

# Teoria da Computação

## Aula - Linguagens e Gramáticas Formais

Simão Melo de Sousa

# Introduction a Teoria das Linguagens Formais

---

prelúdio

- A redacção dos apontamentos da disciplina documento baseou-se fortemente na bibliografia indicada. Parece-nos então óbvio que a leitura e a aprendizagem directa pelas obras originais é recomendada, e mesmo essencial á compreensão profunda das noções aqui apresentadas;
- O português não é a língua materna do autor e o presente documento encontra-se em fase (constante) de elaboração/melhoramento pelo que se agradece e até se incentiva qualquer sugestão ou correcção;

- (Principal) C. H. Papadimitriou, H. R. Lewis. *emph*Elements of the Theory of Computation por Prentice Hall, 1997. Tradução brasileira: **Elementos de Teoria da Computação**, 2a Edição. Bookman, Porto Alegre, 2000.
- (introdutório e de leitura agradável) P. Linz. **An introduction to formal languages and automata**. Jones and Bartlett Publisher, 2006.
- (Uma obra de referência e muito completa... um “must”) John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. **Introduction to Automata Theory, Languages, and Computation** (3rd Edition). Addison Wesley, 2006 (existe em português do Brasil).
- (Abordagem abrangente) M. Sipser. **Introducton to the Theory of Computation**. PWS Publishing, 2006.
- (Introdutório, em francês - embora deva existir algo em inglês algures) P. Wolper. **Introduction à la calculabilité**, 3ª edição, Dunod, 2006.

# Introduction a Teoria das Linguagens Formais

---

## Contexto

# O que é a teoria das linguagens formais?

- A tentativa de **modelar as linguagens** (originalmente as linguagens naturais)
- $\implies$  esta teoria sofreu uma evolução considerável quando se notou a sua particular adequação à descrição de **processos computacionais** e **linguagens de programação**.

- Na utilização duma linguagem  $\mathcal{L}$  num processo de comunicação existem sempre 2 intervenientes:
  1. **O Locutor**: o emissor da mensagem. Este deve dispor de meios adequados de construção de mensagem.
  2. **O Auditor**: o receptor da mensagem. Este deve dispor de meios adequados para reconhecer e perceber a mensagem escrita na linguagem  $\mathcal{L}$ .

Conceitos, propriedades, técnicas e ferramentas para

- descrever e caracterizar linguagens (**formais** - de forma formal)
- gerar palavras de determinadas linguagens (**gramáticas**)
- reconhecer e perceber palavras de determinadas linguagens (**autómatos, expressões regulares, etc...**)

Estes são os objectivos clássicos quando estudados como fundamentos para os processos de compilação (ver segundo semestre)



Os processos geradores e reconhecedores de linguagens têm uma ligação muito forte com a teoria da computação porque

- são processos computacionais fundamentais
- permitam definir e enquadrar adequadamente modelos computacionais e os seus limites.

É essa perspectiva que iremos estudar nesta disciplina.

# Introduction a Teoria das Linguagens Formais

---

## Hierarquia de Chomsky

- Nem todas as linguagens tem o mesmo poder expressivo ou complexidade.
- De facto Chomsky e Schutzenberger desenvolveram uma classificação de linguagens
- Numa primeira abordagem vamos apresentar uma generalização **informal** desta classificação. Esta classificação é uma hierarquia linear. Se  $A$  esta por baixo de  $B$  isto significa que  $A \subseteq B$ .

- **Linguagem:** Conjunto finito de palavras.
- **Gramática associada:** lista de palavras.
- **Mecanismo de reconhecimento:** Uma simples máquina de comparação de texto (i.e. Linguagem = BD, pertence = query à BD).

- **Linguagem:** conjunto (não necessariamente finito) de palavras cuja correcção sintáctica só necessita de memória em quantidade finita numa leitura da esquerda para a direita. Outra caracterização: a estrutura destas palavras segue um padrão (i.e. são de estrutura regular)
- **Gramática associada:** gramáticas regulares, em particular lineares a esquerda ou direita.
- **Mecanismo de reconhecimento:** Autómatos de estados finitos.

- **Linguagem:** conjunto de palavras cujos padrões estruturais podem ser aninhados ou sequencializados. São linguagens que se assemelham em termos de estrutura à **linguagem de Dyck**.
- **Gramática associada:** gramáticas algébricas
- **Mecanismo de reconhecimento:** autómatos com pilha

# Linguagens Contextuais - Dependentes do Contexto

- **Linguagem:** Linguagens de descrição informal difícil. Palavras destas linguagens podem conter padrões que podem depender de outros padrões que já ocorreram.
- **Gramática associada:** gramáticas lineares limitadas (**bounded linear grammars**).
- **Mecanismo de reconhecimento:** Autómatos lineares limitados.

- **Linguagem:** Conjunto de palavras para o qual existe um algoritmo de reconhecimento
- **Gramática associada:** N/A.
- **Mecanismo de reconhecimento:** os métodos de reconhecimento saem da classe dos autómatos para a classe mais vasta dos algoritmos ( $\lambda$ -termos, máquinas de Turing totais, etc...)



- **Linguagem:** Conjunto de palavras para o qual existe um método computacional parcial, no sentido que este sabe determinar a propriedade “pertence”, mas não o “não pertence”.
- **Gramática associada:** N/A.
- **Mecanismo de reconhecimento:** Máquinas de Turing Parciais. se  $m \in \mathcal{L}$ , a máquina diz “sim” em tempo finito, mas se  $m \notin \mathcal{L}$  a máquina entra em ciclo.

- **Linguagem:** Conjunto de palavras formadas sem quaisquer restrições
- **Gramática associada:** N/A
- **Mecanismo de reconhecimento:** Sai do enquadramento do que um computador sabe fazer.

- Na hierarquia clássica de Chomsky só se considera 4 níveis. Por exemplo as duas primeiras classes desta hierarquia formam a classe 3 na hierarquia clássica de Chomsky. Voltaremos a este assunto mais adiante.
- As três primeiras classes são particularmente importantes para o processamento de linguagens e a concepção de linguagens de programação.
- As restantes classes estão ligadas a noções importante em Teoria da Computação.

# Conceitos Preliminares

---

- Seja  $A$  um conjunto (finito) que designaremos por **alfabeto**.
- Tendo em conta o alfabeto  $A$ , uma palavra é uma sequência eventualmente vazia de elementos (letras) de  $A$ .
- Designamos por  $A^*$  o conjunto de todas as palavras constituídas de letras de  $A$ .
- Uma palavra  $a = a_1 a_2 a_3 \dots a_n$  tem por comprimento  $n$ . Notação:  $|a| = n$  ou  $\#(a) = n$ .
- A palavra vazia, notada  $\epsilon$ , é a palavra de comprimento 0.
- Seja  $\cdot$  a operação de concatenação,  $(A^*, \cdot)$  é designado de monoíde livre gerado por  $A$ .
  - $\cdot$  é associativa
  - $\epsilon$  é o elemento neutro de  $\cdot$ .

