PCS3556 - Lógica Computacional

2º Exercício-Programa
Introdução à Programação Funcional

Professor Doutor Ricardo Rocha

9783640 - Luís Henrique Barroso Oliveira

9835623 - Rodrigo Vali Cebrian

11259715 - Vanderson da Silva dos Santos

São Paulo, 26 de Fevereiro de 2023

SUMÁRIO

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 - uma das linguagens Elixir ou Clojure.

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 ` seja ` ≤ |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 ou Clojure 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 ou Clojure 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 ou Clojure 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.

Discussão sobre o problema e solução proposta

Pode-se descrever uma dada gramática da seguinte maneira: $G = (N, T, P, S)$. Em que cada um dos parâmetros se definem como:

- N é o conjunto de símbolos não terminais
- T é o conjunto de símbolos terminais
- P se trata do conjunto com as leis de formação da gramática, dadas no formato $\alpha \rightarrow \beta$
- S é um elemento tal que $S \in N$ e S é o símbolo que dá início à geração de sentenças seguindo as regras estabelecidas por P

Valendo-se da definição de gramática exposta acima, e tendo em vista a sugestão de implementação da solução expressa no enunciado, construiu-se um algoritmo que opera da seguinte forma:

A função $CheckIfChainIsAcceped$ recebe como parâmetros uma cadeia ω e a regra gramatical G, e retorna um valor booleano que representa se tal cadeia se enquadra nas regras de formação da gramática em questão.

Para validar se ω pode ser gerada por G, a função verifica se a cadeia está presente em um vetor que contém todas as possibilidades de cadeias, dado um tamanho máximo de caracteres. O tamanho máximo de caracteres considerados é dado pelo número de caracteres de ω. E, para gerar todas as possíveis cadeias dado um tamanho máximo, é utilizada a função $GenerateAllPossibleChains$, que, por sua vez, recebe como parâmetros uma gramática e um tamanho máximo para as cadeias geradas.

A implementação de $CheckIfChainIsAcceped$ está descrita abaixo:

```
(defn CheckIfChainIsAcceped [rules chain]
      (let [max_size (count chain)]
            (println "max_size (1): " max_size)
            (def all_possibilities (GenerateAllPossibleChains rules max_size))
            (def result (contains? (set all_possibilities) chain) )
      )
      result
)
```

A função $GenerateAllPossibleChains$ funciona da seguinte maneira:

A função utiliza recursão para gerar todas as possíveis cadeias geradas pela gramática, começando com a cadeia inicial S. A função é chamada com um conjunto inicial de cadeias geradas e é atualizada em cada iteração adicionando novas cadeias obtiddas a partir da gramática.

A definição inicial da função tem dois argumentos: $rules$ e max_size. O argumento $rules$ é uma lista de tuplas que especificam as regras de produção a serem aplicadas. Cada tupla representa uma função do tipo $\alpha \rightarrow \beta$, que definem as leis de formação da gramática. O argumento max_size especifica o tamanho máximo permitido para as cadeias geradas.

A segunda definição da função tem três argumentos: $rules$, max_size e $index$. O argumento $index$ é um número inteiro que especifica qual conjunto de cadeias geradas está sendo considerado naquela iteração.

A função utiliza a função auxiliar $GetApplyRulesInChain$, que aplica as regras de produção às cadeias de caracteres. A função $filter$ é usada para limitar o tamanho máximo das cadeias geradas e evitar cadeias duplicadas. A função vec é usada para converter o resultado em um vetor.

A função retorna todas as cadeias geradas pelas regras de produção especificadas nas regras de entrada, com um tamanho máximo permitido.

A implementação de $GenerateAllPossibleChains$ é dada a seguir:

```
(defn GenerateAllPossibleChains
      ([rules, max_size] (GenerateAllPossibleChains rules, max_size, 0, [["S"]]))
      ([rules, max_size, index, generated_chains]
            (if (= index (count generated_chains))
                  generated_chains
                  (do
                        (def chain_applied_changes
                              (GetApplyRulesInChair
                                    rules
                                    (get generated_chains index)
                              )
                        )
                        (def new_generated_chain
                              (vec
                                    (filter
                                          #(not (contains? (set generated_chains) %))
                                          (filter
                                                #(<= (count %) max_size)
                                                chain_applied_changes
                                          )
                                    )
                              )
                        )
                        (def new_possible_chair (vec (concat generated_chains new_generated_chain)))
                        (GenerateAllPossibleChains rules, max_size, (+ index 1), new_possible_chair)
                  )
            )
      )
)
```

A função $GetApplyRulesInChain$ funciona da seguinte forma:

A função aplica as regras descritas na gramática a cada elemento de uma cadeia de caracteres e retorna uma lista de todas as cadeias geradas.

