Aspectos Formais da Computação

Prof. Sergio D. Zorzo
Departamento de Computação – UFSCar
1º semestre / 2017
Aula 13

Reconhecedores de Cadeias para Linguagens Livre de Contexto

Algoritmos de reconhecimento das LLCs

Classificados em:

Top-Down ou Preditivo

● constroi uma árvore de derivação a partir da raiz (símbolo inicial da gramática) com ramos em direção às folhas (terminais)

Bottom-Up

● parte das folhas construindo a árvore de derivação em direção à raiz

Autômatos de pilha como Reconhecedor

- construção relativamente simples e imediata
- relação quase direta entre produções e as transições do AP
- algoritmo é:

– top-down

– simula derivação mais a esquerda

– não-determinismo são as produções alternativas da gramática

Autômatos de pilha como Reconhecedor

a partir de uma Gramática na Forma Normal de Greibach

– cada produção gera exatamente um terminal

– geração de w envolve |w| etapas de derivação

Cada variável pode ter diversas produções associadas

– AP testa as diversas alternativas

– número de passos para reconhecer w é proporcional a k |w|

– aproximação de k: metade da media de produções das variáveis.portanto, o AP construído

– tempo de reconhecimento proporcional ao expoente em |w|

– pode ser muito ineficiente para entradas mais longas

Autômato com Pilha Descendente

forma alternativa de construir AP, igualmente simples e com o mesmo nível de eficiência, a partir de uma GLC sem recursão a esquerda

simula a derivação mais a esquerda

- Algoritmo:

– inicialmente, empilha o símbolo inicial

– topo = variável: substitui, (não-determinismo), por todas as produções da variável

– topo = terminal: testa se e igual ao próximo simbolo da entrada

Autômato com Pilha Descendente

GLC $G = (V, T, P, S)$, sem recursão a esquerda

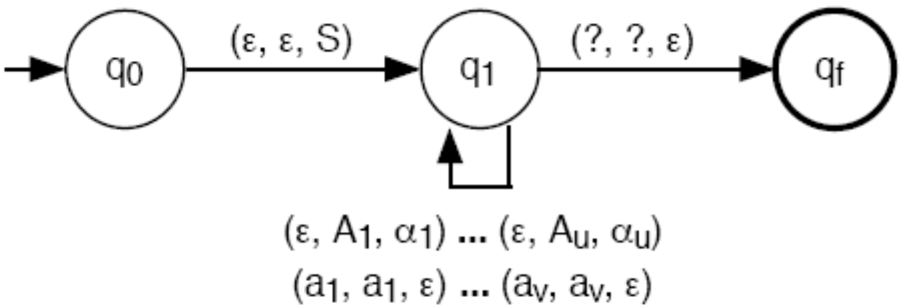
$M = (T, \{ q0, q1, qf \}, \delta, q0, \{ qf \}, V \cup T)$

$\delta(q0, \varepsilon, \varepsilon) = \{ (q1, S) \}$

$\delta(q1, \varepsilon, A) = \{ (q1, \alpha) \mid A \rightarrow \alpha \in P \}$ A de V