- Seja  $A$  um alfabeto.
- Uma linguagem não é nada mais do que um conjunto de palavras, ou seja é um subconjunto de  $A^*$
- Nesta perspectiva: Sejam  $\mathcal{L}$  e  $\mathcal{L}'$  duas linguagens
  - $\mathcal{L} \cup \mathcal{L}'$ ,  $\mathcal{L} \cap \mathcal{L}'$ , etc...
  - $\emptyset$  (repare que  $\emptyset \neq \epsilon$ )
  - $\bar{\mathcal{L}} \triangleq \{u \in A^* \mid u \notin \mathcal{L}\}$
  - $\mathcal{L}\mathcal{L}' \triangleq \{u.v \mid u \in \mathcal{L} \wedge v \in \mathcal{L}'\}$
  - $\mathcal{L}^* \triangleq \{\epsilon\} \cup \mathcal{L} \cup \mathcal{L}\mathcal{L} \cup \mathcal{L}\mathcal{L}\mathcal{L} \dots \triangleq \bigcup_{n \in \mathbb{N}} \mathcal{L}^n$
  - $\mathcal{L}^+ \triangleq \mathcal{L} \cup \mathcal{L}\mathcal{L} \cup \mathcal{L}\mathcal{L}\mathcal{L} \dots \triangleq \bigcup_{n \in \mathbb{N}^*} \mathcal{L}^n$
  - $\mathcal{L}^{<n} \triangleq \{\epsilon\} \cup \mathcal{L} \cup \mathcal{L}\mathcal{L} \cup \mathcal{L}\mathcal{L}\mathcal{L} \dots \underbrace{\mathcal{L} \dots \mathcal{L}}_{n-1}$
  - $\tilde{\mathcal{L}} \triangleq \{a_n a_{n-1} \dots a_2 a_1 \mid a_1 a_2 \dots a_{n-1} a_n \in \mathcal{L}\}$  (linguagem espelho)são igualmente linguagens.

# Gramáticas de Chomsky

---

- Gramática de Chomsky: sistema de produção de palavras, baseado em dois alfabetos  $N$ ,  $\Sigma$  e um conjunto de regras de reescrita (designadas de **produção** ou de **regras**)  $P$ .
- $\Sigma$ : Alfabeto sobre o qual a linguagem  $\mathcal{L}$  é definida. **Conjunto dos símbolos terminais**.
- $N$ : Conjunto de símbolos, distinto de  $\Sigma$ , designados por **símbolos não terminais**
- $P$ : Conjunto finito de **regras de produção** que descrevem como são produzidas as palavras da linguagem  $\mathcal{L}$  a partir dos elementos de  $N$  e de  $\Sigma$ .

## Definition (Gramáticas de Chomsky)

Uma gramática de Chomsky  $G$  é definida pelo tuplo  $(\Sigma, N, P, S)$  onde

- $\Sigma$ ,  $N$  conjuntos finitos distintos ( $\Sigma \cap N = \emptyset$ )
- $S \in N$ , é o **símbolo inicial** (pelo qual se inicia o processo de geração)
- $P$  conjunto finito e  $P \subseteq ((N \cup \Sigma)^* - \Sigma^*) \times (N \cup \Sigma)^*$



- $P$  é finito e é constituído por pares  $(\alpha, \beta)$ . Notação  $\alpha \rightarrow \beta$ .
- Lê-se de  $\alpha$  **produz-se**  $\beta$
- $S$  é um elemento especial de  $N$ . Designa o ponto de partida da gramática e do processo de produção. Por isso é designado de **axioma** ou **símbolo inicial**.

- $G = (\{0, 1\}, \{A, S\}, P, S)$  onde  $P = \{ S \rightarrow 0A1, \quad \}$   
 $0A \rightarrow 00A1,$   
 $A \rightarrow \epsilon$

## Definition (Derivação num passo)

Seja  $G = (\Sigma, N, P, S)$ ,  $\alpha \Longrightarrow \beta \triangleq \{ \exists \gamma_1 \gamma_2, \delta, \eta \in (N \cup \Sigma)^* \mid \alpha = \gamma_1 \delta \gamma_2 \wedge \beta = \gamma_1 \eta \gamma_2 \wedge \delta \rightarrow \eta \in P \}$

A relação binária  $(\Longrightarrow \subseteq ((N \cup \Sigma)^*)^2)$  é designada de **relação de derivação num passo**.

## Definition (Extensões)

- As relações  $\xRightarrow{+}$  e  $\xRightarrow{*}$  são respectivamente o fecho transitivo e o fecho reflexivo-transitivo de  $\Longrightarrow$  e são designados de **relação de derivação não trivial** e de **relação de derivação**.
- Poderemos igualmente utilizar a relação seguinte:

$$\xRightarrow{k} \triangleq \underbrace{\Longrightarrow \circ \Longrightarrow \dots \circ \Longrightarrow}_k$$

## Definition (Linguagem gerada por uma gramática)

Seja  $G$  uma gramática. A linguagem (conjunto) de todas as palavras (constituídas por símbolos terminais) que se consegue produzir por derivação a partir de  $S$  é designada de **linguagem gerada por  $G$** , nota-se  $\mathcal{L}(G)$  e define-se por

$$\mathcal{L}(G) \triangleq \{u \in \Sigma^* \mid S \xRightarrow{*} u\}$$

## Exemplo

Retomemos a gramática anterior:

$$\begin{array}{ccccccc}
 S & \Longrightarrow & 0A1 & \Longrightarrow & 00A11 & \Longrightarrow & 000A111 & \Longrightarrow & \dots \\
 & & \Downarrow & & \Downarrow & & \Downarrow & & \\
 & & 01 \text{ (i.e. } 0 \in 1 = 01) & & 0011 & & 000111 & & 
 \end{array}$$

Uma gramática  $G = (\Sigma, N, P, S)$  é dita

- **Gramática linear esquerda**, se cada regra de produção é da forma  $A \rightarrow By$  ou  $A \rightarrow x$  onde  $A, B \in N$  e  $x, y \in \Sigma^*$
- **Gramática linear direita**, se cada regra de produção é da forma  $A \rightarrow xB$  ou  $A \rightarrow x$  onde  $A, B \in N$  e  $x \in \Sigma^*$
- As gramáticas lineares esquerda ou direita são também designadas de **gramáticas regulares**.
- **Gramática algébrica**, se cada regra de produção é da forma  $A \rightarrow \alpha$  onde  $A \in N$  e  $\alpha \in (N \cup \Sigma)^*$
- **Gramática contextual**, se cada regra de produção é da forma  $\alpha \rightarrow \beta$  com  $|\alpha| \leq |\beta|$  ou  $\alpha \rightarrow \epsilon$ .

- $\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$  para  $\alpha \rightarrow \beta_1$   
 $\alpha \rightarrow \beta_2$   
...  
 $\alpha \rightarrow \beta_n$
- Quanto tanto possível e sem criar ambiguidades utilizaremos caracteres minúsculos para símbolos terminais e caracteres maiúsculas para símbolos não terminais.
- É fácil de ver que as gramáticas lineares esquerdas e direitas são casos particulares de gramáticas livre de contexto que são elas próprias casos particulares de gramáticas contextuais.
- Excepto menção explícita do contrário, **as gramáticas consideradas no resto da lição são algébricas.**

- Designamos por derivação uma sequência de derivações num passo.

