Teoria da Computação

Introdução às Linguagens Formais e as suas Gramáticas

Simão Melo de Sousa

Computer Science Department
University of Beira Interior, Portugal

Plano

Plano

Aviso Prévio

- **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 à 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;**

Referencias bibliográficas

- (Principal) C. H. Papadimitriou, H. R. Lewis. emphElements of the Theory of Computation por Prentice Hall, 1997. Tradução brasileira: *Elementos de Teoria da Computação*, 2a Edição. Bookman, Porto Alegre, 2000.
- (Introdutório, em francês - embora deva existir algo em inglês algures) P. Wolper. *Introduction à la calculabilité*, 3ª edição, Dunod, 2006.
- (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 completo... um "must") John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation* (3nd Edition). Addison Wesley, 2006 (existe em português do Brasil).
- (Abordagem abrangente) M. Sipser. *Introducton to the Theory of Computation*. PWS Publishing, 2006.

O que é a teoria das linguagens formais?

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

Processos de comunicação

- 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 $\mathcal{L}$.

Objectivo da Teoria

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)

e no contexto desta disciplina?

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.

Poder expressivo

- 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 $A \subseteq B$.

Linguagens finitas

- **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).

Linguagens Regulares

- **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.

Linguagens Algébricas - Livres de Contexto

- **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.

Linguagens Decidíveis

- **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 (λ-termos, máquinas de Turing totais, etc...)

Linguagens Semi-Decidíveis

- **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.

Linguagens Quaisquer

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

Alguma considerações

- 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.

Plano

Monoíde Livre

- 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 $a = a_1 a_2 a_3 \ldots a_n$ tem por comprimento n. Notação: $|a| = n$ ou $\sharp(a) = n$.
- A palavra vazia, notada ϵ, é a palavra de comprimento 0.
- Seja . a operação de concatenação, $(A^*, .)$ é designado de monoíde livre gerado por A.
  - . é associativa
  - ϵ é o elemento neutro de .

O que é uma linguagem?

- 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 $\emptyset \neq \epsilon$)
  - $\bar{\mathcal{L}} \triangleq \{u \in A^* | u \notin \mathcal{L}\}$
  - $\mathcal{L}\mathcal{L}' \triangleq \{u.v | u \in \mathcal{L} \wedge v \in \mathcal{L}'\}$
  - $\mathcal{L}^* \triangleq \{\epsilon\} \cup \mathcal{L} \cup \mathcal{L}\mathcal{L} \cup \mathcal{L}\mathcal{L}\mathcal{L} \ldots \triangleq \bigcup_{n \in \mathbb{N}} \mathcal{L}^n$
  - $\mathcal{L}^+ \triangleq \mathcal{L} \cup \mathcal{L}\mathcal{L} \cup \mathcal{L}\mathcal{L}\mathcal{L} \ldots \triangleq \bigcup_{n \in \mathbb{N}^*} \mathcal{L}^n$
  - $\mathcal{L}^{<n} \triangleq \{\epsilon\} \cup \mathcal{L} \cup \mathcal{L}\mathcal{L} \cup \mathcal{L}\mathcal{L}\mathcal{L} \ldots \underbrace{\mathcal{L} \cdots \mathcal{L}}_{n-1}$
  - $\tilde{\mathcal{L}} \triangleq \{a_n a_{n-1} \ldots a_2 a_1 | a_1 a_2 \ldots a_{n-1} a_n \in \mathcal{L}\}$ (linguagem espelho)

são igualmente linguagens.

Plano

Definição

- Gramática de Chomsky: sistema de produção de palavras, baseado em dois alfabetos N, Σ e um conjunto de regras de reescrita (designadas de **produção** ou de **regras**) P.
- Σ: Alfabeto sobre o qual a linguagem $\mathcal{L}$ é definida. **Conjunto dos símbolos terminais**.
- N: Conjunto de símbolos, distinto de Σ, 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 Σ.

Definition (Gramáticas de Chomsky)

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

- Σ, 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 $P \subseteq ((N \cup \Sigma)^* - \Sigma^*) \times (N \cup \Sigma)^*$

Comentários e notação

- P é finito e é constituído por pares (α, β). Notação $\alpha \rightarrow \beta$.
- Lê-se de α *produz-se* β
- 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**.

