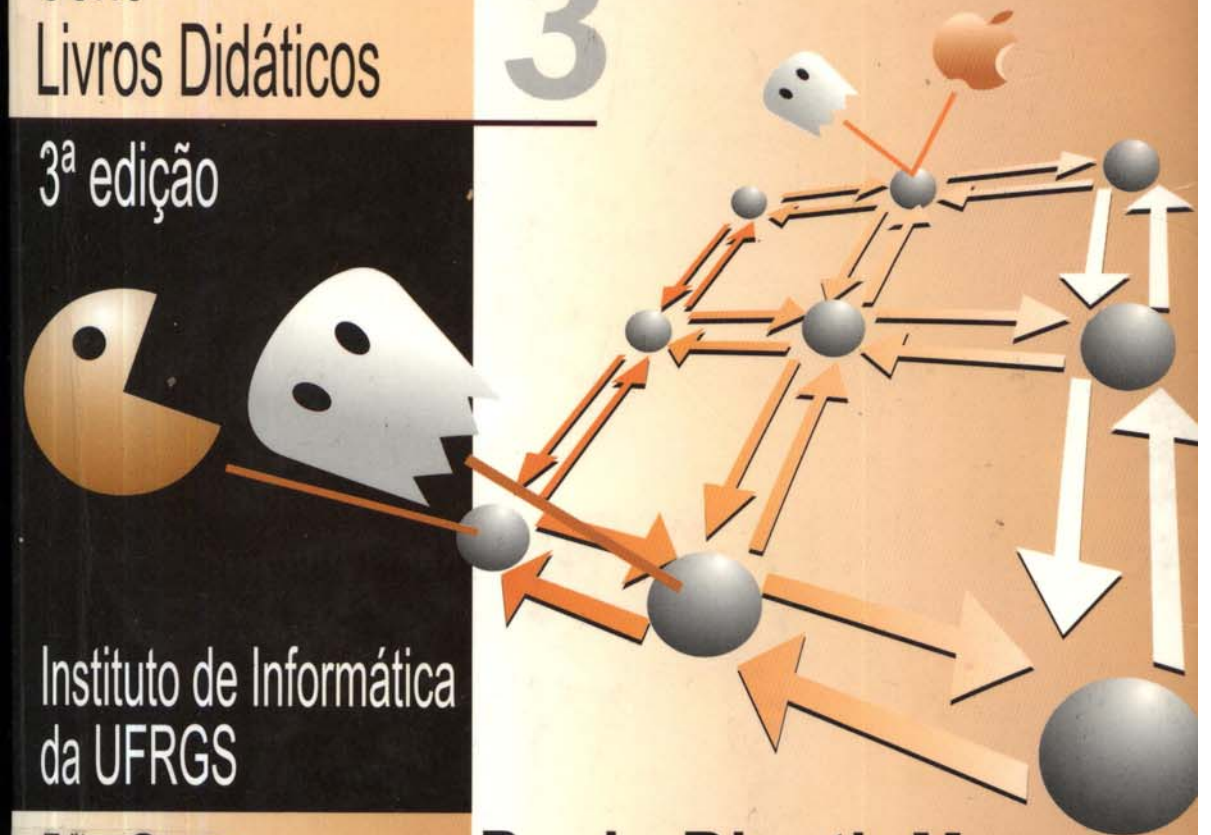
Linguagens Formais e Autômatos

Série Livros Didáticos

Número 3

3ª edição



Instituto de Informática da UFRGS

Editora Sagra Luzzatto

Paulo Blauth Menezes

Linguagens Formais e Autômatos

Informática UFRGS

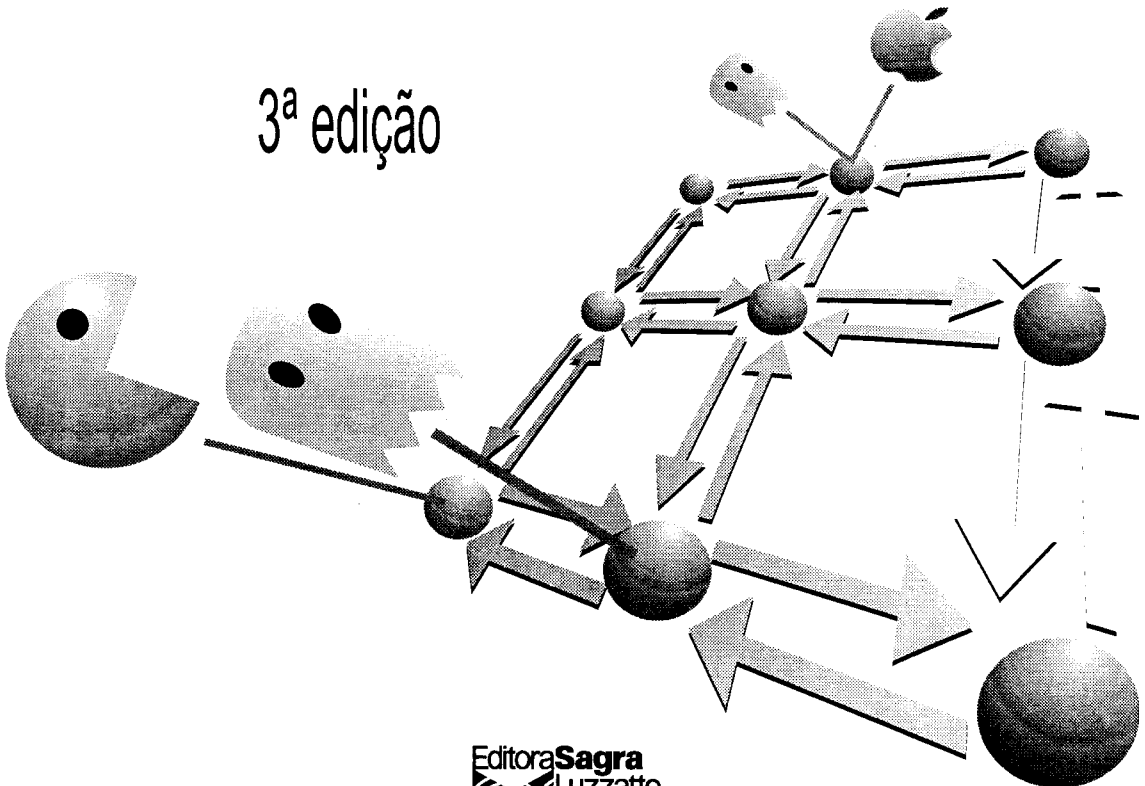
Endereço

UFRGS – Instituto de Informática
Av. Bento Gonçalves, 9500 Bloco IV Bairro Agronomia
Caixa Postal 15064 91501-970 Porto Alegre, RS
Fone 00 55 (051) 316 6165 Fax 00 55 (051) 319 1576
e-mail: informat@inf.ufrgs.br
http://www.inf.ufrgs.br

Paulo Blauth Menezes

Linguagens Formais e Autômatos

3ª edição



Editora Sagra Luzzatto

1ª edição, 1997 • 2ª edição, 1998 • 3ª edição, 2000

EditoraSagraLuzzatto
Rua João Alfredo, 448 – Cidade Baixa
90050-230 – Porto Alegre, RS
Ligue grátis 0800-51-2269
Fone (51) 227-5222 Fax 227-4438
internet: www.sagra-luzzatto.com.br
atendimento@sagra-luzzatto.com.br

Editor: Darcy Loss Luzzatto
Supervisão Editorial: Elisa Schein Wenzel Luzzatto
Ilustrações: Maria Lúcia Menezes
Capa: Carlos Alberto Gravina

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Menezes, Paulo Fernando Blauth
Linguagens formais e autômatos / Paulo Fernando Blauth Menezes. – Porto Alegre : Instituto de Informática da UFRGS : Editora Sagra Luzzatto, 2000.
(Série livros didáticos, número 3)

Bibliografia.
85-241-0554-2

1. Linguagens formais. 2. Autômatos. 3. Gramáticas. 4. Expressão regulares. 5. Algoritmos de reconhecimento. 6. Teoria da computação. I. Título, II. Série.

Prefácio da Série

A série ***Livros Didáticos***, do Instituto de Informática da Universidade Federal do Rio Grande do Sul, é inspirada na idéia de desenvolver material didático para disciplinas ministradas no Bacharelado em Ciência da Computação. Esse material é resultante da experiência dos professores do Instituto de Informática no ensino e na pesquisa.

Em seus primeiros volumes, a série era voltada *para Matemática da Computação e Processamento Paralelo.* Foram publicados três títulos: *Fundamentos da Matemática Intervalar, Programando em Pascal XSC.* Esses dois primeiros títulos foram resultado de pesquisas desenvolvidas dentro do Projeto ArInPar - Aritmética Intervalar Paralela, financiado pelo ProTeM -CC CNPq (Fase II). O terceiro, *Linguagens Formais e Autômatos*, foi o primeiro da série que se voltou ao objetivo de suprir livros-texto para as disciplinas básicas dos cursos de Bacharelado em Ciência da Computação (ou Informática). O conteúdo desses livros é baseado no programa das disciplinas do Bacharelado em Ciência da Computação da UFRGS, sendo adotado, também, por diversas Universidades do Rio Grande do Sul e de outros estados.

O sucesso da experiência com esses livros, bem como a responsabilidade que cabe ao Instituto de Informática na formação de professores e pesquisadores em Computação, conduziu à ampliação da abrangência e à institucionalização da série, que passa a ser de ***Livros Didáticos*** *do Instituto de Informática.*

Neste novo enfoque, foi publicado a segunda edição dos livros *Linguagens Formais e Autômatos* e *Projeto de Banco de Dados*, de autoria, respectivamente, dos Professores Paulo Fernando Blauth Menezes e Carlos Alberto Heuser, e a primeira edição do livro *Teoria da Computação: Máquinas Universais e Computabilidade*, dos autores Tiarajú Asmuz Diverio e Paulo Fernando Blauth Menezes. Esses livros estão tendo uma ampla aceitação pela comunidade, fato comprovado pelas indicações como livros-texto em várias universidades do País, o que, em 2000, levou à publicação da terceira edição do livro *Projeto de Banco de Dados* e, agora, do livro *Linguagens Formais e Autômatos.*

Outros títulos, também compreendendo conteúdos de disciplinas de Bacharelados em Ciência da Computação (ou Informática) e de Bacharelados em Engenharia da Computação, encontram-se em preparação. Entre eles, destacamos:

- Arquitetura de Computadores Pessoais – *Raul Fernando Weber*
- Tabelas: Organização e Pesquisa - *Clesio Saraiva dos Santos e Paulo Alberto de Azeredo*
- Fundamentos da Arquitetura de Computadores - *Raul Fernando Weber*
- Teoria das Categorias e Ciência da Computação - *Paulo Blauth Menezes e Edward Hermann Haeusler*
- Técnicas Digitais para Computação - *Flávio Rech Wagner*

Todos os livros têm em comum a preocupação em manter nível compatível com a elevada qualidade do ensino e da pesquisa desenvolvidos no âmbito do Instituto de Informática da UFRGS.

Comissão Editorial da Série Livros Didáticos

Instituto de Informática da UFRGS

Março de 2000.

Para Maria Fernanda,
Maria Lúcia e
Maria Luiza

Agradecimentos

À Maria Lúcia Menezes pelas bem-humoradas ilustrações de autômatos usadas ao longo de todo o texto.

Aos alunos de mestrado Júlio Pereira Machado e Carlos Tadeu Queiroz de Morais e aos bolsistas de iniciação científica Leonardo Penczek, Gustavo Link Federizzi, Karina Girardi Roggia e Guilherme de Campos Magalhães pelas diversas contribuições para a viabilização de diversos trabalhos afins.

Ao colega Prof. Tiarajú Diverio pelo apoio e incentivo recebidos para a viabilização deste e de outros trabalhos didáticos.

Prefácio do Autor

Linguagens Formais e Autômatos objetiva apresentar os conceitos básicos de Linguagens Formais. É baseado em experiências letivas no Curso de Bacharelado em Ciência da Computação da UFRGS. É destinado, principalmente, como um primeiro curso de Linguagens Formais e Autômatos, sendo auto-contido e podendo ser adotado como bibliografia básica. Possui um texto simples e com diversas ilustrações, exemplos detalhados e exercícios em níveis crescentes de raciocínio.

A primeira edição tinha como principal objetivo suprir a necessidade de um livro-texto didático voltado para o Curso de Bacharelados em Ciência da Computação na UFRGS. Entretanto, foi uma grata surpresa verificar que o livro foi adotado em diversas instituições em todo o Brasil, esgotando rapidamente, fazendo com que uma segunda edição fosse antecipada em cerca de dois anos em relação ao previsto. A segunda edição seguiu a mesma estrutura da primeira e aprimorou pequenos, mas importantes detalhes. A aceitação foi muito boa e a edição também esgotou-se rapidamente.

Com o objetivo de manter o custo do livro acessível (fator importante para muitos estudantes), optou-se, para esta terceira edição, fazer somente pequenas revisões, mantendo, inclusive, as mesmas numerações (páginas, seções, etc.) da segunda edição. Em contrapartida, foi desenvolvido e testado um sistema de apoio ao ensino, via Internet, denominado de *Hyper-Automaton*, que inclui uma série de facilidades para alunos, professores e interessados em geral. Um fato curioso é que este sistema é baseado em Autômatos Finitos com Saída, um dos temas desenvolvidos pelo livro. Uma importante conseqüência é que as páginas HTML não possuem *links* (estes estão codificados na função programa do autômato), permitindo um reuso imediato do material instrucional. Gradativamente, a partir do segundo semestre de 2000, cursos, exercícios, material de apoio, revisões (edições anteriores e atual), etc., serão disponibilizados no seguinte endereço:

http://teia.inf.ufrgs.br/

Agradeço os diversos comentários, contribuições e retornos recebidos de professores, alunos e interessados e destaco o meu interesse em continuar mantendo contato sobre este e outros assuntos correlatos.

Porto Alegre, Outubro de 1998

Prof. Dr. Paulo Blauth Menezes

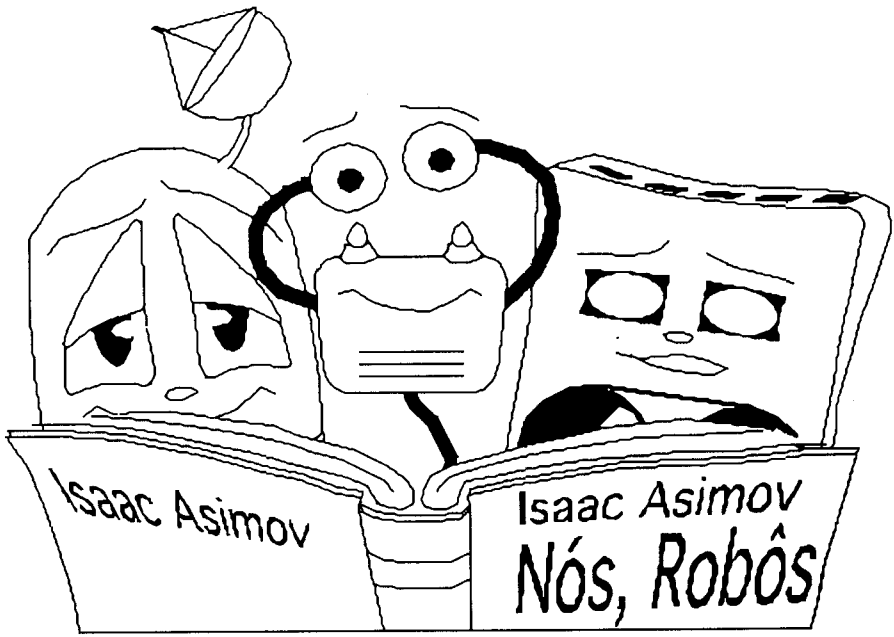
blauth@inf.ufrgs.br
www.inf.ufrgs.br/~blauth

Índice

Linguagens Formais e Autômatos

Teoria das Linguagens Formais foi originariamente desenvolvida na década de 1950 com o objetivo de desenvolver teorias relacionadas com as linguagens naturais. Entretanto, logo foi verificado que esta teoria era importante para o estudo de linguagens artificiais e, em especial, para as linguagens originárias na Ciência da Computação. Desde então, o estudo das Linguagens Formais desenvolveu-se significativamente e com diversos enfoques, com destaque para aplicações em análise léxica e sintática de linguagens de programação, modelos de sistemas biológicos, desenho de hardware e relacionamentos com linguagens naturais. Recentemente, inclui-se a ênfase no tratamento de linguagens não-lineares, como planares, espaciais e n-dimensionais.

Linguagens Formais e Autômatos objetiva apresentar os conceitos básicos de Linguagens Formais. É baseado em experiências letivas no Curso de Bacharelado em Ciência da Computação da UFRGS. É destinado, principalmente, como um primeiro curso de Linguagens Formais, sendo auto-contido e podendo ser adotado como bibliografia básica. Possui um texto simples e com diversas ilustrações, exemplos detalhados e exercícios em níveis crescentes de raciocínio.



1 Introdução e Conceitos Básicos

1.1 Introdução

Teoria das Linguagens Formais foi originariamente desenvolvida na década de 1950 com o objetivo de desenvolver teorias relacionadas com as linguagens naturais. Entretanto, logo foi verificado que esta teoria era importante para o estudo de linguagens artificiais e, em especial, para as linguagens originárias na Ciência da Computação. Desde então, o estudo das Linguagens Formais desenvolveu-se significativamente e com diversos enfoques, com destaque para aplicações em Análise Léxica e Sintática de linguagens de programação, modelos de sistemas biológicos, desenhos de circuitos e relacionamentos com linguagens naturais. Recentemente, incluiu-se a ênfase no tratamento de Linguagens Não-Lineares, como Planares, Espaciais e n-Dimensionais.

Sintaxe e Semântica

Linguagens Formais preocupa-se com os problemas sintáticos das linguagens. Assim, inicialmente, é importante introduzir os conceitos de sintaxe e semântica de linguagens.

Historicamente, no estudo do entendimento das linguagens de programação, o problema sintático foi reconhecido antes do problema semântico e foi o primeiro a receber um tratamento adequado. Adicionalmente, os problemas sintáticos são de tratamento mais simples que os semânticos. Como conseqüência, foi dada uma grande ênfase à sintaxe, ao ponto de levar à idéia de que as questões das linguagens de programação resumiam-se às questões da sintaxe. Atualmente, a teoria da sintaxe possui construções matemáticas bem definidas e universalmente reconhecidas como, por exemplo, as *Gramáticas de Chomsky*.

Uma linguagem de programação (bem como qualquer modelo matemático) pode ser vista de duas formas:

- como uma entidade livre, ou seja, sem qualquer significado associado;
- como uma entidade juntamente com uma interpretação do seu significado.

A *sintaxe* trata das propriedades livres da linguagem como, por exemplo, a verificação gramatical de programas. A *semântica* objetiva dar uma interpretação para a linguagem como, por exemplo, um significado ou valor para um determinado programa. Conseqüentemente, a sintaxe basicamente manipula símbolos sem considerar os seus correspondentes significados. Note-se que, para resolver qualquer problema real, é necessário dar uma interpretação semântica aos símbolos como, por exemplo, "estes símbolos representam os inteiros".

Sintaticamente falando, não existe uma noção de programa "errado": neste caso, simplesmente não é um programa. Por outro lado, um programa sintaticamente válido ("correto"), pode não ser o programa que o programador esperava escrever. Assim, a questão de considerar um programa "correto" ou "errado" deve considerar se o mesmo modela adequadamente o comportamento desejado.

Nem sempre os limites entre a sintaxe e a semântica são claros. Um exemplo é a ocorrência de um nome em um programa o qual pode ser tratado de forma igualmente fácil como um problema sintático ou semântico. Entretanto, a distinção entre sintaxe e semântica em linguagens artificiais é, em geral, óbvia para a maioria dos problemas relevantes.

Abordagem

A abordagem desta publicação é centrada no tratamento sintático de linguagens lineares abstratas com fácil associação às linguagens típicas da Ciência da Computação. Os formalismos usados podem ser classificados nos seguintes tipos:

a) *Operacional*. Define-se um autômato ou uma máquina abstrata, baseada em estados, em instruções primitivas e na especificação de como cada instrução modifica cada estado. Uma máquina abstrata deve ser suficientemente simples para não permitir dúvidas sobre a execução de seu código. Também é dito um formalismo *Reconhecedor*, no sentido em que permite a análise de uma dada entrada para verificar se é "reconhecida" pela máquina. As principais máquinas definidas nesta publicação são Autômato Finito, Autômato com Pilha e Máquina de Turing;

b) *Axiomático*. Associam-se regras às componentes da linguagem. As regras permitem afirmar o que será verdadeiro após a ocorrência de cada cláusula considerando o que era verdadeiro antes da ocorrência. A abordagem axiomática que segue é sobre Gramáticas (Regulares, Livres do Contexto, Sensíveis ao Contexto e Irrestritas). Uma gramática também é dita um formalismo *Gerador* no sentido em que permite verificar se um determinado elemento da linguagem é "gerado";

c) *Denotacional*. Também é denominado formalismo *Funcional*. Define-se uma função que caracteriza o conjunto de palavras admissíveis na linguagem. Em geral, trata-se de uma função construída a partir de funções elementares de forma composicional (horizontalmente) no sentido em que a linguagem denotada pela função pode ser determinada em termos de suas funções componentes. Nesta publicação, a abordagem denotacional é restrita às Expressões Regulares. Como, a partir de uma expressão regular, é simples inferir ("gerar") as palavras da linguagem denotada, freqüentemente também é denominado, de forma não muito precisa, como um formalismo *Gerador*.

Organização dos Capítulos e Carga Horária Recomendada

A estruturação e uma breve introdução dos capítulos desta publicação é, resumidamente, a seguinte:

a) *Capítulo 1*. Os demais tópicos deste capítulo introduzem conceitos básicos necessários para o que segue. Note-se que a parte referente a Conjuntos, Relações, Funções, Lógica e Técnicas de Demonstração objetiva realizar uma revisão e normalização de notações e, portanto, considera algum

conhecimento prévio e não esgota o assunto. Caso o leitor domine adequadamente estes assuntos, recomenda-se passar diretamente para a secção 1.5 - Alfabetos, Palavras, Linguagens e Gramáticas;

b) *Capítulo 2.* As Linguagens Regulares e as noções de Autômato Finito e Expressão Regular são originárias de estudos biológicos de redes de neurônios e circuitos de chaveamentos. Mais recentemente, são usadas para o desenvolvimento de Analisadores Léxicos (parte de um compilador que identifica e codifica as unidades básicas de uma linguagens como variáveis, números, etc.), editores de textos, sistemas de pesquisa e atualização em arquivos (em geral, do tipo busca e substituição de informações não complexas), linguagens simples de comunicação homem-máquina (como interface do sistema operacional) e máquina-máquina (como protocolos de comunicação). Note-se que a análise léxica pode ser considerada como um caso particular e simples de análise sintática;

c) *Capítulo 3.* As Linguagens Livres do Contexto e as correspondentes noções de Gramática Livre do Contexto e Autômato com Pilha são usadas principalmente para o desenvolvimento de Analisadores Sintáticos, uma importante parte de um compilador. Historicamente, o desenvolvimento de analisadores sintáticos era um problema complexo, de difícil depuração e com eficiência relativamente baixa. Hoje, considerando o conhecimento já adquirido relativo às Linguagens Livre do Contexto, o desenvolvimento de um Analisador Sintático é simples (assim como a sua depuração) e somente uma pequena percentagem do tempo de processamento de um compilador é gasto em tal atividade;

d) *Capítulo 4.* As Linguagens Enumeráveis Recursivamente e Sensíveis ao Contexto e as correspondentes noções de Máquina de Turing (e eventuais variações/restrições deste modelo) e as Gramáticas Irrestritas e Sensíveis ao Contexto permitem explorar os limites da capacidade de desenvolvimento de reconhecedores ou geradores de linguagens. Ou seja, estudam a solucionabilidade do problema da existência de algum reconhecedor ou gerador para determinada linguagem;

e) *Capítulo 5.* Conclui os capítulos anteriores, classificando as diversas classes de linguagens em uma ordem hierárquica, denominada Hierarquia de Chomsky (ilustrada na Figura 1.1) e apresenta conclusões gerais e perspectivas futuras.

O trabalho que segue é baseado em experiências letivas no Curso de Bacharelado em Ciência da Computação da Universidade Federal do Rio Grande do Sul. É destinado, principalmente, como um primeiro curso de Linguagens Formais, sendo auto-contido e podendo ser adotado como bibliografia básica. Possui um texto simples, exemplos detalhados e exercícios em níveis crescentes de raciocínio. Embora todos os conceitos

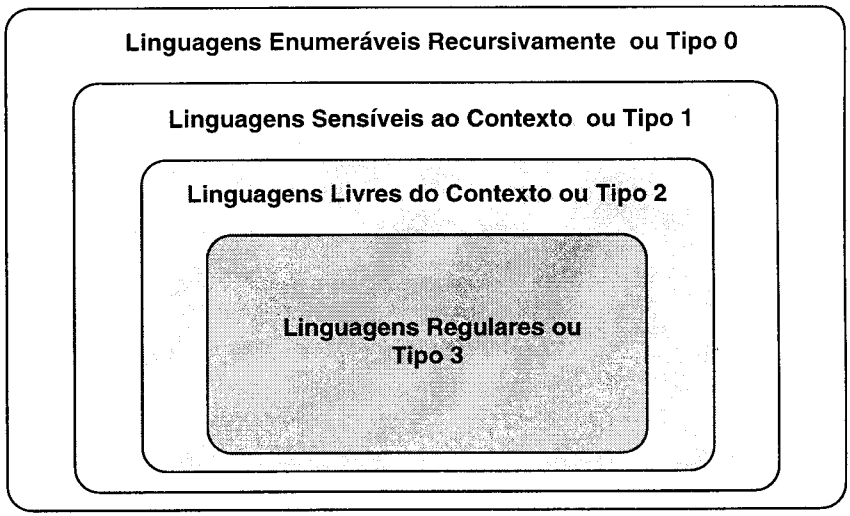


Figura 1.1 Hierarquia de Chomsky

necessários sejam introduzidos, recomenda-se como pré-requisitos conhecimentos básicos de lógica, teoria dos conjuntos e algoritmos. No caso de desenvolvimento de um curso, a carga horária recomendada varia de 45 a 90 horas.

1.2 Conjuntos, Relações e Funções

No texto que segue é suposto que o leitor está familiarizado com os conceitos básicos relativos à *Teoria dos Conjuntos*.

Conjuntos e Operações sobre Conjuntos

Definição 1.1 Conjunto.

Um *Conjunto* é uma coleção de zero ou mais objetos distintos, denominados *Elementos* do conjunto. ❑

Um *elemento* é uma entidade básica a qual não é definida formalmente. Relativamente ao relacionamento entre elementos e conjuntos, tem-se que:

a) Se um elemento a *pertence* a um conjunto A denota-se por $a \in A$; caso contrário, $a \notin A$

b) Se todos os elementos de um conjunto A também são elementos de um conjunto B, então afirma-se que A está *contido* em B ou que A é *subconjunto* de B e denota-se por $A \subseteq B$ (ou ainda B *contém* A e $B \supseteq A$). Adicionalmente, se existe $b \in B$ tal que $b \notin A$, então afirma-se que A está *contido propriamente* em B ou que A é *subconjunto próprio* de B e denota-se por $A \subset B$ (ou ainda B *contém propriamente* A e $B \supset A$)

c) Os conjuntos A e B são *iguais* se, e somente se, possuem os mesmos elementos, ou seja, $A = B$ se, e somente se, $A \subseteq B$ e $B \subseteq A$

Um conjunto pode possuir um número finito ou infinito de elementos. Os conjuntos finitos podem ser denotados por *extensão*, listando todos os seus elementos entre chaves e em qualquer ordem como, por exemplo:

$$\{a, b, c\}$$

O conjunto sem elementos (ou seja, com zero elementos) é denominado *conjunto vazio* e é denotado por $\{\}$ ou $\emptyset$. Conjuntos (finitos ou infinitos) também podem ser denotados por *compreensão* na forma:

$$\{a \mid a \in A \text{ e } p(a)\} \quad \text{ou} \quad \{a \in A \mid p(a)\}$$

a qual é interpretada como:

> *"o conjunto de todos os elementos a pertencentes ao conjunto A tal que p(a) é verdadeiro".*

Quando é claro que $a \in A$, pode-se denotar simplesmente na forma:

$$\{a \mid p(a)\}$$

EXEMPLO 1 Conjuntos, Elementos.

a) $a \in \{b, a\}$ e $c \notin \{b, a\}$

b) $\{a, b\} = \{b, a\}$, $\{a, b\} \subseteq \{b, a\}$ e $\{a, b\} \subset \{a, b, c\}$

c) Os seguintes conjuntos são infinitos:
   - $\mathbb{N}$ *Conjunto dos Números Naturais*;
   - $\mathbb{Z}$ *Conjunto dos Números Inteiros*;
   - $\mathbb{Q}$ *Conjunto dos Números Racionais*;
   - $\mathbb{I}$ *Conjunto dos Números Irracionais*;
   - $\mathbb{R}$ *Conjunto dos Números Reais.*

d) $\{1, 2, 3\} = \{x \in \mathbb{N} \mid x > 0 \text{ e } x < 4\}$ e $\mathbb{N} = \{x \in \mathbb{Z} \mid x \geq 0\}$

e) O conjunto dos números pares pode ser denotado por compreensão como segue:

$$\{y \mid y = 2x \text{ e } x \in \mathbb{N}\}$$ ❑

As principais operações sobre conjuntos são as seguintes.

Definição 1.2 União, Intersecção, Diferença, Complemento, Conjunto das Partes, Produto Cartesiano.

Sejam A e B conjuntos. Então:

a) *União.*
$A \cup B = \{x \mid x \in A \text{ ou } x \in B\}$

b) *Intersecção.*
$A \cap B = \{x \mid x \in A \text{ e } x \in B\}$

c) *Diferença.*
$A - B = \{x \mid x \in A \text{ e } x \notin B\}$

d) *Complemento.* A operação de complemento é definida em relação a um conjunto fixo $\mathbb{U}$ denominado *conjunto universo.*
$A' = \{x \mid x \in \mathbb{U} \text{ e } x \notin A\}$

e) *Conjunto das Partes.*
$2^A = \{S \mid S \subseteq A\}$

f) *Produto Cartesiano.*
$A \times B = \{(a, b) \mid a \in A \text{ e } b \in B\}$ ❑

É usual denotar um produto cartesiano de um conjunto com ele mesmo como um expoente. Por exemplo:

$$A \times A = A^2$$

Um elemento de um produto cartesiano denotado na forma (a, b) é denominado *par ordenado* e não deve ser confundido com o conjunto {a, b}: em um par ordenado, a ordem é importante, pois são distinguidas as duas componentes. O conceito de par ordenado pode ser generalizado para n-upla ordenada, ou seja, com $n > 0$ componentes.

EXEMPLO 2 Operações sobre Conjuntos.

Suponha o universo $\mathbb{N}$ e sejam $A = \{0, 1, 2\}$ e $B = \{2, 3\}$. Então:

a) $A \cup B = \{0, 1, 2, 3\}$

b) $A \cap B = \{2\}$

c) $A - B = \{0, 1\}$

d) $A' = \{x \in \mathbb{N} \mid x > 2\}$

e) $2^B = \{\varnothing, \{2\}, \{3\}, \{2, 3\}\}$

f) $A \times B = \{(0, 2), (0, 3), (1, 2), (1, 3), (2, 2), (2, 3)\}$ ❑

As seguintes propriedades das operações sobre conjuntos podem ser facilmente verificadas (suponha o universo $\mathbb{U}$ e os conjuntos A, B e C):

a) *Idempotência.*

$A \cup A = A$

$A \cap A = A$

b) *Comutatividade.*

$A \cup B = B \cup A$

$A \cap B = B \cap A$

c) *Associatividade.*

$A \cup (B \cup C) = (A \cup B) \cup C$

$A \cap (B \cap C) = (A \cap B) \cap C$

d) *Distributividade.*

$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

e) *Duplo Complemento.*

$(A')' = A$

f) *Morgan.*

$(A \cup B)' = A' \cap B'$

$(A \cap B)' = A' \cup B'$

g) *Universo e Vazio.*

$A \cup A' = \mathbb{U}$

$A \cap A' = \varnothing$

Relações

Definição 1.3 Relação.

Uma *Relação* (Binária) é um subconjunto de um produto cartesiano. ❑

Suponha os conjuntos A e B e a relação $R \subseteq A \times B$. Então A e B são denominados *domínio* e *contra-domínio* (ou *codomínio*) de R, respectivamente. Um elemento $(a, b) \in R$ é usualmente denotado por aRb. Uma relação $R \subseteq A \times A$ (onde o domínio e o contra-domínio coincidem) é dita uma relação em A e, neste caso, é freqüentemente denotada por (A, R).

Definição 1.4 Relação Reflexiva, Simétrica, Antissimétrica, Transitiva.

Sejam A um conjunto e R uma relação em A. Então R é uma:

a) *Relação Reflexiva*, se, para todo $a \in A$, aRa

b) *Relação Simétrica*, se aRb, então bRa

c) *Relação Antissimétrica*, se aRb e bRa, então $a = b$

d) *Relação Transitiva*, se aRb e bRc, então aRc ❑

Note-se que uma relação pode não ser simétrica nem antissimétrica, ou seja, não são noções complementares. Adicionalmente, uma relação pode ser simultaneamente simétrica e antissimétrica.

EXEMPLO 3 Propriedades de Relações.

Considere um conjunto não vazio A. Então:

a) As relações $(\mathbb{N}, \leq)$ e $(2^A, \subseteq)$ são reflexivas, antissimétricas e transitivas;

b) As relações $(\mathbb{Z}, <)$ e $(2^A, \subset)$ são transitivas;

c) A relação $(\mathbb{Q}, =)$ é reflexiva, simétrica, antissimétrica e transitiva;

d) A relação $\{(1, 2), (2, 1), (2, 3)\}$ *não* é reflexiva, simétrica, antissimétrica e nem transitiva. ❑

Definição 1.5 Relação de Ordem, Ordem Parcial, Ordem Total.

Sejam A um conjunto e R uma relação em A. Então R é uma:

a) *Relação de Ordem*, se é transitiva;

b) *Relação de Ordem Parcial*, se é reflexiva, antissimétrica e transitiva;

c) *Relação de Ordem Total*, se é uma relação de ordem parcial e, para todo a, $b \in A$, ou aRb ou bRa. ❑

EXEMPLO 4 Relação de Ordem, Ordem Parcial, Ordem Total.

Considere um conjunto não vazio A. Então:

a) As relações $(\mathbb{N}, \leq)$, $(2^A, \subseteq)$, $(\mathbb{Z}, <)$, $(2^A, \subset)$ e $(\mathbb{Q}, =)$ são de ordem;

b) As relações $(\mathbb{N}, \leq)$, $(2^A, \subseteq)$ e $(\mathbb{Q}, =)$ são de ordem parcial;

c) A relação $(\mathbb{N}, \leq)$ é de ordem total. ❑

Definição 1.6 Relação de Equivalência.

Sejam A um conjunto e R uma relação em A. Então R é uma *Relação de Equivalência* se for reflexiva, simétrica e transitiva. ❑

Um importante resultado é que cada relação de equivalência induz um particionamento do conjunto (em que a relação é definida) em subconjuntos disjuntos e não vazios denominados *classes de equivalência*.

EXEMPLO 5 Relação de Equivalência.

Considere a relação $R = \{(a, b) \in \mathbb{N}^2 \mid a \text{ MOD } 2 = b \text{ MOD } 2\}$ onde MOD é a operação que resulta no resto da divisão inteira. É fácil verificar que R é uma relação de equivalência. Portanto, R induz um particionamento de $\mathbb{N}$ em dois subconjuntos: pares (resto zero) e ímpares (resto um). ❑

Freqüentemente é desejável estender uma relação de forma a satisfazer determinado conjunto de propriedades.

Definição 1.7 Fecho de uma Relação.

Sejam R uma relação e P um conjunto de propriedades. Então, o *Fecho de* R *em Relação ao* P, denotado por FECHO-P(R), é a menor relação que contém R e que satisfaz às propriedades em P. ❑

Dois tipos de fecho de uma relação são especialmente importantes no trabalho que segue.

Definição 1.8 Fecho Transitivo, Fecho Transitivo e Reflexivo.

Sejam R uma relação em A. Então:

a) O fecho de R em relação ao conjunto de propriedades {transitiva}, denominado *Fecho Transitivo* de R e denotado por R^+, é definido como segue:
   a.1) Se $(a, b) \in R$, então $(a, b) \in R^+$;
   a.2) Se $(a, b) \in R^+$ e $(b, c) \in R^+$, então $(a, c) \in R^+$;
   a.3) Os únicos elementos de R^+ são os construídos como acima;

b) O fecho de R em relação ao conjunto de propriedades {transitiva, reflexiva}, denominado *Fecho Transitivo e Reflexivo* de R e denotado por R^*, é tal que:

$$R^* = R^+ \cup \{(a, a) \mid a \in A\}$$ ❑

EXEMPLO 6 Fecho de um Grafo visto como uma Relação.

Um grafo (direto) pode ser definido como uma relação A (de arestas) em um conjunto V (de vértices). Assim:

$$A = \{(1, 2), (2, 3), (3, 4), (1, 5)\}$$

é um grafo em $V = \{1, 2, 3, 4, 5\}$ como ilustrado na Figura 1.2 (esquerda) onde o par ordenado que define uma aresta é representado por uma seta na qual as circunferências de origem e de destino representam a primeira e a segunda componente do par, respectivamente. O fecho transitivo e reflexivo:

$$A^* = \{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4), (5, 5)\}$$

é ilustrado na Figura 1.2 (direita) onde as arestas adicionadas pelo fecho são representadas com um traço diferente. ❑

Funções

Definição 1.9 Função Parcial.

Uma *Função Parcial* é uma relação $f \subseteq A \times B$ tal que se $(a, b) \in f$ e $(a, c) \in f$, então $b = c$. ❑

Portanto, uma função parcial é uma relação onde cada elemento do domínio está relacionado com, no máximo, um elemento do contra-domínio.

