PCS2046- Lógica Computacional

3° Exercício-Programa
Autômatos Finitos

Professor Doutor Ricardo Rocha

9783640 - Luís Henrique Barroso Oliveira

9835623 - Rodrigo Vali Cebrian

11259715 - Vanderson da Silva dos Santos

São Paulo, 12 de março de 2023

SUMÁRIO

1. Enunciado

O objetivo didático desta atividade é experimentar concretamente os conceitos desenvolvidos em sala de aula a respeito de autômatos, não-determinismo, etc. Além disso, deve ser usada uma linguagem funcional como paradigma de linguagem de programação - uma das linguagens Elixir ou Clojure.

O objetivo do exercício é implementar um algoritmo de simulação de autômato determinístico e não-determinístico:

1. Construa um simulador de autômato finito determinístico em Elixir ou Clojure, usando como apoio os artigos abaixo.
2. Construa um simulador de autômatos finitos não-determinísticos em Elixir ou Clojure, a partir do exercício anterior.

2. Simulador de autômato finito

Um autômato finito determinístico (AFD) é uma máquina abstrata que reconhece ou rejeita uma sequência de símbolos de entrada com base em um conjunto finito de estados e uma função de transição que especifica como o autômato se move entre os estados quando lê cada símbolo de entrada.

Formalmente, um AFD é definido como uma tupla M = (Q, Σ, δ, q0, F), onde:

- Q é um conjunto finito de estados
- Σ é um conjunto finito de símbolos de entrada, chamado de alfabeto
- δ é a função de transição δ : Q x Σ -> Q que associa um estado a cada par (estado atual, símbolo de entrada)
- q0 é o estado inicial, q0 ∈ Q
- F é um conjunto de estados finais, F ⊆ Q

O AFD começa no estado inicial q0 e lê um símbolo de entrada de cada vez. A cada símbolo lido, o autômato muda seu estado de acordo com a função de transição δ. Quando o autônomo lê toda a entrada, ele aceita se terminar em um estado final em F e rejeita caso contrário.

O AFD é chamado de determinístico porque, para cada estado e cada símbolo de entrada, há apenas uma transição possível, ou seja, a função de transição δ é determinística.

A partir dessa definição de definição de autômato finito determinístico, buscou-se uma solução para o problema empregando a linguagem de programação Clojure.

Foram construídas uma série de funções auxiliares para a reprodução do comportamento do autômato, que serão descritas a seguir.

3. Descrição das funções utilizadas

3.1. Função SolveFiniteAutomaton

```
(defn SolveFiniteAutomaton [states actions next_states accept_states input]
      (let
            [
                  deterministic_transistions (GenTransistions states actions next_states)
                  final_states (GetResultState deterministic_transistions input)

                  result (some #(contains? (set accept_states) %) final_states)
            ]
            [result deterministic_transistions final_states]
      )
)
```

A função recebe seis argumentos: *states, actions, next_states, accept_states* e *input*. E retorna uma lista contendo três elementos: *result, deterministic_transitions e final_states*.

Seu objetivo é implementar um autômato finito determinístico e verificar se uma determinada entrada *input* é aceita ou rejeitada pelo autômato.

Cada argumento tem o seguinte significado:

- *states* é uma lista de todos os estados possíveis do autômato
- *actions* é uma lista de todos os símbolos de entrada possíveis que o autômato pode ler
- *next_states* é uma lista de transições de estado para cada símbolo de entrada. Por exemplo, se o autômato estiver no estado A e ler o símbolo '0', ele irá para o estado B, e se ler o símbolo '1', ele irá para o estado C. As transições de estado são definidas em uma função auxiliar chamada *GenTransitions*.
- *accept_states* é uma lista de estados finais aceitos pelo autômato
- *input* é a entrada para o autômato

A função começa a chamar a função auxiliar *GenTransitions* para criar as transições do autômato. Em seguida, chama a função *GetResultState*, que executa a entrada no autômato e retorna uma lista de estados finais alcançáveis pelo autômato a partir do estado inicial e da entrada fornecida.

A variável *final_states* contém a lista de estados finais alcançados. A variável *result* é verdadeira se algum dos estados finais calculados estiver na lista de estados finais aceitos e é falsa caso contrário.

A função retorna uma lista que contém a variável *result* indicando se a entrada foi aceita ou não, *deterministic_transitions* que é a lista de transições geradas pelo autômato e *final_states* que é a lista de estados finais alcançados.

