Gramáticas Regulares: são formalismos axiomáticos ou geradores que permitem definir linguagens, incluindo regulares e não regulares

Gramáticas Lineares: Seja $G=(V,T,P,S)$ uma gramática; $A,B \in V$ e $w \in T^*$
- Gramática linear à direita (GLD): $A \to wB \mid w$
- Gramática linear à esquerda (GLE): $A \to Bw \mid w$
- Gramáticas lineares unitárias: pode ser à direita (GLUD) ou à esquerda (GLUE), adicionam a restrição de que $|w| \leq 1$, sendo assim, uma variável deriva, no máximo uma variável.
- Ex: $a(ba)^*$
  - GLD: $G=(\{S,A\},\{a,b\},P,S\}$ onde P é tal que:
    - $S \to aA$
    - $A \to baA \mid \varepsilon$
  - GLE. $G=(\{S\},\{a,b\},P,S\}$ onde P é tal que:
    - $S \to Sba \mid a$
  - GLUD. $G=(\{S,A,B\},\{a,b\},P,S\}$ onde P é tal que:
    - $S \to aA$
    - $A \to bB \mid \varepsilon$
    - $B \to aA \mid \varepsilon$
  - GLUE. $G=(\{S,A\},\{a,b\},P,S\}$ onde P é tal que:
    - $S \to Aa \mid a$
    - $A \to Sb$

Teorema: Se uma linguagem é gerada por uma gramática regular, então a linguagem é regular.

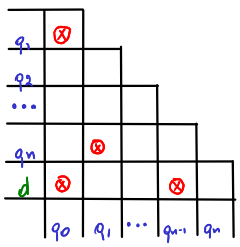
- Passo 4: agrupar estados equivalentes, todos os pares não marcados são estados equivalentes e podem ser unidos em um único estado.
- Passo 5: para cada grupo de estados equivalentes, crie um novo estado único.

Propriedades das Linguagens Regulares
- Lema do Bombeamento: é uma proposição útil para o estudo das propriedades das LR. Afirma que se uma linguagem L é regular, existe um número n (o número de estados do AFD que a aceita) tal que, para qualquer palavra w em L com comprimento $|w| \geq n$, w pode ser dividida em três subpalavras $w=uvz$. As condições são:
  - $|uv| \leq n$
  - $|v| \geq 1$ v não é palavra vazia
  - para todo $i \geq 0$ uv^iz também pertence a L
- Teorema: Se L é LR então: existe n tal que:
  - para qq $w \in L$ onde $|w| \geq n$, w pode ser definida como $w=uvz$, onde $|uv| \leq n$ e $|v| \geq 1$, para todo $i \geq 0$, $uv^iz \in L$.

Linguagem Regular: suficiente representar usando AF, ER ou GR.
Não-Regular: bombeamento, por absurdo

Propriedade do Fechamento das LR
- União: Se L_1 e L_2 são LR, $L_1 \cup L_2$ também é LR
- Concatenação: Se L_1 e L_2 são LR, L_1L_2 também é LR
- Complemento: Se L é uma LR, seu complemento L' (todas as palavras no alfabeto que não está em L) também é uma LR.

Minimização de um Autômato Finito
- objetivo: gerar um AF equivalente com o menor número de estados possível.
- Pré-requisitos:
  - AF deve ser determinístico
  - não pode ter estados inacessíveis (não atingíveis a partir do estado inicial).
  - ter uma função programa total
  - ⚠ caso não satisfaça deve-se gerar um AFD equivalente, eliminar estados inacessíveis e, se necessário, introduzir um estado "morto" para tornar a função programa total.
- Algoritmo de minimização:
  - Passo 1: construir a tabela
  - Passo 2: marcar na tabela os pares obviamente distintos (⊗), $\{q_i,q_j\}$ onde um é final e o outro não é final
  - Passo 3: para cada par $\{q_i,q_j\}$ não marcado e para cada $a \in \Sigma$:
    - calcule $\delta(q_i,a)=p_i$ e $\delta(q_j,a)=p_j$
      - Se $p_i=p_j$ não marcar, q_i é equivalente a q_j para o símbolo a
      - Se $p_i \neq p_j$ e o par $\{p_i,p_j\}$ é marcado, então $\{q_i,q_j\}$ deve ser marcado com X.
      - Se $p_i \neq p_j$ e o par $\{p_i,p_j\}$ não está marcado, adicione $\{q_i,q_j\}$ a uma lista de espera associado a $\{p_i,p_j\}$.



