2º Exercício Programa – Lógica Computacional

Bruno Vasconcelos e Gustavo Wang

I. Introdução

Enunciado: *O objetivo didático desta atividade é experimentar concretamente os conceitos desenvolvidos em sala de aula a respeito de gramáticas, cadeias, etc. Além disso, deve ser usada uma linguagem funcional como paradigma de linguagem de programação - a linguagem Elixir. O objetivo do exercício é implementar o algoritmo de reconhecimento de cadeias geradas por uma gramática de estrutura de frase recursiva, que foi definido em sala de aula - algoritmo para reconhecer cadeias a partir de gramáticas.*

No algoritmo um dos argumentos é a cadeia w a ser verificada, e o outro a gramática. Então, produz-se iterativamente um conjunto Ti contendo todas as formas sentenciais da gramática recursiva cujo comprimento l seja l ≤ |w|, até que o próximo conjunto Ti+1 = Ti . Se w ∈ Ti+1 então a cadeia é aceita (isto é, foi gerada pela gramática), senão é rejeitada. Para cumprir este objetivo sugere-se a seguinte seqüência de etapas de execução:

1. Construa uma função em Elixir que permita percorrer um conjunto recursivamente (recebido em uma lista) e, a cada chamada recursiva da sua função, retorne um dos elementos do conjunto. (Esta função deve ser chamada recursivamente)

2. Construa uma função em Elixir que permita a geração de cadeias (incluindo formas sentenciais e sentenças) em ordem de tamanho. A sua função deve receber as regras de uma gramática de estrutura de frase, recursiva, e deve solicitar um valor de tamanho para as cadeias. A partir destes dados a sua função deverá gerar todas as cadeias cujos tamanhos sejam menores ou iguais ao valor recebido. (use a função do item anterior como apoio)

3. Construa um sistema em Elixir que implemente o algoritmo de reconhecimento de cadeias a partir de uma determinada gramática. O seu sistema deverá utilizar a função desenvolvida no item anterior.

II. Descrição do projeto

No arquivo **ep2.exs** é apresentado o código com as funções pedidas pelo enunciado, contidas em um único módulo **Grammar** e os testes, contidos no módulo **GrammarTest**. Neste relatório, é apresentada a descrição da implementação do código e dos testes realizados.

Para executar o código, basta abrí-lo com Elixir. Quando o código é executado, ele realiza automaticamente os testes apresentados na seção IV. Para verificar se uma determinada cadeia pertence a uma gramática, basta digitar na linha de comando a função Grammar.checkGrammar(sigma, n, p, s, str).

Esta função recebe como entrada:

- A gramática **G**, que é representada por 4 argumentos:
  - **Σ**: um array de strings que representam os caracteres terminais. Ex.: [“a”, “b”, “c”]
  - **N**: um array de strings que represemtam os caracteres não-terminais. Ex.: [“S”, “A”, “B”]
  - **P**: o conjunto de regras de produção da gramática, representada por um array de pares de strings. Ex.: A gramática com as regras S -> Ab e A -> ab teria suas regras representadas por [{“S”, “Ab”}, {“A”, “ab”}]
  - **S**: um array de strings que representa o símbolo inicial. Ex.: [“S”]
- A cadeia str a ser verificada, representada por um string. Ex.: “abcc”

Por exemplo, vamos verificar se a cadeia “**aaabbbccc**” pertence à gramática mostrada em sala de aula,

G_4 = ({S, B, C}, {a, b, c}, {S -> aBC, S -> aSBC, CB -> BC, aB -> ab, bB -> bb, bC -> bc, cC -> cc}, S)

```
iex(1)>  Grammar.checkGrammar(["S", "B", "C"], ["a", "b", "c"], [{"S", "aBC"}, {"S", "aSBC"},
 {"CB", "BC"}, {"aB", "ab"}, {"bB", "bb"}, {"bC", "bc"}, {"cC", "cc"}], ["S"], "aaabbbccc")
true
```

Como foi demonstrado em aula, a gramática G_4 gera cadeias da forma $\mathbf{a^n b^n c^n}$. Portanto, o programa retorna true.

Caso seja fornecida uma cadeia que não possa ser formada pela gramática, o programa retorna false.

```
iex(2)>  Grammar.checkGrammar(["S", "B", "C"], ["a", "b", "c"], [{"S", "aBC"}, {"S", "aSBC"},
 {"CB", "BC"}, {"aB", "ab"}, {"bB", "bb"}, {"bC", "bc"}, {"cC", "cc"}], ["S"], "aaabbbcccc")
false
```

III. Implementação

Seguiu-se a sequência de etapas recomendada no enunciado para implementar o algoritmo. Foram utilizadas as funções associadas às variáveis de tipo String da linguagem Elixir.

1. Percorre

Para percorrer uma lista, foi implementada a função **Traverse**. Ela recebe uma lista e a percorre recursivamente.

```elixir
# TRAVERSE
# Percorre RECURSIVAMENTE uma lista
def traverse([h | t]) do
    IO.inspect(h)
    traverse(t)
end

def traverse([]) do
end
```

2. Gerar cadeias

Para gerar todas as cadeias cujos tamanhos sejam menores ou iguais a um valor recebido, foi necessário implementar algumas funções auxiliares.