3.2. Função GetResultState

```
(defn GetResultState
      ([matrix input] (GetResultState matrix input "Q1"))
      ([matrix input actual_state]

            (let
                  [
                        new_action (first input)
                        next_states (GetNextState matrix actual_state new_action)
                        new_input (rest input)
                  ]
                  (if (not (empty? new_input))
                        (vec (distinct (flatten (map (partial GetResultState matrix new_input) next_states) )))
                        next_states
                  )
            )
      )

)
```

Esta é a implementa uma função *GetResultState*, empregada para obter o estado final alcançado por um autômato finito determinístico ao processar uma determinada entrada.

A função tem duas assinaturas:

- (*[matrix input]*): assume que o estado atual é "Q1".
- (*[matrix input actual_state]*): assume que o estado atual é *actual_state*.
- A função recebe três argumentos: matrix, input e *actual_state*.

matrix é uma matriz que contém as informações do AFD, como as transições de estado, o alfabeto, o conjunto de estados e o estado inicial.

input é a entrada que será processada pelo AFD.

actual_state é o estado atual do AFD.

A função começa obtendo o primeiro símbolo de entrada *new_action* e, em seguida, chama a função auxiliar *GetNextState* para obter o próximo estado do autômato a partir do estado atual e do símbolo de entrada.

Em seguida, a função verifica se há mais símbolos de entrada restantes *new_input* para serem processados. Se houver, a função chama recursivamente *GetResultState* para cada um dos próximos estados *next_states*, passando a entrada restante *new_input* e acumulando todos os estados finais alcançados em uma lista única e distinta usando a função *distinct*.

Por fim, se não houver mais símbolos de entrada restantes, a função retorna a lista de estados finais *next_states*.

Em resumo, a função *GetResultState* percorre recursivamente o AFD para processar uma entrada e obter todos os estados finais alcançados.

3.3. Função GetTransitions

```
(defn GenTransistions [states actions next_states]
      (let
            [
                  GroupByKey (fn [chaves valores]
                        (->>  (map vector chaves valores)
                              (group-by first)
                              (map (fn [[k vs]] [k (vec (map second vs))]))
                              (into {})
                        )
                  )
                  actions->next_state (vec (map GroupByKey actions next_states) )
                  state->action (zipmap states actions->next_state)
            ]
            state->action
      )
)
```

Este é um trecho de código que implementa uma função *GenTransistions* para gerar as transições de um autômato finito a partir dos seus estados, símbolos de entrada e próximos estados.

A função recebe três argumentos: states, actions e *next_states*.

- *states* é um vetor contendo os estados do AFD.
- *actions* é um vetor contendo os símbolos de entrada do AFD.
- *next_states* é uma matriz que contém os próximos estados do AFD. Cada linha representa um estado e cada coluna representa uma ação.

A célula *[i,j]* contém o próximo estado do AFD quando ele está no estado *i* e lê a ação *j*.

A função começa definindo uma função auxiliar *GroupByKey* que recebe duas sequências chaves e valores, mapeia cada elemento de chaves para uma lista de pares com valores e agrupa esses pares pelo primeiro elemento, criando um mapa cujas chaves são as chaves originais e os valores são listas dos seus respectivos valores.

Em seguida, a função aplica *GroupByKey* para cada par de elementos em actions e *next_states* usando a função *map*. Isso cria uma lista de mapas, cada um contendo as transições de um estado para todos os próximos estados possíveis para cada símbolo de entrada. O resultado é um vetor de mapas *actions -> next_state*, onde cada mapa contém as transições de um estado para os próximos estados possíveis para cada símbolo de entrada. Por fim, a função usa a função *zipmap* para criar um mapa *state->action* que mapeia cada estado em *states* para as suas transições em *actions->next_state.*

Em resumo, a função *GenTransistions* gera um mapa que representa as transições de um AFD a partir de seus estados, símbolos de entrada e próximos estados.

3.4. Função GetNextState

```
(defn GetNextState [matrix state action]
      (let
            [
                  next_state (get (get matrix state) action)
            ]
            next_state
      )
)
```

A função *GetNextState* recebe três argumentos: uma matriz *matrix* que representa as transições de um autômato finito determinístico (AFD), um estado atual *state* e um símbolo de entrada *action*. O objetivo da função é retornar ao próximo estado do AFD após ler o símbolo de entrada *action* no estado atual *state*.

Utiliza a função *get* para obter a linha da matriz correspondente ao estado atual *state*. Em seguida, usa novamente a função *get* para obter o próximo estado correspondente ao símbolo de entrada action.

Por fim, a função retorna ao próximo estado obtido.

Em resumo, a função *GetNextState* retorna o próximo estado de um AFD a partir do estado atual e do símbolo de entrada fornecido, usando a matriz de transições *matrix*.

