Relatório - EP2

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

I. Introdução

Esse relatório acompanha o segundo exercício programa (EP2) da disciplina PCS3556 - Lógica Computacional.

04 de Março de 2018

II. Tarefa

A tarefa consiste em implementar um algoritmo de reconhecimento de cadeias geradas por uma gramática de estrutura de frase recursiva. O algoritmo reconhecedor deve retornar se uma dada cadeia pode ser gerada por uma gramática.

III. Algoritmo

O algoritmo deve suportar qualquer gramática sensível ao contexto (ou seja, pela hierarquia de Chomsky, linguagens sensíveis ao contexto, livres de contexto e regulares são suportadas):

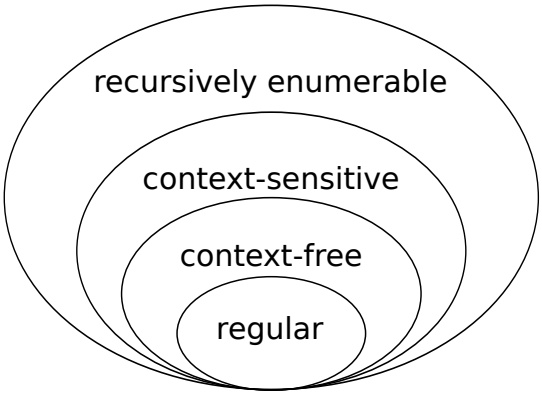

Fig. 1: Hierarquia de Chomsky

Conforme especifiado, a implementação é recursiva e feita em Elixir, uma linguagem funcional. Conforme sugerido no enunciado, foi implementada uma função de expansão a partir de uma forma sentencial inicial, gerando todas as formas sentenciais e sentenças intermediárias, e ao fim verificou-se a pretinência da sentença buscada na lista de sentenças geradas.

IV. Estruturas de dados

Em alguns locais, estruturas de conjunto fornecidas pelo Elixir ($MapSet$) foram usadas tendo em mente performance e facilidade: 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.

V. Código e testes

A função $apply\_rule(rule, state)$ recebe uma regra de substituição e um estado do gerador (uma sentença ou forma sentencial). Ela gera todas as formas sentenciais e sentenças possíveis aplicando essa regra no estado base.

A regra é dada como uma tupla $\{cadeia\ \ inicial, cadeia\ \ a\ \ ser\ \ colocada\}$. Foram escritos testes para essa função:

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

test "applies a rule" do
  assert_have_same_elements(ElixirRegularGrammarMatching.apply_rule(
    {"A", "a"}, "AA"),
    ["AA", "aA", "Aa", "aa"])
  assert_have_same_elements(ElixirRegularGrammarMatching.apply_rule(
    {"AA", "aa"}, "AAA"),
    ["AAA", "aaA", "Aaa"])
  assert_have_same_elements(ElixirRegularGrammarMatching.apply_rule(
    {"A", "a"}, ""),
    [""])
  assert_have_same_elements(ElixirRegularGrammarMatching.apply_rule(
    {"ABA", "a"}, "ABABA"),
    ["ABABA", "aBA", "ABa"])
end
```

Os testes foram essenciais no desenvolvimento, já que essa função apresenta muitos *corner cases*. Por exemplo, casos com regras que batem em posições que se sobrepõe ($AA -> a$ com a forma sentencial AAA, por exemplo) falhariam em implementações básicas usando *String.split* para buscar correspondências.

A função foi implementada buscando a primeira correspondência da condição da regra na forma sentencial com a função *String.split*. Se não há correspondência, ocorre o fim da recursão. Se há, são feitas duas chamadas recursivas: uma realizando a substituição, que ira chamar $apply\_rule$ com o lado direito da cadeia, e outra não realizando a substituição, que avança um caractere da correspondência encontrada e chama $apply\_rule$:

```
def apply_rule(rule, state) do
  {condition, replacement} = rule
  case String.split(state, condition, parts: 2) do
    [left, right] ->
      {head, tail} = String.split_at(condition, 1)
      Enum.map(apply_rule(rule, right), &(left <> replacement <> &1))
      ++ Enum.map(apply_rule(rule, tail <> right), &(left <> head <> &1))
    [_] ->
      [state]
  end
end
```

Depois, foram feitos testes e implementação de uma função que aplique uma lista de regras:

```
test "applies an iteration of rules" do
  assert_have_same_elements(ElixirRegularGrammarMatching.apply_rules(
    [{"A", "aA"}, {"A", "ab"}], "A"),
    ["A", "aA", "ab"])
  assert_have_same_elements(ElixirRegularGrammarMatching.apply_rules(
    [{"A", "ABa"}, {"A", "b"}, {"B", "Bb"}, {"B", "b"}], "aABa"),
    ["aABa", "aABaBa", "abBa", "aABba", "aAba"])
  assert_have_same_elements(ElixirRegularGrammarMatching.apply_rules(
    [{"A", "aAa"}, {"aAa", "BaB"}, {"B", "b"}], "BaAa"),
```