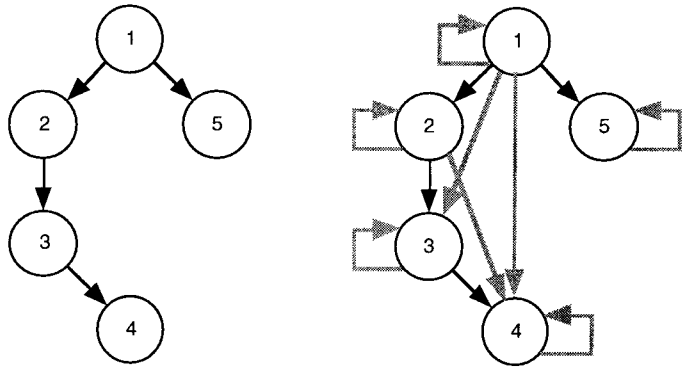


Figura 1.2 Grafo (esquerda) e o correspondente fecho transitivo e reflexivo (direita)

Uma função parcial $f \subseteq A \times B$ normalmente é denotada por $f: A \to B$. Adicionalmente, $(a, b) \in f$ é usualmente denotado por $f(a) = b$. Se $f(a) = b$ então afirma-se que f está definida para a e que b é *imagem* de a. O conjunto:

$$\{ b \in B \mid \text{existe } a \in A \text{ tal que } f(a) = b \}$$

é denominado *conjunto imagem* de f e é denotado por f(A) ou Img(f).

Definição 1.10 Função, Aplicação.

Uma *Função (Total)* ou *Aplicação* é uma função parcial $f: A \to B$ onde para todo $a \in A$ existe $b \in B$ tal que $f(a) = b$. ❑

Portanto, uma função (total) é uma função parcial definida para todos os elementos do domínio.

EXEMPLO 7 Função, Função Parcial.

a) *Função Identidade*. Para um dado conjunto A, a função (total) $id_A: A \to A$ é tal que, para todo $a \in A$, $id_A(a) = a$;

b) *Adição nos naturais*. A operação $ad: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ tal que $ad(a, b) = a + b$ é uma função (total);

c) *Divisão nos reais*. A operação $div: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ tal que $div(x, y) = x/y$ é uma função parcial pois não é definida para (x, 0), qualquer que seja $x \in \mathbb{R}$. ❑

Definição 1.11 Composição de Funções.

Sejam $f: A \to B$ e $g: B \to C$ funções. A *Composição* de f e g é a função $g \circ f: A \to C$ tal que $(g \circ f)(a) = g(f(a))$. ❑

Portanto, a composição de duas funções $f: A \to B$ e $g: B \to C$ é uma função $g \circ f: A \to C$ onde $(g \circ f)(a) = g(f(a))$ é a aplicação da função f ao elemento a e, na seqüência, da função g à imagem f(a).

EXEMPLO 8 Composição de Funções.

A composição das funções ad: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ (Exemplo 7) e quadrado: $\mathbb{N} \rightarrow \mathbb{N}$ é a função quadrado ad: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ e, para $(3, 1) \in \mathbb{N} \times \mathbb{N}$, tem-se que:

$$(\text{quadrado} \circ \text{ad})(3, 1) = \text{quadrado}(\text{ad}(3, 1)) = \text{quadrado}(4) = 16$$

Ou seja, a composição das funções resulta no quadrado da adição. ❑

Definição 1.12 Função Injetora, Sobrejetora, Bijetora ou Isomorfismo.

Uma função $f: A \rightarrow B$ é dita:

a) *Injetora* se, para todo $b \in B$, existe no máximo um $a \in A$ tal que $f(a) = b$;

b) *Sobrejetora* se, para todo $b \in B$, existe pelo menos um $a \in A$ tal que $f(a) = b$;

c) *Bijetora* ou *Isomorfismo* se é injetora e sobrejetora. ❑

Portanto, uma função é:

a) Injetora se cada elemento do contra-domínio é imagem de, no máximo, um elemento do domínio;

b) Sobrejetora se todo elemento do contra-domínio é imagem de pelo menos um elemento do domínio;

c) Bijetora ou isomorfismo se todo elemento do contra-domínio é imagem de exatamente um elemento do domínio.

É comum usar os termos injeção, sobrejeção e bijeção ao invés de função injetora, função sobrejetora e função bijetora, respectivamente. Adicionalmente, se existe um isomorfismo (função bijetora) entre dois conjuntos, estes são ditos *conjuntos isomorfos*.

EXEMPLO 9 Função Injetora, Sobrejetora, Bijetora, Isomorfismo.

a) A função inclusão: $\mathbb{N} \rightarrow \mathbb{Z}$ tal que inclusão(a) = a é injetora;

b) A função módulo: $\mathbb{Z} \rightarrow \mathbb{N}$ tal que módulo(a) = $|a|$ é sobrejetora;

c) A função $f: \mathbb{Z} \rightarrow \mathbb{N}$ tal que $f(a) = 2a$ se $a \geq 0$ e $f(a) = |2a|-1$ se $a < 0$ é bijetora, ou seja, é um isomorfismo. Portanto, os conjuntos $\mathbb{Z}$ e $\mathbb{N}$ são isomorfos. Note-se que, os inteiros não-negativos são associados aos naturais pares e os inteiros negativos aos naturais ímpares. ❑

Cardinalidade de Conjuntos

A cardinalidade de um conjunto é uma medida de seu tamanho e é definida usando funções bijetoras.

Definição 1.13 Cardinalidade Finita, Infinita.

A *Cardinalidade* de um conjunto A, representada por #A é:

a) *Finita* se existe uma bijeção entre A e o conjunto $\{1, 2, 3, ..., n\}$, para algum $n \in \mathbb{N}$. Neste caso, afirma-se que $\#A = n$ (como fica o caso em que $n = 0$?);

b) *Infinita* se existe uma bijeção entre A e um subconjunto próprio de A. ❑

Portanto, um conjunto é finito (ou seja, possui uma cardinalidade finita) se for possível representá-lo por extensão. Um conjunto A é infinito se for possível retirar alguns elementos de A e, mesmo assim, estabelecer uma bijeção com A.

EXEMPLO 10 Cardinalidade de $\mathbb{Z}$.

A função $f: \mathbb{Z} \rightarrow \mathbb{N}$ tal que $f(a) = 2a$ se $a \geq 0$ e $f(a) = |2a|-1$ se $a < 0$ é bijetora. Claramente, $\mathbb{N}$ é subconjunto próprio de $\mathbb{Z}$. Então $\mathbb{Z}$ é infinito. ❑

É importante destacar que nem todos os conjuntos infinitos possuem a mesma cardinalidade, o que contradiz a noção intuitiva da maioria das pessoas. De especial interesse é o cardinal do conjunto dos números naturais $\mathbb{N}$, denotado por $\aleph_0$. O símbolo $\aleph$ (lê-se "alef") é a primeira letra do alfabeto hebraico.

Definição 1.14 Conjunto Contável, Não-Contável.

Um conjunto infinito A é dito:

a) *Contável* ou *Contavelmente Infinito*, se existe uma bijeção entre o conjunto A e um subconjunto infinito de $\mathbb{N}$;

b) *Não-Contável*, caso contrário. ❑

A bijeção que define se um conjunto A é contável é denominada *enumeração* de A. Portanto, um conjunto é contável se for possível enumerar seus elementos como uma seqüência na forma $a_0, a_1, a_2, ...$. O cardinal de qualquer conjunto contável é $\aleph_0$.

EXEMPLO 11 Conjunto Contável, Não-Contável.

Os conjuntos $\mathbb{Z}$ e $\mathbb{Q}$ são contáveis. Os conjuntos $\mathbb{I}$ e $\mathbb{R}$ são não-contáveis. De fato, prova-se que o cardinal de $\mathbb{I}$ e $\mathbb{R}$ é $2^{\aleph_0}$. ❑

1.3 Lógica

No texto que segue é suposto que o leitor está familiarizado com os conceitos básicos relativos à Lógica Booleana. Entende-se por *Lógica Booleana* como o estudo dos princípios e métodos usados para distinguir sentenças verdadeiras de falsas.

Definição 1.15 Proposição.

a) Uma *Proposição* é uma sentença declarativa a qual possui valor lógico *verdadeiro* ou *falso* (não-verdadeiro);

b) Considere um conjunto universo $\mathbb{U}$. Uma *Proposição Sobre* $\mathbb{U}$ é uma proposição cujo valor lógico depende de um elemento $x \in \mathbb{U}$ considerado. ❑

Os valores lógicos verdadeiro e falso são usualmente denotados por V e F, respectivamente. Uma proposição p a qual descreve alguma propriedade de um elemento $x \in \mathbb{U}$ é usualmente denotada por p(x). Toda a proposição p sobre $\mathbb{U}$ induz uma partição de $\mathbb{U}$ em duas classes de equivalência, como segue:

a) $\{x \mid p(x)$ é verdadeira$\}$, denominado *conjunto verdade* de p

b) $\{x \mid p(x)$ é falsa$\}$, denominado *conjunto falsidade* de p

Definição 1.16 Tautologia, Contradição.

Seja p uma proposição sobre o conjunto universo $\mathbb{U}$. Então:

a) p é dita uma *Tautologia* se p(x) é verdadeira para qualquer $x \in \mathbb{U}$

b) p é dita uma *Contradição* se p(x) é falsa para qualquer $x \in \mathbb{U}$ ❑

EXEMPLO 12 Proposição, Tautologia, Contradição.

a) A sentença $3 + 4 > 5$ é uma proposição;

b) Para a proposição $n! < 10$ sobre $\mathbb{N}$, tem-se que $\{0, 1, 2, 3\}$ é o conjunto verdade e $\{n \in \mathbb{N} \mid n > 3\}$ é o conjunto falsidade;

c) A proposição $n + 1 > n$ sobre $\mathbb{N}$ é uma tautologia;

d) A proposição "2n é ímpar" sobre $\mathbb{N}$ é uma contradição. ❑

Uma *operação* ou um *operador* sobre um conjunto A é uma função da forma $op: A^n \rightarrow A$. Portanto, um *operador lógico*, também denominado *conetivo (lógico)*, é um operador sobre o conjunto das proposições. Uma proposição que não contém operadores é denominada *proposição atômica* ou simplesmente *átomo*. O conjunto de todas as proposições lógicas é denotado por $\mathbb{P}$.

Uma *tabela verdade* é uma tabela que descreve os valores lógicos de uma proposição em termos das possíveis combinações dos valores lógicos das proposições componentes.

Definição 1.17 Operadores Lógicos.

Os seguintes *Operadores* ou *Conetivos* sobre o conjunto das proposições lógicas $\mathbb{P}$ são definidos conforme a tabela verdade ilustrada abaixo:

a) *Negação*. Operador denotado pelo símbolo $\neg$

b) *E*. Operador denotado pelo símbolo $\wedge$

c) *Ou*. Operador denotado pelo símbolo $\vee$

d) *Se-Então*. Operador denotado pelo símbolo $\rightarrow$

e) *Se-Somente-Se*. Operador denotado pelo símbolo $\leftrightarrow$

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
V	V	F	V	V	V	V
V	F	F	F	V	F	F
F	V	V	F	V	V	F
F	F	V	F	F	V	V

❑

Note-se que todos os operadores definidos acima são binários e denotados de forma infixada (operador no meio dos operandos), excetuando-se o de Negação o qual é unário e denotado de forma prefixada.

Os operadores lógicos $\rightarrow$ e $\leftrightarrow$ induzem importantes relações entre proposições.

Definição 1.18 Relação de Implicação, Equivalência.

As seguintes relações são induzidas pelos operadores $\rightarrow$ e $\leftrightarrow$ sobre $\mathbb{P}$:

a) A relação $\Rightarrow$, denominada *Relação de Implicação* ou simplesmente *Implicação*, é definida pelo conjunto:

$$\{(p, q) \in \mathbb{P}^2 \mid p \rightarrow q \text{ é uma tautologia}\}$$

b) A relação $\Leftrightarrow$, denominada *Relação de Equivalência* ou simplesmente *Equivalência*, é definida pelo conjunto:

$$\{(p, q) \in \mathbb{P}^2 \mid p \leftrightarrow q \text{ é uma tautologia}\}$$ ❑

É fácil verificar que $\Rightarrow$ e $\Leftrightarrow$ são relações de ordem e de equivalência, respectivamente.

EXEMPLO 13 Relação de Implicação, Equivalência.

Os seguintes pares de proposições pertencem às relações de implicação ou de equivalência, como pode ser verificado pelas tabelas verdade abaixo:

a) *Adição*.

$p \Rightarrow p \vee q$

b) *Simplificação*.

$p \wedge q \Rightarrow p$

c) *Contraposição*.

$p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$

d) *Redução ao Absurdo.*

$p \rightarrow q \Leftrightarrow (p \wedge \neg q) \rightarrow F$

p	q	$p \vee q$	$p \rightarrow (p \vee q)$	$p \wedge q$	$(p \wedge q) \rightarrow p$
V	V	V	**V**	V	**V**
V	F	V	**V**	F	**V**
F	V	V	**V**	F	**V**
F	F	F	**V**	F	**V**

p	q	$\neg p$	$\neg q$	$p \rightarrow q$	$\neg q \rightarrow \neg p$	$(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$
V	V	F	F	V	V	**V**
V	F	F	V	F	F	**V**
F	V	V	F	V	V	**V**
F	F	V	V	V	V	**V**

p	q	$\neg q$	$p \rightarrow q$	$p \wedge \neg q$	$((p \wedge \neg q) \rightarrow F)$	$(p \rightarrow q) \leftrightarrow ((p \wedge \neg q) \rightarrow F)$
V	V	F	V	F	V	**V**
V	F	V	F	V	F	**V**
F	V	F	V	F	V	**V**
F	F	V	V	F	V	**V**

❑

1.4 Técnicas de Demonstração

Um *teorema* é uma proposição do tipo $p \rightarrow q$ a qual prova-se ser verdadeira sempre, ou seja, que $p \Rightarrow q$. As proposições p e q são denominadas *hipótese* e *tese*, respectivamente. É usual denominar por *corolário* um teorema que é uma conseqüência quase direta de um outro já demonstrado (ou seja, cuja prova é trivial ou imediata). Adicionalmente, um teorema auxiliar que possui um resultado importante para a prova de um outro é usualmente denominado por *lema*.

Dado um teorema a ser demonstrado, é fundamental, antes de iniciar a demonstração, identificar claramente a hipótese e a tese. Por exemplo, considere o seguinte teorema:

a intersecção distribui-se sobre a união, ou seja,
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

Uma reescrita identificando claramente a hipótese e a tese é como segue:

se A, B e C são conjuntos quaisquer,
então $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

É usual um teorema ser apresentado na forma $p \leftrightarrow q$ como, por exemplo, (suponha que A é um conjunto qualquer):

A é contável se, e somente se,
existe uma função bijetora entre A e o conjunto dos números pares

Sugere-se como exercício verificar que:

$$p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$$

Assim, neste caso, deve ser demonstrada a "ida" e a "volta", ou seja, que:

se um conjunto A é contável,
então existe uma função bijetora entre A e o conjunto dos números pares

e

se existe uma função bijetora entre A e o conjunto dos números pares,
então A é contável

Para um determinado teorema $p \rightarrow q$ existem diversas técnicas para provar (demonstrar) que, de fato, $p \Rightarrow q$. As seguintes técnicas destacam-se:

a) Direta;

b) Contraposição;

c) Redução ao absurdo;

d) Indução.

Prova Direta

A *prova direta* simplesmente pressupõe verdadeira a hipótese e, a partir desta, prova ser verdadeira a tese.

EXEMPLO 14 Prova Direta.

Considere o teorema:

a intersecção distribui-se sobre a união, ou seja,
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

exemplificado acima. Inicialmente, lembre-se que:

- por definição, $X = Y$ se, e somente se, $X \subseteq Y$ e $Y \subseteq X$;
- por definição, $X \subseteq Y$ se, e somente se, todos os elementos de X também são elementos de Y.

Adicionalmente, é fácil verificar, usando tabela verdade, que o operador lógico $\wedge$ se distribui sobre o $\vee$, ou seja, para quaisquer proposições p, q e r, tem-se que:

$$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$$

Para provar que $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, deve-se provar que:
$A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$
$(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$

Suponha que A, B e C são conjuntos quaisquer.

Caso 1. $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$. Suponha $x \in A \cap (B \cup C)$. Então:

$x \in A \cap (B \cup C) \Rightarrow$	definição de intersecção
$x \in A \wedge x \in (B \cup C) \Rightarrow$	definição de união
$x \in A \wedge (x \in B \vee x \in C) \Rightarrow$	distributividade do $\wedge$ sobre o $\vee$
$(x \in A \wedge x \in B) \vee (x \in A \wedge x \in C) \Rightarrow$	definição de intersecção
$x \in (A \cap B) \vee x \in (A \cap C) \Rightarrow$	definição de união
$x \in (A \cap B) \cup (A \cap C)$	

Portanto, $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$

Caso 2. $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$. Suponha $x \in (A \cap B) \cup (A \cap C)$. Então:

$x \in (A \cap B) \cup (A \cap C) \Rightarrow$	definição de união
$x \in (A \cap B) \vee x \in (A \cap C) \Rightarrow$	definição de intersecção
$(x \in A \wedge x \in B) \vee (x \in A \wedge x \in C) \Rightarrow$	distributividade do $\wedge$ sobre o $\vee$
$x \in A \wedge (x \in B \vee x \in C) \Rightarrow$	definição de união
$x \in A \wedge x \in (B \cup C) \Rightarrow$	definição de intersecção
$x \in A \cap (B \cup C)$	

Portanto, $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$

Logo, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ ❑

Prova por Contraposição

A *prova por contraposição* baseia-se no seguinte resultado, o qual foi verificado no Exemplo 13:

$$p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$$

EXEMPLO 15 Prova por Contraposição.

Para demonstrar o seguinte teorema:

$$n! > n+1 \rightarrow n > 2$$

pode-se, equivalentemente, demonstrar por contraposição que:

$$n \leq 2 \rightarrow n! \leq n+1$$

Note-se que é muito simples provar que $n \leq 2 \rightarrow n! \leq n + 1$ pois é suficiente testar para os casos $n = 0$, $n = 1$ e $n = 2$. ❑

Prova por Redução ao Absurdo

A *prova por redução ao absurdo* ou simplesmente *prova por absurdo* baseia-se no seguinte resultado, o qual foi verificado no Exemplo 13:

$$p \rightarrow q \Leftrightarrow (p \wedge \neg q) \rightarrow F$$

Ou seja, para demonstrar que $p \rightarrow q$ a técnica consiste em supor a hipótese p, supor a negação da tese $\neg q$ e concluir uma contradição a qual, em geral, é $q \wedge \neg q$. Note-se que, a técnica de demonstração denominada *por contra-exemplo*, é uma demonstração por absurdo. De fato, em uma demonstração por absurdo, a construção da contradição $q \wedge \neg q$ é, em geral, a apresentação de um contra-exemplo.

EXEMPLO 16 Prova por Redução ao Absurdo.

Considere o seguinte teorema:

0 é o único elemento neutro da adição em $\mathbb{N}$

ou seja, reescrevendo na forma de $p \rightarrow q$:

se 0 é elemento neutro da adição em $\mathbb{N}$,
então 0 é o único elemento neutro da adição em $\mathbb{N}$

Uma prova por redução ao absurdo é como segue:

a) Suponha que 0 é o elemento neutro da adição em $\mathbb{N}$ e que não é o único elemento neutro da adição em $\mathbb{N}$. Seja e um elemento neutro da adição em $\mathbb{N}$ tal que $e \neq 0$;

b) Então:

- como 0 é elemento neutro, para qualquer $n \in \mathbb{N}$, tem-se que $n = 0 + n$. Em particular, para $n = e$, tem-se que $e = 0 + e$;
- como e é elemento neutro, para qualquer $n \in \mathbb{N}$, tem-se que $n = n + e$. Em particular, para $n = 0$, tem-se que $0 = 0 + e$;
- portanto, como $e = 0 + e$ e $0 = 0 + e$, pela transitividade da igualdade tem-se que $e = 0$, o que é uma contradição, pois foi suposto que $e \neq 0$.

Logo, é absurdo supor que o elemento neutro da adição em $\mathbb{N}$ não é único. ❑

Prova por Indução

A prova por indução é de fundamental importância neste trabalho, pois é usada com freqüência. A prova por indução é usada em proposições que dependem dos números naturais.

Definição 1.19 Princípio da Indução Matemática.

Seja p(n) uma proposição sobre $\mathbb{N}$. O *Princípio da Indução Matemática* é como segue:

a) p(0) é verdadeira;

b) Para qualquer $k \in \mathbb{N}$, $p(k) \rightarrow p(k+1)$ é verdadeira;

c) Então, para qualquer $n \in \mathbb{N}$, p(n) é verdadeira.

Neste caso, p(0), p(k) e a proposição $p(k) \rightarrow p(k+1)$ denominam-se *base de indução*, *hipótese de indução* e *passo de indução*, respectivamente. ❑

Em uma demonstração por indução, deve-se demonstrar a base de indução p(0) e, fixado um k, supor verdadeira a hipótese de indução p(k) e demonstrar o passo de indução.

Na realidade, o princípio da indução matemática pode ser aplicado a qualquer proposição que dependa de um conjunto para o qual exista uma bijeção com os naturais.

EXEMPLO 17 Prova por Indução.

Considere o seguinte teorema:

para qualquer $n \in \mathbb{N}$, *tem-se que*

$$1 + 2 + \ldots + n = (n2 + n)/2$$

Uma prova por indução é como segue:

a) *Base de Indução.* Seja n = 0. Então:

$$(0^2 + 0)/2 = (0 + 0)/2 = 0/2 = 0$$

Portanto, $1 + 2 + \ldots + n = (n^2 + n)/2$ é verdadeira para n = 0. Note-se que, $1 + 2 + \ldots + n = 0 + 1 + 2 + \ldots + n$ (pois 0 é o elemento neutro da adição) e, portanto, o somatório até n = 0 é perfeitamente definido;

b) *Hipótese de Indução.* Suponha que, para algum $n \in \mathbb{N}$, tem-se que $1 + 2 + \ldots + n = (n^2 + n)/2$

c) *Passo de Indução.* Prova para $1 + 2 + \ldots + n + (n + 1)$

$1 + 2 + \ldots + n + (n + 1) =$

$(1 + 2 + \ldots + n) + (n + 1) =$

$(n^2 + n)/2 + (n + 1) =$

$(n^2 + n)/2 + (2n + 2)/2 =$

$(n^2 + n + 2n + 2)/2 =$

$((n^2 + 2n + 1) + (n + 1))/2 =$

$((n + 1)^2 + (n + 1))/2$

Portanto, $1 + 2 + ... + n + (n + 1) = ((n + 1)^2 + (n + 1))/2$

Logo, para qualquer $n \in \mathbb{N}$, tem-se que $1 + 2 + ... + n = (n^2 + n)/2$ ❑

Na realidade, o princípio da indução matemática pode ser usado também em definições. Como exemplo, veja a Definição 1.8. Uma definição de uma construção usando este princípio é denominada *definição indutiva* ou *recursiva*. Neste caso, afirma-se que a construção é *indutivamente* ou *recursivamente definida*.

1.5 Alfabetos, Palavras, Linguagens e Gramáticas

O Dicionário Aurélio define linguagem como "o uso da palavra articulada ou escrita como meio de expressão e comunicação entre pessoas". Entretanto, esta definição não é suficientemente precisa para permitir o desenvolvimento matemático de uma teoria sobre linguagens. Assim, faremos a seguir algumas definições formais necessárias aos estudos posteriores.

Definição 1.20 Alfabeto.

Um *Alfabeto* é um conjunto finito de *Símbolos*. ❑

Portanto, um conjunto vazio também é considerado um alfabeto. Um *símbolo* (ou *caractere*) é uma entidade abstrata básica a qual não é definida formalmente. Letras e dígitos são exemplos de símbolos freqüentemente usados.

Definição 1.21 Palavra, Cadeia de Caracteres ou Sentença.

Uma *Palavra*, *Cadeia de Caracteres* ou *Sentença* sobre um alfabeto é uma seqüência finita de símbolos (do alfabeto) justapostos. ❑

A *palavra vazia*, representada pelo símbolo ε, é uma palavra sem símbolo. Se Σ representa um alfabeto, então Σ^* denota o conjunto de todas as palavras possíveis sobre Σ. Analogamente, Σ^+ representa o conjunto de todas as palavras sobre Σ excetuando-se a palavra vazia, ou seja, $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Definição 1.22 Tamanho ou Comprimento.

O *Tamanho* ou *Comprimento* de uma palavra w, representado por $|w|$, é o número de símbolos que compõem a palavra. ❑

Definição 1.23 Prefixo, Sufixo, Subpalavra.

Um *Prefixo* (respectivamente, *Sufixo*) de uma palavra é qualquer seqüência de símbolos inicial (respectivamente, final) da palavra. Uma *Subpalavra* de uma palavra é qualquer seqüência de símbolos contígüa da palavra. ❑

EXEMPLO 18 Palavra, Prefixo, Sufixo.

a) abcb é uma palavra sobre o alfabeto {a, b, c}

b) Se $\Sigma = \{a, b\}$, então $\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa,...\}$ e $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa,...\}$

c) $|abcb| = 4$ e $|\varepsilon| = 0$

d) ε, a, ab, abc, abcb são os prefixos da palavra abcb e ε, b, cb, bcb, abcb são os respectivos sufixos;

e) Qualquer prefixo ou sufixo de uma palavra é uma subpalavra. ❑

Definição 1.24 Linguagem Formal.

Uma *Linguagem Formal* é um conjunto de palavras sobre um alfabeto. ❑

EXEMPLO 19 Linguagem.

Suponha o alfabeto $\Sigma = \{a, b\}$. Então:

a) O conjunto vazio e o conjunto formado pela palavra vazia são linguagens sobre Σ (obviamente $\{\} \neq \{\varepsilon\}$);

b) O conjunto de palíndromos (palavras que têm a mesma leitura da esquerda para a direita e vice-versa) sobre Σ é um exemplo de linguagem infinita. Assim, ε, a, b, aa, bb, aaa, aba, bab, bbb, aaaa,... são palavras desta linguagem. ❑

Definição 1.25 Concatenação.

A *Concatenação* é uma operação binária, definida sobre uma linguagem, a qual associa a cada par de palavras uma palavra formada pela justaposição da primeira com a segunda. Uma concatenação é denotada pela justaposição dos símbolos que representam as palavras componentes. A operação de concatenação satisfaz às seguintes propriedades (suponha v, w, t palavras):

a) *Associatividade.*

$v(wt) = (vw)t$

b) *Elemento Neutro à Esquerda e à Direita.*

$\varepsilon w = w = w\varepsilon$ ❑

Uma operação de concatenação definida sobre uma linguagem L não é, necessariamente, fechada sobre L, ou seja, a concatenação de duas palavras de L não é, necessariamente, uma palavra de L.

EXEMPLO 20 Concatenação.

Considere a linguagem L de palíndromos sobre {a, b}. A concatenação das palavras aba e bbb resulta na palavra ababbb a qual não é palíndromo. Portanto, a operação de concatenação não é fechada sobre L. ❑

Definição 1.26 Concatenação Sucessiva.

A *Concatenação Sucessiva* de uma palavra (com ela mesma), representada na forma de um expoente w^n onde w é uma palavra e n indica o número de concatenações sucessivas, é definida indutivamente a partir da concatenação binária, como segue:

a) *Caso 1.* $w \neq \varepsilon$

$w^0 = \varepsilon$

$w^n = w^{n-1}w$, para $n > 0$

b) *Caso 2.* $w = \varepsilon$

$w^n = \varepsilon$, para $n > 0$

w^n é indefinida para $n = 0$ ❑

Note-se que a concatenação sucessiva é indefinida para ε^0.

EXEMPLO 21 Concatenação Sucessiva.

Sejam w uma palavra e a um símbolo. Então:

$w^3 = www$

$w^1 = w$

$a^5 = aaaaa$

a^n = aaa...a (o símbolo a repetido n vezes) ❑

Definição 1.27 Gramática.

Uma *Gramática* é uma quádrupla ordenada G = (V, T, P, S) onde:

- V conjunto finito de símbolos *variáveis* ou *não-terminais*;
- T conjunto finito de símbolos *terminais* disjunto de V;
- P conjunto finito de pares, denominados *regras de produção* tal que a primeira componente é palavra de $(V \cup T)^+$ e a segunda componente é palavra de $(V \cup T)^*$;
- S elemento de V denominado *variável inicial*. ❑

Uma regra de produção (α, β) é representada por $\alpha \rightarrow \beta$. As regras de produção definem as condições de geração das palavras da linguagem. Uma seqüência de regras de produção da forma $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, ..., $\alpha \rightarrow \beta_n$ (mesma componente no lado esquerdo) pode ser abreviada como uma única produção na forma:

$$\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

A aplicação de uma regra de produção é denominado derivação de uma palavra. A aplicação sucessiva de regras de produção permite derivar as palavras da linguagem representada pela gramática.

Definição 1.28 Derivação.

Seja G = (V, T, P, S) uma gramática. Uma *Derivação* é um par da relação denotada por $\Rightarrow$ com domínio em $(V \cup T)^+$ e contra-domínio em $(V \cup T)^*$. Um par (α, β) é representado de forma infixada, como segue:

$$\alpha \Rightarrow \beta$$

A relação $\Rightarrow$ é indutivamente definida como segue:

- para toda a produção da forma $S \rightarrow \beta$ (a primeira componente é o símbolo inicial de G) tem-se que:

$$S \Rightarrow \beta$$

- para todo o par $\alpha \Rightarrow \beta$, onde $\beta = \beta_u\beta_v\beta_w$, se $\beta_v \rightarrow \beta_t$ é regra de P então:

$$\beta \Rightarrow \beta_u\beta_t\beta_w$$ ❑

Portanto, uma derivação é a substituição de uma subpalavra de acordo com uma regra de produção. Quando for desejado explicitar a regra de produção $p \in P$ que define a derivação $\alpha \Rightarrow \beta$, a seguinte notação é usada:

$$\alpha \Rightarrow^p \beta$$

Sucessivos passos de derivação são definidos como segue:

- $\Rightarrow^*$ fecho transitivo e reflexivo da relação $\Rightarrow$, ou seja, zero ou mais passos de derivações sucessivos;
- $\Rightarrow^+$ fecho transitivo da relação $\Rightarrow$, ou seja, um ou mais passos de derivações sucessivos;
- $\Rightarrow^i$ exatos i passos de derivações sucessivos, onde i é um número natural.

Gramática é considerado um formalismo de geração, pois permite derivar ("gerar") todas as palavras da linguagem que representa.

Definição 1.29 Linguagem Gerada.

Seja G = (V, T, P, S) uma gramática. A *Linguagem Gerada* pela gramática G, denotada por L(G) ou GERA(G), é composta por todas as palavras de símbolos terminais deriváveis a partir do símbolo inicial S, ou seja:

$$L(G) = \{ w \in T^* \mid S \Rightarrow^+ w \}$$ ❑

EXEMPLO 22 Gramática, Derivação, Linguagem Gerada.

A gramática G = (V, T, P, S) onde:

V = { S, D }

T = { 0, 1, 2,..., 9 }

$P = \{ S \rightarrow D, S \rightarrow DS, D \rightarrow 0 \mid 1 \mid ... \mid 9 \}$

gera, sintaticamente, o conjunto dos números naturais. Note-se que se distingue os zeros à esquerda. Por exemplo, distingue 123 de 0123 (sugere-se como exercício o desenvolvimento de uma gramática que não distinga zeros à esquerda). Como ilustração, uma derivação do número 243 é como segue (existe mais alguma derivação do número 243?):

$$S \Rightarrow DS \Rightarrow 2S \Rightarrow 2DS \Rightarrow 24S \Rightarrow 24D \Rightarrow 243$$

Logo, pode-se indicar que:

$S \Rightarrow^* 243$
$S \Rightarrow^+ 243$
$S \Rightarrow^6 243$ ❑

EXEMPLO 23 Gramática, Derivação, Linguagem Gerada.

A gramática:

$G = (\{ S, X, Y, A, B, F \}, \{ a, b \}, P, S)$, onde:
$P = \{ S \rightarrow XY,$
$X \rightarrow XaA \mid XbB \mid F$
$Aa \rightarrow aA,\ Ab \rightarrow bA,\ AY \rightarrow Ya,$
$Ba \rightarrow aB,\ Bb \rightarrow bB,\ BY \rightarrow Yb,$
$Fa \rightarrow aF,\ Fb \rightarrow bF,\ FY \rightarrow \varepsilon \}$

gera a linguagem:

$$\{ ww \mid w \text{ é palavra de } \{ a, b \}^* \}$$

Como ilustração, uma derivação da palavra baba é como segue (existe mais alguma derivação da palavra baba?):

$$S \Rightarrow XY \Rightarrow XaAY \Rightarrow XaYa \Rightarrow XbBaYa \Rightarrow XbaBYa \Rightarrow XbaYba \Rightarrow$$
$$FbaYba \Rightarrow bFaYba \Rightarrow baFYba \Rightarrow baba$$

A gramática apresentada gera o primeiro w após X e o segundo w após Y, como segue:

- a cada símbolo terminal gerado após X, é gerada uma variável correspondente;
- esta variável "caminha" na palavra até passar por Y, quando deriva o correspondente terminal;
- para encerrar, X deriva a variável F a qual "caminha" até encontrar Y quando FY deriva a palavra vazia. Lembre-se ε é o elemento neutro da concatenação e, portanto, baεba = baba. ❑

Definição 1.30 Gramáticas Equivalentes.

Duas gramáticas G_1 e G_2 são ditas *Gramáticas Equivalentes* se, e somente se, $GERA(G_1) = GERA(G_2)$. ❑

No texto que segue, freqüentemente são usadas as seguintes convenções:

- A, B, C,..., S, T para símbolos variáveis;
- a, b, c,..., s, t para símbolos terminais;
- u, v, w, x, y, z para palavras de símbolos terminais;
- α, β,... para palavras de símbolos variáveis ou terminais.

1.6 Exercícios

1.1 Qual a relação entre Linguagens Formais e as análises léxica, sintática e semântica?

1.2 Para $A = \{1\}$, $B = \{1, 2\}$ e $C = \{\{1\}, 1\}$, discuta a validade das seguintes proposições:

a) $A \subset B$, $A \subseteq B$, $A \in B$, $A = B$

b) $A \subset C$, $A \subseteq C$, $A \in C$, $A = C$

c) $1 \in A$, $1 \in C$, $\{1\} \in A$, $\{1\} \in C$

1.3 Para $A = \{1, 2\}$ e $B = \{\{1\}, \{2\}, 1, 2\}$ determine o conjunto resultante em cada um dos seguintes itens:

a) $A \cup B$, $A \cap B$

b) $B \cup \varnothing$, $B \cap \varnothing$

c) $B \cup \mathbb{N}$, $B \cap \mathbb{N}$, $B \cup \mathbb{N} \cup \mathbb{R}$, $(B \cup \mathbb{N}) \cap \mathbb{R}$

1.4 Prove as seguintes propriedades (suponha que A e B são conjuntos):

Sugestão: verifique as propriedades apresentadas no Exercício 1.14.

a) *Idempotência.*

$A \cup A = A$

$A \cap A = A$

b) *Comutatividade.*

$A \cup B = B \cup A$

$A \cap B = B \cap A$

c) *Associatividade.*

$A \cup (B \cup C) = (A \cup B) \cup C$

$A \cap (B \cap C) = (A \cap B) \cap C$

d) *Distributividade.*

$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

e) *Duplo Complemento.*

$(A')' = A$

f) *Morgan.*

$(A \cup B)' = A' \cap B'$

$(A \cap B)' = A' \cup B'$

g) *Universo e Vazio.*

$A \cup A' = \mathbb{U}$

$A \cap A' = \varnothing$

1.5 Referente às operações sobre conjuntos, discuta a validade das seguintes proposições (suponha que A é um conjunto):

a) A operação diferença é associativa e comutativa;

b) A operação produto cartesiano é associativa e comutativa;

c) O conjunto das partes do conjunto das partes de A é o conjunto das partes de A.

1.6 Exemplifique cada um dos casos abaixo:

a) Relação que não é simétrica nem antissimétrica;

b) Relação que é simultaneamente simétrica e antissimétrica.

1.7 Determine o fecho transitivo e o fecho transitivo e reflexivo da relação

$$R = \{(1, 1), (1, 2), (2, 1)\}$$

1.8 Suponha que são conhecidos todos os trechos parciais que podem ser percorridos por um carteiro (exemplo: da casa A para a casa B). Usando a noção de grafo como uma relação e o conceito de fecho, como podemos representar todos os caminhos possíveis que o carteiro pode fazer?

1.9 Para cada item abaixo, justifique a sua resposta:

a) Toda função é uma função parcial e vice-versa? Toda função parcial é uma relação e vice-versa?

b) Toda relação de ordem é uma relação de equivalência e vice-versa?

1.10 O conjunto vazio é uma relação, função parcial ou função?

1.11 Seja $f: A \rightarrow B$ uma função. Uma função $g: B \rightarrow A$ é dita:

- *inversa à esquerda* de f se $g \circ f = id_A$;
- *inversa à direita* de f se $f \circ g = id_B$.

Prove que:

a) Se f é injetora, então f tem inversa à esquerda;

b) Se f é sobrejetora, então f tem inversa à direita.

1.12 Prove que $\mathbb{Q}$ é contável.

1.13 Prove que $\Rightarrow$ e $\Leftrightarrow$ são relações de ordem e de equivalência, respectivamente. A relação $\Rightarrow$ é de ordem parcial? E de ordem total?

1.14 Prove as seguintes equivalências:

a) *Idempotência.*
$p \wedge p \Leftrightarrow p$
$p \vee p \Leftrightarrow p$

b) *Comutatividade.*
$p \wedge q \Leftrightarrow q \wedge p$
$p \vee q \Leftrightarrow q \vee p$

c) *Associatividade.*
$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$

d) *Distributividade.*
$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$

e) *Dupla negação.*
$\neg\neg p \Leftrightarrow p$

f) *Morgan.*
$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

1.15 Prove que qualquer operador lógico binário pode ser expresso usando somente os operadores $\neg$ e $\wedge$.

1.16 Suponha que $A(n)$ denota $1 + 2 + \ldots + n = ((2n + 1)^2)/8$. Então:

a) Prove que se $A(k)$ é verdadeiro para um $k \in \mathbb{N}$, então $A(k + 1)$ também é verdadeiro;

b) Discuta a afirmação: "portanto, por indução, tem-se que $A(n)$ é verdadeiro para qualquer $n \in \mathbb{N}$";

c) De fato, $A(n)$ é verdadeiro para qualquer $n \in \mathbb{N}$? Prove a sua resposta.

1.17 Prove por indução que, para qualquer $n \in \mathbb{N}$, tem-se que:

$$1 + 8 + \ldots + n^3 = (1 + 2 + \ldots + n)^2$$

Sugestão: para verificar a base de indução ($n = 0$), lembre-se que zero é o elemento neutro da adição.