3.5. Função que implementa o autômato finito determinístico

```
(defn MainNonDeterministic []
      (let
            [
                  states  ["Q1" "Q2" "Q3"]

                  actions [
                        ["a" "a" "c" "d"]
                        ["a" "b" "c" "c"]
                        ["a" "b" "c" "d"]
                  ]

                  next_states [
                        ["Q1" "Q2" "Q2" "Q3"]
                        ["Q2" "Q2" "Q3" "Q2"]
                        ["Q2" "Q3" "Q3" "Q3"]
                  ]

                  accept_states ["Q2" "Q3"]

                  input ["a","b","c"]

                  [result deterministic_transistions final_states] (SolveFiniteAutomaton states actions next_states accept_states input)
            ]
            (do
                  (println "--- Non Deterministic Finite Automaton ---")
                  (println "States = " states)
                  (println "Actions = " actions)
                  (println "Next states = " next_states)
                  (println "Accept states = " accept_states)
                  (println "Input states = " input)
                  (println "Transitions = " (reverse deterministic_transistions))
                  (println "Final States = " final_states)
                  (if result
                        (println "Automata aceita! :)")
                        (println "Automata Não foi aceita :(")
                  )

            )
            result
      )
)
```

A função *MainDeterministic* é uma função que utiliza a função *SolveFiniteAutomaton* para verificar se um autômato finito determinístico (DFA) aceita uma determinada entrada. Ela define alguns parâmetros que representam o DFA, como *states* (os estados do DFA), *actions* (as ações possíveis que podem ser tomadas no DFA), *next_states* (os estados para os quais o DFA transita a partir de um estado dado e uma ação dada) e

accept_states (os estados de aceitação do DFA). Em seguida, ele define uma entrada de teste, *input*, que é uma sequência de símbolos.

A função *MainDeterministic* chama a função *SolveFiniteAutomaton* com esses parâmetros e armazena o resultado em uma tupla que contém o resultado da verificação do DFA, as transições determinísticas do DFA e os estados finais alcançados pelo DFA para a entrada de teste.

A seguir, a função *MainDeterministic* exibe informações sobre o DFA e a entrada de teste e, em seguida, verifica se o DFA aceita a entrada de teste. Se o DFA aceitar a entrada de teste, a função exibe a mensagem "Automata aceita! :)", caso contrário, exibe "Automata Não foi aceita :(". Por fim, a função retorna o resultado da verificação do DFA (true ou false).

3.6. Função que implementa o autômato finito não-determinístico

```
(defn MainNonDeterministic []
      (let
            [
                  states  ["Q1" "Q2" "Q3"]

                  actions [
                        ["a" "a" "c" "d"]
                        ["a" "b" "c" "c"]
                        ["a" "b" "c" "d"]
                  ]

                  next_states [
                        ["Q1" "Q2" "Q2" "Q3"]
                        ["Q2" "Q2" "Q3" "Q2"]
                        ["Q2" "Q3" "Q3" "Q3"]
                  ]

                  accept_states ["Q2" "Q3"]

                  input ["a","b","c"]

                  [result deterministic_transistions final_states] (SolveFiniteAutomaton states actions next_states accept_states input)
            ]
            (do
                  (println "--- Non Deterministic Finite Automaton ---")
                  (println "States = " states)
                  (println "Actions = " actions)
                  (println "Next states = " next_states)
                  (println "Accept states = " accept_states)
                  (println "Input states = " input)
                  (println "Transitions = " (reverse deterministic_transistions))
                  (println "Final States = " final_states)
                  (if result
                        (println "Automata aceita! :)")
                        (println "Automata Não foi aceita :(")
                  )

            )
            result
      )
)
```

A função *MainNonDeterministic* implementa um exemplo de uso do código para resolver um autômato finito não determinístico (AFND) usando a função *SolveFiniteAutomaton*. A função começa definindo os estados do autômato,

as ações possíveis em cada estado, os próximos estados alcançados por cada ação, os estados de aceitação, e a entrada que será testada no autômato.

Em seguida, a função chama a função *SolveFiniteAutomaton* com esses parâmetros, e armazena o resultado em uma variável chamada *result*. Depois disso, a função imprime na tela informações relevantes sobre o autômato, como os estados, ações, próximos estados, estados de aceitação, entrada, transições, e estados finais resultantes. Por fim, a função imprime se o autômato aceitou ou não a entrada.

O objetivo da função *MainNonDeterministic* é apenas mostrar como usar a função *SolveFiniteAutomaton* para resolver um exemplo de um autômato finito não determinístico e apresentar as informações relevantes desse processo para o usuário.