- Interseção: Se L_1 e L_2 são LR, $L_1 \cap L_2$ também é LR

Verificação se uma LR é vazio, finito ou infinito
- Teorema: se L é uma LR aceita por um AF M com n estados, então L é:
  - vazio: sse M não aceita qualquer w tq $|w|<n$
  - finito: sse M não aceita w tq $n \leq |w| < 2n$
  - infinito: sse M aceita w tq $n \leq |w| < 2n$

Igualdade de LRs
- Seja L_1 = Aceito(M_1) e L_2 = Aceito(M_2). As linguagens são iguais sse a linguagem $L_3=(L_1 \cap L_2') \cup (L_1' \cap L_2)$ for vazia.

Autômatos finitos com saída
- Os AFs com saída são uma extensão os AFs, onde o autômato também gera uma palavra de saída ao processar a entrada. A saída gerada não altera o tipo de linguagem que o autômato reconhece e a saída não pode ser usada como memória auxiliar durante o processo.

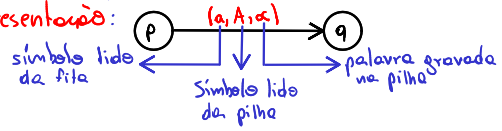
Tipos de AF com saída
- Máquina de Mealy: a saída está associada às transições (movimento entre estados).
  - Para cada transição gera uma palavra de saída, pode ser vazio.
  - Definição
    - 6-upla $M=(\Sigma,Q,\delta,q_0,F,\Delta)$
      - Σ,Q,q_0,F igual ao AF

* $\delta: Q \times \Sigma \to Q \times \Delta^*$, função parcial
  - recebe um par (estado, símbolo)
  - devolve um par (estado, símbolo de saída)
  - Ex: $\delta(q_0, a) \to (q_1, x)$

- **Máquina de Moore**: a saída está associada aos estados, cada estado possui uma saída associada, ao entrar em um estado a máquina gera uma saída.
  - **Definição**:
    - 7-upla $M = (\Sigma, Q, \delta, q_0, F, \Delta, \delta_s)$
      - $\Sigma, Q, \delta, q_0, F$ igual ao AF
      - Δ: alfabeto de saída.
      - $\delta_s: Q \to \Delta^*$, é total, todo estado possui uma saída.
- **Equivalência Moore e Mealy**: são equivalentes porém há uma diferença quando a entrada é palavra vazia:
  - **Moore**: gera a saída do estado inicial, mesmo que a entrada seja vazia.
  - **Mealy**: não gera nada se a entrada estiver vazia, pois não há transição para executar.

Autômato com Pilha

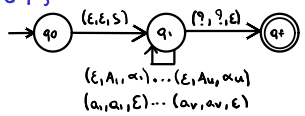
- formalismo tipo autômato associado à classe das linguagens livre de contexto (LLC). Ele é análogo ao AF, mas inclui não-determinismo e uma estrutura de pilha.
- **Estrutura e componentes**: 6-upla $M = (\Sigma, Q, \delta, q_0, F, V)$
  - Σ, Q, q_0, F igual à um AFD.
  - $\delta: Q \times (\Sigma \cup \{\varepsilon, ?\}) \times (V \cup \{\varepsilon, ?\}) \to 2^{Q \times V^*}$, função parcial
    - ?: indica que toda a palavra de entrada foi lida e pilha vazia.
    - ε: movimento vazio e nenhuma gravação é realizada na pilha.
  - V: alfabeto da pilha.
  - **Fita de entrada**: igual ao AF, com uma cabeça que lê símbolos da esquerda para a direita. Pode-se usar um símbolo especial para indicar que a fita foi completamente lida.
  - **Pilha**: memória auxiliar ilimitada, que funciona no modelo LIFO. Só se lê e escreve no topo da pilha, pode-se usar um símbolo especial para indicar que a fita foi completamente lida.
- **Critério de parada**:
  - **Aceita**: se parar em um estado final
  - **Rejeita**: se parar em um estado não-final.
- **Representação**:



Algoritmos de Reconhecimento

- Determinam se uma palavra pertence ou não a uma linguagem, o objetivo é gerar dispositivos de reconhecimento válidos para qualquer linguagem dentro de uma classe, sendo os algoritmos apresentados específicos para LLC. São construídos a partir de uma GLC. Reconhecedores que usam AP são simples, porém ineficientes.
- **Tipos de reconhecedores**:
  - **Top-Down ou preditivo**: constrói uma árvore de decisão para a entrada a partir da raiz (símbolo inicial da gramática), gerando os ramos em direção às folhas (símbolos terminais que compõem a palavra)
  - **Bottom-Up**: é o oposto do Top-Down, parte das folhas e constrói a árvore de derivação em direção a raiz.

