Arthur Andrade Ilario
NUSP: 10432515

Exercício Prático 4

PCS3556
04 de Abril de 2021

Introdução

Essa atividade coloca alguns dos conceitos explorados em sala de aula em prática. O objetivo é implementar um algoritmo de reconhecimento de linguagens livres de contexto, a partir da gramática em forma normal de Chomsky. Para isso será usado o algoritmo CYK apresentado durante as aulas e diversas funções para normalizar uma gramática livre de contexto qualquer.

O algoritmo CYK

Para realizar o reconhecimento de cadeias em tempo polinomial, foi usado o algoritmo de programação dinâmica. A ideia geral do algoritmo é quebrar o problema maior (a cadeia completa) em sub-problemas(sub-cadeias) para os quais já sabemos a resposta, e a partir daí construir a solução completa.

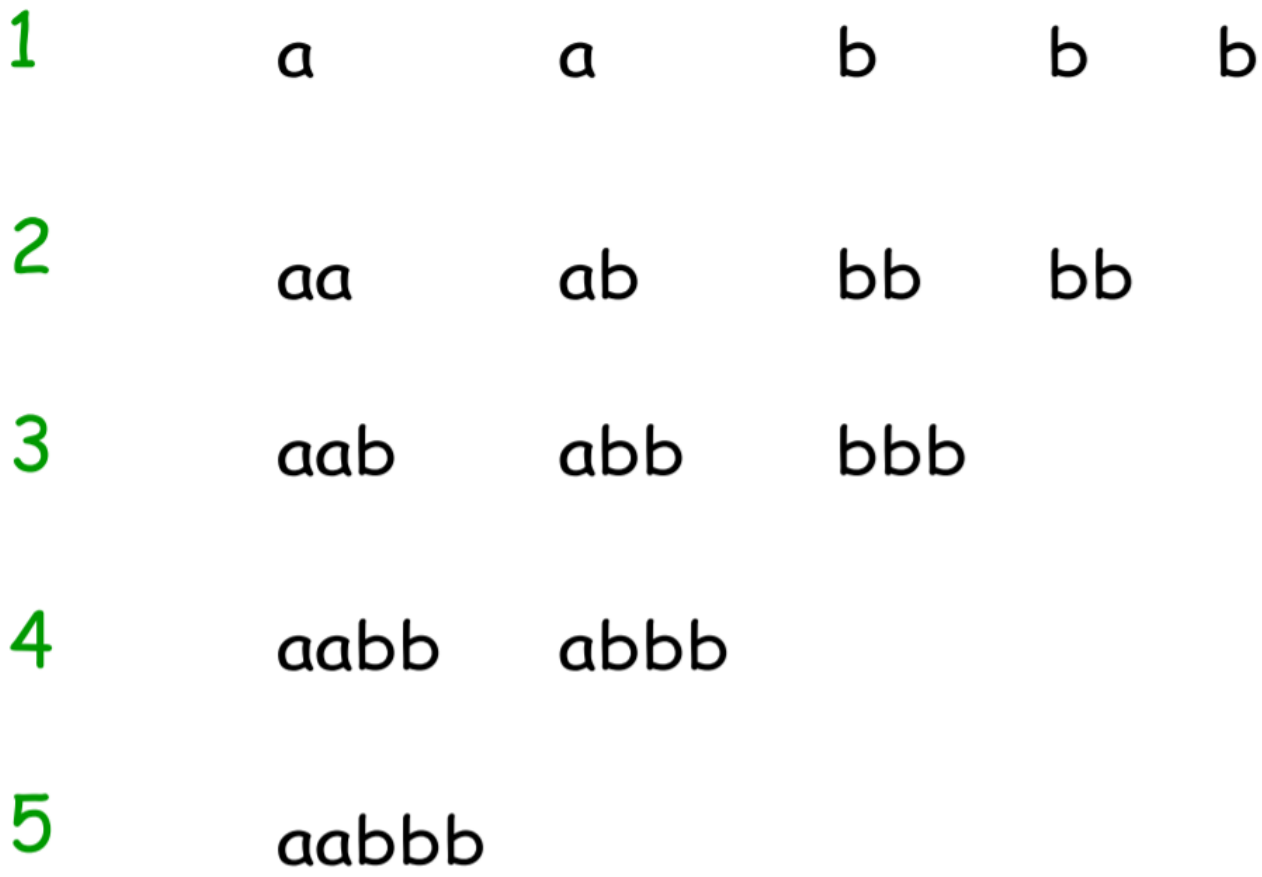

Figura 1: Divisão em sub-cadeias

Começando então com sub-cadeias de tamanho um e uma gramática na forma normal de

Chomsky, são gerados conjuntos contendo os símbolos não-terminais que derivam os terminais de cada sub-cadeia(figura 2).