4. Testes

Para testar a execução do programa, foram produzidos testes unitários para cada uma das funções implementadas e depois testes completos do algoritmo. Todos os testes criados passaram com sucesso, e estão descritos a seguir:

Testes Unitários

Autômato finito determinístico

```
;; Deterministic functions tests --------------------
(deftest GenDeterministicTransistionsTest []
  (let [
      states0 ["Q1" "Q2" "Q3"]
      actions0 [["a"] ["a"] ["a"]]
      next_state0 [["Q1"] ["Q2"] ["Q2"]]
      result0 (ep3.core/GenTransistions states0 actions0 next_state0)

      states1 ["Q1" "Q2" "Q3"]
      actions1 [["a" "b" "c"] ["a" "c"] ["a" "d"]]
      next_state1 [["Q1" "Q1" "Q2"] ["Q2" "Q3"] ["Q2" "Q1"]]
      result1 (ep3.core/GenTransistions states1 actions1 next_state1)
    ]

    (testing "Testando a função CreateR"
      (is (= result0 {"Q3" {"a" ["Q2"]}, "Q2" {"a" ["Q2"]}, "Q1" {"a" ["Q1"]}} ) )
      (is (= result1 {"Q3" {"a" ["Q2"] "d" ["Q1"]},
                      "Q2" {"a" ["Q2"], "c" ["Q3"]},
                      "Q1" {"a" ["Q1"], "b" ["Q1"], "c" ["Q2"]
                      }}))
    )
  )
)      You, 7 hours ago • 🖌 add ep3 deterministic tests …
```

```
(deftest GetDeterministicNextStateTest []
  (let [
      matrix {"Q3" {"d" ["Q3"], "c" ["Q3"], "b" ["Q3"], "a" ["Q2"]},
              "Q2" {"c" ["Q3"], "b" ["Q2"], "a" ["Q2"]},
              "Q1" {"d" ["Q3"], "c" ["Q2"], "b" ["Q2"], "a" ["Q1"]}}

      result0 (ep3.core/GetNextState matrix "Q3" "a")
      result1 (ep3.core/GetNextState matrix "Q3" "b")
      result2 (ep3.core/GetNextState matrix "Q1" "a")
    ]

    (testing "Testando a função CreateR"
      (is (= result0 ["Q2"]))
      (is (= result1 ["Q3"]))
      (is (= result2 ["Q1"]))
    )
  )
)
```

```
(deftest GetDeterministicResultStateTest []
  (let [
      matrix {"Q3" {"d" ["Q3"], "c" ["Q3"], "b" ["Q3"], "a" ["Q2"]},
              "Q2" {"c" ["Q3"], "b" ["Q2"], "a" ["Q2"]},
              "Q1" {"d" ["Q3"], "c" ["Q2"], "b" ["Q2"], "a" ["Q1"]}}

      input0 ["a" "b" "c"]
      result0 (ep3.core/GetResultState matrix input0 )

      input1 ["a" "b"]
      result1 (ep3.core/GetResultState matrix input1 )

      input2 ["a" "d"]
      result2 (ep3.core/GetResultState matrix input2 )

      input3 ["a" "a" "a"]
      result3 (ep3.core/GetResultState matrix input3 )
    ]

    (testing "Testando a função CreateR"
      (is (= result0 ["Q3"]))
      (is (= result1 ["Q2"]))
      (is (= result2 ["Q3"]))
      (is (= result3 ["Q1"]))
    )
  )
)
```

```
(deftest SolveDeterministicFiniteAutomatonTest []
  (let [
      states  ["Q1" "Q2" "Q3"]

      actions [
                ["a" "b" "c" "d"]
                ["a" "b" "c"]
                ["a" "b" "c" "d"]
              ]

      next_states [
                    ["Q1" "Q2" "Q2" "Q3"]
                    ["Q2" "Q2" "Q3"]
                    ["Q2" "Q3" "Q3" "Q3"]
                  ]

      accept_states ["Q2" "Q3"]

      TestFunction (partial ep3.core/SolveFiniteAutomaton states actions next_states accept_states)

      input0 ["a","b","c"]
      [result0 _ _] (TestFunction input0)

      input1 ["a","a","a"]
      [result1 _ _] (TestFunction input1)

      input2 ["a","b"]
      [result2 _ _] (TestFunction input2)
    ]

    (testing "Testando a função CreateR"
      (is result0)
      (is (not result1) )
      (is result2)
    )
  )
)
```

