

1.3 - Linguagens Livres de Contexto

- ♦ Recordação: Uma gramática livre de contexto (GLC) é uma quádrupla $G = (N, \Sigma, P, S)$ onde:
 - N - conjunto de símbolos não terminais
 - Σ - alfabeto terminal ($N \cap \Sigma = \emptyset$; $V = N \cup \Sigma$)
 - S - símbolo inicial ($S \in N$)
 - P - conjunto de produções da forma: $A \rightarrow \alpha$ ($A \in N$; $\alpha \in V^*$)

Exemplo: $G = (N, \Sigma, P, S)$

$N = \{ S, <op>, <var>, <cte> \}$

$\Sigma = \{ 1, 0, x, y, z, (,), +, * \}$

$P : S \rightarrow <var> \mid <cte> \mid S <op> S \mid (S)$

$<op> \rightarrow + \mid *$

$<var> \rightarrow x \mid y \mid z$

$<cte> \rightarrow 0 \mid 1 \mid 0<cte> \mid 1<cte>$

$(x + 101) * y \in L(G)$ pois:

$S \Rightarrow S <op> S \Rightarrow (S) <op> S \Rightarrow (S <op> S) <op> S \Rightarrow (<var> <op> S)$
 $<op> S \Rightarrow (x <op> S) <op> S \Rightarrow (x + S) <op> S \Rightarrow (x + <cte>) <op> S \Rightarrow$
 $(x + 1<cte>) <op> S \Rightarrow (x + 10<cte>) <op> S \Rightarrow (x + 101) <op> S \Rightarrow (x +$
 $101) * S \Rightarrow (x + 101) * y.$

Árvores Sintáticas e Ambigüidade

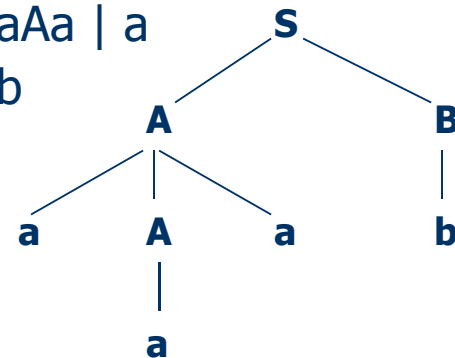
- Seja GLC com produções:

$S \rightarrow AB$

$A \rightarrow aAa \mid a$

$B \rightarrow b$

1. $S \Rightarrow AB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaab$

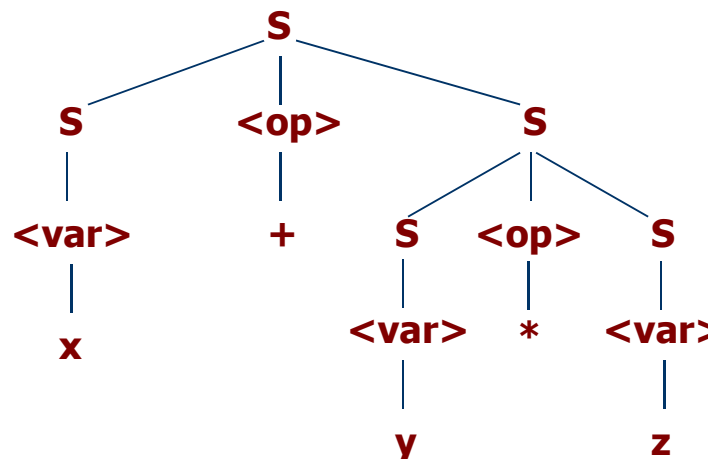
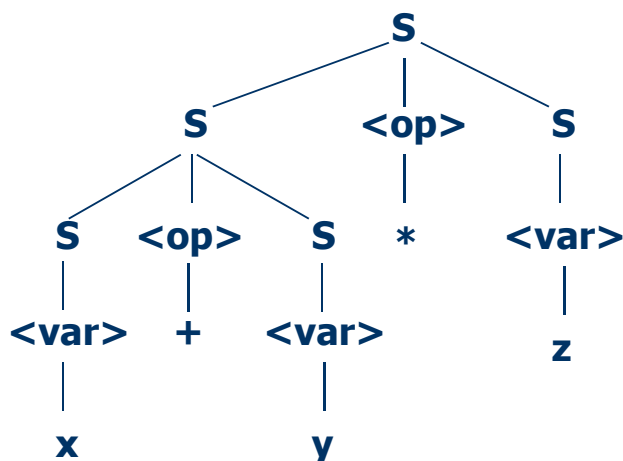


árvore sintática
correspondente
à derivação (1)

2. $S \Rightarrow AB \Rightarrow Ab \Rightarrow aAab \Rightarrow aaab$

Essa derivação é diferente da derivação (1). No entanto, as árvores sintáticas associadas às duas derivações é a mesma.

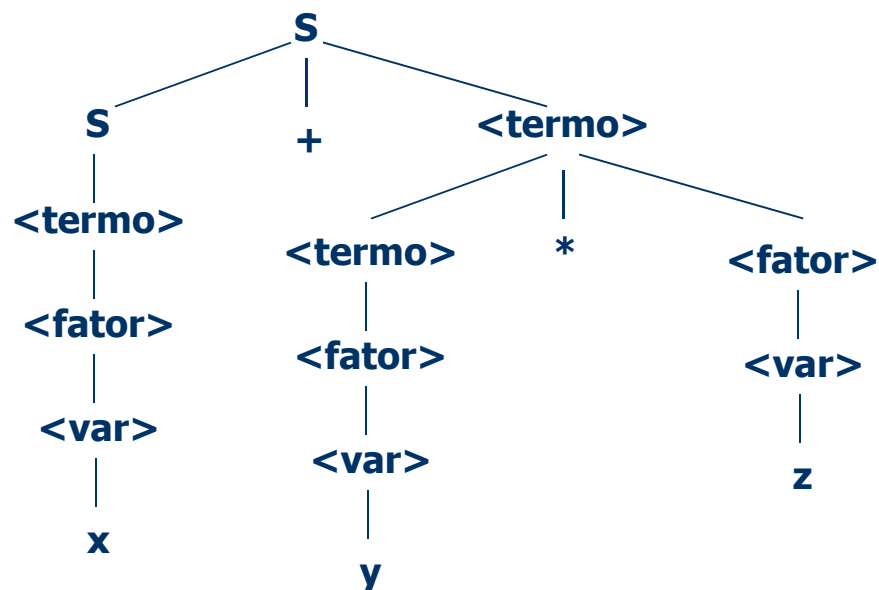
- Considerando a gramática do exemplo anterior, para a sentença $x + y * z$ podemos determinar as seguintes árvores sintáticas:



neste caso,
como existem
duas árvores
sintáticas
distintas para a
mesma sentença
há ambigüidade.

Árvores Sintáticas e Ambigüidade

- Seja agora a gramática (equivalente à anterior):
 $S \rightarrow S + \langle \text{termo} \rangle \mid \langle \text{termo} \rangle$
 $\langle \text{termo} \rangle \rightarrow \langle \text{termo} \rangle * \langle \text{fator} \rangle \mid \langle \text{fator} \rangle$
 $\langle \text{fator} \rangle \rightarrow \langle \text{var} \rangle \mid \langle \text{cte} \rangle \mid (S)$
 $\langle \text{var} \rangle \rightarrow x \mid y \mid z$
 $\langle \text{cte} \rangle \rightarrow 0 \mid 1 \mid 0\langle \text{cte} \rangle \mid 1\langle \text{cte} \rangle$
- Para a sentença $x + y * z$ temos:



- Neste caso a árvore de derivação é única.

Árvores Sintáticas e Ambigüidade

- ♦ Derivação mais à esquerda: substituir sempre o não terminal mais à esquerda da forma sentencial.
- ♦ Derivação mais à direita: substituir sempre o não terminal mais à direita da forma sentencial.
- ♦ **Exemplo:**
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid a$$
- ♦ **derivação mais à esquerda de $a + a$:**
$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + F \Rightarrow a + a$$
- ♦ **derivação mais à direita de $a + a$:**
$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + a \Rightarrow T + a \Rightarrow F + a \Rightarrow a + a$$
- ♦ Definição: Uma gramática G é ambigüa se existir uma sentença que possua duas árvores de derivação diferentes (ou duas derivações mais à esquerda, ou duas derivações mais à direita).
- ♦ Definição: Uma linguagem livre de contexto (LLC), L , é inerentemente ambigüa se não existir uma gramática livre de contexto G , não ambigüa, tal que $L = L(G)$.
- ♦ Exemplo: $L = \{ a^i b^j c^k \mid i = j \text{ ou } j = k; i, j, k \geq 1 \}$ é inerentemente ambigüa.

Árvores Sintáticas e Ambigüidade

- ♦ Intuitivamente: $L = \{ a^i b^j c^k \mid i = j \text{ ou } j = k; i, j, k \geq 1 \}$ é inerentemente ambígua porque o processo de gerar sentenças com mesmo número de a's e b's é independente (e, portanto, diferente) do processo de gerar sentenças com mesmo número de b's e c's. Logo, para cadeias com mesmo número de a's, b's e c's é inevitável que existam duas árvores sintáticas.

(para uma prova formal, ver: HARRISON, M.A. "Introduction do formal language theory", p. 233-239)

- ♦ Exercício: Construir uma GLC que gere essa linguagem!

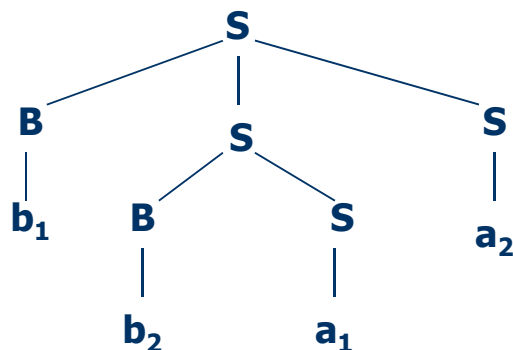
- ♦ Outro exemplo: "dangling else"

$S \rightarrow \text{if } B \text{ then } S \text{ else } S \mid \text{if } B \text{ then } S \mid a_1 \mid a_2$

$B \rightarrow b_1 \mid b_2$

se esse S for substituído por "if B then S" dará ambigüidade.

- ♦ Seja: if b_1 then if b_2 then a_1 else a_2

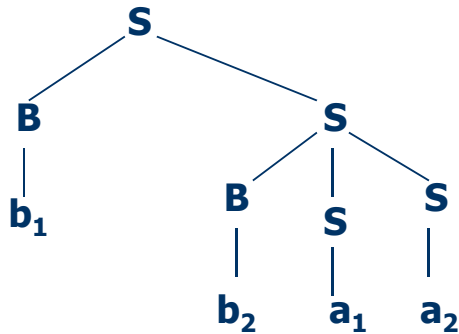


corresponde a:

if b_1 then
if b_2 then a_1
else a_2

Árvores Sintáticas e Ambigüidade

- ♦ if b_1 then if b_2 then a_1 else a_2



corresponde a:
if b_1 then
if b_2 then a_1 else a_2

- ♦ Neste caso, o programador e o compilador podem interpretar a sentença de formas diferentes, o que não é desejável. Em linguagens de programação a ambigüidade é resolvida por uma convenção semântica: “o else sempre corresponde ao if mais próximo”. Com essa convenção, a sentença:
if b_1 then if b_2 then a_1 else a_2
deve ser interpretada como if b_1 then (if b_2 then a_1 else a_2), ou seja, com árvore sintática como a mostrada acima.
- ♦ Definição: Uma GLC $G = (N, \Sigma, P, S)$ é Λ -livre se:
 - a) P não contém produções da forma $A \rightarrow \Lambda$ ($A \in N$), ou
 - b) a única produção desse tipo é $S \rightarrow \Lambda$ e S não aparece no lado direito de nenhuma produção.

Forma Normal de Chomsky

- ♦ Definição: Uma GLC $G = (N, \Sigma, P, S)$ está na forma normal de Chomsky (FNC) se G for Λ -livre e suas produções forem da forma:

$A \rightarrow BC$ ou $A \rightarrow a$ ($A, B, C \in N$; $a \in \Sigma$).