Exemplo

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

Derivação e Linguagem Gerada

Definition (Derivação num passo)

Seja $G = (\Sigma, N, P, S)$, $\alpha \implies \beta \triangleq \{\exists \gamma_1\gamma_2, \delta, \eta \in (N \cup \Sigma)^* | \; \alpha = \gamma_1\delta\gamma_2 \wedge \; \beta = \gamma_1\eta\gamma_2 \wedge \; \delta \to \eta \in P \}$

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

Definition (Extensões)

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

$$\overset{k}{\implies} \triangleq \underbrace{\implies \circ \implies \ldots \circ \implies}_{k}$$

Linguagens e Gramáticas

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

$$\mathcal{L}(G) \triangleq \{u \in \Sigma^* | S \stackrel{*}{\Longrightarrow} u\}$$

Exemplo

Retomemos a gramática anterior:

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

Algumas gramáticas particulares

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

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

Açúcar Sintáctico a outras considerações

- $\alpha \to \beta_1|\beta_2|\ldots|\beta_n$ para $\begin{array}{l}\alpha \to \beta_1\\ \alpha \to \beta_2\\ \ldots\\ \alpha \to \beta_n\end{array}$
- 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**.

Derivação

- Designamos por derivação uma sequência de derivações num passo. Como por exemplo $S \implies 0A1 \implies 00A11 \implies 000A111 \implies 000111$. De facto $S \overset{*}{\implies} 000111$. (Como $S \overset{4}{\implies} 000111$).
- Cada passo interno duma derivação $\ldots \alpha \implies \beta \ldots$ expande um não terminal (de α).
- Uma derivação $\alpha_1 \implies \alpha_2 \implies \ldots \implies \alpha_n$ com $\alpha_1, \alpha_2, \ldots, \alpha_n \in (N \cup \Sigma)^*$ é designada de **derivação esquerda** (respectivamente **derivação direita**) se cada passo da derivação $\alpha_i \implies \alpha_{i+1}$ expandir o não-terminal mais a esquerda (resp. direita) de α_i.

Derivação

- Diz-se duma derivação $\alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ldots \Longrightarrow \alpha_n$ que tem por tamanho n ou que deriva α_n em n passos.
- Uma derivação $\alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ldots \Longrightarrow \alpha_n$ é designada de **completa** quando nenhuma regra de produção pode ser aplicada a α_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 gramaticas lineares esquerda, direita e gramáticas algébricas.

Hierarquia de Chomsky revisitada

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 gramaticas 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.

Exemplos

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

Exemplos

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

$$\begin{array}{lcl} S & \rightarrow & BVB \\ P & \rightarrow & N \\ P & \rightarrow & AP \\ P & \rightarrow & PA \\ A & \rightarrow & grande \mid verde \\ B & \rightarrow & CP \mid P \\ C & \rightarrow & o \mid um \\ N & \rightarrow & Jorge \\ N & \rightarrow & queijo \\ V & \rightarrow & come \end{array}$$

- $G = \{\Sigma, M, R, S\}$ 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*.

Gramáticas Livre de Contexto

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

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

Plano

O que são?

Um Exemplo

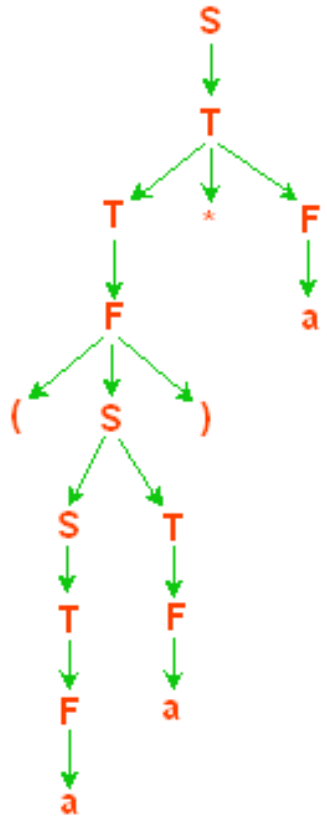
Consideremos:

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

A derivação esquerda

$$\begin{array}{l} S \Longrightarrow T \Longrightarrow T * F \Longrightarrow F * F \Longrightarrow (S) * F \Longrightarrow \\ (S + T) * F \Longrightarrow (T + T) * F \Longrightarrow (F + T) * F \Longrightarrow \\ (a + T) * F \Longrightarrow (a + F) * F \Longrightarrow (a + a) * F \Longrightarrow (a + a) * a \end{array}$$

$(S \overset{11}{\Longrightarrow} (a + a) * a$ ou $S \overset{*}{\Longrightarrow} (a + a) * a)$
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 raíz 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 $(a + a) * a$.