Autômato finito não-determinístico

```
(deftest GenNonDeterministicTransistionsTest []
  (let [
      states0 ["Q1" "Q2" "Q3"]
      actions0 [["a" "a"] ["a"] ["a"]]
      next_state0 [["Q1" "Q2"] ["Q2"] ["Q2"]]
      result0 (ep3.core/GenTransistions states0 actions0 next_state0)
    ]

    (testing "Testando a função CreateR"
      (is (= result0 {"Q3" {"a" ["Q2"]},
                      "Q2" {"a" ["Q2"]},
                      "Q1" {"a" ["Q1" "Q2"]}} ) )
    )
  )
)
```

```
(deftest GetNonDeterministicNextStateTest []
  (let [
      matrix {"Q3" {"a" ["Q2"]},
              "Q2" {"a" ["Q2"]},
              "Q1" {"a" ["Q1" "Q2"]}}

      result0 (ep3.core/GetNextState matrix "Q3" "a")
      result1 (ep3.core/GetNextState matrix "Q2" "a")
      result2 (ep3.core/GetNextState matrix "Q1" "a")
    ]

    (testing "Testando a função CreateR"
      (is (= result0 ["Q2"]))
      (is (= result1 ["Q2"]))
      (is (= result2 ["Q1" "Q2"]))
    )
  )
)
```

```
(deftest GetNonDeterministicResultStateTest []
  (let [
      matrix {"Q3" {"a" ["Q2"], "b" ["Q3"]},
              "Q2" {"a" ["Q2"], "b" ["Q1"]},
              "Q1" {"a" ["Q1" "Q2"], "b" ["Q3"], "d" ["Q3" "Q2"] }}

      input0 ["b" "a" "a"]
      result0 (ep3.core/GetResultState matrix input0 )

      input1 ["a" "b"]
      result1 (ep3.core/GetResultState matrix input1 )

      input2 ["d" "b"]
      result2 (ep3.core/GetResultState matrix input2 )

      input3 ["a" "a" "a"]
      result3 (ep3.core/GetResultState matrix input3 )
    ]

    (testing "Testando a função CreateR"
      (is (= result0 ["Q2"]))
      (is (= result1 ["Q3" "Q1"]))
      (is (= result2 ["Q3" "Q1"]))
      (is (= result3 ["Q1" "Q2"]))
    )
  )
)
```

```
(deftest SolveNonDeterministicFiniteAutomatonTest []
  (let [
      states  ["Q1" "Q2" "Q3"]

      actions [
                ["a" "a" "b" "d" "d"]
                ["a" "b"]
                ["a" "b"]
              ]

      next_states [
                    ["Q1" "Q2" "Q3" "Q3" "Q2"]
                    ["Q2" "Q1"]
                    ["Q2" "Q3"]
                  ]

      accept_states ["Q3"]

      TestFunction (partial ep3.core/SolveFiniteAutomaton states actions next_states accept_states)

      input0 ["b" "a" "a"]
      [result0 _ _] (TestFunction input0)

      input1 ["a" "b"]
      [result1 _ _] (TestFunction input1)

      input2 ["a" "a" "a"]
      [result2 _ _] (TestFunction input2)
    ]

    (testing "Testando a função CreateR"
      (is (not result0))
      (is result1 )
      (is (not result2) )
    )
  )
)
```

O resultado de todos foi positivo. De todos os 8 teste com as 23 verificações, todas passaram com sucesso. Esse resultado pode ser visto abaixo:

```
root@1c1cd64f90d6:/usr/app/src/ep3# lein test

lein test ep3.core-test

Ran 8 tests containing 23 assertions.
0 failures, 0 errors.
```

Testes De Integração

Autômato finito determinística

Nesse primeiro caso aqui, podemos observar que a automata foi aceita, pois o estado final estava incluso nos estados de aceitação.

```
--- Deterministic Finite Automaton ---
States =  [Q1 Q2 Q3]
Actions =  [[a b c d] [a b c d] [a b c d]]
Next states =  [[Q1 Q2 Q2 Q3] [Q2 Q2 Q3 Q1] [Q2 Q3 Q3 Q3]]
Accept states =  [Q2 Q3]
Input states =  [a b c]
Transitions =  ([Q1 {a [Q1], b [Q2], c [Q2], d [Q3]}] [Q2 {a [Q2], b [Q2], c [Q3], d [Q1]}] [Q3 {a [Q2], b [Q3], c [Q3], d [Q3]}]
)
Final States =  [Q3]
Automata aceita! :)
```

Nesse segundo caso, podemos observar uma situação que a automata não é aceita. Diferente do primeiro teste completo, nessa situação o estado final não se encontra no estado de aceitação.

```
--- Deterministic Finite Automaton ---
States =  [Q1 Q2 Q3]
Actions =  [[a b c d] [a b c d] [a b c d]]
Next states =  [[Q1 Q2 Q2 Q3] [Q2 Q2 Q1 Q1] [Q2 Q3 Q3 Q3]]
Accept states =  [Q2 Q3]
Input states =  [a b c]
Transitions =  ([Q1 {a [Q1], b [Q2], c [Q2], d [Q3]}] [Q2 {a [Q2], b [Q2], c [Q1], d [Q1]}] [Q3 {a [Q2], b [Q3], c [Q3], d [Q3]}])
Final States =  [Q1]
Automata Não foi aceita :(
```

