Relatório - EP3

Tiago Koji Castro Shibata - 8988730
Escola Politécnica
Universidade de São Paulo
tiago.shibata@usp.br

I. Introdução

Esse relatório acompanha o terceiro exercício programa (EP3) da disciplina PCS3556 - Lógica Computacional. Nesse exercício programa, é implementado um simulador de autômato finito determinístico e não determinístico em Elixir.

26 de Março de 2018

II. Tarefa

A tarefa consiste em implementar um algoritmo de simulação de autômato determinístico e não determinístico em Elixir, experimentando com conceitos vistos em aula.

Um autômato finito é definido pela tupla:

$$M = (Q, \Sigma, \delta, q_0, F) \tag{1}$$

Onde Q é a lista de estados do autômato, Σ o alfabeto de entrada, δ a função de transição, q_0 o estado inicial, e F o conjunto de estados de aceitação. Dado um autômato, estado inicial e cadeia, o algoritmo deve retornar se é possível que o autômato aceite a cadeia (ou seja, há uma sequência de transições iniciando no estado inicial e acabando em estado de aceitação que gere a cadeia desejada).

Como vimos em aula, o estudo de autômatos é bastante importante. Linguagens reconhecidas por autômatos são regulares, e toda linguagem regular pode ser representada por um autômato. Autômatos determinísticos (DFA) e não determinísticos (NFA) são equivalentes (é possível converter qualquer NFA em DFA equivalente, que aceita e rejeita as mesmas cadeias; no entanto, para um NFA de n estados, o DFA equivalente pode ter até 2^n estados, portanto a representação em NFA pode ser mais conveniente. No entanto, a simulação de DFA é muito mais fácil, já que não há várias transições possíveis a se testar).

Na hierarquia de Chomsky, autômatos finitos estão na classe de linguagens regulares:

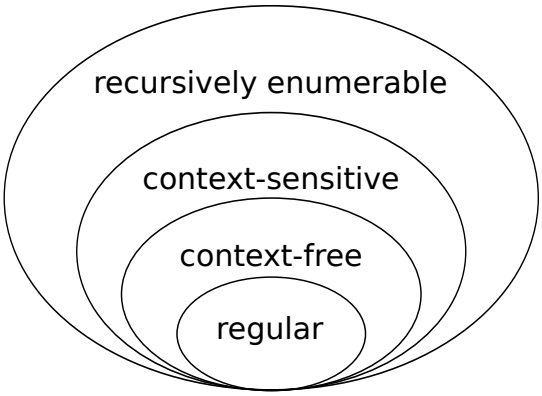

Fig. 1: Hierarquia de Chomsky

III. Estruturas de dados

Em alguns locais, estruturas de conjunto fornecidas pelo Elixir (*MapSet* foram usadas. O uso de conjunto evita que varramos a lista toda para buscar um elemento, e o conjunto permite operações fáceis e rápidas de união ou diferença quando necessário. Funções do módulo *Enum* foram usadas para facilitar a programação funcional.

Nessa implementação, δ é dado como uma lista de transições (lista de tuplas do tipo $\{estado, caractere, próximo\_estado\}$, significando que $\delta(estado, caractere) = próximo\_estado$). Os estados de aceitação são dados como uma lista de estados e a cadeia desejada é dada como uma lista de elementos.

IV. Algoritmo

O algoritmo deve suportar autômatos não determinísticos, incluíndo transições vazias. O uso de transições vazias adiciona uma dificuldade à tarefa: ao seguir uma sequência de transições, o algoritmo deve acompanhar estados visitados para não ficar preso em um ciclo de transições em vazio.

A função *next_state* calcula os possíveis próximos estados dado o estado atual, regras de transição e caractere atual. Para evitar ciclos infinitos em caso de transição vazia, os estados já vistos são mantidos em um conjunto; o estado não é visitado novamente se já pertencer ao conjunto. A função filtra regras que iniciam no estado inicial e transitam pelo caractere na entrada. Ela também se chama recursivamente por transições vazias, e concatena os resultados. Finalmente, é usado um *pipe* para um *MapSet*, removendo quaisquer estados repetidos:

```
def next_state(state, input, rules, already_visited \\ MapSet.new) do
  if MapSet.member?(already_visited, state) do
    []
  else
```

```
    already_visited = MapSet.put(already_visited, state)
    rules_from_state = Enum.filter(rules, &(elem(&1, 0) == state))
    (Enum.filter(rules_from_state, &elem(&1, 1) == input)
    |> Enum.map(&elem(&1, 2)))
    ++ (Enum.filter(rules_from_state, &is_nil(elem(&1, 1)))
    |> Enum.flat_map(&next_state(elem(&1, 2), input, rules, already_visited)))
  end |> MapSet.new
end
```