- ♦ Exemplo: Seja G_1 com produções:

$S \rightarrow \Lambda \mid ab \mid aAb$

$A \rightarrow aAb \mid ab$

Note que $L(G_1) = \{ a^n b^n \mid n \geq 0 \}$. G_1 é Λ -livre mas não está na FNC.

Seja G_2 com produções:

$S \rightarrow \Lambda \mid AB \mid AX$

$X \rightarrow YB$

$Y \rightarrow AX \mid AB$

$A \rightarrow a$

$B \rightarrow b$

$S \Rightarrow AX \Rightarrow AYB \Rightarrow AAXB \Rightarrow AAYBB \Rightarrow AAABBB \Rightarrow^6 aaabbb$

G_2 está na FNC e $L(G_2) = L(G_1)$.

- ♦ **Exercício: Mostrar que $L(G_2) = L(G_1)$.**
- ♦ Por que o interesse pela FNC?
Por uma razão teórica: para uma gramática na FNC as árvores sintáticas são sempre árvores binárias (incompletas) e essa limitação na ramificação das árvores sintáticas facilita algumas provas de teoremas.

Gramáticas Λ -livres

- ♦ Lema: Existe um algoritmo que transforma uma GLC $G = (N, \Sigma, P, S)$ qualquer em uma outra GLC $G' = (N', \Sigma, P', S')$ equivalente e Λ -livre.
- ♦ Prova: Seja $\#N = n$ e $w_1 = \{ A \mid A \rightarrow \Lambda \in P \}$. Para todo $k \geq 1$, seja:
$$w_{k+1} = w_k \cup \{ A \mid A \rightarrow \alpha_1 \dots \alpha_m; \alpha_i \in w_k, 1 \leq i \leq m \}$$

Temos então:

- $w_i \subseteq w_{i+1}$ para todo $i \geq 1$
- se $w_i = w_{i+1}$ então $w_i = w_{i+m}$ para $m \geq 1$
- $w_{n+1} = w_n$, ou seja, a construção dos conjuntos w_i pára, na pior das hipóteses, quando $i = n$.
- $w_n = \{ A \in N \mid A \Rightarrow^+ \Lambda \}$
- $\Lambda \in L(G) \Leftrightarrow S \in w_n$

Então, para construir a gramática $G' = (N \cup \{ S' \}, \Sigma, P', S')$ seja P' com as produções:

- $S' \rightarrow S$
- $S' \rightarrow \Lambda \in P' \Leftrightarrow S \in w_n$
- $A \rightarrow A_1 \dots A_k, k \geq 1, A_i \in N \cup \Sigma$ tal que:
(existem $\alpha_1, \dots, \alpha_{k+1} \in w_n^*$ tal que $A \rightarrow \alpha_1 A_1 \dots \alpha_k A_k \alpha_{k+1} \in P$)

Claramente, S' não aparece no lado direito de qualquer produção de P' . Além disso,

$$\Lambda \in L(G') \Leftrightarrow S' \rightarrow \Lambda \in P' \Leftrightarrow S \in w_n \Leftrightarrow S \Rightarrow^* \Lambda \Leftrightarrow \Lambda \in L(G)$$

Gramáticas Λ -livres

Portanto, $\Lambda \in L(G') \Leftrightarrow \Lambda \in L(G)$. Como em G' não existem produções levando a Λ (exceto, possivelmente, $S' \rightarrow \Lambda$), então G' é Λ -livre.

Devemos mostrar que $L(G') = L(G)$.

- 1) se $A \rightarrow A_1 \dots A_k \in P'$ então $A \rightarrow \alpha_1 A_1 \dots \alpha_k A_k \alpha_{k+1} \in P$ e $\alpha_i \xRightarrow{G}^* \Lambda$, $i = 1, \dots, k+1$. Logo:

$$A \xRightarrow{G} \alpha_1 A_1 \dots \alpha_k A_k \alpha_{k+1} \xRightarrow{G}^* A_1 \dots A_k$$

Portanto, toda produção $A \rightarrow A_1 \dots A_k \in P'$ pode ser simulada em G por uma derivação. Logo: $L(G') \subseteq L(G)$.

- 2) Vamos mostrar, por indução no comprimento da derivação, que se:

$$A \xRightarrow{G}^h \alpha, \alpha \in \Sigma^+ \text{ então } A \xRightarrow{G'}^* \alpha$$

Base: $h = 1$. Neste caso, $A \rightarrow \alpha \in P$. Logo, como $\alpha \neq \Lambda$, $A \rightarrow \alpha \in P'$ (porque é sempre possível escolher $\alpha_1 = \dots = \alpha_{k+1} = \Lambda \in w_n^*$)

Hipótese indutiva: $A \xRightarrow{G}^h \alpha, \alpha \in \Sigma^+ \text{ então } A \xRightarrow{G'}^* \alpha$, $h \geq 1$

Passo da indução: Seja $A \xRightarrow{G} A_1 \dots A_k \xRightarrow{G}^h \alpha$, $\alpha \in \Sigma^+$

Então existem $\alpha_i \in \Sigma^*$ tais que $\alpha = \alpha_1 \dots \alpha_k$ e $A_i \xRightarrow{G}^{h'} \alpha_i$ com $h' \leq h$. Pela hipótese indutiva, $A_i \xRightarrow{G'}^* \alpha_i$, $i = 1, \dots, k$. Se $\alpha_i = \Lambda$ então $A_i \in w_n$ e $A \rightarrow A_{j1} \dots A_{jp} \in P'$, onde $A_{j1} \dots A_{jp}$ é uma subsequência de $A_1 \dots A_k$ obtida retirando-se A_i . Logo, em qualquer caso:

$$A \xRightarrow{G'} A_{j1} \dots A_{jp} \xRightarrow{G'}^* \alpha_{j1} \dots \alpha_{jp} = \alpha. \text{ Logo: } L(G) \subseteq L(G').$$

Portanto, de (1) e (2) temos que: $L(G') = L(G)$.

Gramáticas Λ -livres

- Exemplo: Seja G , GLC com produções:
 $S \rightarrow aAb$
 $A \rightarrow P \mid aAb \mid AA$
 $P \rightarrow x \mid E$
 $E \rightarrow \Lambda$

Então:

$$w_1 = \{ E \}$$
$$w_2 = \{ E, P \}$$
$$w_3 = \{ E, P, A \} = w_4$$

Para construir G' (Λ -livre e equivalente a G) devemos lembrar que:

- toda produção de G deve estar em G' , exceto produções do tipo $A \rightarrow \Lambda$, pois sempre é possível escolher $\alpha_1, \dots, \alpha_{k+1}$ todos iguais a Λ .
- uma produção de P' pode ser obtida de uma produção de P retirando do lado direito um não-terminal que pertença a w_n , com o cuidado de não tornar o lado direito vazio.

Logo: $G' = (\{ S, A, P, E, S' \}, \{a, b, x\}, P', S')$ com P' dado por:

$$S' \rightarrow S$$
$$S \rightarrow aAb \mid ab$$
$$A \rightarrow P \mid aAb \mid AA \mid ab$$
$$P \rightarrow x \mid E$$

Gramáticas Λ -livres

- ♦ Lema: Existe um algoritmo que transforma qualquer gramática livre de contexto $G = (N, \Sigma, P, S)$ em outra $G' = (N, \Sigma, P', S)$ equivalente, Λ -livre e sem produções do tipo $A \rightarrow B$, $A, B \in N$ (produção inútil).
- ♦ Prova (informal). Pelo lema anterior, pode-se admitir que G é Λ -livre.

Algoritmo:

1. Construir para cada $A \in N$, o conjunto $N_A = \{ B \in N \mid A \Rightarrow^* B \}$ da seguinte maneira:
 - a) $N_0 = \{ A \}$; $i = 1$.
 - b) $N_i = \{ C \mid B \rightarrow C \in P, B \in N_{i-1} \} \cup N_{i-1}$
 - c) se $N_i \neq N_{i-1}$ então fazer $i = i + 1$ e voltar ao passo (b). Caso contrário, fazer $N_A = N_i$.
2. Construir P' da seguinte forma:
se $B \rightarrow \alpha \in P$ e não é da forma $B \rightarrow A$, colocar $A \rightarrow \alpha$ em P' para todo A tal que $B \in N_A$

- ♦ **Exemplo:**

$E \rightarrow E + T \mid T$	$N_E = \{ E, T, F \}$
$T \rightarrow T * F \mid F$	$N_T = \{ T, F \}$
$F \rightarrow (E) \mid a$	$N_F = \{ F \}$

A nova gramática será:

$E \rightarrow E + T \mid T * F \mid (E) \mid a$
$T \rightarrow T * F \mid (E) \mid a$
$F \rightarrow (E) \mid a$

Forma Normal de Chomsky

- ♦ Teorema: Seja L uma LLC. Então existe G na FNC tal que $L = L(G)$.
- ♦ Prova: Pelos dois lemas anteriores, $L = L(G)$ para alguma GLC $G = (N, \Sigma, P, S)$ Λ -livre e sem produções do tipo $A \rightarrow B$. Construir G' na FNC da seguinte maneira: $G' = (N', \Sigma, P', S)$ tal que N' e P' são dados por:
 - 1) se $A \rightarrow a \in P$ então $A \rightarrow a \in P'$ ($a \in \Sigma$)
 - 2) se $A \rightarrow BC \in P$ então $A \rightarrow BC \in P'$ ($B, C \in N$)
 - 3) se $S \rightarrow \Lambda \in P$ então $S \rightarrow \Lambda \in P'$
 - 4) se $A \rightarrow X_1 \dots X_k \in P$ ($k > 2$), adicionar a N' e a P' os seguintes não-terminais e produções, respectivamente:
$$A \rightarrow X_1' \langle X_2 \dots X_k \rangle$$
$$\langle X_2 \dots X_k \rangle \rightarrow X_2' \langle X_3 \dots X_k \rangle$$
$$\dots$$
$$\langle X_{k-1} X_k \rangle \rightarrow X_{k-1}' X_k'$$
onde: $X_i' = X_i$, se $X_i \in N$ e $X_i' =$ um novo não-terminal se $X_i \in \Sigma$.
 - 5) para todo não-terminal X_i' criado no passo (4), acrescentar a P' a produção $X_i' \rightarrow X_i$
- ♦ Exercício. Mostrar que $L(G') = L(G)$.

Forma Normal de Chomsky

- ♦ Exemplo: Passar para a FNC:
 $S \rightarrow aAB \mid BA$
 $A \rightarrow BBB \mid a$
 $B \rightarrow AS \mid b$

Neste caso, temos:

$S \rightarrow \langle a \rangle \langle AB \rangle \mid BA$

$A \rightarrow B \langle BB \rangle \mid a$

$B \rightarrow AS \mid b$

$\langle a \rangle \rightarrow a$

$\langle AB \rangle \rightarrow AB$

$\langle BB \rangle \rightarrow BB$

- ♦ O teorema a seguir (“pumping lemma”) é um resultado importante para provar que certas linguagens não são livres de contexto.
- ♦ Teorema: Para toda linguagem livre de contexto L , podem ser determinados inteiros k e m tais que, se $z \in L$, $|z| > k$ então:
 1. $z = uvwxy$
 2. $|vwx| \leq m$
 3. $vx \neq \Lambda$
 4. para todo $i \geq 0$, $uv^iwx^iy \in L$

“Pumping lemma” para LLC

- ♦ Prova: Seja $L = L(G)$ para alguma gramática G na FNC. Logo, as árvores de derivação em G serão árvores binárias.

Fato 1: Se o caminho mais longo na árvore de derivação de z possuir ordem n então $|z| \leq 2^{n-1}$.

Porque: $|z|$ = número de folhas da árvore de derivação de z . Numa árvore binária de ordem n existem, no máximo, 2^n folhas (o número de folhas será igual a 2^n somente se a árvore binária for completa). Entretanto, as árvores de derivação de ordem n de uma gramática na FNC possuem, no máximo, 2^{n-1} folhas, porque a última derivação, em qualquer caminho, não irá produzir ramificação pois irá usar produções da forma $A \rightarrow a$.

Fato 2: Se $|z| > 2^n$ então algum caminho na árvore de derivação de z possui, no mínimo, ordem igual a $n+2$.