Autômato finito não-determinístico

Nesse primeiro caso aqui, podemos observar que a automata não determinística foi aceita, pois os estados finais (pelo menos um deles) estavam inclusos nos estados de aceitação.

```
--- Non Deterministic Finite Automaton ---
States =  [Q1 Q2 Q3]
Actions =  [[a a c d] [a b c c] [a b c d]]
Next states =  [[Q1 Q2 Q2 Q3] [Q2 Q2 Q3 Q2] [Q2 Q3 Q3 Q3]]
Accept states =  [Q2 Q3]
Input states =  [a b c]
Transitions =  ([Q1 {a [Q1 Q2], c [Q2], d [Q3]}] [Q2 {a [Q2], b [Q2], c [Q3 Q2]}] [Q3 {a [Q2], b [Q3], c [Q3], d [Q3]}])
Final States =  [Q3 Q2]
Automata aceita! :)
```

Em contrapartida, no segundo caso, podemos observar uma situação que a automata não é aceita. Diferente do primeiro teste completo, nessa situação os estados finais não se encontram em nenhum estado de aceitação.

```
--- Non Deterministic Finite Automaton ---
States =  [Q1 Q2 Q3]
Actions =  [[a a c d] [a b c c] [a b c d]]
Next states =  [[Q1 Q2 Q2 Q3] [Q2 Q2 Q3 Q2] [Q2 Q3 Q3 Q3]]
Accept states =  [Q3]
Input states =  [a a a]
Transitions =  ([Q1 {a [Q1 Q2], c [Q2], d [Q3]}] [Q2 {a [Q2], b [Q2], c [Q3 Q2]}] [Q3 {a [Q2], b [Q3], c [Q3], d [Q3]}])
Final States =  [Q1 Q2]
Automata Não foi aceita :(
```

5. Conclusão

O programa desenvolvido tem por objetivo simular os mecanismos de funcionamento que caracteriam os autômatos finitos determinísticos e não-determinísticos.

Lançou-se mão dos referenciais teóricos para definir a forma em que o programa deveria funcionar, e os exemplos em aula foram fundamentais para facilitar o entendimento de uma perspectiva prática.

6. Bibliografia

Vídeo sobre NDA:
https://www.youtube.com/watch?v=W8Uu0inPmU8&list=PLbj1IxX9Gw0Mfk1UYkMDEwpBoAlS0xgF3&index=2

Material Teórico da disciplina:

https://edisciplinas.usp.br/pluginfile.php/7432246/mod_resource/content/1/Teaching_Nondeterministic_and_Universal_Automata_using_Scheme.pdf

Repositório do github do projeto:
https://github.com/VanderSant/PCS3556_computational_logic

7. Anexo A: Código Completo

```
;; Definition of namespace -------------

(ns ep3.core
```

```
      (:gen-class)
      (:require [clojure.set :as set]))

;; Finite Automaton Functions --------------------

(defn GetNextState [matrix state action]
      (let
            [
                  next_state (get (get matrix state) action)
            ]
            next_state
      )
)

(defn GenTransistions [states actions next_states]
      (let
            [
                  GroupByKey (fn [chaves valores]
                        (->>  (map vector chaves valores)
                              (group-by first)
                              (map (fn [[k vs]] [k (vec (map second
vs))]))
                              (into {})
                        )
                  )
                  actions->next_state (vec (map GroupByKey actions
next_states) )
                  state->action (zipmap states actions->next_state)
            ]
            state->action
      )
)

(defn GetResultState
```

```
     ([matrix input] (GetResultState matrix input "Q1"))
     ([matrix input current_state]

           (let
                 [
                       new_action (first input)
                       next_states (GetNextState matrix current_state
new_action)
                       new_input (rest input)
                 ]
                 (if (not (empty? new_input))
                       (vec (distinct (flatten (map (partial
GetResultState matrix new_input) next_states) )))
                       next_states
                 )
           )
     )

)

(defn SolveFiniteAutomaton [states actions next_states accept_states
input]
     (let
           [
                 deterministic_transistions (GenTransistions states
actions next_states)
                 final_states (GetResultState
deterministic_transistions input)

                 result (some #(contains? (set accept_states) %)
final_states)
           ]
           [result deterministic_transistions final_states]
     )
)
```

```
;; Main functions --------------------

(defn MainDeterministic []
     (let
           [
                 states  ["Q1" "Q2" "Q3"]

                 actions [
                       ["a" "b" "c" "d"]
                       ["a" "b" "c" "d"]
                       ["a" "b" "c" "d"]
                 ]

                 next_states [
                       ["Q1" "Q2" "Q2" "Q3"]
                       ["Q2" "Q2" "Q1" "Q1"]
                       ["Q2" "Q3" "Q3" "Q3"]
                 ]

                 accept_states ["Q2" "Q3"]

                 input ["a" "b" "c"]

                 [result deterministic_transistions final_states]
(SolveFiniteAutomaton states actions next_states accept_states input)
           ]
           (do
                 (println "--- Deterministic Finite Automaton ---")
                 (println "States = " states)
                 (println "Actions = " actions)
                 (println "Next states = " next_states)
                 (println "Accept states = " accept_states)
                 (println "Input states = " input)
```

```
                (println "Transitions = " (reverse
deterministic_transistions))
                (println "Final States = " final_states)
                (if result
                      (println "Automata aceita! :)")
                      (println "Automata Não foi aceita :(")
                )

          )
          result
     )
)


(defn MainNonDeterministic []
     (let
          [
                states  ["Q1" "Q2" "Q3"]

                actions [
                      ["a" "a" "c" "d"]
                      ["a" "b" "c" "c"]
                      ["a" "b" "c" "d"]
                ]

                next_states [
                      ["Q1" "Q2" "Q2" "Q3"]
                      ["Q2" "Q2" "Q3" "Q2"]
                      ["Q2" "Q3" "Q3" "Q3"]
                ]

                accept_states ["Q3"]

                input ["a","a","a"]
```

```
                [result deterministic_transistions final_states]
(SolveFiniteAutomaton states actions next_states accept_states input)
            ]
            (do
                (println "--- Non Deterministic Finite Automaton
---")
                (println "States = " states)
                (println "Actions = " actions)
                (println "Next states = " next_states)
                (println "Accept states = " accept_states)
                (println "Input states = " input)
                (println "Transitions = " (reverse
deterministic_transistions))
                (println "Final States = " final_states)
                (if result
                    (println "Automata aceita! :)")
                    (println "Automata Não foi aceita :(")
                )

            )
            result
    )
)

;; Main function ---------------------

(defn -main []
    (MainDeterministic)
    (MainNonDeterministic)
)
```