Exercício

Um Exemplo

Qual é a árvore de derivação de:

$$S \Longrightarrow T \Longrightarrow T * F \Longrightarrow T * a \Longrightarrow F * a \Longrightarrow$$
$$(S) * a \Longrightarrow (S + T) * a \Longrightarrow (S + F) * a \Longrightarrow$$
$$(S + a) * a \Longrightarrow (T + a) * a \Longrightarrow (F + a) * a \Longrightarrow (a + a) * a$$

Definição Indutiva

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)
  - $\forall (A \to \epsilon) \in P, \quad \begin{array}{c} A \\ \downarrow \\ \epsilon \end{array} \in \mathcal{D}_G$
  - $\forall Ar_1, Ar_2, \ldots, Ar_n \in \mathcal{D}_g$ (onde Ar_1 tem por raíz A_1, Ar_2 tem por raíz A_2, ..., e Ar_n tem por raíz A_n) e $\forall (A \to A_1 A_2 \ldots A_n) \in R,$ $\begin{array}{ccc} & A & \\ \swarrow & \ldots & \searrow \\ Ar_1 & \ldots & Ar_n \end{array} \in \mathcal{D}_G$

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 origens a autómatos reconhecedores deterministas (*deterministic pushdown automata*) são tratadas algoritmicamente de forma eficiente por analisadores sintácticos ascendentes e descendentes.

Plano

Theoremas...

Sejam L_1 e L_2 duas linguagens regulares sobre o alfabeto Σ

- $L_1 \cup L_2$ 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 $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_1}}$)

.. e algoritmia

Sejam L_1 e L_2 duas linguagens regulares sobre o alfabeto Σ, seja w uma palavra de Σ^*. 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 $L_2 \subseteq L_1)$

Theoremas...

Sejam L_r uma linguagem regular, L_1 e L_2 duas linguagens algébricas sobre o alfabeto Σ

- $L_1 \cup L_2$ 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.

.. e algoritmos

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

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

Os seguinte problemas não têm solução algorítmica:

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

Algumas demonstrações

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 $G = \{V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1; S \to S_2\}, S\}$ á algébrica e gera a linguagem $L_1 \cup L_2$. Para ficar convencido deste facto basta verificar que $\forall w \in (\Sigma_1 \cup \Sigma_2)^*, S \stackrel{*}{\Longrightarrow} w$ se e só se $(S_1 \stackrel{*}{\Longrightarrow} w) \vee (S_2 \stackrel{*}{\Longrightarrow} w)$. Como os conjuntos V_1 e V_2 são disjuntos então a disjunção equivale a aformar que $w \in \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$
- $L_1.L_2$ é algébrica. (considerar a gramática $\{V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S\})$
- ${L_1}^*$ é algébrica. (considerar a gramática $\{V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \to \epsilon,\ S \to S S_1\}, S\})$

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 $S \rightarrow S_6|S_7$

Plano

Kleene revisitado

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

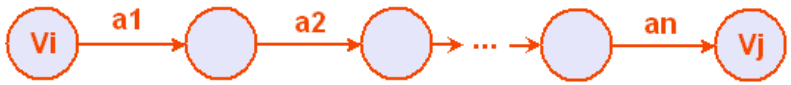
Gramática linear direita $\Longrightarrow$ Autómatos finitos

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

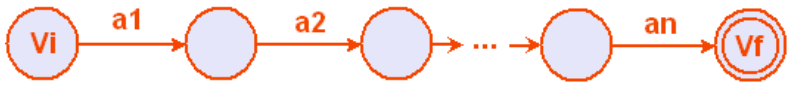
Seja $G = (N, \Sigma, S, P)$ uma gramática linear direita, então $L(G)$ é uma linguagem regular.

Gramática linear direita $\Longrightarrow$ 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 $N = \{V_0, V_1, \ldots\}$ 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 úico estado final V_f.
- Para cada produção $V_i = a_1 a_2 \ldots a_n V_j$, criar :



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



Gramática linear direita $\Longrightarrow$ Autómatos finitos

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

Autómatos finitos $\Longrightarrow$ 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 $\Longrightarrow$ Gramática linear direita

- Seja $M = \{Q = \{q_0, q_1, \ldots, q_n\}, \Sigma = \{a_1, \ldots, a_m\}, \delta, q_0, F\}$ 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.

Gramática linear direita $\Leftrightarrow$ Autómatos finitos

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....

Aquecimento...