1.18 Por que a "prova por indução" que segue não é correta?

a) *Proposição.* Dado um conjunto de n torcedores de futebol, se pelo menos um torcedor é Gremista, então todos os demais torcedores também são Gremistas;

b) "*Prova*". A proposição é trivialmente verdadeira para $n = 1$. O passo de indução pode ser facilmente entendido pelo seguinte exemplo:
   - suponha que a proposição é verdadeira para $n = 3$;
   - sejam T_1, T_2, T_3 e T_4, quatro torcedores dos quais pelo menos um é Gremista (suponha que é T_1);
   - supondo o conjunto $\{T_1, T_2, T_3\}$ e a hipótese de que é verdadeiro para $n = 3$, então T_2 e T_3 são Gremistas;
   - analogamente para $\{T_1, T_2, T_4\}$, tem-se que T_2 e T_4 são Gremistas;
   - portanto, os quatro torcedores são Gremistas!
   - a generalização da construção acima para k e $k+1$, é a prova desejada.

1.19 Desenvolva uma gramática que gere a linguagem correspondente aos identificadores da linguagem Pascal (palavras formadas por uma ou mais letras ou dígitos, as quais sempre iniciam por uma letra). Analogamente para os identificadores em Pascal com tamanho máximo de seis caracteres.

1.20 Desenvolva uma gramática que gere expressões aritméticas com parênteses balanceados, dois operadores (representados por * e +) e um operando (representado por x). Por exemplo, x, x*(x+x) e (((((x))))) são expressões aritméticas válidas.

1.21 Desenvolva uma gramática que gere a linguagem $\{a^n b^n c^n \mid n \geq 0\}$.

2 Linguagens Regulares

O estudo das *Linguagens Regulares* ou *Tipo 3*, conforme será visto ao longo deste capítulo, pode ser abordado através de formalismos:

- operacional ou reconhecedor: *Autômato Finito*, o qual pode ser Determinístico, Não-Determinístico ou com Movimentos Vazios;
- axiomático ou gerador: *Gramática Regular*;
- denotacional: *Expressão Regular*.

Como introduzido no Capítulo 1 - Introdução e Conceitos Básicos, o formalismo Expressão Regular também pode ser considerado como um formalismo gerador.

De acordo com a Hierarquia de Chomsky, trata-se da classe de linguagens mais simples, sendo possível desenvolver algoritmos de reconhecimento ou de geração de pouca complexidade, grande eficiência e de fácil implementação.

Por simplicidade, no texto que segue, "se, e somente se," é abreviado por "sse".

2.1 Sistema de Estados Finitos

Um *Sistema de Estados Finitos* é um modelo matemático de sistema com entradas e saídas discretas. Pode assumir um número *finito* e *pré-definido* de estados. Cada estado resume somente as informações do passado necessárias para determinar as ações para a próxima entrada.

Um forte motivacional para o estudo de Sistemas de Estados Finitos é o fato de poderem ser associados a diversos tipos de sistemas naturais e construídos.

Um exemplo clássico e de simples entendimento é um elevador. Trata-se de um sistema que não memoriza as requisições anteriores. Cada "estado" sumariza as informações "andar corrente" e "direção de movimento". As entradas para o sistema são requisições pendentes.

Analisadores Léxicos e Processadores de Texto (ou algumas ferramentas de Processadores de Texto) também são exemplos de sistemas de estados finitos, onde cada estado, basicamente, memoriza a estrutura do prefixo da palavra em análise.

Entretanto, nem todos os Sistemas de Estados Finitos são adequados para serem estudados por esta abordagem. Um contra-exemplo é o cérebro humano. Existem evidências de que um neurônio pode ser representado por um número finito de bits. O cérebro é composto por cerca de 2^{35} células. Portanto, a princípio, é possível representá-lo por um número finito de estados. Entretanto, o elevado número de combinações de células (e, conseqüentemente, de estados) determina uma abordagem pouco eficiente.

Outro contra-exemplo é o computador. Os estados determinados pelos processadores e memórias podem ser representados como um sistema de estados finitos. Entretanto, o estudo adequado da noção de computabilidade exige uma memória sem limite predefinido. Adiante, é apresentado um outro formalismo de autômato, a Máquina de Turing, mais adequado ao estudo da computabilidade. Note-se que, o estudo da computabilidade e solucionabilidade de problemas não é um dos objetivos desta publicação.

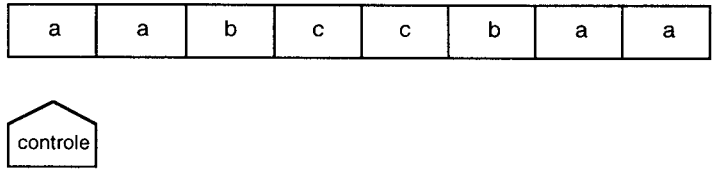


Figura 2.1 Autômato finito como uma máquina com controle finito

2.2 Autômato Finito

Um Autômato Finito Determinístico ou simplesmente Autômato Finito pode ser visto como uma máquina composta, basicamente, de três partes:

a) *Fita*. Dispositivo de entrada que contém a informação a ser processada;

b) *Unidade de Controle*. Reflete o estado corrente da máquina. Possui uma unidade de leitura (cabeça da fita) a qual acessa uma célula da fita de cada vez e movimenta-se exclusivamente para a direita;

c) *Programa* ou *Função de Transição*. Função que comanda as leituras e define o estado da máquina.

A fita é finita (à esquerda e à direita), sendo dividida em células, onde cada uma armazena um símbolo. Os símbolos pertencem a um alfabeto de entrada. Não é possível gravar sobre a fita (e não existe memória auxiliar). Inicialmente, a palavra a ser processada (ou seja, a informação de entrada para a máquina) ocupa toda a fita.

A unidade de controle possui um número finito e predefinido de estados. A unidade de leitura lê o símbolo de uma célula de cada vez. Após a leitura, a *cabeça da fita* move-se uma célula para a direita. Inicialmente, a cabeça está posicionada na célula mais à esquerda da fita, como ilustrado na Figura 2.1.

O programa é uma função parcial que, dependendo do estado corrente e do símbolo lido, determina o novo estado do autômato. Deve-se reparar que o Autômato Finito não possui memória de trabalho. Portanto, para armazenar as informações passadas necessárias ao processamento, deve-se usar o conceito de estado.

Definição 2.1 Autômato Finito Determinístico.

Um *Autômato Finito Determinístico* (*AFD*) ou simplesmente *Autômato Finito* (*AF*) M é uma 5-upla:

$$M = (\Sigma, Q, \delta, q_0, F)$$

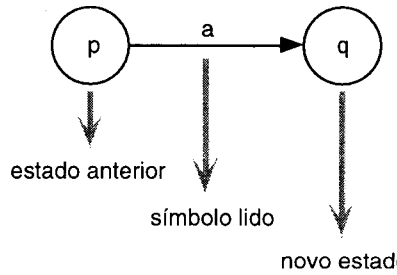


Figura 2.2 Representação da função programa como um grafo

Figura 2.3 Representação de um estado inicial (esq.) e final (dir.) como nodos de grafos

onde:

- Σ *alfabeto de símbolos de entrada*;
- Q *conjunto de estados* possíveis do autômato o qual é finito;
- δ *função programa* ou *função de transição*:

$$\delta: Q \times \Sigma \rightarrow Q$$

a qual é uma função parcial;

- q_0 *estado inicial* tal que q_0 é elemento de Q;
- F *conjunto de estados finais* tal que F está contido em Q. ❑

A função programa pode ser representada como um grafo finito direto, como ilustrado na Figura 2.2. Neste caso, os estados iniciais e finais são representados como ilustrado na Figura 2.3.

O processamento de um Autômato Finito M, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa para cada símbolo de w (da esquerda para a direita) até ocorrer uma condição de parada.

EXEMPLO 1 Autômato Finito.

Considere a linguagem:

$$L_1 = \{ w \mid w \text{ possui aa ou bb como subpalavra} \}$$

O Autômato Finito:

$$M_1 = (\{a, b\}, \{q_0, q_1, q_2, q_f\}, \delta_1, q_0, \{q_f\})$$

onde δ_1 é como abaixo, representada na forma de uma tabela, reconhece a linguagem L_1.

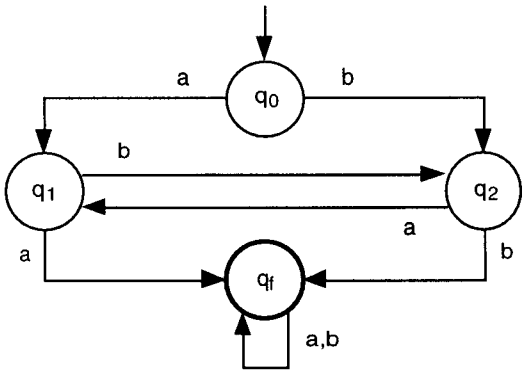


Figura 2.4 Grafo do Autômato Finito Determinístico

δ_1	a	b
q_0	q_1	q_2
q_1	q_f	q_2
q_2	q_1	q_f
q_f	q_f	q_f

O autômato pode ser representado pelo grafo ilustrado na Figura 2.4. O algoritmo apresentado usa os estados q_1 e q_2 para "memorizar" o símbolo anterior. Assim, q_1 representa "símbolo anterior é a" e q_2 representa "símbolo anterior é b". Após identificar dois a ou dois b consecutivos, o autômato assume o estado q_f (final) e varre o sufixo da palavra de entrada, sem qualquer controle lógico, somente para terminar o processamento. A Figura 2.5 ilustra o processamento do Autômato Finito M_1 para a entrada w = abba, a qual é aceita. ❑

Note-se que um Autômato Finito sempre pára ao processar qualquer entrada pois, como qualquer palavra é finita e como um novo símbolo da entrada é lido a cada aplicação da função programa, não existe a possibilidade de ciclo (*loop*) infinito. A parada de um processamento pode ser de duas maneiras: aceitando ou rejeitando uma entrada w. As condições de parada são as seguintes:

a) Após processar o último símbolo da fita, o Autômato Finito assume um estado final: o autômato pára e a entrada w é aceita;

b) Após processar o último símbolo da fita, o Autômato Finito assume um estado não-final: o autômato pára e a entrada w é rejeitada;

c) A função programa é indefinida para o argumento (estado corrente e símbolo lido): a máquina pára e a palavra de entrada w é rejeitada.

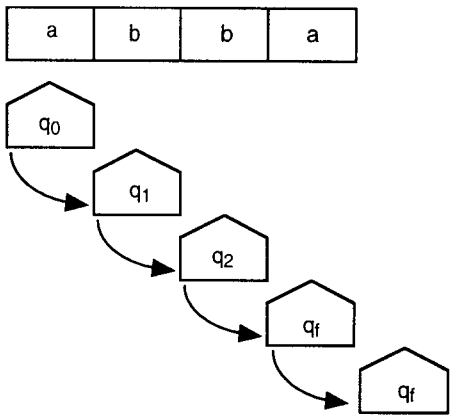


Figura 2.5 Seqüência de processamento

Para definir formalmente o comportamento de um Autômato Finito (ou seja, dar semântica à sintaxe de um Autômato Finito), é necessário estender a definição da função programa usando como argumento um estado e uma palavra.

Definição 2.2 Função Programa Estendida.

Seja $M = (\Sigma, Q, \delta, q_0, F)$ um Autômato Finito Determinístico. A *Função Programa Estendida* denotada por:

$$\underline{\delta}: Q \times \Sigma^* \rightarrow Q$$

é a função programa $\delta: Q \times \Sigma \rightarrow Q$ estendida para palavras e é indutivamente definida como segue:

$\underline{\delta}(q, \varepsilon) = q$
$\underline{\delta}(q, aw) = \underline{\delta}(\delta(q, a), w)$ ❑

EXEMPLO 2 Função Programa Estendida.

Considere o Autômato Finito $M_1 = (\{a, b\}, \{q_0, q_1, q_2, q_f\}, \delta_1, q_0, \{q_f\})$ definido no Exemplo 1. Então, a função programa estendida aplicada à palavra abaa a partir do estado inicial q_0 é como segue:

$\underline{\delta}(q_0, abaa) =$	função estendida sobre abaa
$\underline{\delta}(\delta(q_0, a), baa) =$	processa abaa
$\underline{\delta}(q_1, baa) =$	função estendida sobre baa
$\underline{\delta}(\delta(q_1, b), aa) =$	processa baa
$\underline{\delta}(q_2, aa) =$	função estendida sobre aa
$\underline{\delta}(\delta(q_2, a), a) =$	processa aa
$\underline{\delta}(q_1, a) =$	função estendida sobre a
$\underline{\delta}(\delta(q_1, a), \varepsilon) =$	processa abaa
$\underline{\delta}(q_f, \varepsilon) = q_f$	função estendida sobre ε: fim da indução

e, portanto, a palavra é aceita. ❑

Por simplicidade, no texto que segue, uma função programa δ e a sua correspondente extensão $\underline{\delta}$ são ambas denotadas simplesmente por δ. Esta simplificação de notação será adotada em todas as definições de extensões subseqüentes.

A linguagem aceita por um Autômato Finito $M = (\Sigma, Q, \delta, q_0, F)$, denotada por ACEITA(M) ou L(M), é o conjunto de todas as palavras pertencentes a Σ^* aceitas por M, ou seja:

$$\text{ACEITA(M)} = \{ w \mid \delta(q_0, w) \in F \}$$

Analogamente, REJEITA(M) é o conjunto de todas as palavras pertencentes a Σ^* rejeitadas por M. As seguintes afirmações são verdadeiras:

- $\text{ACEITA(M)} \cap \text{REJEITA(M)} = \varnothing$
- $\text{ACEITA(M)} \cup \text{REJEITA(M)} = \Sigma^*$
- o complemento de ACEITA(M) é REJEITA(M)
- o complemento de REJEITA(M) é ACEITA(M)

Definição 2.3 Autômatos Finitos Equivalentes.

Dois Autômatos Finitos M_1 e M_2 são ditos *Autômatos Finitos Equivalentes* se, e somente se:

$$\text{ACEITA}(M_1) = \text{ACEITA}(M_2)$$ ❑

Definição 2.4 Linguagem Regular ou Tipo 3.

Uma linguagem aceita por um Autômato Finito Determinístico é uma *Linguagem Regular* ou *Tipo 3*. ❑

EXEMPLO 3 Autômato Finito.

Considere as linguagens:

$$L_2 = \{ \} \quad \text{e} \quad L_3 = \Sigma^*$$

Os Autômatos Finitos:

$$M_2 = (\{a, b\}, \{q_0\}, \delta_2, q_0, \{ \}) \quad \text{e} \quad M_3 = (\{a, b\}, \{q_0\}, \delta_3, q_0, \{q_0\})$$

onde δ_2 e δ_3 são como abaixo, representadas na forma de uma tabela, são tais que $\text{ACEITA}(M_2) = L_2$ e $\text{ACEITA}(M_3) = L_3$:

δ_2	a	b
q_0	q_0	q_0

δ_3	a	b
q_0	q_0	q_0

Relativamente aos autômatos M_2 e M_3, sugere-se, como exercício, o esclarecimento das seguintes questões:

Figura 2.6 Grafos dos Autômatos Finitos

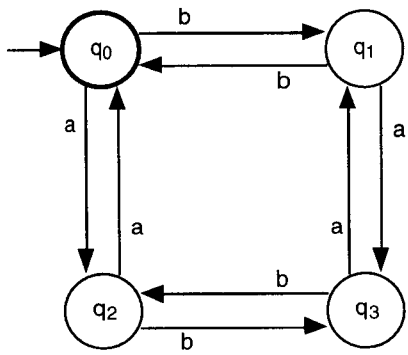


Figura 2.7 Grafo do Autômato Finito Determinístico

a) Existe alguma diferença entre as funções δ_2 e δ_3?

b) O que, exatamente, diferencia M_2 de M_3? ❑

EXEMPLO 4 Autômato Finito.

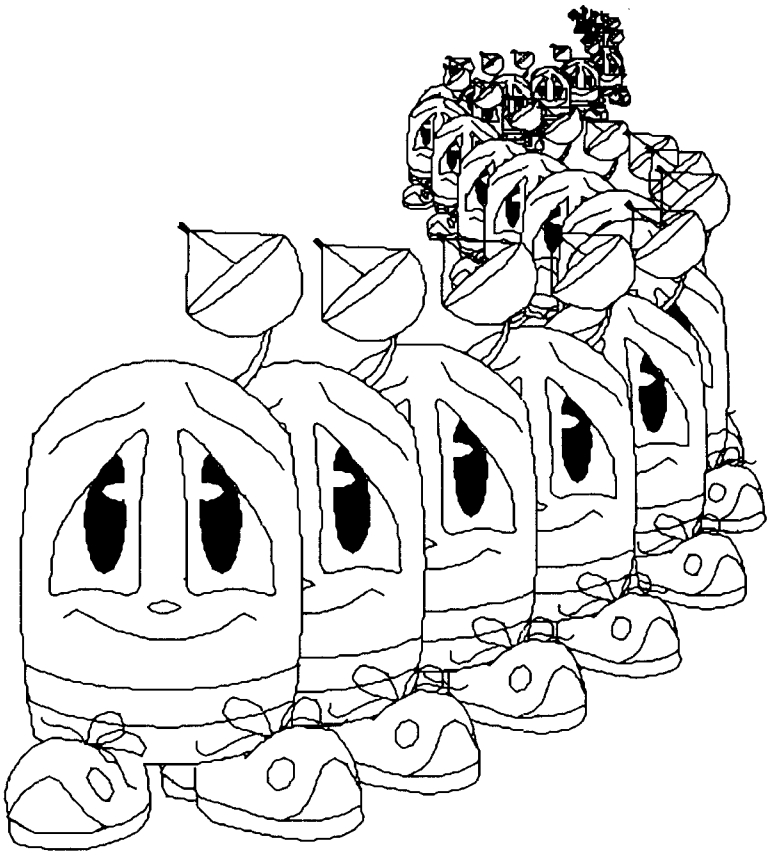
Considere a linguagem:

$$L_4 = \{ w \mid w \text{ possui um número par de a e b} \}$$

O Autômato Finito:

$$M_4 = (\{ a, b \}, \{ q_0, q_1, q_2, q_3 \}, \delta_4, q_0, \{ q_0 \})$$

ilustrado na Figura 2.7, é tal que $ACEITA(M_4) = L_4$. ❑



2.3 Autômato Finito Não-Determinístico

O não-determinismo é uma importante generalização dos modelos de máquinas, sendo de fundamental importância no estudo da teoria da computação e da teoria das linguagens formais. Nem sempre a facilidade de não-determinismo aumenta o poder de reconhecimento de linguagens de uma classe de autômatos. Por exemplo, conforme será mostrado adiante, qualquer Autômato Finito Não-Determinístico pode ser simulado por um Autômato Finito Determinístico.

A facilidade de *não-determinismo* para autômatos finitos é interpretada como segue: a função programa, ao processar uma entrada composta pelo estado corrente e símbolo lido, tem como resultado um conjunto de novos estados. Visto como uma máquina composta por fita, unidade de controle e

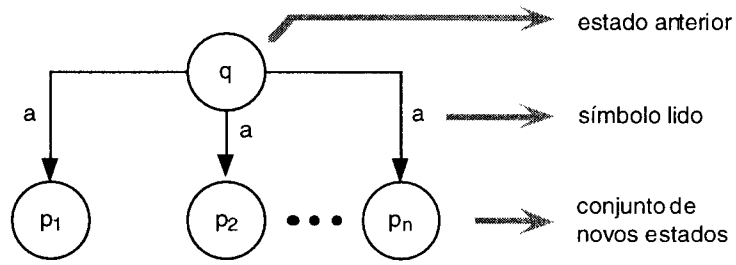


Figura 2.8 Representação da função programa não-determinística como um grafo

programa, pode-se afirmar que um Autômato Não-Determinístico assume um conjunto de estados alternativos, como se houvesse uma multiplicação da unidade de controle, uma para cada alternativa, processando independentemente, sem compartilhar recursos com as demais. Assim, o processamento de um caminho não influi no estado, símbolo lido e posição da cabeça dos demais caminhos alternativos.

Definição 2.5 Autômato Finito Não-Determinístico.

Um *Autômato Finito Não-Determinístico* (*AFN*) M é uma 5-upla:

$$M = (\Sigma, Q, \delta, q_0, F)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- Q *conjunto de estados* possíveis do autômato o qual é finito;
- δ *função programa* ou *função de transição*:

$$\delta: Q \times \Sigma \rightarrow 2^Q$$

  a qual é uma função parcial;
- q_0 *estado inicial* tal que q_0 é elemento de Q;
- F *conjunto de estados finais* tal que F está contido em Q. ❑

Portanto, excetuando-se pela função programa, as componentes Σ, Q, q_0 e F são como na definição do AFD. A função programa pode ser representada como um grafo finito direto, como ilustrado na Figura 2.8, supondo que:

$$\delta(q, a) = \{p_1, p_2, ..., p_n\}$$

O processamento de um Autômato Finito Não-Determinístico M para um conjunto de estados, ao ler um símbolo, é a união dos resultados da função programa aplicada a cada estado alternativo. Para definir formalmente o comportamento de um Autômato Finito Não-Determinístico, é necessário estender a definição da função programa usando como argumento um conjunto finito de estados e uma palavra.

Definição 2.6 Função Programa Estendida.

Seja $M = (\Sigma, Q, \delta, q_0, F)$ um Autômato Finito Não-Determinístico. A *Função Programa Estendida* de M denotada por:

$$\underline{\delta}: 2^Q \times \Sigma^* \rightarrow 2^Q$$

é a função programa $\delta: Q \times \Sigma \rightarrow 2^Q$ estendida para palavras e é indutivamente definida como segue:

$\underline{\delta}(P, \varepsilon) = P$

$\underline{\delta}(P, aw) = \underline{\delta}(\bigcup_{q \in P} \delta(q, a), w)$ ❑

Assim, tem-se que, para um dado conjunto de estados $\{q_1, q_2, ..., q_n\}$ e para um dado símbolo a:

$$\underline{\delta}(\{q_1, q_2, ..., q_n\}, a) = \delta(q_1, a) \cup \delta(q_2, a) \cup ... \cup \delta(q_n, a)$$

A linguagem aceita por um Autômato Finito Não-Determinístico $M = (\Sigma, Q, \delta, q_0, F)$, denotada por ACEITA(M) ou L(M), é o conjunto de todas as palavras pertencentes a Σ^* tais que existe pelo menos um caminho alternativo que aceita a palavra, ou seja:

$$ACEITA(M) = \{w \mid \text{existe } q \in \delta(q_0, w) \text{ tal que } q \in F\}$$

Analogamente, REJEITA(M) é o conjunto de todas as palavras pertencentes a Σ^* rejeitadas por todos os caminhos alternativos de M (a partir de q_0).

EXEMPLO 5 Autômato Finito Não-Determinístico.

Considere a linguagem:

$$L_5 = \{w \mid w \text{ possui aa ou bb como subpalavra}\}$$

O Autômato Finito Não-Determinístico:

$$M_5 = (\{a, b\}, \{q_0, q_1, q_2, q_f\}, \delta_5, q_0, \{q_f\})$$

onde δ_5 é como abaixo, representada na forma de tabela (compare com o Autômato Finito Determinístico do Exemplo 1), é tal que $ACEITA(M_5) = L_5$:

δ_5	a	b
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$
q_1	$\{q_f\}$	-
q_2	-	$\{q_f\}$
q_f	$\{q_f\}$	$\{q_f\}$

O autômato pode ser representado pelo grafo ilustrado na Figura 2.9. O algoritmo apresentado realiza uma varredura sobre a palavra de entrada. A cada ocorrência de a (ou b) uma alternativa é iniciada, para verificar se o

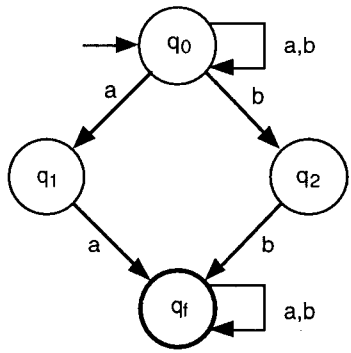


Figura 2.9 Grafo do Autômato Finito Não-Determinístico

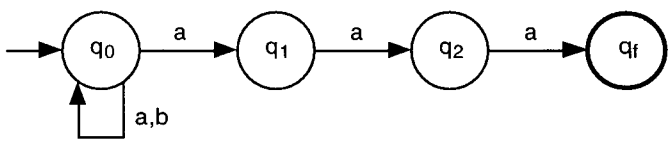


Figura 2.10 Grafo do Autômato Finito Não-Determinístico

símbolo seguinte também é a (ou b). Assim, existem três caminhos alternativos de processamento:

- o ciclo em q_0 realiza uma varredura em toda a entrada;
- o caminho $q_0/q_1/q_f$ garante a ocorrência de aa
- o caminho $q_0/q_2/q_f$ garante a ocorrência de bb ❑

EXEMPLO 6 Autômato Finito Não-Determinístico.

Considere a linguagem:

$$L_6 = \{ w \mid w \text{ possui aaa como sufixo} \}$$

O Autômato Finito Não-Determinístico:

$$M_6 = (\{ a, b \}, \{ q_0, q_1, q_2, q_f \}, \delta_6, q_0, \{ q_f \})$$

ilustrado na Figura 2.10, é tal que $\text{ACEITA}(M_6) = L_6$. ❑

Embora a facilidade de não-determinismo seja, aparentemente, um significativo acréscimo ao Autômato Finito, na realidade não aumenta seu poder computacional. Assim, para cada AFN, é possível construir um AFD equivalente que realiza o mesmo processamento. O contrário também é verdadeiro.

Teorema 2.7 Equivalência entre AFD e AFN.

A Classe dos Autômatos Finitos Determinísticos é equivalente à Classe dos Autômatos Finitos Não-Determinísticos.

Prova: A prova consiste em mostrar que, a partir de um AFN M qualquer, é possível construir um AFD M' que realiza o mesmo processamento (ou seja, M' simula M). A demonstração apresenta um algoritmo para converter um AFN qualquer em um AFD equivalente. A idéia central do algoritmo é a construção de estados de M' que simulem as diversas combinações de estados alternativos de M. O contrário (construir um não-determinístico a partir de um determinístico) não necessita ser mostrado, pois decorre trivialmente das definições (por quê?).

Seja $M = (\Sigma, Q, \delta, q_0, F)$ um AFN qualquer. Seja $M' = (\Sigma, Q', \delta', \langle q_0 \rangle, F')$ um AFD construído a partir de M como segue:

- Q' conjunto de todas as combinações, sem repetições, de estados de Q as quais são denotadas por $\langle q_1 q_2 ... q_n \rangle$, onde q_i pertence a Q, para i em $\{1, 2, ..., n\}$. Note-se que a ordem dos elementos não distingue mais combinações. Por exemplo, $\langle q_u q_v \rangle = \langle q_v q_u \rangle$;
- δ' tal que $\delta'(\langle q_1 ... q_n \rangle, a) = \langle p_1 ... p_m \rangle$ se, e somente se, $\delta(\{q_1, ..., q_n\}, a) = \{p_1, ..., p_m\}$. Ou seja, um estado de M' representa uma imagem dos estados de todos os caminhos alternativos de M;
- $\langle q_0 \rangle$ estado inicial;
- F' conjunto de todos os estados $\langle q_1 q_2 ... q_n \rangle$ pertencentes a Q' tal que alguma componente q_i pertence a F, para i em $\{1, 2, ..., n\}$.

A demonstração de que o AFD M' simula o processamento do AFN M é por indução no tamanho da palavra. Deve-se mostrar que (suponha w uma palavra qualquer de Σ^*):

$$\delta'(\langle q_0 \rangle, w) = \langle q_1 ... q_u \rangle \quad \text{sse} \quad \delta(\{q_0\}, w) = \{q_1, ..., q_u\}$$

a) *Base de indução.* Seja w tal que $|w| = 0$. Então:

$$\delta'(\langle q_0 \rangle, \varepsilon) = \langle q_0 \rangle \quad \text{sse} \quad \delta(\{q_0\}, \varepsilon) = \{q_0\}$$

o que é verdadeiro, por definição de função programa estendida;

b) *Hipótese de indução.* Seja w tal que $|w| = n$ e $n \geq 1$. Suponha verdadeiro que:

$$\delta'(\langle q_0 \rangle, w) = \langle q_1 ... q_u \rangle \quad \text{sse} \quad \delta(\{q_0\}, w) = \{q_1, ..., q_u\}$$

c) *Passo de Indução.* Seja w tal que $|wa| = n + 1$ e $n \geq 1$.

$$\delta'(\langle q_0 \rangle, wa) = \langle p_1 ... p_v \rangle \quad \text{sse} \quad \delta(\{q_0\}, wa) = \{p_1, ..., p_v\}$$

o que equivale, por hipótese de indução:

$$\delta'(\langle q_1 ... q_u \rangle, a) = \langle p_1 ... p_v \rangle \quad \text{sse} \quad \delta(\{q_1, ..., q_u\}, a) = \{p_1, ..., p_v\}$$

o que é verdadeiro, por definição de δ'.

Logo, M' simula M para qualquer entrada w pertencente a Σ^*. □

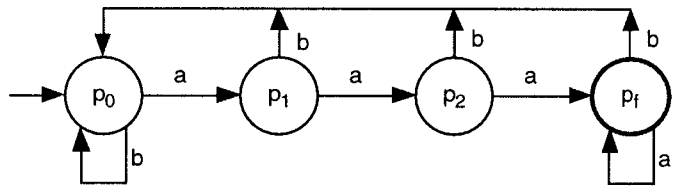


Figura 2.11 Grafo do AFD construído a partir de um AFN

EXEMPLO 7 Construção de um AFD a partir de um AFN.

Considere o AFN $M_6 = (\{a, b\}, \{q_0, q_1, q_2, q_f\}, \delta_6, q_0, \{q_f\})$ do Exemplo 6. O correspondente AFD $M_6' = (\{a, b\}, Q', \delta_6', \langle q_0\rangle, F')$ construído conforme o algoritmo apresentado na demonstração do Teorema 2.7 é como segue:

$Q' = \{\langle q_0\rangle, \langle q_1\rangle, \langle q_2\rangle, \langle q_f\rangle, \langle q_0q_1\rangle, \langle q_0q_2\rangle, ..., \langle q_0q_1q_2q_f\rangle\}$

$F' = \{\langle q_f\rangle, \langle q_0q_f\rangle, \langle q_1q_f\rangle, ..., \langle q_0q_1q_2q_f\rangle\}$

δ_6' é tal que (na tabela que segue são explicitados somente os estados para os quais a função programa é definida):

δ_6'	a	b
$\langle q_0\rangle$	$\langle q_0q_1\rangle$	$\langle q_0\rangle$
$\langle q_0q_1\rangle$	$\langle q_0q_1q_2\rangle$	$\langle q_0\rangle$
$\langle q_0q_1q_2\rangle$	$\langle q_0q_1q_2q_f\rangle$	$\langle q_0\rangle$
$\langle q_0q_1q_2q_f\rangle$	$\langle q_0q_1q_2q_f\rangle$	$\langle q_0\rangle$

O grafo que representa M_6' é ilustrado na Figura 2.11 onde os estados p_0, p_1, p_2 e p_f denotam $\langle q_0\rangle$, $\langle q_0q_1\rangle$, $\langle q_0q_1q_2\rangle$ e $\langle q_0q_1q_2q_f\rangle$, respectivamente. ❑

2.4 Autômato Finito com Movimentos Vazios

Movimentos vazios constituem uma generalização dos modelos de máquinas não-determinística. Um *movimento vazio* é uma transição sem leitura de símbolo algum da fita. Pode ser interpretado como um *não-determinismo interno* ao autômato o qual é encapsulado, ou seja, excetuando-se por uma eventual mudança de estados, nada mais pode ser observado sobre um movimento vazio. Uma das vantagens dos Autômatos Finitos com Movimentos Vazios no estudo das linguagens formais é o fato de facilitar algumas construções e demonstrações relacionadas com os autômatos.

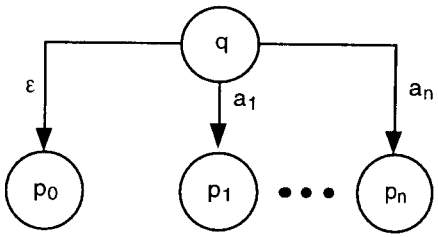


Figura 2.12 Representação como um grafo da função programa com movimentos vazios

No caso dos Autômatos Finitos, a facilidade de movimentos vazios não aumenta o poder de reconhecimento de linguagens. Conforme será mostrado adiante, qualquer Autômato Finito com Movimentos Vazios pode ser simulado por um Autômato Finito Não-Determinístico.

Definição 2.8 Autômato Finito com Movimentos Vazios.

Um *Autômato Finito Não-Determinístico e com Movimentos Vazios* (*AFNε*) ou simplesmente *Autômato Finito com Movimentos Vazios* (*AFε*) M é uma 5-upla:

$$M = (\Sigma, Q, \delta, q_0, F)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- Q *conjunto de estados* possíveis do autômato o qual é finito;
- δ *função programa* ou *função de transição*:

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$$

  a qual é uma função parcial;
- q_0 *estado inicial* tal que q_0 é elemento de Q;
- F *conjunto de estados finais* tal que F está contido em Q. ❑

Portanto, excetuando-se pela função programa, as componentes Σ, Q, F e q_0 são como na definição de AFN. Um *movimento vazio* (ou *transição vazia*) é representado pela aplicação da função programa, em um dado estado q ao símbolo especial ε, ou seja, $\delta(q, \varepsilon)$. A função programa com movimentos vazios pode ser interpretada como um grafo finito direto, como ilustrado na Figura 2.12, supondo que:

$$\delta(q, \varepsilon) = \{p_0\}, \delta(q, a_1) = \{p_1\}, \ldots, \delta(q, a_n) = \{p_n\}$$

O processamento de um AFε é análogo ao de um AFN. Adicionalmente, o processamento de uma transição para uma entrada vazia também é não-determinística. Assim, um AFε ao processar uma entrada vazia assume simultaneamente os estados destino e origem. Ou seja, a origem de um movimento vazio sempre é um caminho alternativo.

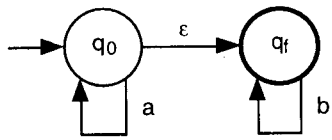


Figura 2.13 Grafo do Autômato Finito com Movimentos Vazios

EXEMPLO 8 Autômato Finito com Movimentos Vazios.

Considere a linguagem:

$$L_7 = \{ w \mid \text{qualquer símbolo a antecede qualquer símbolo b} \}$$

O Autômato Finito com Movimentos Vazios:

$$M_7 = (\{ a, b \}, \{ q_0, q_f \}, \delta_7, q_0, \{ q_f \})$$

ilustrado na Figura 2.13, onde δ_7 é como abaixo, representada na forma de tabela, é tal que $ACEITA(M_7) = L_7$:

δ_7	a	b	ε
q_0	$\{ q_0 \}$	-	$\{ q_f \}$
q_f	-	$\{ q_f \}$	

❑

No que segue, é introduzida a função fecho vazio de um AFε a qual resulta no conjunto de todos os estados atingíveis por zero ou mais movimentos vazios a partir de um determinado estado (ou conjunto de estados). A seguir, é apresentada a função programa estendida onde um argumento é composto por um conjunto finito de estados e uma palavra. A função programa estendida define formalmente o comportamento de um AFε.

Definição 2.9 Função Fecho Vazio, Fecho Vazio Estendida.

Seja $M = (\Sigma, Q, \delta, q_0, F)$ um Autômato Finito com Movimentos Vazios.

a) A *Função Fecho Vazio* denotada por:

$$FECHO\text{-}\varepsilon: Q \rightarrow 2^Q$$

é indutivamente definida como segue:

$FECHO\text{-}\varepsilon(q) = \{ q \}$, se $\delta(q, \varepsilon)$ é indefinido;

$FECHO\text{-}\varepsilon(q) = \{ q \} \cup \delta(q, \varepsilon) \cup (\bigcup_{p \in \delta(q,\varepsilon)} FECHO\text{-}\varepsilon(p))$, caso contrário.

b) A *Função Fecho Vazio Estendida* denotada por:

$$\underline{FECHO}\text{-}\varepsilon: 2^Q \rightarrow 2^Q$$

é a função fecho vazio estendida para conjunto de estados e é tal que:

$\underline{FECHO}\text{-}\varepsilon(P) = \bigcup_{q \in P} FECHO\text{-}\varepsilon(q)$ ❑

Portanto, a função fecho vazio é o fecho reflexivo e transitivo da função programa restrita à palavra vazia.

EXEMPLO 9 Função Fecho Vazio, Fecho Vazio Estendida.

Considere o Autômato Finito com Movimentos Vazios M_7 do Exemplo 8. Então:

$\text{FECHO-}\varepsilon(q_0) = \{ q_0, q_f \}$
$\text{FECHO-}\varepsilon(q_f) = \{ q_f \}$
$\underline{\text{FECHO}}\text{-}\varepsilon(\{ q_0, q_f \}) = \{ q_0, q_f \}$ ❑

Definição 2.10 Função Programa Estendida.

Seja $M = (\Sigma, Q, \delta, q_0, F)$ um Autômato Finito com Movimentos Vazios. A *Função Programa Estendida* de M denotada por:

$$\underline{\delta}: 2^Q \times \Sigma^* \rightarrow 2^Q$$

é a função programa $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ estendida para um conjunto finito de estados e para uma palavra e é indutivamente definida como segue:

$\underline{\delta}(P, \varepsilon) = \text{FECHO-}\varepsilon(P)$
$\underline{\delta}(P, wa) = \text{FECHO-}\varepsilon(R)$ onde $R = \{ r \mid r \in \delta(s, a) \text{ e } s \in \underline{\delta}(P, w) \}$ ❑

A linguagem aceita por um Autômato Finito com Movimentos Vazios $M = (\Sigma, Q, \delta, q_0, F)$, denotada por ACEITA(M) ou L(M), é o conjunto de todas as palavras pertencentes a Σ^* tais que existe pelo menos um caminho alternativo que aceita a palavra, ou seja:

$$\text{ACEITA(M)} = \{ w \mid \text{existe } q \in \delta(q_0, w) \text{ tal que } q \in F \}$$

Analogamente, REJEITA(M) é o conjunto de todas as palavras pertencentes a Σ^* rejeitadas por todos os caminhos alternativos de M (a partir de q_0).

EXEMPLO 10 Função Programa Estendida.