8. Anexo B: Código De Testes Completo

```
(ns ep3.core-test
 (:require [clojure.test :refer :all]
           [ep3.core :refer :all]
   )
)

;; Deterministic functions tests --------------------
(deftest GenDeterministicTransistionsTest []
 (let [
     states0 ["Q1" "Q2" "Q3"]
     actions0 [["a"] ["a"] ["a"]]
     next_state0 [["Q1"] ["Q2"] ["Q2"]]
     result0 (ep3.core/GenTransistions states0 actions0 next_state0)

     states1 ["Q1" "Q2" "Q3"]
     actions1 [["a" "b" "c"] ["a" "c"] ["a" "d"]]
     next_state1 [["Q1" "Q1" "Q2"] ["Q2" "Q3"] ["Q2" "Q1"]]
     result1 (ep3.core/GenTransistions states1 actions1 next_state1)
   ]
    (testing "Testando a função CreateR"
     (is (= result0 {"Q3" {"a" ["Q2"]}, "Q2" {"a" ["Q2"]}, "Q1" {"a"
["Q1"]}} ) )
     (is (= result1 {"Q3" {"a" ["Q2"] "d" ["Q1"]},
                     "Q2" {"a" ["Q2"], "c" ["Q3"]},
                     "Q1" {"a" ["Q1"], "b" ["Q1"], "c" ["Q2"]
                     }}))
   )
 )
)

(deftest GetDeterministicNextStateTest []
```

```
 (let [
     matrix {"Q3" {"d" ["Q3"], "c" ["Q3"], "b" ["Q3"], "a" ["Q2"]},
             "Q2" {"c" ["Q3"], "b" ["Q2"], "a" ["Q2"]},
             "Q1" {"d" ["Q3"], "c" ["Q2"], "b" ["Q2"], "a" ["Q1"]}}

     result0 (ep3.core/GetNextState matrix "Q3" "a")
     result1 (ep3.core/GetNextState matrix "Q3" "b")
     result2 (ep3.core/GetNextState matrix "Q1" "a")
   ]
    (testing "Testando a função CreateR"
     (is (= result0 ["Q2"]))
     (is (= result1 ["Q3"]))
     (is (= result2 ["Q1"]))
   )
 )
)

(deftest GetDeterministicResultStateTest []
 (let [
     matrix {"Q3" {"d" ["Q3"], "c" ["Q3"], "b" ["Q3"], "a" ["Q2"]},
             "Q2" {"c" ["Q3"], "b" ["Q2"], "a" ["Q2"]},
             "Q1" {"d" ["Q3"], "c" ["Q2"], "b" ["Q2"], "a" ["Q1"]}}

     input0 ["a" "b" "c"]
     result0 (ep3.core/GetResultState matrix input0 )

     input1 ["a" "b"]
     result1 (ep3.core/GetResultState matrix input1 )

     input2 ["a" "d"]
     result2 (ep3.core/GetResultState matrix input2 )

     input3 ["a" "a" "a"]
     result3 (ep3.core/GetResultState matrix input3 )
```

```
   ]
    (testing "Testando a função CreateR"
     (is (= result0 ["Q3"]))
     (is (= result1 ["Q2"]))
     (is (= result2 ["Q3"]))
     (is (= result3 ["Q1"]))
   )
 )
)

(deftest SolveDeterministicFiniteAutomatonTest []
 (let [
     states  ["Q1" "Q2" "Q3"]

     actions [
               ["a" "b" "c" "d"]
               ["a" "b" "c"]
               ["a" "b" "c" "d"]
             ]

     next_states [
                   ["Q1" "Q2" "Q2" "Q3"]
                   ["Q2" "Q2" "Q3"]
                   ["Q2" "Q3" "Q3" "Q3"]
                 ]

     accept_states ["Q2" "Q3"]

     TestFunction (partial ep3.core/SolveFiniteAutomaton states
actions next_states accept_states)

     input0 ["a","b","c"]