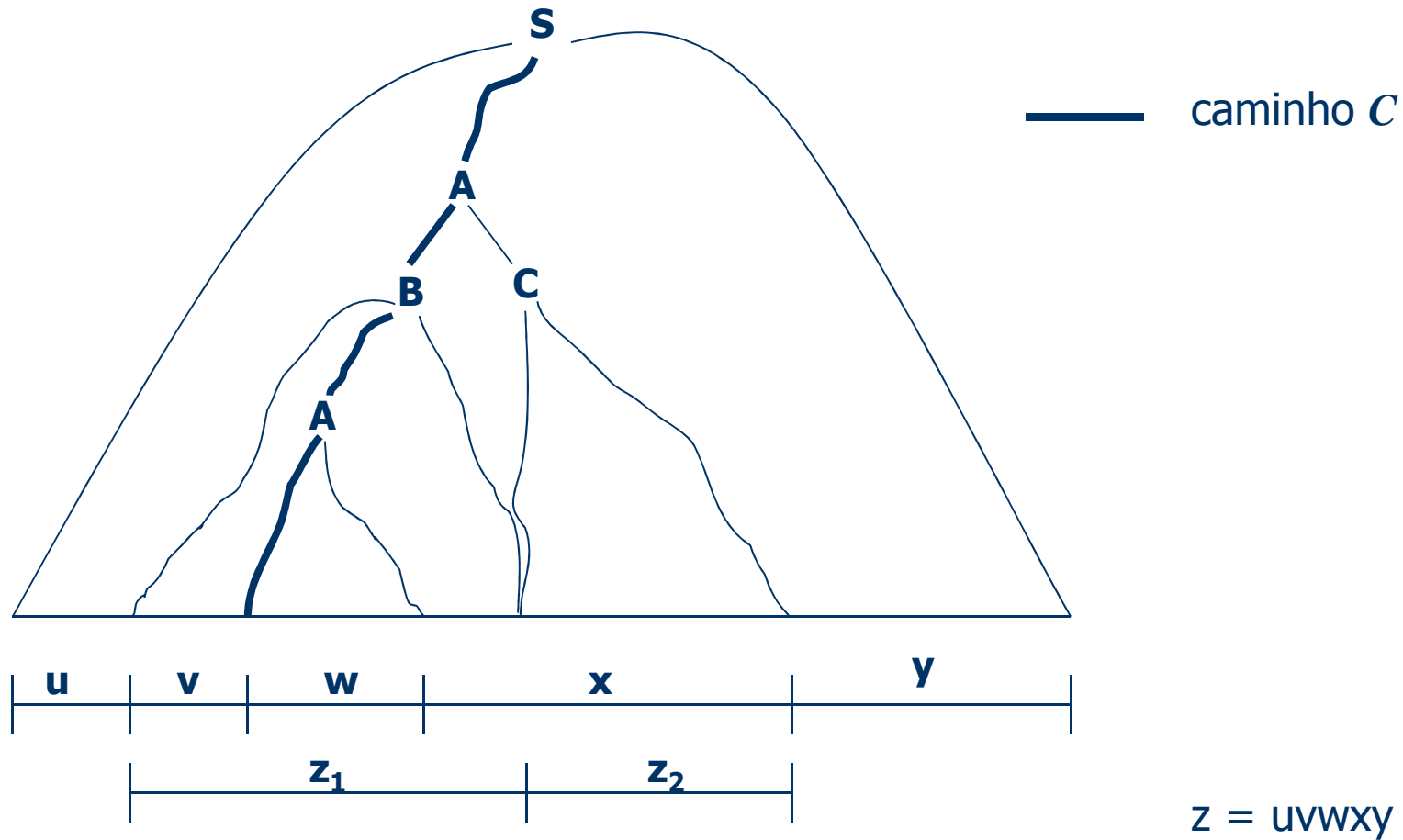
Porque: Se não existir caminho de ordem, no mínimo, $n+2$, o caminho mais longo na árvore de derivação de z terá, no máximo, ordem $n+1$ e, pelo Fato 1, $|z| \leq 2^n$.

Sejam então $k = 2^{n-1}$ e $m = 2^n$ com $n = \#N$. Seja $z \in L$, $|z| > k$. Para a árvore de derivação de z temos:

- pelo Fato 2, algum caminho tem ordem, pelo menos, $n+1$. Seja C o caminho de maior comprimento da árvore de derivação de z . Logo, C tem pelo menos $n+2$ nós.

“Pumping lemma” para LLC

- sejam os últimos $n+2$ nós de C . Logo, existem $n+1$ não terminais neste sub-caminho. Como existem apenas n não terminais, deve existir, pelo menos, um não-terminal repetido. Seja A um não-terminal repetido neste sub-caminho, tal que:



“Pumping lemma” para LLC

Sobre o sub-caminho de C a partir do primeiro A podemos dizer:

- tem, no máximo, $n+2$ nós
- a ordem é de, no máximo, $n+1$
- a partir de A , não existe caminho mais longo do que este.

Logo:

1. $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy = z$
2. $A \Rightarrow^* vAx \Rightarrow^* vwx$. Como a ordem desse caminho é, no máximo, $n+1$, pelo Fato 1, $|vwx| \leq 2^n = m$.
3. $A \Rightarrow BC \Rightarrow^* vwx = z_1 z_2$, onde $B \Rightarrow^* z_1$ e $C \Rightarrow^* z_2$. Então, como G está na FNC, G é Λ -livre e, portanto, $z_1 \neq \Lambda$ e $z_2 \neq \Lambda$. Logo, $x \neq \Lambda$ e, portanto, $vx \neq \Lambda$ (note que foi suposto que a repetição de A se dá no caminho a partir de B . Se a repetição fosse no caminho a partir de C , então $v \neq \Lambda$ e, novamente, $vx \neq \Lambda$).
4. $A \Rightarrow^* vAx \Rightarrow^* vvAxx \Rightarrow^* v^i Ax^i \quad (i \geq 0)$

Portanto: $uv^iwx^iy \in L$.

- ♦ Exemplo: $L = \{ a^n b^n c^n \mid n \geq 0 \}$ não é LLC.

Considere que L é LLC. Então para n suficientemente grande $|a^n b^n c^n| > k$ (k do “pumping lemma”). Logo: $a^n b^n c^n = uvwxy$, $vx \neq \Lambda$ e $uv^iwx^iy \in L$.

Admitindo (sem perda de generalidade) que $v \neq \Lambda$, temos:

“Pumping lemma” para LLC

- a) $v = a^k$. Neste caso, $uv^2wx^2y \notin L$ porque, para equilibrar os números de a’s, b’s e c’s, a subcadeia x deverá ter k b’s e k c’s e, portanto, x^2 vai “embaralhar” os b’s e c’s
(o mesmo argumento vale para $v = b^k$ ou para $v = c^k$).
- b) para qualquer outra forma, v terá pelo menos dois símbolos diferentes (por exemplo, $v = apbq$) e, novamente, $uv^2wx^2y \notin L$ porque existirão símbolos “embaralhados”.

Forma Normal de Greibach

- ♦ Definição: Um não-terminal A numa GLC $G = (N, \Sigma, P, S)$ é recursivo à esquerda se $A \Rightarrow^+ A\beta$, $\beta \in (N \cup \Sigma)^*$. Uma gramática com, pelo menos, um não-terminal recursivo à esquerda é uma gramática recursiva à esquerda.
- ♦ Vamos ver mais à frente que alguns algoritmos de análise sintática não funcionam com gramáticas recursivas à esquerda. Vamos mostrar, no entanto, que toda LLC tem, pelo menos, uma gramática não recursiva à esquerda que a gera.
- ♦ Lema: Seja $G = (N, \Sigma, P, S)$ uma GLC na qual $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \dots \mid \beta_n$ são todas as A -produções de P e nenhum β_i ($i = 1, \dots, n$) começa com A . Seja $G' = (N \cup \{A'\}, \Sigma, P', S)$ onde P' é obtido substituindo-se as A -produções por:
$$A \rightarrow \beta_1 \mid \dots \mid \beta_n \mid \beta_1 A' \mid \dots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 \mid \dots \mid \alpha_m \mid \alpha_1 A' \mid \dots \mid \alpha_m A'.$$

Então: $L(G) = L(G')$.

Gramáticas Recursivas à Esquerda

- ♦ Prova: Exercício!
- ♦ Exemplo:
$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid a$$

Eliminando-se a recursão à esquerda temos:

$$E \rightarrow T \mid TE'$$
$$E' \rightarrow + T \mid + TE'$$
$$T \rightarrow F \mid FT'$$
$$T' \rightarrow * F \mid * FT'$$
$$F \rightarrow (E) \mid a$$

Algoritmo para eliminação da recursão à esquerda

- ♦ Seja $G = (N, \Sigma, P, S)$ uma GLC tal que $N = \{ A_1, \dots, A_n \}$. O algoritmo irá transformar G de tal maneira que suas produções terão a forma $A_i \rightarrow \alpha$, onde α começa com um símbolo terminal ou com A_j , $j > i$, ou então $A_i' \rightarrow \beta$, onde A_i' é um novo não-terminal e β começa com um símbolo de $(N \cup \Sigma)$.
- 1) $i = 1$
 - 2) Sejam as A_i -produções na forma: $A_i \rightarrow A_i\alpha_1 \mid \dots \mid A_i\alpha_m \mid \beta_1 \mid \dots \mid \beta_p$ onde nenhum β_k ($k = 1, \dots, p$) começa com A_j , $j \leq i$ (é sempre possível deixar as A_i -produções nessa forma). Trocar as A_i -produções por:

Gramáticas Recursivas à Esquerda

$$A_i \rightarrow \beta_1 \mid \dots \mid \beta_p \mid \beta_1 A'_i \mid \dots \mid \beta_p A'_i$$

$$A'_i \rightarrow \alpha_1 \mid \dots \mid \alpha_m \mid \alpha_1 A'_i \mid \dots \mid \alpha_p A'_i$$

onde A'_i é um novo não-terminal (após essa troca, todas as A_i -produções começam com um símbolo terminal ou com A_j , $j > i$).

Se $i = n$, parar. Caso contrário, fazer: $i = i + 1$, $j = 1$.

- 3) Sejam $A_j \rightarrow \gamma_1 \mid \dots \mid \gamma_q$ todas as A_j -produções. Trocar toda produção $A_i \rightarrow A_j \delta$ pelas produções:

$$A_j \rightarrow \gamma_1 \delta \mid \dots \mid \gamma_q \delta$$

(após essa troca, todas as A_j -produções começam com um símbolo terminal ou com A_k , $k > j$)

Se $j = i-1$, voltar para (2). Caso contrário, fazer: $j = j + 1$ e voltar para (3).

- Exemplo: Eliminar a recursão à esquerda de:

$$A_1 \rightarrow A_2 A_3 \mid a$$

$$A_2 \rightarrow A_3 A_1 \mid A_1 b$$

$$A_3 \rightarrow A_1 A_2 \mid A_3 A_3 \mid a$$

(passo 2 - $i = 1$): as A_1 -produções já estão na forma desejada.

(passo 3 - $i = 2$; $j = 1$): $A_2 \rightarrow A_3 A_1 \mid A_2 A_3 b \mid ab$

(passo 2 - $i = 2$): $A_2 \rightarrow A_3 A_1 \mid ab \mid A_3 A_1 A'_2 \mid ab A'_2$

$$A'_2 \rightarrow A_3 b \mid A_3 b A'_2$$

Forma Normal de Greibach

(passo 3 - i = 3; j = 1): $A_3 \rightarrow A_3A_3 \mid a \mid A_2A_3A_2 \mid aA_2$

(passo 3 - i = 3; j = 2):

$A_3 \rightarrow A_3A_3 \mid a \mid aA_2 \mid A_3A_1A_3A_2 \mid abA_3A_2 \mid A_3A_1A_2'A_3A_2 \mid abA_2'A_3A_2$

(passo 2 - i = 3):

$A_3 \rightarrow a \mid aA_2 \mid abA_3A_2 \mid abA_2'A_3A_2 \mid aA_3 \mid aA_2A_3' \mid abA_3A_2A_3' \mid abA_2'A_3A_2A_3'$
 $A_3' \rightarrow A_3 \mid A_1A_3A_2 \mid A_1A_2'A_3A_2 \mid A_3A_3' \mid A_1A_3A_2A_3' \mid A_1A_2'A_3A_2A_3'$

ou seja:

$A_1 \rightarrow A_2A_3 \mid a$

$A_2 \rightarrow A_3A_1 \mid ab \mid A_3A_1A_2' \mid abA_2'$

$A_2' \rightarrow A_3b \mid A_3bA_2'$

$A_3 \rightarrow a \mid aA_2 \mid abA_3A_2 \mid abA_2'A_3A_2 \mid aA_3 \mid aA_2A_3' \mid abA_3A_2A_3' \mid abA_2'A_3A_2A_3'$

$A_3' \rightarrow A_3 \mid A_1A_3A_2 \mid A_1A_2'A_3A_2 \mid A_3A_3' \mid A_1A_3A_2A_3' \mid A_1A_2'A_3A_2A_3'$

- ♦ Definição: Uma GLC $G = (N, \Sigma, P, S)$ está na forma normal de Greibach se G é Λ -livre e toda produção de P é da forma $A \rightarrow a\alpha$; $a \in \Sigma$, $\alpha \in N^*$.