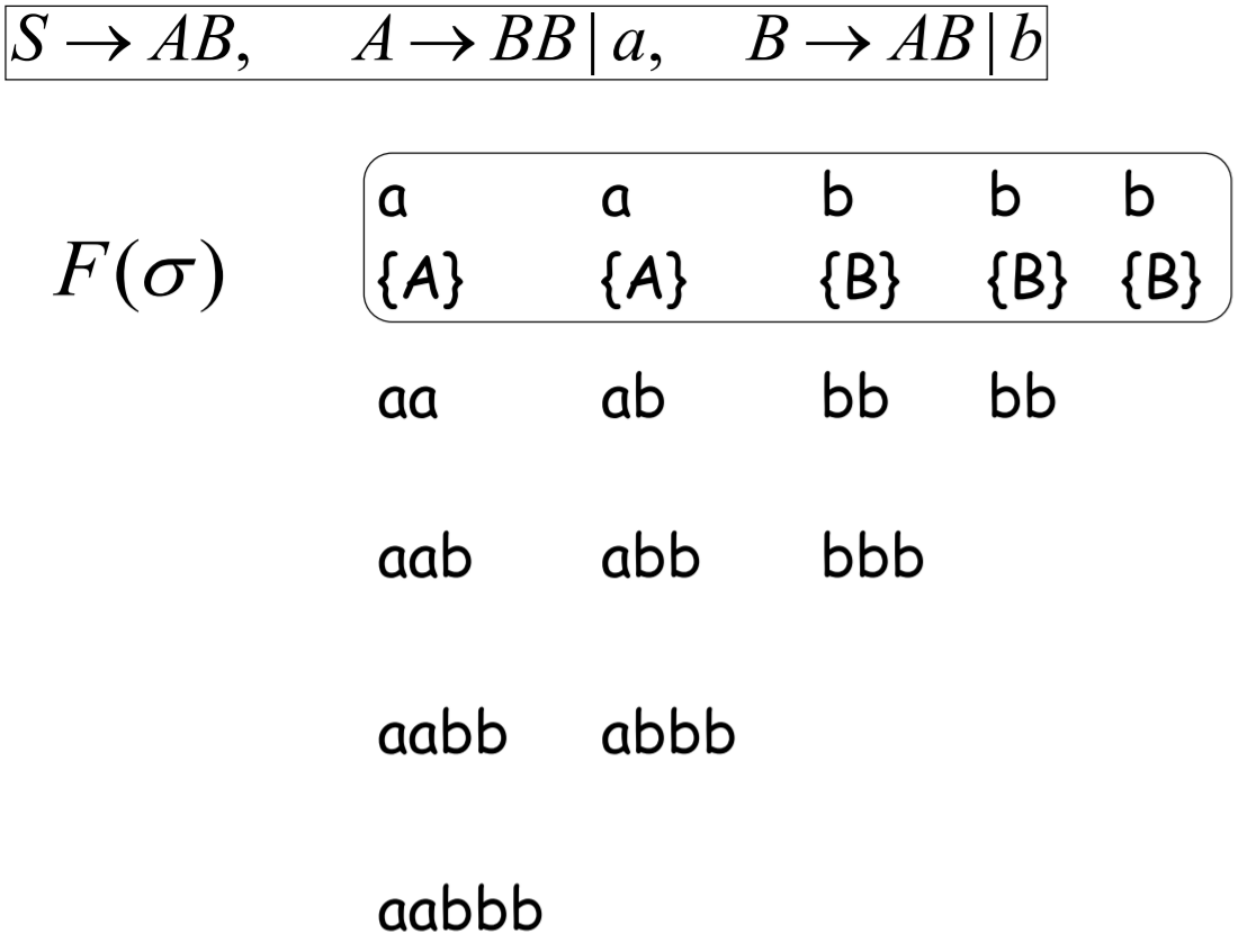

Figura 2: Solução para sub-cadeias de tamanho um

Em seguida, a cadeia é dividida em sub-cadeias de tamanho dois. Para cada sub-cadeia, dividimos ela em sub-cadeias de tamanho um, para as quais já temos a resposta! Verificamos então quais regras de derivação geram a união dos conjuntos resposta de cada sub-sub-cadeia, gerando um novo conjunto resposta para a sub-cadeia de tamanho dois.

$$S \to AB, \quad A \to BB \mid a, \quad B \to AB \mid b$$

$F(\sigma) =$

a	a	b	b	b
{A}	{A}	{B}	{B}	{B}

$F(\cdot) =$

aa	ab	bb	bb
{}	{S,B}	{A}	{A}

aab abb bbb

aabb abbb

aabbb

Figura 3: Solução para sub-cadeias de tamanho dois

O algoritmo segue aumentando o tamanho das sub-cadeias até que tenham o mesmo tamanho que a cadeia original. Quando chega nesse ponto, avaliamos se o conjunto de símbolos que derivam a cadeia original inclui o símbolo inicial da gramática. Se incluir, a cadeia é reconhecida, caso contrário, não é.

Sobre o projeto

Os arquivos estão organizados em 2 diretórios principais: *lib* e *test*. O código fonte para o programa se encontra dividido em três arquivos diferentes: */lib/normalizer.ex*, responsável por normalizar gramáticas livres de contexto para a forma normal de Chomsky, */lib/fecho.ex*, responsável por gerar o fecho transitivo de um conjunto de regras (criado no EP2 e usado no módulo Normalizer para consertar produções unitárias), e */lib/parser.ex*, módulo principal,

responsável por implementar o algoritmo CYK para reconhecer cadeias. A suíte de testes se encontra nos arquivo */test/normalizer_test.exs* e */test/parser_test.exs*.

1 Abordagem e implementação

Para realizar as tarefas computacionais propostas, foi utilizada a linguagem funcional Elixir (1.11.2).

