Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Real - Time Programming

Project 0

Functional Programming / The Actor Model

Author:
Daniel Pogorevici
std. gr. FAF-202

Supervisor:
Alex Osadcenco

Chișinău 2023

1 Code implementation

Week 1

1. Write a script that would print the message “Hello PTR” on the screen. Execute it.

```
def helloPTR() do
      "Hello PTR"
    end
end
```

Listing 1: helloPTR function.

2. Create a unit test for your project. Execute it.

```
test "Hello PTR" do
    assert Week1.helloPTR() == "Hello PTR"
  end
```

Listing 2: Unit Test.

Week 2

1. Write a function that determines whether an input integer is prime.

```
def isPrime?(n) do
      if n == 0 || n == 1 || n == 2 do
        false
      else
        Enum.all?(2..(n - 1), fn i -> rem(n, i) != 0 end)
      end
    end
```

Listing 3: isPrime Function.

Determines whether an input integer n is prime. If n is 0, 1, or 2, the function returns false. Otherwise, it checks if n is divisible by any number between 2 and n-1. If n is not divisible by any of these numbers, the function returns true.

2. Write a function to calculate the area of a cylinder, given it’s height and radius.

```
def cylinderArea(height, radius) do
      2 * :math.pi * radius * (radius + height)
    end
```

Listing 4: cylinderArea Function.

Calculates the surface area of a cylinder with given height and radius using the formula 2 * pi * radius * (radius + height).

3. Write a function to reverse a list.

```
def reverse(list) do
      Enum.reverse(list)
    end
```

Listing 5: reverse Function.

Reverses the order of elements in the given list.

4. Write a function to calculate the sum of unique elements in a list.

```
def uniqueElements(list) do
    Enum.sum(Enum.uniq(list))
  end
```

Listing 6: uniqueElements Function.

Calculates the sum of unique elements in the given list using Enum.uniq to remove duplicates and Enum.sum to sum the resulting list.

5. Write a function that extracts a given number of randomly selected elements from a list.

```
def randomNumbers(list, n) do
    Enum.take_random(list, n)
  end
```

Listing 7: randomNumbers Function.

Extracts n randomly selected elements from the given list using Enum.take.random.

6. Write a function that returns the first n elements of the Fibonacci sequence

```
def firstFibonacci(n) do
    fibonacci = firstFibonacci(n - 1)
    next_fibonacci = Enum.at(fibonacci, n - 3) + Enum.at(fibonacci, n - 2)
    fibonacci ++ [next_fibonacci]
  end
```

Listing 8: firstFibonacci Function.

Returns the first n elements of the Fibonacci sequence. If n is 1 or 2, the function returns [1] or [1, 1], respectively. Otherwise, it calculates the nth element by summing the last two elements and appending the result to the list.

7. Write a function that, given a dictionary, would translate a sentence. Words not found in the dictionary need not be translated.

```
def translator(dict, str) do
    str
    |> String.split(" ")
    |> Enum.map(fn word -> Map.get(dict, String.to_atom(word), word) end)
    |> Enum.join(" ")
  end
```

Listing 9: translator Function.

Translates a given sentence str using a dictionary dict that maps words to their translations. The function splits str into words, maps each word to its translation using Map.get, and then joins the translated words back into a sentence.

8. Write a function that receives as input three digits and arranges them in an order that would create the smallest possible number. Numbers cannot start with a 0.

```
def smallestNumber(a, b, c) do
    list = [a, b, c]
    list = Enum.sort(list)
    if List.first(list) == 0 do
      if Enum.at(list, 1) == 0 do
        [Enum.at(list, 2), 0, 0]
      else
        [Enum.at(list, 1), 0, Enum.at(list, 2)]
      end
    else
```

```
        list
      end
    end
```

Listing 10: smallestNumber Function.

Arranges the digits a, b, and c in an order that creates the smallest possible number. If any digit is 0, the function moves it to the beginning of the list. Otherwise, it sorts the digits in increasing order.

9. Write a function that would rotate a list n places to the left.

```
def rotateLeft(list, n) when n > 0 do
      [first | last] = list
      rotateLeft(last ++ [first], n - 1)
    end
```

Listing 11: rotateLeft Function.