Forma Normal de Greibach

Algoritmo para conversão para a forma normal de Greibach

- Seja $G = (N, \Sigma, P, S)$ uma GLC não recursiva à esquerda. Seja $<$ uma relação de ordenação parcial tal que toda A-produção começa com um terminal ou com um não-terminal B tal que $A < B$. Seja $N = \{ A_1, \dots, A_n \}$ tal que $A_1 < A_2 < \dots < A_n$.
- 1. para $i = n-1$ até 1 fazer: trocar toda produção da forma $A_i \rightarrow A_j \alpha$, com $A_i < A_j$ por $A_i \rightarrow \beta_1 \alpha \mid \dots \mid \beta_m \alpha$, onde $A_j \rightarrow \beta_1 \mid \dots \mid \beta_m$ são todas as A_j -produções.
- 2. após as trocas do passo 1, todas as produções (exceto $S \rightarrow \Lambda$, possivelmente) começam com um terminal. Para cada produção $A \rightarrow aX_1 \dots X_k$, trocar os X_j que são terminais por novos não-terminais X'_j ($j = 1, \dots, k$) e adicionar a produção $X'_j \rightarrow X_j$.

- Exemplo: Seja a GLC não recursiva à esquerda:

$$\begin{aligned} E &\rightarrow T \mid TE' \\ E' &\rightarrow +T \mid +TE' \\ T &\rightarrow F \mid FT' \\ T' &\rightarrow *F \mid *FT' \\ F &\rightarrow (E) \mid a \end{aligned}$$

Seja a ordenação sobre os não-terminais: $E' < E < T' < T < F$. Como F é o "maior" na ordenação, todas as F-produções começam com um terminal.

Forma Normal de Greibach

$$E \rightarrow T \mid TE'$$

$$E' \rightarrow +T \mid +TE'$$

$$T \rightarrow F \mid FT'$$

$$T' \rightarrow *F \mid *FT'$$

$$F \rightarrow (E) \mid a$$

Continuando:

$$T' \rightarrow *F \mid *FT'$$

$$E \rightarrow (E) \mid a \mid (E)T' \mid aT' \mid (E)E' \mid aE' \mid (E)T'E' \mid aT'E'$$

$$E' \rightarrow +T \mid +TE'$$

Após essas transformações, todas as produções começam com um terminal. Basta agora transformar os outros símbolos do lado direito das produções em não-terminais, introduzindo a produção $P \rightarrow)$. A gramática na forma normal de Greibach resultante será:

$$E \rightarrow (EP \mid a \mid (EPT' \mid aT' \mid (EPE' \mid aE' \mid (EPT'E' \mid aT'E'$$

$$E' \rightarrow +T \mid +TE'$$

$$T \rightarrow (EP \mid a \mid (EPT' \mid aT'$$

$$T' \rightarrow *F \mid *FT'$$

$$F \rightarrow (EP \mid a$$

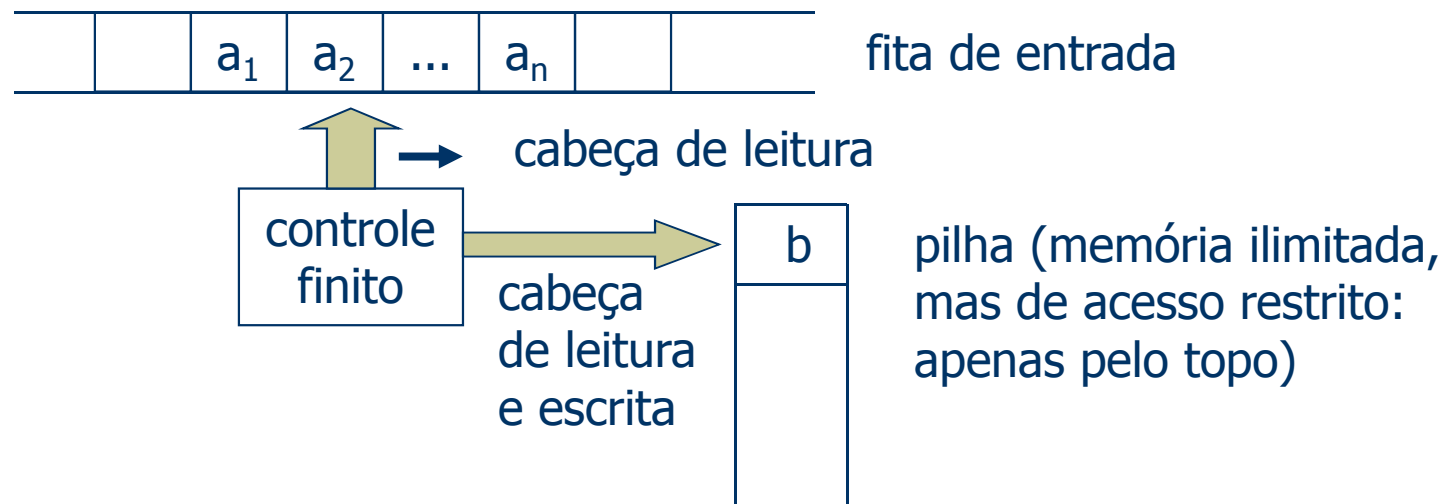
$$P \rightarrow)$$

Ordenação: $E' < E < T' < T < F$

O próximo na ordenação é T. Substituindo F nas T-produções temos:

$$T \rightarrow (E) \mid a \mid (E)T' \mid aT'$$

Autômatos com Pilha (“Pushdown automata”)



Intuitivamente:

Dependendo do:

- estado do controle finito
- símbolo que está sendo lido na fita de entrada
- símbolo que está no topo da pilha

o autômato com pilha irá:

- mudar de estado
- escrever um número finito de símbolos na pilha (escrever o símbolo Λ corresponde a apagar o símbolo no topo)
- mover sua cabeça leitora uma posição para a direita.

Autômatos com Pilha

- ♦ Definição: Um autômato com pilha é uma septupla $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ onde:
 - Q - conjunto finito não vazio de estados
 - Σ - alfabeto de entrada
 - Γ - alfabeto da pilha
 - $q_0 \in Q$ - estado inicial
 - $z_0 \in \Gamma$ - símbolo inicial da pilha
 - $F \subseteq Q$ - conjunto de estados finais
 - $\delta : Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma \rightarrow \mathbf{P}(Q \times \Gamma^*)$ (não-determinístico)
- ♦ Definição: Uma configuração de um autômato com pilha $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ é um elemento de $Q \times \Sigma^* \times \Gamma^*$.
- ♦ Definição: Transição no autômato com pilha:
$$(q, ax, zw) \mapsto (q', x, yw) \Leftrightarrow (q', y) \in \delta(q, a, z)$$
com: $q, q' \in Q; a \in \Sigma \cup \{\Lambda\}; x \in \Sigma^*; w \in \Gamma^*; z \in \Gamma; y \in \Gamma^*$

Devemos observar que, pela definição acima, o autômato com pilha não faz transições com a pilha vazia. Observe também que o autômato com pilha pode efetuar Λ -movimentos, que corresponde a mudar de estado e mudar o conteúdo da pilha, sem ler o símbolo na fita de entrada (isto é, sem mover sua cabeça leitora para a direita).

Autômatos com Pilha

- ♦ Definição: A linguagem aceita por $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, por estado final, é: $L(A) = \{ w \in \Sigma^* \mid (q_0, w, z_0) \mapsto^* (q, \Lambda, x); q \in F; x \in \Gamma^* \}$
- ♦ Definição: A linguagem aceita por $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, por pilha vazia, é: $N(A) = \{ w \in \Sigma^* \mid (q_0, w, z_0) \mapsto^* (q, \Lambda, \Lambda); q \in Q \}$
- ♦ Exemplo: Construir um autômato com pilha que aceita $L = \{ 0^n 1^n \mid n \geq 0 \}$
 $A = (\{q_0, q_1, q_2\}, \{0, 1\}, \{z, 0\}, \delta, q_0, z, \{q_0\})$, com δ dada por:
$$\begin{aligned} \delta(q_0, 0, z) &= \{ (q_1, 0z) \} \\ \delta(q_1, 0, 0) &= \{ (q_1, 00) \} \end{aligned} \quad \text{(empilha 0's)}$$

$$\begin{aligned} \delta(q_1, 1, 0) &= \{ (q_2, \Lambda) \} \\ \delta(q_2, 1, 0) &= \{ (q_2, \Lambda) \} \end{aligned} \quad \text{(para cada 1 encontrado, desempilha um 0)}$$

$$\delta(q_2, \Lambda, z) = \{ (q_0, z) \} \quad \text{(reconhecer por estado final)}$$
- ♦ Exercício: Mostrar que $L = L(A)$!
- ♦ Definição: Um autômato com pilha $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$ é determinístico se, para todo $(q, a, z) \in Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma$, temos:
 1. $|\delta(q, a, z)| \leq 1$
 2. se $\delta(q, \Lambda, z) \neq \emptyset$ então $\delta(q, a, z) = \emptyset, \forall a \in \Sigma$

Observe que o autômato A do exemplo acima é determinístico.

Autômatos com Pilha

- ♦ Considere, no entanto, a linguagem:

$$L = \{ ww^r \mid w \in \{0, 1\}^*; w^r \text{ é o reverso de } w \}$$

Intuitivamente, um autômato com pilha para recolher L deverá ser não determinístico porque durante o processamento de uma cadeia ww^r não haverá maneira de saber quando termina a cadeia w e começa a cadeia w^r . Assim, para cada símbolo lido, o autômato deverá prever duas possíveis situações:

- a) a cadeia w ainda não terminou
- b) a cadeia w terminou e os próximos símbolos serão de w^r

Para a linguagem $L' = \{ wcw^r \mid w \in \{0, 1\}^* \}$, é possível construir um autômato com pilha determinístico A tal que $L' = L(A)$.

- ♦ Exercício: Construir um autômato com pilha que reconhece L .
- ♦ Definição: Seja Σ um alfabeto e A um autômato com pilha.
 $L_\Sigma = \{ L \subseteq \Sigma^* \mid L = L(A) \}$ (conjunto das linguagens aceitas por estado final)
 $N_\Sigma = \{ L \subseteq \Sigma^* \mid L = N(A) \}$ (conjunto das linguagens aceitas por pilha vazia)
- ♦ Proposição: $L_\Sigma = N_\Sigma$
- ♦ Lema: Se $L = L(A)$ para algum autômato com pilha A , então existe autômato com pilha B tal que $L = N(B)$.

Autômatos com Pilha

- ♦ Prova: Seja $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$.
 Construir $B = (Q \cup \{d, q_0'\}, \Sigma, \Gamma \cup \{y_0\}, \delta', q_0', y_0, \emptyset)$ com δ' dada por:
 - a) $\delta'(q_0', \Lambda, y_0) = \{(q_0, z_0 y_0)\}$
 (o autômato B se prepara para simular A colocando-se na mesma situação inicial de A, a menos de y_0 na pilha)
 - b) $(q', \alpha) \in \delta(q, a, z) \Rightarrow (q', \alpha) \in \delta'(q, a, z); q \in Q; a \in \Sigma \cup \{\Lambda\}; z \in \Gamma$
 (ou seja, B simula A)
 - c) $(d, \Lambda) \in \delta'(q, \Lambda, z); q \in F; z \in \Gamma$
 - d) $\delta'(d, \Lambda, z) = \{(d, \Lambda)\}$
 (ou seja, sempre que o autômato A entrar num estado final, o autômato B divide-se em duas cópias: uma delas continua a simular A (transições devidas a (b)) e a outra esvazia a pilha (transições devidas a (c) e (d)).