$\delta(q1, a, a) = \{ (q1, \varepsilon) \}$ a de T

$\delta(q1, ?, ?) = \{ (qf, \varepsilon) \}$



Exemplo: Autômato com Pilha Descendente

AP Descendente: Duplo Balanceamento

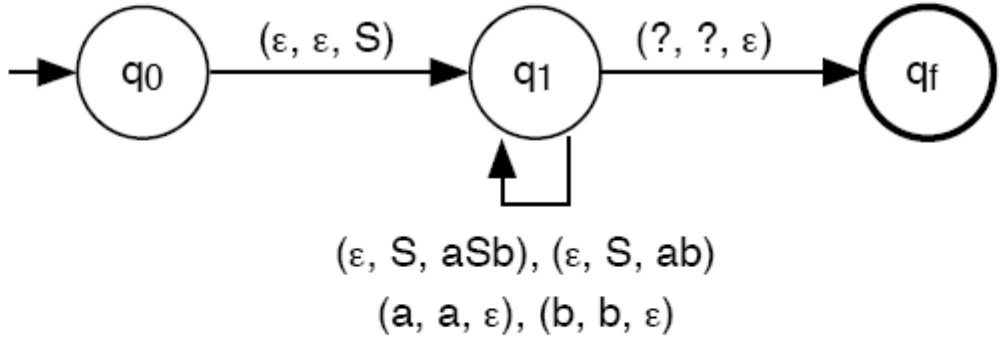
$L = \{ a^n b^n \mid n \geq 1 \}$

$G = (\{ S \}, \{ a, b \}, P, S)$ GLC sem recursão a esquerda

$P = \{ S \rightarrow aSb \mid ab \}$

- Automato com pilha descendente

$M = (\{ a, b \}, \{ q0, q1, qf \}, \delta, q0, \{ qf \}, \{ S, a, b \})$



Reconhecedores de LLCs -

Análise sintática descendente:

– **Com retrocesso (back track):** Quando a gramática permite, em um determinado estágio da derivação, a aplicação de mais de uma regra. Isto acontece quando o mesmo símbolo terminal aparece no inicio do lado direito de mais de uma regra de produção.

Exemplo:

A -> aα

A -> aβ

onde: α , $\beta \in (V_N \cup V_T)^*$ $A \in V_N$ $a \in V_T$

Análise Sintática Descendente

Analise sintática descendente:

- Sem retrocesso (preditive): Quando a gramática só permite um caminho a ser derivado. Desta forma, olhando apenas para o próximo símbolo de entrada podemos determinar a próxima derivação.

Requisitos:

- Durante o processo de derivação, sempre haverá uma única regra que possa levar a cadeia que esta sendo analisada.
- Não pode haver recursão a esquerda.

Ex : $A \rightarrow A\ \alpha$ onde $\alpha \in (V_N \cup V_T)^*$ e $A \in V_N$

Parser Preditivo Recursivo
ou Analisador Preditivo Recursivo

É executado um conjunto de procedimentos recursivos para processar a entrada. A cada não terminal é associado um procedimento;

- Existe também um procedimento adicional (Analisador Léxico) para o reconhecimento dos simbolos (tokens) ;

Vantagens:

- Simplicidade;

Desvantages:

- Maior tempo de processamento;
- Não é geral, pois existem linguagens que não aceitam recursividade.

Parser Preditivo Recursivo ou Analisador Preditivo Recursivo

- First(A) (ou Primeiro(A)) é o conjunto de tokens (símbolos) que figuram como primeiro elemento de uma ou mais cadeias geradas a partir de A.

Ex: S -> AS | BA

A -> aB | C

B -> bA | d

C -> c

First(S) = First(A) U First(B) = {a, c, b, d}

First(A) = {a, c}

First(B) = {b, d}

First(C) = {c}

Parser Preditivo Recursivo
ou Analisador Preditivo Recursivo

Só se pode aplicar o analisador preditivo recursivo em uma gramática se para todas as regras do tipo $A \rightarrow \alpha$, $A \rightarrow \beta$, ...(onde α , $\beta \in (V_N \cup V_T)^*$)os conjuntos First(α) e First(β) forem disjuntos

Algoritmo de Early

possivelmente o mais rápido algoritmo para LLC em geral

tempo de processamento proporcional a:

– em geral: $|w|^3$

– gramáticas nao-ambiguas: $|w|^2$

– muitas gramáticas de interesse prático: $|w|$

Algoritmo top-down

– a partir de uma GLC sem produções vazias

– parte do simbolo inicial

– executa sempre a derivação mais a esquerda

– cada ciclo gera um terminal

● compara com o símbolo da entrada

● sucesso -> construção do conjunto de produções que, potencialmente, pode gerar o próximo símbolo

Algoritmo de Early

Seja G = (V, T, P, S) uma GLC sem produções vazias

– $w = a_1a_2 \ldots a_n$ palavra a ser verificada

● marcador "•"

– antecedendo a posição, em cada produção, que será analisada na tentativa de gerar o próximo simbolo terminal

● sufixo "/u" adicionado a cada produção

– indica o u-ésimo ciclo em que passou a ser considerada

Algoritmo de Early

Etapa 1: construção de D_0: primeiro conjunto de produções

(1)produções que partem de S

(2)produções que podem ser aplicadas em sucessivas derivações mais a esquerda (a partir de S)