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

Uma Regra de Substituição

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* $x1, x_2 \in (N \cup \Sigma)^*$, *e* $B \in N, \wedge B \neq A$. *Suponha igualmente que as produções com* B *no lhs não são recursivas, ou seja da forma* $B \rightarrow y_1 | y_2 | \ldots | y_n$ *com* $y_i \in (N - \{B\} \cup \Sigma)^*$.
Seja $G = \{N, \Sigma, S, P'\}$ *onde*

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

Então $L(G) = L(G')$

Uma Regra de Substituição

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 $S \overset{*}{\Longrightarrow}_G w$. Dois casos se apresentam:

1. A regra (*) não participa à derivação. Neste caso é trivial ver que $S \overset{*}{\Longrightarrow}_{G'} w$, 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

     $$S \overset{*}{\Longrightarrow}_G u_1 A u_2 \underbrace{\Longrightarrow_G u_1 x_1 B x_2 u_2 \Longrightarrow_G u_1 x_1 y_j x_2 u_2}_{(*) \text{ e } B \to y_j} \overset{*}{\Longrightarrow}_G w.$$

     Esta derivação pode ser substituída por $S \overset{*}{\Longrightarrow} u_1 A u_2 \Longrightarrow u_1 x_1 y_j x_2 u_2 \overset{*}{\Longrightarrow} w$ utilizando directamente a regra $A \to x_1 y_j x_2$ de G'. Done.

Uma Regra de Substituição

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

Remoção de regras inúteis

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

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

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

Em
$$\begin{array}{lcl} S & \rightarrow & aSb \mid bA \mid cC \\ A & \rightarrow & aA \mid \epsilon \\ B & \rightarrow & bA \mid BaS \\ C & \rightarrow & cC \end{array}$$

S e A são úteis, B e C são inúteis.

Remoção de regras inúteis

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 duma 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

Comecemos por ilustrar os princípios do algoritmo via um exemplo.

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

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 analise B não tem possibilidade de intervir. Logo B é inútil (apesar de ser produtor).
6. no final resta-nos $\begin{array}{llll} 1 & S & \rightarrow & aS \mid A \\ 2 & A & \rightarrow & a \end{array}$

Remoção de regras inúteis

- Mais formalmente. Um não terminal é inútil quando *não produz* **ou** quando é *inatingível*.
- Assim, como definir (e descriminar) 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 β são produtivos.
  - ou seja um não terminal é produtivo se é *lhs* duma regra produtiva.

Remoção de regras inúteis

- 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 ($A \to \alpha$, com $\alpha \in \Sigma^*$).
- As equações por considerar são: Para toda a regra $A \to \alpha$, esta produz na etapa $i+1$ se $\alpha \in \Sigma^*$ ou se $\alpha \in (N \cup \Sigma)^+$ onde todos os não terminais de α 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.

Remoção de regras inúteis

- Como definir (e descriminar) todos os terminais inatingíveis ?
- um não terminal A é inatingível quando
  - $A = S$
  - Caso contrário, quando existe uma produção $B \to \alpha$, 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 $B \to \beta$ 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.

Remoção de regras inúteis

Um exemplo

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

	etapa 0		*etapa* 1		*etapa* 2		*etapa* 3	
Regras	*Prod.*	*Ating.*	*Prod.*	*Ating.*	*Prod.*	*Ating.*	*Prod.*	*Ating.*
$S \rightarrow aS$		$\surd$		$\surd$	$\surd$	$\underline{\surd}$	$\underline{\surd}$	
$S \rightarrow A$		$\surd$	$\surd$	$\surd$	$\surd$	$\underline{\surd}$	$\underline{\surd}$	
$S \rightarrow C$		$\surd$		$\surd$		$\underline{\surd}$		
$A \rightarrow a$	$\surd$		$\surd$	$\surd$	$\surd$	$\underline{\surd}$	$\underline{\surd}$	
$B \rightarrow aa$	$\surd$		$\surd$		$\surd$		$\underline{\surd}$	
$C \rightarrow aCb$				$\surd$		$\underline{\surd}$		

Produtíveis: A,B,S
Atingíveis: S,A,C

Elementos de prova de correcção

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

Remoção de regras ϵ

Definition

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

Remoção de regras ϵ

Theorem

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

Demonstração: Construtiva.
Primeiro é preciso determinar quais são os não terminais anuláveis.

Remoção de regras ϵ

- Para obter a lista dos não-terminais anuláveis procede-se por procura de ponto fixo.
- Para começar, sabemos que uma regra da forma $A \to \beta$ tal que β contenha um terminal nunca será anulável
- Listamos todas as regras numa tabela. Na etapa zero marcamos as produções ϵ.
- As equações por considerar são: Para toda a produção $A \to A_1 A_2 \cdots A_n$ de P (com $A_1, A_2, \ldots, A_n \in N$), $A \to A_1 A_2 \cdots A_n$ é anulável na etapa $i+1$ se todos os $A_1 A_2 \cdots A_n$ 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.