- **AP descendente**: é uma forma alternativa de construir um AP a partir de uma GLC, requer que a gramática não tenha recursão à esquerda e simula a derivação mais à esquerda.
  - **Algoritmo**:
    - Inicialmente, empilha o símbolo inicial
    - Sempre que há uma variável no topo da pilha, ela é substituída (de forma não-determinística) por todas as produções desse variável.
    - Se o topo da pilha for um terminal, ele é verificado se é igual ao próximo símbolo da entrada.
  - **Construção de um Autômato com Pilha**
    - Seja $G_1 = (V, T, P, S)$, GLC e sem recursão à esquerda
    - $M = (T, \{q_0, q_1, q_f\}, \delta, q_0, \{q_f\}, V \cup T)$ onde
      - $\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, S)\}$
      - $\delta(q_1, \varepsilon, A) = \{(q_1, \alpha) \mid A \to \alpha \in P\}$
      - $\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$
      - $\delta(q_1, ?, ?) = \{(q_f, \varepsilon)\}$

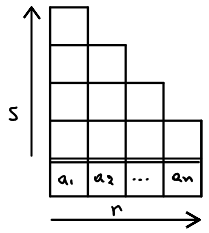


  - **Eliminando recursividade à esquerda**
    - Para: $A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$, onde as partes do tipo $A\alpha_i$ são recursivas à esquerda e β_i são produções não recursivas.
    - Faz-se:

$$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$$

Algoritmo Cocke-Younger-Kasami (CYK)

- é construído sobre uma gramática na FNC é do tipo Bottom-Up, gera todas as árvores de derivação de entrada, com tempo de pré-cessamento proporcional a $|w|^3$.
- **Algoritmo**: baseia-se na construção de uma tabela triangular de derivação, onde cada célula V[r][s] guarda um conjunto de variáveis que podem gerar a subpalavra de comprimento s começando na posição r.



  - **Etapas**:
    1. Inicialização (s=1)
       - Para cada posição r de 1 à n:
         - Coloque em V[r][1] todos os $A \mid A \to a_r \in P$
    2. Construção da tabela de s=2 até n:
       - Para cada tamanho s (2 a n):
         - Para cada início r (1 até n-s+1):
           - Para cada K de 1 até s-1
             - Para cada produção $A \to BC$:
               - Se $B \in V[r][K]$ e $C \in V[r+K][s-K]$, então $A \in V[r][s]$.
    3. Aceitação, w é aceito se o símbolo inicial S estiver em V[1][n].

↳ Ex. Algoritmo de CYK.

$G = (\{S, A\}, \{a, b\}, P, S\}$, onde
$P = \{S \to AA \mid AS \mid b,$
$A \to SA \mid AS \mid a\}$

⚠ pensar em subpalavras

S,A				
S,A	S,A			
S,A	S	S,A		
S,A	A	S	S,A	
A	S	A	A	S
a	b	a	a	b

abaab pertence a linguagem

Forma normal de Chomsky (FNC): Uma gramática $G = (V, T, P, S)$ está na FNC se todas as regras de produções em P forem de um dos dois tipos:

I) $A \to BC$, onde A, B e C são variáveis, e nem B nem C podem ser o símbolo vazio.

II) $A \to a$, onde A é uma variável e $a \in T$

obs: $S \to \varepsilon$, apenas se $\varepsilon \in L(G)$

Simplificação de Gramáticas

↳ Ambiguidade: uma gramática é considerada ambígua se existe uma palavra que pode ter duas ou mais árvores de derivação, ou de forma equivalente, duas ou mais derivações mais à esquerda (ou direita).

↳ As simplificações de GLC são importantes para a construção e otimização de algoritmos.

↳ Tipos de simplificação:

I) Símbolos inúteis: variáveis ou terminais não usados na geração de palavras de terminais.