- D0 = $\varnothing$
- para toda $S \rightarrow \alpha \in P$ (1)
- faca $D_0 = D_0 \cup \{ S \rightarrow \bullet\alpha/0 \}$
- repita para toda $A \rightarrow \bullet B\beta/0 \in D_0$ (2)

– faca para toda $B \rightarrow \varphi \in P$

- faca $D_0 = D_0 \cup \{ B \rightarrow \bullet\varphi/0 \}$

até que D_0 nao aumente

Algoritmo de Early

Etapa 2: construção dos demais conjuntos de produção

– • $n = |w|$ conjuntos de produção a partir de D_0

– • ao gerar a_r, constroi D_r: produções que podem gerar a_{r+1}

● para r variando de 1 ate n (1)

● faça $D_r = \varnothing$;

para toda $A \rightarrow \alpha \bullet a_r \beta / s \in D_{r-1}$ (2)

– faça $D_r = D_r \cup \{ A \rightarrow \alpha a_r \bullet \beta / s \}$;

– repita para toda $A \rightarrow \alpha \bullet B \beta / s \in D_r$ (3)

● faça para toda $B \rightarrow \varphi \in P$ faça $D_r = D_r \cup \{ B \rightarrow \bullet \varphi / r \}$

● para toda $A \rightarrow \alpha \bullet / s$ de D_r (4)

● faça para toda $B \rightarrow \beta \bullet A \varphi / k \in D_s$ faça $D_r = D_r \cup \{ B \rightarrow \beta A \bullet \varphi / k \}$

– até que D_r não aumente

Algoritmo de Early

(1) cada ciclo gera um conjunto de produções D_r

(2) gera o simbolo a_r

(3) produções que podem derivar o próximo simbolo

(4) uma subpalavra de w foi reduzida a variável A

– inclui em D_r todas as produções de D_s que referenciaram •A;

Algoritmo de Early

Etapa 3: condição de aceitação da entrada.

– uma produção da forma $S \rightarrow \alpha \bullet /0$ pertence a D_n

w foi aceita se

$S \rightarrow \alpha \bullet /0$ e uma produção que

- parte do simbolo inicial S
- foi incluída em D_0 ("/0")
- todo o lado direito da produção foi analisado com sucesso ("•" esta no final de α)

Otimização do Algoritmo de Early

– ciclos repita-ate podem ser restritos exclusivamente as produções recentemente incluidas em D_r ou em D_0 ainda não-analisadas.

Algoritmo de Early

"Expressao simples" da linguagem Pascal

$G = (\{ E, T, F \}, \{ +, *, [,], x \}, P, E)$, na qual:

$P = \{ E \to T \mid E+T,\ T \to F \mid T*F,\ F \to [E] \mid x \}$

- Reconhecimento da palavra $x*x$
- D_0:

– $E \to \bullet T/0$	produções que partem
– $E \to \bullet E+T/0$	do símbolo inicial
– $T \to \bullet F/0$	produções que podem ser aplicadas
– $T \to \bullet T*F/0$	em derivação mais a esquerda
– $F \to \bullet [E]/0$	a partir do símbolo inicial
– $F \to \bullet x/0$	

Algoritmo de Early

D_1: reconhecimento de x em x∗x

– $F \rightarrow x\bullet/0$ x	foi reduzido a F
– $T \rightarrow F\bullet/0$	inclui todas as produções de D_0 que
– $T \rightarrow T\bullet * F/0$	referenciaram •F direta ou indiretamente
– $E \rightarrow T\bullet/0$	movendo o marcador "•"
– $E \rightarrow E\bullet+T/0$	um simbolo para a direita

D_2: reconhecimento de ∗ em x∗x

– $T \rightarrow T*\bullet F/0$	gerou ∗; o proximo sera gerado por F
– $F \rightarrow \bullet[E]/2$	inclui todas as producoes de P que
– $F \rightarrow \bullet x/2$	podem gerar o prox terminal a partir de F•

Algoritmo de Early

D_3: reconhecimento de x em $x*x$

- $F \rightarrow x\bullet/2$ x foi reduzido a F
- $T \rightarrow T*F\bullet/0$ incluido de D_2 (pois $F \rightarrow x\bullet/2$); entrada reduzida a T
- $E \rightarrow T\bullet/0$ incluido de D_0 (pois $T \rightarrow T*F\bullet/0$); entrada reduzida a E
- $T \rightarrow T\bullet*F/0$ incluido de D_0 (pois $T \rightarrow T*F\bullet/0$)
- $E \rightarrow E\bullet+T/0$ incluido de D_0 (pois $E \rightarrow T\bullet/0$)

