Algoritmos de Reconhecimento

- **Uma palavra pertence ou não a uma linguagem?**
  - uma das principais questões relacionadas com o estudo de Ling.Formais
- **“Dispositivo” de reconhecimento**
  - pode ser especificado como um
    - modelo de autômato ou algoritmo implementável em computador
  - em qualquer caso, é importante determinar
    - "quantidade de recursos" necessários
    - ex: tempo e espaço
  - objetivo
    - gerar dispositivos de reconhecimento válidos para qualquer linguagem dentro de uma classe
    - algoritmos apresentados: específicos para LLC

♦ Algoritmos de reconhecimento

- construídos a partir de uma GLC
- reconhecedores que usam Autômato com Pilha
  - * muito simples
  - * em geral, ineficientes
  - * tempo de processamento é proporcional a $k^{|w|}$ (w - entrada; k depende do autômato)
  - * não são recomendáveis para entradas de tamanhos consideráveis
- existe uma série de algoritmos bem mais eficientes
  - * tempo de processamento proporcional a $|w|^3$
  - * ou até um pouco menos
  - * não é provado se o tempo proporcional a $|w|^3$ é efetivamente necessário para que um algoritmo genérico reconheça LLC

♦ Tipos de reconhecedores

- *Top-Down* ou *Preditivo*
- *Bottom-Up*

♦ ***Top-Down*** **ou** ***Preditivo***

- constrói uma árvore de derivação para a entrada
  - a partir da raiz (símbolo inicial da gramática)
  - gera os ramos em direção às folhas (símbolos terminais que compõem a palavra

♦ ***Bottom-Up***

- basicamente, o oposto do Top-Down
- parte das folhas e constrói a árvore de derivação em direção à raiz

AP como Reconhecedor

♦ Construção de reconhecedores usando AP

- relativamente simples e imediata
- Existe uma relação quase direta entre
  - * produções da gramática
  - * transições do AP
- algoritmos
  - * tipo Top-Down
  - * simulam a derivação mais à esquerda da palavra a ser reconhecida
- não-determinismo
  - * testa as diversas produções alternativas da gramática para gerar os símbolos terminais

AP Descendente

♦ Forma alternativa de construir um AP a partir de uma GLC

- algoritmo igualmente simples
- mesmo nível de eficiência
- construção:
  - * gramática sem recursão à esquerda
  - * simula a derivação mais à esquerda

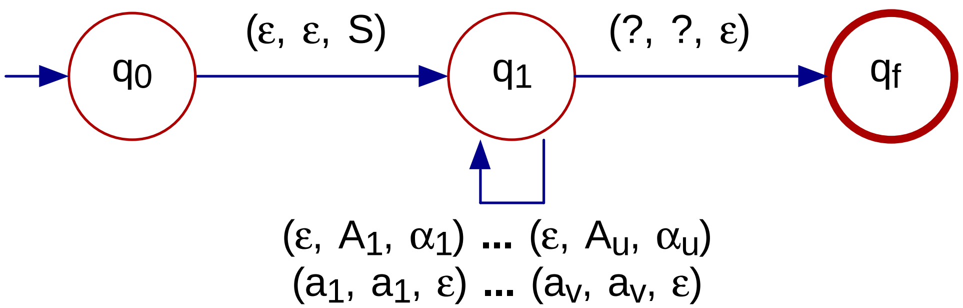
♦ Algoritmo

- inicialmente, empilha o símbolo inicial
- sempre que existir uma variável no topo da pilha, substitui (de forma não-determinística) por todas as produções da variável
- se o topo da pilha for um terminal, verifica se é igual ao próximo símbolo da entrada

♦ Construção de um Autômato com Pilha Descendente

- seja $G = (V, T, P, S)$
  - ∗ GLC
  - ∗ sem recursão à esquerda

- $M = (T, \{q_0, q_1, q_f\}, \delta, q_0, \{q_f\}, V \cup T)$, onde
  - ∗ $\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, S)\}$
  - ∗ $\delta(q_1, \varepsilon, A) = \{(q_1, \alpha) \mid A \to \alpha \in P\}$
  - ∗ $\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$
  - ∗ $\delta(q_1, ?, ?) = \{(q_f, \varepsilon)\}$



Eliminação da Recursividade à Esquerda

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

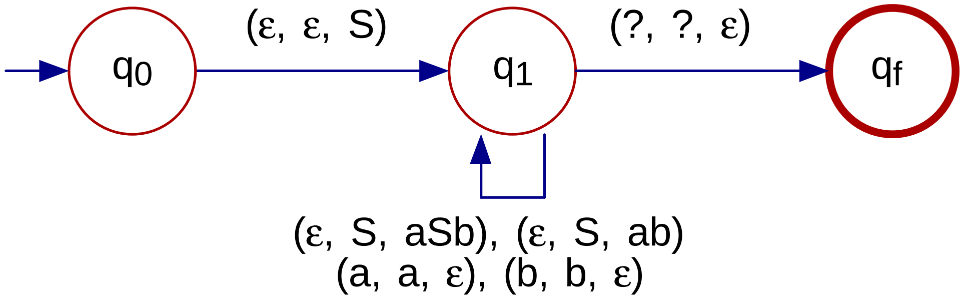
$$\begin{aligned} E &\rightarrow E + T \\ &\mid E - T \\ &\mid T \\ T &\rightarrow c \\ &\mid (E) \end{aligned}$$

