Relatório - EP4

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

I. Introdução

Esse relatório acompanha o quarto exercício programa (EP4) da disciplina PCS3556 - Lógica Computacional. Nesse exercício programa, é implementado o algoritmo de conversão de gramática livre de contexto em forma normal de Chomsky. Em seguida, é implementado um algoritmo de reconhecimento de sentenças utilizando programação dinâmica, a partir da forma normal de Chomsky.

21 de Abril de 2018

II. Tarefa

Como estudado na disciplina, algumas gramáticas são livres de contexto. A manipulação delas é mais simples: escrever reconhecedores de dadas gramáticas é muito mais fácil e existem algoritmos rápidos de reconhecimento.

Para facilitar o reconhecimento, pode ser usada uma forma normal, como a forma normal de Chomsky. Uma característica importante da forma normal de Chomsky é que toda gramática em forma normal de Chomsky é livre de contexto, e toda gramática livre de contexto pode ser convertida a uma gramática equivalente em forma normal de Chomsky (no entanto, no processo de conversão regras são adicionadas, e a gramática normalizada pode possuir tamanho até o quadrado do número de regras da gramática original, no pior caso. A facilidade de reconhecimento na forma normal pode implicar em maior dificuldade de leitura e compreensão da gramática por um humano).

Uma gramática livre de contexto G é dita em forma normal de Chomsky se todas as suas regras de produção são da forma:

$A \to BC$, ou
$A \to a$, ou
$S \to \varepsilon$

Onde A, B, e C são símbolos não terminais, a é um símbolo terminal, S é o símbolo não terminal inicial, e ε é a cadeia vazia. B e C também não podem ser o símbolo inicial (representado aqui por S) e a regra de produção $S \to \varepsilon$ está disponível apenas se $\varepsilon \in L(G)$, ou seja, a cadeia vazia pertence à linguagem gerada por G.

No Exercício Programa 2, trabalhamos com um reconhecedor de linguagens sensíveis ao contexto. Nesse Exercício Programa, apenas linguagens livres de contexto serão cobertas; observando a hierarquia de Chomsky, apenas linguagens regulares e livres de contexto são suportadas:

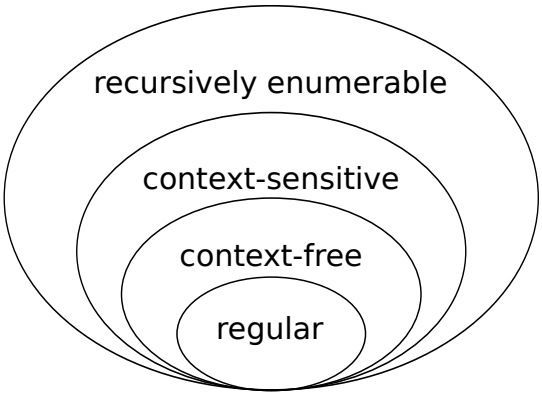

Fig. 1: Hierarquia de Chomsky

Após a conversão para forma normal de Chomsky, implementou-se um algoritmo de reconhecimento com programação dinâmica, muito mais rápido que testar todas as expansões possíveis a partir do não terminal inicial. O algoritmo CYK ou CKY (*Cocke-Kasami-Younger*) [2] foi escolhido.

III. Estruturas de dados

Em alguns locais, estruturas de conjunto fornecidas pelo Elixir (*MapSet*) foram usadas. 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. Funções do módulo *Enum* foram usadas para facilitar a programação funcional.

Nessa implementação, a gramática G é dada como uma tupla $\{$*regras de produção, símbolo inicial*$\}$. As regras de produção são dadas como uma lista de tuplas do tipo $\{$*símbolo inicial, [substituição]*$\}$. Após conversão para forma normal de Chomsky, todas as regras serão da forma $\{$*símbolo inicial, [terminal]*$\}$ ou $\{$*símbolo inicial, [não terminal, não terminal]*$\}$. Além disso, terminais serão sempre representados por nomes maiúsculos (como *A, NP, ADD*) e não terminais por símbolos ou nomes minúsculos (como *a, he,* +).

