Automata Simulator, an implementation in Elixir

Luciana da Costa Marques
Polytechnic School of the University of Sao Paulo
e-mail: luciana.marques@usp.br

1. Introduction

This article presents concepts of deterministic and non-deterministic finite state automata and an implementation of a simulator using the Elixir functional programming language. This work was mainly based on the work done in class and also [1].

2. Definitions

This section presents theory definitions used in this article.

2.1. Deterministic Automaton

A deterministic automaton can be described as a function that decided whether an input string can be generated or not given a pre-defined grammar.

The simulator uses a graph to describe the automaton's behavior and its transition function. The chosen automaton to be simulated is the one seen in class that accepts strings indexed by "ab".

It's accepted character set is (a,b).

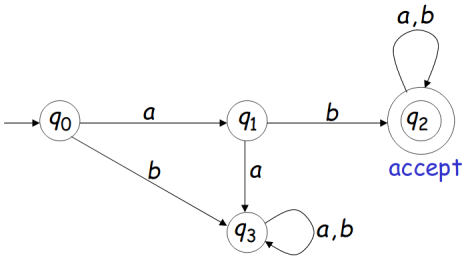

Figure 1. Deterministic Finite Automaton

2.2. Non Deterministic Automaton

A non-deterministic automaton can also be described as a function that states whether a given input string can be generated by a pre-defined grammar. It may, however, have more than one possible computation.

The transition function must, hence, to compute possible computation given the automaton's graph. The one chosen for the simulation is for inputs containing exactly and only two a's ("aa"), no more or less than two. It's accepted character set is (a).

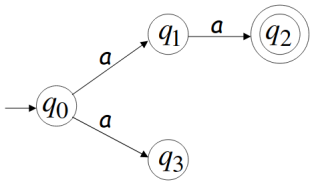

Figure 2. Non-Deterministic Finite Automaton

3. Implementation

Both mix projects are submitted with this report.

As previously mentioned, the automaton simulator here presented was done using the Elixir programming language, which is built under the functional paradigm.

There are two mix projects in this work, one for the deterministic finite automaton and one for the non-deterministic one. There were two examples chosen from class resources (slides), one for each type of automaton.

The automaton's functionalities are basically read an input tape given an alphabet and make the state transition based on its current state and tape index being read at the same given moment.

3.1. Read the tape

The function used to describe the tape's read is "read(tape)" present in determautomata.ex. It was the first function produced in order to understand the mechanism for reading an input string. It is a recursive function and it basically outputs the current top of the tape, deletes it and call itself again with the rest of the tape. It stops once the tape is empty.

With Elixir, there is no possibility of changing a List size. So whenever there is the need to analyze a different

tape index, another list is created and there is a new internal call to the function.

```
def read(tape) do
    if (List.first(tape) == nil) do
      IO.puts("End of tape")
    else
      IO.puts(List.first(tape))
      tape2 = List.delete_at(tape,0)
      read(tape2)
    end
end
```

3.2. State transition function

To compute the automaton's transitions, there is actually two functions, automaton-transition() and get-next-state(). The first one is the one which gets the final automaton's state after reading the whole tape. It is present in both projects (non-deterministic and deterministic automaton).

```
def automata_transition(current_state, tape) do
    next_state = get_next_state(current_state,
    tape)
    if (next_state == "q2") do
      true
    else
      false
    end
end
```

The second function needs to be programmed accordingly to the automaton being simulated. For the deterministic automaton, we know that an acceptance state is generated for tapes starting with "ab" and "q2" is that final acceptance state, which is tested in the above function once every call to get-next-state() is done.

The functionality of reading through the tape is embedded in the function. Every new call to get-next-state() is done after deleting the first index of the previous tape version and thus creating another List.

Once the list being analyzed is empty, the function returns the last detected state.

```
def get_next_state(current_state, tape) do
    cond do
      List.first(tape) == nil -> #end of tape
        current_state

      current_state == "q0" ->
        IO.puts("Beginning of tape")
        if (List.first(tape) == "a") do
          tape2 = List.delete_at(tape,0)
          get_next_state("q1",tape2)
        else
          tape2 = List.delete_at(tape,0)
          get_next_state("q3",tape2)
        end

      current_state == "q1" ->
        if (List.first(tape) == "a") do
          tape2 = List.delete_at(tape,0)
          get_next_state("q3",tape2)
        else
          tape2 = List.delete_at(tape,0)
          get_next_state("q2",tape2)
        end

      current_state == "q2" ->
        tape2 = List.delete_at(tape,0)
        get_next_state("q2",tape2)

      current_state == "q3" ->
        tape2 = List.delete_at(tape,0)
        get_next_state("q3",tape2)
    end
end
```

For the non-deterministic automaton, a similar function was made. Again, this needs to be done accordingly to the automaton being simulated. The chosen automaton has more than one transition possibility when the current state is "q0" and the current input is "a". It first tries to go to the path throughout "q1", which is an invalid final state. If the tape is empty and the final state is q1, it will try to follow the path through "q3".