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \\ &\mid -TE' \\ &\mid \varepsilon \\ T &\rightarrow c \\ &\mid (E) \end{aligned}$$

♦ *Exemplo.* $L = \{a^n b^n \mid n \geq 1\}$

- $G = (\{S\}, \{a, b\}, P, S)$, onde
  $P = \{S \rightarrow aSb \mid ab\}$ (sem recursão à esquerda)

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



Algoritmo de Cocke-Younger-Kasami

♦ Cocke-Younger-Kasami (CYK)

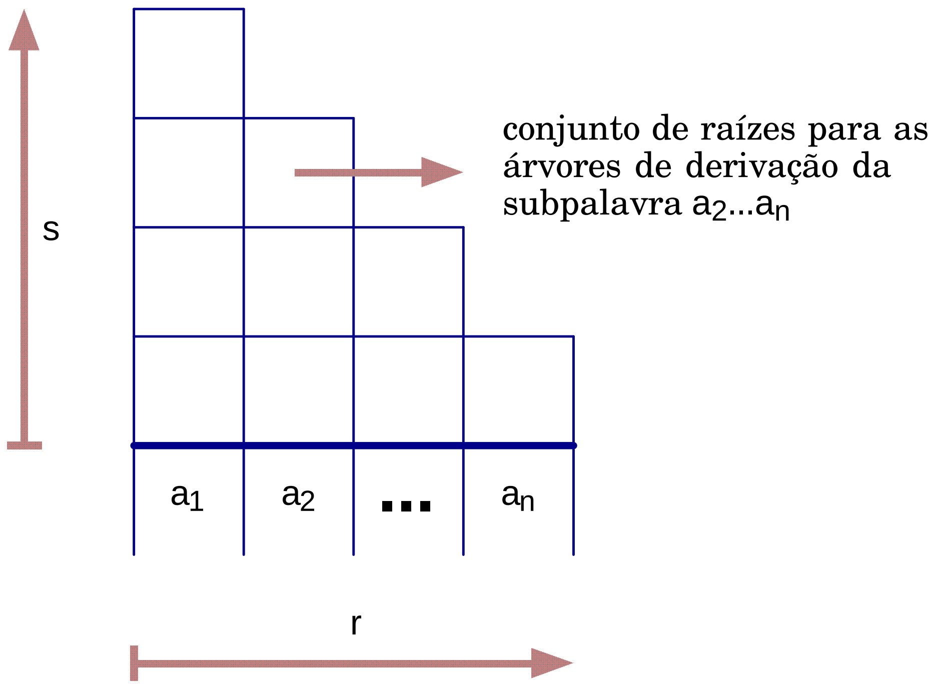
- desenvolvido independentemente por
  - Cocke, Younger e Kasami
  - em 1965
- construído sobre uma gramática na FNC
- tipo *bottom-up*
- gera todas as árvores de derivação da entrada
- tempo de processamento proporcional a $|w|^3$

♦ Algoritmo

- construção de uma tabela triangular de derivação
- cada célula representa o conjunto de raízes que pode gerar a correspondente sub-árvore

♦ Algoritmo de CYK

- $G = (V, T, P, S)$ na FNC onde $T = \{a_1, a_2, ..., a_t\}$
- V_{r_s} representa as células da tabela triangular de derivação (suponha $w = a_1a_2...a_n$)



a) *Variáveis q. geram terminais diretamente* $A \rightarrow a$

```
para r variando de 1 até n
faça V_r1 = {A | A → a_r ∈ P}
```

$$\text{para } r \text{ variando de } 1 \text{ até } n \quad \text{faça } V_{r_1} = \{A \mid A \rightarrow a_r \in P\}$$

b) *Produção que gera duas variáveis* $A \rightarrow BC$

```
para s variando de 2 até n
faça para r variando de 1 até n-s+1
     faça V_rs = ∅
          para k variando de 1 até s-1
          faça V_rs = V_rs ∪
               {A | A → BC ∈ P, B ∈ V_rk e C ∈ V_(r+k)(s-k)}
```

- Note-se que:
  - ∗ limite de iteração para r é $n - s + 1$, pois a tabela de derivação é triangular
  - ∗ os vértices V_{r_k} e $V_{(r+k)_{(r-k)}}$ são as raízes das sub-árvores de V_{r_s}
  - ∗ se uma célula for vazia, significa que esta célula não gera qualquer sub-árvore

c) *Condição de aceitação da entrada*

- se o símbolo inicial pertence ao vértice V_{1n} (raiz da árvore de derivação de toda palavra), então a entrada é aceita

♦ *Exemplo*

- G = ({S, A}, {a, b}, P, S), onde
  P = {S → AA | AS | b, A → SA | AS | a}

- Tabela triangular de derivação para abaab

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

Como S é raiz da árvore de derivação,
a entrada é aceita