IV. Algoritmo - Conversão para forma normal de Chomsky

Foram implementadas toda a série de operações para transformação em forma normal de Chomsky, baseado no material da disciplina.

Primeiramente, o símbolo inicial foi removido do lado direito de regras de produção, se presente, com adição de um não terminal novo (*S0*), e o código foi testado:

```
def eliminate_right_start({rules, start}) do
  if Enum.map(rules, &(Enum.member?(elem(&1, 1), start)))
  |> Enum.any? do
    {MapSet.put(rules, {"S0", [start]}), "S0"}
  else
    {rules, start}
  end
end



test "eliminates start symbols at the right side of rules"
     do
  grammar = {[], "S"}
  assert eliminate_right_start(grammar) == grammar
  rules = [{"S", ["A"]}, {"A", ["a"]}, {"A", ["B", "C"]},
       {"B", ["b"]}, {"C", ["c"]}]
  grammar = {rules, "S"}
  # Should be unchanged if no start symbols in the right
       side
  assert eliminate_right_start(grammar) == grammar
  # Should work with MapSet
  rules = MapSet.new [{"S", ["A"]}, {"A", ["S", "A"]}, {"A
       ", ["a"]}]
  assert eliminate_right_start({rules, "S"}) == {MapSet.put
       (rules, {"S0", ["S"]}), "S0"}
end
```

Depois, terminais presentes no lado direito de regras de produção e não solitários foram substituídos. Cada regra do tipo:

$$A \rightarrow X_1...a...X_n$$

Foi trocada por um par de regras equivalentes:

$$N_a \rightarrow a$$
$$A \rightarrow X_1...N_a...X_n$$

Foi implementada a função *eliminate_nonsolitary_terminal*, funções auxiliares e testes:

```
def is_terminal(s) do
  String.downcase(s) == s
end

def terminal_alias_nonterminal(terminal) do
  "N_" <> String.upcase(terminal)
end

def terminal_alias_nonterminal_rule(terminal) do
  {terminal_alias_nonterminal(terminal), [terminal]}
end

def replace_solitary_terminals(rule_right) do
  Enum.map(rule_right, &(
    if is_terminal(&1) do
      terminal_alias_nonterminal(&1)
    else
      &1
    end
  ))
end

def eliminate_nonsolitary_terminal(rules) do
  Enum.reduce(rules, MapSet.new, fn({rule_left, rule_right
       }, acc) ->
    right_terminals = Enum.filter(rule_right, &is_terminal
         (&1))
    if right_terminals == [] or length(rule_right) <= 1 do
      MapSet.put(acc, {rule_left, rule_right})
    else
      Enum.reduce(right_terminals, acc, &MapSet.put(&2,
           terminal_alias_nonterminal_rule &1))
      |> MapSet.put({rule_left, replace_solitary_terminals
           rule_right})
    end
  end)
end



test "checks for terminals" do
  assert is_terminal("+")
  assert is_terminal("a")
  assert is_terminal("0")
  assert not is_terminal("A")
  assert not is_terminal("VB")
  assert not is_terminal("N_A")
  assert not is_terminal("N_+")
end

test "aliases terminals as nonterminals" do
  assert terminal_alias_nonterminal_rule("+") == {"N_+",
       ["+"]}
end

test "replaces solitary terminals" do
  assert replace_solitary_terminals(["X", "+", "a"]) == ["X
       ", "N_+", "N_A"]
end

test "rewrites rules with nonsolitary terminals" do
  assert eliminate_nonsolitary_terminal([
    {"A", ["B"]},
    {"A", ["a"]},
    {"B", ["b"]},
    {"B", ["B", "b"]},
    {"B", ["b", "+", "b"]},
    {"B", ["a", "b"]},
  ]) == MapSet.new [
    {"A", ["B"]},
    {"A", ["a"]},
    {"B", ["b"]},
    {"N_B", ["b"]},
    {"B", ["B", "N_B"]},
    {"N_+", ["+"]},
    {"B", ["N_B", "N_+", "N_B"]},
    {"N_A", ["a"]},
    {"B", ["N_A", "N_B"]},
  ]
end
```