Considere a linguagem:

$$L_8 = \{ w \mid w \text{ possui como sufixo a ou bb ou ccc} \}$$

O Autômato Finito com Movimentos Vazios:

$$M_8 = (\{ a, b, c \}, \{ q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_f \}, \delta_8, q_0, \{ q_f \})$$

ilustrado na Figura 2.14, é tal que $\text{ACEITA}(M_8) = L_8$. Relativamente ao fecho vazio, tem-se que, por exemplo, $\text{FECHO-}\varepsilon(\{ q_0 \}) = \{ q_0, q_1, q_2, q_4 \}$. Exemplificando a função estendida, tem-se que:

$\underline{\delta}(\{ q_0 \}, abb) = \text{FECHO-}\varepsilon(\{ r \mid r \in \delta(s, b) \text{ e } s \in \underline{\delta}(\{ q_0 \}, ab) \})$, onde:
$\underline{\delta}(\{ q_0 \}, ab) = \text{FECHO-}\varepsilon(\{ r \mid r \in \delta(s, b) \text{ e } s \in \underline{\delta}(\{ q_0 \}, a) \})$, onde:
$\underline{\delta}(\{ q_0 \}, a) = \text{FECHO-}\varepsilon(\{ r \mid r \in \delta(s, a) \text{ e } s \in \underline{\delta}(\{ q_0 \}, \varepsilon) \})$

Como $\underline{\delta}(\{ q_0 \}, \varepsilon) \} = \text{FECHO-}\varepsilon(\{ q_0 \}) = \{ q_0, q_1, q_2, q_4 \}$ tem-se que:

$\underline{\delta}(\{ q_0 \}, a) = \{ q_0, q_1, q_2, q_4, q_f \}$
$\underline{\delta}(\{ q_0 \}, ab) = \{ q_0, q_1, q_2, q_3, q_4 \}$
$\underline{\delta}(\{ q_0 \}, abb) = \{ q_0, q_1, q_2, q_3, q_4, q_f \}$ ❑

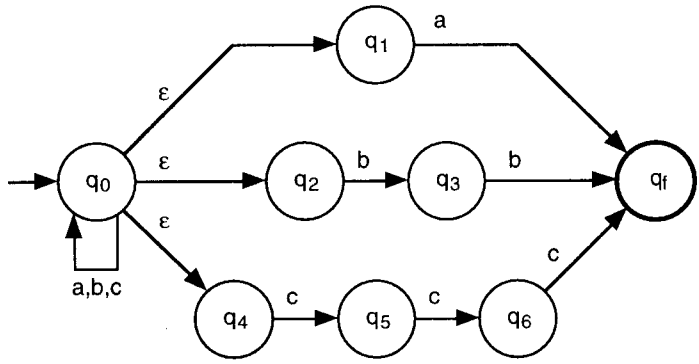


Figura 2.14 Grafo do Autômato Finito com Movimentos Vazios

Analogamente ao não-determinismo, a facilidade de movimentos vazios não aumenta o poder computacional dos Autômatos Finitos. Assim, para cada AFε, é possível construir um AFN que realiza o mesmo processamento. O contrário é trivialmente verdadeiro.

Teorema 2.11 Equivalência entre AFN e AFε.

A Classe dos Autômatos Finitos com Movimentos Vazios é equivalente à Classe dos Autômatos Finitos Não-Determinísticos.

Prova: A prova consiste em mostrar que, a partir de um AFε qualquer, é possível construir um AFN que realiza o mesmo processamento. A demonstração apresenta um algoritmo para converter um AFε em um AFN equivalente. A idéia central do algoritmo é a construção de uma função programa sem movimentos vazios onde o conjunto de estados destino de cada transição não vazia é estendido com todos os estados possíveis de serem atingidos por transições vazias. O contrário (construir um AFε a partir de um AFN) não necessita ser mostrado, pois decorre trivialmente das definições.

Seja $M = (\Sigma, Q, \delta, q_0, F)$ um AFε qualquer. Seja $M' = (\Sigma, Q, \delta', q_0, F')$ um AFN construído a partir de M como segue:

- δ' tal que δ': $Q \times \Sigma \rightarrow 2^Q$ onde $\delta'(q, a) = \underline{\delta}(\{ q \}, a)$
- F' conjunto de todos os estados q pertencentes a Q tal que algum elemento do FECHO-ε(q) pertence a F.

A demonstração de que o AFN M' simula o AFε M é feita por indução no tamanho da palavra e é sugerida como exercício. ❑

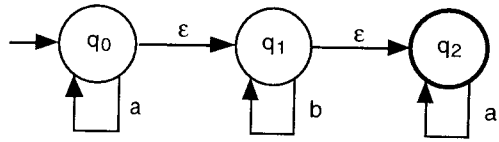


Figura 2.15 Grafo do Autômato Finito com Movimentos Vazios

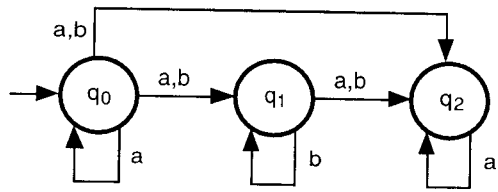


Figura 2.16 Grafo do Autômato Finito Não-Determinístico equivalente

EXEMPLO 11 Construção de um AFN a partir de um AFε.

O Autômato Finito com Movimentos Vazios $M_9 = (\{a, b\}, \{q_0, q_1, q_2\}, \delta_9, q_0, \{q_2\})$, onde δ_9 é como segue:

δ_9	a	b	ε
q_0	$\{q_0\}$	-	$\{q_1\}$
q_1	-	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_2\}$	-	-

é representado pelo grafo ilustrado na Figura 2.15 (qual a linguagem aceita por M_9?). O correspondente AFN $M_9' = (\{a, b\}, \{q_0, q_1, q_2\}, \delta_9', q_0, F')$ construído conforme o algoritmo da demonstração do Teorema 2.11 é como segue (por simplicidade, FECHO-ε é abreviado para Fε) e é ilustrado na Figura 2.16:

$F' = \{q_0, q_1, q_2\}$, pois:

$F\varepsilon(q_0) = \{q_0, q_1, q_2\}$

$F\varepsilon(q_1) = \{q_1, q_2\}$

$F\varepsilon(q_2) = \{q_2\}$

Na construção de δ_9' note-se que:

$\underline{\delta}_9(\{q_0\}, \varepsilon) = \{q_0, q_1, q_2\}$

$\underline{\delta}_9(\{q_1\}, \varepsilon) = \{q_1, q_2\}$

$\underline{\delta}_9(\{q_2\}, \varepsilon) = \{q_2\}$

Assim, δ_9' é tal que:

$\delta_9'(q_0, a) = \underline{\delta}_9(\{ q_0 \}, a) = F\varepsilon(\{ r \mid r \in \delta(s, a) \text{ e } s \in \underline{\delta}(\{ q_0 \}, \varepsilon) \}) = \{ q_0, q_1, q_2 \}$
$\delta_9'(q_0, b) = \underline{\delta}_9(\{ q_0 \}, b) = F\varepsilon(\{ r \mid r \in \delta(s, b) \text{ e } s \in \underline{\delta}(\{ q_0 \}, \varepsilon) \}) = \{ q_1, q_2 \}$
$\delta_9'(q_1, a) = \underline{\delta}_9(\{ q_1 \}, a) = F\varepsilon(\{ r \mid r \in \delta(s, a) \text{ e } s \in \underline{\delta}(\{ q_1 \}, \varepsilon) \}) = \{ q_2 \}$
$\delta_9'(q_1, b) = \underline{\delta}_9(\{ q_1 \}, b) = F\varepsilon(\{ r \mid r \in \delta(s, b) \text{ e } s \in \underline{\delta}(\{ q_1 \}, \varepsilon) \}) = \{ q_1, q_2 \}$
$\delta_9'(q_2, a) = \underline{\delta}_9(\{ q_2 \}, a) = F\varepsilon(\{ r \mid r \in \delta(s, a) \text{ e } s \in \underline{\delta}(\{ q_2 \}, \varepsilon) \}) = \{ q_2 \}$
$\delta_9'(q_2, b) = \underline{\delta}_9(\{ q_2 \}, b) = F\varepsilon(\{ r \mid r \in \delta(s, b) \text{ e } s \in \underline{\delta}(\{ q_2 \}, \varepsilon) \})$ é indefinida. ❑

2.5 Expressão Regular

Toda Linguagem Regular pode ser descrita por uma expressão simples, denominada Expressão Regular. Trata-se de um formalismo denotacional, também considerado gerador, pois pode-se inferir como construir ("gerar") as palavras de uma linguagem. Uma Expressão Regular é definida a partir de conjuntos (linguagens) básicos e operações de concatenação e união. As Expressões Regulares são consideradas adequadas para a comunicação homem × homem e, principalmente, para a comunicação homem × máquina.

Definição 2.12 Expressão Regular.

Uma *Expressão Regular* (*ER*) sobre um alfabeto Σ é indutivamente definida como segue:

a) $\varnothing$ é uma ER e denota a linguagem vazia;

b) ε é uma ER e denota a linguagem contendo exclusivamente a palavra vazia, ou seja, $\{\varepsilon\}$

c) Qualquer símbolo x pertencente a Σ é uma ER e denota a linguagem contendo a palavra unitária x, ou seja, $\{x\}$

d) Se r e s são ER e denotam as linguagens R e S, respectivamente, então:
   d.1) (r+s) é ER e denota a linguagem $R \cup S$
   d.2) (rs) é ER e denota a linguagem $RS = \{ uv \mid u \in R \text{ e } v \in S \}$
   d.3) (r*) é ER e denota a linguagem R^* ❑

A omissão de parênteses em uma ER é usual, respeitando as seguintes convenções:

- a concatenação sucessiva tem precedência sobre a concatenação e a união;
- a concatenação tem precedência sobre a união.

Uma linguagem gerada por uma Expressão Regular r é representada por L(r) ou GERA(r).

EXEMPLO 12 Expressão Regular.

Na tabela abaixo, são apresentadas Expressões Regulares e as correspondentes linguagens:

Expressão Regular	Linguagem Representada
aa	somente a palavra aa
ba*	todas as palavras que iniciam por b, seguido por zero ou mais a
(a + b)*	todas as palavras sobre {a, b}
(a + b)*aa(a + b)*	todas as palavras contendo aa como subpalavra
a*ba*ba*	todas as palavras contendo exatamente dois b
(a + b)*(aa + bb)	todas as palavras que terminam com aa ou bb
(a + ε)(b + ba)*	todas as palavras que não possuem dois a consecutivos

❑

Os teoremas a seguir mostram que a classe das Expressões Regulares denota exatamente a classe das Linguagens Regulares.

Teorema 2.13 Expressão Regular → Linguagem Regular.

Se r é uma Expressão Regular, então GERA(r) é uma Linguagem Regular.

Prova: Por definição, uma linguagem é Regular se, e somente se, é possível construir um Autômato Finito (Determinístico, Não-Determinístico ou com Movimentos Vazios), que reconheça a linguagem. Assim, é necessário mostrar que, dada uma Expressão Regular r qualquer, é possível construir um Autômato Finito M tal que ACEITA(M) = GERA(r). Na construção do correspondente AFε M apresentada a seguir, a demonstração de que ACEITA(M) = GERA(r) é por indução no número de operadores.

a) *Base de indução.* Seja r uma ER com zero operadores. Então r só pode ser da forma:

$r = \varnothing$

$r = \varepsilon$

$r = x$ (x pertencente a Σ)

Os Autômatos Finitos:

$M_1 = (\varnothing, \{q_0\}, \delta_1, q_0, \varnothing)$

$M_2 = (\varnothing, \{q_f\}, \delta_2, q_f, \{q_f\})$

$M_3 = (\{x\}, \{q_0, q_f\}, \delta_3, q_0, \{q_f\})$

ilustrados na Figura 2.17, aceitam as linguagens acima, respectivamente;

b) *Hipótese de Indução.* Seja r uma ER com até $n > 0$ operadores. Suponha que é possível definir um Autômato Finito que aceita a linguagem gerada por r;

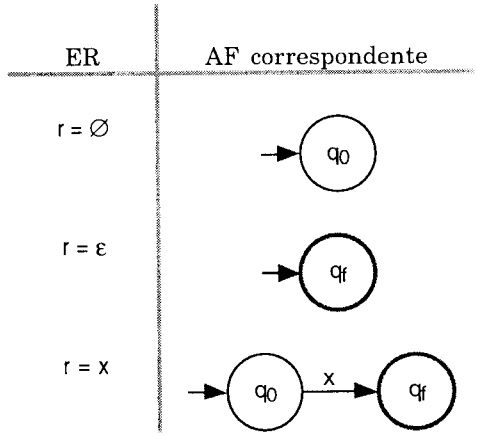


Figura 2.17 Autômatos finitos correspondentes às Expressões Regulares com zero operadores

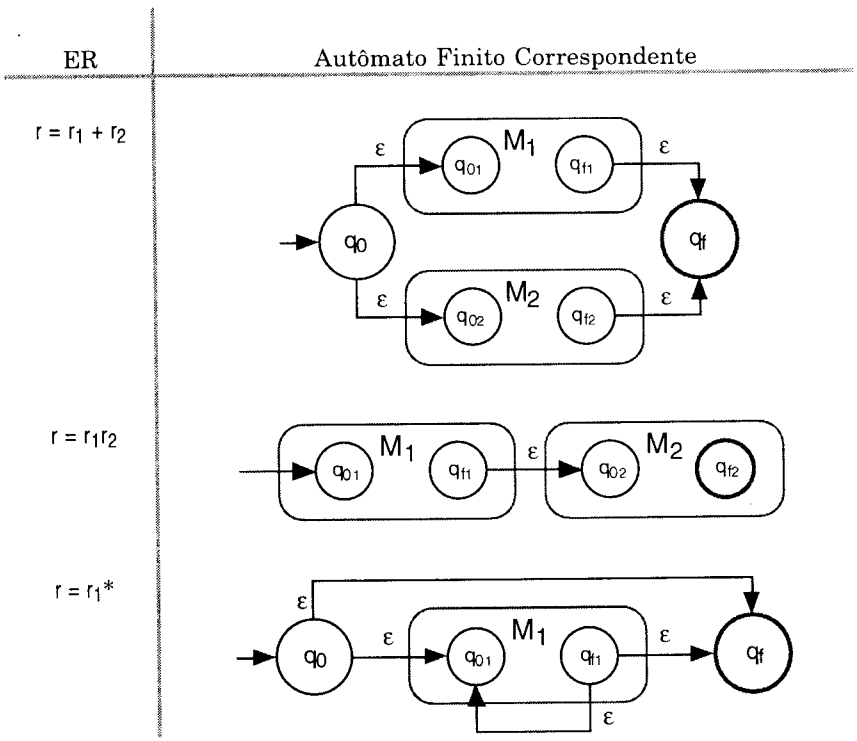


Figura 2.18 Autômatos finitos correspondentes às Expressões Regulares com n+1 operadores

c) *Passo de Indução.* Seja r uma ER com n + 1 operadores. Então r pode ser representada por um dos seguintes casos, onde r_1 e r_2 possuem conjuntamente no máximo n operadores:

$r = r_1 + r_2$

$r = r_1 r_2$

$r = r_1^*$

Portanto, por hipótese de indução é possível construir os autômatos:

$$M_1 = (\Sigma_1, Q_1, \delta_1, q_{0_1}, \{q_{f_1}\}) \quad \text{e} \quad M_2 = (\Sigma_2, Q_2, \delta_2, q_{0_2}, \{q_{f_2}\})$$

tais que ACEITA(M_1) = GERA(r_1) e ACEITA(M_2) = GERA(r_2). Note-se que, sem perda de generalidade, é possível assumir que M_1 e M_2 possuem exatamente um estado final (ver exercícios). Adicionalmente, suponha que os conjuntos de alfabetos e de estados dos dois autômatos são disjuntos (se não forem disjuntos, é suficiente realizar uma simples renomeação). Os Autômatos Finitos com Movimentos Vazios que aceitam a linguagem GERA(r) para cada caso são como segue:

c.1) $r = r_1 + r_2$. O autômato:

$$M = (\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2 \cup \{q_0, q_f\}, \delta, q_0, \{q_f\})$$

ilustrado na Figura 2.18 é tal que, a partir do estado inicial q_0, realiza transições vazias para os estados q_{0_1} e q_{0_2}. Assim, M_1 e M_2 processam de forma não-determinística e, portanto, é suficiente um dos módulos aceitar a entrada para o autômato M aceitar;

c.2) $r = r_1 r_2$. O autômato:

$$M = (\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2, \delta, q_{0_1}, \{q_{f_2}\})$$

ilustrado na Figura 2.18, ao processar os módulos M_1 e M_2 em seqüência, aceita uma entrada se, e somente se, o prefixo pertencer a ACEITA(M_1) e o sufixo pertencer a ACEITA(M_2);

c.3) $r = r_1^*$. O autômato:

$$M = (\Sigma_1, Q_1 \cup \{q_0, q_f\}, \delta, q_0, \{q_f\})$$

ilustrado na Figura 2.18, é tal que a transição vazia de q_0 para q_f garante a aceitação da palavra vazia e o ciclo de q_{f_1} para q_{0_1} permite o sucessivo processamento de M_1 para assegurar o reconhecimento de duas ou mais concatenações sucessivas. ❑

É interessante observar que, na prova do Teorema 2.13, algumas possíveis alterações na construção do autômato resultante podem gerar resultados indesejados, como por exemplo (por quê?):

- no caso $r = r_1 + r_2$, não introduzir os estados q_0 e q_f e identificar ("unificar") os estados iniciais e finais de M_1 com os correspondentes de M_2;

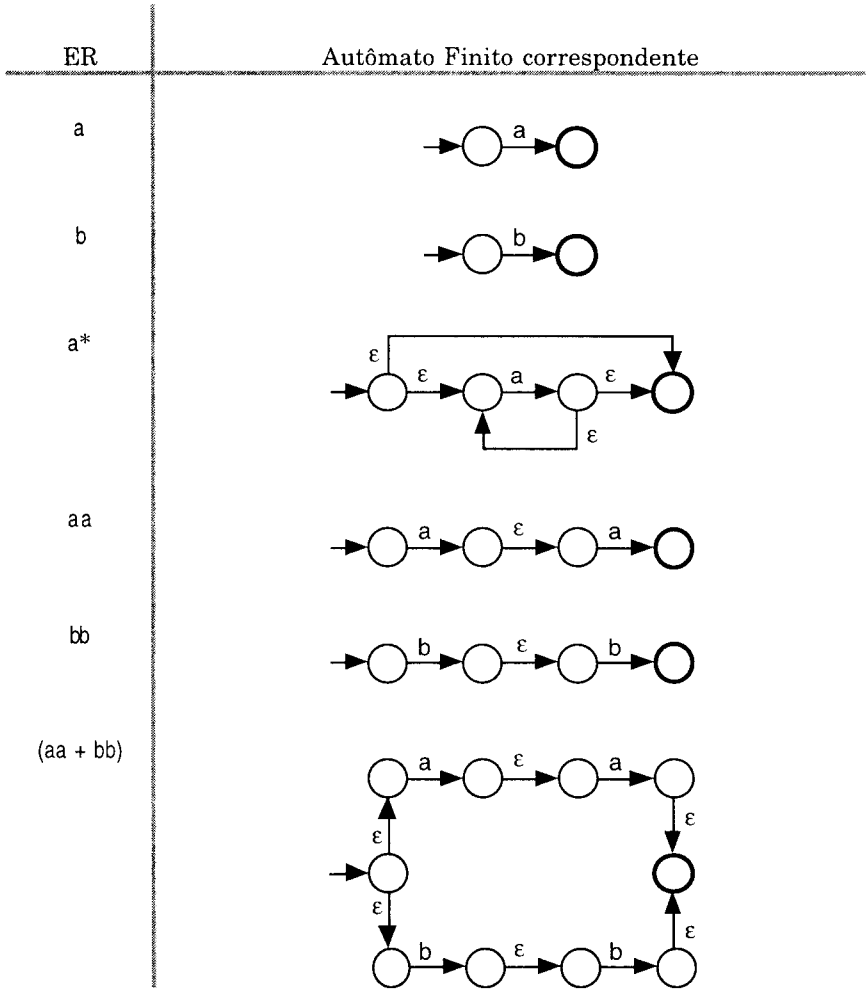


Figura 2.19 Expressões Regulares e os correspondentes Autômatos Finitos

- no caso $r = r_1^*$, não introduzir o estado q_f e manter q_{f_1} como o estado final do autômato resultante. Neste caso, a transição vazia que parte de q_0 teria q_{f_1} como imagem.

EXEMPLO 13 Construção de um AFε a partir de uma Expressão Regular.

A construção dos módulos de um AFε que aceita a linguagem gerada pela ER a*(aa + bb) é como ilustrada na Figura 2.19 e o autômato resultante é ilustrado na Figura 2.20 (por simplicidade, a identificação dos estados é omitida). ❑

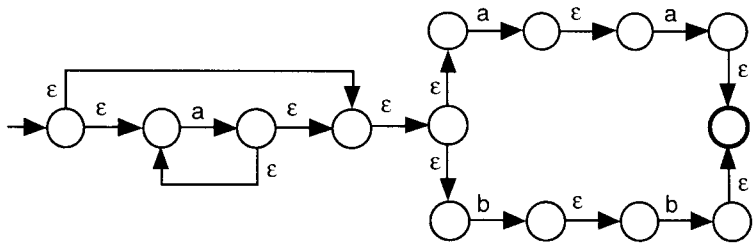


Figura 2.20 Autômato finito resultante

O seguinte teorema não será provado.

Teorema 2.14 Linguagem Regular → Expressão Regular.

Se L é uma Linguagem Regular, então existe uma Expressão Regular r tal que GERA(r) = L. ❑

2.6 Gramática Regular

Usando o conceito geral de gramáticas como introduzido no Capítulo 1 - Introdução e Conceitos Básicos, é possível definir Linguagens Regulares e Não-Regulares (conforme será visto adiante, é fácil definir Linguagens Não-Regulares). Entretanto, é possível estabelecer restrições nas regras de produção, de tal forma a definir exatamente a Classe de Linguagens Regulares.

Definição 2.15 Gramáticas Lineares.

Seja G = (V, T, P, S) uma gramática e sejam A e B elementos de V e w uma palavra de T*. Então G é uma:

a) *Gramática Linear à Direita* (*GLD*). Se todas as regras de produção são da forma:

$$A \rightarrow wB \quad \text{ou} \quad A \rightarrow w$$

b) *Gramática Linear à Esquerda* (*GLE*). Se todas as regras de produção são da forma:

$$A \rightarrow Bw \quad \text{ou} \quad A \rightarrow w$$

c) *Gramática Linear Unitária à Direita* (*GLUD*). Se todas as regras de produção são como na linear à direita e, adicionalmente, $|w| \leq 1$

d) *Gramática Linear Unitária à Esquerda* (*GLUE*). Se todas as regras de produção são como na linear à esquerda e, adicionalmente, $|w| \leq 1$ ❑

Note-se que, nas Gramáticas Lineares, o lado direito de uma produção é constituído por, no máximo, uma variável. Adicionalmente, esta variável, se existir, sempre antecede (linear à esquerda) ou sucede (linear à direita) qualquer subpalavra de terminais.

Teorema 2.16 Equivalência das Gramáticas Lineares.

Seja L uma linguagem. Então:

L é gerada por uma GLD se, e somente se,
L é gerada por uma GLE se, e somente se,
L é gerada por uma GLUD se, e somente se,
L é gerada por uma GLUE. ❑

Ou seja, as diversas formas das Gramáticas Lineares são formalismos equivalentes. A demonstração do teorema é sugerida como exercício.

Definição 2.17 Gramática Regular.

Uma *Gramática Regular* é qualquer Gramática Linear. ❑

Uma linguagem gerada por uma Gramática Regular G é representada por L(G) ou GERA(G).

EXEMPLO 14 Gramática Regular.

A linguagem a(ba)* é gerada pelas seguintes Gramáticas Regulares:

a) *Linear à Direita.* G = ({S, A}, {a, b}, P, S) onde P possui as seguintes regras de produção:
   $S \rightarrow aA$
   $A \rightarrow baA \mid \varepsilon$

b) *Linear à Esquerda.* G = ({S}, {a, b}, P, S) onde P possui as seguintes regras de produção:
   $S \rightarrow Sba \mid a$

c) *Linear Unitária à Direita.* G = ({S, A, B}, {a, b}, P, S) onde P possui as seguintes regras de produção:
   $S \rightarrow aA$
   $A \rightarrow bB \mid \varepsilon$
   $B \rightarrow aA$

d) *Linear Unitária à Esquerda.* G = ({S, A}, {a, b}, P, S) onde P possui as seguintes regras de produção:
   $S \rightarrow Aa \mid a$
   $A \rightarrow Sb$ ❑

EXEMPLO 15 Gramática Regular.

A linguagem (a + b)*(aa + bb) é gerada pelas seguintes Gramáticas Regulares:

a) *Linear à Direita.* G = ({S, A}, {a, b}, P, S) onde P possui as seguintes regras de produção:

$S \rightarrow aS \mid bS \mid A$

$A \rightarrow aa \mid bb$

b) *Linear à Esquerda.* G = ({S, A}, {a, b}, P, S) onde P possui as seguintes regras de produção:

$S \rightarrow Aaa \mid Abb$

$A \rightarrow Aa \mid Ab \mid \varepsilon$ ❑

Os teoremas a seguir mostram que a Classe das Gramáticas Regulares denota exatamente a Classe das Linguagens Regulares.

Teorema 2.18 Gramática Regular → Linguagem Regular.

Se L é uma linguagem gerada por uma Gramática Regular, então L é uma Linguagem Regular.

Prova: Para mostrar que uma linguagem é Regular, é suficiente construir um Autômato Finito que a reconheça. Suponha G = (V, T, P, S) uma Gramática Linear Unitária à Direita. Então o AFε M = (Σ, Q, δ, q_0, F) construído abaixo simula as derivações de G, ou seja, ACEITA(M) = GERA(G):

$\Sigma = T$

$Q = V \cup \{q_f\}$

$F = \{q_f\}$

$q_0 = S$

δ é construída como segue (suponha A e B variáveis e a terminal):

Tipo da Produção	Transição Gerada
$A \rightarrow \varepsilon$	$\delta(A, \varepsilon) = q_f$
$A \rightarrow a$	$\delta(A, a) = q_f$
$A \rightarrow B$	$\delta(A, \varepsilon) = B$
$A \rightarrow aB$	$\delta(A, a) = B$

A demonstração que, de fato, ACEITA(M) = GERA(G) é no número de derivações, como segue (suponha α elemento de $(T \cup V)^*$ e w elemento de T^*):

a) *Base de indução.* Suponha que $S \Rightarrow^1 \alpha$. Então, se:

a.1) $\alpha = \varepsilon$, existe uma regra $S \rightarrow \varepsilon$. Por construção de M, $\delta(S, \varepsilon) = q_f$

a.2) $\alpha = a$, existe uma regra $S \rightarrow a$. Por construção de M, $\delta(S, a) = q_f$

a.3) $\alpha = A$, existe uma regra $S \rightarrow A$. Por construção de M, $\delta(S, \varepsilon) = A$

a.4) $\alpha = aA$, existe uma regra $S \rightarrow aA$. Por construção de M, $\delta(S, a) = A$

b) *Hipótese de indução.* Suponha que $S \Rightarrow^n \alpha$, $n > 1$ tal que, se:

b.1) $\alpha = w$, então $\underline{\delta}(S, w) = q_f$

b.2) $\alpha = wA$, então $\underline{\delta}(S, w) = A$

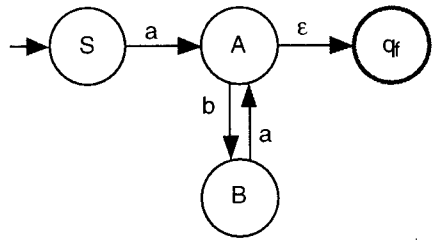


Figura 2.21 Autômato finito construído a partir de uma Gramática Regular

c) *Passo de Indução*. Suponha que $S \Rightarrow^{n+1} \alpha$. Então, obrigatoriamente ocorre somente a hipótese b.2) acima e $S \Rightarrow^{n} wA \Rightarrow^{1} \alpha$. Portanto, se:

c.1) $\alpha = w\varepsilon = w$, existe uma regra $A \rightarrow \varepsilon$. Assim:

$$\underline{\delta}(S, w\varepsilon) = \delta(\underline{\delta}(S, w), \varepsilon) = \delta(A, \varepsilon) = q_f$$

c.2) $\alpha = wb$, existe uma regra $A \rightarrow b$. Assim:

$$\underline{\delta}(S, wb) = \delta(\underline{\delta}(S, w), b) = \delta(A, b) = q_f$$

c.3) $\alpha = wB$, existe uma regra $A \rightarrow B$. Assim:

$$\underline{\delta}(S, w\varepsilon) = \delta(\underline{\delta}(S, w), \varepsilon) = \delta(A, \varepsilon) = B$$

c.4) $\alpha = wbB$, existe uma regra $A \rightarrow bB$. Assim:

$$\underline{\delta}(S, wb) = \delta(\underline{\delta}(S, w), b) = \delta(A, b) = B$$ ❑

EXEMPLO 16 Construção de um AFε a partir de uma Gramática Regular.

Considere a seguinte Gramática Linear Unitária à Direita, introduzida no Exemplo 14:

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

onde P é tal que:

$S \rightarrow aA$
$A \rightarrow bB \mid \varepsilon$
$B \rightarrow aA$

O AFε M que reconhece a linguagem gerada pela gramática G é:

$$M = (\{a, b\}, \{S, A, B, q_f\}, \delta, S, \{q_f\})$$

o qual é ilustrado na Figura 2.21 e onde δ é construída como segue:

Produção	Transição
$S \rightarrow aA$	$\delta(S, a) = A$
$A \rightarrow bB$	$\delta(A, b) = B$
$A \rightarrow \varepsilon$	$\delta(A, \varepsilon) = q_f$
$B \rightarrow aA$	$\delta(B, a) = A$

❑

Teorema 2.19 Linguagem Regular → Gramática Regular.

Se L é Linguagem Regular, então existe G, Gramática Regular que gera L.

Prova: Se L é uma Linguagem Regular, então existe um AFD $M = (\Sigma, Q, \delta, q_0, F)$ tal que ACEITA(M) = L. A idéia central da demonstração é construir uma Gramática Linear à Direita G a partir de M tal que GERA(G) = ACEITA(M), ou seja, cuja derivação simula a função programa estendida. Seja $G = (V, T, P, S)$ tal que:

$V = Q \cup \{S\}$
$T = \Sigma$

e P é construído como segue (suponha q_i, q_k elementos de Q, q_f elemento de F e a elemento de Σ):

Transição	Produção
-	$S \rightarrow q_0$
-	$q_f \rightarrow \varepsilon$
$\delta(q_i, a) = q_k$	$q_i \rightarrow aq_k$

A demonstração que, de fato, GERA(G) = ACEITA(M) é no tamanho da palavra, como segue (suponha w elemento de Σ^*):

a) *Base de Indução.* Seja w tal que $|w| = 0$. Por definição, tem-se que $S \rightarrow q_0$ é produção. Se $\varepsilon \in$ ACEITA(M), então q_0 é estado final e, por definição, tem-se que $q_0 \rightarrow \varepsilon$ é produção. Logo:

$$S \Rightarrow q_0 \Rightarrow \varepsilon$$

b) *Hipótese de Indução.* Seja w tal que $|w| = n\ (n \geq 1)$ e $\underline{\delta}(q_0, w) = q$. Assim, se:
   b.1) q não é estado final, então suponha que $S \Rightarrow wq$
   b.2) q é estado final, então suponha que $S \Rightarrow wq \Rightarrow w$ (este caso não é importante para o passo de indução);

c) *Passo de Indução.* Seja w tal que $|wa| = n + 1$ e $\underline{\delta}(q_0, wa) = p$. Então:

$$\delta(\underline{\delta}(q_0, w), a) = \delta(q, a) = p$$

Portanto, obrigatoriamente ocorre somente a hipótese b.1) acima e, se:
   c.1) p não é estado final, então $S \Rightarrow^n wq \Rightarrow^1 wap$
   c.2) p é estado final, então $S \Rightarrow^n wq \Rightarrow^1 wap \Rightarrow^1 wa$ ❑

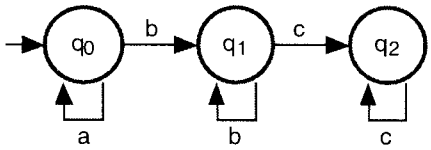


Figura 2.22 Grafo do Autômato Finito Determinístico

EXEMPLO 17 Construção de uma Gramática Regular a partir de um AFD.

Considere o Autômato Finito Determinístico:

$$M = (\{a, b, c\}, \{q_0, q_1, q_2\}, \delta, q_0, \{q_0, q_1, q_2\})$$

ilustrado na Figura 2.22. A correspondente Gramática Regular construída conforme o algoritmo da prova do Teorema 2.19 é:

$$G = (\{q_0, q_1, q_2, S\}, \{a, b, c\}, S, P)$$

onde P é como segue:

Transição	Produção
-	$S \rightarrow q_0$
-	$q_0 \rightarrow \varepsilon$
-	$q_1 \rightarrow \varepsilon$
-	$q_2 \rightarrow \varepsilon$
$\delta(q_0, a) = q_0$	$q_0 \rightarrow aq_0$
$\delta(q_0, b) = q_1$	$q_0 \rightarrow bq_1$
$\delta(q_1, b) = q_1$	$q_1 \rightarrow bq_1$
$\delta(q_1, c) = q_2$	$q_1 \rightarrow cq_2$
$\delta(q_2, c) = q_2$	$q_2 \rightarrow cq_2$

❑

2.7 Propriedades das Linguagens Regulares

Uma das principais características das Linguagens Regulares é o fato de serem representadas por formalismos de pouca complexidade, grande eficiência e fácil implementação. Entretanto, por ser uma classe relativamente simples, é restrita e limitada, sendo fácil definir Linguagens Não-Regulares. Assim, algumas questões sobre Linguagens Regulares necessitam ser analisadas:

a) Como determinar se uma linguagem é Regular?

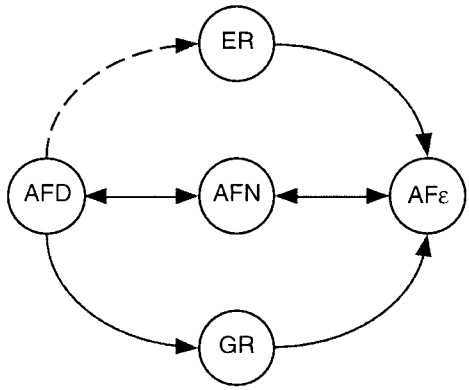


Figura 2.23 Tradução dos formalismos das Linguagens Regulares

b) Como verificar se uma Linguagem Regular é infinita ou finita (ou até mesmo vazia)?

c) É possível analisar duas Linguagens Regulares e concluir se são iguais ou diferentes?

d) A Classe das Linguagens Regulares é fechada para operações de união, concatenação e intersecção?

No estudo que segue, a análise de cada propriedade é desenvolvida para um dos formalismos estudados. Para os demais formalismos, é suficiente traduzi-los usando os algoritmos apresentados nos correspondentes teoremas como ilustrado na Figura 2.23 (note-se que, a construção de uma ER a partir de um AFD não foi provada).

O lema a seguir, conhecido por *Lema do Bombeamento para as Linguagens Regulares*, é útil no estudo das propriedades. Resumidamente, a idéia básica é a seguinte:

- se uma linguagem é Regular, então é aceita por um Autômato Finito Determinístico o qual possui um número finito e predefinido de n estados;
- se o autômato reconhece uma entrada w de comprimento maior ou igual a n, obrigatoriamente o autômato assume algum estado q mais de uma vez e, portanto, existe um ciclo na função programa que passa por q;
- logo, w pode ser dividida em três subpalavras $w = uvz$ tal que $|uv| \leq n$, $|v| \geq 1$ e onde v é a parte de w reconhecida pelo ciclo. Portanto, claramente uv^iz, para qualquer $i \geq 0$, é aceita pelo autômato (ou seja, é palavra da linguagem), executando o ciclo i vezes.

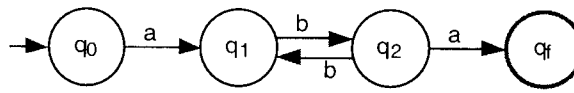


Figura 2.24 Autômato Finito Determinístico

Lema 2.20 Bombeamento para as Linguagens Regulares.

Se L é uma Linguagem Regular, então:

existe uma constante n tal que,
para qualquer palavra w de L onde $|w| \geq n$,
w pode ser definida como w = uvz onde $|uv| \leq n$, $|v| \geq 1$ e,
para todo $i \geq 0$, uv^iz é palavra de L.

Prova: Seja L uma Linguagem Regular. Então:

existe um AFD $M = (\Sigma, Q, \delta, q_0, F)$ tal que ACEITA(M) = L
seja n o cardinal de Q
seja $w = a_1a_2...a_m$ uma palavra de L tal que $m \geq n$
suponha que $\delta(q_0, a_1) = q_1$, $\delta(q_1, a_2) = q_2$, ..., $\delta(q_{m-1}, a_m) = q_m$
como $m \geq n$ existem r, s onde $0 \leq r < s \leq n$ tais que:
$q_r = q_s$, $\delta(q_0, a_1...a_r) = q_r$, $\delta(q_r, a_{r+1}...a_s) = q_s$ e $\delta(q_s, a_{s+1}...a_m) = q_m$
sejam $u = a_1...a_r$, $v = a_{r+1}...a_s$ e $z = a_{s+1}...a_m$
como $r < s \leq n$, então $|v| \geq 1$ e $|uv| \leq n$
como $q_r = q_s$, então v é reconhecida em um ciclo.

Portanto, uv^iz é palavra de L, para todo $i \geq 0$ ❑

EXEMPLO 18 Bombeamento para as Linguagens Regulares.

Considere o autômato ilustrado na Figura 2.24 (qual a linguagem aceita?). Exemplificando o Lema do Bombeamento, tem-se que:

n = 4
no caso particular de w = abbba, tem-se que:
$q_r = q_s = q_1$
u = a, v = bb, z = ba ❑

Linguagens Regulares e Não-Regulares

Para mostrar que uma determinada linguagem é Regular, é suficiente representá-la usando um dos formalismos apresentados anteriormente (Autômato Finito, Expressão Regular ou Gramática Regular). Entretanto, a prova de que é Não-Regular necessita ser desenvolvida para cada caso. Uma ferramenta útil é o Lema do Bombeamento, como exemplificado a seguir.

EXEMPLO 19 Linguagem Não-Regular.

A seguinte linguagem sobre {a, b} não é Regular:

$$L = \{w \mid w \text{ possui o mesmo número de símbolos a e b}\}$$

A prova que segue usa o Lema do Bombeamento e é por absurdo. Suponha que L é Regular. Então:

existe um AFD M com n estados que aceita L
seja $w = a^n b^n$

Pelo Lema do Bombeamento, w pode ser definida como w = uvz onde:
$|uv| \leq n$, $|v| \geq 1$ e,
para todo $i \geq 0$, uv^iz é palavra de L

o que é um absurdo, pois, como $|uv| \leq n$, uv é composta exclusivamente por símbolos a. Neste caso, por exemplo, uv^2z não pertence a L, pois não possui o mesmo número de símbolos a e b. ❑

Operações Fechadas sobre Linguagens Regulares

No texto que segue, para uma determinada linguagem L sobre Σ^*, L' denota o seu complemento (sobre Σ^*).

Teorema 2.21 Operações Fechadas sobre as Linguagens Regulares.