↳ Algoritmo: GLC $G = (V, T, P, S)$

↳ Etapa 1: Remover variáveis que não geram terminais

- Passo 1: Crie um conjunto $V_1 = \emptyset$
- Passo 2: Repita até V_1 não mudar:
  - Para cada produção $A \to \alpha$, se todos os símbolos de α estão em $T \cup V_1$, adicione A a V_1.
- Passo 3: Elimine de P todas as produções com variáveis fora de V_1.

Resultado: $G_1 = (V_1, T, P_1, S)$

↳ Etapa 2: Remover símbolos inalcançáveis

- Passo 1: Crie o conjuntos $V_2 = \{S\}$ e $T_2 = \emptyset$
- Passo 2: Repita até parar de crescer
  - Para cada produção $A \to \alpha$, com $A \in V_2$
    - Se α contém variável β, adicione β em V_2
    - Se α contém terminal a, adicione a em T_2
- Passo 3: elimine de P_1, todas as produções que usam símbolos fora de $V_2 \cup T_2$
- Resultado: $G_2 = (V_2, T_2, P_2, S)$

↳ Exemplo: $G_1 = \{\{S, A, B, C\}, \{a, b, c\}, P, S\}$ onde
$P = \{S \to aAa \mid bBb,\ A \to a \mid S,\ C \to c\}$

- Etapa 1:
  - $A \to a$, $a \in T \to V_1 = \{A\}$
  - $C \to c$, $c \in T \to V_1 = \{A, C\}$
  - $S \to aAa$, $A \in V_1$, $a \in T \to V_1 = \{A, C, S\}$
  - B foi excluído
- Etapa 2: $G_1 = \{\{S, A, C\}, \{a, b, c\}, P, S\}$ onde
  $P = \{S \to aAa \mid bBb,\ A \to a \mid S,\ C \to c\}$
  - $V_2 = \{S\}$, $T_2 = \emptyset$ → proximos à analisar
  - $S \to aAa$: $V_2 = \{S, A\}$, $T_2 = \{a\}$
  - $A \to a \mid S$: $a \in T_2$ e $S \in V_2$
  - C não é alcançável por S, elimino C.
  - $G_2 = \{\{S, A\}, \{a\}, P_2, S\}$ onde
    $P_2 = \{S \to aAa,\ A \to a \mid S\}$

II) Exclusão de Produções Vazias: objetivo eliminar todas as produções do tipo $A \to \varepsilon$ (exceto $S \to \varepsilon$, se necessário)

↳ Algoritmo: GLC $G = (V, T, P, S)$

↳ Etapa 1: encontrar variáveis que geram ε.

- Passo 1: inicializo $V_\varepsilon = \{A \mid A \to \varepsilon\}$;
- Passo 2: enquanto estiver crescendo:
  - adicione X se existe produção $X \to X_1 X_2 \dots X_n$ tal que $X_i \in V_\varepsilon$ (ou seja, X pode gerar indiretamente ε)

↳ Etapa 2: Criar novas produções sem usar $A \to \varepsilon$

- Passo 1: copiar todas as produções não vazias
  $P_1 = \{A \to \alpha \mid \alpha \neq \varepsilon \text{ e } \alpha \notin V_\varepsilon\}$
- Passo 2: para cada produção que contém variáveis em V_ε crie versões alternativas com essas variáveis omitidas
  Exemplo: se $A \to BCD$ e $C \in V_\varepsilon$, então adiciono $A \to BD$.
  Repito até P_1 não crescer mais.

↳ Etapa 3: Verificar se ε deve ser mantido, se $\varepsilon \in L(G)$ então adicione $S \to \varepsilon$.

↳ Exemplo: $G_1 = \{\{S, X, Y\}, \{a, b\}, P, S\}$, $P = \{S \to aXa \mid bXb \mid \varepsilon,\ X \to a \mid b \mid Y,\ Y \to \varepsilon\}$

Etapa 1

- $V_\varepsilon = \{S, Y\}$, pois $S \to \varepsilon$ e $Y \to \varepsilon$
- $X \to Y$, $V_\varepsilon = \{S, X, Y\}$

Etapa 2: $P_1 = \{S \to aXa \mid bXb,\ X \to a \mid b\}$

- $S \to aXa$, $X \in V_\varepsilon$, logo adiciono $S \to aa$ em P_1
- $S \to bXb$, $X \in V_\varepsilon$, logo adiciono $S \to bb$ em P_1
- $P_1 = \{S \to aXa \mid bXb \mid aa \mid bb \mid \varepsilon,\ X \to a \mid b\}$

Etapa 3: $S \to \varepsilon \in P_1$

III) Eliminar produções $A \to B$: B não acrescenta nada além de redirecionar.