Rotates the given list n places to the left by recursively appending the first element to the end of the list n times.

10. Write a function that lists all tuples a, b, c such that a2+b2 = c2 and a, b 20.

```
def listRightAngleTriangles() do
      for a <- 1..20, b <- 1..20,
      c = :math.floor(:math.sqrt(a*a + b*b)),
      c*c == a*a + b*b, do: {a, a, c}
    end
```

Listing 12: listRightAngleTriangles Function.

The function uses a for comprehension to generate all possible values of a, b, and c that satisfy the condition.

11. Write a function that eliminates consecutive duplicates in a list.

```
def removeConsecutiveDuplicates(list) do
      Enum.dedup(list)
    end
```

Listing 13: removeConsecutiveDuplicates Function.

Removes consecutive duplicates from the given list using Enum.dedup.

Week 3

1. Create an actor that prints on the screen any message it receives.

```
def printMessage do
    receive do
      msg ->
        IO.puts(msg)
    end
    printMessage()
  end
```

Listing 14: printMessage Actor.

This function receives a message and prints it to the console using IO.puts. It then calls itself recursively to receive and print more messages.

2. Create an actor that returns any message it receives, while modifying it

```
def modifier do
    receive do
      message when is_integer(message) ->
        msg = message + 1
        IO.puts("Received: " <> to_string(msg))
      message when is_bitstring(message) ->
        msg = String.downcase(message)
        IO.puts("Received: "  <> to_string(msg))
      message ->
        IO.puts("Received: I don't know how to HANDLE this!")
    end
    modifier()
  end
```

Listing 15: modifier Actor.

This function receives a message and checks its type using guards. If the message is an integer, it increments it by 1 and prints the result. If the message is a string, it converts it to lowercase and prints the result. If the message is of any other type, it prints an error message. It then calls itself recursively to receive and process more messages.

3. Create a two actors, actor one "monitoring" the other. If the second actor stops, actor one gets notified via a message.

```
def actor, do: exit(:bye)

  def actor_monitor(pid) do
    Process.monitor(pid)

    receive do
      {:DOWN, _ref, :process, ^pid, reason} ->
        IO.puts("Actor stopped!")
        IO.puts("Reason: "  <> to_string(reason))
    end
  end
```

Listing 16: monitor Actor.

This function receives a process ID (pid) and sets up a monitor for that process using Process.monitor. It then waits for a :DOWN message, indicating that the monitored process has exited. When it receives the :DOWN message, it prints a message indicating that the actor has stopped and the reason for its exit.

4. Create an actor which receives numbers and with each request prints out the current average.

```
def averager(sum, n) do
  receive do
    nr when is_number(nr) ->
      sum = sum + nr
      n = n + 1
      average = sum / n
      IO.puts("Current average is: " <> to_string(average))
      averager(sum, n)
    end
  end
```

Listing 17: averager Actor.

This function receives a sum and a count, and then waits for messages containing numbers. When it receives a number message, it adds the number to the sum and increments the count. It then

calculates the current average and prints it to the console. It then calls itself recursively to receive and process more messages.

5. Create an actor which maintains a simple FIFO queue. You should write helper functions to create an API for the user, which hides how the queue is implemented.

```
def new_queue do
    spawn(fn -> loop([]) end)
  end

  def push(pid, value) do
    send(pid, {:queue, value})
    :ok
  end
  def pop(pid) do
    ref = make_ref()
    send(pid, {:dequeue, self(), ref})
    receive do
      {:ok, ^ref, value} -> value
    end
  end
  defp loop(queue) do
    receive do
      {:queue, value} ->
        loop([value | queue])

      {:dequeue, sender, ref} ->
        if queue == [] do
          send(sender, {:error, ref, nil})
        else
          last = List.last(queue)
          send(sender, {:ok, ref, last})
          loop(List.delete_at(queue, -1))
        end
    end
  end
```

Listing 18: FIFO Actor.

This actor takes a queue and waits for messages containing :queue, value or :dequeue, sender, ref. If it receives a :queue, value message, it adds the value to the front of the queue and calls itself recursively to continue processing messages. If it receives a :dequeue, sender, ref message and the queue is empty, it sends an error message back to the sender with a :error, ref, nil tuple. If the queue is not empty, it removes the last element from the queue, sends a message back to the sender with a :ok, ref, last tuple, and calls itself recursively to continue processing messages.