Depois, regras de produção foram modificadas para possuir no máximo dois não terminais do lado direito. Cada regra do tipo

$$A \rightarrow X_1X_2X_3...X_n$$

Foi trocada por regras do tipo:

$$A \rightarrow X_1A_1,$$
$$A_1 \rightarrow X_2A_2,$$
$$...,$$
$$A_{n-2} \rightarrow X_{n-1}X_n,$$

Foi implementada a função *eliminate_right_side_with_multiple_nonterminals*, funções auxiliares e testes:

```
def subsymbol(symbol, depth) do
  if depth == 0 do
    symbol
  else
    "#{symbol}_#{depth}"
  end
end

def eliminate_right_side_with_multiple_nonterminals({
     rule_left, rule_right}, depth) do
  if length(rule_right) > 2 do
    [head | tail] = rule_right
```

```
    new_rule = {subsymbol(rule_left, depth), [head,
        subsymbol(rule_left, depth + 1)]}
    MapSet.put
        eliminate_right_side_with_multiple_nonterminals({
        rule_left, tail}, depth + 1), new_rule
  else
    MapSet.new [{subsymbol(rule_left, depth), rule_right}]
  end
end

def eliminate_right_side_with_multiple_nonterminals(rules)
    do
  Enum.reduce(rules, MapSet.new, fn(rule, acc) ->
    MapSet.union(acc,
        eliminate_right_side_with_multiple_nonterminals(
        rule, 0))
  end)
end
```

```
test "aliases symbols as subsymbols" do
  assert subsymbol("A", 0) == "A"
  assert subsymbol("A", 1) == "A_1"
  assert subsymbol("B", 2) == "B_2"
end

test "eliminates right sides with more than two
    nonterminals" do
  rule = {"A", ["X1", "X2"]}
  assert eliminate_right_side_with_multiple_nonterminals(
      rule, 0) == MapSet.new [rule]
  rule = {"A", ["X1", "X2", "X3", "X4"]}
  new_rules =
      eliminate_right_side_with_multiple_nonterminals(rule,
       0)
  assert Enum.all?(new_rules, fn({_, rule_right}) -> length
      (rule_right) == 2 end)
  assert new_rules == MapSet.new [
    {"A", ["X1", "A_1"]},
    {"A_1", ["X2", "A_2"]},
    {"A_2", ["X3", "X4"]},
  ]
  rules = [{"S", ["A", "B", "C"]}, {"A", ["B", "A", "C"]},
      {"B", ["b"]}, {"C", ["c"]}]
  new_rules =
      eliminate_right_side_with_multiple_nonterminals(rules
      )
  assert Enum.all?(new_rules, fn({_, rule_right}) -> length
      (rule_right) <= 2 end)
  assert new_rules == MapSet.new [
    {"S", ["A", "S_1"]},
    {"S_1", ["B", "C"]},
    {"A", ["B", "A_1"]},
    {"A_1", ["A", "C"]},
    {"B", ["b"]},
    {"C", ["c"]},
  ]
end
```

Regras vazias foram reescritas, propagando o não terminal possivelmente vazio para todas as regras que o possuem no lado direito, e apagando a regra vazia:

Foi implementada a função *eliminate_empty_rules*, funções auxiliares e testes:

```
def split_first(list, divider) do
  {left, right} = Enum.split_while(list, divider)
  if right == [] do
    {list, nil}
  else
    {left, tl(right)}
  end
end

def rewrite_empty_symbol(rule_right, empty_symbol) do
  {left, right} = split_first(rule_right, &(&1 !=
      empty_symbol))
  if right == nil do
    [rule_right]
  else
    Enum.flat_map(rewrite_empty_symbol(right, empty_symbol)
        , &(
      [left ++ &1, left ++ [empty_symbol | &1]]
    ))
  end |> MapSet.new
end

def eliminate_empty_rules(rules, start) do
  eliminate_empty_rules({rules, start})
end

def eliminate_empty_rules({rules, start}) do
  empty_rule = Enum.find(rules, &(elem(&1, 0) != start and
      elem(&1, 1) == []))
  if is_nil(empty_rule) do
    rules
  else
    MapSet.delete(rules, empty_rule)
    |> Enum.flat_map(fn({rule_left, rule_right}) ->
      Enum.map(rewrite_empty_symbol(rule_right, elem(
          empty_rule, 0)), &({rule_left, &1}))
    end)
    |> MapSet.new
    |> eliminate_empty_rules(start)
  end
end
```

```
test "splits first occurence of a divider" do
  assert split_first(["A"], &(&1 != "B")) == {["A"], nil}
  assert split_first(["A", "A"], &(&1 != "A")) == {[], ["A
      "]}
  assert split_first(["A", "B", "C"], &(&1 != "B")) == {["A
      "], ["C"]}
  assert split_first(["A", "B", "C", "B", "B"], &(&1 != "B
      ")) == {["A"], ["C", "B", "B"]}
  assert split_first(["A", "B", "C", "D"], &(&1 != "D")) ==
       {["A", "B", "C"], []}
end

test "rewrites empty symbols" do
  assert rewrite_empty_symbol(["A", "A"], "B") == MapSet.
      new [
    ["A", "A"],
  ]
  assert rewrite_empty_symbol(["A", "A"], "A") == MapSet.
      new [
    ["A", "A"],
    ["A"],
    [],
  ]
  assert rewrite_empty_symbol(["A", "B", "C"], "A") ==
      MapSet.new [
    ["A", "B", "C"],
    ["B", "C"],
  ]
  assert rewrite_empty_symbol(["A", "B", "A", "C", "A", "A
      "], "A") == MapSet.new [
    ["A", "B", "A", "C", "A", "A"],
    ["B", "A", "C", "A", "A"],
    ["A", "B", "C", "A", "A"],
    ["A", "B", "A", "C", "A"],
    ["B", "C", "A", "A"],
    ["B", "A", "C", "A"],
    ["A", "B", "C",  "A"],
    ["A", "B", "A", "C"],
    ["B", "C", "A"],
    ["B", "A", "C"],
    ["A", "B", "C"],
    ["B", "C"],
  ]
end

test "eliminates empty rules" do
  assert eliminate_empty_rules(MapSet.new([
    {"S", ["A", "a"]},
    {"A", ["a"]},
    {"A", []},
  ]), "S") == MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["a"]},
    {"A", ["a"]},
```

```
  ]
  assert eliminate_empty_rules(MapSet.new([
    {"S", ["A", "b", "B"]},
    {"S", ["C"]},
    {"B", ["A", "A"]},
    {"B", ["A", "C"]},
    {"C", ["b"]},
    {"C", ["c"]},
    {"A", ["a"]},
    {"A", []},
  ]), "S") == MapSet.new [
    {"S", ["A", "b", "B"]},
    {"S", ["A", "b"]},
    {"S", ["b", "B"]},
    {"S", ["b"]},
    {"S", ["C"]},
    {"B", ["A", "A"]},
    {"B", ["A"]},
    {"B", ["A", "C"]},
    {"B", ["C"]},
    {"C", ["b"]},
    {"C", ["c"]},
    {"A", ["a"]},
  ]
end
```

Foi implementada a remoção de regras de produção unitárias (do tipo $A \rightarrow B$, onde A e B são não terminais). Regras do tipo:

$$A \rightarrow B$$

Com A e B não terminais, foram removidas, e para todas as regras do tipo

$$B \rightarrow X_1...X_n$$

Foi adicionada uma regra do tipo:

$$A \rightarrow X_1...X_n$$

Representando a regra removida de A para B. Iniciou-se com uma implementação simples, que buscava regras unitárias e realizava as substituições:

```
test "eliminates unit rules" do
  assert eliminate_unit_rules(MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["A"]},
    {"A", ["a"]},
  ]) == MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["a"]},
    {"A", ["a"]},
  ]
  assert eliminate_unit_rules(MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["A"]},
    {"A", ["a"]},
    {"A", ["B"]},
    {"B", ["b"]},
  ]) == MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["a"]},
    {"S", ["b"]},
    {"A", ["a"]},
    {"A", ["b"]},
    {"B", ["b"]},
  ]
end
```