● Como $w = x*x$ foi reduzida a E e como $E \rightarrow T\bullet/0$ pertence a D_3

Então a entrada é aceita

Algoritmo de Cocke-Younger-Kasami

a partir de uma GLC na Forma Normal de Chomsky

– gera bottom-up todas as arvores de derivacao da entrada w

– tempo de processamento proporcional a $|w|^3$

- idéia basica

– tabela triangular de derivação

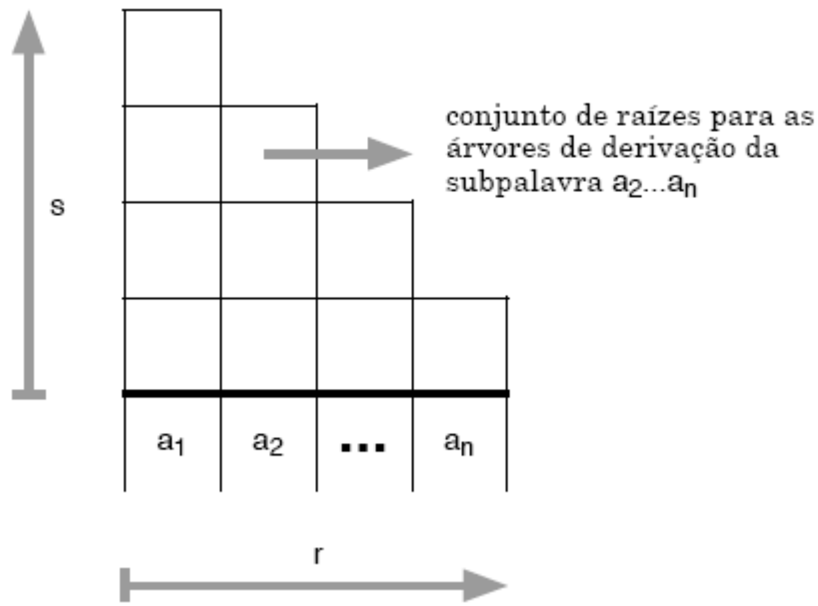
– célula: raizes que podem gerar a correspondente sub-arvore

Algoritmo de Cocke-Younger-Kasami

Seja G = (V, T, P, S) uma GLC na Forma Normal de Chomsky

$w = a_1a_2\ldots a_n$ uma entrada entrada

V_{rs} células da tabela



Algoritmo de Cocke-Younger-Kasami

Etapa 1: variáveis que geram diretamente terminais ($A \rightarrow a$)

– para r variando de 1 ate n faca $V_{r1} = \{ A \mid A \rightarrow a_r \in P \}$

Etapa 2: produções que geram duas variaveis ($A \rightarrow BC$)

– para s variando de 2 ate n

– faça para r variando de 1 ate n - s + 1

faça $V_{rs} = \varnothing$

– para k variando de 1 ate s - 1

– faça $V_{rs} = V_{rs} \cup \{ A \mid A \rightarrow BC \in P,\ B \in V_{rk} \text{ e } C \in V_{(r+k)(s-k)} \}$

- limite de iteração para r e (n - s + 1): a tabela é triangular
- V_{rk} e $V_{(r+k)(s-k)}$ são as raizes das sub-árvores de V_{rs}
- celula vazia: não gera qualquer sub-árvore

Etapa 3: condição de aceitação da entrada.

símbolo inicial pertence a V_{1n} (raiz de toda palavra)

Algoritmo de Cocke-Younger-Kasami

G = ({ S, A }, { a, b }, P, S)

P = { $S \rightarrow AA \mid AS \mid b$, $A \rightarrow SA \mid AS \mid a$ }

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

Algoritmo de Cocke-Younger-Kasami

G = ({ S, A }, { a, b }, P, S)