Então:

$$\begin{aligned}
 x \in L(A) &\Leftrightarrow (q_0, x, z_0) \xrightarrow{*}_A (q, \Lambda, w); q \in F; w \in \Gamma^* \\
 &\Leftrightarrow (q_0', x, y_0) \xrightarrow{*}_B (q_0, x, z_0 y_0) && \text{(devido a (a))} \\
 &\quad \quad \quad \xrightarrow{*}_B (q, \Lambda, w y_0) && \text{(devido a (b))} \\
 &\quad \quad \quad \xrightarrow{*}_B (d, \Lambda, \Lambda) && \text{(devido a (c) e (d))} \\
 &\Leftrightarrow x \in N(B).
 \end{aligned}$$

Logo: $L(A) = N(B)$ e, portanto, $L_\Sigma \subseteq N_\Sigma$

Autômatos com Pilha

- ♦ Lema: Para todo autômato com pilha A existe um autômato com pilha B tal que $L(B) = N(A)$.
- ♦ Prova: Seja $A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$.
Construir $B = (Q \cup \{f, q_0'\}, \Sigma, \Gamma \cup \{y_0\}, \delta', q_0', y_0, \{f\})$ com δ' dada por:
 - a) $\delta'(q_0', \Lambda, y_0) = \{(q_0, z_0 y_0)\}$
 - b) $\delta'(q, a, \alpha) = \delta(q, a, \alpha); q \in Q; a \in \Sigma \cup \{\Lambda\}; \alpha \in \Gamma$
 - c) $\delta'(q, \Lambda, y_0) = \{(f, y_0)\}; q \in Q$

Então:

$$\begin{aligned}x \in N(A) &\Leftrightarrow (q_0, x, z_0) \xrightarrow{*}_A (q, \Lambda, \Lambda) \\&\Leftrightarrow (q_0', x, y_0) \xrightarrow{*}_B (q_0, x, z_0 y_0) && \text{(devido a (a))} \\&\quad \xrightarrow{*}_B (q, \Lambda, y_0) && \text{(devido a (b))} \\&\quad \xrightarrow{*}_B (f, \Lambda, y_0) && \text{(devido a (c))} \\&\Leftrightarrow x \in L(B).\end{aligned}$$

Logo: $L(B) = N(A)$ e, portanto, $N_\Sigma \subseteq L_\Sigma$

Então, pelos dois lemas anteriores, fica provada a proposição $L_\Sigma = N_\Sigma$, ou seja, a classe das linguagens aceitas por estado final e a classe das linguagens aceitas por pilha vazia são a mesma classe.

- ♦ Proposição: $L_\Sigma = N_\Sigma = \{ L \subseteq \Sigma^* \mid L \text{ é linguagem livre de contexto} \}$ (ou seja, a classe das linguagens reconhecidas por autômatos com pilha é a classe das linguagens livres de contexto).

Autômatos com Pilha

- Lema: Se L é uma LLC então $L = N(A)$ para algum autômato com pilha A .
- Prova: Seja $G = (N, \Sigma, P, S)$ uma GLC tal que $L = L(G)$.

Construir $A = (\{q\}, \Sigma, N \cup \Sigma, \delta, q, S, \emptyset)$ com δ dada por:

$$a) \quad \delta(q, \Lambda, B) = \{ (q, x) \mid B \rightarrow x \in P \}$$

$$b) \quad \delta(q, a, a) = \{ (q, \Lambda) \}; \quad a \in \Sigma$$

Será provado, inicialmente, por indução no comprimento da derivação, que:

$$[S \Rightarrow^* u\alpha ; u \in \Sigma^*, \alpha \in (N \cup \Sigma)^*] \Rightarrow [\forall w \in \Sigma^*, (q, uw, S) \vdash_A^* (q, w, \alpha)]$$

$$\text{Base: } [S \Rightarrow^0 S] \Rightarrow [\forall w \in \Sigma^*, (q, w, S) \vdash_A^0 (q, w, S)] \text{ (trivialmente)}$$

Hipótese indutiva: a proposição vale para derivações de comprimento n .

$$\text{Passo da indução: Seja } S \Rightarrow^n u_1 \underbrace{Au_3\alpha}_{\alpha'} \Rightarrow \underbrace{u_1u_2u_3\alpha}_u$$

Então:

$$\forall w \in \Sigma^*, (q, u_1u_2u_3w, S) \vdash^* (q, u_2u_3w, Au_3\alpha) \quad (\text{hipótese indutiva})$$

$$\vdash (q, u_2u_3w, u_2u_3\alpha) \quad (\text{de (a) e de } A \rightarrow u_2 \in P)$$

$$\vdash^* (q, w, \alpha) \quad (\text{de (b)})$$

Particularizando esse resultado para $\alpha = w = \Lambda$, temos:

$$[S \Rightarrow^* u ; u \in \Sigma^*] \Rightarrow [(q, u, S) \vdash_A^* (q, \Lambda, \Lambda)]$$

e, portanto, $L(G) \subseteq N(A)$.

Autômatos com Pilha

- Exercício: Mostrar, por indução no número de transições, que:

$$[(q, uw, S)_A \mapsto^* (q, w, \alpha)] \Rightarrow [S \Rightarrow^* u\alpha]$$

Particularizando esse resultado para $\alpha = w = \Lambda$, temos que $N(A) \subseteq L(G)$.

Portanto: $L = L(G) = N(A)$.

- Exemplo:
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow x \mid (E) \end{aligned}$$

O reconhecedor para a linguagem gerada por essa gramática pode ser construído a partir do lema anterior, ou seja:

$A = (\{q\}, \Sigma, \Sigma \cup \{E, T, F\}, \delta, q, E, \emptyset)$ onde $\Sigma = \{x, +, *, (,)\}$ e δ dada por:

$$\delta(q, \Lambda, E) = \{(q, E+T), (q, T)\}$$

$$\delta(q, \Lambda, T) = \{(q, T*F), (q, F)\}$$

$$\delta(q, \Lambda, F) = \{(q, x), (q, (E))\}$$

$$\delta(q, a, a) = \{(q, \Lambda)\}; \quad a \in \Sigma$$

Seja a expressão: $x + x * x$. Temos as seguintes transições de A :

$$\begin{aligned} (q, x+x*x, E) &\mapsto (q, x+x*x, E+T) \mapsto (q, x+x*x, T+T) \mapsto (q, x+x*x, F+T) \mapsto \\ (q, x+x*x, x+T) &\mapsto^2 (q, x*x, T) \mapsto (q, x*x, T*F) \mapsto (q, x*x, F*F) \mapsto \\ (q, x*x, x*F) &\mapsto^2 (q, x, F) \mapsto (q, x, x) \mapsto (q, \Lambda, \Lambda) \end{aligned}$$

Autômatos com Pilha

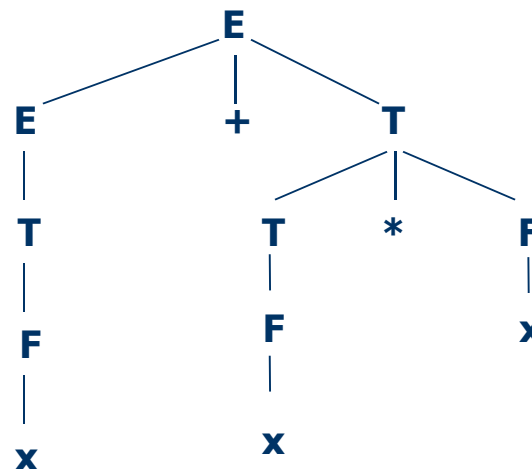
- Observando as transições de A:

$(q, x+x^*x, E) \mapsto (q, x+x^*x, E+T) \mapsto (q, x+x^*x, T+T) \mapsto (q, x+x^*x, F+T) \mapsto$
 $(q, x+x^*x, x+T) \mapsto^2 (q, x^*x, T) \mapsto (q, x^*x, T^*F) \mapsto (q, x^*x, F^*F) \mapsto$
 $(q, x^*x, x^*F) \mapsto^2 (q, x, F) \mapsto (q, x, x) \mapsto (q, \Lambda, \Lambda)$

notamos que a fita de entrada contém, a cada instante, a parte da cadeia de entrada que falta ser analisada e que o conteúdo da pilha simula uma derivação mais à esquerda:

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow x+T \Rightarrow x+T^*F \Rightarrow x+F^*F \Rightarrow x+x^*F \Rightarrow x+x^*x$

Esse autômato com pilha é conhecido como reconhecedor descendente ou analisador sintático descendente ("top-down parser") porque como simula derivações mais à esquerda, constrói a árvore sintática da cadeia de entrada, de cima para baixo. Note que:



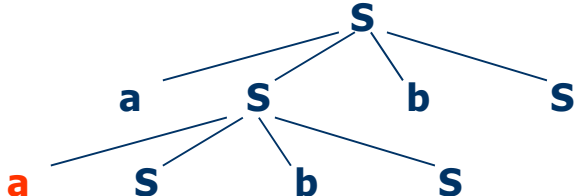
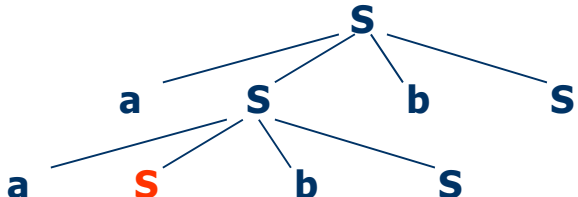


Autômatos com Pilha

- ♦ Entretanto, como podemos observar da definição da função δ , o autômato A é não determinístico, sendo que as transições mostradas foram escolhidas “convenientemente” a cada passo. Uma questão interessante: “É possível automatizar esse processo?”. Resposta: Sim. Algoritmo: análise sintática descendente com retorno (“top-down backtrack parsing”):
 1. começar a árvore sintática pelo símbolo inicial da pilha (que é o símbolo inicial da gramática). Esse símbolo é o nó ativo inicial. Executar, em seguida, os passos 2 e 3 recursivamente.
 2. se o nó ativo é um não-terminal A, escolher a primeira A-produção ainda não utilizada, $A \rightarrow X_1X_2...X_k$ e criar na árvore sintática, $X_1, X_2, ..., X_k$ como descendentes diretos de A. Marcar o primeiro desses descendentes (o mais à esquerda) como ativo.
 3. se o nó ativo é um terminal a, então compará-lo com o símbolo atual de entrada (símbolo que está sendo lido na fita de entrada). Se eles forem iguais, então tornar ativo o símbolo imediatamente à direita de a na árvore e avançar a cabeça leitora para o próximo símbolo de entrada. Se eles forem diferentes, retornar ao último não-terminal A expandido para o qual existe uma A-produção ainda não utilizada e atualizar o símbolo de entrada.
 4. se não existe nó ativo e todos os símbolos de entrada foram lidos, então a análise sintática termina com sucesso (a cadeia foi reconhecida). Caso contrário, rejeitar a cadeia (existe um erro sintático na cadeia).