Os testes foram essenciais no desenvolvimento, já que essa função apresenta alguns *corner cases*, como ciclos de transições vazias. A função foi testada com um autômato com ciclo de transições vazias e funcionou corretamente:

```
@nondeterministic_with_empty_loop [
  {:q0, :a, :q0},
  {:q0, :a, :q1},
  {:q0, :a, :q2},
  {:q0, :b, :q1},
  {:q0, nil, :q0},
  {:q0, nil, :q3},
  {:q3, nil, :q0},
  {:q3, :a, :q4},
  {:q4, :b, :q5},
]

def assert_have_same_elements(a, b) do
  assert MapSet.new(a) == MapSet.new(b)
end

test "computes next state" do
  assert_have_same_elements(
    FiniteAutomaton.next_state(:q0, :a, @nondeterministic_with_empty_loop),
    [:q0, :q1, :q2, :q4])
end
```

A função *final_states* computa os possíveis estados finais de um autômato não determinístico. Ele recebe o estado inicial, sentença de entrada e regras de transição, e retorna estados finais possíveis consumindo um caractere por vez e chamando-se recursivamente:

```
def final_states(state, [], _), do: MapSet.new [state]

def final_states(state, [head | tail], rules) do
  MapSet.new Enum.flat_map(next_state(state, head, rules), &final_states(&1, tail,
        rules))
end
```

A função foi testada:

```
test "computes final state" do
  assert_have_same_elements(
    FiniteAutomaton.final_states(:q0, [:b], @nondeterministic_with_empty_loop),
    [:q1])
  assert_have_same_elements(
    FiniteAutomaton.final_states(:q0, [:a, :b], @nondeterministic_with_empty_loop),
    [:q1, :q5])
end
```

A função *accepts_sentence?* verifica se uma sentença pode ser aceita por um NFA. Para isso, a função recebe uma lista de estados de aceitação, e busca por intersecção com os estados retornados por *final_states*:

```
def accepts_sentence?(state, input, rules, accept_states) do
  MapSet.intersection(final_states(state, input, rules), MapSet.new accept_states)
  |> MapSet.size > 0
end
```

Finalmente, foram feitos testes de *accepts_sentence?*:

```
test "tests sentence acceptance" do
  assert FiniteAutomaton.accepts_sentence?(:q0, [],
        @nondeterministic_with_empty_loop, [:q0])
  assert not FiniteAutomaton.accepts_sentence?(:q0, [],
        @nondeterministic_with_empty_loop, [:q1, :q2, :q3, :q4, :q5])
  assert not FiniteAutomaton.accepts_sentence?(:q0, [:b],
        @nondeterministic_with_empty_loop, [:q0])
  assert FiniteAutomaton.accepts_sentence?(:q0, [:b],
        @nondeterministic_with_empty_loop, [:q1])
  assert FiniteAutomaton.accepts_sentence?(:q0, [:a, :b],
        @nondeterministic_with_empty_loop, [:q1])
  assert FiniteAutomaton.accepts_sentence?(:q0, [:a, :b],
        @nondeterministic_with_empty_loop, [:q5])
  assert not FiniteAutomaton.accepts_sentence?(:q0, [:a, :b],
        @nondeterministic_with_empty_loop, [:q0, :q2, :q3, :q4])
end
```

Mais testes foram feitos com autômatos vistos em aula, e com sucesso:

```
simple_grammar = [
  {:q0, :a, :q1},
  {:q0, :b, :q2},
  {:q1, :a, :q0},
]
assert FiniteAutomaton.accepts_sentence?(:q0, [:b], simple_grammar, [:q2])
assert FiniteAutomaton.accepts_sentence?(:q0, [:a, :a, :b], simple_grammar, [:q2])
assert FiniteAutomaton.accepts_sentence?(:q0, [:a, :a, :a, :a, :b], simple_grammar,
       [:q2])
assert not FiniteAutomaton.accepts_sentence?(:q0, [:a], simple_grammar, [:q2])
assert not FiniteAutomaton.accepts_sentence?(:q0, [:a, :b, :a], simple_grammar, [:
      q2])
assert not FiniteAutomaton.accepts_sentence?(:q0, [:a, :a, :b, :b], simple_grammar,
       [:q2])
```

V. Resultados

A implementação foi um sucesso e passou nos testes realizados. Casos com transições vazias, incluindo ciclos, foram testados e aprovados, e o código funcionou em qualquer DFA ou NFA testado.

VI. Conclusão

Pude aplicar os conhecimentos da disciplina em um problema prático e realizar reconhecimento de cadeias usando autômatos finitos, uma das opções para reconhecimento de linguagens regulares. Pude aplicar conhecimentos de gramáticas e hierarquia de Chomsky na resolução de um problema em Elixir. A implementação foi, inesperadamente, bastante menos trabalhosa que a do exercício programa passado (EP2).

References

[1] Friedel Ziegelmayer. *Elixir ExDoc*. https://hexdocs.pm/elixir/, acessado em 11/02/2018

[2] Wagenknecht, C., Friedman, P.D. *Teaching Nondeterministic and Universal Automata Using SCHEME*. Computer Science Education, vol. 8, Issue 3, 1998, p.197-227