Week 4

1. Create a supervised pool of identical worker actors. The number of actors is static, given at initialization. Workers should be individually addressable. Worker actors should echo any message they receive. If an actor dies (by receiving a "kill" message), it should be restarted by the supervisor. Logging is welcome.

```
defmodule Week4.WorkersPool do
  def start_group() do
    w_list = [Worker1, Worker2, Worker3, Worker4, Worker5]

    children = for w_name <- w_list do
      %{id: w_name, start: {Week4.WorkersPool.WorkerNode, :start_link, [w_name]}}
```

```
    end

    {:ok, supervisor_pid} = Supervisor.start_link(children, strategy: :one_for_one)
    worker_pids = Supervisor.which_children(supervisor_pid) |> Enum.map(&elem(&1, 1))
    {supervisor_pid, worker_pids}
  end
end

defmodule Week4.WorkersPool.WorkerNode do
  def start_link(w_name) do
    pid = spawn_link(fn -> loop(w_name) end)
    {:ok, pid}
  end

  def loop(w_name) do
    receive do
      {:kill} ->
        Process.flag(:trap_exit, true)
        exit(:stop)

      {:message, message} ->
        message |> IO.puts()
        loop(w_name)
    end
  end

  def kill(pid) do
    send(pid, {:kill})
  end

  def message(pid, message) do
    send(pid, {:message, message})
  end
end
```

Listing 19: Supervised workers Pool.

The Week4.WorkersPool.WorkerNode module defines a worker process that performs a simple loop. The startlink function starts a new worker process and returns its pid. The loop function waits for incoming messages, and either exits the process when it receives a :kill message, or prints the received message and continues looping when it receives a :message, message tuple.

The kill and message functions provide simple APIs to send messages to worker processes to stop or handle incoming messages.

Week 5

1. Write an application that would visit this link. Print out the HTTP response status code, response headers and response body.

Continue your previous application. Extract all quotes from the HTTP response body. Collect the author of the quote, the quote text and tags. Save the data into a list of maps, each map representing a single quote.

Continue your previous application. Persist the list of quotes into a file. Encode the data into JSON format. Name the file quotes.json.

```
def get_quotes do
  case HTTPoison.get("https://quotes.toscrape.com/") do
    {:ok, %HTTPoison.Response{body: body}} ->
      Floki.find(body, "div.quote")
```

```
        |> Enum.map(&parse_quote/1)
      {:error, reason} ->
        IO.puts "Failed to fetch quotes: #{inspect(reason)}"
        []
    end
  end

  defp parse_quote(div) do
    %{
      quote: div |> Floki.find("span.text") |> Floki.text(),
      author: div |> Floki.find("small.author") |> Floki.text(),
      tags: div |> Floki.find("div.tags a.tag") |> Enum.map(&Floki.text/1)
    }
  end

  def write_quotes_to_json() do
    case File.write("quotes.json", Jason.encode!(get_quotes())) do
      :ok -> IO.puts "Quotes written to quotes.json"
      {:error, reason} -> IO.puts "Failed to write quotes: #{inspect(reason)}"
    end
  end
```

Listing 20: Quotes Fetcher.

The get.quotes function uses the HTTPoison library to fetch the HTML content of the website and then uses Floki to extract the quotes and their information from the HTML. The parse.quote function extracts the quote, author, and tags from the HTML for each quote. Finally, the write.quotes.to.json function encodes the quotes as JSON using the Jason library and writes them to a file named quotes.json in the current directory. If the write operation succeeds, the function outputs a success message; otherwise, it outputs an error message.

2 Conclusion

After implementing the task, I've learned that functional Programming and the Actor Model are powerful paradigms that can be leveraged in building scalable, fault-tolerant, and highly available systems. The Elixir programming language is built on top of the Erlang Virtual Machine, which provides a solid foundation for building concurrent and distributed applications.

Overall, Functional Programming and the Actor Model, along with the Elixir programming language, can be a powerful combination in building reliable and scalable systems. With the increasing demand for highly concurrent and distributed systems, this paradigm can provide a way to build such systems that are easier to maintain, test, and reason about.