P = { S $\rightarrow$ AA | AS | b, A $\rightarrow$ SA | AS | a }

1. $s = 2$, $r = 1$, $k = 1$: $A \rightarrow BC$, $B \in V(1,1)$ e $C \in V(2,1)$. $V(1,2) = \emptyset \bigcup \{S, A\} = \{S, A\}$;
2. $s = 2$, $r = 2$, $k = 1$: $B \in V(2,1)$ e $C \in V(3,1)$. $V(2,2) = \emptyset \bigcup \{A\} = \{A\}$;
3. $s = 2$, $r = 3$, $k = 1$: $B \in V(3,1)$ e $C \in V(4,1)$. $V(3,2) = \emptyset \bigcup \{S\} = \{S\}$;
4. $s = 2$, $r = 4$, $k = 1$: $B \in V(4,1)$ e $C \in V(5,1)$. $V(4,2) = \emptyset \bigcup \{S, A\} = \{S, A\}$;
5. $s = 3$, $r = 1$, $k = 1$: $B \in V(1,1)$ e $C \in V(2,2)$. $V(1,3) = \emptyset \bigcup \{S\} = \{S\}$;

Algoritmo de Cocke-Younger-Kasami

G = ({ S, A }, { a, b }, P, S)

P = { S → AA | AS | b, A → SA | AS | a }

6. $s = 3$, $r = 1$, $k = 2$: $B \in V(1,2)$ e $C \in V(3,1)$.
   $V(1,3) = \{S\} \bigcup \{S, A\} = \{S, A\}$;
7. $s = 3$, $r = 2$, $k = 1$: $B \in V(2,1)$ e $C \in V(3,2)$.
   $V(2,3) = \emptyset \bigcup \emptyset = \emptyset$;
8. $s = 3$, $r = 2$, $k = 2$: $B \in V(2,2)$ e $C \in V(4,1)$.
   $V(2,3) = \emptyset \bigcup \{S\} = \{S\}$;
9. $s = 3$, $r = 3$, $k = 1$: $B \in V(3,1)$ e $C \in V(4,2)$.
   $V(3,3) = \emptyset \bigcup \{S, A\} = \{S, A\}$;
10. $s = 3$, $r = 3$, $k = 2$: não precisa.
11. $s = 4$, $r = 1$, $k = 1$: $B \in V(1,1)$ e $C \in V(2,3)$.
   $V(1,4) = \emptyset \bigcup \{S, A\} = \{S, A\}$;

Algoritmo de Cocke-Younger-Kasami

G = ({ S, A }, { a, b }, P, S)

P = { S $\rightarrow$ AA | AS | b, A $\rightarrow$ SA | AS | a }

12. $s = 4$, $r = 1$, $k = 2, 3$: não precisa;
13. $s = 4$, $r = 2$, $k = 1$: $B \in V(1, 1)$ e $C \in V(2, 3)$.
    $V(2, 4) = \emptyset \bigcup \{S, A\} = \{S, A\}$;
14. $s = 4$, $r = 2$, $k = 2, 3$: não precisa;
15. $s = 5$, $r = 1$, $k = 1$: $B \in V(1, 1)$ e $C \in V(2, 4)$.
    $V(1, 5) = \emptyset \bigcup \{S, A\} = \{S, A\}$;
16. $s = 5$, $r = 1$, $k = 2, 3, 4$: não precisa.

Implementando autômatos de pilha

Analisadores LL(k)

- LL = <u>**L**</u>eft-to-right, <u>**L**</u>eftmost derivation
- Análise *top-down* – derivações mais à esquerda
- Problema principal
  - Qual produção usar em cada substituição?
  - Ex:

    stmt → if (expr) then stmt;