A Classe das Linguagens Regulares é fechada para as seguintes operações:

- união;
- concatenação;
- complemento;
- intersecção.

Prova: A prova referente aos casos de união e concatenação decorrem trivialmente da definição de Expressão Regular (por quê?). Nos demais casos, tem-se que:

a) *Complemento.* Seja L uma Linguagem Regular sobre Σ^*. Seja:

$$M = (\Sigma, Q, \delta, q_0, F)$$

um AFD tal que ACEITA(M) = L. A idéia do que segue consiste em inverter as condições de ACEITA/REJEITA de M para reconhecer L'. Entretanto, como M pode rejeitar por indefinição, é necessário modificar o autômato, garantindo que somente irá parar ao terminar de ler toda a entrada. Assim, a inversão das condições ACEITA/REJEITA pode ser obtida transformando os estados finais em não-finais e vice-versa. A construção do AFD:

$$M' = (\Sigma, Q', \delta', q_0, F')$$

tal que ACEITA(M') = L' é como segue (suponha $a \in \Sigma$ e $q \in Q$):

$Q' = Q \cup \{d\}$

$F' = Q' - F$

δ' é como δ, com as seguintes transições adicionais, para todo $a \in \Sigma$:

$\delta'(q, a) = d$ se $\delta(q, a)$ não é definida

$\delta'(d, a) = d$

Claramente, o Autômato Finito M' construído acima é tal que ACEITA(M') = L'

b) *Intersecção*. Sejam L_1 e L_2 Linguagens Regulares. Como a seguinte igualdade é válida:

$$L_1 \cap L_2 = (L_1' \cup L_2')'$$

e como já foi verificado que a Classe das Linguagens Regulares é fechada para as operações de complemento e união, então também é fechada para a intersecção. ❑

Investigação se uma Linguagem Regular é Vazia, Finita ou Infinita

O teorema a seguir mostra que existe um algoritmo para verificar se uma Linguagem Regular representada por um Autômato Finito é vazia, finita ou infinita.

Teorema 2.22 Linguagem Regular Vazia, Finita ou Infinita.

Se L é uma Linguagem Regular aceita por um Autômato Finito $M = (\Sigma, Q, \delta, q_0, F)$ com n estados (o cardinal de Q é n), então L é:

a) Vazia se, e somente se, M não aceita qualquer palavra w tal que $|w| < n$

b) Finita se, e somente se, M não aceita uma palavra w tal que $n \leq |w| < 2n$

c) Infinita se, e somente se, M aceita uma palavra w tal que $n \leq |w| < 2n$

Prova:

Infinita ($\leftarrow$). Processa M para toda a palavra w de comprimento $n \leq |w| < 2n$. Se o autômato aceita alguma palavra então L é infinita. De fato, se $w \in L$ é tal que $n \leq |w| < 2n$, então, pelo Bombeamento, $w = uvz$ onde $|uv| \leq n$, $|v| \geq 1$ e, para todo $i \geq 0$, uv^iz é palavra de L. Logo, L é infinita.

Infinita ($\rightarrow$). Se L é infinita, então obviamente existe w tal que $|w| \geq n$. Assim:

- se $|w| < 2n$, então a prova está completa;
- suponha que não existe palavra de comprimento menor que 2n. Então suponha que $|w| \geq 2n$ mas de comprimento menor ou igual a qualquer outra palavra t tal que $|t| \geq 2n$. Pelo Bombeamento, $w = uvz$ onde $|uv| \leq n$, $|v| \geq 1$ e, para todo $i \geq 0$, uv^iz é palavra de L. Em, particular, $1 \leq |v| \leq n$ e uz é palavra de L, o que é um absurdo, pois:

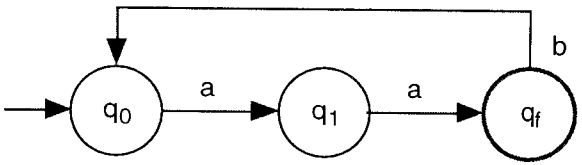


Figura 2.25 Autômato Finito Determinístico

- se $|uz| \geq 2n$, então contradiz a suposição de que w é palavra de menor comprimento tal que $|w| \geq 2n$
- se $|uz| < 2n$, então $n \leq |uz| < 2n$ (pois $|uvz| \geq 2n$, $1 \leq |v| \leq n$) e, portanto, contradiz a suposição de que não existe w tal que $n \leq |w| < 2n$

Vazia. Processa M para todas as palavras de comprimento menor que n. Se rejeita todas as palavras, a linguagem é vazia. O detalhamento da prova é simples usando o Bombeamento e é sugerida como exercício.

Finita. É conseqüência direta do caso onde L é infinita, provado acima. Lembre-se que:

$$(p \leftrightarrow q) \Leftrightarrow (\neg p \leftrightarrow \neg q)$$

Ou seja, processa M para toda palavra w de comprimento $n \leq |w| < 2n$. Se o autômato rejeita todas as palavras, então L é finita. ❑

EXEMPLO 20 Linguagem Regular Infinita.

Considere o autômato ilustrado na Figura 2.25 (qual a linguagem aceita?). Exemplificando o Teorema 2.22 onde a linguagem é infinita se, e somente se, o autômato aceita uma palavra w tal que $n \leq |w| < 2n$, note-se que a menor palavra aceita cujo comprimento maior ou igual a 3 é aabaa a qual possui comprimento 5. Ou seja, $3 \leq |aabaa| < 6$. ❑

Igualdade de Linguagens Regulares

O teorema a seguir mostra que existe um algoritmo para verificar se dois Autômatos Finitos são equivalentes, ou seja, se reconhecem a mesma linguagem.

Teorema 2.23 Igualdade de Linguagens Regulares.

Se M_1 e M_2 são Autômatos Finitos, então existe um algoritmo para determinar se $ACEITA(M_1) = ACEITA(M_2)$.

Prova: Sejam M_1 e M_2 Autômatos Finitos tais que $ACEITA(M_1) = L_1$ e $ACEITA(M_2) = L_2$. Portanto, é possível construir um Autômato Finito M_3 tal que $ACEITA(M_3) = L_3$ onde:

$$L_3 = (L_1 \cap L_2') \cup (L_1' \cap L_2)$$

Claramente, $L_1 = L_2$ se, e somente se, L_3 é vazia. Como visto no Teorema 2.22, existe um algoritmo para verificar se uma Linguagem Regular é vazia. ❑

Eficiência de um Autômato Finito como Algoritmo de Reconhecimento

A implementação de um simulador de Autômato Finito Determinístico (sugerida como exercício) consiste, basicamente, em um algoritmo que controla a mudança de estado do autômato a cada símbolo lido da entrada. Assim, o tempo de processamento necessário para aceitar ou rejeitar é diretamente proporcional ao tamanho da entrada. Em termos de Complexidade (parte da Teoria da Computação que estuda os recursos necessários ao processamento), este procedimento pertence à mais rápida classe de algoritmos.

Deve-se destacar que o tempo de processamento não depende do autômato de reconhecimento. Ou seja, qualquer Autômato Finito Determinístico que reconheça a linguagem terá a mesma eficiência. Entretanto, uma otimização possível é a redução do número de estados. Conforme é visto adiante, existe um algoritmo para construir o Autômato Finito Determinístico mínimo (com o menor número de estados) de qualquer Linguagem Regular.

2.8 Minimização de um Autômato Finito

O objetivo da minimização é gerar um Autômato Finito equivalente com o menor número de estados possível. Esta definição de Autômato Mínimo é universalmente aceita e adotada na maioria das soluções práticas. Entretanto, em algumas aplicações especiais, a minimização do número de estados não implica necessariamente no menor custo de implementação. Um exemplo típico seria o desenho de circuitos eletrônicos, quando pode ser desejável introduzir estados intermediários em determinadas posições (do circuito), de forma a melhorar a eficiência, ou simplesmente facilitar as ligações físicas. Portanto, o algoritmo de minimização nestes casos deve ser modificado, prevendo as variáveis específicas da aplicação.

O Autômato Finito Mínimo é único (a menos de isomorfismo). Assim, dois autômatos distintos que aceitam a mesma linguagem ao serem minimizados geram o mesmo Autômato Mínimo, diferenciando-se, eventualmente, na identificação dos estados. Neste caso, os conjuntos dos estados são isomorfos.

Basicamente, o algoritmo de minimização unifica os estados equivalentes. Dois estados q e p são ditos equivalentes se, e somente se, para qualquer palavra w pertencente a Σ^*, $\delta(q, w)$ e $\delta(p, w)$ resultam simultaneamente em estados finais, ou não-finais. Ou seja, o processamento de uma entrada qualquer a partir de estados equivalentes gera, em qualquer caso, o mesmo resultado aceita/rejeita.

Na definição que segue, lembre-se que, para um dado conjunto A, #A denota o cardinal de A.

Definição 2.24 Autômato Mínimo.

Um *Autômato Mínimo* de uma Linguagem Regular L é um Autômato Finito Determinístico $M = (\Sigma, Q, \delta, q_0, F)$ tal que ACEITA(M) = L e que, para qualquer outro Autômato Finito Determinístico $M' = (\Sigma, Q', \delta', q_0', F')$ tal que ACEITA(M') = L, tem-se que $\#Q' \geq \#Q$. ❐

Definição 2.25 Pré-Requisitos do Algoritmo de Minimização.

Um Autômato Finito a ser minimizado deve satisfazer aos seguintes pré-requisitos:

a) Deve ser determinístico;

b) Não pode ter estados inacessíveis (não-atingíveis a partir do estado inicial);

c) A função programa deve ser total (a partir de qualquer estado são previstas transições para todos os símbolos do alfabeto). ❐

Caso o autômato não satisfaça algum dos pré-requisitos, é necessário gerar um autômato equivalente, como segue:

a) Gerar um autômato determinístico equivalente, usando os algoritmos introduzidos nas correspondentes demonstrações dos teoremas (de AFε para AFN e de AFN para AFD);

b) Eliminar os estados inacessíveis e suas correspondentes transições. Trata-se de um algoritmo relativamente simples e é sugerido como exercício;

c) Para transformar a função programa em total, é suficiente introduzir um novo estado não-final d e incluir as transições não-previstas, tendo d como estado destino. Por fim, incluir um ciclo em d para todos os símbolos do alfabeto.

O algoritmo apresentado a seguir identifica os estados equivalentes por exclusão. A partir de uma tabela de estados, são marcados os estados não-equivalentes. Ao final do algoritmo, as referências não-marcadas representam os estados equivalentes.

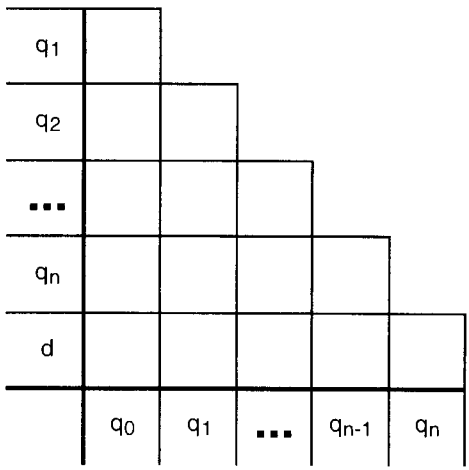


Figura 2.26 Tabela de estados do algoritmo de minimização de Autômatos Finitos

Definição 2.26 Algoritmo de Minimização.

Suponha um Autômato Finito Determinístico $M = (\Sigma, Q, \delta, q_0, F)$ que satisfaz aos pré-requisitos de minimização. Os passos do *Algoritmo de Minimização* são os seguintes:

a) *Tabela*. Construir uma tabela relacionando os estados distintos, onde cada par de estados ocorre somente uma vez, como ilustrado na Figura 2.26;

b) *Marcação dos estados trivialmente não-equivalentes*. Marcar todos os pares do tipo {estado final, estado não-final}, pois, obviamente, estados finais não são equivalentes a não-finais;

c) *Marcação dos estados não-equivalentes*. Para cada par $\{q_u, q_v\}$ não-marcado e para cada símbolo $a \in \Sigma$, suponha que $\delta(q_u, a) = p_u$ e $\delta(q_v, a) = p_v$ e:
   - se $p_u = p_v$, então q_u é equivalente a q_v para o símbolo a e não deve ser marcado;
   - se $p_u \neq p_v$ e o par $\{p_u, p_v\}$ não está marcado, então $\{q_u, q_v\}$ é incluído em uma lista a partir de $\{p_u, p_v\}$ para posterior análise;
   - se $p_u \neq p_v$ e o par $\{p_u, p_v\}$ está marcado, então:
     - $\{q_u, q_v\}$ não é equivalente e deve ser marcado;
     - se $\{q_u, q_v\}$ encabeça uma lista de pares, então marcar todos os pares da lista (e, recursivamente, se algum par da lista encabeça outra lista);

d) *Unificação dos estados equivalentes*. Os estados dos pares não-marcados são equivalentes e podem ser unificados como segue:
   - a equivalência de estados é transitiva;

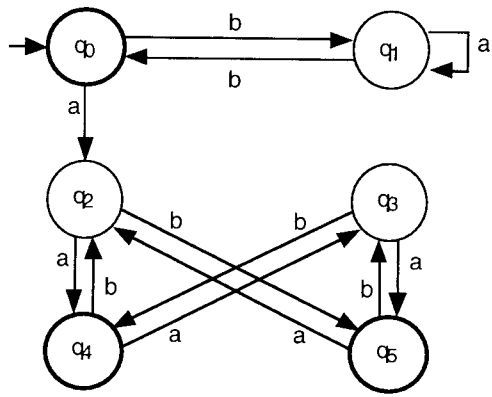


Figura 2.27 Autômato finito a ser minimizado

- pares de estados não-finais equivalentes podem ser unificados como um único estado não-final;
- pares de estados finais equivalentes podem ser unificados como um único estado final;
- se algum dos estados equivalentes é inicial, então o correspondente estado unificado é inicial;

e) *Exclusão dos estados inúteis.* Por fim, os estados chamados inúteis devem ser excluídos. Um estado q é inútil se é não-final e a partir de q não é possível atingir um estado final. Deve-se reparar que o estado d (se incluído) sempre é inútil (o algoritmo para excluir os estados inúteis é simples e é sugerido como exercício). ❑

EXEMPLO 21 Minimização.

Considere o Autômato Finito Determinístico ilustrado na Figura 2.27 (qual a linguagem aceita?). O autômato satisfaz os pré-requisitos de minimização (e, conseqüentemente, não é necessário incluir o estado d). Os passos do algoritmo são como segue:

a) Construção da tabela, como ilustrado na Figura 2.28;

b) Marcação dos pares do tipo {estado final, estado não-final}, como ilustrado na Figura 2.28;

c) Análise dos pares de estado não-marcados, onde a tabela resultante é ilustrada na Figura 2.29, sendo o símbolo ⊗ usado para representar os pares marcados neste etapa:

c.1) Análise do par $\{q_0, q_4\}$:

$$\delta(q_0, a) = q_2 \qquad \delta(q_0, b) = q_1$$
$$\delta(q_4, a) = q_3 \qquad \delta(q_4, b) = q_2$$

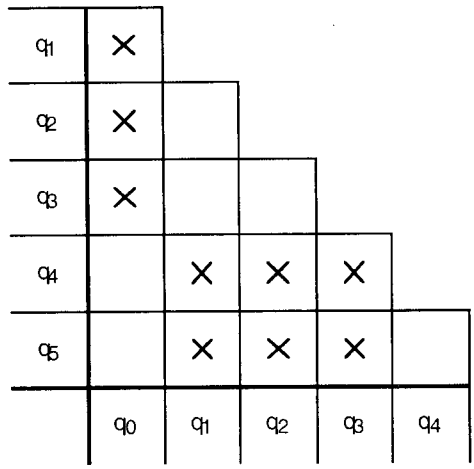


Figura 2.28 Tabela de estados e os pares do tipo {estados finais, estados não-finais} marcados

Como $\{q_1, q_2\}$ e $\{q_2, q_3\}$ são não-marcados, então $\{q_0, q_4\}$ é incluído nas listas encabeçadas por $\{q_1, q_2\}$ e $\{q_2, q_3\}$;

c.2) Análise do par $\{q_0, q_5\}$:

$\delta(q_0, a) = q_2$ $\quad$ $\delta(q_0, b) = q_1$
$\delta(q_5, a) = q_2$ $\quad$ $\delta(q_5, b) = q_3$

Como $\{q_1, q_3\}$ é não-marcado (e como $\{q_2, q_2\}$ é trivialmente equivalente), então $\{q_0, q_5\}$ é incluído na lista encabeçada por $\{q_1, q_3\}$;

c.3) Análise do par $\{q_1, q_2\}$:

$\delta(q_1, a) = q_1$ $\quad$ $\delta(q_1, b) = q_0$
$\delta(q_2, a) = q_4$ $\quad$ $\delta(q_2, b) = q_5$

Como $\{q_1, q_4\}$ é marcado, então $\{q_1, q_2\}$ também é marcado.
Como $\{q_1, q_2\}$ encabeça uma lista, o par $\{q_0, q_4\}$ também é marcado;

c.4) Análise do par $\{q_1, q_3\}$:

$\delta(q_1, a) = q_1$ $\quad$ $\delta(q_1, b) = q_0$
$\delta(q_3, a) = q_5$ $\quad$ $\delta(q_3, b) = q_4$

Como $\{q_1, q_5\}$ bem como $\{q_0, q_4\}$ são marcados, então $\{q_1, q_3\}$ também é marcado.
Como $\{q_1, q_3\}$ encabeça uma lista, o par $\{q_0, q_5\}$ também é marcado;

c.5) Análise do par $\{q_2, q_3\}$:

$\delta(q_2, a) = q_4$ $\quad$ $\delta(q_2, b) = q_5$
$\delta(q_3, a) = q_5$ $\quad$ $\delta(q_3, b) = q_4$

Como $\{q_4, q_5\}$ é não-marcado, então $\{q_2, q_3\}$ é incluído na lista encabeçada por $\{q_4, q_5\}$;

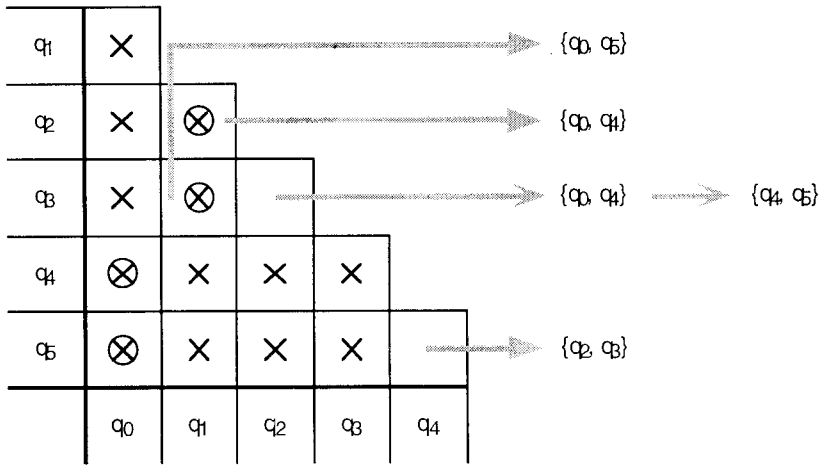


Figura 2.29 Tabela de estados resultante

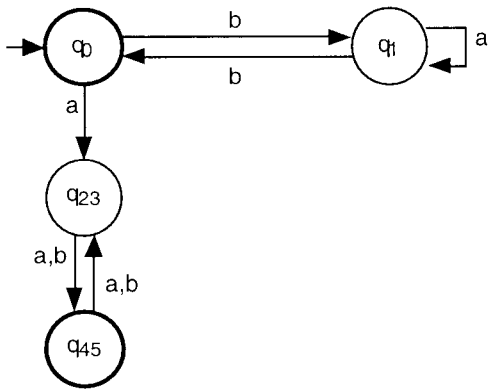


Figura 2.30 Autômato finito mínimo

c.6) Análise do par $\{q_4, q_5\}$:

$\delta(q_4, a) = q_3$ $\quad$ $\delta(q_4, b) = q_2$

$\delta(q_5, a) = q_2$ $\quad$ $\delta(q_5, b) = q_3$

Como $\{q_2, q_3\}$ é não-marcado, então $\{q_4, q_5\}$ é incluído na lista encabeçada por $\{q_2, q_3\}$;

d) Como os pares $\{q_2, q_3\}$ e $\{q_4, q_5\}$ são não-marcados, as seguintes unificações podem ser feitas:

- q_{23} representa a unificação dos estados não-finais q_2 e q_3;
- q_{45} representa a unificação dos estados finais q_4 e q_5.

O Autômato Mínimo resultante possui quatro estados e é ilustrado na Figura 2.30. ❑

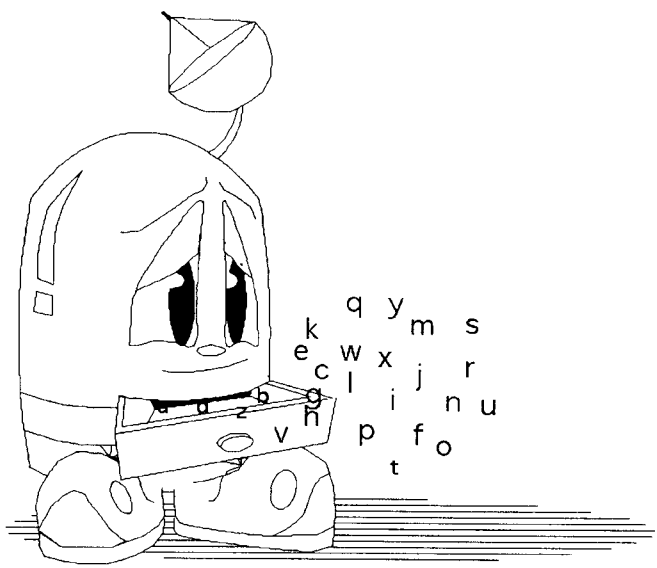
Os seguintes teoremas garantem que o Autômato Mínimo de uma linguagem é o construído pelo algoritmo de minimização apresentado e é único. Os teoremas não serão demonstrados.

Teorema 2.27 Autômato Mínimo.

O Autômato Finito Determinístico construído usando o algoritmo de minimização apresentado é o autômato com menor número de estados para a linguagem. ❑

Teorema 2.28 Unicidade do Autômato Mínimo.

O Autômato Finito Determinístico mínimo de uma linguagem é único, a menos de isomorfismo. ❑



2.9 Autômato Finito com Saída

O conceito básico de Autômato Finito possui aplicações restritas, pois a informação de saída é limitada à lógica binária aceita/rejeita. Sem alterar a classe de linguagens reconhecidas, é possível estender a definição de Autômato Finito incluindo a geração de uma palavra de saída. As saídas podem ser associadas às transições (Máquina de Mealy) ou aos estados (Máquina de Moore). Em ambas as máquinas, a saída não pode ser lida, ou seja, não pode ser usada como memória auxiliar, e é como segue:

- é definida sobre um alfabeto especial, denominado Alfabeto de Saída (pode ser igual ao alfabeto de entrada);
- a saída é armazenada em uma fita independente da de entrada;
- a cabeça da fita de saída move uma célula para direita a cada símbolo gravado;
- o resultado do processamento do Autômato Finito é o seu estado final (condição de aceita/rejeita) e a informação contida na fita de saída.

As Máquinas de Mealy e Moore são modificações sobre o Autômato Finito Determinístico. A extensão da definição, prevendo as facilidades de Não-Determinismo e Movimentos Vazios, é sugerida como exercício.

2.9.1 Máquina de Mealy

A Máquina de Mealy é um Autômato Finito modificado de forma a gerar uma palavra de saída para cada transição.

Definição 2.29 Máquina de Mealy.

Uma *Máquina de Mealy* M é Autômato Finito Determinístico com saídas associadas às transições. É representada por uma 6-upla:

$$M = (\Sigma, Q, \delta, q_0, F, \Delta)$$

onde:

Σ *alfabeto de símbolos de entrada*;

Q *conjunto de estados* possíveis do autômato o qual é finito;

δ *função programa* ou *função de transição*:

$$\delta: Q \times \Sigma \rightarrow Q \times \Delta^*$$

a qual é uma função parcial;

q_0 *estado inicial* do autômato tal que q_0 é elemento de Q;

F *conjunto de estados finais* tal que F está contido em Q;

Δ *alfabeto de símbolos de saída*. ❑

Portanto, as componentes Σ, Q, q_0 e F são como no Autômato Finito Determinístico. A função programa pode ser representada como um grafo finito direto como nos AFD, adicionando, na etiqueta de cada transição, a saída associada, quando diferente da palavra vazia.

O processamento de uma Máquina de Mealy, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa para cada símbolo de w (da esquerda para a direita) até ocorrer uma condição de parada. A palavra vazia como saída da função programa indica que nenhuma gravação é realizada e, obviamente, não move a cabeça da fita de saída. Se todas as transições geram saída vazia, então a Máquina de Mealy processa

como se fosse um Autômato Finito. A definição formal da função programa estendida de uma Máquina de Mealy é sugerida como exercício.

EXEMPLO 22 Máquina de Mealy.

Uma aplicação comum e freqüentemente recomendada para os autômatos com saída é o projeto de diálogo entre um programa (de computador) e o seu usuário. Basicamente, um diálogo pode ser de dois tipos:

- comandado pelo programa;
- comandado pelo usuário.

Em qualquer caso, uma das principais dificuldades do projetista é a visualização do conjunto de eventos e ações que definam um diálogo adequado para as diversas situações possíveis.

O exemplo que segue é uma Máquina de Mealy que trata algumas situações típicas de um diálogo que cria e atualiza arquivos. A seguinte simbologia é adotada no grafo da função de transição:

- ⟨...⟩: entrada fornecida pelo usuário (em um teclado, por exemplo);
- "...": saída gerada pelo programa (em um vídeo, por exemplo);
- [...]: ação interna ao programa, sem comunicação com o usuário;
- (...): resultado de uma ação interna ao programa; é usado como entrada no grafo.

A Máquina de Mealy é $M = (\Sigma, \{ q_0, q_1, ..., q_8, q_f \}, \delta, q_0, \{ q_f \}, \Delta)$ como ilustrada na Figura 2.31, onde $\Sigma = \Delta$ e representam o conjunto de símbolos (palavras do português) válidos no diálogo. ❑

2.9.2 Máquina de Moore

A Máquina de Moore possui uma segunda função, que gera uma palavra de saída (que pode ser vazia) para cada estado da máquina.

Definição 2.30 Máquina de Moore.

Uma *Máquina de Moore* M é um Autômato Finito Determinístico com saídas associadas aos estados. É representada por uma 7-upla:

$$M = (\Sigma, Q, \delta, q_0, F, \Delta, \delta_S)$$

onde:

Σ *alfabeto de símbolos de entrada*;

Q *conjunto de estados* possíveis do autômato o qual é finito;

δ *função programa* ou *função de transição*:

$$\delta: Q \times \Sigma \rightarrow Q$$

a qual é uma função parcial;

q_0 *estado inicial* tal que q_0 é elemento de Q;

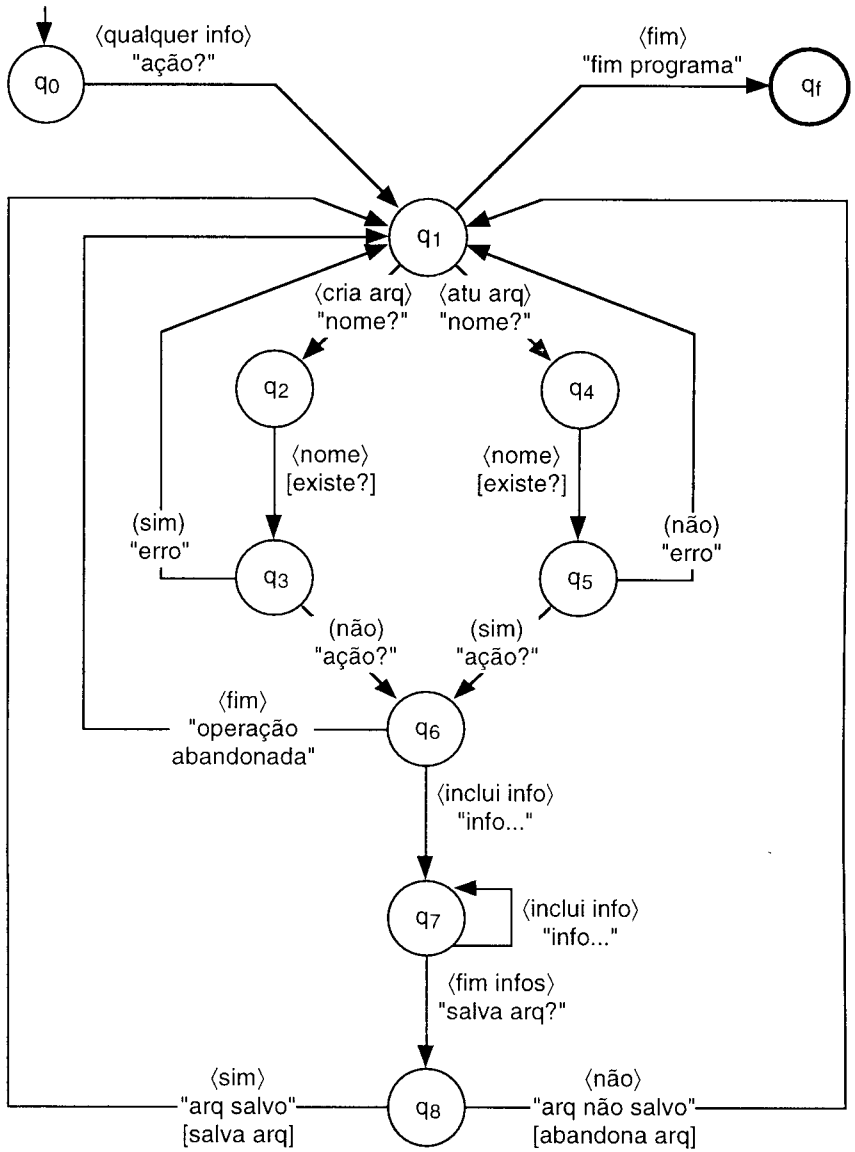


Figura 2.31 Grafo da Máquina de Mealy

F *conjunto de estados finais* tal que F está contido em Q;

Δ *alfabeto de símbolos de saída*;

δ_S *função de saída*:

$$\delta_S: Q \rightarrow \Delta^*$$

a qual é uma função total. ❑

Portanto, as componentes Σ, Q, δ, q_0 e F são como no Autômato Finito Determinístico e Δ é como na Máquina de Mealy. A função programa pode ser representada como um grafo finito direto como nos AFD, adicionando, na etiqueta de cada estado, a saída associada, quando diferente da palavra vazia.

O processamento de uma Máquina de Moore, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa para cada símbolo de w (da esquerda para a direita) até ocorrer uma condição de parada. A palavra vazia resultado da função de saída indica que nenhuma gravação é realizada e, obviamente, não move a cabeça da fita de saída. Se todos os estados geram saída vazia, então a Máquina de Moore processa como se fosse um Autômato Finito. A definição formal da função programa estendida de uma Máquina de Moore é sugerida como exercício.

EXEMPLO 23 Máquina de Moore.

Um exemplo comum de aplicação do conceito de Máquina de Moore é o desenvolvimento de Analisadores Léxicos de compiladores ou tradutores de linguagens em geral. Basicamente, um analisador léxico é um Autômato Finito (em geral, determinístico) que identifica os componentes básicos da linguagem como, por exemplo, números, identificadores, separadores, etc. Uma Máquina de Moore como um Analisador Léxico é como segue:

- um estado final é associado a cada unidade léxica;
- cada estado final possui uma saída (definida pela Função de Saída) que descreve ou codifica a unidade léxica identificada;
- para os demais estados (não-finais) a saída gerada é a palavra vazia. ❑

2.9.3 Equivalência das Máquina de Moore e Mealy

A equivalência dos dois modelos de Autômato Finito com Saída não é válida para a entrada vazia. Neste caso, enquanto a Máquina de Moore gera a palavra correspondente ao estado inicial, a Máquina de Mealy não gera qualquer saída, pois não executa transição alguma. Entretanto, para os demais casos, a equivalência pode ser facilmente mostrada.

Teorema 2.31 Máquina de Moore → Máquina de Mealy.

Toda Máquina de Moore pode ser simulada por uma Máquina de Mealy, para entradas não vazias.

Prova: Suponha $MO = (\Sigma, Q, \delta_{MO}, q_0, F, \Delta, \delta_S)$, uma Máquina de Moore qualquer. Seja:

$$ME = (\Sigma, Q \cup \{q_e\}, \delta_{ME}, q_e, F, \Delta)$$

uma Máquina de Mealy onde a função δ_{ME} é definida como segue (suponha q um estado de Q e a um símbolo de Σ):

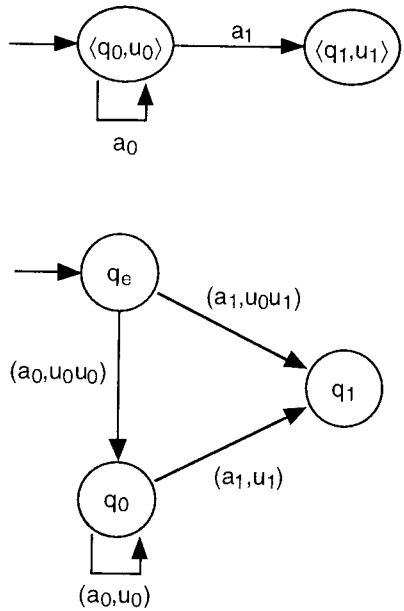


Figura 2.32 Simulação da Máquina de Moore (acima) usando a Máquina de Mealy (abaixo)

a) $\delta_{ME}(q_e, a) = (\delta_{MO}(q_0, a), \delta_S(q_0)\delta_S(\delta_{MO}(q_0, a)))$

b) $\delta_{ME}(q, a) = (\delta_{MO}(q, a), \delta_S(\delta_{MO}(q, a)))$

Em b), é construída a função programa da Máquina de Mealy, a partir das funções de transição e de saída da Máquina de Moore. O estado q_e introduzido em a) é referenciado somente na primeira transição a ser executada. Seu objetivo é garantir a geração da saída referente ao estado inicial q_0 de Moore (lembre-se que Mealy necessita executar a transição para gerar a saída), como ilustrado na Figura 2.32.

Uma indução em $n > 0$, prova que, ao reconhecer a entrada $a_1...a_n$, se MO passa pelos estados $q_0, q_1, ..., q_n$ e gera as saídas $u_0, u_1, ..., u_n$ então ME passa pelos estados $q_e, q_0, q_1, ..., q_n$ e gera as saídas $u_0u_1, ..., u_n$. ❑

Teorema 2.32 Máquina de Mealy → Máquina de Moore.

Toda Máquina de Mealy pode ser simulada por uma Máquina de Moore.

Prova: Suponha $ME = (\Sigma, Q, \delta_{ME}, q_0, F, \Delta)$, uma Máquina de Mealy qualquer. Seja $S(\delta_{ME})$ a imagem da função programa δ_{ME} restrita à componente da palavra de saída (ou seja, o conjunto de todas as saídas possíveis de ME).

A construção da Máquina de Moore:

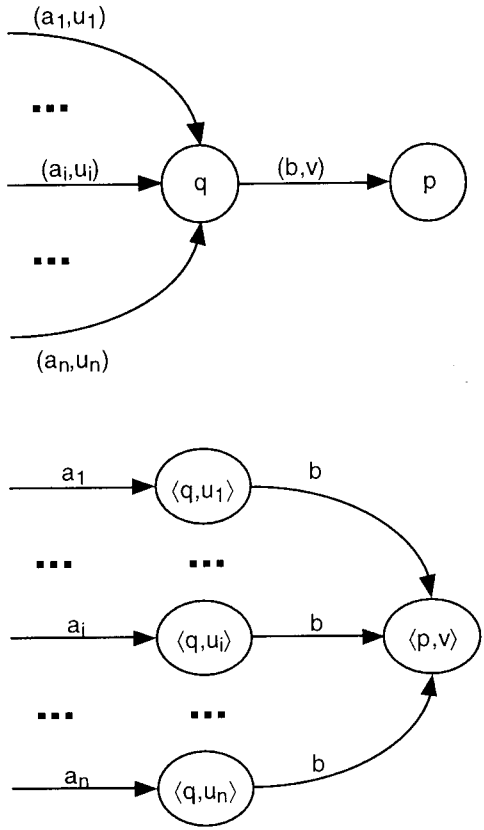


Figura 2.33 Máquina de Moore (abaixo) que simula uma Máquïna de Mealy (acima)

$$MO = (\Sigma, (Q \times S(\delta_{ME})) \cup \{\langle q_0, \varepsilon\rangle\}, \delta_{MO}, \langle q_0, \varepsilon\rangle, F \times S(\delta_{ME}), \Delta, \delta_S)$$

correspondente determina, em geral, um maior número de estados que a máquina ME que está sendo simulada. Isto é necessário, pois se diversas transições com saídas diferentes atingem um mesmo estado, este necessita ser simulado por diversos estados, um para cada símbolo de saída, ou seja, em Moore os estados são construídos como um par ordenado, onde a segunda componente representa a palavra de saída, como ilustrado na Figura 2.33.

Assim, as funções da Máquina de Moore podem ser construídas a partir de δ_{ME} onde a função programa δ_{MO} é como segue:

- para a em Σ, se $\delta_{ME}(q_0, a) = (q, u)$, então:

$$\delta_{MO}(\langle q_0, \varepsilon\rangle, a) = \langle q, u\rangle$$

- para b em Σ e para q em Q, se $\delta_{ME}(q, b) = (p, v)$, então, para a_i em Σ e para q_i em Q tais que $\delta_{ME}(q_i, a_i) = (q, u_i)$, tem-se que:

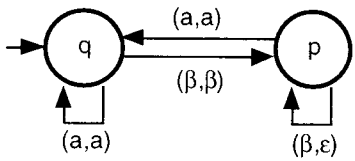


Figura 2.34 Grafo da Máquina de Mealy

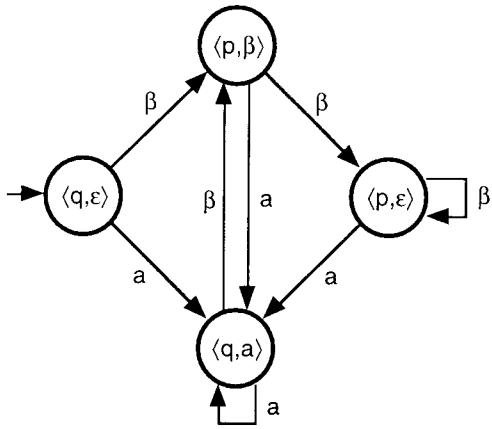


Figura 2.35 Grafo da correspondente Máquina de Moore

$$\delta_{MO}(\langle q, u_i\rangle, b) = \langle p, v\rangle$$

e onde a função de saída δ_S é tal que, para o estado $\langle q, u\rangle$ de MO:

$$\delta_S(\langle q, u\rangle) = u$$

Uma indução em n, prova que, ao reconhecer a entrada $a_1...a_n$, se ME passa pelos estados $q_0, q_1, ..., q_n$ e gera as saídas $u_1, ..., u_n$ então MO passa pelos estados $\langle q_0, \varepsilon\rangle, \langle q_1, u_1\rangle, ..., \langle q_n, u_n\rangle$ e gera as saídas $\varepsilon, u_1, ..., u_n$. ❑

EXEMPLO 24 Máquina de Moore Simulando uma Máquina de Mealy.

