ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

PCS3556 – Lógica Computacional

CAIO VINICIUS SOARES AMARAL - NUSP: 10706083

THALES AUGUSTO SOUTO RODRIGUEZ - NUSP: 10706110

GUSTAVO PRIETO – NUSP 4581945

PROF. DR. Ricardo Rocha

Exercício de Programação 2

São Paulo

Introdução

O objetivo deste exercício de programação é implementar o algoritmo de reconhecimento de cadeias geradas por uma gramática. O algoritmo desenvolvido utiliza o algoritmo recursivo desenvolvido em aula, que produz iterativamente um conjunto ti contendo todas as formas sentenciais da gramática.

O repositório do projeto pode ser encontrado em: https://github.com/master0022/ep2

Algoritmo

O código desenvolvido apresenta um módulo chamado **Reconhecedor**, que apresenta seis funções. As funções são divididas de forma que cada função chama uma outra função para ser aplicada a um grupo de elementos.

Os termos “palavras” usados no código representam cadeias de símbolos (por exemplo, “aabcAASa”), e regras se tratam de regras de produção G (por exemplo, “S->aS”).

Assim, grupos de palavras e regras são representados como:
Palavras = [[“aabvAS”], [palavra2], [palavra3] ...]
Regras = [[“S”,”aS”], [“A”,”as”], [regra3], [regra4], ...]

Agora, descrevendo as funções:

usa_regra(palavra,regra, lista\\[],character_idx\\0):

Aplica uma única regra de produção, a uma única palavra, **do máximo de formas distintas possíveis**.

Assim, caso os inputs sejam “aaAAaa” e A->x, o output seria:
[[aaxAaa], [aaAxaa]]

Para obter todas as combinações possíveis, esta função começa separando a palavra em 2 partes (“aaAAaa” -> “aaA” , “Aaa”), e aplicando a regra uma vez no primeiro elemento em que for possível no lado direito. Uma palavra de tamanho n é separada e possui a regra aplicada n vezes.

aplica_regras(palavra,regras,idx_regra\\0,lista\\[])

Aplica a função “usa_regra” várias vezes para uma mesma palavra, usando todas as regras possíveis. Retorna uma lista com todas as palavras que podem ser derivadas a partir de uma palavra, dada um conjunto de regras.

aumenta_conjunto(palavras,regras,idx_palavra\\0,conjunto\\[])

Esta função aplica “aplica_regras” várias vezes, a cada palavra recebida no input. Usualmente, o input **“palavras”** representa o conjunto T_i do algoritmo recursivo visto em aula, e retorna T_{i+1}.

O único detalhe é que este conjunto ainda não está “filtrado” para eliminar as palavras com tamanho maior que |w|.

calcula_conjunto(palavras,regras,w,iteracao\\0)
Esta função aplica “aumenta conjunto” várias vezes. Isto é equivalente a partir de um conjunto T_i até o conjunto $T_{|w+1|,}$ com |w| = **w** = tamanho da palavra que estamos verificando se pode ser gerada.

Nesta função é aplicada também **reduzir_tamanho(lista,tamanho_maximo)**, uma função auxiliar que elimina palavras maiores que |w| de uma lista de palavras, efetivamente filtrando T_i.

checa_palavra_em_gramatica(palavra,regras,inicio\\["S"])
Finalmente, esta função verifica se uma palavra pertence ao conjunto Ti+1 gerado por **calcula_conjunto**. Por padrão, ela assume que a cadeia inicial da gramática é “S”, mas isso pode ser alterado.

Em resumo, partindo do nível mais alto para o mais baixo:
Checa_palavra_em_gramatica: Função principal, verifica se a partir de uma cadeia inicial ([“S”] geralmente) é possível gerar uma palavra.
Calcula conjunto: Aplica várias regras a várias palavras, n vezes (T_0 ->T_{n+1})
Aumenta_conjunto: Aplica várias regras a várias palavras (T_i -> T_{i+1})
Aplica_regras: Aplica várias regras a uma palavra
Usa_regra: Aplica uma regra a uma palavra

E **reduzir_tamanho** que é aplicada em **calcula_conjunto**, que elimina palavras maiores que |w+1|.

Casos de teste

Para rodar os casos de teste criados, basta entrar na pasta do projeto elixir e executar

mix test

Ou, caso seja desejado usar as funções com casos novos, elas estão disponíveis na pasta /lib, no arquivo r**econhecedor.ex**