Como citado nas seções anteriores, o programa possui 3 módulos diferentes, um responsável pela normalização da gramática, outra para auxiliar a normalização e um para implementar o reconhecimento de cadeias através do algoritmo CYK.

1.1 Normalizador

O Normalizador de gramática executa uma série de transformações na gramática de entrada que vão "consertando"ela progressivamente. As alterações executadas são:

1. **Modificação do símbolo inicial**: caso o símbolo inicial apareça do lado direito de alguma regra de derivação, ele é substituído por um novo símbolo, fazendo as alterações necessárias.

2. **Remoção de produções nulas**: regras de derivação da forma $X \to \epsilon$ são removidas, gerando novas ao remover X do lado direito de regras pré-existentes. Ex.: $S \to AXBXC$ vira $S \to ABXC|AXBC|ABC$

3. **Remoção de produções unitárias**: regras que possuem apenas um símbolo não-terminal do lado direito são removidas. Elas são substituídas pelo conjunto de derivações que o símbolo unitário "alcançava", e processo se repete até que não restem produções unitárias. Ex.: $A \to B, B \to BC, B \to b, C \to c$ vira $A \to b|BC, B \to b, C \to c$

4. **Remoção de regras longas**: regras com o lado direito maior que dois símbolos são divididas em novas regras de tamanho dois através de símbolos intermediários. Ex.: $S \to ABC$ vira $S \to AV_1, V_1 \to BC$

5. **Remoção de regras mistas**: regras que possuem símbolos terminais e não-terminais do lado direito (ou com dois terminais) tem os símbolos terminais substituídos por novos não-terminais que têm como única derivação o símbolo terminal. Ex.: $A \to aB$ vira $A \to T_aB, T_a \to a$