Como por exemplo

$S \implies 0A1 \implies 00A11 \implies 000A111 \implies 000111$ . De facto  
 $S \xRightarrow{*} 000111$ . (Como  $S \xRightarrow{4} 000111$ ).

- Cada passo interno numa derivação  $\dots \alpha \implies \beta \dots$  expande um não terminal (de  $\alpha$ ).
- Uma derivação  $\alpha_1 \implies \alpha_2 \implies \dots \implies \alpha_n$  com  $\alpha_1, \alpha_2, \dots, \alpha_n \in (N \cup \Sigma)^*$  é designada de **derivação esquerda** (respectivamente **derivação direita**) se cada passo da derivação  $\alpha_j \implies \alpha_{j+1}$  expandir o não-terminal mais a esquerda (resp. direita) de  $\alpha_j$ .

- Diz-se duma derivação  $\alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \dots \Longrightarrow \alpha_n$  que tem por tamanho  $n$  ou que deriva  $\alpha_n$  em  $n$  passos.
- Uma derivação  $\alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \dots \Longrightarrow \alpha_n$  é designada de **completa** quando nenhuma regra de produção pode ser aplicada a  $\alpha_n$  (ou seja quando não é possível derivar mais).
- Os conceitos aqui definidos aplicam-se a todo o tipo de gramáticas, excepto as de derivação direita ou esquerda que se aplicam somente para as gramáticas lineares esquerda, direita e gramáticas algébricas.



Com esta definição precisa da noção de gramática, podemos abordar novamente e de forma mais formal a Hierarquia de Chomsky através a sua definição clássica:

- Tipo 0 : Linguagens cujas gramáticas geradoras não padecem de nenhuma restrição.
- Tipo 1 (contextuais): Linguagens cujas gramáticas geradores são as gramáticas contextuais.
- Tipo 2 (algébricas ou livres de contexto): Linguagens cujas gramáticas geradores são as gramáticas algébricas.
- Tipo 3 (regulares): Linguagens cujas gramáticas geradores são as gramáticas lineares direita ou esquerdas.

Temos, obviamente, Tipo 3  $\subseteq$  Tipo 2  $\subseteq$  Tipo 1  $\subseteq$  Tipo 0.

- Define formalmente a gramática  $G$  induzidas pelo conjunto de produções seguintes:  
$$S \rightarrow a S B C \mid a b C$$
$$C B \rightarrow B C$$
$$b B \rightarrow b b$$
$$b C \rightarrow b c$$
$$c C \rightarrow c c$$
- Qual é o tipo desta gramática?
- Apresenta uma derivação completa de  $aaabbbccc$ .
- Qual é a linguagem  $\mathcal{L}(G)$ , gerada por  $G$ ?

- Considere  $\Sigma = \{Jorge, grande, verde, queijo, come, um, o\}$ ,  
 $M = \{S, A, B, C, N, P, V\}$ ,  $R =$ 

$S$	$\rightarrow$	$BVB$
$P$	$\rightarrow$	$N$
$P$	$\rightarrow$	$AP$
$P$	$\rightarrow$	$PA$
$A$	$\rightarrow$	$grande \mid verde$
$B$	$\rightarrow$	$CP \mid P$
$C$	$\rightarrow$	$o \mid um$
$N$	$\rightarrow$	$Jorge$
$N$	$\rightarrow$	$queijo$
$V$	$\rightarrow$	$come$
- $G = \{\Sigma, M, R, S\}$  gera frase como *o Jorge come queijo* mas também *o queijo come grande verde grande verde Jorge grande*
- Apresenta uma derivação completa de *Jorge come um grande queijo verde*.

De onde vem o apelido “livre de contexto”? do teorema seguinte:

## Theorem

Seja  $G = (\Sigma, N, P, S)$  uma gramática de tipo 2 (algébrica). Sejam  $\alpha_1, \alpha_2, \beta$  três palavras de  $(N \cup \Sigma)^*$ , seja  $k \in \mathbb{N}^*$

Se  $\alpha_1 \alpha_2 \xrightarrow{k} \beta$  então existem  $\beta_1, \beta_2 \in (N \cup \Sigma)^*$  tais que  $\beta = \beta_1 \beta_2$  e  $\alpha_1 \xrightarrow{k_1} \beta_1$  e  $\alpha_2 \xrightarrow{k_2} \beta_2$  e  $k = k_1 + k_2$ .

Ou seja, as derivações incidentes de não-terminais distintos nunca interferem entre eles.

# Árvores de Derivação

---

## Um Exemplo

Consideremos:

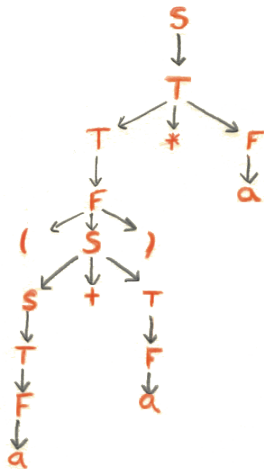
$$G \triangleq (\{a, +, *, (, )\}, \{S, T, F\}, \left\{ \begin{array}{l} S \rightarrow S + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (S) \mid a \end{array} \right\}, S)$$

A derivação esquerda

$$\begin{aligned} S &\Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (S) * F \Rightarrow \\ (S + T) * F &\Rightarrow (T + T) * F \Rightarrow (F + T) * F \Rightarrow \\ (a + T) * F &\Rightarrow (a + F) * F \Rightarrow (a + a) * F \Rightarrow (a + a) * a \end{aligned}$$

$$(S \xrightarrow{11} (a + a) * a \text{ ou } S \xrightarrow{*} (a + a) * a)$$

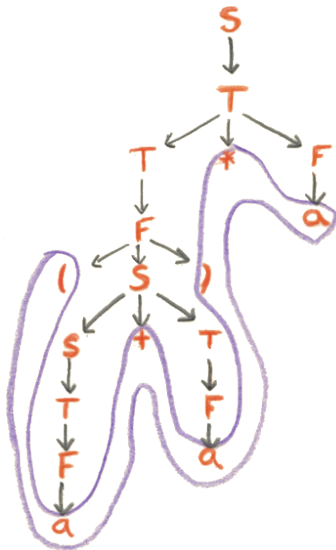
pode ser representada graficamente por uma árvore, designada de **árvore de derivação** ou **árvore de sintaxe** ou ainda de **árvore de derivação sintáctica**.



## Palavra gerada por uma árvore de derivação

- Só estamos interessados nas árvores de derivação cuja raiz é o axioma da gramática
- As folhas destas árvores são necessariamente terminais (símbolos do alfabeto da linguagem gerada)
- a palavra gerada por uma árvore de derivação é a concatenação de todos os terminais encontrados num percurso em profundidade em primeiro da esquerda para a direita
- No exemplo recedente a palavra gerada é sem surpresa  $(a + a) * a$ .