Mas percebeu-se que a implementação falhava em regras unitárias encadeadas (como $A \rightarrow B \rightarrow C$), dependendo da sequência de aplicação. Foi refeita a implementação, dessa vez buscando se a as novas regras adicionadas possuiam ainda novas regras de produção unitárias, e testes foram escritos:

```
def unit_rules_reachable_from(rules, from) do
  unit_rules = Enum.filter(rules, fn({rule_left, rule_right
      }) ->
    rule_left == from and length(rule_right) == 1 and not
        is_terminal(hd(rule_right))
  end)
  expanded_unit_rules = Enum.filter(rules, &Enum.any?(
      unit_rules, fn({_, [unit_right]}) -> elem(&1, 0) ==
      unit_right end))
  |> Enum.map(fn({_, rule_right}) -> {from, rule_right} end
      )
  |> MapSet.new
  |> MapSet.union(rules)
  if expanded_unit_rules == rules do
    rules
  else
    unit_rules_reachable_from(expanded_unit_rules, from)
  end |> MapSet.difference(MapSet.new unit_rules)
end

def eliminate_unit_rules(rules) do
  Enum.reduce(Enum.map(rules, &elem(&1, 0)) |> Enum.uniq,
      rules, fn(rule, acc) ->
    unit_rules_reachable_from(acc, rule)
  end)
end
```

```
test "eliminates unit rules" do
  assert eliminate_unit_rules(MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["A"]},
    {"A", ["a"]},
  ]) == MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["a"]},
    {"A", ["a"]},
  ]
  assert eliminate_unit_rules(MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["A"]},
    {"A", ["a"]},
    {"A", ["B"]},
    {"B", ["b"]},
  ]) == MapSet.new [
    {"S", ["A", "a"]},
    {"S", ["a"]},
    {"S", ["b"]},
    {"A", ["a"]},
    {"A", ["b"]},
    {"B", ["b"]},
  ]
end
```