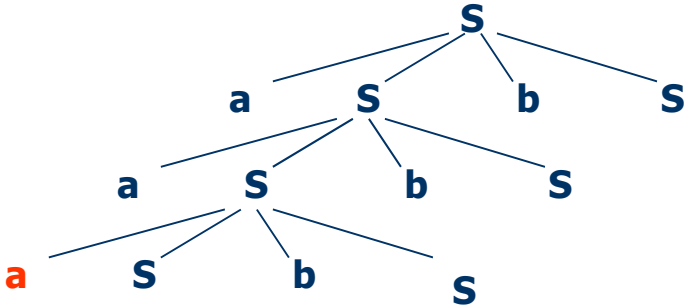
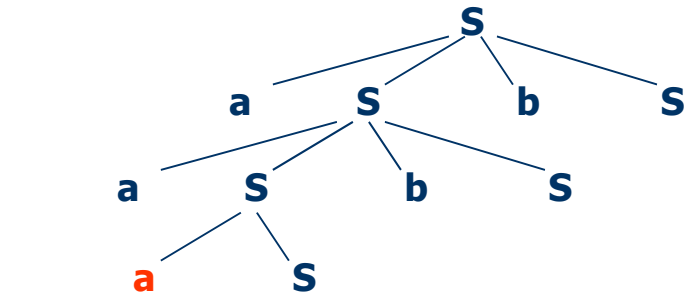
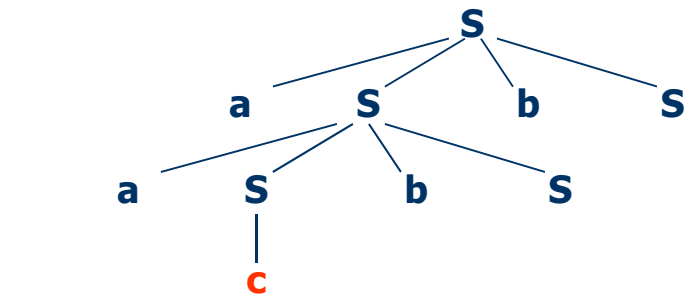
Autômatos com Pilha

- Exemplo: Seja a gramática: $S \rightarrow aSbS \mid aS \mid c$.
Seja a cadeia de entrada: aacbc.

Árvore	Cadeia	Observações
S	aacbc	em vermelho : nó ativo e símbolo de entrada que está sendo lido.
	aacbc	escolhe-se a primeira alternativa para expandir S.
	aa c bc	o símbolo de entrada "casa" com o nó da árvore; o próximo símbolo é considerado.
	aa c bc	escolhe-se a primeira alternativa para expandir S.
	aac c bc	"casamento" de símbolos; passar ao próximo nó da árvore.

Autômatos com Pilha

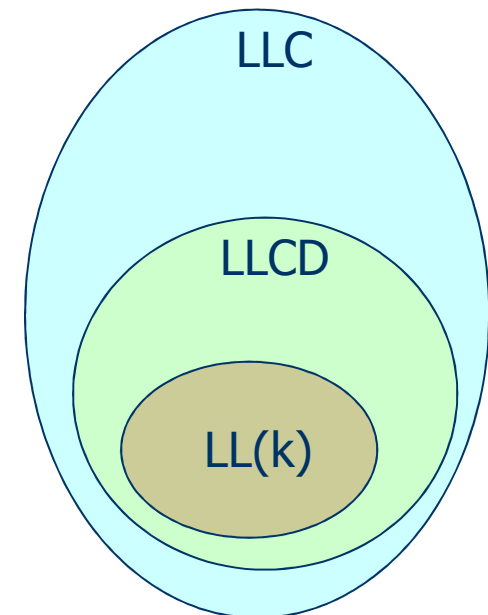
♦ **continuando...**

Árvore	Cadeia	Observações
	aa c bc	o símbolo S foi expandido; a escolha, no entanto, é incorreta porque não há "casamento" de terminais; tentar a próxima alternativa para a expansão de S.
	aa c bc	essa alternativa também é incorreta; tentar a próxima alternativa.
	aa c bc	"casamento" ...

Autômatos com Pilha

- ♦ Para mais detalhes sobre esse algoritmo ver:
AHO & ULLMAN - "The theory of parsing, translation and compiling", vol. I, p. 289-291.
- ♦ Esse algoritmo apresenta os seguintes inconvenientes:
 - a) o número de passos necessários para uma análise pode ser muito grande (o algoritmo tem complexidade assintótica exponencial);
 - b) a capacidade de indicar erros em cadeias de entrada mal formadas é muito pequena.
- ♦ Existe um grande interesse, para a construção de compiladores, em analisadores sintáticos eficientes, ou seja, analisadores que evitem esses inconvenientes.

A seguir serão estudadas classes de GLC para as quais podemos evitar o "backtracking" e construir reconhecedores eficientes (analisadores com complexidade linear no espaço e no tempo). Essas classes de gramáticas - $LL(k)$ - no entanto, não geram todas as LLC, apesar de existir forte evidência de que essas gramáticas sejam adequadas para especificar as características sintáticas de linguagens de programação.



Gramáticas LL(k)

- ♦ Notação: LL(k) - classe das linguagens geradas por gramáticas LL(k).

- ♦ Intuitivamente:

Seja $G = (N, \Sigma, P, S)$ uma gramática não ambígua e $w = a_1 \dots a_n$ uma cadeia de $L(G)$. Existe então uma única sequência de formas sentenciais obtidas por derivação mais à esquerda, $\alpha_0, \dots, \alpha_m$ tal que:

$$S = \alpha_0; \alpha_i \Rightarrow \alpha_{i+1} \ (0 \leq i < m); \alpha_m = w$$

Seja $\alpha_i = a_1 \dots a_j A \beta$. Se α_{i+1} puder ser determinada conhecendo-se apenas $a_1 \dots a_j$ (isto é, a parte da cadeia de entrada que já foi lida), A (isto é, o não-terminal a ser substituído em seguida) e $a_{j+1} \dots a_{j+k}$ para algum inteiro k , então a gramática G é uma gramática LL(k).

- ♦ Definição: Seja $G = (N, \Sigma, P, S)$ uma GLC. Define-se o conjunto $\text{First}_k(\alpha)$, de todos os prefixos de comprimento k (ou das cadeias de comprimento menor do que k) que podem ser derivados de α por:

$$\text{First}_k(\alpha) = \{ x \in \Sigma^* \mid \alpha \Rightarrow^* x\beta, \mid x \mid = k \text{ ou } \alpha \Rightarrow^* x, \mid x \mid < k \}$$

- ♦ Definição: Seja $G = (N, \Sigma, P, S)$ uma GLC. Dizemos que G é LL(k), para algum inteiro k , se sempre que existirem duas derivações mais à esquerda:

$$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx \quad \text{e}$$

$$S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy \quad \text{tais que } \text{First}_k(x) = \text{First}_k(y), \text{ então } \beta = \gamma.$$

Gramáticas LL(k)

- ♦ Exemplo: Seja G com produções:
 $S \rightarrow aAS \mid b$
 $A \rightarrow a \mid bSA$

Neste caso, G é LL(1) porque dado o não-terminal mais a esquerda de uma forma sentencial (S ou A) e o próximo símbolo de entrada (a ou b, pois do contrário a cadeia é mal formada) só existe uma produção de G capaz de derivar esse símbolo.

- ♦ Exercício: Mostrar (formalmente) que G é LL(1).
 - ♦ Definição: Uma GLC $G = (N, \Sigma, P, S)$ é LL(1) simples se G é Λ -livre e dado um par (A, a) com $A \in N$ e $a \in \Sigma$, existe no máximo uma produção $A \rightarrow a\alpha$, $\alpha \in (N \cup \Sigma)^*$.
 - ♦ Exemplos:
1. Seja G com produções: $S \rightarrow \Lambda \mid abA$
 $A \rightarrow Saa \mid b$

Vamos mostrar que G é LL(2). Seja:

$$S \Rightarrow^* wS\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx \quad \text{e} \quad S \Rightarrow^* wS\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy$$

onde x e y começam com os mesmos 2 símbolos. As únicas duas maneiras de S aparecer numa forma sentencial são:

- a) $w = \alpha = \Lambda$ (ou seja, $S \Rightarrow^* wS\alpha$ é, na verdade, $S \Rightarrow^0 S$)
- b) a última produção usada para se chegar na forma sentencial $wS\alpha$ foi:
 $A \rightarrow Saa$. Logo: $\alpha = \Lambda$ ou α começa com aa.

Gramáticas LL(k)

Sejam as derivações: $wS\alpha \Rightarrow w\beta\alpha$ e $wS\alpha \Rightarrow w\gamma\alpha$

Como x e y devem começar com os mesmos 2 símbolos, deve-se usar $S \rightarrow \Lambda$ ou $S \rightarrow abA$ em ambas as derivações, pois do contrário, como $\alpha = \Lambda$ ou α começa com aa , uma cadeia (por exemplo, x) seria Λ ou começaria com aa , ao passo que a outra começaria com ab .

Logo:

se for usada a produção $S \rightarrow \Lambda$ teremos que $\beta = \gamma = \Lambda$

se for usada a produção $S \rightarrow abA$ teremos que $\beta = \gamma = abA$

- **Exercício:** Completar a prova, isto é, considerar o caso:

$$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx \quad \text{e} \quad S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy$$

2. Seja G com produções: $S \rightarrow A \mid B$

$$A \rightarrow aAb \mid 0$$

$$B \rightarrow aBbb \mid 1$$

Vamos mostrar (informalmente) que G não é LL(k), para qualquer k .

Ao analisar uma cadeia de a 's, não é possível saber qual a produção $S \rightarrow A$ ou $S \rightarrow B$ foi usada no início da derivação dessa cadeia, até que seja encontrado um 0 ou um 1. Logo, como a cadeia de a 's pode ser arbitrariamente grande, para qualquer inteiro k , será sempre possível escrever:

$$S \Rightarrow^0 S \Rightarrow A \Rightarrow^* a^k 0 b^k \quad \text{e} \quad S \Rightarrow^0 S \Rightarrow B \Rightarrow^* a^k 1 b^{2k}$$

Gramáticas LL(k)

- ♦ Teorema: Seja $G = (N, \Sigma, P, S)$ uma GLC. G é LL(k) \Leftrightarrow se $A \rightarrow \beta$ e $A \rightarrow \gamma$ são produções distintas de P , então $\text{First}_k(\beta\alpha) \cap \text{First}_k(\gamma\alpha) = \emptyset$ para todo $wA\alpha$ tal que $S \Rightarrow^* wA\alpha$.

- ♦ Prova:

(\Leftarrow) Sejam $w, A, \alpha, \beta, \gamma$ como no enunciado. Seja x um elemento de $\text{First}_k(\beta\alpha) \cap \text{First}_k(\gamma\alpha)$. Logo, pela definição de First_k temos:

$$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wxy$$

$$S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wxz$$

para algum y e z (se $|x| < k$ então $y = z = \Lambda$). Portanto, como $\beta \neq \gamma$, G não é LL(k).

(\Rightarrow) **Exercício!**

- ♦ Seja $G = (N, \Sigma, P, S)$ uma GLC, Λ -livre. Queremos determinar se G é LL(1). Pelo teorema acima temos:

G é LL(1) \Leftrightarrow para todo não-terminal A , com A -produções $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$, $\text{First}_1(\alpha_1), \dots, \text{First}_1(\alpha_n)$ são disjuntos dois a dois.

- ♦ Exemplo:
 $S \rightarrow aS$
 $S \rightarrow a$ não é LL(1), pois: $\text{First}_1(aS) = \{a\} = \text{First}_1(a)$.

E se a gramática não for Λ -livre?

Gramáticas LL(k)

- ♦ Definição: Seja $G = (N, \Sigma, P, S)$ uma GLC. Seja $\beta \in (N \cup \Sigma)^*$. Define-se o conjunto $\text{Follow}_k(\beta)$ (conjunto de cadeias de símbolos terminais que podem aparecer imediatamente à direita de β numa forma sentencial) como:
$$\text{Follow}_k(\beta) = \{ w \mid S \Rightarrow^* \alpha\beta\gamma \text{ e } w \in \text{First}_k(\gamma) \}$$
- ♦ Exemplo: $\text{Follow}_1(A)$ é o conjunto de símbolos terminais que podem aparecer imediatamente à direita de A numa forma sentencial qualquer. Além disso, se αA é uma forma sentencial possível, então $\Lambda \in \text{Follow}_1(A)$.
- ♦ Teorema: Uma GLC, $G = (N, \Sigma, P, S)$, é LL(1) \Leftrightarrow para cada não-terminal A , se $A \rightarrow \beta$ e $A \rightarrow \gamma$ são produções distintas de P , então:
$$\text{First}_1(\beta\text{Follow}_1(A)) \cap \text{First}_1(\gamma\text{Follow}_1(A)) = \emptyset$$
- ♦ Prova: Exercício!
- ♦ Definição: Seja G uma GLC. Sejam $A \rightarrow \beta$ e $A \rightarrow \gamma$, A -produções distintas de G . Se $\text{First}_k(\beta\text{Follow}_k(A)) \cap \text{First}_k(\gamma\text{Follow}_k(A)) = \emptyset$ então G é denominada gramática fortemente LL(k).
- ♦ Observe que, numa gramática fortemente LL(k), dadas duas derivações mais à esquerda:
$$S \Rightarrow^* wA\alpha_1 \Rightarrow w\beta\alpha_1 \Rightarrow^* wx$$
$$S \Rightarrow^* wA\alpha_2 \Rightarrow w\gamma\alpha_2 \Rightarrow^* wy$$
tal que os primeiros k símbolos de x são iguais aos primeiros k símbolos de y , então $\beta = \gamma$. Do teorema acima, nota-se que toda gramática LL(1) é uma gramática fortemente LL(1). Entretanto, para $k > 1$, existem gramáticas LL(k) que não são fortemente LL(k).