# Palavra gerada por uma árvore de derivação





## Um Exemplo

Qual é a árvore de derivação de:

$$\begin{aligned} S &\Longrightarrow T \Longrightarrow T * F \Longrightarrow T * a \Longrightarrow F * a \Longrightarrow \\ (S) * a &\Longrightarrow (S + T) * a \Longrightarrow (S + F) * a \Longrightarrow \\ (S + a) * a &\Longrightarrow (T + a) * a \Longrightarrow (F + a) * a \Longrightarrow (a + a) * a \end{aligned}$$

As árvores de derivação podem ser definidas por indução.

## Definition (Árvores de derivação)

Seja  $G = (\Sigma, N, P, S)$  uma gramática algébrica. O conjunto das árvores de derivação  $\mathcal{D}_G$  sobre a gramática  $G$  é indutivamente definido por:

- (Base)  $\forall a \in \Sigma, a \in \mathcal{D}_G$

- (Indutivo)

- $\forall (A \rightarrow \epsilon) \in P, A \in \mathcal{D}_G$

$$\downarrow$$

$$\epsilon$$

- $\forall Ar_1, Ar_2, \dots, Ar_n \in \mathcal{D}_G$  (onde  $Ar_1$  tem por raiz  $A_1$ ,  $Ar_2$  tem por raiz  $A_2$ , ..., e  $Ar_n$  tem por raiz  $A_n$ ) e

$$\forall (A \rightarrow A_1 A_2 \dots A_n) \in R, A \in \mathcal{D}_G$$

$$\begin{array}{ccc} \swarrow & \dots & \searrow \\ Ar_1 & \dots & Ar_n \end{array}$$

# Princípios de indução e métodos de demonstração sobre gramáticas

- Qual é o princípios de indução associada a definição indutiva de árvore de derivação?
- Este princípios é particularmente adequado a demonstração de propriedades sobre gramáticas algébricas.
- Podem igualmente ser consideradas induções estruturais sobre o comprimento das derivações, alturas das árvores de derivação. etc...

# Definições e algumas considerações

- Duas derivações que originam a mesma árvore de derivação são ditas **equivalentes**. De facto a relação binária “**ter a mesma árvore de derivação**” é uma relação de equivalência.
- Seja  $u \in \Sigma^*$  a conclusão duma derivação completa a partir de  $S$  (como  $(a + a) * a$ , por exemplo). Se existir duas árvores de derivação diferentes que geram a palavra  $u$  então diz-se que a gramática em questão é **ambígua**.
- Existem linguagens algébricas para as quais não existe nenhuma gramática geradora que não seja ambígua. Tais linguagens são designadas de **inerentemente ambíguas**.
- As gramáticas que não são inerentemente ambíguas são particularmente importantes no desenho das linguagens de programação, veremos em detalhe estas gramáticas na disciplina de compiladores. Estas dão origem a autómatos reconhedores deterministas (**deterministic pushdown automata**) são tratadas algoritmicamente de forma eficiente por analisadores sintácticos ascendentes e descendentes.

# Propriedades de fecho das linguagens regulares e algébricas

---

## Linguagens Regulares

Sejam  $L_1$  e  $L_2$  duas linguagens regulares sobre o alfabeto  $\Sigma$

- $L_1 \cup L_2$  é regular (por definição)
- $L_1.L_2$  é regular (por definição)
- $L_1^*$  é regular (por definição)
- $\tilde{L}_1$  é regular
- $\overline{L_1}$  é regular
- $L_1 \cap L_2$  é regular (porque  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ )

Sejam  $L_1$  e  $L_2$  duas linguagens regulares sobre o alfabeto  $\Sigma$ , seja  $w$  uma palavra de  $\Sigma^*$ . Os seguintes problemas têm solução algorítmica.

- $w \in L_1$ ?
- $L_1 = \emptyset$ ?
- $L_1 = \Sigma^*$ ? ( $\overline{L_1} = \emptyset$ ?)
- $L_1 \subseteq L_2$ ? ( $\overline{L_2} \cup L_1 = \emptyset$ ?)
- $L_1 = L_2$ ? ( $L_1 \subseteq L_2$  e  $L_2 \subseteq L_1$ )

# Propriedades de fecho das linguagens regulares e algébricas

---

## Linguagens Algébricas



Sejam  $L_r$  uma linguagem regular,  $L_1$  e  $L_2$  duas linguagens algébricas sobre o alfabeto  $\Sigma$

- $L_1 \cup L_2$  é algébrica
- $L_1.L_2$  é algébrica
- $L_1^*$  é algébrica
- $\overline{L_1}$  e  $L_1 \cap L_2$  **não são** necessariamente algébricas!
- $L_r \cap L_1$  é algébrica.

Sejam  $L_1$  e  $L_2$  duas linguagens algébricas sobre o alfabeto  $\Sigma$ , seja  $w$  uma palavra de  $\Sigma^*$ . Os seguintes problemas têm solução algorítmica.

- $w \in L_1$ ?
- $L_1 = \emptyset$ ?

Os seguinte problemas não têm solução algorítmica:

- $L_1 = \Sigma^*$ ?
- $L_1 \cap L_2$ ?
- $L_1 = L_2$ ?

Tendo  $L_1$  e  $L_2$  algébricas,

- $L_1 \cup L_2$  é algébrica.

**Demonstração:** Sejam  $G_1 = \{V_1, \Sigma_1, R_1, S_1\}$  e  $G_2 = \{V_2, \Sigma_2, R_2, S_2\}$  as gramáticas geradoras de  $L_1$  e  $L_2$ . Sem perda de generalidade vamos assumir que  $V_1 \cap V_2 = \emptyset$ . Então a gramática  $G = \{V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1; S \rightarrow S_2\}, S\}$  é algébrica e gera a linguagem  $L_1 \cup L_2$ . Para ficar convencido deste facto basta verificar que  $\forall w \in (\Sigma_1 \cup \Sigma_2)^*, S \xRightarrow{*} w$  se e só se  $(S_1 \xRightarrow{*} w) \vee (S_2 \xRightarrow{*} w)$ . Como os conjuntos  $V_1$  e  $V_2$  são disjuntos então a disjunção equivale a afirmar que  $w \in \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$

- $L_1.L_2$  é algébrica. (considerar a gramática  $\{V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1S_2\}, S\}$ )
- $L_1^*$  é algébrica. (considerar a gramática  $\{V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \rightarrow \epsilon, S \rightarrow SS_1\}, S\}$ )

## Algumas consequências interessantes

Para  $L_1$  e  $L_2$  linguagens algébricas, ter  $L_1 \cup L_2$ ,  $L_1.L_2$  e  $L_1^*$  algébricas permite um método modular de construção de gramáticas:

- Como definir uma gramática para a linguagem algébrica  $\{c^p(ab^n)^k \mid p, n > 0 \wedge k \geq 0\}$  sobre o alfabeto  $\Sigma = \{a, b, c\}$ ?
- $S_1 \rightarrow a$  é a gramática que gera a linguagem  $\{a\}$
- $S_2 \rightarrow bS_2 \mid b$  é a gramática que gera a linguagem  $\{b^n \mid n > 0\}$
- Que linguagem gera a gramática com as produções  $\{S_1 \rightarrow a; S_2 \rightarrow bS_2 \mid b; S_3 \rightarrow S_1 \mid S_2\}$  e com o axioma  $S_3$ ? Pelas propriedades de fecho das linguagens algébricas, é a linguagem  $a|b^+$ . Logo, não é a gramática desejada.
- Alteramos a produção implicando  $S_3$  em  $S_3 \rightarrow S_1S_2$ , a linguagem gerada é então  $ab^+$ .
- Se juntarmos a regra  $S_4 \rightarrow S_3S_4 \mid \epsilon$  e utilizar  $S_4$  como o axioma então geramos  $(ab^+)^*$ .
- A gramática  $S_5 \rightarrow cS_5 \mid \epsilon$  gera a linguagem  $c^*$
- A gramática  $S_6 \rightarrow S_5 \mid S_4$  com  $S_6$  axioma gera a linguagem  $c^* + (ab^+)^*$

## Algumas consequências interessantes

- Alteremos a gramática. A gramática  $S_6 \rightarrow S_5 S_4$  com  $S_6$  axioma gera a linguagem  $c^*(ab^+)^*$
- Ainda não é exactamente o que pretendemos. De forma semelhante (exercício: como?) é simples construir de forma modular uma gramática de axioma, digamos,  $S_7$  que gere a linguagem  $c^+(ab^*)^*$ .
- A linguagem desejada é assim a união destas duas linguagens (i.e.  $(c^*(ab^+)^*)|(c^+(ab^*)^*)$  ).
- Basta assim considerar a gramática de axioma  $S_8$  que junta a todas as produções definidas a produção  $S \rightarrow S_6|S_7$

# Transformações Gramaticais

---

Gramáticas Regulares vs. Autómatos Finitos

- Já sabemos pelo teorema de Kleene que Linguagem Regular = Autómatos Finitos = Expressões Regulares
- Vamos aqui completar esta equação com Autómatos Finitos = Gramáticas Regulares

**Theorem (As gramáticas lineares (direita) geram linguagens regulares)**

*Seja  $G = (N, \Sigma, S, P)$  uma gramática linear direita, então  $L(G)$  é uma linguagem regular.*



# Gramática linear direita $\implies$ Autómatos finitos

- Demonstração construtiva. Existe um algoritmo que estabelece um autómato finito que reconhece a linguagem gerada por  $G$
- Para começar, vamos assumir (sem perda de generalidade) que  $N = \{V_0, V_1, \dots\}$  com  $V_0 = S$ . As produções são da forma  $V_i \rightarrow x_a V_j$  ou  $V_i \rightarrow x_a$  (com  $x_a \in \Sigma^*$  e  $V_i, V_j \in N$ ).
- Informalmente Cada Não terminal vai produzir um estado no autómato. Vamos do estado  $V_i$  para um estado  $V_j$  com a transição  $x$  se existir uma produção  $V_i \rightarrow xV_j$ . Formalizemos essa ideia...
- O estado inicial é  $V_0$ . Considera-se um único estado final  $V_f$ .
- Para cada produção  $V_i \rightarrow a_1 a_2 \dots a_n V_j$ , criar :



- Para cada produção  $V_i \rightarrow a_1 a_2 \dots a_n$ , criar:



- **Esquema** de demonstração da correcção algorítmica: Seja  $A = \{Q, \Sigma, \delta, S, F\}$  o autómato resultante. Provemos que se  $a_1 a_2 \dots a_m \in L(G)$  então  $w = a_1 a_2 \dots a_m \in L(A)$ .
- Se  $a_1 a_2 \dots a_m \in L(G)$  então existe uma derivação completa da forma  $S \implies a_1 a_2 \dots a_j V_j \implies a_1 a_2 \dots a_j a_{j+1} \dots a_k V_k \implies \dots \implies a_1 a_2 \dots a_m$
- É fácil ver que existe então um caminho em  $A$  de  $V_0$  para  $V_f$  que passa pelos estados  $V_j, V_k, \dots$ . Ou seja  $w$  é reconhecido por  $A$ .
- de forma semelhante. Se uma palavra  $w = a_1 a_2 \dots a_m$  é aceite por  $A$ , então existe um caminho bem sucedido em  $A$  de etiqueta  $w$  que liga  $V_0$  a  $V_f$  passando por estados, digamos  $V_i \dots V_k \dots$ . Neste caso existe uma derivação  $V_0 \implies a_1 a_2 \dots a_j V_j \implies a_1 a_2 \dots a_j a_{j+1} \dots a_k V_k \implies \dots \implies a_1 a_2 \dots a_m$ .  $w \in L(G)$ .

# Autómatos finitos $\implies$ Gramática linear direita

Theorem (As Linguagens regulares são reconhecidas por Gramáticas lineares (direita))

*Se uma linguagem  $L$  é regular então existe uma gramatica linear direita  $G$  que a reconhece*

## Autómatos finitos $\implies$ Gramática linear direita

- Seja  $M = \{Q = \{q_0, q_1, \dots, q_n\}, \Sigma = \{a_1, \dots, a_m\}, \delta, q_0, F\}$  um DFA tal que  $L(M) = L$ .
- Vamos construir a gramática  $G = \{N, \Sigma, S, P\}$  da seguinte forma:
- $N = Q$
- $S = q_0$
- Para cada transição  $\delta(q_i, a_j) = q_k$  definir a transição  $q_i \rightarrow a_j q_k$
- se  $q_k \in F$  juntar  $q_k \rightarrow \epsilon$  a  $P$
- Demonstração deixada em exercício.

## Theorem

*Gramáticas lineares = Linguagem Regulares Uma linguagem  $L$  é regular se e só se é gerada por uma gramática regular (aqui linear direita)*

A demonstração é trivial....

# Transformações Gramaticais

---

Algoritmos de transformações gramaticais

- Defina um algoritmo que transforme uma gramática linear esquerda em gramática linear direita.
- Demonstre a correcção do algoritmo.

## Theorem

Seja  $G = \{\Sigma, N, S, P\}$  uma gramática algébrica. Suponha que  $P$  contenha uma produção da forma  $A \rightarrow x_1 B x_2$  (\*), onde  $x_1, x_2 \in (N \cup \Sigma)^*$ , e  $B \in N, \wedge B \neq A$ .

Suponha igualmente que as produções com  $B$  no lhs não são recursivas, ou seja da forma  $B \rightarrow y_1 | y_2 | \dots | y_n$  com  $y_i \in (N - \{B\} \cup \Sigma)^*$ .

