  {:suspend, acc}, fun) do
       {:suspended, acc, fn(cmd) -> reduce(primes, cmd, fun) end}
    end
    def reduce(primes,  {:cont, acc}, fun) do
      {p, next} = Primes.next(primes)
      reduce(next, fun.(p,acc), fun)
    end
  end
```

Now for you to implement the function `next/1`, it should return a tuple, the next prime and a Primes struct that represents the continuation. If everything works you should be able to do something like this.

```
iex(59)> Enum.take( Stream.map(Primes.primes(), fn(x) -> 2*x end), 5)
[4, 6, 10, 14, 22]
```