A função utiliza recursão para aplicar as regras a cada elemento da cadeia de caracteres, começando pelo primeiro elemento. É chamada com um conjunto inicial de cadeias geradas e é atualizada em cada iteração adicionando novas cadeias geradas pela aplicação das regras.

A primeira definição da função tem dois argumentos: $rules$ e $chain$. O argumento $rules$ é uma lista de tuplas que especificam as regras a serem aplicadas.. O argumento $chain$ é a cadeia de caracteres na qual as regras de produção serão aplicadas.

A segunda definição da função tem três argumentos: $rules$, $chain$ e $index$. O argumento $index$ é um número inteiro que especifica qual elemento da cadeia está sendo considerado naquela iteração.

A terceira definição da função tem quatro argumentos: $rules$, $chain$, $index$ e $chain\_applied$. O argumento $chain\_applied$ é uma lista das cadeias geradas pelas regras aplicadas.

Se utiliza a função auxiliar $GetApplyRuleInElement$, que aplica as regras a um único elemento da cadeia de caracteres. A função map é usada para aplicar as regras de a cada elemento da cadeia de caracteres. A função vec é usada para converter o resultado em um vetor.

A função retorna uma lista de todas as cadeias geradas a partir de cada elemento da cadeia de caracteres.

A implementação da função $GetApplyRulesInChain$ é expressa a seguir:

```
(defn GetApplyRulesInChair
      ([rules, chain] (GetApplyRulesInChair rules, chain, 0, []))
      ([rules, chain, index] (GetApplyRulesInChair rules, chain, index, []))
      ([rules, chain, index, chair_applied]

            (if (= index (count chain))
                  chair_applied
                  (do
                        (def curr_elem (chain index))

                        (def tranf_curr_elem (GetApplyRuleInElement rules curr_elem) )

                        (def new_transf
                              (vec
                                    (map
                                          #(vec (concat (subvec chain 0 index) % (subvec chain (inc index))) )
                                          tranf_curr_elem
                                    )
                              )
                        )
                        (def new_chair_applied (vec (concat chair_applied new_transf)))

                        (GetApplyRulesInChair rules chain (+ index 1) new_chair_applied)
                  )
            )
      )
)
```

A função $GetApplyRuleInElement$ funciona da seguinte forma:

A função recebe dois argumentos: uma lista de regras $rules$ e um elemento $elem$ e retorna uma lista de valores que correspondem a $elem$ na lista de regras.

A implementação de $GetApplyRuleInElement$ está expressa a seguir:

```
(defn GetApplyRuleInElement [rules elem]
      (do
            (def values [])
            (def indexes (range (count rules)))
            (doseq [index indexes]
                  (do
                        (def rule (GetGramRule rules index))
                        (if (= (rule "simbol") elem)
                              (def values (clojure.set/union values (vector (rule "value")) ))
                        )
                  )
            )
      )
      values
)
```

Por fim, a função $GetGramRule$ tem por recebe dois argumentos: uma lista de regras $rules$ e uma posição pos e retorna um mapa contendo o símbolo e os valores da regra na posição especificada.

```
(defn GetGramRule [rules pos]
      (let
            [simbol (get (get rules pos) 0)
             simbol_values (get (get rules pos) 1)
             result {"simbol" simbol, "value" simbol_values } ]

            result
      )
)
```