Seja  $G = \{N, \Sigma, S, P'\}$  onde

$$P' = P - \{A \rightarrow x_1 B x_2\} \cup \{A \rightarrow x_1 y_1 x_2 | x_1 y_2 x_2 | \dots | x_1 y_n x_2\}$$

Então  $L(G) = L(G')$



**Demonstração:** Vamos nos limitar a demonstração de que  $L(G) \subseteq L(G')$  (a demonstração no outro sentido é muito semelhante)

Suponha que exista um  $w \in L(G)$  tal que  $S \xRightarrow{*}_G w$ . Dois casos se apresentam:

1. A regra (\*) não participa à derivação. Neste caso é trivial ver que  $S \xRightarrow{*}_{G'} w$ , visto que, então, só foram utilizadas regras que estão em ambas as gramáticas.
2. A regra (\*) participa à derivação. Vamos proceder por indução sobre o numero de vezes que a regra (\*) é utilizada na derivação.

- caso de base. A regra é utilizada uma vez. Neste caso consideremos a derivação

$$S \xRightarrow{*}_G u_1 A u_2 \implies \underbrace{G u_1 x_1 B x_2 u_2 \implies G u_1 x_1 y_j x_2 u_2}_{(*) \text{ e } B \rightarrow y_j} \xRightarrow{*}_G w. \text{ Esta}$$

derivação pode ser substituída por

$$S \xRightarrow{*} u_1 A u_2 \implies u_1 x_1 y_j x_2 u_2 \xRightarrow{*} w \text{ utilizando directamente a regra } A \rightarrow x_1 y_j x_2 \text{ de } G'. \text{ Done.}$$

## Demonstração (continuação):

- Caso indutivo. (HI) Sabemos transformar derivações com  $n$  utilizações da regra (\*) em derivações que utilizam exclusivamente regras de  $G'$ . Como transformar derivações com  $n + 1$  utilização de regras (\*)? Primeiro, começamos por remover  $n$  ocorrências e removemos a ocorrência que falta utilizando o mesmo princípio.
- QED

## Definition

Seja  $G = \{\Sigma, N, S, P\}$  uma gramática algébrica. O não-terminal  $A$  é designado de **útil** se existe pelo menos uma palavra  $w$  de  $L(G)$  tal que a sua derivação utilize o não terminal  $A$ , ou seja

$$\exists x, y \in (N \cup \Sigma)^*, S \xRightarrow{*} xAy \xRightarrow{*} w$$

Por outras palavras, o não terminal  $A$  tem forma de participar na produção  $w$ .  
Um não terminal não útil é designado por **inútil**

Ou seja, um não terminal é inútil ou porque nunca será chamado num processo de produção ou porque a sua chamada nunca produzirá palavras (derivações que entram em “ciclo”).

## útil vs. inútil

	$S$	$\rightarrow$	$aSb \mid bA \mid cC$	
Em	$A$	$\rightarrow$	$aA \mid \epsilon$	$S$ e $A$ são úteis, $B$ e $C$ são inúteis.
	$B$	$\rightarrow$	$bA \mid BaS$	
	$C$	$\rightarrow$	$cC$	

## Theorem

*Seja  $G = \{\Sigma, N, S, P\}$  uma gramática algébrica. Então existe uma gramática  $G' = \{N', \Sigma', S, P\}$  equivalente a  $G$  que não contém nenhum não terminal e nenhuma regra inútil.*

- Uma demonstração construtiva completa obriga a uma definição dum algoritmo de transformação seguida dum demonstração de correcção (de que o algoritmo faz bem o que é pretendido, ou seja que produz uma gramática **equivalente** e **sem regras/produções inúteis**).
- Vamos aqui nos limitar ao desenho do algoritmo.
- É no entanto um excelente exercício (ver teorema anterior) demonstrar a correcção do algoritmo.

# Algoritmo para remover regras inúteis

Começemos por ilustrar os princípios do algoritmo via um exemplo.

$$\begin{array}{lcl} 1 & S & \rightarrow aS \mid A \mid C \\ 2 & A & \rightarrow a \\ 3 & B & \rightarrow aa \\ 4 & C & \rightarrow aCb \end{array}$$

1. Identificar os não terminais que produzem directamente terminais. Pelas regras 2 e 3 os não terminais  $A$  e  $B$ .  $A$  e  $B$  são não terminais produtores.
2.  $S$  gera  $A$  que gera  $a$ , logo  $S$  produz.
3. Este raciocínio não pode ser feito para  $C$  que de facto possui uma (única) regra recursiva logo iniciadora dum processo de redução infinita.  $C$  não é produtor, logo é inútil.
4. De forma similar,  $S$  gera algo graças a  $A$  e a ele próprio (regra 1). Por seu turno  $A$  gera algo sem depender de nenhum outro terminal. Ou seja  $S$  e  $A$  intervenham no processo de produção (são **atingíveis**).  $A$  e  $S$  são úteis.
5. Nesta análise  $B$  não tem possibilidade de intervir. Logo  $B$  é inútil (apesar de ser produtor).

6. no final resta-nos

$$\begin{array}{lcl} 1 & S & \rightarrow aS \mid A \\ 2 & A & \rightarrow a \end{array}$$

- Mais formalmente. Um não terminal é inútil quando **não produz** ou quando é **inatingível**.
- Assim, como definir (e discriminar) todos os terminais produtivos?
- um não terminal  $A$  é produtivo quando
  - Existe uma produção  $A \rightarrow \alpha$ , com  $\alpha \in \Sigma^*$ .
  - Caso contrário, quando existe uma produção  $A \rightarrow \beta$  com  $\beta \in (N \cup \Sigma)^+$  onde todos os não terminais de  $\beta$  são produtivos.
  - ou seja um não terminal é produtivo se é *lhs* duma regra produtiva.

- Para obter a lista dos não-terminais produtíveis basta assim proceder por procura de ponto fixo.
- Listamos todas as regras numa tabela. Na etapa zero marcamos todas as regras que produzem directamente ( $A \rightarrow \alpha$ , com  $\alpha \in \Sigma^*$ ).
- As equações por considerar são: Para toda a regra  $A \rightarrow \alpha$ , esta produz na etapa  $i + 1$  se  $\alpha \in \Sigma^*$  ou se  $\alpha \in (N \cup \Sigma)^+$  onde todos os não terminais de  $\alpha$  são produtivos na etapa  $i$  (*lhs* duma regra produtiva).
- Paramos quando a etapa  $i + 1$  gera a mesma resposta que  $i$ , a resposta final. Os não terminais produtivos são os não terminais que são *lhs* de regras assinaladas como produtivas.

- Como definir (e discriminar) todos os terminais inatingíveis ?
- um não terminal  $A$  é inatingível quando
  - $A = S$
  - Caso contrário, quando existe uma produção  $B \rightarrow \alpha$ , com  $\alpha \in (N \cup \Sigma)^*$  tal que  $A \in \alpha$  e  $B$  seja ele próprio atingível.
  - Ou seja, um não terminal é atingível se é  $S$  ou se está no *rhs* duma regra atingível
- Para obter a lista dos não-terminais atingíveis basta proceder por procura de ponto fixo.
- Listamos todas as regras numa tabela. Na etapa zero marcamos as regras onde  $S$  é *lhs* como atingível.
- As equações por considerar são: Para todo o  $A$  de  $N$ ,  $A$  é atingível na etapa  $i + 1$  se  $A = S$ , ou se existe uma produção  $B \rightarrow \beta$  com  $\beta \in (N \cup \Sigma)^*$  tal que  $A \in \alpha$  e  $B$  seja ele próprio atingível na etapa  $i$ .
- Paramos quando a etapa  $i + 1$  gera a mesma resposta que  $i$ , a resposta final. Os não terminais atingíveis são os não terminais *lhs* de regras marcadas como atingíveis.