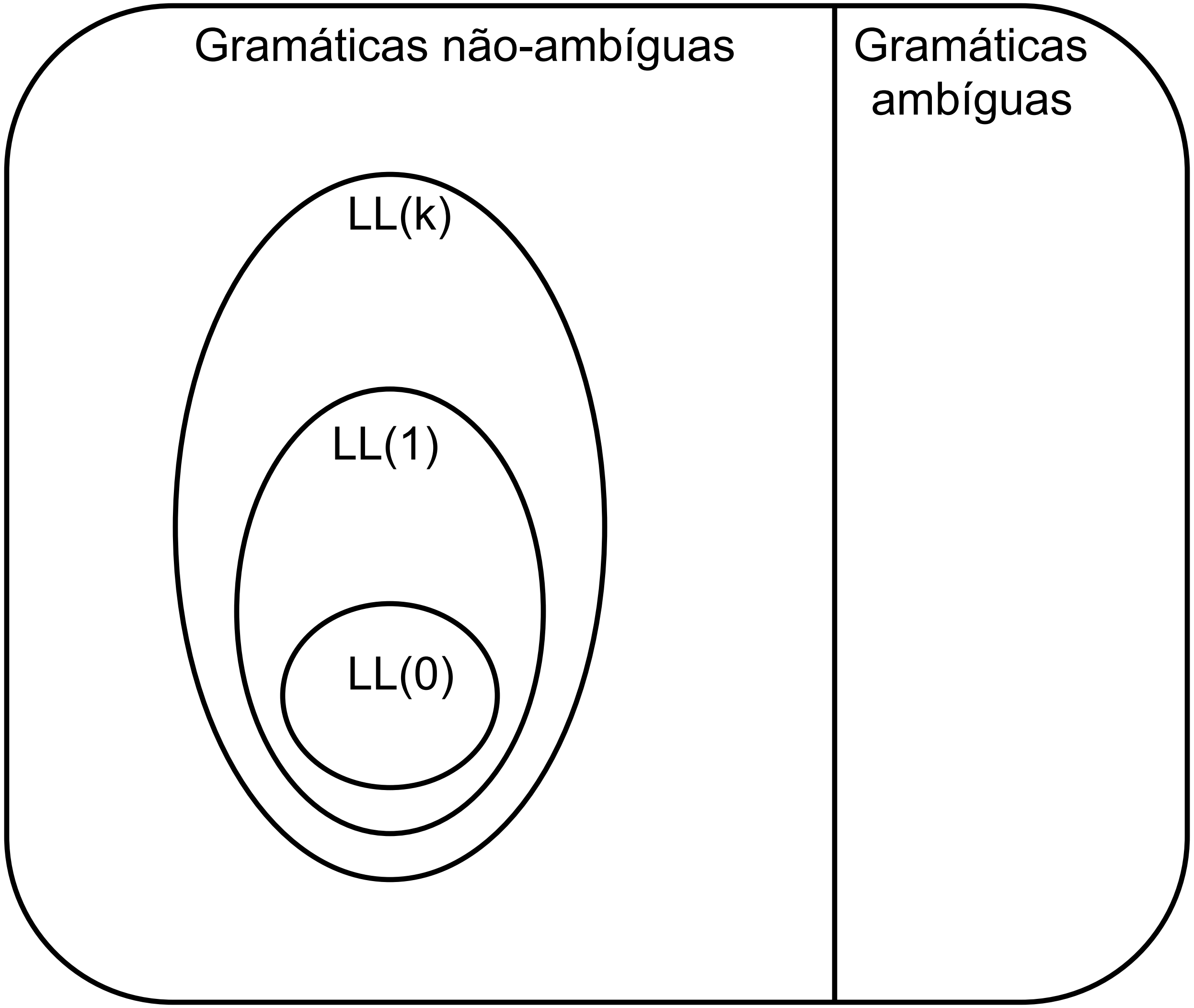
    stmt → if (expr) then stmt else stmt;

    stmt → ...
  - Característica principal das gramáticas LL(k): é possível determinar qual produção escolher olhando-se k símbolos à frente
    - LL(0) não olha nenhum símbolo à frente
    - LL(1) olha um símbolo à frente
    - ...

Analisadores LL(k)

- Reconhecem as linguagens denotadas pelas gramáticas LL(k)
- Limitações principais dessa classe de gramáticas:
  - Não é possível ter recursividade à esquerda
    - Ex: $A \rightarrow A\ b\ c$
    - Como sempre é feita derivação mais à esquerda, ocorre uma recursão infinita
    - É necessário remover essa recursividade (para mais detalhes, aguarde a disciplina de compiladores)
  - Regras com um prefixo em comum exigem valor muito grande de k
    - Ex: $A \rightarrow b\ c\ d\ e \mid b\ c\ d\ f$
      - Neste exemplo, é necessário $k = 4$ para decidir qual corpo utilizar na substituição
    - Pode-se fatorar a regra para reduzir k:
      - $A \rightarrow b\ c\ d\ X$