A função *normalize/1* recebe uma gramática livre de contexto qualquer e aplica as transformações listadas acima na mesma ordem.

```
1   @type symbol :: String.t() | atom
2   @type ruleset :: [{[symbol], [symbol]}]
3   @type grammar :: {[symbol], [symbol], ruleset, symbol}
4   @spec normalize(grammar) :: grammar
5
6   def normalize({non_terminal, terminal, ruleset, initial}) do
7     start_grammar(non_terminal, terminal, initial)
8     ruleset = ruleset |> fix_starter |> fix_nullable |> fix_unit |>
          fix_long |> fix_terminal
9     {non_terminal, terminal, initial} = get_grammar()
10    {non_terminal, terminal, ruleset, initial}
11  end
```

O módulo é implementado usando um *Agent*, isto é, uma estrutura própria da linguagem que gerencia o estado de uma variável em um processo paralelo. Ele é usado para guardar novos símbolos que foram adicionados ao vocabulário da linguagem, bem como mudanças no símbolo inicial. Dessa forma, cada uma das outras funções apenas retornam as alterações feitas no conjunto de regras.

```
1   def start_grammar(non_terminal, terminal, initial) do
2     # Inicializar com simbolos nao existentes
3     stream = Stream.iterate(1, &(&1 + 1))
4     vocabulary = non_terminal ++ terminal
5
6     nt = Enum.find(stream, &(!(("V" <> to_string(&1)) in vocabulary))
          )
7     t = Enum.find(stream, &(!(("T" <> to_string(&1)) in vocabulary)))
8     init = Enum.find(stream, &(!(("S" <> to_string(&1)) in vocabulary
          )))
9     tmap = %{}
10
11    Agent.start_link(fn -> [{nt, t, init, tmap}, {non_terminal,
          terminal, initial}] end,
12      name: __MODULE__
13    )
14  end
15
16  def get_grammar() do
17    Agent.get(__MODULE__, fn [_ | tail] -> hd(tail) end)
18  end
```

Essas duas funções são responsáveis por inicializar e recuperar o estado da gramática no

agente. O estado é guardado como uma lista de duas partes: a primeira possui variáveis que auxiliam funções específicas e serão explicadas dentro das suas respectivas funções, a segunda parte guarda os elementos da gramática (o vocabulário dividido em símbolos terminais e não-terminais, e o símbolo inicial).

1.1.1 Modificação do símbolo inicial

O método *fix_starter* cria um novo símbolo inicial caso o símbolo atual apareça do lado direito de alguma regra

```
1   def fix_starter(ruleset) do
2     {_, _, initial} = get_grammar()
3     result = Enum.flat_map(ruleset, &fix_starter_map(&1, initial)) |>
          Enum.uniq()
4     #    IO.inspect(result)
5     result
6   end
7
8   defp fix_starter_map({left, right}, initial) do
9     if(Enum.member?(right, initial)) do
10      [{[change_initial()], [initial]}, {left, right}]
11    else
12      [{left, right}]
13    end
14  end
```

Verifica se alguma das regras é inválida e caso seja, substitui o simbolo inicial e cria uma regra para que o novo símbolo derive o antigo símbolo inicial.

É interessante notar que para modificar o símbolo inicial, o método faz uso da função *change_initial/0*:

```
1   def change_initial() do
2     Agent.get_and_update(__MODULE__, fn [
3                                           {nt, t, init, tmap}
4                                           | [{non_terminal, terminal,
                                                initial}]
5                                         ] ->
6       {"S" <> to_string(init),
7        [
8          {nt, t, init, tmap},
9          {["S" <> to_string(init)] ++ non_terminal, terminal, "S" <>
               to_string(init)}
10       ]}
```

```
11      end)
12    end
```

Nela o estado da gramática é atualizado, trocando o símbolo inicial para S_n (onde n é o primeiro valor ainda não usado para criar um novo símbolo nesse formato) e incluindo ele no vocabulário. Além disso, é retornado o valor do novo símbolo para ser usado pelo método.