## Um exemplo

- 1  $S \rightarrow aS \mid A \mid C$
- 2  $A \rightarrow a$
- 3  $B \rightarrow aa$
- 4  $C \rightarrow aCb$

Regras	etapa 0		etapa 1		etapa 2		etapa 3	
	Prod.	Ating.	Prod.	Ating.	Prod.	Ating.	Prod.	Ating.
$S \rightarrow aS$		✓		✓	✓	<u>✓</u>	<u>✓</u>	
$S \rightarrow A$		✓	✓	✓	✓	<u>✓</u>	<u>✓</u>	
$S \rightarrow C$		✓		✓		<u>✓</u>		
$A \rightarrow a$	✓		✓	✓	✓	<u>✓</u>	<u>✓</u>	
$B \rightarrow aa$	✓		✓		✓		<u>✓</u>	
$C \rightarrow aCb$				✓		<u>✓</u>		

Produtíveis: A,B,S

Atingíveis: S,A,C

- Um não-terminal  $A$  é marcado como produtivo por este processo quando existe uma derivação  $A \xRightarrow{*} w$ . Este processo pode ser demonstrado por indução sobre as arvores de derivação resultantes.
- Um não-terminal  $A$  é marcado como atingível quando existe uma derivação  $S \xRightarrow{*} \alpha A \beta$  (com  $\alpha, \beta \in (N \cup \Sigma)^*$ ). Este facto pode ser demonstrado por indução sobre o tamanho da derivação.

## Definition

- Uma **produção**  $\epsilon$  é uma produção da forma  $A \rightarrow \epsilon$ .
- Um não-terminal  $A$  é **anulável** se  $A \xRightarrow{*} \epsilon$ .

## Theorem

*Seja  $G$  uma gramática livre de contexto tal que  $\epsilon \notin L(G)$ , existe então uma gramática  $G'$  livre de contexto sem produções  $\epsilon$  tal que  $L(G) = L(G')$ .*

Demonstração: Construtiva.

Primeiro é preciso determinar quais são os não terminais anuláveis.

- Para obter a lista dos não-terminais anuláveis procede-se por procura de ponto fixo.
- Para começar, sabemos que uma regra da forma  $A \rightarrow \beta$  tal que  $\beta$  contenha um terminal nunca será anulável
- Listamos todas as regras numa tabela. Na etapa zero marcamos as produções  $\epsilon$ .
- As equações por considerar são: Para toda a produção  $A \rightarrow A_1A_2 \cdots A_n$  de  $P$  (com  $A_1, A_2, \dots, A_n \in N$ ),  $A \rightarrow A_1A_2 \cdots A_n$  é anulável na etapa  $i + 1$  se todos os  $A_1A_2 \cdots A_n$  são anuláveis na etapa  $i$  (ou seja *lhs* de regras marcadas como anuláveis).
- Paramos quando a etapa  $i + 1$  gera a mesma resposta que  $i$ , a resposta final. Os não terminais anuláveis são os não terminais *lhs* de regras marcadas como anuláveis.

- Sabendo agora quais são os não terminais anuláveis, podemos construir o novo conjunto de produção da forma seguinte:
- Considerar todas as regras da forma  $A \rightarrow x_1 x_2 \dots x_m$  (com  $m \geq 1$  e  $\forall i \in \{1..m\}, x_i \in N \cup \Sigma$ ) onde exista pelo menos um  $x_j$  não terminal anulável. Cada uma delas é substituída por um conjunto  $\mathcal{C}$  de regras determinado da forma seguinte:
- Seja  $X = \{x_{i_1} x_{i_2} \dots x_{i_p}\}$  o subconjunto de  $\{x_1 x_2 \dots x_m\}$  dos não-terminais anuláveis presentes na regra considerada.
- Considera-se então o conjunto  $\mathcal{P}(X)$  dos subconjuntos de  $X$ .
- Cada elemento de  $\mathcal{P}(X)$  vai gerar uma nova regra de produção, da forma seguinte: para todo o  $\{x_{j_1} x_{j_2} \dots x_{j_q}\} \in \mathcal{P}(X)$ , considerar a regra inicial  $A \rightarrow x_1 x_2 \dots x_m$  e retirar-lhe todos os não terminais anuláveis excepto  $\{x_{j_1} x_{j_2} \dots x_{j_q}\}$ . A regra resultante é acrescentada a  $\mathcal{C}$ .
- existe no entanto uma excepção a este algoritmo: se todos os  $x_i$ s são anuláveis, não se considera a regra  $A \rightarrow \epsilon$  no conjunto  $\mathcal{C}$ .

- Seja  $G$  uma gramática livre de contexto. Seja  $G'$  a gramática livre de contexto obtida a partir deste algoritmo
- Demonstração da correcção do algoritmo consiste em verificar que  $L(G) = L(G')$ .
- $L(G) \subseteq L(G')$  (ou seja  $\forall w \in L(G), w \in L(G')$ ). (esqueleto)  
Demonstração sobre o número de produções  $\epsilon$  utilizadas para produzir  $w$ .
- $L(G') \subseteq L(G)$  (ou seja  $\forall w \in L(G'), w \in L(G)$ ). (esqueleto)  
Demonstração sobre o número de regras produzidas pelo algoritmo utilizadas para produzir  $w$ .

## Um exemplo

$$S \rightarrow ABaC$$

$$A \rightarrow BC$$

$$B \rightarrow b \mid \epsilon$$

$$C \rightarrow D \mid \epsilon$$

$$D \rightarrow d$$

$A, B, C$  são anuláveis. Logo a gramática obtida é:

$$S \rightarrow ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Aa \mid Ba \mid a$$

$$A \rightarrow BC \mid B \mid C$$

$$B \rightarrow b$$

$$C \rightarrow D$$

$$D \rightarrow d$$



## Definition

Uma **produção unitária** é uma produção da forma  $A \rightarrow B$  com  $A, B \in N$ .

## Theorem

*Seja  $G$  uma gramática livre de contexto sem produções  $\epsilon$ . Então existe uma gramática  $G'$  equivalente a  $G$  tal que  $G'$  não contenha produções unitárias.*

Demonstração: Mais uma vez, construtiva.