Por fim, foi implementada a função *to_chomsky_nf*, que chama cada uma das funções anteriores, e mais testes foram escritos:

```
def to_chomsky_nf({rules, start}) do
  {rules, start} = eliminate_right_start({rules, start})
  rules = eliminate_nonsolitary_terminal(rules)
  |> eliminate_right_side_with_multiple_nonterminals
  |> eliminate_empty_rules(start)
  |> eliminate_unit_rules
  {rules, start}
end
```

```
test "generates the Chomsky normal form" do
  math_rules = MapSet.new [
    {"EXPR", ["TERM"]},
    {"EXPR", ["EXPR", "ADD", "TERM"]},
    {"EXPR", ["ADD", "TERM"]},
    {"TERM", ["FACTOR"]},
    {"TERM", ["TERM", "MUL", "FACTOR"]},
    {"FACTOR", ["PRIMARY"]},
    {"FACTOR", ["FACTOR", "^", "PRIMARY"]},
```

```
  {"PRIMARY", ["number"]},
  {"PRIMARY", ["variable"]},
  {"PRIMARY", ["(", "EXPR", ")"]},
  {"ADD", ["+"]},
  {"ADD", ["-"]},
  {"MUL", ["*"]},
  {"MUL", ["/"]},
]
{new_rules, new_start} = eliminate_right_start({
    math_rules, "EXPR"})
assert {new_rules, new_start} == {MapSet.put(math_rules,
    {"S0", ["EXPR"]}), "S0"}

new_rules = eliminate_nonsolitary_terminal(new_rules)
assert new_rules == MapSet.new [
  {"S0", ["EXPR"]},
  {"EXPR", ["TERM"]},
  {"EXPR", ["EXPR", "ADD", "TERM"]},
  {"EXPR", ["ADD", "TERM"]},
  {"TERM", ["FACTOR"]},
  {"TERM", ["TERM", "MUL", "FACTOR"]},
  {"FACTOR", ["FACTOR", "N_^", "PRIMARY"]},
  {"FACTOR", ["PRIMARY"]},
  {"PRIMARY", ["number"]},
  {"PRIMARY", ["variable"]},
  {"PRIMARY", ["N_(", "EXPR", "N_)"]},
  {"N_(", ["("]},
  {"N_)", [")"]},
  {"N_^", ["^"]},
  {"ADD", ["+"]},
  {"ADD", ["-"]},
  {"MUL", ["*"]},
  {"MUL", ["/"]},
]

  new_rules =
      eliminate_right_side_with_multiple_nonterminals(
      new_rules)
assert new_rules == MapSet.new [
  {"S0", ["EXPR"]},
  {"EXPR", ["TERM"]},
  {"EXPR", ["EXPR", "EXPR_1"]},
  {"EXPR_1", ["ADD", "TERM"]},
  {"EXPR", ["ADD", "TERM"]},
  {"TERM", ["FACTOR"]},
  {"TERM", ["TERM", "TERM_1"]},
  {"TERM_1", ["MUL", "FACTOR"]},
  {"FACTOR", ["FACTOR", "FACTOR_1"]},
  {"FACTOR_1", ["N_^", "PRIMARY"]},
  {"FACTOR", ["PRIMARY"]},
  {"PRIMARY", ["number"]},
  {"PRIMARY", ["variable"]},
  {"PRIMARY", ["N_(", "PRIMARY_1"]},
  {"PRIMARY_1", ["EXPR", "N_)"]},
  {"N_(", ["("]},
  {"N_)", [")"]},
  {"N_^", ["^"]},
  {"ADD", ["+"]},
  {"ADD", ["-"]},
  {"MUL", ["*"]},
  {"MUL", ["/"]},
]

assert eliminate_empty_rules({new_rules, new_start}) ==
    new_rules

new_rules = eliminate_unit_rules(new_rules)
expected_rules = MapSet.new [
  {"S0", ["FACTOR", "FACTOR_1"]},
  {"S0", ["number"]},
  {"S0", ["variable"]},
  {"S0", ["N_(", "PRIMARY_1"]},
  {"S0", ["TERM", "TERM_1"]},
  {"S0", ["EXPR", "EXPR_1"]},
  {"S0", ["ADD", "TERM"]},
  {"EXPR", ["FACTOR", "FACTOR_1"]},
  {"EXPR", ["number"]},
  {"EXPR", ["variable"]},
  {"EXPR", ["N_(", "PRIMARY_1"]},
  {"EXPR", ["TERM", "TERM_1"]},
  {"EXPR", ["EXPR", "EXPR_1"]},
  {"EXPR_1", ["ADD", "TERM"]},
  {"EXPR", ["ADD", "TERM"]},
  {"TERM", ["FACTOR", "FACTOR_1"]},
  {"TERM", ["number"]},
  {"TERM", ["variable"]},
  {"TERM", ["N_(", "PRIMARY_1"]},
  {"TERM", ["TERM", "TERM_1"]},
  {"TERM_1", ["MUL", "FACTOR"]},
  {"FACTOR", ["FACTOR", "FACTOR_1"]},
  {"FACTOR_1", ["N_^", "PRIMARY"]},
  {"FACTOR", ["number"]},
  {"FACTOR", ["variable"]},
  {"FACTOR", ["N_(", "PRIMARY_1"]},
  {"PRIMARY", ["number"]},
  {"PRIMARY", ["variable"]},
  {"PRIMARY", ["N_(", "PRIMARY_1"]},
  {"PRIMARY_1", ["EXPR", "N_)"]},
  {"N_(", ["("]},
  {"N_)", [")"]},
  {"N_^", ["^"]},
  {"ADD", ["+"]},
  {"ADD", ["-"]},
  {"MUL", ["/"]},
  {"MUL", ["*"]},
]
assert new_rules == expected_rules

assert to_chomsky_nf({math_rules, "EXPR"}) == {
    expected_rules, "S0"}
end
```

V. ALGORITMO - CKY

Depois, implementou-se CKY. O algoritmo opera de forma *bottom-up*, iniciando por cada caractere da sentença sendo analizada e subindo até gerar a sentença completa. Resumidamente, o algoritmo:

1) Varre a cadeia de entrada, e para cada caractere c busca alguma regra $R \rightarrow c$ correspondente. Todos os não terminais que podem produzir c são salvos em uma lista.
2) Agrupa pares de caracteres da sentença: Para cada par de não terminais adjacentes gerados no passo anterior, busca se alguma regra pode produzir o par, e salva em outra lista.
3) Agrupa trios de caracteres da sequência: Busca regras que gerem um não terminal gerado no passo 1 seguido de um não terminal gerado no passo 2 (ou seja, um não terminal que engloba 3 caracteres da sentença sendo analizada) ou um não terminal gerado no passo 2 seguido de um não terminal gerado no passo 1.
4) Agrupa quartetos de caracteres da sequência: Busca regras que gerem um não terminal gerado no passo 1 seguido de um não terminal gerado no passo 3 ou dois não terminais gerados no passo 2.

Observa-se que para agrupar n caracteres a partir de estados já processados, são feitos $n-1$ testes em janelas da sequência (ou seja, para agrupar 4 caracteres da sequência de entrada, são testados os agrupamentos de 1 e 3 caracteres, 2 e 2, e 3 e 1).

Mais detalhes sobre o funcionamento do algoritmo podem ser vistos na referência indicada [2]. O comportamento *bottom-up* do algoritmo pode ser visto na seguinte imagem, onde o processamento inicia caractere por caractere, agrupando-os até chegar ao símbolo inicial:

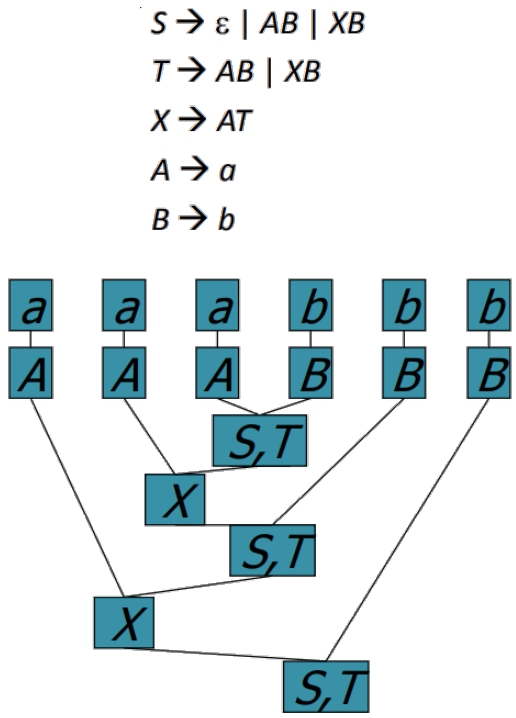

Fig. 2: Comportamento *bottom-up* do CKY

O algoritmo comumente utiliza uma tabela de aplicação de símbolos, subindo na tabela conforme aumenta-se o número de caracteres englobados da cadeia sendo analisada:

j / i	1 a	2 a	3 a	4 b	5 b	6 b
0	A	-	-	-	X	S,T
1		A	-	X	S,T	-
2			A	S,T	-	-
3				B	-	-
4					B	-
5						B

Fig. 3: Tabela do CKY

Na minha implementação, utilizei uma matriz implementada como uma lista de listas. A posição $[a][b]$ da matriz indica a subcadeia de comprimento a iniciando em b. Cada item da matriz é um conjunto (*MapSet*) de não terminais que podem gerar a subcadeia.

Foi criada uma função para inicializar a tabela CKY. Para isso, a função busca regras que gerem cada terminal da cadeia e os coloca no primeiro item da lista, seguido da tabela vazia. Foram feitos testes:

```
def init_cky({rules, _}, sentence) do
  first_expansion = Enum.reverse(sentence)
  |> Enum.reduce([], fn(c, acc) ->
    rules_to_c = Enum.filter(rules, &(elem(&1, 1) == [c]))
    |> Enum.map(&elem(&1, 0))
    |> MapSet.new
    [rules_to_c | acc]
  end)

  n = length(sentence)
  [first_expansion] ++ (
    List.duplicate(MapSet.new, n)
    |> List.duplicate(n - 1)
  )
end
```

```
test "initializes CKY" do
  cky = init_cky({[
    {"S", ["a"]},
    {"A", ["a"]},
    {"B", ["b"]},
    {"C", ["c"]},
  ], nil}, ["a", "b", "a"])
  assert length(cky) == 3
  assert Enum.map(cky, &length/1) == [3, 3, 3]
  assert cky == [[MapSet.new(["S", "A"]), MapSet.new(["B"])
       , MapSet.new(["S", "A"])] | List.duplicate(MapSet.new
       , 3) |> List.duplicate(2)]
end
```