There was an extra state added "q4". This was done in order to make the computation easier: if the input tape is "aa" then its final state is "q2" and it is accepted. If it's "aaa" or "aaaa" and so on, its final state is "q4" and it is not accepted.

```
def get_next_state(current_state, tape) do
    IO.puts(current_state)
    cond do
      List.first(tape) == nil -> #end of tape
        current_state

      # This state can be split in two paths
      current_state == "q0" ->
        if (List.first(tape) == "a") do
          tape2 = List.delete_at(tape,0)
          # if path through q1 doesn't work, try
    trough q3
          if (get_next_state("q1",tape2) == "q1")
    do
            get_next_state("q3",tape2)
          end
        end

      current_state == "q1" ->
        if (List.first(tape) == "nil") do
          current_state
        else
          tape2 = List.delete_at(tape,0)
          get_next_state("q2",tape2)
        end

      current_state == "q2" ->
        if (List.first(tape) == nil) do
          current_state
        else
```

```
            tape2 = List.delete_at(tape,0)
            get_next_state("q4",tape2)
          end

        current_state == "q3" ->
          tape2 = List.delete_at(tape,0)
          get_next_state("q3",tape2)

        current_state == "q4" ->
          tape2 = List.delete_at(tape,0)
          get_next_state("q4",tape2)
    end
end
```

3.3. Main function

There was a main function written in order to judge if the automaton's output is true or false given an input tape. This was done mainly to test support and could be melted with the get-next-state() function.

```
def automata_transition(current_state, tape) do
    next_state = get_next_state(current_state,
    tape)
    if (next_state == "q2") do
      true
    else
      false
    end
end
```

For both automatons the final acceptance state is "q2" so there was no necessary modifications.

3.4. General Assumptions

Since the main goal in this project was to make the automaton simulator, some assumptions were made in order to make it more direct and simple.

First, it was assumed that the input tape would only contain letter from the alphabet accepted by the automaton.

It was also assumed that the acceptance states would be known by the programmer using this project, that is, no user input would be necessary in order to know which states would produce an acceptance. The only input necessary is the tape in a string format.

4. Tests

There were tests written for both simulators.

For the deterministic automaton, the following tests were run:

```
  test "iterate through tape" do
    # creates tape
    tape = String.codepoints("ab")
    # iterates through tape
    DETERMAUTOMATA.read(tape)
  end

  test "automaton transition - #1" do
    tape = String.codepoints("abb")
    assert DETERMAUTOMATA.determ_automaton(tape)
    == true
  end

  test "automaton transition - #2" do
    tape = String.codepoints("aab")
    assert DETERMAUTOMATA.determ_automaton(tape)
    == true
  end
```

The first test was to check whether the read() function worked. The two other tests checked the automaton's output for "abb" and "aab" input tapes. All tests passed.

For the non-deterministic automaton simulator, the following tests were run, and all of them passed.

```
  test "input tape on automaton - #1" do
    IO.puts("Test - #1")
    tape = String.codepoints("a")
    assert NONDETAUTOM.determ_automaton(tape) ==
    false
  end

  test "input tape on automaton - #2" do
    IO.puts("Test - #2")
    tape = String.codepoints("aa")
    assert NONDETAUTOM.determ_automaton(tape) ==
    false
  end

  test "input tape on automaton - #3" do
    IO.puts("Test - #3")
    tape = String.codepoints("aaaa")
    assert NONDETAUTOM.determ_automaton(tape) ==
    false
  end
```

5. Conclusion

Both simulators worked accordingly to their expected behavior. The following steps of this project could be to do a general automaton simulation framework for other programmers willing to test their designed automata.

Also, there could be some defensive programming. For example, the user could input the desired acceptance states and the program could check whether an input is part of the allowed alphabet.

References

[1] Christian Wagenknecht and Daniel P. Friedman. *Teaching Nondeterministic and Universal Automata using Scheme*. Indiana Univeristy, Bloomington, IN 47405, October 2, 1996.