Considere a Máquina de Mealy ME = ({ a, β }, { q, p }, δ_{ME}, q, { q, p }, { a, β }) que compacta os brancos de um texto onde a representa um símbolo qualquer do texto e β o símbolo branco, como ilustrado na Figura 2.34 (na etiqueta de uma transição, a primeira componente representa o símbolo lido e a segunda a palavra gravada).

A Máquina de Moore construída conforme a prova do Teorema 2.32 é MO = ({ a, β }, Q, δ_{MO}, ⟨q, ε⟩, F, { a, β }, δ_S) tal que Q = F = { q, p }×{ ε, a, β }, ilustrada na Figura 2.35 onde a segunda componente de cada estado representa a saída. ❑

2.10 Exercícios

2.1 Sobre as Linguagens Regulares:

a) Qual a importância do seu estudo?

b) Exemplifique suas aplicações (para os diversos formalismos);

c) Você imagina algum tipo de linguagem cujo algoritmo de reconhecimento seja mais eficiente que o das Regulares? E menos eficiente? Explique a sua resposta.

2.2 Desenvolva Autômatos Finitos Determinísticos que reconheçam as seguintes linguagens sobre $\Sigma = \{a, b\}$:

a) $\{w \mid w$ possui aaa como subpalavra$\}$

b) $\{w \mid$ o sufixo de w é aa$\}$

c) $\{w \mid w$ possui número ímpar de a e b$\}$

d) $\{w \mid w$ possui número par de a e ímpar de b ou w possui número par de b e ímpar de a$\}$

e) $\{w \mid$ o quinto símbolo da direita para a esquerda de w é a$\}$

2.3 Desenvolva Autômatos Finitos Não-Determinísticos, com ou sem movimentos vazios, que reconheçam as seguintes linguagens:

a) sobre o alfabeto $\Sigma = \{a, b, c\}$:

$$\{w \mid aa \text{ ou } bb \text{ é subpalavra e } cccc \text{ é sufixo de } w\}$$

b) sobre o alfabeto $\Sigma = \{a, b\}$:

b.1) $\{w_1 w_2 w_1 \mid w_2$ é qualquer e $\mid w_1 \mid = 3\}$

b.2) $\{w \mid$ o décimo símbolo da direita para a esquerda de w é a$\}$

b.3) $\{w \mid w$ possui igual número de símbolos a e b e (qualquer prefixo de w possui, no máximo, dois a a mais que b ou qualquer prefixo de w possui, no máximo, dois b a mais que a)$\}$

2.4 Desenvolva Expressões e Gramáticas Regulares que gerem as seguintes linguagens sobre $\Sigma = \{a, b\}$:

a) $\{w \mid w$ tem no máximo um par de a como subpalavra e no máximo um par de b como subpalavra$\}$

b) $\{w \mid$ qualquer par de a antecede qualquer par de b$\}$

c) $\{w \mid w$ não possui aba como subpalavra$\}$

2.5 Represente a seguinte linguagem baseada em unidades léxicas da linguagem de programação Pascal (ou alguma outra de seu domínio), usando os formalismos Autômato Finito Determinístico, Expressão Regular e Gramática Regular:

{w | w é número inteiro ou w é número real ou w é identificador da linguagem Pascal}

2.6 Descreva em palavras as linguagens geradas pelas seguintes Expressões Regulares:

a) (aa + b)*(a + bb)

b) (b + ab)*(ε + a)

c) (aa + bb + (aa + bb)(ab + ba)(aa + bb))*

2.7 Aplique o algoritmo de tradução de formalismo de Expressão Regular para Autômato Finito:

a) (ab + ba)*(aa + bb)*

b) ab(abb* + baa*)*ba

2.8 Aplique os algoritmos de tradução de formalismos apresentados e, a partir da Expressão Regular (b + ε)(a + bb)*, realize as diversas etapas até gerar a Gramática Regular correspondente (ER → AFε → AFN → AFD → GR).

2.9 Minimize os Autômatos Finitos ilustrados na Figura 2.36.

2.10 Por que pode-se afirmar que um Autômato Finito Determinístico sempre pára (ao processar qualquer entrada)? O mesmo pode ser afirmado para o não-determinístico? E com movimentos vazios? Em particular, no caso do Autômato Finito com Movimentos Vazios, analise para a seguinte situação de ciclo (suponha que q e p são estados do autômato):

$\delta(q, \varepsilon) = p$
$\delta(p, \varepsilon) = q$

2.11 Complete a prova referente ao teorema: a classe dos Autômatos Finitos com Movimentos Vazios é equivalente à classe dos Autômatos Finitos Não-Determinísticos.

2.12 Demonstre a equivalência dos quatro tipos de Gramáticas Lineares.

2.13 Demonstre as seguintes propriedades das Expressões Regulares (suponha que r, s e t são Expressões Regulares):

a) *Comutatividade da União.*

r + s = s + r

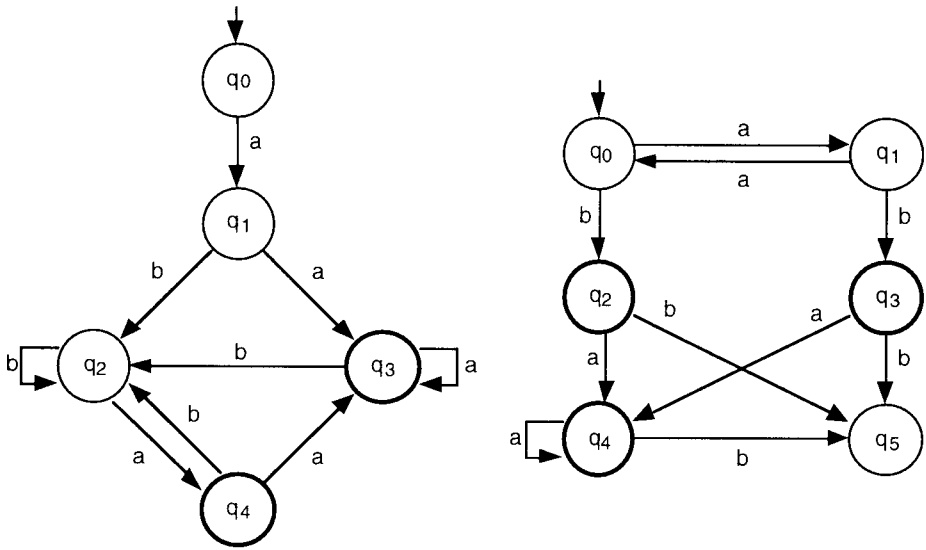


Figura 2.36 Autômatos finitos a serem minimizados

b) *Associatividade da União.*
$(r + s) + t = r + (s + t)$

c) *Não-Comutatividade da Concatenação.*
$rs = sr$ pode não ser satisfeita

d) *Associatividade da Concatenação.*
$(rs)t = r(st)$

e) *Distributividade (à Esquerda e à Direita) da Concatenação sobre a União.*
$t(r + s) = tr + ts$
$(r + s)t = rt + st$

f) *Dupla Concatenação Sucessiva.*
$(r^*)^* = r^*$

g) *Outras Propriedades da Concatenação Sucessiva.*
$(r + \varepsilon)^* = r^*$
$(r^*s^*)^* = (r + s)^*$

2.14 Prove que as seguintes linguagens não são Regulares:

a) $\{ww \mid w \text{ é palavra de } \{a, b\}^*\}$

b) $\{a^n b^n \mid n \geq 0\}$

2.15 Desenvolva um algoritmo que elimine os estados inacessíveis de um Autômato Finito Determinístico. Um estado é dito inacessível se for não-atingível a partir do estado inicial.

2.16 Desenvolva algoritmos mais otimizados que os apresentados (e compare a eficiência) para determinar se uma Linguagem Regular é vazia, finita ou infinita, usando as seguintes sugestões:

a) Vazia, eliminando os estados inacessíveis;

b) Finita, combinando os algoritmos de minimização e o apresentado no Lema do Bombeamento;

c) Infinita, analisando a função programa do autômato minimizado.

2.17 Como pode ser verificada a equivalência de dois autômatos, usando o algoritmo de minimização?

2.18 Defina formalmente a função programa estendida para as Máquinas de Mealy e de Moore.

2.19 Desenvolva uma Máquina de Mealy e uma de Moore que realize a conversão da representação de valores monetários de dólares para reais. Por exemplo, dado o valor US$25,010.59 na fita de entrada, deve ser gravado o valor R$25.010,59 na fita de saída (atenção para o uso da vírgula e do ponto nos dois valores). Adicionalmente, o autômato deve verificar se a entrada é um valor monetário válido.

2.20 Considere o exemplo de diálogo apresentado para a Máquina de Mealy:

a) Desenvolva uma Máquina de Moore que realize o mesmo processamento;

b) Usando o algoritmo que demonstra a equivalência dos dois modelos de autômato com saída construa a Máquina de Moore equivalente;

c) Escolha uma das máquinas, e amplie o diálogo prevendo:
   - operações de modificação e exclusão de informações;
   - facilidade de *help* (opção que fornece ao usuário informações de ajuda de como proceder no diálogo, no ponto em que se encontra).

2.21 Modifique a definição das Máquinas de Mealy e de Moore, como segue:

a) A saída é restrita a um símbolo ou à palavra vazia;

b) Não possui estados finais.

Estas modificações alteram o poder computacional das máquinas?

2.22 Modifique a definição das Máquinas de Mealy e de Moore, incluindo as facilidades de:

a) Não-determinismo;

b) Movimentos vazios.

Para cada caso acima, em relação à equivalência dos dois tipos de máquina, como fica a restrição referente à palavra vazia?

2.23 Modifique o algoritmo de minimização, prevendo que o autômato possua saída, como segue:

a) Minimização da Máquina de Mealy;

b) Minimização da Máquina de Moore.

2.24 Desenvolva um programa em computador que simule o processamento de qualquer Autômato Finito Determinístico, como segue:

- entrada: função de transição δ, estado inicial, conjunto de estados finais e as palavras a serem processadas;
- saída: condição de parada ACEITA/REJEITA e identificação do estado de parada.

2.25 Os seguintes algoritmos também são sugeridos para implementação em computador:

a) Tradução de AFN para AFD;

b) Tradução de AFε para AFN;

c) Tradução de ER para AFε;

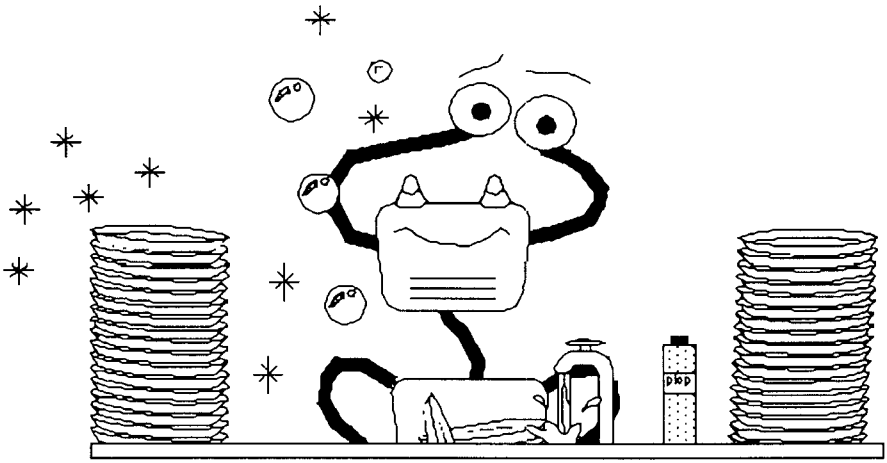
d) Tradução de GR para AFε equivalente;

e) Minimização de AFD.

2.26 Desenvolva um algoritmo que gere em ordem lexicográfica todas as palavras representadas por:

a) Uma Expressão Regular qualquer;

b) Uma Gramática Regular qualquer.



3 Linguagens Livres do Contexto

A *Classe das Linguagens Livres do Contexto* ou *Tipo 2* contém propriamente a Classe das Linguagens Regulares. Seu estudo é de fundamental importância na informática pois:

- compreende um universo mais amplo de linguagens (comparativamente com as regulares) tratando, adequadamente, questões como parênteses balanceados, construções bloco-estruturadas, entre outras, típicas de linguagens de programação como Pascal, C, Algol, etc.;
- os algoritmos reconhecedores e geradores que implementam as Linguagens Livres do Contexto são relativamente simples e possuem uma boa eficiência;
- exemplos típicos de aplicações dos conceitos e resultados referentes às Linguagens Livres do Contexto são analisadores sintáticos, tradutores de linguagens e processadores de texto em geral.

O estudo da Classe das Linguagens Livres do Contexto é desenvolvido a partir de um formalismo axiomático ou gerador (gramática) e um operacional ou reconhecedor (autômato), como segue:

a) *Gramática Livre do Contexto.* Gramática onde as regras de produção são definidas de forma mais livre que na Gramática Regular;

b) *Autômato com Pilha.* Autômato cuja estrutura básica é análoga à do Autômato Finito, adicionando uma memória auxiliar tipo pilha (a qual pode ser lida ou gravada) e a facilidade de não-determinismo.

3.1 Gramática Livre do Contexto

As Linguagens Livres do Contexto são definidas a partir das Gramáticas Livres do Contexto.

Definição 3.1 Gramática Livre do Contexto.

Uma *Gramática Livre do Contexto* (*GLC*) G é uma gramática:

$$G = (V, T, P, S)$$

com a restrição de que qualquer regra de produção de P é da forma $A \rightarrow \alpha$, onde A é uma variável de V e α uma palavra de $(V \cup T)^*$. ❑

Portanto, uma Gramática Livre do Contexto é uma gramática onde o lado esquerdo das produções contém exatamente uma variável.

Definição 3.2 Linguagem Livre do Contexto ou Tipo 2.

Uma linguagem é dita *Linguagem Livre do Contexto* (*LLC*) ou *Tipo 2* se for gerada por uma Gramática Livre do Contexto. ❑

O nome "Livre do Contexto" se deve ao fato de representar a mais geral classe de linguagens cuja produção é da forma $A \rightarrow \alpha$. Ou seja, em uma derivação, a variável A deriva α sem depender ("Livre") de qualquer análise dos símbolos que antecedem ou sucedem A ("Contexto") na palavra que está sendo derivada. Assim, claramente, toda Linguagem Regular é Livre do Contexto. A relação entre as classes de linguagens estudadas até o momento é ilustrada na Figura 3.1.

EXEMPLO 1 Linguagem Livre do Contexto - Duplo Balanceamento.

Considere a linguagem:

$$L_1 = \{a^n b^n \mid n \geq 0\}$$

A seguinte Gramática Livre do Contexto:

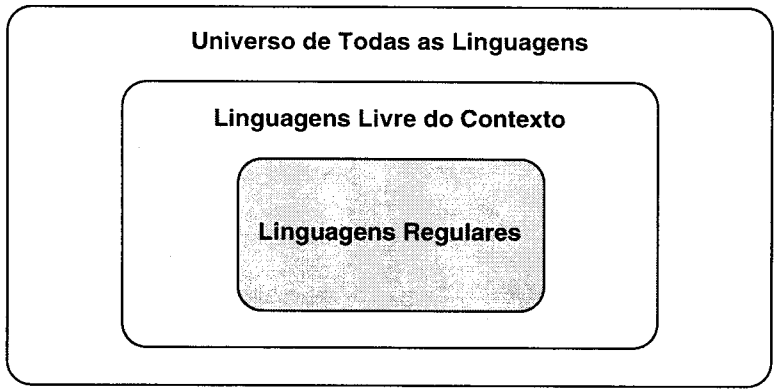


Figura 3.1 Relação entre as classes de linguagens

$G_1 = (\{S\}, \{a, b\}, P_1, S)$, onde:
$P_1 = \{S \rightarrow aSb \mid S \rightarrow \varepsilon\}$

é tal que $GERA(G_1) = L_1$. Por exemplo, a palavra aabb pode ser gerada pela seguinte seqüência de derivação:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb = aabb$$

Esta linguagem é um exemplo clássico e de fundamental importância no estudo das Linguagens Livres do Contexto, pois permite estabelecer analogia entre a^nb^n e linguagens que possuem duplo balanceamento como, por exemplo:

a) Linguagens bloco-estruturadas do tipo $BEGIN^nEND^n$

b) Linguagens com parênteses balanceados na forma $(^n)^n$ ❑

EXEMPLO 2 Linguagem Livre do Contexto - Expressões Aritméticas.

A linguagem L_2 gerada pela Gramática Livre do Contexto abaixo é composta de expressões aritméticas contendo colchetes balanceados, dois operadores e um operando:

$G_2 = (\{E\}, \{+, *, [,], x\}, P_2, E)$, onde:
$P_2 = \{E \rightarrow E+E \mid E*E \mid [E] \mid x\}$

Por exemplo, a expressão [x+x]*x pode ser gerada pela seguinte seqüência de derivação:

$$E \Rightarrow E*E \Rightarrow [E]*E \Rightarrow [E+E]*E \Rightarrow [x+E]*E \Rightarrow [x+x]*E \Rightarrow [x+x]*x$$

É possível gerar a mesma expressão com outra seqüência de derivação? Quantas seqüências distintas são possíveis? ❑

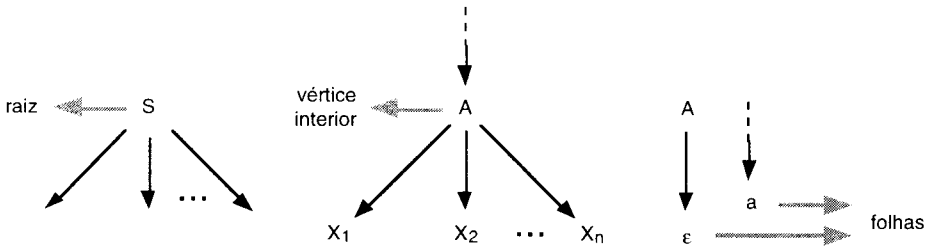


Figura 3.2 Representação de árvore de derivação

3.2 Árvore de Derivação

Em algumas aplicações como compiladores e processadores de textos, freqüentemente é conveniente representar a derivação de palavras na forma de árvore, partindo do símbolo inicial como a raiz, e terminando em folhas de terminais.

Definição 3.3 Árvore de Derivação.

Para uma determinada Gramática Livre do Contexto, a representação da derivação de palavras na forma de árvore, denominada *Árvore de Derivação*, é como segue (considere a Figura 3.2):

a) A *raiz* é o símbolo inicial da gramática;

b) Os *vértices interiores* obrigatoriamente são variáveis. Se A é um vértice interior e $X_1, X_2, ..., X_n$ são os filhos de A, então $A \rightarrow X_1X_2...X_n$ é uma produção da gramática e os vértices $X_1, X_2, ..., X_n$ estão ordenados da esquerda para a direita;

c) Um *vértice folha* é um símbolo terminal, ou o símbolo vazio. Neste caso, o vazio é o único filho de seu pai ($A \rightarrow \varepsilon$). ❑

EXEMPLO 3 Árvore de Derivação.

A palavra aabb e a expressão [x+x]*x dos Exemplos 1 e 2, são geradas pelas árvores de derivação ilustradas na Figura 3.3, à esquerda e à direita, respectivamente. ❑

Uma única árvore de derivação pode representar derivações distintas de uma mesma palavra.

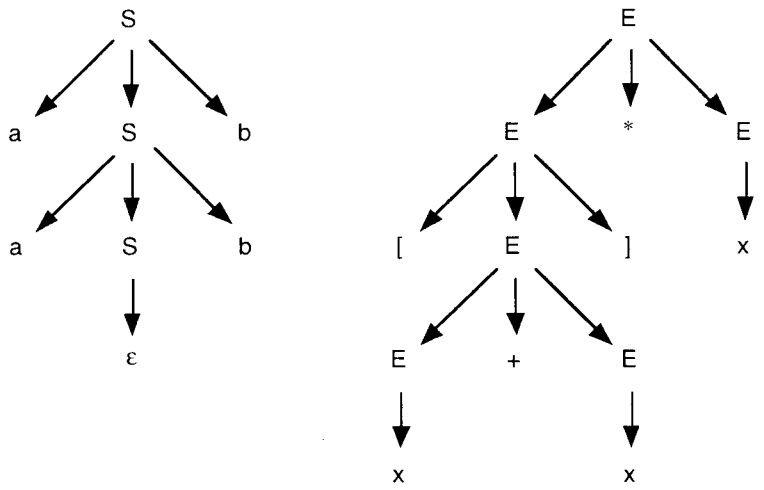


Figura 3.3 Árvores de derivação

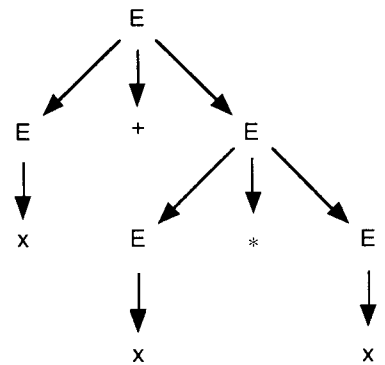


Figura 3.4 Árvore de derivação

EXEMPLO 4 Árvore de Derivação × Derivações.

Na árvore representada na Figura 3.4, a palavra x+x*x pode ser gerada por diversas derivações, como segue:

a) $E \Rightarrow E+E \Rightarrow x+E \Rightarrow x+E*E \Rightarrow x+x*E \Rightarrow x+x*x$

b) $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*x \Rightarrow E+x*x \Rightarrow x+x*x$

c) $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow x+E*E \Rightarrow x+x*E \Rightarrow x+x*x$

d) etc... ❑

Definição 3.4 Derivação mais à Esquerda (Direita).

A *Derivação mais à Esquerda (Direita)* de uma árvore de derivação é a seqüência de produção aplicada sempre à variável mais à esquerda (direita). ❑

No Exemplo 4, a) representa uma derivação mais à esquerda, e b) mais à direita.

3.3 Ambigüidade

Eventualmente, uma mesma palavra pode ser associada a duas ou mais árvores de derivação, determinando uma ambigüidade. Em muitas aplicações como, por exemplo, no desenvolvimento e otimização de alguns tipos de algoritmos de reconhecimento, é conveniente que a gramática usada não seja ambígua. Entretanto, nem sempre é possível eliminar ambigüidades. Na realidade, é fácil definir linguagens para as quais qualquer Gramática Livre do Contexto é ambígua.

Definição 3.5 Gramática Ambígua.

Uma Gramática Livre do Contexto é dita uma *Gramática Ambígua*, se existe uma palavra que possua duas ou mais árvores de derivação. ❑

EXEMPLO 5 Gramática Ambígua.

Relativamente ao Exemplo 4, a palavra x+x*x pode ser gerada por árvores distintas, como ilustrado na Figura 3.5. Portanto, a gramática em questão é ambígua (tente desenvolver uma gramática não-ambígua para esta linguagem). ❑

Note-se que, no Exemplo 4, a palavra x+x*x possui mais de uma derivação à esquerda (direita), como segue:

a) Derivação mais à esquerda:

$E \Rightarrow E+E \Rightarrow x+E \Rightarrow x+E*E \Rightarrow x+x*E \Rightarrow x+x*x$

$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow x+E*E \Rightarrow x+x*E \Rightarrow x+x*x$

b) Derivação mais à direita:

$E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*x \Rightarrow E+x*x \Rightarrow x+x*x$

$E \Rightarrow E*E \Rightarrow E*x \Rightarrow E+E*x \Rightarrow E+x*x \Rightarrow x+x*x$

Prova-se que, uma forma equivalente de definir ambigüidade de uma gramática é a existência de uma palavra com duas ou mais derivações mais à esquerda (direita).

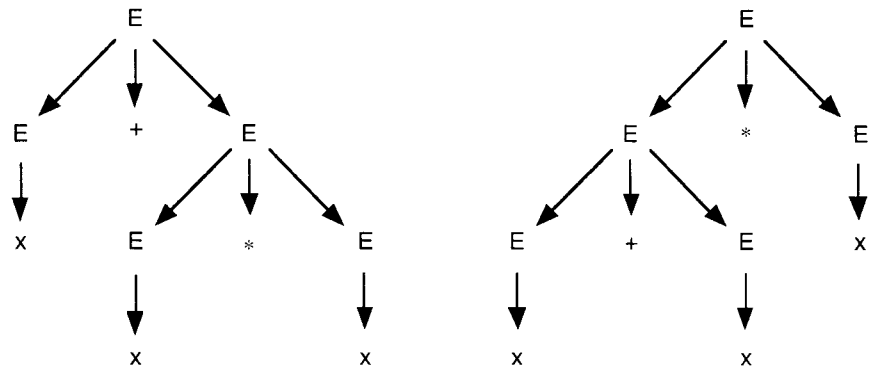


Figura 3.5 Ambigüidade: árvores diferentes para uma mesma palavra

Definição 3.6 Linguagem Inerentemente Ambígua.

Uma linguagem é uma *Linguagem Inerentemente Ambígua* se qualquer Gramática Livre do Contexto que a define é ambígua. ❑

EXEMPLO 6 Linguagem Inerentemente Ambígua.

A linguagem:

$$\{ w \mid w = a^n b^n c^m d^m \text{ ou } w = a^n b^m c^m d^n, n \geq 1, m \geq 1 \}$$

é inerentemente ambígua. ❑

3.4 Simplificação de Gramáticas Livres do Contexto

É possível simplificar alguns tipos de produções sem reduzir o poder de geração das Gramáticas Livres do Contexto. Em geral, as simplificações de gramáticas são usadas na construção e otimização de algoritmos e na demonstração de teoremas.

As seguintes simplificações são apresentadas:

- exclusão de variáveis ou terminais não-usados para gerar palavras;
- exclusão de produções vazias da forma $A \rightarrow \varepsilon$ (se a palavra vazia pertence à linguagem, é incluída uma produção vazia específica para tal fim);
- exclusão de produções da forma $A \rightarrow B$, ou seja, que simplesmente substituem uma variável por outra e, conseqüentemente, não adicionam qualquer informação de geração de palavras.

No texto que segue, são omitidas as provas de que os algoritmos introduzidos, de fato, atingem os objetivos propostos.

Símbolos Inúteis

A exclusão de símbolos inúteis (não-usados na geração de palavras de terminais) é realizada excluindo as produções que fazem referência a estes símbolos, bem como os próprios símbolos inúteis. Não é necessária qualquer modificação adicional nas produções da gramática. O algoritmo apresentado é dividido em duas etapas, como segue:

a) *Qualquer variável gera palavra de terminais.* O algoritmo gera um novo conjunto de variáveis, analisando as produções da gramática a partir de terminais gerados. Inicialmente, considera todas as variáveis que geram terminais diretamente (exemplo: $A \rightarrow a$). A seguir, sucessivamente são adicionadas as variáveis que geram palavras de terminais indiretamente (exemplo: $B \rightarrow Ab$);

b) *Qualquer símbolo é atingível a partir do símbolo inicial.* Após a execução da etapa acima, o algoritmo analisa as produções da gramática a partir do símbolo inicial. Inicialmente, considera exclusivamente o símbolo inicial. Após, sucessivamente as produções da gramática são aplicadas e os símbolos referenciados adicionados aos novos conjuntos.

Definição 3.7 Algoritmo para Exclusão dos Símbolos Inúteis.

Considere uma Gramática Livre do Contexto $G = (V, T, P, S)$. O *Algoritmo para Exclusão dos Símbolos Inúteis* é composto por duas etapas, como segue:

a) *Etapa 1: garante que qualquer variável gera terminais.* A gramática resultante desta etapa é:

$$G_1 = (V_1, T, P_1, S)$$

onde V_1 é construído como segue:

$V_1 = \emptyset$;
repita
$V_1 = V_1 \cup \{ A \mid A \rightarrow \alpha \in P \text{ e } \alpha \in (T \cup V_1)^* \}$
até que o cardinal de V_1 não aumente;

O conjunto P_1 possui os mesmos elementos que P excetuando-se as produções cujas variáveis não pertencem a V_1;

b) *Etapa 2: garante que qualquer símbolo é atingível a partir do símbolo inicial.* A gramática resultante desta etapa é:

$$G_2 = (V_2, T_2, P_2, S)$$

onde V_2 e T_2 são construídos como segue:

$T_2 = \varnothing$;
$V_2 = \{ S \}$;
repita
$V_2 = V_2 \cup \{ A \mid X \rightarrow \alpha A \beta \in P_1, X \in V_2 \}$;
$T_2 = T_2 \cup \{ a \mid X \rightarrow \alpha a \beta \in P_1, X \in V_2 \}$
até que os cardinais de V_2 e T_2 não aumentem;

O conjunto P_2 possui os mesmos elementos que P_1 excetuando-se as produções cujos símbolos não pertencem a V_2 ou T_2. ❑

Deve-se reparar que se as etapas acima forem executadas em ordem inversa (etapa 2 antes da 1), o algoritmo pode não gerar o resultado desejado. Para demonstrar, é suficiente apresentar um contra-exemplo, o que se sugere como exercício (lembre-se que, uma demonstração por contra-exemplo é, de fato, uma demonstração por absurdo).

EXEMPLO 7 Exclusão dos Símbolos Inúteis.

Considere a seguinte Gramática Livre do Contexto:

$G = (\{ S, A, B, C \}, \{ a, b, c \}, P, S)$, onde:
$P = \{ S \rightarrow aAa \mid bBb, A \rightarrow a \mid S, C \rightarrow c \}$

A exclusão dos símbolos inúteis é como segue:

a) *Qualquer variável gera palavra de terminais.* A coluna "iteração" representa o número de ciclos do comando repita-até, e a coluna "variáveis" o conjunto de variáveis construído após a iteração:

iteração	variáveis
início	$\varnothing$
1	$\{A, C\}$
2	$\{A, C, S\}$
3	$\{A, C, S\}$

O algoritmo pára na terceira iteração, pois nenhuma variável foi adicionada ao conjunto. A produção $S \rightarrow bBb$ é excluída, pois B não pertence ao novo conjunto de variáveis;

b) *Qualquer símbolo é atingido a partir de S.*

iteração	variáveis	terminais
início	$\{S\}$	$\varnothing$
1	$\{S, A\}$	$\{a\}$
2	$\{S, A\}$	$\{a\}$

A produção $C \rightarrow c$ é excluída, pois C e c não pertencem aos novos conjuntos de variáveis e terminais, respectivamente. A gramática resultante é a seguinte:

$G = (\{S, A\}, \{a\}, P, S)$, onde:

$P = \{S \rightarrow aAa, A \rightarrow a \mid S\}$ ❑

Produções Vazias

A exclusão de produções vazias (da forma $A \rightarrow \varepsilon$) pode determinar modificações diversas nas produções da gramática. O algoritmo é dividido em três etapas, como segue:

a) *Variáveis que constituem produções vazias.* Considera, inicialmente, todas as variáveis que geram diretamente ε (exemplo: $A \rightarrow \varepsilon$). A seguir, sucessivamente são determinadas as variáveis que indiretamente geram ε (exemplo: $B \rightarrow A$);

b) *Exclusão de produções vazias.* Inicialmente, são consideradas todas as produções não-vazias. A seguir, cada produção cujo lado direito possui uma variável que gera a palavra vazia, determina uma produção adicional, sem esta variável;

c) *Inclusão de geração da palavra vazia, se necessário.* Se a palavra vazia pertence à linguagem, então é incluída uma produção para gerar a palavra vazia.

Definição 3.8 Algoritmo para Exclusão das Produções Vazias.

Considere uma Gramática Livre do Contexto $G = (V, T, P, S)$. O *Algoritmo para Exclusão das Produções Vazias* é composto por três etapas, como segue:

a) *Etapa 1: Conjunto de variáveis que constituem produções vazias.* O algoritmo para construir V_ε (conjunto das variáveis que geram ε) é o seguinte:

$V_\varepsilon = \{ A \mid A \rightarrow \varepsilon \}$;
repita
$V_\varepsilon = V_\varepsilon \cup \{ X \mid X \rightarrow X_1 ... X_n \in P$ tal que $X_1, ..., X_n \in V_\varepsilon \}$
até que o cardinal de V_ε não aumente;

b) *Etapa 2: Conjunto de produções sem produções vazias.* A gramática resultante desta etapa é:

$$G_1 = (V, T, P_1, S)$$

onde P_1 é construído como segue:

$P_1 = \{ A \rightarrow \alpha \mid \alpha \neq \varepsilon \}$;
repita
para toda $A \rightarrow \alpha \in P_1$ e $X \in V_\varepsilon$ tal que $\alpha = \alpha_1 X \alpha_2$ e $\alpha_1\alpha_2 \neq \varepsilon$
faça $P_1 = P_1 \cup \{ A \rightarrow \alpha_1\alpha_2 \}$
até que o cardinal de P_1 não aumente;

c) *Etapa 3: inclusão de geração da palavra vazia, se necessário.* Se a palavra vazia pertence à linguagem, então a gramática resultante desta etapa é:

$G_2 = (V, T, P_2, S)$ onde:

$P_2 = P_1 \cup \{S \rightarrow \varepsilon\}$ ❑

EXEMPLO 8 Exclusão das Produções Vazias.

Considere a seguinte Gramática Livre do Contexto:

$G = (\{S, X, Y\}, \{a, b\}, P, S)$, onde:
$P = \{S \rightarrow aXa \mid bXb \mid \varepsilon,\ X \rightarrow a \mid b \mid Y,\ Y \rightarrow \varepsilon\}$

A exclusão das produções vazias é como segue:

a) *Conjunto de variáveis que geram ε.*

iteração	V_ε
início	{S, Y}
1	{S, Y, X}
2	{S, Y, X}

b) *Conjunto de produções sem produções vazias.*

iteração	produções
inicial	$\{S \rightarrow aXa \mid bXb, X \rightarrow a \mid b \mid Y\}$
1	$\{S \rightarrow aXa \mid bXb \mid aa \mid bb, X \rightarrow a \mid b \mid Y\}$
2	$\{S \rightarrow aXa \mid bXb \mid aa \mid bb, X \rightarrow a \mid b \mid Y\}$

c) *Inclusão da geração da palavra vazia.* Como a palavra vazia pertence à linguagem, $S \rightarrow \varepsilon$ é incluída no conjunto de produções.

A gramática resultante é a seguinte:

$G = (\{S, X, Y\}, \{a, b\}, P, S)$, onde:
$P = \{S \rightarrow aXa \mid bXb \mid aa \mid bb \mid \varepsilon,\ X \rightarrow a \mid b \mid Y\}$

Deve-se reparar que Y, originalmente um símbolo útil, resultou em um símbolo inútil. Ou seja, a exclusão de produções vazias gerou um símbolo inútil. Veja adiante seção específica sobre a combinação de simplificações de gramáticas. ❑

Produções da Forma A → B

Uma produção da forma $A \rightarrow B$ não adiciona informação alguma em termos de geração de palavras, a não ser que a variável A pode ser substituída por B. Neste caso, se $B \rightarrow \alpha$, então a produção $A \rightarrow B$ pode ser substituída por $A \rightarrow \alpha$. A generalização desta idéia é o algoritmo proposto, dividido em duas etapas, como segue:

a) *Construção do fecho da cada variável.* Entende-se por fecho de uma variável o conjunto de variáveis que podem substituí-la transitivamente. Ou seja, se $A \rightarrow B$ e $B \rightarrow C$, então B e C pertencem ao fecho de A;

b) *Exclusão das produções da forma* $A \rightarrow B$. Substitui as produções da forma $A \rightarrow B$ por produções da forma $A \rightarrow \alpha$, onde α é atingível a partir de A através de seu fecho.

Definição 3.9 Algoritmo para Exclusão das Produções da Forma A → B.

Considere uma Gramática Livre do Contexto G = (V, T, P, S). O *Algoritmo para Exclusão de Produções da Forma* $A \rightarrow B$ é composto por duas etapas, como segue:

a) *Construção do fecho da cada variável.*

```
para   toda A ∈ V
faça   FECHO-A = { B | A ≠ B e A ⇒+ B
       usando exclusivamente produções da forma X → Y };
```

b) *Exclusão das produções da forma* $A \rightarrow B$. A gramática resultante desta etapa é:

$$G_1 = (V, T, P_1, S)$$

onde P_1 é construído como segue:

```
P1 = { A → α | α ∉ V };
para   toda A ∈ V e B ∈ FECHO-A
faça   se     B → α ∈ P e α ∉ V
       então  P1 = P1 ∪ { A → α };
```

❑

EXEMPLO 9 Exclusão das Produções da Forma $A \rightarrow B$.

Considere a seguinte Gramática Livre do Contexto:

G = ({S, X}, {a, b}, P, S), onde:
P = {S → aXa | bXb, X → a | b | S | ε}

A exclusão da produção $X \rightarrow S$ é como segue:

a) *Construção do fecho de cada variável.*

FECHO-S = ∅
FECHO-X = {S}

b) *Exclusão das produções da forma* $A \rightarrow B$. Construção do conjunto de produções (a coluna iteração representa a execução do algoritmo para a variável referenciada):

iteração	produções
inicial	$\{S \rightarrow aXa \mid bXb, X \rightarrow a \mid b \mid \varepsilon\}$
S	$\{S \rightarrow aXa \mid bXb, X \rightarrow a \mid b \mid \varepsilon\}$
X	$\{S \rightarrow aXa \mid bXb, X \rightarrow a \mid b \mid \varepsilon \mid aXa \mid bXb\}$

A gramática resultante é a seguinte:

$G = (\{S, X\}, \{a, b\}, P, S)$, onde:

$P = \{S \rightarrow aXa \mid bXb, X \rightarrow a \mid b \mid \varepsilon \mid aXa \mid bXb\}$ ❑

Simplificações Combinadas

Deve-se reparar que não é qualquer seqüência de simplificação de gramática que atinge os resultados desejados. Por exemplo, em uma gramática *sem* símbolos inúteis, mas *com* produções da forma $A \rightarrow B$, o algoritmo para excluir este tipo de produção pode gerar símbolos inúteis (por quê?). Portanto, caso os algoritmos sejam combinados, a seguinte seqüência de simplificação é recomendada:

a) *Exclusão de produções vazias*;

b) *Exclusão de produções da forma* $A \rightarrow B$;

c) *Exclusão de símbolos inúteis.*

3.5 Formas Normais

As formas normais estabelecem restrições rígidas na definição das produções, sem reduzir o poder de geração das Gramáticas Livres do Contexto. São usadas principalmente no desenvolvimento de algoritmos (com destaque para reconhecedores de linguagens) e na prova de teoremas.

