Task 8 - Stream of Prime Numbers

Casper Kristiansson

February 16, 2022

Introduction

Task eight of the course Programming II consisted of implementing a data structure which holds an infinity sequence of primes. The sequence should be able to be used by the protocols **Enum** and **Stream**.

Method

To solve the task the author started off by watching last week lecture on "Higher order functions" which gave insight in working with anonymous functions and creating infinity sequences. The author also read the respective articles on "Anonymous functions", "Enumerable and Streams" and lastly "Comprehensions." This different material gave sufficient knowledge to solve the task.

Result

The author will be dividing up the result section into five different sections, which is one for each sub task of the task and one for the result of running the **Enum** command.

Creating an infinity sequence

The first task involves in creating a basic infinity sequence which should return the current number and a function which returns the next number in the sequence. The first way the author solved this was to have a separate function which would increase the value and return the function. But to be consistence with the goal the author was able to create one function which can perform the necessary operations.

This means when the function is called the first time it just returns a function, if that function is applied it will return the current number and a function for the next number.

```
def z(n) do
  fn() -> {n, z(n + 1)} end
end
```

Figure 1: Infinity sequence of numbers

Filtering and finding next number

The second task involves in creating a function called **filter** which receives a sequence and a number. The goal of the function is that it should return the next number in the sequence that is not divisible by the number. The function will than return the number and a function which consists of the filter function. This way the new function will give the next number in the sequence that is not divisible by the original number.

```
def filter(func, f) do
  {numb, funcNext} = func.()
  if rem(numb, f) == 0 do
    filter(funcNext, f)
  else
    {numb, fn() -> filter(funcNext, f) end}
  end
end
```

Figure 2: Filtering out and finding non divisible numbers

Creating a sequence of infinity primes

Figure two builds the foundation of finding primes in a sequence. With the help of a function called **Sieves** and creating the correct functions calls the two methods is able to return a infinity sequence of primes. The function **Sieves** starts of by receiving an infinity sequence of **Z** and the number two. The function will than pass the two arguments to the filter function which is able to find the next number that is not divisible by two.

The function will than return a tuple of the new number and a function call to method **Sieves** with the next number and the filter function. When the function is called the function Sieves will be passing a filter function to the filter function. This way the program creates a recursive chain of filter functions for all the primes that have been found.

This method builds on the same theory as "Task 6 - Primes" where we can check if the next number in a sequence is divisible by any of the found primes. If it is the function will move on to the next one and if its not divisible by any of the found primes, the program will count it as a new

prime.

```
def sieve(n, p) do
  {numb, funcNext} = filter(n, p)
  {numb, fn() -> sieve(funcNext, numb) end}
end
def primef() do
  fn() -> {2, fn() -> sieve(z(3), 2) end} end
end
```

Figure 3: Infinity sequences of primes with the helper function Sieve

Implementing Enumerable Protocol

The instructions of the task gave the author the solution for implementing the **Enumerable** protocol and the only thing that was needed to be implemented was the data structure to handle the functions for creating an infinity sequence of primes. The data structure is firstly declared in the beginning of the defmodule **Primes**. The author than uses the function **primef** from figure three with the data structure to store the values and the function call to calculate the next prime number in the sequence.

```
defstruct [:next]
def primes() do
  %Primes{next: fn() -> {2, fn() -> sieve(z(3), 2) end} end}
end
```

Figure 4: Implementing the data structure for the Enumerable protocol

The next thing needed was to create a function which should be able to handle the next prime and storing the new function in the **:next** data structure. This could be done by simply calling the function stored in data structure which will return a prime and the function for finding the next prime.

```
def next(primes) do
  {prime, funcNext} = primes.next.()
  {prime, %Primes{next: funcNext}}
end
```

Figure 5: Calling the function and storing the next function call in the data structure

Printout

With the help of all the functions the author is able to run the command in figure six. The command will generate a sequence of 50 primes and will multiply each prime with two. The result of the command can be seen in figure seven.

```
Enum.take( Stream.map(Primes.primes(), fn(x) -> 2*x end), 50)
```

Figure 6: Desired Command

```
[4, 6, 10, 14, 22, 26, 34, 38, 46, 58, 62, 74, 82, 86, 94,
106, 118, 122, 134,142, 146, 158, 166, 178, 194, 202, 206,
214, 218, 226, 254, 262, 274, 278, 298,302, 314, 326, 334,
346, 358, 362, 382, 386, 394, 398, 422, 446, 454, 458]
```

Figure 7: Output