- **onlyTerminals(s, n)**: essa função checa a lista de cadeias s e retorna true se todos os seus elementos são compostos apenas por caracteres terminais. Para isso, utiliza a função *String.contains?*.

```elixir
# ONLY_TERMINALS
# Checa se na lista de cadeias [h | t] há apenas terminais
def onlyTerminals([h | t], n) do
    if String.contains?(h, n) do
        false
    else
        onlyTerminals(t, n)
    end
end

def onlyTerminals([], _) do
    true
end
```

- **applyRules(p, n, [h | t], size, newStrings)**: aplica as regras de produção P recursivamente na lista de cadeias [h | t]. A cada chamada da função, adiciona à lista newStrings as novas cadeias criadas a partir dessas regras. Utiliza as funções **swapNTtoT**, apresentada abaixo, e **join**, que já foi utilizada no EP1.

```elixir
# APPLY_RULES
# Aplica as regras P na lista [h | t]
def applyRules(p, n, [h | t], size, newStrings) do
    applyRules(p, n, t, size, join(newStrings, swapNTtoT(h, n, p, size, [])))
end

def applyRules(_, _, [], _, newStrings) do
    newStrings
end
```

- **swapNTtoT(str, n, [h | t], size, newStr)**: efetua as regras de produção contidas na lista [h | t] na cadeia str. A função checa inicialmente se a cadeia contém apenas terminais. Em caso positivo, retorna a cadeia. Em seguida, checa se o elemento a ser adicionado tem tamanho menor do que o tamanho definido. Em caso positivo, efetua adiciona o resultado da troca na lista newStr. Caso contrário, vai para a próxima regra sem adicionar nenhum elemento à lista.

Para aplicar a regra, trocando os caracteres, utiliza a função *String.replace*. Para checar o tamanho das cadeias, utiliza a função *String.length*.

```
# SWAP_NT_TO_T
# Percorre a cadeia str
# Troca não terminais por terminais,
# levando em conta o tamanho máximo size permitido das cadeias
def swapNTtoT(str, n, [h | t], size, newStr) do
    cond do
        String.contains?(str, n) == false -> # Checa se a cadeia é composta apenas por terminais. Em caso positivo, retorna a cadei
            [str]
        String.length(str) - String.length(elem(h, 0)) + String.length(elem(h, 1)) < size and String.contains?(str, elem(h, 0)) ->
            swapNTtoT(str, n, t, size, newStr ++ [String.replace(str, elem(h, 0), elem(h, 1))])
        true -> # Senão, vai para a próxima regra
            swapNTtoT(str, n, t, size, newStr)
    end
end
```

Com essas funções, pôde-se definir a função **generate(sigma, n, p, s, size)**. Essa função recebe a gramática G e o tamanho máximo das cadeias a serem criadas. Retorna uma lista de strings com todas as cadeias que respeitam essas condições.

```
# GENERATE
# Gera todos as cadeias de tamanho menor que size.
def generate(sigma, n, p, s, size) do
    if onlyTerminals(s, n) do
        s
    else
        generate(sigma, n, p, applyRules(p, n, s, size, []), size)
    end
end
```

3. Reconhecimento de cadeia

Enfim definimos a função **checkGrammar**. Como explicado na seção II, ela recebe uma cadeia e analisa se ela pode ser gerada pela gramática que também recebe como parâmetro.

Para isso, cria-se, com a função **generate**, um array com todas as cadeias produzidas pela gramática de tamanho menor ou igual à cadeia recebida como parâmetro. Em seguida, utilizando a expressão *in*, verifica se a cadeia recebida está contida nesse array. Em caso positivo, retorna true. Caso contrário, retorna false.

```
# CHECK_GRAMMAR
# Checa se a cadeia pode ser criada pela gramática definida
def checkGrammar(n, sigma, p, s, str) do
    if str in generate(sigma, n, p, s, String.length(str) + 1) do
        true
    else
        false
    end
end
```

IV. Testes

Foram realizados testes para verificar o bom funcionamento da função CheckGrammar.. As declarações de teste estão localizadas na segunda parte do arquivo ep2.exs, no módulo **GrammarTest**.

Foram utilizados os exemplos de gramática apresentados em sala de aula:

- **G_1 = ({S}, {a, b}, {S -> ab, S -> abS}, S)**

  cujas cadeias geradas são da forma ab(ab)*

- **G_2 = ({S}, {a, b}, {S -> ab, S -> aSb}, S)**

  cujas cadeias geradas são da forma $a^n b^n$, n > 0

- **G_3 = ({S, B}, {a, b}, {S -> aB, B -> b, B -> Sb}, S)**

  cujas cadeias geradas são da forma $a^n b^n$, n > 0

- **G_4 = ({S, B, C}, {a, b, c}, {S -> aBC, S -> aSBC, CB -> BC, aB -> ab, bB -> bb, bC -> bc, cC -> cc}, S)**

  cujas cadeias geradas são da forma $a^n b^n c^n$, n > 0

As saídas obtidas pelo algoritmo foram comparadas com aquelas demonstradas em sala. A função obteve os resultados esperados, como mostra a figura abaixo.

```
Finished in 0.1 seconds
15 tests, 0 failures

Randomized with seed 138000
Interactive Elixir (1.8.1) - press Ctrl+C to exit (type h() ENTER for help)
```

V. Conclusão

Este segundo exercício programa nos permitiu desenvolver nossos conhecimentos em linguagem funcional e recursão utilizando a linguagem Elixir para realizar um exercício que já havíamos feito manualmente em sala de aula. Tivemos um primeiro contato com as bibliotecas de funções dessa linguagem, em especial as funções de *String*.

VI. Referências

https://hexdocs.pm/elixir/String.html

https://elixir-lang.org/getting-started/case-cond-and-if.html