↳ Algoritmo: GLC $G_1 = (V, T, P, S)$

↳ Passo 1: construir fecho de cada variável

- Para cada $A \in V$:
  - Fecho(A) = $\{B \mid A \Rightarrow^* B\}$, usando apenas produções do tipo $X \to Y$.

↳ Passo 2: Construir as novas produções

- copia todas as produções $A \to \alpha$, onde α não é só uma variável (ou seja, $\alpha \in V$)
- Para cada $B \in$ Fecho(A):
  - Se $B \to \alpha \in P$ e $\alpha \notin V$ (α não é produção unitário) adiciono $A \to \alpha$

↳ Exemplo: $G_1 = \{\{S, X\}, \{a, b\}, P, S\}$, $P = \{S \to aXa \mid bXb, X \to a \mid b \mid S \mid \varepsilon\}$

Etapa 1: Fecho(S) = $\emptyset$, Fecho(A) = $\{S\}$

Etapa 2: $P_1 = \{S \to aXa \mid bXb, X \to a \mid b \mid \varepsilon\}$

- Fecho(X) = $\{S\}$ $\begin{cases} S \to aXa \to P_1 \cup \{X \to aXa\} \\ S \to bXb \to P_1 \cup \{X \to bXb\} \end{cases}$
- $P_1 = \{S \to aXa \mid bXb, X \to aXa \mid bXb \mid a \mid b \mid \varepsilon\}$ $X \to S$ foi eliminado

↳ Simplificações Combinadas

↳ Sequência Recomendada:

I) Exclusão produções vazias
II) Exclusão forma $A \to B$
III) Exclusão símbolos inúteis

Forma Normal de Greibach (FNG)

↳ Todas as produções tem a forma: $A \to a\alpha$ onde:

* A = variável (não terminal)
* a = símbolo terminal
* α = (zero ou mais) variáveis $\Rightarrow \alpha \in V^*$

↳ Não permite:

* $A \to \varepsilon$ (vazias)
* $A \to B$ (unitárias)
* $A \to aBb$

Linguagens livres de contexto e AP

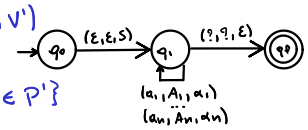
↳ A classe das linguagens reconhecidas pelos APs é igual à classe das LLC. Para qualquer GLC existe um AP que reconhece a linguagem gerada.

↳ GLC → AP

↳ $G_1 = (V, T, P, S)$

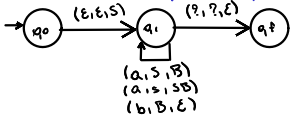
↳ Transformar G_1 para FNG → $G_1' = (V', T', P', S)$

* $M = (T', \{q_0, q_1, q_f\}, \delta, q_0, \{q_f\}, V')$
* $\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, S)\}$
* $\delta(q_1, a, A) = \{(q_1, \alpha) \mid A \to a\alpha \in P'\}$
* $\delta(q_1, ?, ?) = \{(q_f, \varepsilon)\}$



↳ Ex: $G_1 = (\{S, B\}, \{a, b\}, P, S)$, $P = \{S \to aB \mid aSB, B \to b\}$

$M = (\{a, b\}, \{q_0, q_1, q_f\}, \delta, q_0, \{q_f\}, \{S, B\})$



Algoritmo de Early

↳ Características e eficiência:

↳ Possui um tempo de processamento proporcional a $|w|^3$ (onde w é a palavra de entrada), mas pode ser mais eficiente para gramáticas não-ambíguas.

↳ é um algoritmo do tipo top-down (descendente), que executa sempre a derivação mais a esquerda de uma GLC qualquer.

↳ a cada ciclo ele gera um símbolo terminal, que é então comparado com o símbolo correspondente de entrada.

↳ Algoritmo: GLC $G_1 = (V, T, P, S)$, uma GLC qualquer

• $w = a_1 a_2 a_3 \ldots a_n$ palavra a ser reconhecida.

• Marcador "." (ponto): é usado para indicar a posição na produção que será analisada na tentativa de gerar o próximo terminal.

• Sufixo "/u": indica o u-ésimo ciclo em que a produção começou a ser considerada.