Relações entre as classes das gramáticas livres de contexto



Analisadores LL(k)

- Vantagens:
  - É mais natural / intuitivo de implementar
  - Fácil recuperação de erros
  - Facilita a análise semântica (significado)
- Desvantagens
  - Ineficiente para $k > 1$
  - Gramática fica mais difícil → LL(1)
  - Fatoração e recursividade à esquerda
- Exemplo
  - JavaCC (Desenvolvido inicialmente pela Sun)

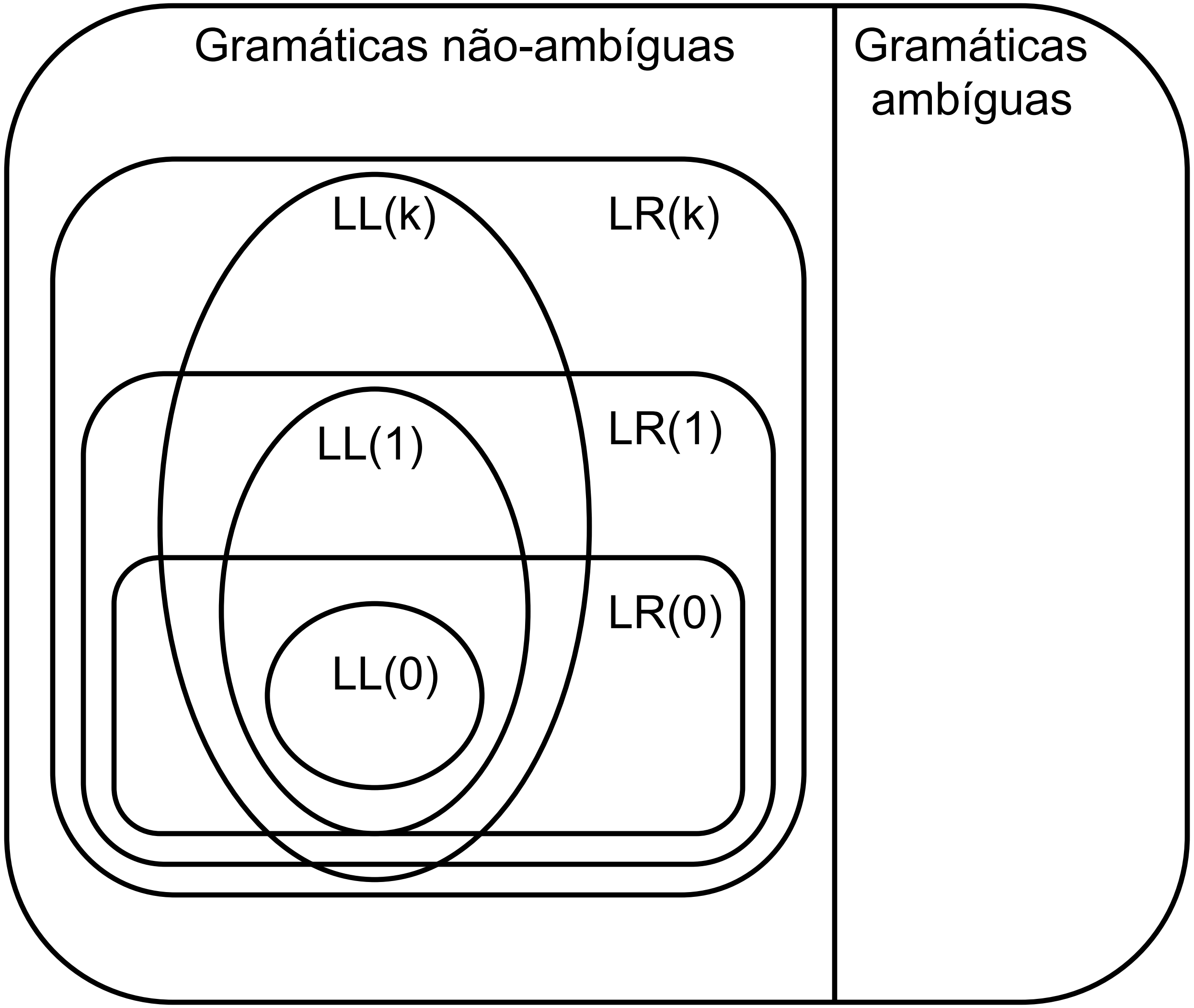
Analisadores LR(k)