Os casos de teste gerados foram os seguintes, e podem ser encontrados em /test/reconhecedor_test.exs:

testes checa_palavra_em_gramatica:

```
input1 =  "aaa"
    g1 = [["A","S"], ["A","aS"], ["S","a"], ["S","aA"]]
    # resultado esperado: cadeia aceita (A->aS->aaA->aaS->aaa)

    input2 =  "bbab"
    g2 = [["S","aAa"], ["A","a"], ["A","ba"]]
    # resultado esperado: cadeia rejeitada

    input3 =  "aSb"
    g3 = [["S","aA"], ["S","A"], ["A","S"], ["A","Ab"]]
    # resultado esperado: cadeia aceita (S->aA->aAb->aSb)

    input4 = "aabab"
    g4 = [["A","S"], ["A","aS"], ["A","abS"], ["S","A"], ["S","aS"],
["S","b"]]
    # resultado esperado: cadeia aceita (A->aS->aA->aabS->aabaS->aabab)

    input5 = "aaabaa"
    g5 = [["A","S"], ["A","aS"], ["A","abS"], ["S","A"], ["S","aS"]]
    # resultado esperado: cadeia rejeitada


    assert Reconhecedor.checa_palavra_em_gramatica(input1,g1) == true
    assert Reconhecedor.checa_palavra_em_gramatica(input2,g2) == false
    assert Reconhecedor.checa_palavra_em_gramatica(input3,g3) == true
    assert Reconhecedor.checa_palavra_em_gramatica(input4,g4) == true
    assert Reconhecedor.checa_palavra_em_gramatica(input5,g5) == false
```

testes usa regra:

```
    input1 =  "aaaaA"
    r1 = ["A","X"]
    resultado_esperado= Enum.sort(  ["aaaaA"],["aaaaX"]  )

    input2 =  "aaaAAAaaa"
    r2 = ["A","X"]
    resultado_esperado2=
Enum.sort([  "aaaXAAaaa","aaaAXAaaa","aaaAAXaaa","aaaAAAaaa"  ])

    input3 =  "aaSAAaaaSAa"
    r3 = ["aSA","YXZ"]
    resultado_esperado3=
Enum.sort([  "aYXZAaaaSAa","aaSAAaaYXZa","aaSAAaaaSAa"  ])

    assert Enum.sort(Reconhecedor.usa_regra(input1,r1)) == resultado_esperado
    assert Enum.sort(Reconhecedor.usa_regra(input2,r2)) == resultado_esperado2
    assert Enum.sort(Reconhecedor.usa_regra(input3,r3)) == resultado_esperado3
```

testes aplica_regras:

```
    input1 =  "aaaaA"
    r1 = [["A","X"],["aA","ZZ"]]
    resultado_esperado= Enum.sort(  [
      "aaaaA",
      "aaaaX",
      "aaaZZ",
      ]  )

    input2 =  "aaaAAAaaa"
    r2 = [["AA","XX"],["aaaA","BBBA"],["Aaaa","ABBB"]]
    resultado_esperado2= Enum.sort([
        "aaaXXAaaa",
        "aaaAXXaaa",
        "BBBAAAaaa",
        "aaaAAABBB",
        "aaaAAAaaa"  ])

    input3 =  "aaSAAaaaSAa"
    r3 = [["aSA","YXZ"],["S",""]]
    resultado_esperado3= Enum.sort([
      "aYXZAaaaSAa",
      "aaSAAaaYXZa",
      "aaAAaaaSAa",
      "aaSAAaaaAa",
      "aaSAAaaaSAa",
        ])

    assert Enum.sort(Reconhecedor.aplica_regras(input1,r1)) ==
resultado_esperado
    assert Enum.sort(Reconhecedor.aplica_regras(input2,r2)) ==
resultado_esperado2
    assert Enum.sort(Reconhecedor.aplica_regras(input3,r3)) ==
resultado_esperado3
```

testes aumenta conjunto:

```
    input1 =  ["aaA","bbB"]
    r1 = [["A","X"],["B","X"]]
    resultado_esperado= Enum.sort(  [
      "aaA",
      "aaX",

      "bbB",
      "bbX",
      ]  )

    input2 =  [["aaaAAAaaa"],["AA"],["AAvvvAAA"]]
    r2 = [["AA","XX"],["aaaA","BBBA"],["Aaaa","ABBB"]]
    resultado_esperado2= Enum.sort([
        "aaaXXAaaa",
        "aaaAXXaaa",
        "BBBAAAaaa",
        "aaaAAABBB",
        "aaaAAAaaa",

        "AA",
        "XX",

        "AAvvvAAA",
        "XXvvvAAA",
        "AAvvvXXA",
        "AAvvvAXX",
        ])

    assert Enum.sort(Reconhecedor.aumenta_conjunto(input1,r1)) ==
resultado_esperado
    assert Enum.sort(Reconhecedor.aumenta_conjunto(input2,r2)) ==
resultado_esperado2
```

testes calcula conjunto:

```
    input1 =  ["A"]
    r1 = [["A","aA"],["S","X"]]
    w1= 5 #Isso significa 5 iteracoes, de T0 = "A" ate T6
    resultado_esperado= Enum.sort([
      "A",
      "aA",
      "aaA",
      "aaaA",
      "aaaaA",

#     "aaaaaA",  Nao aparece pois tem tamanho = 5+1=6
#     "aaaaaaA", Nao aparece pois tem tamanho = 5+2=7
      ])

    assert Enum.sort(Reconhecedor.calcula_conjunto(input1,r1,w1)) ==
resultado_esperado
```