     [result0 _ _] (TestFunction input0)
```

```
     input1 ["a","a","a"]
     [result1 _ _] (TestFunction input1)

     input2 ["a","b"]
     [result2 _ _] (TestFunction input2)
   ]
    (testing "Testando a função CreateR"
     (is result0)
     (is (not result1) )
     (is result2)
   )
 )
)

;; Non Deterministic functions Tests --------------------

(deftest GenNonDeterministicTransistionsTest []
 (let [
     states0 ["Q1" "Q2" "Q3"]
     actions0 [["a" "a"] ["a"] ["a"]]
     next_state0 [["Q1" "Q2"] ["Q2"] ["Q2"]]
     result0 (ep3.core/GenTransistions states0 actions0 next_state0)
   ]

   (testing "Testando a função CreateR"
     (is (= result0 {"Q3" {"a" ["Q2"]},
                     "Q2" {"a" ["Q2"]},
                     "Q1" {"a" ["Q1" "Q2"]}} ) )
   )
 )
)

(deftest GetNonDeterministicNextStateTest []
 (let [
```

```
    matrix {"Q3" {"a" ["Q2"]},
            "Q2" {"a" ["Q2"]},
            "Q1" {"a" ["Q1" "Q2"]}}

    result0 (ep3.core/GetNextState matrix "Q3" "a")
    result1 (ep3.core/GetNextState matrix "Q2" "a")
    result2 (ep3.core/GetNextState matrix "Q1" "a")
  ]
   (testing "Testando a função CreateR"
    (is (= result0 ["Q2"]))
    (is (= result1 ["Q2"]))
    (is (= result2 ["Q1" "Q2"]))
  )
 )
)

(deftest GetNonDeterministicResultStateTest []
 (let [
    matrix {"Q3" {"a" ["Q2"], "b" ["Q3"]},
            "Q2" {"a" ["Q2"], "b" ["Q1"]},
            "Q1" {"a" ["Q1" "Q2"], "b" ["Q3"], "d" ["Q3" "Q2"] }}

    input0 ["b" "a" "a"]
    result0 (ep3.core/GetResultState matrix input0 )

    input1 ["a" "b"]
    result1 (ep3.core/GetResultState matrix input1 )

    input2 ["d" "b"]
    result2 (ep3.core/GetResultState matrix input2 )

    input3 ["a" "a" "a"]
    result3 (ep3.core/GetResultState matrix input3 )
  ]
```

```
    (testing "Testando a função CreateR"
     (is (= result0 ["Q2"]))
     (is (= result1 ["Q3" "Q1"]))
     (is (= result2 ["Q3" "Q1"]))
     (is (= result3 ["Q1" "Q2"]))
   )
 )
)

(deftest SolveNonDeterministicFiniteAutomatonTest []
 (let [
     states  ["Q1" "Q2" "Q3"]

     actions [
               ["a" "a" "b" "d" "d"]
               ["a" "b"]
               ["a" "b"]
             ]

     next_states [
                   ["Q1" "Q2" "Q3" "Q3" "Q2"]
                   ["Q2" "Q1"]
                   ["Q2" "Q3"]
                 ]

     accept_states ["Q3"]

     TestFunction (partial ep3.core/SolveFiniteAutomaton states
actions next_states accept_states)

     input0 ["b" "a" "a"]
     [result0 _ _] (TestFunction input0)

     input1 ["a" "b"]
```

```
      [result1 _ _] (TestFunction input1)


      input2 ["a" "a" "a"]
      [result2 _ _] (TestFunction input2)
    ]
     (testing "Testando a função CreateR"
      (is (not result0))
      (is result1 )
      (is (not result2) )
    )
  )
)
```