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

```
(deftest GetGramRuleTest []
  (let [
      rules_test [
        [ "S" ["a", "A", "S"]  ]
        [ "A" ["a"]            ]
      ]

      value_result (ep2.core/GetGramRule rules_test 0)

      value_result1 (ep2.core/GetGramRule rules_test 1)
    ]

    (testing "Testando a função GetGramRule"
      (is (= value_result {"simbol" "S", "value" ["a", "A", "S"] }))
      (is (= (value_result "simbol") "S" ))
      (is (= (value_result "value" ) ["a", "A", "S"] ))
      (is (= value_result1 {"simbol" "A", "value" ["a"]} ))
    )
  )
)
```

```
(deftest GetApplyRuleInElementTest []
  (let [
      rules_test0 [
        [ "S" ["a", "A", "S"]  ]
        [ "S" ["a"]            ]
        [ "A" ["b" "a"]        ]
      ]

      value_result0 (ep2.core/GetApplyRuleInElement rules_test0 "S")
      value_result1 (ep2.core/GetApplyRuleInElement rules_test0 "A")
      value_result2 (ep2.core/GetApplyRuleInElement rules_test0 "a")
    ]

    (testing "Testando a função GetApplyRuleInElement"
      (is (= value_result0 [["a", "A", "S"] ["a"]] ))
      (is (= value_result1 [["b" "a"]] ))
      (is (= value_result2 [] ))
    )
  )
)
```

```
(deftest GetApplyRulesInChainTest []
  (let [
      rules_test0 [
        [ "S" ["a", "A", "S"]  ]
        [ "S" ["a"]            ]
        [ "A" ["b" "a"]        ]
      ]

      rules_test1 [
        [ "S" ["a", "A", "S"]  ]
        [ "S" ["a"]            ]
      ]

      value_result0 (ep2.core/GetApplyRulesInChain rules_test0 ["S", "A"])
      value_result1 (ep2.core/GetApplyRulesInChain rules_test1 ["a", "A", "S"])
    ]

    (testing "Testando a função GetApplyRulesInChain"
      (is (= value_result0 [["a", "A", "S", "A"], ["a", "A"], ["S", "b", "a"]]  ))
      (is (= value_result1 [["a", "A", "a", "A", "S"], ["a", "A", "a"]]  ))
    )
  )
)
```

```clojure
(deftest GenerateAllPossibleChainsTest []
  (let [
      rules_test0 [
        [ "S" ["a"]           ]
      ]

      rules_test1 [
        [ "S" ["a", "A", "S"]  ]
        [ "S" ["a"]           ]
      ]

      value_result0 (ep2.core/GenerateAllPossibleChains rules_test0 2)
      value_result1 (ep2.core/GenerateAllPossibleChains rules_test1 3)
    ]

    (testing "Testando a função GenerateAllPossibleChains"
      (is (= value_result0 [["S"], ["a"]]  ))
      (is (= value_result1 [["S"], ["a", "A", "S"], ["a"], ["a", "A", "a"]]  ))
    )
  )
)
```

```
(deftest CheckIfChainIsAccepedTest []
  (let [
      rules_test0 [
        [ "S" ["a", "A", "S"]  ]
        [ "S" ["a"]            ]
        [ "S" ["S", "S"]       ]
        [ "A" ["b", "a"]       ]
        [ "A" ["S", "S"]       ]
      ]

      chain_to_be_recognized0 ["a", "a"]
      chain_to_be_recognized1 ["a", "a","a","a"]
      chain_to_be_recognized2 ["b", "b"]

      value_result0 (ep2.core/CheckIfChainIsAcceped rules_test0 chain_to_be_recognized0)
      value_result1 (ep2.core/CheckIfChainIsAcceped rules_test0 chain_to_be_recognized1)
      value_result2 (ep2.core/CheckIfChainIsAcceped rules_test0 chain_to_be_recognized2)
    ]

    (testing "Testando a função CheckIfChainIsAcceped"
      (is value_result0)
      (is value_result1)
      (is (not value_result2))
    )
  )
)
```

O resultado de todos foi positivo. De todos os 5 teste com as 14 verificações, todas passaram com sucesso. Esse resultado pode ser visto abaixo:

```
root@9566100f04ef:/usr/app/src/ep2# lein test

lein test ep2.core-test

Ran 5 tests containing 14 assertions.
0 failures, 0 errors.
```

Teste Completo

Por fim, Com todas as funções verificar, o algoritmo príncipal foi executado para algumas situações. Nessa situação, vale ressaltar que as regras gramaticais usadas em todos os testes foi a seguinte:

"S" -> “aAS”
"S" -> “a”
"S" -> “SS”

"A" -> "ba"
"A" -> "SS"

1. Teste com w = "aaaa"

```
root@9566100f04ef:/usr/app/src/ep2# lein run
Regras gramaticais:  [[S [a A S]] [S [a]] [S [S S]] [A [b a]] [A [S S]]]
Cadeia para ser reconhecida (w):  [a a a a]
max_size (l):  4
Cadeia (w) foi Aceita :)
```

2. Teste com w = "aa"

```
root@9566100f04ef:/usr/app/src/ep2# lein run
Regras gramaticais:  [[S [a A S]] [S [a]] [S [S S]] [A [b a]] [A [S S]]]
Cadeia para ser reconhecida (w):  [a a]
max_size (l):  2
Cadeia (w) foi Aceita :)
```