1.1.2 Remoção de produções nulas

A ideia aqui é simples, mas trabalhosa: Primeiro criaremos uma lista com todos os símbolos que possuem uma produção nula (gera um símbolo ϵ do lado direito). Em seguida, para cada um deles, encontraremos um conjunto de regras no qual eles apareçam ao lado direito. Para cada uma das regras nesse conjunto removeremos o símbolo de produção nula, criando uma nova regra. Caso ele apareça mais de uma vez do lado direito da regra, criaremos uma nova regra para cada possível escolha de símbolo(s) para serem removidos

```
1   # Cria uma lista com todas as variaveis que geram elementos nulos
2   # E depois percorre ela com a  fun  o  fix_nullable/2 para gerar
       uma lista com as novas regras
3   # Retorna o ruleset modificado
4   def fix_nullable(ruleset) do
5     only_nullable = Enum.filter(ruleset, fn {_, right} -> right == :
         epsilon end)
6     filtered = ruleset -- only_nullable
7     #    IO.inspect(filtered)
8     nullable = Enum.map(only_nullable, fn {[symbol], _} -> symbol end
         )
9     result = Enum.reduce(nullable, filtered, &fix_nullable(&1, &2))
10    #    IO.inspect(result)
11    result
12  end
13
14  # Para cada regra do ruleset que possui o elemento anulavel executa
       a funcao replace_nullable/4
15  # Retorna uma lista de novas regras desse elemento anulavel
16  def fix_nullable(nullable, ruleset) do
17    acc =
18      ruleset
19      |> Enum.filter(fn {_, right} -> Enum.member?(right, nullable)
           end)
20      |> Enum.flat_map(&replace_nullable(&1, nullable, [&1], []))
21
22    (acc ++ ruleset) |> Enum.uniq()
```

```
23    end
24
25    # Para cada regra realiza um passo de deriva  o nula (eliminando
         uma ocorrencia do elemento nullable)
26    # Repete a derivacao em cima das regras geradas at  que seja
         gerada uma regra sem nenhum elemento nullable
27    # Retorna a lista com todas as novas regras criadas
28    defp replace_nullable({left, right}, nullable, new_rules, acc) do
29      if(Enum.member?(right, nullable)) do
30        #      IO.inspect(new_rules)
31        new_rules =
32          Enum.reduce(new_rules, [], fn {left, right}, ac ->
33            replace_nullable_step(
34              {left, right},
35              nullable,
36              right,
37              -1,
38              ac
39            )
40          end)
41
42        #      IO.inspect(new_rules)
43        replace_nullable(hd(new_rules), nullable, new_rules, new_rules
             ++ acc)
44      else
45        acc
46      end
47    end
48
49    # Gera as derivacoes com um passo de profundidade
50    defp replace_nullable_step({left, right}, nullable, suffix, index,
         acc) do
51      replace = Enum.find_index(suffix, &(&1 == nullable))
52      #    IO.inspect(replace)
53      if(replace != nil) do
54        {prefix, [_ | suffix]} = Enum.split(right, replace + index + 1)
55        #      IO.inspect(suffix)
56        replace_nullable_step(
57          {left, right},
58          nullable,
59          suffix,
60          replace + index + 1,
```

```
61          [{left, prefix ++ suffix}] ++ acc
62        )
63      else
64        acc
65      end
66    end
```