As formas normais introduzidas são as seguintes:

- *Forma Normal de Chomsky* onde as produções são da forma:

$$A \rightarrow BC \quad \text{ou} \quad A \rightarrow a$$

- *Forma Normal de Greibach* onde as produções são da forma:

$$A \rightarrow a\alpha$$

onde α é uma palavra de variáveis.

Para cada caso, é apresentado um algoritmo de conversão de uma Gramática Livre do Contexto qualquer para a correspondente forma normal. As provas de que os algoritmos atingem os objetivos propostos são omitidas.

3.5.1 Forma Normal de Chomsky

Definição 3.10 Forma Normal de Chomsky.

Uma Gramática Livre do Contexto é dita na *Forma Normal de Chomsky* (*FNC*) se todas as suas produções são da forma:

$$A \rightarrow BC \quad \text{ou} \quad A \rightarrow a$$

onde A, B e C são variáveis e a é um terminal. ❑

O algoritmo a seguir transforma uma Gramática Livre do Contexto qualquer, cuja linguagem gerada não possua a palavra vazia, em uma gramática na Forma Normal de Chomsky. O algoritmo é dividido em três etapas, como segue:

a) *Simplificação da Gramática*. Simplifica a gramática excluindo as produções vazias (como a linguagem não possui a palavra vazia, todas as produções da forma $A \rightarrow \varepsilon$ podem ser excluídas), produções da forma $A \rightarrow B$ (se o lado direito de alguma produção tiver somente um símbolo, este será terminal) e, opcionalmente, os símbolos inúteis;

b) *Transformação do lado direito das produções de comprimento maior ou igual a dois*. Garante que o lado direito das produções de comprimento maior ou igual a dois é composto exclusivamente por variáveis. A exclusão de um terminal a pode ser realizada substituindo-o por uma variável intermediária C_a e incluindo a produção $C_a \rightarrow a$;

c) *Transformação do lado direito das produções de comprimento maior ou igual a três, em produções com exatamente duas variáveis*. Garante que o lado direito das produções de comprimento maior do que um é composto exatamente por duas variáveis. Após a execução da etapa acima, o lado direito das produções da forma $A \rightarrow B_1B_2...B_n$ $(n \geq 2)$ é composto exclusivamente por variáveis. Portanto, para concluir a transformação, é suficiente garantir que o lado direito é composto por exatamente duas variáveis. Isto é possível gerando $B_1B_2...B_n$ em diversas etapas, usando variáveis intermediárias.

Definição 3.11 Algoritmo para Transformar uma GLC na FNC.

Considere uma Gramática Livre do Contexto $G = (V, T, P, S)$, tal que $\varepsilon \notin L(G)$. O *Algoritmo para Transformar uma GLC na Forma Normal de Chomsky* é como segue:

a) *Etapa 1: Simplificação da Gramática.* As seguintes simplificações:
- produções vazias;
- produções da forma $A \rightarrow B$;
- símbolos inúteis (opcional);

devem ser realizadas usando os algoritmos de simplificação introduzidos anteriormente, resultando na gramática:

$$G_1 = (V_1, T_1, P_1, S)$$

b) *Etapa 2: Transformação do lado direito das produções de comprimento maior ou igual a dois.* A gramática resultante desta etapa é:

$$G_2 = (V_2, T_1, P_2, S)$$

onde V_2 e P_2 são construídos como segue:

$V_2 = V_1$;
$P_2 = P_1$;
para toda $A \rightarrow X_1X_2...X_n \in P_2$ tal que $n \geq 2$
faça se para $r \in \{1,...,n\}$, X_r é um símbolo terminal
então (suponha $X_r = a$)
$V_2 = V_2 \cup \{ C_a \}$;
substitui a pela variável C_a em $A \rightarrow X_1X_2...X_n \in P_2$;
$P_2 = P_2 \cup \{ C_a \rightarrow a \}$;

c) *Etapa 3: Transformação do lado direito das produções de comprimento maior ou igual a três, em produções com exatamente duas variáveis.* A gramática resultante desta etapa é:

$$G_3 = (V_3, T_1, P_3, S)$$

onde V_3 e P_3 são construídos como segue:

$V_3 = V_2$;
$P_3 = P_2$;
para toda $A \rightarrow B_1B_2...B_n \in P_3$ tal que $n \geq 3$
faça $P_3 = P_3 - \{ A \rightarrow B_1B_2...B_n \}$;
$V_3 = V_3 \cup \{ D_1,...,D_{n-2} \}$;
$P_3 = P_3 \cup \{ A \rightarrow B_1D_1,\ D_1 \rightarrow B_2D_2,\ ...,$
$D_{n-3} \rightarrow B_{n-2}D_{n-2},\ D_{n-2} \rightarrow B_{n-1}B_n \}$;

❑

EXEMPLO 10 Transformação de uma GLC na FNC.

Considere a gramática G_2 que gera expressões aritméticas introduzida no Exemplo 2:

$G_2 = (\{E\}, \{+, *, [,], x\}, P_2, E)$, onde:
$P_2 = \{E \rightarrow E+E \mid E*E \mid [E] \mid x\}$

a) *Simplificação.* A gramática já está simplificada;

b) *Variáveis do lado direito.* Excetuando-se a produção $E \rightarrow x$, as demais devem ser substituídas como segue:

$E \rightarrow EC_+E \mid EC_*E \mid C_[EC_]$
$C_+ \rightarrow +$
$C_* \rightarrow *$
$C_[\rightarrow [$
$C_] \rightarrow]$

c) *Exatamente duas variáveis do lado direito.* As produções:

$E \rightarrow EC_+E \mid EC_*E \mid C_[EC_]$

necessitam ser substituídas como segue:

$E \rightarrow ED_1 \mid ED_2 \mid C_[D_3$
$D_1 \rightarrow C_+E$
$D_2 \rightarrow C_*E$
$D_3 \rightarrow EC_]$

A gramática na Forma Normal de Chomsky resultante é a seguinte:

$G_2' = (\{E, C_+, C_*, C_[, C_], D_1, D_2, D_3\}, \{+, *, [,], x\}, P_2', E)$, onde:
$P_2' = \{E \rightarrow ED_1 \mid ED_2 \mid C_[D_3 \mid x,$
$D_1 \rightarrow C_+E, D_2 \rightarrow C_*E, D_3 \rightarrow EC_],$
$C_+ \rightarrow +, C_* \rightarrow *, C_[\rightarrow [, C_] \rightarrow]\}$ ❑

3.5.2 Forma Normal de Greibach

Definição 3.12 Forma Normal de Greibach.

Uma Gramática Livre do Contexto é dita na *Forma Normal de Greibach* (*FNG*) se todas as suas produções são da forma:

$$A \rightarrow a\alpha$$

onde A é uma variável, a é um terminal e α é uma palavra de variáveis. ❑

O algoritmo a seguir transforma uma Gramática Livre do Contexto qualquer, cuja linguagem gerada não possua a palavra vazia, em uma gramática na Forma Normal de Greibach. O algoritmo é dividido nas seguintes etapas:

a) *Simplificação da Gramática.* Análoga à correspondente etapa do algoritmo referente a Forma Normal de Chomsky;

b) *Renomeação das variáveis em uma ordem crescente qualquer.* As variáveis da gramática são renomeadas em uma ordem crescente qualquer, como, por exemplo, $A_1, A_2, ..., A_n$, onde n é o cardinal do conjunto de variáveis. Diferentes critérios de renomeação podem resultar em diferentes gramáticas na Forma Normal de Greibach. Entretanto, todas são equivalentes (geram a mesma linguagem);

c) *Transformação de produções para a forma* $A_r \rightarrow A_s\alpha$, *onde* $r \leq s$. As produções são modificadas garantindo que a primeira variável do lado direito é maior ou igual que a do lado esquerdo, considerando a ordenação da etapa anterior. As produções $A_r \rightarrow A_s\alpha$ tais que $r > s$ são modificadas substituindo a variável A_s pelas suas correspondentes produções ($A_s \rightarrow \beta_1 \mid ... \mid \beta_m$), resultando em $A_r \rightarrow \beta_1\alpha \mid ... \mid \beta_m\alpha$ e assim sucessivamente. Entretanto, como o conjunto de variáveis é finito, existe um limite para as produções crescentes, que pode ser a geração de um terminal ($A_r \rightarrow a\alpha$) ou de uma recursão ($A_r \rightarrow A_r\alpha$);

d) *Exclusão das recursões. da forma* $A_r \rightarrow A_r\alpha$. As recursões (à esquerda) podem existir originalmente na gramática, ou serem geradas pela etapa anterior. A eliminação da recursão à esquerda pode ser realizada introduzindo variáveis auxiliares e incluindo recursão à direita ($B_r \rightarrow \alpha B_r$);

e) *Um terminal no início do lado direito de cada produção.* Após a execução da etapa anterior, todas as produções da forma $A_r \rightarrow A_s\alpha$ são tais que $r < s$. Conseqüentemente, as produções da maior variável A_n só podem iniciar por um terminal no lado direito. Assim, se em $A_{n-1} \rightarrow A_n\alpha$ for substituído A_n pelas suas correspondentes produções (exemplo: $A_n \rightarrow a\beta$), o lado direito das produções de A_{n-1} também iniciarão por um terminal (exemplo: $A_{n-1} \rightarrow a\beta\alpha$). A repetição do algoritmo para $A_{n-2}, ..., A_1$ resultará em produções exclusivamente da forma $A_r \rightarrow a\alpha$;

f) *Produções na forma* $A \rightarrow a\alpha$ *onde* α *é composta por variáveis.* É análoga a correspondente etapa do algoritmo relativo à Forma Normal de Chomsky.

Definição 3.13 Algoritmo para Transformar uma GLC na FNG.

Considere uma Gramática Livre do Contexto $G = (V, T, P, S)$, tal que $\varepsilon \notin L(G)$. O *Algoritmo para Transformar uma GLC na Forma Normal de Greibach* é como segue:

a) *Etapa 1: Simplificação da Gramática.* As seguintes simplificações:
- produções vazias;
- produções da forma $A \rightarrow B$;
- símbolos inúteis (opcional);

devem ser realizadas usando os algoritmos de simplificação introduzidos anteriormente, resultando na gramática:

$$G_1 = (V_1, T_1, P_1, S)$$

b) *Etapa 2: Renomeação das variáveis em uma ordem crescente qualquer.* A gramática resultante desta etapa é:

$$G_2 = (V_2, T_1, P_2, S)$$

onde V_2 e P_2 são construídos como segue (suponha que o cardinal de V_1 é n):

$V_2 = \{A_1, A_2, ..., A_n\}$ é V_1 onde as variáveis são renomeadas;

P_2 é P_1 renomeando as variáveis nas produções;

c) *Etapas 3 e 4: Transformação de produções para a forma* $A_r \rightarrow A_s\alpha$, *onde* $r \leq s$ e *exclusão das recursões da forma* $Ar \rightarrow A_r\alpha$. A gramática resultante destas duas etapas realizadas em conjunto é:

$$G_3 = (V_3, T_1, P_3, S)$$

onde V_3, P_3 são construídos como segue, supondo que o cardinal de V_2 é n:

```
P3 = P2
para  r variando de 1 até n
faça
      para  s variando de 1 até r-1                          Etapa 3
      faça  para  toda Ar → Asα ∈ P3
            faça  excluir Ar → Asα de P3;
                  para  toda As → β ∈ P3
                  faça  P3 = P3 ∪ { Ar → βα }

      para  toda Ar → Arα ∈ P3                               Etapa 4
      faça  excluir Ar → Arα de P3;
            V3 = V3 ∪ { Br };
            P3 = P3 ∪ { Br → α } ∪ { Br → αBr };
      para  toda Ar → ϕ ∈ P3 tal que ϕ não inicia por Ar e
            alguma Ar → Arα foi excluída
      faça  P3 = P3 ∪ { Ar → ϕBr };
```

d) *Etapa 5: Um terminal no início do lado direito de cada produção.* A gramática resultante desta etapa é:

$$G_4 = (V_3, T_1, P_4, S)$$

onde P_4 é construído como segue:

```
P4 = P3;
para  r variando de n-1 até 1 e toda Ar → Asα ∈ P4
faça  excluir Ar → Asα de P4;
      para  toda As → β de P4
      faça  P4 = P4 ∪ { Ar → βα };
```

Também é necessário garantir que as produções relativas às variáveis auxiliares B_r iniciam por um terminal do lado direito, como segue:

```
para  toda Br → AsβrP4
faça  excluir Br → Asβr de P4;
      para  toda As → aα
      faça  P4 = P4 ∪ { Br → aαβr };
```

e) *Etapa 6: Produções na forma* $A \rightarrow a\alpha$ *onde* α *é composta por variáveis.* É análoga à correspondente etapa do algoritmo relativo à Forma Normal de Chomsky. ❑

EXEMPLO 11 Transformação de uma GLC na FNG.

Considere a seguinte Gramática Livre do Contexto:

$G = (\{S, A\}, \{a, b\}, P, S)$, onde:
$P = \{S \rightarrow AA \mid a, A \rightarrow SS \mid b\}$

A transformação na correspondente Forma Normal de Greibach é como segue:

a) *Simplificação.* A gramática já está simplificada;

b) *Renomeação das variáveis em ordem crescente.* As variáveis S e A são renomeadas para A_1 e A_2, respectivamente. As produções da gramática ficam como segue:

$A_1 \rightarrow A_2A_2 \mid a$
$A_2 \rightarrow A_1A_1 \mid b$

c) $A_r \rightarrow A_s\alpha$, *com* $r \leq s$ e *recursões* $A_r \rightarrow A_r\alpha$. A produção $A_2 \rightarrow A_1A_1$ necessita ser modificada. As produções da gramática ficam como segue:

$A_1 \rightarrow A_2A_2 \mid a$
$A_2 \rightarrow A_2A_2A_1 \mid aA_1 \mid b$

A produção $A_2 \rightarrow A_2A_2A_1$ contém um recursão. Portanto, é necessário introduzir uma variável auxiliar B, como segue:

$A_1 \rightarrow A_2A_2 \mid a$
$A_2 \rightarrow aA_1 \mid b \mid aA_1B \mid bB$
$B \rightarrow A_2A_1 \mid A_2A_1B$

d) *Terminal no início de cada lado direito das produções.* O lado direito das produções da maior variável A_2 iniciam por um terminal. Substituindo A_2 no lado direito de $A_1 \rightarrow A_2A_2$ pelas suas correspondentes derivações, as produções de A_1 também iniciarão por um terminal:

$A_1 \rightarrow aA_1A_2 \mid bA_2 \mid aA_1BA_2 \mid bBA_2 \mid a$
$A_2 \rightarrow aA_1 \mid b \mid aA_1B \mid bB$
$B \rightarrow A_2A_1 \mid A_2A_1B$

As produções referentes à variável B também são modificadas:

$B \rightarrow aA_1A_1 \mid bA_1 \mid aA_1BA_1 \mid bBA_1 \mid aA_1A_1B \mid bA_1B \mid aA_1BA_1B \mid bBA_1B$

e) $A \rightarrow a\alpha$, *com* α *composta exclusivamente por variáveis.* Nenhum procedimento é necessário, pois as produções já se encontram nesta forma.

A gramática na Forma Normal de Greibach resultante é a seguinte:

$G = (\{A_1, A_2, B\}, \{a, b\}, P, A_1)$, onde:
$P = \{ A_1 \rightarrow aA_1A_2 \mid bA_2 \mid aA_1BA_2 \mid bBA_2 \mid a,$
$A_2 \rightarrow aA_1 \mid b \mid aA_1B \mid bB,$
$B \rightarrow aA_1A_1 \mid bA_1 \mid aA_1BA_1 \mid bBA_1 \mid aA_1A_1B \mid bA_1B \mid aA_1BA_1B \mid bBA_1B\}$ ❑

3.6 Recursão à Esquerda

Em diversas situações, como no desenvolvimento de algoritmos reconhecedores, é desejável que a gramática que representa a linguagem não seja recursiva à esquerda.

Entende-se por recursão à esquerda a ocorrência da seguinte situação:

$$A \Rightarrow^{+} A\alpha$$

Ou seja, uma variável deriva ela mesma, de forma direta ou indireta, como o símbolo mais à esquerda de uma subpalavra gerada.

A transformação de uma gramática qualquer em uma sem recursão à esquerda pode ser realizada executando as quatro primeiras etapas do algoritmo referente à Forma Normal de Greibach, que são as seguintes:

a) *Etapa 1: Simplificação da Gramática*;

b) *Etapa 2: Renomeação das variáveis em uma ordem crescente qualquer*;

c) *Etapa 3: Qualquer produção é da forma* $A_r \rightarrow A_s\alpha$, *onde* $r < s$;

d) *Etapa 4: Exclusão das recursões da forma* $A_r \rightarrow A_r\alpha$.

3.7 Autômato com Pilha

Analogamente às Linguagens Regulares, a Classe das Linguagens Livres do Contexto pode ser associada a um formalismo do tipo autômato, denominado Autômato com Pilha.

O Autômato com Pilha é análogo ao Autômato Finito, incluindo uma pilha como memória auxiliar e a facilidade de não-determinismo. A pilha é independente da fita de entrada e não possui limite máximo de tamanho ("infinita"). Estruturalmente, sua principal característica é que o último símbolo gravado é o primeiro a ser lido, como ilustrado na Figura 3.6. A *base* de uma pilha é fixa e define o seu início. O *topo* é variável e define a posição do último símbolo gravado.

A facilidade de não-determinismo é importante e necessária, pois aumenta o poder computacional dos Autômatos com Pilha, permitindo reconhecer exatamente a Classe das Linguagens Livres do Contexto. Por exemplo, o reconhecimento da linguagem:

$$\{ ww^r \mid w \text{ é palavra sobre } \{a, b\} \}$$

só é possível por um Autômato com Pilha Não-Determinístico.

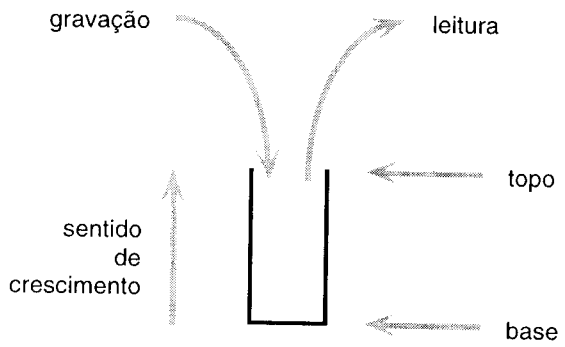


Figura 3.6 Estrutura do tipo pilha

Embora o poder computacional do Autômato com Pilha seja muito superior ao do Autômato Finito, ainda é restrito, não sendo possível reconhecer linguagens simples, como, por exemplo:

$$\{ww \mid w \text{ é palavra sobre } \{a, b\}\} \quad \text{ou} \quad \{a^n b^n c^n \mid n \geq 0\}$$

Um resultado interessante mostrado adiante é que qualquer Linguagem Livre do Contexto pode ser reconhecida por um Autômato com Pilha com somente um estado (ou três estados, dependendo da definição). Isto significa que a estrutura de pilha é suficiente como única memória, não sendo necessário usar os estados para "memorizar" informações passadas. Ou seja, a estrutura de estados no Autômato com Pilha poderia ser excluída sem reduzir o poder computacional.

3.7.1 Definição do Autômato com Pilha

O modelo Autômato com Pilha possui duas definições universalmente aceitas que diferem no critério de parada do autômato, como segue:

- o valor inicial da pilha é vazio e o autômato pára aceitando ao atingir um estado final;
- a pilha contém, inicialmente, um símbolo especial denominado símbolo inicial da pilha. Não existem estados finais e o autômato pára aceitando quando a pilha estiver vazia.

As duas definições são equivalentes (possuem o mesmo poder computacional), sendo fácil modificar um Autômato com Pilha para satisfazer a outra definição. Nesta publicação, é adotada a com estados finais.

Um Autômato com Pilha ou Autômato com Pilha Não-Determinístico é composto, basicamente, por quatro partes, como segue:

a) *Fita*. Análoga à do Autômato Finito;

b) *Pilha*. Memória auxiliar que pode ser usada livremente para leitura e gravação;

c) *Unidade de Controle*. Reflete o estado corrente da máquina. Possui uma cabeça de fita e uma cabeça de pilha;

d) *Programa ou Função de Transição*. Comanda a leitura da fita, leitura e gravação da pilha e define o estado da máquina.

A pilha é dividida em células, armazenando, cada uma, um símbolo do alfabeto auxiliar (pode ser igual ao alfabeto de entrada). Em uma estrutura do tipo pilha, a leitura ou gravação é sempre na mesma extremidade, denominada topo. Não possui tamanho fixo e nem máximo, sendo seu tamanho corrrente igual ao tamanho da palavra armazenada. Seu valor inicial é vazio.

A unidade de controle possui um número finito e predefinido de estados. Possui uma cabeça de fita e uma cabeça de pilha, como segue:

a) *Cabeça da Fita*. Unidade de leitura a qual acessa uma célula da fita de cada vez e movimenta-se exclusivamente para a direita. É possível testar se a entrada foi completamente lida;

b) *Cabeça da Pilha*. Unidade de leitura e gravação a qual move para a esquerda (ou "para cima") ao gravar e para a direita (ou "para baixo") ao ler um símbolo. Acessa um símbolo de cada vez, estando sempre posicionada no topo. A leitura exclui o símbolo lido. É possível testar se a pilha está vazia. Em uma mesma operação de gravação, é possível armazenar uma palavra composta por mais de um símbolo. Neste caso, o símbolo do topo é o mais à esquerda da palavra gravada.

O programa é uma função parcial que, dependendo do estado corrente, símbolo lido da fita e símbolo lido da pilha, determina o novo estado e a palavra a ser gravada (na pilha). Possui a facilidade de movimento vazio (análoga à do Autômato Finito), permitindo mudar de estado sem ler da fita.

Definição 3.14 Autômato com Pilha.

Um *Autômato com Pilha Não-Determinístico* (*APN*) ou simplesmente *Autômato com Pilha* (*AP*) M é uma 6-upla:

$$M = (\Sigma, Q, \delta, q_0, F, V)$$

onde:

Σ *alfabeto de símbolos de entrada*;

Q *conjunto de estados* possíveis do autômato o qual é finito;

δ *função programa* ou de *função de transição*:

$$\delta: Q \times (\Sigma \cup \{\varepsilon, ?\}) \times (V \cup \{\varepsilon, ?\}) \rightarrow 2^{Q \times V^*}$$

a qual é uma função parcial;

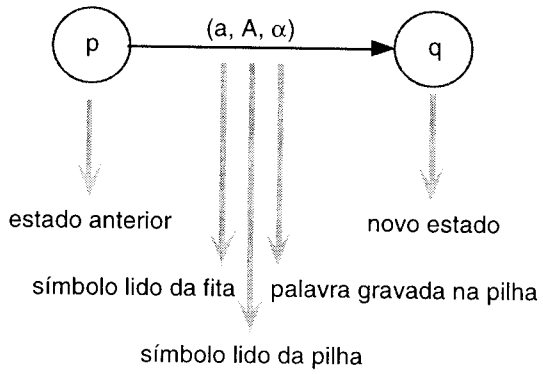


Figura 3.7 Representação da função programa como um grafo

q_0 *estado inicial* do autômato tal que q_0 é elemento de Q;
F *conjunto de estados finais* tal que F está contido em Q;
V *alfabeto auxiliar* ou *alfabeto da pilha*. ❑

As seguintes características da função programa devem ser consideradas:

- a função pode não ser total, ou seja, indefinida para alguns argumentos do conjunto de partida; a omissão do parâmetro de leitura, representada por "?", indica o teste de pilha vazia ou toda palavra de entrada lida;
- o símbolo ε na leitura indica a facilidade de movimento vazio da fita ou da pilha (o autômato não lê nem move a cabeça). Note-se que, para o movimento ser considerado não-determinístico, é suficiente que o movimento seja vazio na fita;
- o símbolo ε na gravação indica que nenhuma gravação é realizada na pilha (e não move a cabeça).

Por exemplo, $\delta(p, ?, \varepsilon) = \{(q, \varepsilon)\}$ indica que no estado p se a entrada foi completamente lida, não lê da pilha, assume o estado q e não grava na pilha. A função programa pode ser representada como um grafo direto, como ilustrado na Figura 3.7.

O processamento de um Autômato com Pilha, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa para cada símbolo de w (da esquerda para a direita) até ocorrer uma condição de parada. Entretanto, é possível que um Autômato com Pilha nunca atinja uma condição de parada. Neste caso, fica processando indefinidamente (ciclo ou *loop* infinito). Um exemplo simples de ciclo infinito é um programa que empilha e desempilha um mesmo símbolo indefinidamente, sem ler da fita.

Um Autômato com Pilha pode parar aceitando ou rejeitando a entrada ou ficar em *loop* infinito, como segue:

a) Um dos caminhos alternativos assume um estado final: o autômato pára e a palavra é aceita;

b) Todos os caminhos alternativos rejeitam a entrada: o autômato pára e a palavra é rejeitada;

c) Pelo menos um caminho alternativo está em *loop* infinito e os demais rejeitam (ou também estão em *loop* infinito): o autômato está em *loop* infinito.

Para definir formalmente o comportamento de um Autômato com Pilha, é necessário estender a definição da função programa usando como argumento um estado e uma palavra. Esta extensão é sugerida como exercício.

As seguintes notações são adotadas (suponha que M é um Autômato com Pilha):

a) ACEITA(M) ou L(M): conjunto de todas as palavras de Σ^* aceitas por M;

b) REJEITA(M): conjunto de todas as palavras de Σ^* rejeitadas por M;

c) LOOP(M): conjunto de todas as palavras de Σ^* para as quais M fica processando indefinidamente.

As seguintes afirmações são verdadeiras:

- $ACEITA(M) \cap REJEITA(M) \cap LOOP(M) = \varnothing$
- $ACEITA(M) \cup REJEITA(M) \cup LOOP(M) = \Sigma^*$
- o complemento de:
  ACEITA(M) é $REJEITA(M) \cup LOOP(M)$
  REJEITA(M) é $ACEITA(M) \cup LOOP(M)$
  LOOP(M) é $ACEITA(M) \cup REJEITA(M)$

EXEMPLO 12 Autômato com Pilha.

Considere a seguinte linguagem introduzida no Exemplo 1:

$$L_1 = \{ a^n b^n \mid n \geq 0 \}$$

O Autômato com Pilha $M_1 = (\{ a, b \}, \{ q_0, q_1, q_f \}, \delta_1, q_0, \{ q_f \}, \{ B \})$, onde δ_1 é como abaixo, é tal que $ACEITA(M_1) = L_1$ (o conjunto $LOOP(M_1)$ é vazio?):

$\delta_1 (q_0, a, \varepsilon) = \{ (q_0, B) \}$
$\delta_1 (q_0, b, B) = \{ (q_1, \varepsilon) \}$
$\delta_1 (q_0, ?, ?) = \{ (q_f, \varepsilon) \}$
$\delta_1 (q_1, b, B) = \{ (q_1, \varepsilon) \}$
$\delta_1 (q_1, ?, ?) = \{ (q_f, \varepsilon) \}$

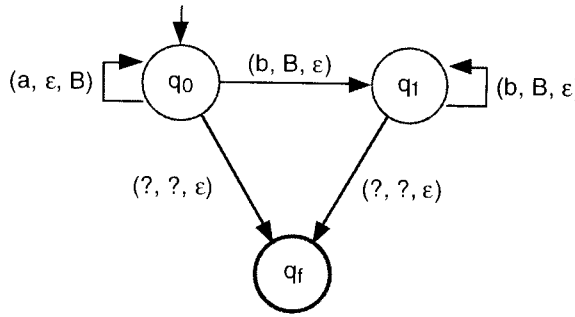


Figura 3.8 Grafo do Autômato com Pilha

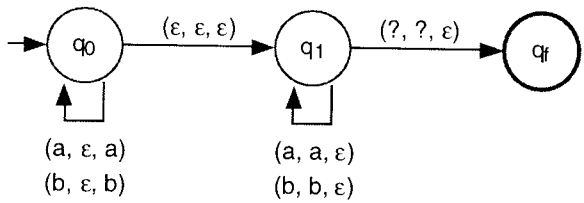


Figura 3.9 Grafo do Autômato com Pilha

O autômato pode ser representado pelo grafo ilustrado na Figura 3.8. Note-se que o autômato é determinístico. No estado q_0, para cada símbolo a lido da fita é armazenado um símbolo B na pilha. No estado q_1, é realizado um batimento, verificando se para cada símbolo b da fita, existe um correspondente B na pilha. O algoritmo somente aceita se ao terminar de ler toda a palavra de entrada a pilha estiver vazia. ❑

EXEMPLO 13 Autômato com Pilha.

Considere a seguinte linguagem:

$$L_3 = \{ ww^r \mid w \text{ pertence a } \{a, b\}^* \}$$

O autômato M_3 ilustrado na Figura 3.9 é tal que $ACEITA(M_3) = L_3$ (o conjunto $LOOP(M_3)$ é vazio?). Note-se que M_3 é não-determinístico devido ao movimento vazio de q_0 para q_1. Adicionalmente, o alfabeto auxiliar é igual ao de entrada. Em q_0, é empilhado o reverso do prefixo. A cada símbolo empilhado, ocorre um movimento não-determinista para q_1 o qual verifica se o sufixo da palavra é igual ao conteúdo da pilha. ❑

EXEMPLO 14 Autômato com Pilha.

Considere a seguinte linguagem:

$$L_4 = \{ a^n b^m a^{n+m} \mid n \geq 0, m \geq 0 \}$$

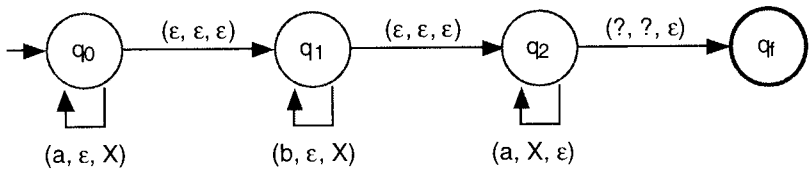


Figura 3.10 Grafo do Autômato com Pilha

O autômato não-determinístico M_4 ilustrado na Figura 3.10 é tal que $ACEITA(M_4) = L_4$ (o conjunto $LOOP(M_4)$ é vazio?). M_4 empilha um símbolo auxiliar X para cada a ou b em q_0 ou q_1, respectivamente. Após, em q_2, verifica se o número de a no sufixo é igual ao de X empilhado. ❑

3.7.2 Autômato com Pilha e Linguagens Livres do Contexto

A classe das linguagens reconhecidas pelos Autômatos com Pilha é igual à Classe das Linguagens Livres do Contexto (ou seja, é igual à classe das linguagens geradas pelas Gramáticas Livres do Contexto). A demonstração apresentada a seguir é dividida em dois teoremas. O primeiro teorema apresenta a construção de um AP a partir de uma GLC qualquer, permitindo estabelecer as seguintes conclusões:

- a construção de um reconhecedor para uma Linguagem Livre do Contexto a partir de sua gramática é simples e imediata;
- qualquer Linguagem Livre do Contexto pode ser reconhecida por um Autômato com Pilha com somente um estado de controle lógico, o que significa que a facilidade de memorização de informações através de estados (como nos Autômatos Finitos) não aumenta o poder computacional dos AP.

Teorema 3.15 Gramática Livre do Contexto → Autômato com Pilha.

Se L é uma Linguagem Livre do Contexto, então existe M, Autômato com Pilha tal que $ACEITA(M) = L$.

Prova: Suponha que a palavra vazia não pertence à L. A demonstração consiste na construção de um Autômato com Pilha a partir da gramática transformada na Forma Normal de Greibach (produções da forma $A \rightarrow a\alpha$, α palavra de variáveis). O Autômato com Pilha gerado simula a derivação mais à esquerda, como segue (suponha a produção $A \rightarrow a\alpha$):

- lê o símbolo a da fita;
- lê o símbolo A da pilha;
- empilha a palavra α.

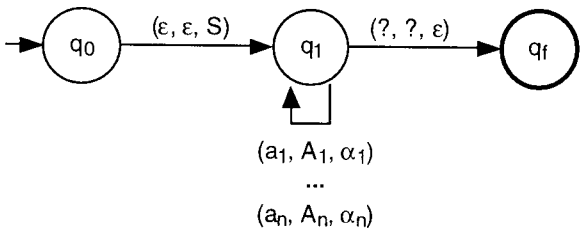


Figura 3.11 Grafo do AP construído a partir de uma gramática na FNG

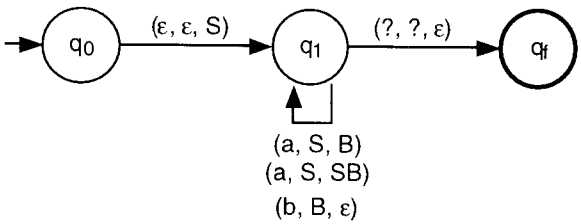


Figura 3.12 Grafo do AP construído a partir de uma gramática na FNG

A simulação acima é realizada para cada produção, usando um único estado de controle. A construção do AP M a partir da gramática G = (V, T, P, S) é como segue (veja a Figura 3.11):

- seja G' = (V', T', P', S), a transformação de G na Forma Normal de Greibach;
- seja M = (T', { q_0, q_1, q_f }, δ, q_0, { q_f }, V'), onde:
  $\delta(q_0, \varepsilon, \varepsilon) = \{ (q_1, S) \}$
  $\delta(q_1, a, A) = \{ (q_1, \alpha) \mid A \rightarrow a\alpha \in P' \}$
  $\delta(q_1, ?, ?) = \{ (q_f, \varepsilon) \}$

A demonstração de que ACEITA(M) = GERA(G') é realizada por indução no número de movimentos de M (ou derivação de G') e é sugerida como exercício. Como o autômato pode ser modificado para tratar a palavra vazia? ❑

EXEMPLO 15 Construção de um AP a partir de uma Gramática na FNG.

A linguagem (compare com a linguagem L_1 introduzida no Exemplo 1 - por que a diferença?):

$$L_5 = \{ a^n b^n \mid n \geq 1 \}$$

representada pela seguinte gramática na Forma Normal de Greibach:

$G_5 = (\{ S, B \}, \{ a, b \}, P_5, S)$, onde:
$P_5 = \{ S \rightarrow aB \mid aSB, B \rightarrow b \}$

é reconhecida pelo Autômato com Pilha $M_5 = (\{a, b\}, \{q_0, q, q_f\}, \delta_5, q_0, \{q_f\}, \{S, B\})$ ilustrado na Figura 3.12, construído a partir de G_5. ❑

As seguintes proposições são corolários do Teorema 3.15.

Corolário 3.16 Autômato com Pilha × Número de Estados.

Se L é uma Linguagem Livre do Contexto, então:

a) Existe M, Autômato com Pilha com controle de aceitação por estados finais, com somente três estados tal que ACEITA(M) = L;

b) Existe M, Autômato com Pilha com controle de aceitação por pilha vazia, com somente um estado tal que ACEITA(M) = L. ❑

O detalhamento da demonstração do item b) do Corolário 3.16 é simples e é sugerida como exercício.

Corolário 3.17 Existência de um Autômato com Pilha que Sempre Pára.

Se L é uma Linguagem Livre do Contexto, então existe M, Autômato com Pilha tal que:

$ACEITA(M) = L$
$REJEITA(M) = \Sigma^* - L$
$LOOP(M) = \varnothing$ ❑

Ou seja, para qualquer Linguagem Livre do Contexto existe um Autômato com Pilha que sempre pára para qualquer entrada (por quê?).

A demonstração do seguinte teorema é omitida.

Teorema 3.18 Autômato com Pilha → Gramática Livre do Contexto.

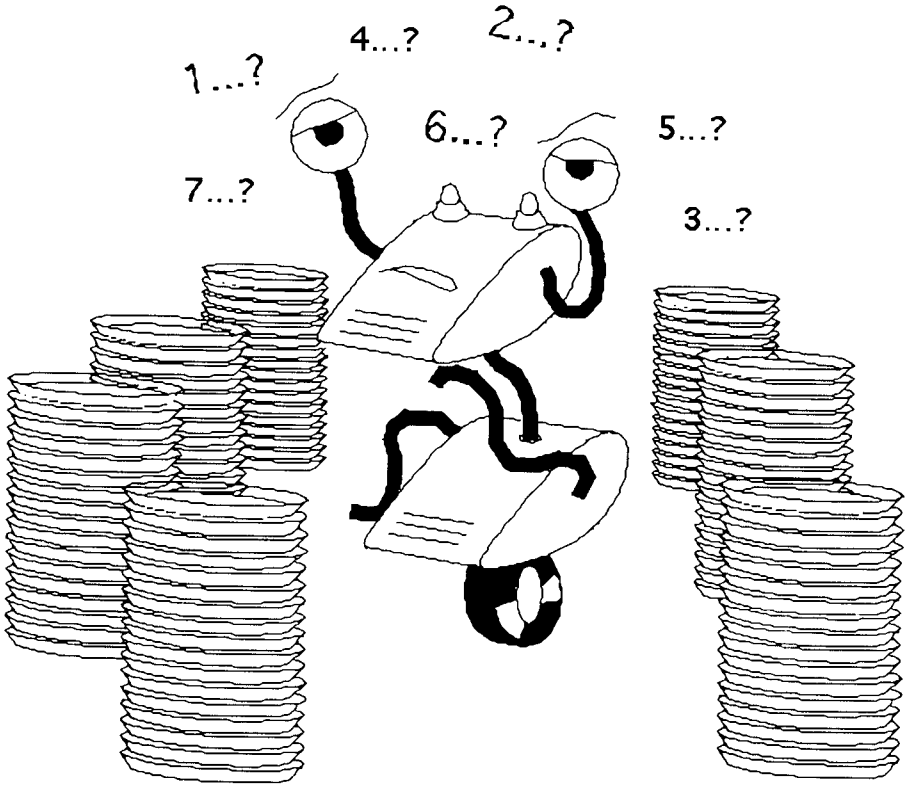
Se L é aceita por um Autômato com Pilha, então L é Linguagem Livre do Contexto. ❑

Observação 3.19 Estados × Poder Computacional dos Autômatos com Pilha.