- LR – **<u>L</u>**eft-to-right, **<u>R</u>**ightmost derivation
- Análise *bottom-up* – inferência recursiva
- Modelo empilhar-reduzir
  - Símbolos lidos são armazenados em uma pilha
  - Eventualmente, os símbolos no topo da pilha são "reduzidos"
    - Ex Java: "package" "teste" ";" → package
- A cada etapa, o analisador precisa decidir:
  - Empilhar
  - Reduzir
- Olhando k símbolos à frente para tomar essa decisão

Analisadores LR(k)

- Reconhecem as linguagens denotadas pelas gramáticas LR(k)
- Prinicipais características
  - Toleram recursividade à esquerda e alguns tipos de ambiguidade
  - Recursividade à direita é suportada, mas deve ser evitada, pois causa ineficiência
  - Possibilitam resolução de conflitos através de definição de precedência e associatividade de terminais

Relações entre as classes das gramáticas livres de contexto



Analisadores LR(k)

- Vantagens:
  - Facilita a geração automática
  - Implementação eficiente
  - Gramáticas mais abrangentes e mais fáceis de construir do que as LL(k)
- Desvantagens:
  - Conflitos podem ser difíceis de resolver
  - Código de implementação não se assemelha à gramática (em contraste com LL)
  - Dificulta a depuração
- Exemplos:
  - YACC (AT&T)
  - Bison (GNU)
  - CUP (GATech, TUM)

Outros tipos de implementações de DPDAs

- LALR(1)
  - Utiliza um símbolo à frente adicional (LookAhead LR) para reduzir o consumo de espaço e desempenho do método LR
- LL(*)
  - Semelhante ao LL(k), mas com k variável (k=*)
    - Nesta técnica, o k pode ser modificado em tempo de execução, para eliminar a necessidade de fatoração de regras

Hierarquia de Chomsky

Hierarquia de Chomsky

- Descreve as classes de linguagens formais

Hierarquia	Gramáticas	Linguagens	Autômato mínimo
Tipo-0	?	?	?
Tipo-1	?	?	?
Tipo-2	Livres de contexto	Livres de contexto	Autômatos de pilha
Tipo-3	Regulares (Expressões regulares)	Regulares	Autômatos finitos

Hierarquia de Chomsky

- Podemos detalhar o tipo-2

Hierarquia	Gramáticas	Linguagens	Autômato mínimo
Tipo-0	?	?	?
Tipo-1	?	?	?
Tipo-2	Livres de contexto	Livres de contexto	Autômatos de pilha não-determinísticos (NPDA)
	Livres de contexto determinísticas	Livres de contexto determinísticas	Autômatos de pilha determinísticos (DPDA)
Tipo-3	Regulares (Expressões regulares)	Regulares	Autômatos finitos (NFA, DFA, ε-NFA

Hierarquia de Chomsky

- Podemos detalhar ainda mais a classe das gramáticas/linguagens determinísticas

<table>
<tr><th>Hierarquia</th><th colspan="4">Gramáticas</th><th colspan="4">Linguagens</th><th colspan="4">Autômato mínimo</th></tr>
<tr><td>Tipo-0</td><td colspan="4">?</td><td colspan="4">?</td><td colspan="4">?</td></tr>
<tr><td>Tipo-1</td><td colspan="4">?</td><td colspan="4">?</td><td colspan="4">?</td></tr>
<tr><td>Tipo-2</td><td colspan="4">Livres de contexto</td><td colspan="4">Livres de contexto</td><td colspan="4">Autômatos de pilha não-determinísticos (NPDA)</td></tr>
<tr><td></td><td>LL</td><td>LR</td><td>LALR</td><td>...</td><td>LL</td><td>LR</td><td>LALR</td><td>...</td><td>LL</td><td>LR</td><td>LALR</td><td>...</td></tr>
<tr><td>Tipo-3</td><td colspan="4">Regulares
(Expressões regulares)</td><td colspan="4">Regulares</td><td colspan="4">Autômatos finitos (NFA, DFA, ε-NFA</td></tr>
</table>

Fim