↳ Construção do Primeiro Conjunto (D_0):

• D_0 é inicializado com todas as produções que partem do símbolo inicial da gramática ($S \to .\alpha / 0$)

• Em seguida, D_0 é expandido para incluir todas as produções que podem ser aplicadas em derivações mais a esquerda a partir das variáveis que aparecem logo após o ponto em produções já em D_0 ($A \to .B\beta/0$, então $B \to .\phi/0$) este processo se repete até que não haja mais inclusões.

↳ Construção dos conjuntos subsequentes (D_r):

• Para cada ciclo r (de 1 a n), o algoritmo constrói o conjunto D_r, a partir de D_{r-1}

Propriedades da LLC

↳ São mais gerais que as LR, mas ainda são restritas, para provar que uma linguagem não é LLC é necessário usar o lema do bombeamento para LLCs.

↳ Operações sobre LLC:

↳ As LLCs são fechadas para União, concatenação

↳ As LLCs não são fechadas para interseção e complemento.

↳ Verificar se é vazio: seja $G_1 = (V, T, P, S)$, GLC tq $Gera(G_1) = L$, seja $G_1' = (V', T', P', S)$ equivalente a G_1, eliminando os símbolos inúteis se P' for vazio, então L é vazia

↳ Verificar se é finita ou infinita: seja $G = (V, T, P, S)$ uma GLC tq $Gera(G) = L$.

↳ seja $G_1' = (V', T', P', S)$ equivalente a G_1 e G' uma FNC

↳ considere somente as produções da forma $A \to BC$:

* Se existe A tq:
  * $A \to BC$ (A no lado esquerdo)
  * $X \to YA$ ou $X \to AY$ (A no lado direito)

  e se existe um ciclo do tipo $A \Rightarrow^+ \alpha A \beta$ então A é capaz de gerar palavras de qq tamanho, sendo assim a linguagem é infinita.

↳ caso contrário é finita.

• Primeiro, para cada produção em D_{r-1} onde o símbolo da entrada 'a_r' é reconhecido ($A \to \alpha.B\beta/s$), a produção com o ponto avançado é adicionada a D_r ($A \to \alpha a_r.\beta/s$)

• Em seguida, D_r é expandido em duas etapas repetitivas até não ocorrerem mais inclusões:

  • Predição: Para cada produção em D_r que tem uma variável após o ponto ($A \to \alpha.B\beta/s$), todas as produções que iniciam com essa variável ($B \to .\phi/r$) são adicionadas a D_r.

  • Completude: Para cada produção em D_r, que tem o ponto no final ($A \to \alpha./s$), o algoritmo procura produções no conjunto D_s (onde 's' é o ciclo de origem da produção) que referenciam essa variável ($B \to \beta.A\phi/k$) e adiciona a versão com o ponto avançado a D_r ($B \to \beta A.\phi/k$). Esta etapa reduz uma subpalavra à sua variável correspondente.

↳ Condição de aceitação: a palavra de entrada w é aceita se uma produção $S \to \alpha./0$ (indicando que a derivação partiu do símbolo inicial 'S', foi incluído em D_0, e todo o lado direito da produção foi analisado com sucesso) pertencer ao último conjunto de produções, D_n.

↳ Pseudo-Algoritmo: $D_0 = \emptyset$

para toda produção $S \to \alpha \in P$:
faça $D_0 = D_0 \cup \{S \to .\alpha / 0\}$
para toda produção $A \to .B\beta/0 \in D_0$:
faça para toda produção $B \to \phi \in P$:
faça $D_0 = D_0 \cup \{B \to .\phi/0\}$
até não ocorrer mais inclusões

} D_0

n = |w|

para $D_r = \emptyset$:

  para toda $A \to \alpha . a r \beta / s \in D_{r-1}$: // gera símbolo ar

    faça $D_r = D_r \cup \{A \to \alpha a r . \beta / s\}$

  repito

    para toda $A \to \alpha . B \beta / s \in D_r$:

      faça para toda $B \to \phi \in P$:

        faça $D_r = D_r \cup \{B \to . \phi / r\}$

      para toda $A \to \alpha . / s$ de D_r

        faça para toda $B \to \beta . A \phi / K \in D_s$

          faça $D_r = D_r \cup \{B \to \beta A . \phi / K$

  até não ocorrer mais inclusões

D_1 a D_n