Foi feita a função de atualização da tabela:

```
def cky_table({rules, start}, sentence) do
  m = init_cky({rules, start}, sentence)
  n = length(sentence)
  nt_rules = Enum.filter(rules, &(length(elem(&1, 1)) == 2)
       )
  Enum.reduce(2..n, m, fn(l, acc_l) ->
    Enum.reduce(0..(n - l), acc_l, fn(s, acc_s) ->
      Enum.reduce(0..l - 2, acc_s, fn(p, acc_p) ->
        Enum.reduce(nt_rules, acc_p, fn({a, [b, c]}, acc)
            ->
          if MapSet.member?(Enum.at(acc, p) |> Enum.at(s),
               b) and MapSet.member?(Enum.at(acc, l - p - 2)
                |> Enum.at(s + p + 1), c) do
            List.update_at(acc, l, fn(sublist) -> List.
                 update_at(sublist, s, &MapSet.put(&1, a))
                 end)
          else
            acc
          end
        end)
      end)
    end)
  end)
end
```

E a função *cky* principal, com testes:

```
def cky({rules, start}, sentence) do
  cky_table({rules, start}, sentence)
  |> Enum.at(length(sentence) - 1)
  |> Enum.at(0)
  |> MapSet.member?(start)
end
```

```
test "runs CKY" do
  rules = [
    {"S", ["NP", "VP"]},
    {"VP", ["VP", "PP"]},
    {"VP", ["V", "NP"]},
    {"VP", ["eats"]},
    {"PP", ["P", "NP"]},
    {"NP", ["DET", "N"]},
    {"NP", ["she"]},
    {"V", ["eats"]},
    {"P", ["with"]},
    {"N", ["fish"]},
    {"N", ["fork"]},
    {"DET", ["a"]},
  ]
  assert cky_table({rules, "S"}, ["she", "eats", "a", "fish
       ", "with", "a", "fork"]) == [
    [
      MapSet.new(["NP"]),
      MapSet.new(["V", "VP"]),
      MapSet.new(["DET"]),
      MapSet.new(["N"]),
      MapSet.new(["P"]),
      MapSet.new(["DET"]),
      MapSet.new(["N"]),
    ],
    List.duplicate(MapSet.new, 7),
```

```
    [
      MapSet.new(["S"]),
      MapSet.new([]),
      MapSet.new(["NP"]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new(["NP"]),
      MapSet.new([]),
    ],
    List.duplicate(MapSet.new, 7),
    [
      MapSet.new([]),
      MapSet.new(["VP"]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new([]),
    ],
    List.duplicate(MapSet.new, 7),
    [
      MapSet.new(["S"]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new([]),
      MapSet.new([]),
    ]
  ]
  assert cky({rules, "S"}, ["she", "eats", "a", "fish", "
      with", "a", "fork"])
end
```

VI. Resultados

A implementação foi um sucesso e passou nos testes realizados. Foram realizados testes com gramáticas longas e complexas, como análise da estrutura de frases simples, e o código funcionou como esperado.

VII. Conclusão

Pude aplicar os conhecimentos da disciplina em um problema prático e realizar a transformação de qualquer gramática livre de contexto em uma gramática equivalente, em forma normal de Chomsky. Além disso, implementei um algoritmo rápido de reconhecimento de cadeias usando gramáticas normalizadas em forma normal de Chomsky, utilizando programação dinâmica.

References

[1] Friedel Ziegelmayer. *Elixir ExDoc*. https://hexdocs.pm/elixir/, acessado em 12/04/2018

[2] Scott Farrar. *CKY Algorithm*. CLMA, University of Washington. http://courses.washington.edu/ling571/ling571_fall_2010/slides/cky_cnf.pdf, acessado em 12/04/2018