3. Teste com w = "ba"

```
root@9566100f04ef:/usr/app/src/ep2# lein run
Regras gramaticais:  [[S [a A S]] [S [a]] [S [S S]] [A [b a]] [A [S S]]]
Cadeia para ser reconhecida (w):  [b a]
max_size (l):  2
Cadeia (w) foi Rejeitada :(
```

Conclusão

Essa atividade teve complexidade consideravelmente maior que a desenvolvida no primeiro EP da disciplina. Foi necessário desenvolver uma série de funções, a serem chamadas de forma encadeada, com o objetivo de gerar todas as possíveis combinações de cadeias para uma determinada gramática.

O maior nível de complexidade e abstração das funções motivou o desenvolvimento de testes unitários, para validar o comportamento de forma isolada e identificar os erros com maior facilidade.

O algoritmo implementado se assemelha muito ao raciocínio desenvolvido nas atividades em aula para solução dos problemas envolvendo gramáticas. Nesse sentido, não houbе grandes dificuldades no entendimento do escopo do exercício, e o maior esforço foi direcionado na codificação de um algoritmo previamente compreendido de forma intuitiva. Ou seja, foi essencialmente uma atividade de construir um programa capaz de reproduzir uma rotina que já fora executada manualmente em aula.

Anexo A - Código Principal

```
;; Definition of namespace ------------
(ns ep2.core
    (:gen-class)
    (:require [clojure.set :as set]))

;; Basic functions --------------------

(defn GetGramRule [rules pos]
    (let
        [simbol (get (get rules pos) 0)
         simbol_values (get (get rules pos) 1)
         result {"simbol" simbol, "value" simbol_values } ]

        result
    )
)

;; Chain recognition functions --------------------

(defn GetApplyRuleInElement [rules elem]
    (do
        (def values [])
        (def indexes (range (count rules)))
        (doseq [index indexes]
            (do
                (def rule (GetGramRule rules index))
                (if (= (rule "simbol") elem)
                    (def values (clojure.set/union values (vector (rule "value")) ))
                )
            )
        )
    )
    values
)
```

```
(defn GetApplyRulesInChain
    ([rules, chain] (GetApplyRulesInChain rules, chain, 0, []))
    ([rules, chain, index] (GetApplyRulesInChain rules, chain, index, []))
    ([rules, chain, index, chain_applied]

        (if (= index (count chain))
            chain_applied
            (do
                (def curr_elem (chain index))

                (def tranf_curr_elem (GetApplyRuleInElement rules curr_elem) )

                (def new_transf
                    (vec
                        (map
                            #(vec (concat (subvec chain 0 index) % (subvec chain (inc
index))) )
                            tranf_curr_elem
                        )
                    )
                )
                (def new_chain_applied (vec (concat chain_applied new_transf)))

                (GetApplyRulesInChain rules chain (+ index 1) new_chain_applied)
            )
        )
    )
)

(defn GenerateAllPossibleChains
    ([rules, max_size] (GenerateAllPossibleChains rules, max_size, 0, [["S"]]))
    ([rules, max_size, index, generated_chains]
        (if (= index (count generated_chains))
            generated_chains
            (do
                (def chain_applied_changes
                    (GetApplyRulesInChain
                        rules
```

```
                    (get generated_chains index)
                )
            )
            (def new_generated_chain
                (vec
                    (filter
                        #(not (contains? (set generated_chains) %))
                        (filter
                            #(<= (count %) max_size)
                            chain_applied_changes
                        )
                    )
                )
            )
            (def new_possible_chain (vec (concat generated_chains
new_generated_chain)))
            (GenerateAllPossibleChains rules, max_size, (+ index 1),
new_possible_chain)
        )
    )
)
)

(defn CheckIfChainIsAcceped [rules chain]
    (let [max_size (count chain)]
        (println "max_size (l): " max_size)
        (def all_possibilities (GenerateAllPossibleChains rules max_size))
        (def result (contains? (set all_possibilities) chain) )
    )
    result
)

;; Main function ----------------------
(defn -main []
    (def gram_rules [
            [ "S" ["a", "A", "S"] ]
            [ "S" ["a"]          ]
            [ "S" ["S", "S"]     ]
```

```
            [ "A" ["b", "a"]      ]
            [ "A" ["S", "S"]      ]
        ]
    )
    (println "Regras gramaticais: " gram_rules)

    (def chain_to_be_recognized
        ["b", "a"]
    )
    (println "Cadeia para ser reconhecida (w): " chain_to_be_recognized)

    (def chain_is_acceped
        (CheckIfChainIsAcceped gram_rules chain_to_be_recognized)
    )

    (if chain_is_acceped
        (println "Cadeia (w) foi Aceita :)")
        (println "Cadeia (w) foi Rejeitada :(")
    )
)
```