```
    ["BaAa", "BaaAaa", "BBaB", "baAa"])
end


def apply_rules(rules, state) do
  Enum.flat_map(rules, &(apply_rule(&1, state)))
end
```

Então, foi feita uma função que aplica regras até um comprimento limite. Foram feitos testes com gramáticas livres de contexto, dependentes de contexto, e alguns *corner cases*. Um caso notavelmente complicado foi regras que incluíssem expressões como $A -> B, B -> A$, já que há um ciclo entre A e B sem aumentar o tamanho da cadeia. A implementação deve acompanhar quais formas sentenciais já foram visitadas para não ficar presa em um ciclo no qual o comprimento se mantém.

```
test "applies rules until an specific length" do
  assert ElixirRegularGrammarMatching.apply_rules_until_length("ab",
    [{"A", "aA"}, {"A", "ab"}], "A", 4) ==
    MapSet.new(["A", "aA", "aaA", "aaaA", "aaab", "aab", "ab"])
  assert ElixirRegularGrammarMatching.apply_rules_until_length("ab",
    [{"A", "a"}, {"A", "aAa"}, {"aAa", "bAb"}], "A", 5) ==
    MapSet.new(["A", "a", "aAa", "aaAaa", "aaa", "aaaaa", "abAba", "ababa", "bAb",
          "baAab", "baaab", "bab", "bbAbb", "bbabb"])
  assert ElixirRegularGrammarMatching.apply_rules_until_length("ab",
    [{"A", "B"}, {"B", "A"}, {"A", "a"}], "A", 3) ==
    MapSet.new(["A", "B", "a"])
end
```

A implementação utiliza um $MapSet$ para acompanhar formas sentenciais geradas e outro para já visitadas. São feitas chamadas recursivas a $apply\_rules$ enquanto houver formas não visitadas.

```
def apply_rules_until_length(terminals, rules, {states, visited}, max_length) do
  not_visited = MapSet.difference(states, visited)
  case Enum.fetch(not_visited, 0) do
    {:ok, state} ->
      new_states = apply_rules(rules, state)
      |> Enum.filter(&(String.length(&1) <= max_length))
      |> MapSet.new
      apply_rules_until_length(terminals, rules, {MapSet.union(states, new_states),
            MapSet.put(visited, state)}, max_length)
    :error ->
      {states, visited}
  end
end
```

Algumas funções adicionais com interface mais simples foram definidas:

```
def apply_rules_until_length(terminals, rules, state, max_length) do
  apply_rules_until_length(terminals, rules, {MapSet.new([state]), MapSet.new},
        max_length)
  |> elem(1)
end

def can_generate_sentence(grammar, sentence) do
  {terminals, rules, start} = grammar
  apply_rules_until_length(terminals, rules, start, String.length(sentence))
  |> MapSet.member?(sentence)
end
```

E testes finais feitos:

```
test "checks whether a grammar can generate a sentence" do
  grammar = {
    "ab",
    [{"A", "ABa"}, {"A", "a"}, {"B", "Bb"}, {"B", "b"}],
    "A"
  }
  assert ElixirRegularGrammarMatching.can_generate_sentence(grammar, "a")
  assert ElixirRegularGrammarMatching.can_generate_sentence(grammar, "aba")
  assert ElixirRegularGrammarMatching.can_generate_sentence(grammar, "abbabbba")
  assert not ElixirRegularGrammarMatching.can_generate_sentence(grammar, "ba")
  assert not ElixirRegularGrammarMatching.can_generate_sentence(grammar, "abab")
  context_sensitive_grammar = {
    "ab",
    [{"A", "aAa"}, {"aAa", "BaB"}, {"B", "Bb"}, {"B", "b"}],
    "A"
  }
  assert ElixirRegularGrammarMatching.can_generate_sentence(
        context_sensitive_grammar, "bab")
  assert ElixirRegularGrammarMatching.can_generate_sentence(
        context_sensitive_grammar, "aabbbabaa")
  assert ElixirRegularGrammarMatching.can_generate_sentence(
        context_sensitive_grammar, "aaabbabaaa")
  assert not ElixirRegularGrammarMatching.can_generate_sentence(
        context_sensitive_grammar, "aabaaa")
  context_sensitive_grammar = {
    "ab",
    [{"A", "B"}, {"B", "A"}, {"A", "a"}],
    "A"
  }
  assert ElixirRegularGrammarMatching.can_generate_sentence(
        context_sensitive_grammar, "a")
  assert not ElixirRegularGrammarMatching.can_generate_sentence(
        context_sensitive_grammar, "aaa")
end
```

REFERENCES

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