1.1.3 Remoção de produções unitárias

A remoção de produções unitárias é feita por meio do método abaixo. Ele é auxiliado pelo módulo criado durante o EP1. A ideia aqui é encontrar todos os símbolos terminais e pares de não-terminais que a produção unitária alcança, por isso é muito útil aplicar um fecho de transitividade. Esta propriedade garante que um símbolo X com regra de produção $X \to A$, que alcança o símbolo A, também possa alcançar/derivar todos os símbolos que A alcança (ou seja, para uma regra $A \to BC$ também será criada uma regra $X \to BC$)

```
1   def fix_unit(ruleset) do
2     {nonterminal, terminal, _} = get_grammar()
3     # Filtra apenas as regras de producao unitaria de nao terminais
4     filtered =
5       Enum.filter(ruleset, fn {left, right} ->
6         is_atom(right) or (length(right) < 2 and Enum.member?(
              nonterminal, hd(right)))
7       end)
8
9     # Cria uma lista dos simbolos que realizam producao unitaria
10    unit_producers = filtered |> Enum.map(fn {left, right} -> left
         end) |> Enum.uniq()
11    # Encontra o fecho transitivo do conjunto de regras
12    transitive_closure = SetClosure.make_transitive(ruleset)
13    # Filtra apenas as novas regras, removendo tambem as
         intermediarias
14    new_rules =
15      Enum.filter(transitive_closure -- ruleset, fn {left, right} ->
16        Enum.member?(unit_producers, left) and
17          (length(right) >= 2 or Enum.member?(terminal, hd(right)))
18      end)
19
20    # Remove as producoes unitarias
21    ruleset = ruleset -- filtered
22    # Adiciona as novas regras produzidas
23    result = ruleset ++ new_rules
```

```
24 #    IO.inspect(result)
25     result
26   end
```

1.1.4 Remoção de regras longas

A ideia para consertar regras longas é quebrá-la em novas regras de tamanho dois, adicionando novos símbolos intermediários (observe o exemplo no início da seção)

```
1    def fix_long(ruleset) do
2      long_rules = Enum.filter(ruleset, fn {left, right} -> length(
           right) > 2 end)
3      fixed = ruleset -- long_rules
4      result = fixed ++ Enum.flat_map(long_rules, &break_long(&1))
5      #    IO.inspect(result)
6      result
7    end
8
9    defp break_long({left, right}) do
10     {steps, final} = Enum.split(right, length(right) - 2)
11
12     new_rules =
13       Enum.reduce(
14         tl(steps),
15         [{left, [hd(right), new_symbol(:non_terminal)]}],
16         &break_long(&1, &2)
17       )
18
19     [{_, [_ | last]} | _] = new_rules
20     [{last, final}] ++ new_rules
21   end
22
23   defp break_long(symbol, acc) do
24     [{_, [_ | last]} | _] = acc
25     [{last, [symbol, new_symbol(:non_terminal)]}] ++ acc
26   end
```

Aqui vemos o uso das funções $new_symbol/1, queusaooestadogerenciadopeloAgentparaconstruirnovossmbol losaovocabulriodagramtica.$

`filtered = Enum.filter(ruleset, fn left, right -> Enum.any?(right, Enum.member?(terminal, 1)) end)`

$\texttt{new}_rules = Enum.flat_map(filtered, break_terminal(1, terminal))| > Enum.uniq()ruleset = ruleset - -filteredresult = new_rules + +rulesetIO.inspect(result)resultend$

defp break$_t$erminal$(left, right, terminals)$doconddoEnum.all?$(right, Enum.member?(terminals, 1))->[head|tail] = rightterminal1 = new$_s$ymbol$(:terminal, head)$terminal2 = new$_s$ymbol$(:terminal, hd(tail))[left, [terminal1, terminal2], [terminal1], [head], [terminal2], tail]$

Enum.member?(terminals, hd(right)) -> [head | tail] = right terminal = new$_s$ymbol$(:terminal, head)[left, [terminal] + +tail, [term$

Enum.member?(terminals, hd(tl(right))) -> [head | tail] = right terminal = new$_s$ymbol$(:terminal, hd(tail))[left, [head] + +[termi$

A função *new_ symbol/2* retorna um novo (ou algum que já foi criado para representar o terminal em questão) símbolo não-terminal.

```
1   def new_symbol(:terminal, t) do
2     Agent.get_and_update(__MODULE__, &fetch_terminal(&1, t))
3   end
4
5   defp fetch_terminal([{nt, t, init, tmap} | [{non_terminal, terminal
       , initial}]], key) do
6     if(Map.has_key?(tmap, key)) do
7       {tmap[key], [{nt, t, init, tmap}, {non_terminal, terminal,
          initial}]}
8     else
9       val = "T" <> to_string(t)
10      tmap = Map.put(tmap, key, val)
11      {val, [{nt, t + 1, init, tmap}, {[val] ++ non_terminal,
          terminal, initial}]}
12    end
13  end
```