- Remover directamente todas as regras da forma  $A \rightarrow A$
- Para cada não terminal  $A$  de  $N$  calcular (com base numa procura de ponto fixo) todos os  $B \in N$  tais que  $A \xRightarrow{*} B$
- As produções de  $G'$  são então calculadas da seguinte forma:
  - primeiro, considerar todas as produções não unitárias de  $G$ , colocá-las em  $P'$
  - Considerar todos os casos  $A \xRightarrow{*} B$  e para cada um deles juntar a  $P'$   $A \rightarrow y_1 \mid y_2 \mid y_3 \mid \dots \mid y_n$  para cada regra  $B \rightarrow y_1 \mid y_2 \mid y_3 \mid \dots \mid y_n$  retirado de  $P'$ .

A demonstração de correcção segue os moldes da demonstração da correcção da regra de substituição.

# Como simplificar uma gramática?

Para ter uma gramáticas sem produções  $\epsilon$  (excepto  $S \rightarrow \epsilon$ ), produções unidade e inúteis, é preciso aplicar os algoritmos de remoção de regras **nesta ordem** (exercício: porquê?)

1. Remover transições  $\epsilon$
2. Remover produções unidade
3. Remover produções inúteis

# Transformações Gramaticais

---

## Formas Normais

## Definition

Uma gramática  $G$  livre de contexto está em **forma normal de Chomsky** se todas as produções são da forma:

- $A \rightarrow BC$  (com  $A, B, C \in N$ ) ou
- $A \rightarrow a$  (com  $A \in N$  e  $a \in \Sigma$ )

## Theorem

*Seja  $G$  uma gramática algébrica tal que  $\epsilon \notin L(G)$ , então existe uma gramática  $G'$  em forma normal de Chomsky equivalente a  $G$ .*

- Demonstração construtiva fácil.
- Para começar, simplificar a gramática usando as transformações anteriores.
- Considere as seguintes transformações:
- $A \rightarrow x_1^1 \dots x_{p_1}^1 B_1 x_1^2 \dots x_{p_2}^2 B_2 \dots x_1^q \dots x_{p_q}^q B_q x_1^{q+1} \dots x_{p+1}^{q+1}$  (com  $x_i \in \Sigma$  e  $B_j \in N$ )

Transformar em

$$\left\{ \begin{array}{l} A \rightarrow C_1^1 \dots C_{p_1}^1 B_1 C_1^2 \dots C_{p_2}^2 B_2 \dots C_1^q \dots C_{p_q}^q B_q C_1^{q+1} \dots C_{p+1}^{q+1} \\ C_1^1 \rightarrow x_1^1 \\ \vdots \\ C_{p_1}^1 \rightarrow x_{p_1}^1 \\ C_1^2 \rightarrow x_1^2 \\ \vdots \\ C_{p+1}^{q+1} \rightarrow x_{p+1}^{q+1} \end{array} \right.$$

- $A \rightarrow B_1 B_2 \dots B_n$  (com  $B_i \in N$ ) Transformar em

$$\left\{ \begin{array}{l} A \rightarrow B_1 B'_2 \\ B'_2 \rightarrow B_2 B'_3 \\ \vdots \\ B'_{n-1} \rightarrow B_{n-1} B_n \end{array} \right.$$

- Estas regras quando aplicadas resultam numa gramática na forma normal de Chomsky.
- Todas estas transformações utilizam a regra de substituição simples cuja correcção foi comprovada.

# Parsing e membership

---

Um algoritmo para determinar se  $w \in L(G)$



# Um algoritmo de parsing para gramáticas em Forma Normal de Chomsky

- Vamos expor aqui um algoritmo que permita determinar se uma palavra é gerada por uma gramática.
- Este algoritmo supões que a gramática em cause está na forma normal de Chomsky
- Autores: J. Cocke, D.H. Younger e T. Kasami.
- Nome: **algoritmo CYK**
- Complexida  $\mathcal{O}(|w|^3)$ , sendo  $w$  a palavra por analisar.

# Um algoritmo de parsing para gramáticas em Forma Normal de Chomsky

## CYK

- $G = (N, \Sigma, P, S)$  em forma normal de Chomsky.
- $w = a_1 a_2 a_3 \cdots a_n$
- $w_{ij} = a_i \cdots a_j$
- $V_{ij} = \{A \in V \mid A \xRightarrow{*} w_{ij}\}$
- neste caso  $w \in L(G)$  se e só se  $S \in V_{1n}$
- Como calcular os  $V_{ij}$ ?
- Observa-se que  $A \in V_{ii}$  se  $(A \rightarrow a_i) \in P$
- Temos assim forma de calcular todos os  $V_{ij}$  associados a  $w$  (com  $1 \leq i \leq n$ ).

# Um algoritmo de parsing para gramáticas em Forma Normal de Chomsky

## CYK - continuação

- Observa-se igualmente que  $A \xRightarrow{*} w_{ij}$  se e só se existem um inteiro  $k$  tal que  $i \leq k < j$  e uma produção  $A \rightarrow BC$  tal que  $B \xRightarrow{*} w_{ik}$  e  $C \xRightarrow{*} w_{kj}$ .
- Por outras palavras

$$V_{ij} = \bigcup_{k \in \{i, i+1, \dots, j-1\}} \{A \mid A \rightarrow BC, B \in V_{ik}, C \in V_{k+1j}\} (*)$$

- Assim de (\*) vemos que podemos calcular os diferentes  $V_{ij}$  sequencialmente:
  - $V_{11}, V_{22}, \dots, V_{33}$
  - $V_{12}, V_{23}, \dots, V_{(n-1)n}$
  - $V_{13}, V_{24}, \dots, V_{(n-2)n}$
  - etc.

# Um algoritmo de parsing para gramáticas em Forma Normal de Chomsky

## CYK - continuação

- Há exactamente  $\frac{n(n+1)}{2}$  conjuntos  $V_{ij}$
- o calculo de cada um necessita do calculo de  $n$  componentes de (\*)
- daí a complexidade  $\mathcal{O}(n^3)$  (com  $|w| = n$ )

$$w = aabbb, \in G?$$

com  $G$  contendo as produções:

$$S \rightarrow UV$$

$$U \rightarrow UV \mid a$$

$$V \rightarrow UV \mid b$$

- $N_{11} = \{U\}$ ,  $N_{22} = \{U\}$ ,  $N_{33} = \{V\}$ ,  $N_{44} = \{V\}$ ,  $N_{55} = \{V\}$
- $N_{12} = \{A \mid A \rightarrow BC, B \in N_{11}, C \in N_{22}\} = \emptyset$   
 $N_{23} = \{A \mid A \rightarrow BC, B \in N_{22}, C \in N_{33}\} = \{S, V\}$ ,  
da mesma forma  $N_{34} = \{U\}$ ,  $N_{45} = \{U\}$
- $N_{13} = \{A \mid A \rightarrow BC, B \in N_{11}, C \in N_{23}\} \cup \{A \mid A \rightarrow BC, B \in N_{12}, C \in N_{33}\} = \{S, V\}$ ,  
 $N_{24} = \{U\}$ ,  $N_{35} = \{S, V\}$
- $N_{14} = \{U\}$ ,  $N_{25} = \{S, V\}$
- $N_{15} = \{S, V\}$ , ou seja  $S$  está em  $N_{15}$  logo

SIM.