Anexo B - Código De Testes

```
(ns ep2.core-test
 (:require [clojure.test :refer :all]
           [ep2.core :refer :all]
  )
)

(deftest GetGramRuleTest []
 (let [
    rules_test [
     [ "S" ["a", "A", "S"]  ]
     [ "A" ["a"]            ]
    ]

    value_result (ep2.core/GetGramRule rules_test 0)

    value_result1 (ep2.core/GetGramRule rules_test 1)
  ]
   (testing "Testando a função GetGramRule"
    (is (= value_result {"simbol" "S", "value" ["a", "A", "S"] }))
    (is (= (value_result "simbol") "S" ))
    (is (= (value_result "value" ) ["a", "A", "S"] ))
    (is (= value_result1 {"simbol" "A", "value" ["a"]} ))
  )
 )
)



(deftest GetApplyRuleInElementTest []
 (let [
    rules_test0 [
     [ "S" ["a", "A", "S"]  ]
     [ "S" ["a"]            ]
     [ "A" ["b" "a"]        ]
    ]
```

```
    value_result0 (ep2.core/GetApplyRuleInElement rules_test0 "S")
    value_result1 (ep2.core/GetApplyRuleInElement rules_test0 "A")
    value_result2 (ep2.core/GetApplyRuleInElement rules_test0 "a")
  ]
   (testing "Testando a função GetApplyRuleInElement"
    (is (= value_result0 [["a", "A", "S"] ["a"]] ))
    (is (= value_result1 [["b" "a"]] ))
    (is (= value_result2 [] ))
  )
 )
)

(deftest GetApplyRulesInChainTest []
 (let [
    rules_test0 [
     [ "S" ["a", "A", "S"] ]
     [ "S" ["a"]          ]
     [ "A" ["b" "a"]      ]
    ]

    rules_test1 [
     [ "S" ["a", "A", "S"] ]
     [ "S" ["a"]          ]
    ]

    value_result0 (ep2.core/GetApplyRulesInChain rules_test0 ["S", "A"])
    value_result1 (ep2.core/GetApplyRulesInChain rules_test1 ["a", "A", "S"])
  ]
   (testing "Testando a função GetApplyRulesInChain"
    (is (= value_result0 [["a", "A", "S", "A"], ["a", "A"], ["S", "b", "a"]] ))
    (is (= value_result1 [["a", "A", "a", "A", "S"], ["a", "A", "a"]] ))
  )
 )
)

(deftest GenerateAllPossibleChainsTest []
 (let [
    rules_test0 [
```

```
    [ "S" ["a"]          ]
   ]

   rules_test1 [
    [ "S" ["a", "A", "S"] ]
    [ "S" ["a"]          ]
   ]

   value_result0 (ep2.core/GenerateAllPossibleChains rules_test0 2)
   value_result1 (ep2.core/GenerateAllPossibleChains rules_test1 3)
  ]
  (testing "Testando a função GenerateAllPossibleChains"
   (is (= value_result0 [["S"], ["a"]] ))
   (is (= value_result1 [["S"], ["a", "A", "S"], ["a"], ["a", "A", "a"]] ))
  )
 )
)

(deftest CheckIfChainIsAccepedTest []
 (let [
   rules_test0 [
    [ "S" ["a", "A", "S"] ]
    [ "S" ["a"]          ]
    [ "S" ["S", "S"]     ]
    [ "A" ["b", "a"]     ]
    [ "A" ["S", "S"]     ]
   ]

   chain_to_be_recognized0 ["a", "a"]
   chain_to_be_recognized1 ["a", "a","a","a"]
   chain_to_be_recognized2 ["b", "b"]

   value_result0 (ep2.core/CheckIfChainIsAcceped rules_test0
chain_to_be_recognized0)
   value_result1 (ep2.core/CheckIfChainIsAcceped rules_test0
chain_to_be_recognized1)
   value_result2 (ep2.core/CheckIfChainIsAcceped rules_test0
chain_to_be_recognized2)
```

```
  ]
   (testing "Testando a função CheckIfChainIsAcceped"
    (is value_result0)
    (is value_result1)
    (is (not value_result2))
   )
 )
)
```