Gramáticas LL(k)

- ♦ Exemplo: Seja a gramática G: $S \rightarrow aAaa \mid bAba$
 $A \rightarrow b \mid \Lambda$

a) Verificação se G é LL(1):

1) $S \rightarrow aAaa$

$S \rightarrow bAba$

$S \Rightarrow^0 S \quad (w = \alpha = \Lambda)$

$\text{First}_1(\beta\alpha) = \text{First}_1(aAaa) = \{a\}$

$\text{First}_1(\gamma\alpha) = \text{First}_1(bAba) = \{b\}$

Logo: $\text{First}_1(\beta\alpha) \cap \text{First}_1(\gamma\alpha) = \emptyset$ para as S-produções.

2) $A \rightarrow b$

$A \rightarrow \Lambda$

Uma possível forma sentencial na qual A aparece é bAba ($S \Rightarrow bAba$).

Nesse caso ($w = b$ e $\alpha = ba$) temos:

$\text{First}_1(\beta\alpha) = \text{First}_1(bba) = \{b\}$

$\text{First}_1(\gamma\alpha) = \text{First}_1(ba) = \{b\}$

Logo: $\text{First}_1(\beta\alpha) \cap \text{First}_1(\gamma\alpha) \neq \emptyset$ para as A-produções e, portanto, G não é LL(1).

- Intuitivamente: estando A no topo da pilha e olhando b na cadeia de entrada, não é possível decidir se A deve ser expandido (usando $A \rightarrow b$) ou se A deve ser desempilhado (usando $A \rightarrow \Lambda$).

Gramáticas LL(k)

b) Verificação se G é LL(2):

- 1) $\text{First}_2(aAaa) \cap \text{First}_2(bAba) = \emptyset$
- 2) Para $S \Rightarrow aAaa$ ($w = a; \alpha = aa$)
 $\text{First}_2(baa) \cap \text{First}_2(aa) = \emptyset$

Para $S \Rightarrow bAba$ ($w = b; \alpha = ba$)
 $\text{First}_2(bba) \cap \text{First}_2(ba) = \emptyset$

Logo: G é LL(2).

c) Verificação se G é fortemente LL(2):

$\text{Follow}_2(A) = \{ w \mid S \Rightarrow aAaa \text{ e } w \in \text{First}_2(aa) \text{ ou } S \Rightarrow bAba \text{ e } w \in \text{First}_2(ba) \} = \{ aa, ba \}$

Então: $\text{First}_2(\beta\text{Follow}_2(A)) \cap \text{First}_2(\gamma\text{Follow}_2(A))$ pode ser:

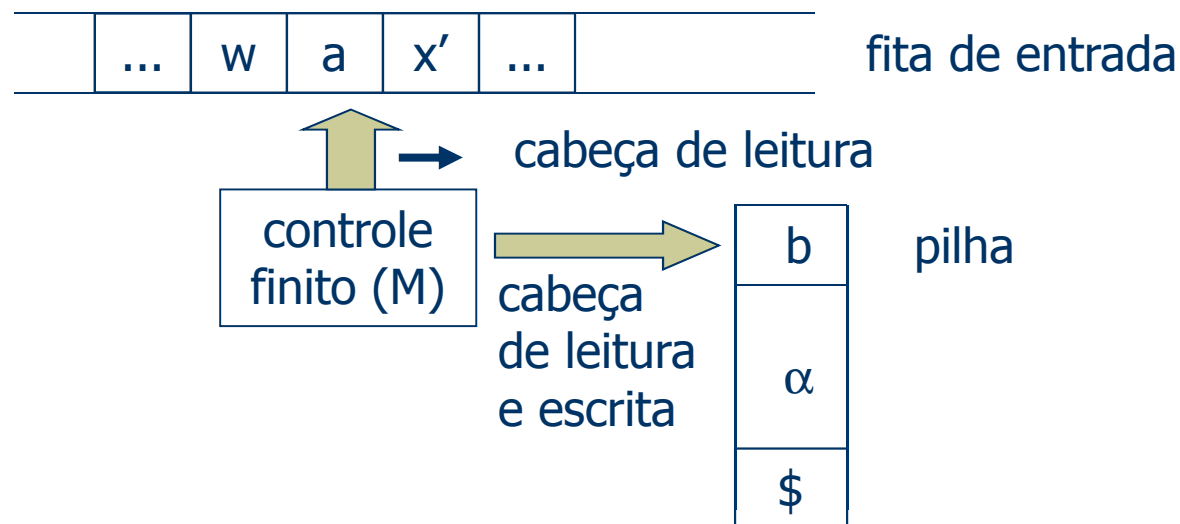
- 1) $\text{First}_2(baa) \cap \text{First}_2(aa) = \emptyset$
- 2) $\text{First}_2(bba) \cap \text{First}_2(aa) = \emptyset$
- 3) $\text{First}_2(bba) \cap \text{First}_2(ba) = \emptyset$
- 4) $\text{First}_2(baa) \cap \text{First}_2(ba) = \{ba\}$

Portanto: G não é fortemente LL(2).

$A \rightarrow b$
 $S \Rightarrow bAba \Rightarrow bbba$
 $A \rightarrow \Lambda$
 $S \Rightarrow bAba \Rightarrow bba$

Portanto, com **A** no topo da pilha e os símbolos **ba** como os dois próximos símbolos da entrada não é possível decidir se **A** deve ser **expandido** (para **b**) ou se **A** deve ser **desempilhado**.

Analísadores Sintáticos LL(1)



- Os movimentos do reconhecedor serão especificados por uma tabela de análise $M : (N \cup \Sigma \cup \{\$, \Lambda\}) \times (\Sigma \cup \{\Lambda\})$ definida por:
 - se $X \rightarrow \beta$ é a i -ésima produção em P , então:
 $M(X, a) = (\beta, i)$, para todo $a \in \text{First}_1(\beta)$, $a \neq \Lambda$.
Se $\Lambda \in \text{First}_1(\beta)$ então $M(X, b) = (\beta, i)$, para todo $b \in \text{Follow}_1(X)$.
 - $M(a, a) = \text{POP}$, para todo $a \in \Sigma$.
 - $M(\$, \Lambda) = \text{SUCESSO}$
 - $M(X, a) = \text{ERRO}$, para os demais casos.

Analísadores Sintáticos LL(1)

- Num movimento, o símbolo a ser lido (a) e o símbolo no topo da pilha (X) são determinados. Então a entrada $M(X, a)$ na tabela de análise é consultada para determinar a transição a ser feita. Seja $a = \text{First}_1(x)$.
 - se $M(X, a) = (\beta, i)$ então $(x, X\alpha) \mapsto (x, \beta\alpha)$
 - se $M(a, a) = \text{POP}$ então $(x, a\alpha) \mapsto (x', \alpha)$ com $x = ax'$
 - $M(\$, \Lambda) = \text{SUCESSO}$ indica que $(\Lambda, \$)$ é configuração final
 - se $M(X, a) = \text{ERRO}$ então $(x, X\alpha)$ é configuração terminal.
- Exemplo: Seja a gramática:
 - $E \rightarrow TE'$
 - $E' \rightarrow +TE'$
 - $E' \rightarrow \Lambda$
 - $T \rightarrow FT'$
 - $T' \rightarrow *FT'$
 - $T' \rightarrow \Lambda$
 - $F \rightarrow (E)$
 - $F \rightarrow a$

Para construir a tabela M , devemos determinar:

$\text{First}_1(TE') = \{ (, a \}$

$\text{First}_1(+TE') = \{ + \}$

Como $E' \rightarrow \Lambda \in P$, $\text{Follow}_1(E') = \{ \Lambda,) \}$

$\text{First}_1(FT') = \{ (, a \}$

$\text{First}_1(*FT') = \{ * \}$

Como $T' \rightarrow \Lambda \in P$, $\text{Follow}_1(T') = \{ \Lambda, +,) \}$

$\text{First}_1((E)) = \{ (\}$

$\text{First}_1(a) = \{ a \}$

Analísadores Sintáticos LL(1)

Tabela de análise:

	+	*	()	a	Λ
E			TE', 1		TE', 1	
E'	+TE', 2			Λ , 3		Λ , 3
T			FT', 4		FT', 4	
T'	Λ , 6	*FT', 5		Λ , 6		Λ , 6
F			(E), 7		a, 8	
+	POP					
*		POP				
(POP			
)				POP		
a					POP	
\$						SUC

Análise sintática (sequência de derivações mais à esquerda):

$1 \Rightarrow 4 \Rightarrow 8 \Rightarrow 6 \Rightarrow 2 \Rightarrow 4 \Rightarrow 8 \Rightarrow 5 \Rightarrow 8 \Rightarrow 6 \Rightarrow 3$

Seja a cadeia: $a+a*a$

$(a+a*a, E\$)$

$\mapsto (a+a*a, TE'\$)$ 1

$\mapsto (a+a*a, FT'E'\$)$ 4

$\mapsto (a+a*a, aT'E'\$)$ 8

$\mapsto (+a*a, T'E'\$)$

$\mapsto (+a*a, E'\$)$ 6

$\mapsto (+a*a, +TE'\$)$ 2

$\mapsto (a*a, TE'\$)$

$\mapsto (a*a, FT'E'\$)$ 4

$\mapsto (a*a, aT'E'\$)$ 8

$\mapsto (*a, T'E'\$)$

$\mapsto (*a, *FT'E'\$)$ 5

$\mapsto (a, FT'E'\$)$

$\mapsto (a, aT'E'\$)$ 8

$\mapsto (\Lambda, T'E'\$)$

$\mapsto (\Lambda, E'\$)$ 6

$\mapsto (\Lambda, \$)$ 3

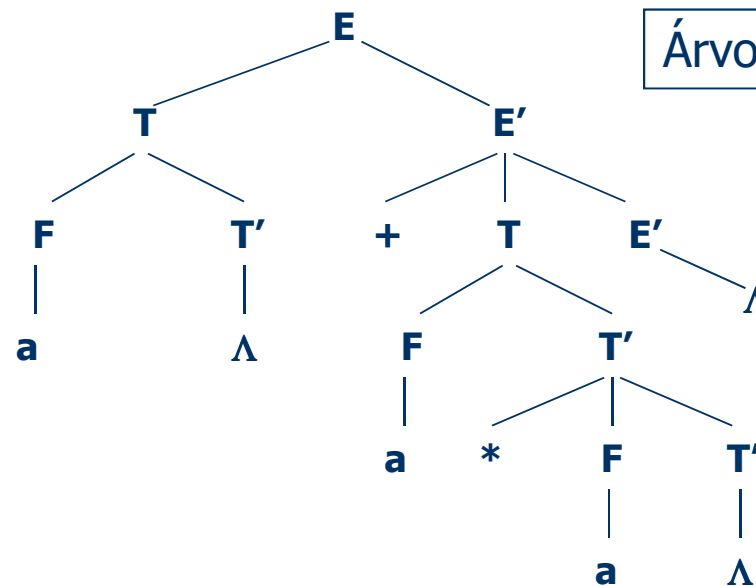
sucesso!

Analísadores Sintáticos LL(1)

1. $E \rightarrow TE'$
2. $E' \rightarrow +TE'$
3. $E' \rightarrow \Lambda$
4. $T \rightarrow FT'$
5. $T' \rightarrow *FT'$
6. $T' \rightarrow \Lambda$
7. $F \rightarrow (E)$
8. $F \rightarrow a$

Análise sintática (sequência de derivações mais à esquerda):

$1 \Rightarrow 4 \Rightarrow 8 \Rightarrow 6 \Rightarrow 2 \Rightarrow 4 \Rightarrow 8 \Rightarrow 5 \Rightarrow 8 \Rightarrow 6 \Rightarrow 3$



Árvore sintática de $a+a*a$

- ♦ Como vimos, uma maneira de reconhecer LLC é por meio de um analisador sintático descendente. Outra maneira é por meio da análise sintática ascendente ("bottom-up parsing"). Na análise sintática descendente tenta-se construir a árvore sintática da raiz (símbolo inicial da gramática) para as folhas (símbolos da cadeia de entrada).