1.2 Parser

O parser é o módulo principal do EP. Por implementar um algoritmo de programação dinâmica precisamos de uma lista para armazenar os resultados dos subproblemas, ou seja, precisamos "memoizá-los". Para realizar a memoização, o módulo se utiliza de um Agent que guarda um mapeamento da função: para um input (sub-cadeia) w que já tenha sido calculado retorna o seu output (conjunto de símbolos que podem produzir aquela sub-cadeia)

```
1   def start_memo(terminal, ruleset) do
2     memo =
3       ruleset
4       |> Enum.filter(fn {_left, right} -> Enum.member?(terminal, hd(
          right)) end)
5       |> Enum.map(fn {left, right} -> {right, left} end)
6       |> Map.new()
7
8     Agent.start_link(fn -> memo end,
9       name: __MODULE__
```

```
10      )
11    end
12
13    def memoize(input, output) do
14      Agent.update(__MODULE__, &Map.update(&1, input, output, fn val ->
            output ++ val end))
15    end
```

Acima podemos ver como o memo é inicializado com as derivações triviais, isto é, com as regras de derivação de comprimento um que são da forma $X \rightarrow a$, onde X é um símbolo não-terminal e a um terminal. Além disso, a função *memoize/2* adiciona uma nova entrada na memoização.

```
1   use Agent
2   @type symbol :: String.t() | atom
3   @type ruleset :: [{[symbol], [symbol]}]
4   @type grammar :: {[symbol], [symbol], ruleset, symbol}
5   @spec parse([String.t()], grammar) :: boolean
6   def parse(w, {_nonterminal, terminal, ruleset, initial}) do
7     start_memo(terminal, ruleset)
8     ruleset = Enum.filter(ruleset, fn {_left, right} -> length(right)
          > 1 end)
9     cyk(w, ruleset, 2)
10    memo = Agent.get(__MODULE__, & &1)
11    #    IO.inspect(memo)
12    Enum.member?(memo[w], initial)
13  end
```

O método principal, *parse/2*, recebe uma lista de strings (já que um símbolo pode ser composto de mais de um caractere) e a gramática já na forma normal (o módulo Normalizer pode ser usado para normalizar uma gramatica livre de contexto caso necessário).

```
1   use Agent
2   @type symbol :: String.t() | atom
3   @type ruleset :: [{[symbol], [symbol]}]
4   @type grammar :: {[symbol], [symbol], ruleset, symbol}
5   @spec parse([String.t()], grammar) :: boolean
6   def parse(w, {_nonterminal, terminal, ruleset, initial}) do
7     start_memo(terminal, ruleset)
8     ruleset = Enum.filter(ruleset, fn {_left, right} -> length(right)
          > 1 end)
9     cyk(w, ruleset, 2)
10    memo = Agent.get(__MODULE__, & &1)
```

```
11      #    IO.inspect(memo)
12      Enum.member?(memo[w], initial)
13    end
```

Nele é possível observar a chamada da função que implementa o algoritmo CYK (*cyk/3*). Ela recebe como argumentos a string a ser parseada, o conjunto de regras (já filtrado, retirando as que foram usadas para solucionar o conjunto de subproblemas trivial, sub-cadeias de tamanho um) e o tamanho inicial das sub-cadeias(dois).

```
1   def cyk(w, ruleset, size) do
2     chunked = Enum.chunk_every(w, size, 1, :discard)
3     Enum.each(chunked, &cyk(&1, ruleset, 0, []))
4
5     if(length(w) > size) do
6       cyk(w, ruleset, size + 1)
7     end
8   end
```