Remoção de regras ϵ

- Sabendo agora quais são os não terminais anuláveis, podemos construir o novo conjunto de produção da forma seguinte:
- Considerar todas as regras da forma $A \rightarrow x_1 x_2 \ldots x_m$ (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 $X = \{x_{i_1} x_{i_2} \ldots x_{i_p}\}$ o subconjunto de $\{x_1 x_2 \ldots x_m\}$ 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 um nova regra de produção, da forma seguinte: para todo o $\{x_{j_1} x_{j_2} \ldots x_{j_q}\} \in \mathcal{P}(X)$, considerar a regra inicial $A \rightarrow x_1 x_2 \ldots x_m$ e retirar-lhe todos os não terminais anuláveis excepto $\{x_{j_1} x_{j_2} \ldots x_{j_q}\}$. A regra resultante é acrescentada a $\mathcal{C}$.
- existe no entanto uma excepção a este algoritmo: se todos os x_is são anuláveis, não se considera a regra $A \rightarrow \epsilon$ no conjunto $\mathcal{C}$.

Remoção de regras ϵ

- 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 ϵ 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.

Remoção de regras ϵ

Um exemplo

$$\begin{array}{lcl} S & \rightarrow & ABaC \\ A & \rightarrow & BC \\ B & \rightarrow & b \mid \epsilon \\ C & \rightarrow & D \mid \epsilon \\ D & \rightarrow & d \end{array}$$

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

$$\begin{array}{lcl} S & \rightarrow & ABaC \mid BaC \mid AaC \mid ABa \mid aC \mid Aa \mid Ba \mid a \\ A & \rightarrow & BC \mid B \mid C \\ B & \rightarrow & b \\ C & \rightarrow & D \\ D & \rightarrow & d \end{array}$$

Remoção de regras unitárias

Definition

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

Theorem

Seja G uma gramatica livre de contexto sem produções ϵ. Então existe uma gramática G' equivalente a G tal que G' não contenha produções unitárias.

Demonstração: Mais uma vez, construtiva.

Remoção de regras unitárias

- Remover directamente todas as regras da forma $A \to A$
- Para cada não terminal A de N calcular (com base numa procura de ponto fixo) todos os $B \in N$ tais que $A \stackrel{*}{\Longrightarrow} B$
- 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 $A \stackrel{*}{\Longrightarrow} B$ e para cada um deles juntar a P' $A \to y_1 \mid y_2 \mid y_3 \mid \ldots \mid y_n$ para cada regra $B \to y_1 \mid y_2 \mid y_3 \mid \ldots \mid y_n$ 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 ϵ (excepto $S \to \epsilon$), produções unidade e inúteis, é preciso aplicar os algoritmos de remoção de regras **nesta ordem** (exercício: porquê?)

1. Remover transições ϵ
2. Remover produções unidade
3. Remover produções inúteis

Forma Normal de Chomsky

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.

Forma Normal de Chomsky

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

$$\begin{cases} A & \to & C_1^1 \dots C_{p_1}^1 B_1 C_1^2 \dots C_{p_2}^2 B_2 \ \dots \ C_1^q \dots C_{p_q}^q B_q C_1^{q+1} \dots C_{p+1}^{q+1} \\ C_1^1 & \to & x_1^1 \\ \vdots & & \\ C_{p_1}^1 & \to & x_{p_1}^1 \\ C_1^2 & \to & x_1^2 \\ \vdots & & \\ C_{p+1}^{q+1} & \to & x_{p+1}^{q+1} \end{cases}$$

Forma Normal de Chomsky

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

$$\begin{cases} A & \rightarrow & B_1 B_2' \\ B'2 & \rightarrow & B_2 B_3' \\ & \vdots & \\ B_{n-1}' & \rightarrow & B_{n-1} B_n \end{cases}$$

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

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.
- $w = a_1 a_2 a_3 \cdots a_n$
- $w_{ij} = a_i \cdots a_j$
- $V_{ij} = \{A \in V \mid A \stackrel{*}{\Longrightarrow} w_{ij}\}$
- 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 $(A \to a_i) \in P$
- Temos assim forma de calcular todos os V_{ii} 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 $A \overset{*}{\Longrightarrow} w_{ij}$ se e só se existem um inteiro k tal que $i \leq k < j$ e uma produção $A \to BC$ tal que $B \overset{*}{\Longrightarrow} w_{ik}$ e $C \overset{*}{\Longrightarrow} w_{kj}$.
- Por outras palavras

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

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

Exemplo

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

com G contendo as produções:

$$\begin{array}{lcl} S & \rightarrow & UV \\ U & \rightarrow & VV \mid a \\ V & \rightarrow & UV \mid b \end{array}$$

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

SIM.