Análise Sintática Ascendente

- ♦ Na análise sintática ascendente constrói-se a árvore das folhas para a raiz, da seguinte maneira:
 - a) considera-se a cadeia que está no topo da pilha e verifica-se se existe uma produção cujo lado direito corresponde a esses símbolos. Se existir, troca-se os símbolos do topo da pilha - que serão denominados leque ("handle") - pelo lado esquerdo da produção (essa operação será denominada redução).
 - b) se nenhuma redução é possível, empilha-se o próximo símbolo da cadeia de entrada, move-se a cabeça leitora para o próximo símbolo de entrada e o processo é reiterado.
 - c) ao se chegar ao fim da cadeia sem que nenhuma redução seja possível, retorna-se à configuração (isto é, posição da cabeça leitora e conteúdo da pilha) anterior à última redução e tenta-se outra alternativa.
- ♦ Esse método, conhecido como análise sintática ascendente com retorno ("bottom-up backtrack parsing") considera todos os possíveis movimentos de um autômato com pilha não determinístico para a gramática sendo, portanto, muito ineficiente. Além disso, Λ -produções causam dificuldade pois, pode-se fazer um número arbitrário de reduções nas quais Λ é "reduzida" a um não terminal.
- ♦ Para mais detalhes sobre esse algoritmo ver: AHO & ULLMAN - "The theory of parsing, translation and compiling", vol. I, p. 303-304.

Análise Sintática Ascendente

- Exemplo: Seja a gramática:
 - $S \rightarrow AB$
 - $A \rightarrow ab$
 - $B \rightarrow aba$

e a cadeia de entrada: ababa. A análise sintática ascendente será:

fita de entrada	pilha	operação	árvore
a ababa	\$	empilha	
a ababa	A	empilha	a
ab a ba	ab	reduz com 2	a b
ab a ba	A	empilha	<pre> A / \ a b </pre>
abab a	Aa	empilha	<pre> A / \ a b a </pre>
ababa a	Aab	reduz com 2	<pre> A / \ a b a b </pre>
ababa a	AA	empilha	<pre> A A / \ / \ a b a b </pre>
ababa _	AAa		<pre> A A / \ / \ a b a b a </pre>

Análise Sintática Ascendente

fita de entrada	pilha	operação	árvore
ababa_	AAa		

- Nesse ponto chegamos ao fim da cadeia de entrada e nenhuma redução é possível. Devemos voltar à configuração anterior à ultima redução:

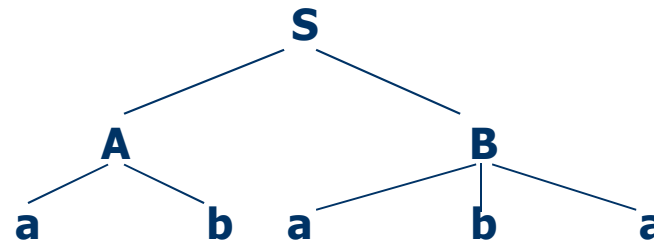
ababa	Aab	reduz com 2	
-------	-----	-------------	--

- Como não há alternativa para redução, devemos tentar o empilhamento:

ababa_	Aaba	reduz com 3	
ababa_	AB	reduz com 1	
ababa_	S		

Análise Sintática Ascendente

- Gramática:
 1. $S \rightarrow AB$
 2. $A \rightarrow ab$
 3. $B \rightarrow aba$



Reverso da sequência de produções usadas nas reduções: $1 \Rightarrow 3 \Rightarrow 2$

- Nesse caso, como chegamos ao fim da cadeia de entrada com o símbolo inicial da gramática no topo da pilha, a análise sintática termina com sucesso. O reverso da sequência de produções usadas nas reduções irá corresponder a uma derivação mais à direita da cadeia de entrada:
 $S \Rightarrow AB \Rightarrow Aaba \Rightarrow ababa$
- A seguir vamos mostrar uma classe de gramáticas livres de contexto - denominadas gramáticas LR(k) - para as quais é possível construir analisadores sintáticos ascendentes determinísticos e, portanto, eficientes.
- Existem duas decisões que um analisador ascendente determinístico deve tomar:
 - a) determinar, em cada movimento, se empilha um símbolo de entrada ou se faz uma redução (isto é equivalente a determinar a extremidade direita de um leque);

Análise Sintática Ascendente

- b) uma vez determinado o leque na pilha (isto é, determinado sua extremidade esquerda) determinar qual produção usar na redução.

Intuitivamente, uma GLC é LR(k) se dada uma derivação mais à direita:

$$S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_m = Z$$

onde $\alpha_{i-1} = \alpha A w$, $\alpha_i = \alpha \beta w$ (ou seja, β é o leque de α_i) e $\beta = X_1 X_2 \dots X_n$, então:

- a) conhecendo-se $\alpha X_1 \dots X_j$ e os primeiros k símbolos de $X_{j+1} \dots X_n w$, pode-se estar certo de que o extremo direito do leque só será alcançado quando $j = n$.
- b) conhecendo-se $\alpha \beta$ e, no máximo, os k primeiros símbolos de w, sempre será possível determinar que β é o leque e que β deve ser reduzido para A.

- ♦ Definição: Seja $G = (N, \Sigma, P, S)$ uma GLC. G é LR(k) ($k \geq 0$) se, sempre que:

1. $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$
2. $S \Rightarrow^* \gamma B x \Rightarrow \alpha \beta y$
3. $\text{First}_k(w) = \text{First}_k(y)$

então: $\alpha A y = \gamma B x$ (ou seja, $\alpha = \gamma$; $A = B$; $y = x$).

- ♦ Intuitivamente, essa definição diz que se $\alpha \beta w$ e $\alpha \beta y$ são formas sentenciais de G, se $\text{First}_k(w) = \text{First}_k(y)$, e se $A \rightarrow \beta$ foi a última produção usada para obter $\alpha \beta w$ numa derivação mais à direita, então $A \rightarrow \beta$ deve também ser usada para reduzir $\alpha \beta y$.

Analizador Sintático LR(k)

- ♦ Vamos, a seguir, discutir (informalmente) o funcionamento de um algoritmo de análise sintática para gramáticas LR(k). Os movimentos de um analisador LR(k) são especificados por tabelas que contêm todas as informações necessárias à análise, ou seja:
 - empilhar ou reduzir?
 - que produção usar numa redução?

Uma das tabelas é denominada tabela inicial e cada tabela LR(k) consiste de duas funções:

1. Função de análise:

$$f : \Sigma^k \rightarrow \{ \text{empilha, reduz com } i, \text{ erro, sucesso} \}$$

2. Função de movimento:

$$g : (N \cup \Sigma) \rightarrow (\mathbf{P} \cup \{ \text{erro} \}), \text{ onde } \mathbf{P} \text{ é o conjunto de todas as tabelas LR(k).}$$

A idéia da função de movimento g é evitar que a pilha precise ser analisada a cada passo para saber quando o leque aparece no topo (os valores de g são como estados de um autômato reconhecedor de leques; os estados do autômato serão empilhados também e o estado no topo da pilha corresponde ao estado que o autômato estaria se tivesse lido a pilha de baixo para cima).

Analizador Sintático LR(k)

Algoritmo de análise

- ♦ Seja (w, T_0) a configuração inicial do analisador, onde w é a cadeia de entrada (conteúdo da fita de entrada) e T_0 é a tabela LR(k) inicial (topo da pilha).
- 1) determinar a cadeia a ser lida u , constituída pelos próximos k símbolos de entrada.
- 2) determinar $f_T(u)$ para a tabela T do topo da pilha
 - a) se $f_T(u) = \text{"empilha"}$, então $(a\beta, \alpha) \mapsto (\beta, \alpha a g_T(a))$
 - b) se $f_T(u) = \text{"reduz com } i\text{"}$ e a i -ésima produção é $A \rightarrow \delta$, então:
 $(\gamma, \alpha X \beta) \mapsto (\gamma, \alpha X A g_X(A))$, onde $X \in \mathbf{P}$ e $|\beta| = 2|\delta|$ (ou seja, são removidos da pilha, o leque e todos os valores de g associados a ele)
 - c) se $f_T(u) = \text{"erro"}$, então terminar
 - d) se $f_T(u) = \text{"sucesso"}$, então o reverso da sequência de números de produções usadas nas reduções do passo (b) é a análise sintática ascendente de w (ou, equivalentemente, é a sequência de derivações mais à direita de w).

Analizador Sintático LR(k)

Exemplo: Seja a gramática LR(1):

1. $A \rightarrow AaAb$
2. $A \rightarrow \Lambda$

Sejam as tabelas:

	f			g		
	a	b	Λ	A	a	b
T_0	2		2	T_1		
T_1	E		S		T_2	
T_2	2	2		T_3		
T_3	E	E			T_4	T_5
T_4	2	2		T_6		
T_5	2		1			
T_6	E	E			T_4	T_7
T_7	1	1				

onde: E \equiv "empilha"; número i \equiv "reduz usando produção i"; S \equiv "sucesso" e as entradas em branco correspondem a configurações de erro.

Seja a cadeia de entrada $w = aabb$.

$(aabb, T_0) \mapsto (aabb, T_0AT_1)$

pois $f_{T_0}(a) = 2$; 2. $A \rightarrow \Lambda$; $g_{T_0}(A) = T_1$
 $\mapsto (abb, T_0AT_1aT_2)$

pois $f_{T_1}(a) = \text{"empilha"}; g_{T_1}(a) = T_2$
 $\mapsto (abb, T_0AT_1aT_2AT_3)$

$\mapsto (bb, T_0AT_1aT_2AT_3aT_4)$

$\mapsto (bb, T_0AT_1aT_2AT_3aT_4AT_6)$

$\mapsto (b, T_0AT_1aT_2AT_3aT_4AT_6bT_7)$

$\mapsto (b, T_0AT_1aT_2AT_3)$

$\mapsto (\Lambda, T_0AT_1aT_2AT_3bT_5)$

$\mapsto (\Lambda, T_0AT_1)$

$\mapsto \text{SUCESSO}$

Analizador Sintático LR(k)

- Neste caso, a sequência de produções usadas nas reduções é 22211. Logo, 11222 é a análise sintática ascendente de aabb. Como as produções são:

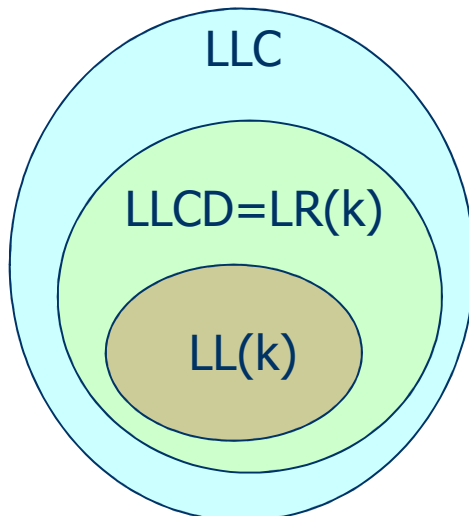
1. $A \rightarrow AaAb$

2. $A \rightarrow \Lambda$

a sequência de derivações mais à direita é:

$A \Rightarrow AaAb \Rightarrow AaAaAbb \Rightarrow AaAabb \Rightarrow Aaabb \Rightarrow aabb$

- Como mostramos anteriormente (slide 80), a classe das linguagens geradas por gramáticas LL(k) é um subconjunto próprio das linguagens livres de contexto determinísticas. A classe das linguagens geradas por gramáticas LR(k), no entanto, é a mesma classe das linguagens livres de contexto determinísticas, ou seja:



Para mais detalhes sobre gramáticas LR(k) e analisadores sintáticos ascendentes determinísticos, ver:

AHO & ULLMAN - "Principles of compiler design", p. 197-244.

AHO & ULLMAN - "The theory of parsing, translation and compiling", vol I, p. 368-396.