A primeira parte do algoritmo quebra a cadeia em grupos de tamanho $l = size$, que são colocados na lista *chunked*. Para cada bloco da lista a função *cyk/4* é invocada. Enquanto o tamanho dos blocos não for do mesmo tamanho que a cadeia, essa função é chamada recursivamente, aumentando o tamanho dos blocos em um.

```
1   def cyk(chunk, ruleset, index, acc) do
2     memo = Agent.get(__MODULE__, & &1)
3
4     if(Map.has_key?(memo, chunk)) do
5       #      IO.inspect(chunk, label: "Has memo")
6       memo[chunk]
7     else
8       #      IO.inspect(chunk, label: "Doesn't has memo")
9       cyk_new(chunk, memo, ruleset, index, acc)
10    end
11  end
```

Ao ser chamada, a função tenta primeiro recuperar o resultado que ela deve retornar da lista memoizada. Isso é realizado através da chamada Agent.get/2, que retorna a lista. Caso uma subcadeia idêntica já tenha sido analisada, o algoritmo retorna o resultado armazenado, senão, chama a função *cyk_new/5* para encontrar um novo resultado.

```
1   def cyk_new(chunk, memo, ruleset, index, acc) do
2     if(length(chunk) > index + 1) do
3       {prefix, suffix} = Enum.split(chunk, index + 1)
4
```

```
5         if(
6           memo[prefix] != nil and memo[prefix] != [] and memo[suffix]
               != nil and memo[suffix] != []
7         ) do
8           prefix = memo[prefix]
9           suffix = memo[suffix]
10
11          f =
12            ruleset
13            |> Enum.filter(fn {_left, [head, tail]} ->
14              Enum.member?(prefix, head) and Enum.member?(suffix, tail)
15            end)
16            |> Enum.flat_map(fn {left, _right} -> left end)
17
18          acc = f ++ acc
19          cyk_new(chunk, memo, ruleset, index + 1, acc)
20        else
21          cyk_new(chunk, memo, ruleset, index + 1, acc)
22        end
23      else
24        memoize(chunk, Enum.uniq(acc))
25      end
26    end
```

Esse último método divide cada sub-cadeia em duas partes, prefixo e sufixo. Com sub-cadeias maiores que dois, existe mais de uma possibilidade de escolha para prefixo e sufixo, sendo assim, a função é chamada recursivamente até que todas as possibilidades tenham sido analisadas. Para cada possível escolha, o algoritmo verifica se as sub-sub-cadeias estão presentes no memo: caso não estejam, o algoritmo segue para próxima possibilidade de prefixo/sufixo, caso contrário, o algoritmo obtém as respostas já calculadas e usa elas para encontrar um novo conjunto de símbolos que possa derivar a concatenação dos símbolos que substituíram o prefixo/sufixo. Além disso, após analisar todas as possibilidades a função memoiza o novo resultado.

2 Testes

Os testes são divididos em dois grupos: os testes da unidade de comunicação e os testes do autômato. Os testes da unidade de comunicação se encontram no arquivo */test/parser_test.exs* e os testes do automato se encontram no arquivo */test/normalizer_test.exs*.

2.1 Testes do normalizador

Existe um conjunto de testes para cada método do normalizador. Eles simplesmente verificam se a função "conserta"o conjunto de regras da gramática da forma esperada.

2.2 Testes do parser

Para testar o parser, foi testada diretamente a função parser. Foram analisados dois casos: um presente no material da aula (slides da aula Parser) e um caso vindo do material de referência (uma gramática para reconhecer fechamento de parenteses)

3 Próximos passos

Atualmente, a cadeia para a função parser precisa ser inserida de maneira inconveniente. O formato de uma lista de strings é sujeito a erros de digitação facilmente. Uma ideia seria acrescentar um sistema que a partir do vocabulário de símbolos terminais, fosse capaz de quebrar a string corretamente em uma lista de símbolos terminais. Assim, poderiamos facilmente escrever a entrada como uma string. Outra solução seria limitar os símbolos que podem ser usados para construir uma cadeia, reservando alguns para quebrar a cadeia ou garantir que todos os símbolos tivessem o mesmo tamanho, quebrando a string em pedaços do mesmo tamanho.

Referências

[1] H. Lewis, C. Papadimitriou, *Elements of the Theory of Computation*, (Section 3.6), Prentice-Hall (1998).