A combinação dos seguintes resultados:

- Corolário 3.16;
- Corolário 3.17;
- Teorema 3.18;

comprovam que o uso dos estados como "memória" não aumenta o poder de reconhecimento do Autômato com Pilha. ❑



3.7.3 Número de Pilhas e o Poder Computacional

O Autômato com Pilha é um modelo freqüentemente referenciado em estudos aplicados e formais pois, além da estrutura de pilha ser adequada para implementação em computadores, poucas modificações sobre a definição básica determinam significativas alterações no poder computacional. Especificamente, os principais estudos de linguagens e computabilidade podem ser desenvolvidos usando exclusivamente o AP, variando o número de pilhas com ou sem a facilidade de não-determinismo. Os seguintes resultados podem ser enunciados:

a) *Autômato com Pilha, sem usar a estrutura de pilha.* Se a estrutura de pilha não for usada, a única forma de memorizar informações passadas é usando os estados. Assim, claramente o AP sem usar a pilha é muito semelhante ao Autômato Finito. De fato, é fácil mostrar que a Classe das Linguagens aceitas por AP sem pilha, com ou sem a facilidade de não-determinismo, é igual a Classe das Linguagens Regulares (o que se sugere como exercício);

b) *Autômato com Pilha Determinístico.* O *Autômato com uma Pilha Determinístico* (*APD*) aceita um subconjunto próprio das Linguagens Livres do Contexto, denominadas *Linguagens Livres do Contexto Determinísticas* (*LLCD*). A Classe das LLCD inclui muitas das linguagens aplicadas em informática, com destaque para as de programação. Uma das razões é que a implementação de um APD em um computador é simples e eficiente, facilitando, assim, o desenvolvimento de tradutores de linguagens. Algumas propriedades das LLCD são as seguintes:
   b.1) É possível definir um tipo de gramática que gera exatamente a Classe das LLCD. Entretanto, não são restrições simples sobre a definição geral de gramática;
   b.2) A Classe das LLCD é fechada para a operação de complemento;
   b.3) A Classe das LLCD não é fechada para as operações de união, intersecção e concatenação;

c) *Autômato com Pilha Não-Determinístico.* Conforme já verificado, a classe das linguagens reconhecida pelo AP é exatamente a Livre do Contexto;

d) *Autômato com Duas Pilhas.* O *Autômato com Duas Pilhas* (*A2P*) é equivalente, em termos de poder computacional, à Máquina de Turing (a ser introduzida adiante), considerado o dispositivo mais geral de computação. Assim, se existe um algoritmo para resolver um problema (por exemplo, reconhecer uma determinada linguagem), então este algoritmo pode ser expresso como um A2P. A facilidade de não-determinismo não aumenta o poder computacional do Autômato com Duas Pilhas;

e) *Autômato com Mais de Duas Pilhas.* O poder computacional de um *Autômato com Múltiplas Pilhas* (*AnP*, $n > 2$) é equivalente ao do A2P. Ou seja, se um problema é solucionado por um AnP, então o mesmo problema pode ser solucionado por um A2P.

3.8 Propriedades das Linguagens Livres do Contexto

Embora as Linguagens Livres do Contexto sejam mais gerais que as Regulares, ainda são relativamente restritas. Ou seja, é fácil definir linguagens que não são Livres do Contexto, como, por exemplo:

$\{ww \mid w \text{ pertence a } \{a, b\}^*\}$

$\{a^n b^n c^n \mid n \geq 0\}$

Assim, algumas questões sobre as Linguagens Livres do Contexto necessitam ser analisadas:

a) Como determinar se uma linguagem é Livre do Contexto?

b) A Classe das Linguagens Livres do Contexto é fechada para operações como união, intersecção, concatenação e complemento?

c) Como verificar se uma Linguagem Livre do Contexto é infinita ou finita (ou até mesmo vazia)?

Investigação se é Linguagem Livre do Contexto

Para mostrar que uma determinada linguagem é Livre do Contexto, é suficiente expressá-la usando os formalismos Gramática Livre do Contexto ou Autômato com Pilha. Entretanto, a demonstração que não é Livre do Contexto necessita ser realizada caso a caso.

Analogamente às Linguagens Regulares, as Linguagens Livres do Contexto possuem um Lema de Bombeamento o qual é útil no estudo das propriedades.

Lema 3.20 Bombeamento para as Linguagens Livres do Contexto.

Se L é uma Linguagem Livre do Contexto, então:

existe uma constante n tal que,
para qualquer palavra w de L onde $|w| \geq n$,
w pode ser definida como w = uxvyz onde $|xvy| \leq n$, $|xy| \geq 1$ e,
para todo $i \geq 0$, ux^ivy^iz é palavra de L. ❑

Note-se que para w = uxvyz, tem-se que ou x ou y pode ser a palavra vazia (mas não ambas). Uma forma de demonstrar o lema é usando gramáticas na Forma Normal de Chomsky. Neste caso, se a gramática possui s variáveis, pode-se assumir que $n = 2^s$. O lema não será demonstrado.

EXEMPLO 16 Linguagem Não-Livre do Contexto.

A seguinte linguagem não é Livre do Contexto:

$$L = \{ a^n b^n c^n \mid n \geq 0 \}$$

A prova que segue usa o bombeamento e é por absurdo. Suponha que L é Livre do Contexto. Então:

existe uma gramática na FNC G com s variáveis que gera L;
sejam $r = 2^s$ e $w = a^r b^r c^r$

Pelo bombeamento, w pode ser definida como w = uxvyz onde:

$|xvy| \leq r$, $|xy| \geq 1$ e,
para todo $i \geq 0$, ux^ivy^iz é palavra de L,

o que é um absurdo, pois, como $|xvy| \leq r$, não é possível supor que xy possui símbolos a e c, pois quaisquer ocorrências de a e c estão separadas por, pelo menos, r ocorrências de b. Conseqüentemente, xy jamais possuirá ocorrências de a, b e c simultaneamente. Assim, tem-se que:

- se xy possui somente símbolos a, é fácil verificar que a aplicação do bombeamento pode desbalancear as ocorrências de a, b e c;
- analogamente para os seguintes casos: somente símbolos c, somente símbolos a e b e somente símbolos b e c. ❑

Operações sobre Linguagens Livres do Contexto

Relativamente à Classe das Linguagens Livres do Contexto, tem-se que:

- é fechada para as operações de união e concatenação;
- não é fechada para as operações de intersecção e complemento.

O teorema a seguir mostra que a Classe das Linguagens Livres do Contexto é fechada para as operações de união e concatenação.

Teorema 3.21 Operações Fechadas sobre Linguagens Livres do Contexto.

A Classe das Linguagens Livres do Contexto é fechada para as seguintes operações:

a) União;

b) Concatenação.

Prova:

União. A demonstração que segue é baseada no formalismo Autômato com Pilha e usa a facilidade de não-determinismo. Sugere-se como exercício a demonstração usando o formalismo Gramática Livre do Contexto.

Suponha L_1 e L_2, LLC. Então, existem Autômatos com Pilha:

$M_1 = (\Sigma_1, Q_1, \delta_1, q_{0_1}, F_1, V_1)$
$M_2 = (\Sigma_2, Q_2, \delta_2, q_{0_2}, F_2, V_2)$

tais que $ACEITA(M_1) = L_1$ e $ACEITA(M_2) = L_2$. Seja M_3 construído como segue e ilustrado na Figura 3.13 (suponha que $Q_1 \cap Q_2 \cap \{q_0\} = \varnothing$ e $V_1 \cap V_2 = \varnothing$):

$M_3 = (\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2 \cup \{q_0\}, \delta_3, q_0, F_1 \cup F_2, V_1 \cup V_2)$

Claramente, M_3 reconhece $L_1 \cup L_2$.

Concatenação. A demonstração a seguir usa o formalismo Gramática Livre do Contexto. Sugere-se como exercício a demonstração usando o formalismo Autômato com Pilha.

Suponha L_1 e L_2, LLC. Então, existem Gramáticas Livres do Contexto:

$G_1 = (V_1, T_1, P_1, S_1)$

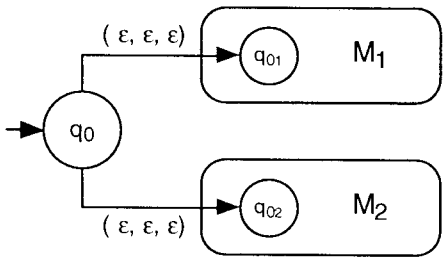


Figura 3.13 Grafo que representa a união de dois Autômatos com Pilha

$G_2 = (V_2, T_2, P_2, S_2)$

tais que $GERA(G_1) = L_1$ e $GERA(G_2) = L_2$. Seja G_3 construída como segue (suponha que $V_1 \cap V_2 \cap \{S\} = \emptyset$):

$G_3 = (V_1 \cup V_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}, S)$

a qual é Livre do Contexto (por quê?). Como a única produção de S é $S \rightarrow S_1S_2$, claramente qualquer palavra gerada por G_3 terá, como prefixo, uma palavra de L_1 e, como sufixo, uma palavra de L_2. Logo, L_1L_2 é LLC. ❑

O teorema a seguir mostra que a Classe das Linguagens Livres do Contexto não é fechada para as operações de intersecção e complemento. Note-se que, o fato de não ser fechada para a operação de complemento é, aparentemente, uma contradição, pois:

- foi verificado no Corolário 3.17 que, se L é LLC, então existe M, AP tal que ACEITA(M) = L e REJEITA(M) = L'. Ou seja, M é capaz de rejeitar qualquer palavra que não pertença à L;
- o teorema a seguir mostra que, se L é LLC, não se pode afirmar que L' também é LLC.

Assim, é perfeitamente possível *rejeitar* o complemento de uma LLC, embora nem sempre seja possível *aceitar* o complemento. Uma explicação intuitiva usando o formalismo AP é a seguinte onde, sem perda de generalidade, suponha que a função programa do AP em questão é total (por quê pode-se afirmar isto?):

- a condição para um AP aceitar uma entrada é que pelo menos um dos caminhos alternativos reconheça a palavra (mesmo que os demais rejeitem). Esta condição pode ser representada pelo diagrama ilustrado na Figura 3.14;
- se os estados de aceita e rejeita forem invertidos, na tentativa de aceitar o complemento, a situação resultante continua sendo de aceitação (e não de rejeição), pois permanecerá pelo menos um caminho alternativo reconhecendo a entrada. O diagrama ilustrado na Figura 3.15 representa a "negação" da condição acima.

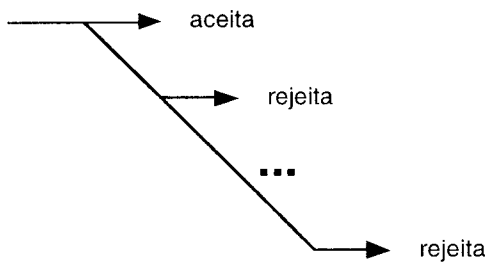


Figura 3.14 Exemplo de situação não-determinística para aceitação

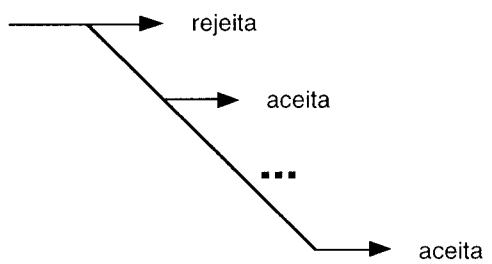


Figura 3.15 Inversão das condições de ACEITA/REJEITA

Portanto, considerando a facilidade de não-determinismo, o fato de existir um AP capaz de rejeitar o complemento de uma linguagem não implica que existe um AP capaz de aceitar o mesmo complemento.

Teorema 3.22 Operações Não-Fechadas sobre Linguagens Livres do Contexto.

A Classe das Linguagens Livres do Contexto não é fechada para as seguintes operações:

a) Intersecção;

b) Complemento.

Prova:

Intersecção. Para mostrar que a Classe das LLC não é fechada para a operação de intersecção, é suficiente mostrar um contra-exemplo. Assim, sejam:

$L_1 = \{ a^n b^n c^m \mid n \geq 0 \text{ e } m \geq 0 \}$
$L_2 = \{ a^m b^n c^n \mid n \geq 0 \text{ e } m \geq 0 \}$

É fácil mostrar que L_1 e L_2 são LLC. Entretanto, L_3 como abaixo, resultante da intersecção de L_1 com L_2, não é LLC:

$L3 = \{ a^n b^n c^n \mid n \geq 0 \}$

Complemento. Como a operação de intersecção pode ser representada em termos da união e do complemento, e considerando que a Classe das LLC não é fechada para a operação de intersecção, então não pode-se afirmar que o complemento de uma LLC é LLC. ❑

Investigação se uma Linguagem Livre do Contexto é Vazia, Finita ou Infinita

O teorema a seguir mostra que é possível construir algoritmos para determinar se uma Linguagem Livre do Contexto é vazia, finita ou infinita.

Teorema 3.23 Linguagem Livre do Contexto Vazia, Finita ou Infinita.

Se L é LLC, então é possível determinar se L é:

a) Vazia;

b) Finita;

c) Infinita.

Prova: Suponha L uma LLC.

Vazia. Seja G = (V, T, P, S), GLC tal que GERA(G) = L. Seja G'= (V', T', P', S) equivalente a G, eliminando os símbolos inúteis. Se P' for vazio, então L é vazia.

Finita e Infinita. Seja G = (V, T, P, S) uma GLC tal que GERA(G) = L. Seja G' = (V', T', P', S) equivalente a G na Forma Normal de Chomsky. Se existe A, variável de V' tal que:

- $A \rightarrow BC$, ou seja, A é referenciada no lado esquerdo de uma produção que não gera diretamente terminais;
- $X \rightarrow YA$ ou $X \rightarrow AY$, ou seja, se A é referenciada no lado direito de alguma produção;
- existe um ciclo em A do tipo $A \Rightarrow^{+} \alpha A \beta$;

então A é capaz de gerar palavras de qualquer tamanho e, conseqüentemente, a linguagem é infinita. Caso não exista tal A, então a linguagem é finita. ❑

3.9 Algoritmos de Reconhecimento

Verificar se uma determinada palavra pertence ou não a uma linguagem é uma das principais questões relacionadas com o estudo de Linguagens Formais. Um "dispositivo de reconhecimento" de uma classe de linguagens pode ser especificado como um modelo de autômato ou como um algoritmo

implementável em um computador. Em qualquer caso, é importante determinar a "quantidade de recursos" (por exemplo: tempo e espaço) que o dispositivo necessita para realizar o reconhecimento. Deve-se destacar que o objetivo em questão é gerar dispositivos de reconhecimento válidos para qualquer linguagem dentro de uma classe. Os algoritmos apresentados a seguir são específicos para as Linguagens Livres do Contexto.

Os algoritmos de reconhecimento que seguem são construídos a partir de uma gramática que define a linguagem. Os reconhecedores gerados usando Autômato com Pilha são muito simples, mas, em geral, ineficientes. Para uma entrada w, seu tempo de processamento é proporcional a $k^{|w|}$ (o valor de k depende do autômato), não sendo recomendáveis para entradas de tamanhos consideráveis. Existe uma série de algoritmos bem mais eficientes, com tempo de processamento proporcional a $|w|^3$ ou até um pouco menos. Não é provado se o tempo proporcional a $|w|^3$ é efetivamente necessário para que um algoritmo genérico reconheça Linguagens Livres do Contexto.

Os reconhecedores podem ser, basicamente, de dois tipos, como segue:

a) *Top-Down* ou *Preditivo*. Constrói uma árvore de derivação para a palavra de entrada (a ser reconhecida) a partir da raiz (símbolo inicial da gramática), gerando os ramos em direção às folhas (símbolos terminais que compõem a palavra);

b) *Bottom-Up*. É, basicamente, o oposto do *top-down*, partindo das folhas e construindo a árvore de derivação em direção à raiz.

3.9.1 Autômato com Pilha como Reconhecedor

A construção de reconhecedores usando Autômato com Pilha é relativamente simples e imediata, havendo uma relação quase direta entre as produções da gramática e as transições do autômato. Os algoritmos apresentados são do tipo *top-down* e simulam a derivação mais à esquerda da palavra a ser reconhecida. A facilidade de não-determinismo é usada para testar as diversas produções alternativas da gramática para gerar os símbolos terminais.

Autômato com Pilha Gerado a Partir de uma Gramática na Forma Normal de Greibach

Já foi mostrado que qualquer Linguagem Livre do Contexto pode ser especificada por um Autômato com Pilha. O algoritmo em questão define um AP a partir de uma Gramática na Forma Normal de Greibach. Como cada produção (na FNG) gera exatamente um terminal, uma palavra w é gerada

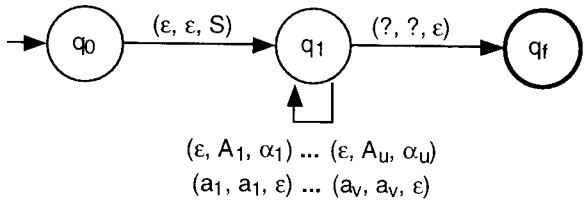


Figura 3.16 Grafo do AP descendente construído a partir de uma GLC sem recursão à esquerda

em $|w|$ etapas de derivação. Entretanto, como cada variável pode ter mais de uma produção associada, é necessário testar, sistematicamente, as diversas alternativas. Assim, o número de passos para reconhecer w é proporcional a $k^{|w|}$, onde k depende do AP (uma aproximação para k é a metade do número médio de produções associadas às diversas variáveis). Portanto, o Autômato construído possui um tempo de reconhecimento proporcional ao expoente em $|w|$, o que pode ser muito ineficiente para entradas mais longas.

Autômato com Pilha Descendente

O Autômato com Pilha Descendente é uma forma alternativa de construir um AP a partir de uma Gramática Livre do Contexto. Trata-se de um algoritmo igualmente simples e com o mesmo nível de eficiência. A construção é gerada a partir de uma gramática sem recursão à esquerda e simula a derivação mais à esquerda, como segue:

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

Definição 3.24 Algoritmo para Construção de um AP Descendente.

A *Algoritmo para Construção de um Autômato com Pilha Descendente* M a partir de uma gramática $G = (V, T, P, S)$, sem recursão à esquerda, é como segue e é ilustrada na Figura 3.16:

$M = (T, \{q_0, q_1, q_f\}, \delta, q_0, \{q_f\}, V \cup T)$, onde:

$\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, S)\}$

$\delta(q_1, \varepsilon, A) = \{(q_1, \alpha) \mid A \rightarrow \alpha \in P\}$, para toda a variável A de V

$\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$, para todo o terminal a de T

$\delta(q_1, ?, ?) = \{(q_f, \varepsilon)\}$ ❑

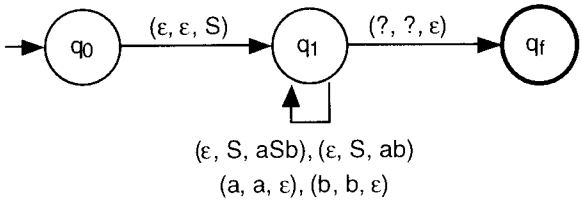


Figura 3.17 Grafo de um AP

EXEMPLO 17 Autômato com Pilha Descendente.

A linguagem $L_5 = \{a^n b^n \mid n \geq 1\}$, representada pela gramática (sem recursão à esquerda:

$G_5' = (\{S\}, \{a, b\}, P_5', S)$, onde:
$P_5' = \{S \rightarrow aSb \mid ab\}$

é reconhecida pelo seguinte Autômato com Pilha Descendente o qual é ilustrado na Figura 3.17 (compare a gramática e o autômato com os introduzidos no Exemplo 15):

$$M_5' = (\{a, b\}, \{q_0, q_1, q_f\}, \delta_5', q_0, \{q_f\}, \{S, a, b\})$$ ❑

3.9.2 Algoritmo de Cocke-Younger-Kasami

O Algoritmo de Cocke-Younger-Kasami foi desenvolvido independentemente por Cocke, Younger e Kasami, em 1965. É construído sobre uma gramática na Forma Normal de Chomsky. Gera *bottom-up* todas as árvores de derivação da entrada em um tempo de processamento proporcional a $|w|^3$.

A idéia básica do algoritmo é a construção de uma tabela triangular de derivação, sendo que cada célula representa o conjunto de raízes que pode gerar a correspondente sub-árvore.

Definição 3.25 Algoritmo de Cocke-Younger-Kasami.

Suponha $G = (V, T, P, S)$ uma gramática na Forma Normal de Chomsky onde $T = \{a_1, a_2, ..., a_t\}$ e suponha $w = a_1 a_2 ... a_n$ uma entrada a ser verificada. O *Algoritmo de Cocke-Younger-Kasami* (*CYK*) é composto pelas seguintes etapas onde V_{r_s} representa as células de uma tabela triangular de derivação a qual é ilustrada na Figura 3.18:

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

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

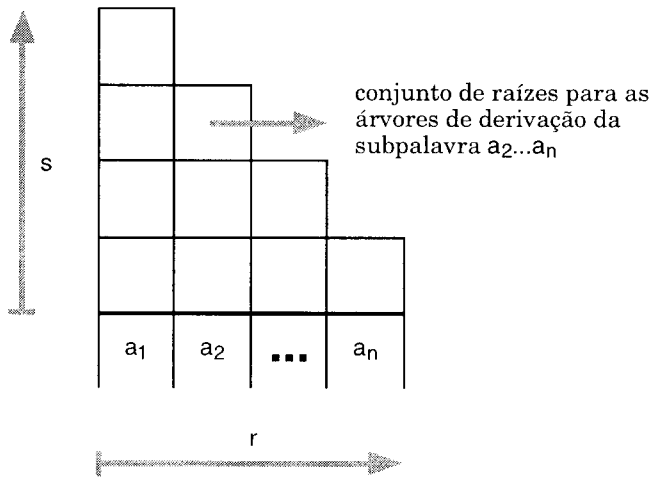


Figura 3.18 Tabela triangular de derivação

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

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

Note-se que:

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

c) *Condição de aceitação da entrada.* Se o símbolo inicial da gramática pertence ao vértice V_{1_n} (raiz da árvore de derivação de toda palavra), então a entrada é aceita. ❑

EXEMPLO 18 Algoritmo de Cocke-Younger-Kasami.

Considere a seguinte gramática:

$G = (\{S, A\}, \{a, b\}, P, S)$, onde:
$P = \{S \rightarrow AA \mid AS \mid b, A \rightarrow SA \mid AS \mid a\}$

A tabela triangular de derivação para a palavra de entrada abaab é ilustrada na Figura 3.19. ❑

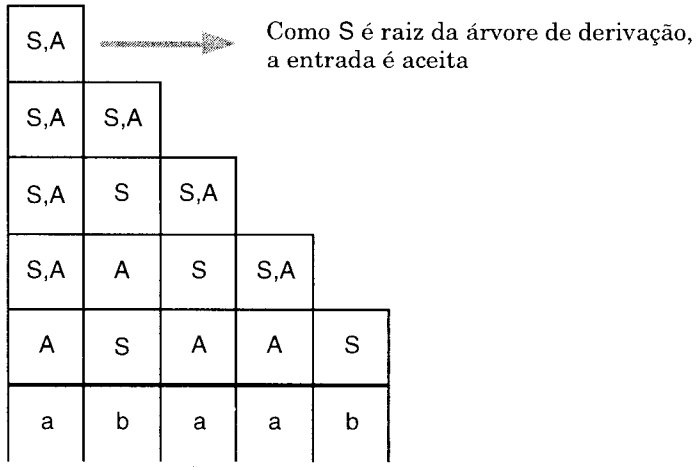


Figura 3.19 Tabela triangular de derivação

3.9.3 Algoritmo de Early

O Algoritmo EARLY foi desenvolvido em 1968. É considerado o mais rápido algoritmo de reconhecimento conhecido para Gramáticas Livres do Contexto. É construído a partir de uma GLC qualquer e possui tempo de processamento proporcional a $|w|^3$. Entretanto, para gramáticas não-ambígüas, pode ser implementado em um tempo proporcional a $|w|^2$. Adicionalmente, para muitas gramáticas de interesse prático, o tempo é proporcional ao tamanho da entrada, ou seja, $|w|$.

Trata-se de um algoritmo do tipo *top-down*, que parte do símbolo inicial e executa sempre a derivação mais à esquerda. Cada ciclo gera um terminal, o qual é comparado com o símbolo da entrada. A comparação com sucesso determina a construção de um conjunto de produções que, potencialmente, pode gerar o próximo símbolo.

Definição 3.26 Algoritmo de Early.

Suponha $G = (V, T, P, S)$ uma Gramática Livre do Contexto qualquer e $w = a_1a_2...a_n$ uma palavra a ser verificada. No que segue, o símbolo "." é usado como um marcador, antecedendo a posição, em cada produção, que será analisada na tentativa de gerar o próximo símbolo terminal. Adicionalmente, o sufixo "/u" adicionado a cada produção indica o u-ésimo ciclo em que esta produção passou a ser considerada. As etapas do *Algoritmo de Early* são as seguintes:

a) *Construção de D_0, o primeiro conjunto de produções.* São incluídas em D_0 todas as produções que partem do símbolo inicial S, bem como todas as

produções que podem ser aplicadas em sucessivas derivações mais à esquerda (a partir de S). O algoritmo desta etapa é o seguinte:

$D_0 = \varnothing$
para toda $S \rightarrow \alpha \in P$ (1)
faça $D_0 = D_0 \cup \{ S \rightarrow .\alpha/0 \}$
repita (2)
para toda $A \rightarrow .B\beta/0 \in D_0$
faça para toda $B \rightarrow \phi \in P$
faça $D_0 = D_0 \cup \{ B \rightarrow .\phi/0 \}$
até que o cardinal de D_0 não aumente

No algoritmo acima, tem-se que:

(1) representa as produções que partem de S;

(2) inclui todas as produções que podem derivar (mais à esquerda) o próximo símbolo;

b) *Construção dos demais conjuntos de produção.* São construídos n conjuntos de produção a partir de D_0, onde $n = |w|$. Ao gerar o símbolo a_r de w, o algoritmo constrói o conjunto D_r, contendo as produções que potencialmente podem gerar o símbolo a_{r+1}. O algoritmo desta etapa é o seguinte:

para r variando de 1 até n (1)
faça $D_r = \varnothing$;
para toda $A \rightarrow \alpha.a_r\beta/s \in D_{r-1}$ (2)
faça $D_r = D_r \cup \{ A \rightarrow \alpha a_r.\beta/s \}$;
repita
para toda $A \rightarrow \alpha.B\beta/s \in D_r$ (3)
faça para toda $B \rightarrow \phi \in P$
faça $D_r = D_r \cup \{ B \rightarrow .\phi/r \}$
para toda $A \rightarrow \alpha./s$ de D_r (4)
faça para toda $B \rightarrow \beta.A\phi/k \in D_s$
faça $D_r = D_r \cup \{ B \rightarrow \beta A.\phi/k \}$
até que o cardinal de D_r não aumente

No algoritmo acima, tem-se que:

(1) cada ciclo gera um conjunto de produções D_r;

(2) gera o símbolo a_r;

(3) inclui todas as produções que podem derivar (mais à esquerda) o próximo símbolo (análogo ao realizado na etapa a) acima);

(4) uma subpalavra de w foi reduzida à variável A: inclui em D_r todas as produções de D_s que referenciaram .A;

c) *Condição de aceitação da entrada.* Se uma produção da forma $S \rightarrow \alpha./0$ pertence a D_n, então a palavra w de entrada foi aceita. Deve-se reparar que $S \rightarrow \alpha./0$ é uma produção que:

- parte do símbolo inicial S;
- foi incluída em D_0 ("/0");
- todo o lado direito da produção foi analisado com sucesso (o marcador "." está no final de α). ❑

Note-se que, para otimizar as etapas a) e b) do algoritmo acima, os ciclos `repita-até` podem ser restritos exclusivamente às produções recentemente incluídas em D_f ou em D_0 ainda não-analisadas.

EXEMPLO 19 Algoritmo de Early.

A seguinte gramática gera expressões com parênteses balanceados e duas operações (análoga à "expressão simples" da linguagem PASCAL):

G = ({E, T, F}, {+, *, [,], x}, P, E), onde:
P = {E → T | E+T, T → F | T*F, F → [E] | x}

O reconhecimento da palavra x*x é como segue:

D_0:

E → .T/0	produções que partem
E → .E+T/0	do símbolo inicial;
T → .F/0	produções que podem ser aplicadas
T → .T*F/0	em derivação mais à esquerda
F → .[E]/0	a partir do símbolo inicial.
F → .x/0	

D_1: reconhecimento de x em x*x

F → x./0	x foi reduzido à F;
T → F./0	inclui todas as produções de D_0 que
T → T.*F/0	referenciaram .F direta ou indiretamente, (pois F → x./0)
E → T./0	movendo o marcador "."
E → E.+T/0	um símbolo para a direita.

D_2: reconhecimento de * em x*x

T → T*.F/0	gerou *; o próximo será gerado por F;
F → .[E]/2	inclui todas as produções de P que
F → .x/2	podem gerar o próximo terminal a partir de F.

D_3: reconhecimento de x em x*x

F → x./2	x foi reduzido à F;
T → T*F./0	incluído de D_2 (pois F → x./2); a entrada foi reduzida à T;
E → T./0	incluído de D_0 (pois T → T*F./0); a entrada foi reduzida à E;
T → T.*F/0	incluído de D_0 (pois T → T*F./0);
E → E.+T/0	incluído de D_0 (pois E → T./0).

Como w = x*x foi reduzida ao símbolo inicial E, ou seja, E → T./0 pertence a D_3, a entrada foi aceita. ❑

3.10 Exercícios

3.1 Sobre as Linguagens Livres do Contexto:

a) Qual a importância do seu estudo?

b) Exemplifique suas aplicações (para os formalismos de autômato e gramática);

c) Faça um quadro comparativo com as Linguagens Regulares, destacando as principais características, semelhanças e diferenças.

3.2 Desenvolva Gramáticas Livres do Contexto que gerem as seguintes linguagens:

a) $L_1 = \varnothing$

b) $L_2 = \{\varepsilon\}$

c) $L_3 = \{a, b\}^*$

d) $L_4 = \{w \mid w \text{ é palíndromo em } \{a, b\}^*\}$, onde palíndromo significa que $w = w^r$

e) $L_5 = \{ww^r \mid w \text{ é palavra de } \{a, b\}^*\}$. Qual a diferença entre as linguagens L_4 e L_5?

f) $L_6 = \{a^i b^j c^k \mid i = j \text{ ou } j = k \text{ e } i, j, k \geq 0\}$

g) $L_7 = \{w \mid w \text{ é palavra de } \{x, y, (,)\}^* \text{ com parênteses balanceados}\}$

h) $L_8 = \{w \mid w \text{ é Expressão Regular sobre o alfabeto } \{x\}\}$

3.3 Desenvolva Autômatos com Pilha que reconheçam as seguintes linguagens:

a) $L_1 = \varnothing$

b) $L_2 = \{\varepsilon\}$

c) $L_3 = \{a, b\}^*$

d) $L_4 = \{w \mid w \text{ é palíndromo em } \{a, b\}^*\}$

e) $L_8 = \{w \mid w \text{ é Expressão Regular sobre o alfabeto } \{x\}\}$

f) $L_9 = \{ua^n va^n w \mid n \in \{1, 2\}, u, v, w \text{ são palavras de } \{a, b\}^* \text{ e } |u| = |v| = 5\}$

3.4 Construa uma Gramática Livre do Contexto e um Autômato com Pilha que representem a seguinte linguagem de programação:

- os `comandos` podem ser como segue:

`simples`,
`composto`,
`enquanto-faça`,
`repita-até`;

- `comando simples`:
  qualquer palavra de {`a`, `b`}*;
- `comando composto`:
  `i` (início),
  seguido de um ou mais comandos separados por ";",
  seguidos de `t` (término);
- `comando enquanto-faça`:
  `e` (enquanto),
  seguido de uma `expressão`,
  seguida de `f` (faça),
  seguida de um `comando`;
- `comando repita-até`:
  `r` (repita),
  seguido de um `comando`,
  seguido de `a` (até),
  seguida de uma `expressão`;
- `expressão`: como definida na linguagem L_7 no Exercício 3.2, excetuando-se a palavra vazia.

3.5 Considere a seguinte gramática:
$G = (\{S\}, \{a, b\}, P, S)$, onde:
$P = \{S \rightarrow SS \mid aSa \mid bSb \mid \varepsilon\}$

a) Qual a linguagem gerada?

b) A gramática é ambígua?

c) Para a palavra aabbaaaa:
- construa uma árvore de derivação;
- para a árvore construída, determine as derivações mais à esquerda e a mais à direita.

3.6 No Exemplo 5, foi afirmado que a gramática abaixo é ambígua:
$G_2 = (\{E\}, \{+, *, [,], x\}, P_2, E)$, onde:
$P_2 = \{E \rightarrow E+E \mid E*E \mid [E] \mid x\}$

Construa uma gramática não-ambígüa equivalente.

Sugestão: faça uma pesquisa bibliográfica e verifique como são definidas, usando gramáticas, expressões em algumas linguagens de programação

reais. A definição de uma "expressão simples" na linguagem Pascal é um bom exemplo.

3.7 Para qualquer Linguagem Livre do Contexto é possível garantir que existe um Autômato com Pilha que aceita a linguagem e que sempre pára para qualquer entrada? Por quê?

3.8 Demonstre que se L é uma Linguagem Livre do Contexto, então L^* também é Livre do Contexto.

3.9 Demonstre que o Autômato com Pilha sem usar a estrutura de pilha para armazenar informações do processamento possui o mesmo poder computacional do Autômato Finito.

3.10 Estenda a função programa do Autômato com Pilha usando como argumento um estado e uma palavra, de forma similar à realizada para os Autômatos Finitos.

3.11 Considere a seguinte gramática:

$G = (\{S, X, Y, Z, A, B\}, \{a, b, u, v\}, P, S)$, onde:

$P = \{ S \rightarrow XYZ,$
$X \rightarrow AXA \mid BXB \mid Z \mid \varepsilon,$
$Y \rightarrow AYB \mid BYA \mid Z \mid \varepsilon,$
$A \rightarrow a, B \rightarrow b$
$Z \rightarrow Zu \mid Zv \mid \varepsilon \}$

a) Qual a linguagem gerada?

b) Simplifique a gramática.

3.12 Sobre os algoritmos de simplificação de Gramáticas Livres do Contexto:

a) por que, no algoritmo referente ao tratamento dos símbolos inúteis, se a etapa *qualquer símbolo é atingível a partir do símbolo inicial* for executada antes da etapa *qualquer variável gera palavra de terminais*, o resultado pode não ser o esperado?

b) por que a execução combinada dos algoritmos de simplificação (produções vazias, produções da forma $A \rightarrow B$ e símbolos inúteis) não deve ser realizada em qualquer ordem?

3.13 Para as gramáticas abaixo, construa as gramáticas equivalentes na Forma Normal de Chomsky e na de Greibach:

a) $L8 = \{ w \mid w$ é Expressão Regular sobre o alfabeto $\{x\}\}$ introduzida no Exercício 3.3.

b) Gramática construída para a linguagem de programação do Exercício 3.4;

c) Faça um comparativo das gramáticas originais com as correspondentes na Forma Normal de Chomsky e na Forma Normal de Greibach.

3.14 Explique intuitivamente por que e prove que as seguintes linguagens não são Livres do Contexto:

a) $L_{10} = \{ ww \mid w \text{ é palavra de } \{a, b\}^* \}$

b) $L_{11} = \{ a^n b^n a^m \mid n \geq 0, m \geq 0 \text{ e } n \neq m \}$

3.15 Demonstre que a Classe das Linguagens Livres do Contexto é fechada para as seguintes operações:

a) União, usando o formalismo de gramática;

b) Concatenação, usando o formalismo de autômato.

3.16 As linguagens geradas pelas gramáticas cujas produções estão representadas abaixo são vazias, finitas ou infinitas?

a)

$S \rightarrow AB \mid CA$
$A \rightarrow a$
$B \rightarrow BC$
$C \rightarrow AB \mid \varepsilon$

b)

$S \rightarrow aS \mid aSbS \mid X$
$X \rightarrow SS$

3.17 Por que os algoritmos de reconhecimento baseados em Autômatos com Pilha são tão ineficientes em termos de tempo de processamento?

3.18 No algoritmo de reconhecimento Autômato com Pilha Descendente, qual a conseqüência se a gramática usada tiver recursão à esquerda?

3.19 Para as gramáticas da linguagem $L8 = \{ w \mid w \text{ é Expressão Regular sobre o alfabeto } \{x\} \}$ construídas no Exercício 3.13, faça o reconhecimento da entrada $(x+x)^*$ para cada um dos seguintes algoritmos de reconhecimento:

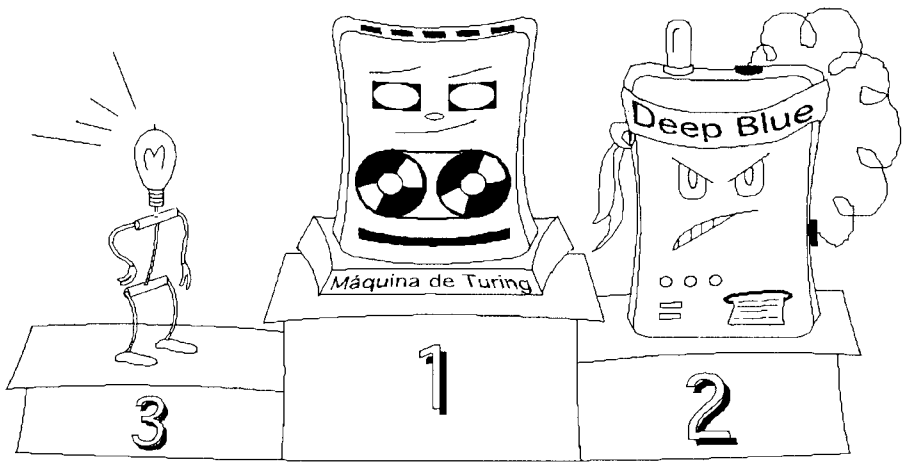
a) Autômato com Pilha a partir da gramática na Forma Normal de Greibach;

b) Autômato com Pilha Descendente;

c) Cocke-Younger-Kasami (CYK);

d) Early.

Campeonato de Autômatos



4 Linguagens Enumeráveis Recursivamente e Sensíveis ao Contexto

Ciência da Computação é o conhecimento sistematizado relativo à computação. Sua origem é remota, tendo exemplos na antiga Grécia (século III a.C., no desenho de algoritmos por Euclides) e Babilônia (com estudos sobre complexidade e reducibilidade de problemas). No início do século XX, diversas pesquisas foram desenvolvidas com o objetivo de definir um modelo computacional suficientemente genérico, capaz de implementar qualquer "função computável".

Em 1936, Alan Turing propôs um modelo conhecido como *Máquina de Turing*. Atualmente, a Máquina de Turing é aceita como uma formalização de um *procedimento efetivo* (*algoritmo* ou *função computável*), ou seja, uma seqüência finita de instruções, as quais podem ser realizadas mecanicamente, em um tempo finito.

Ainda em 1936, Alonzo Church apresentou a *Hipótese de Church*, a qual afirma que qualquer função computável pode ser processada por uma Máquina de Turing, ou seja, existe um procedimento expresso na forma de uma Máquina de Turing capaz de processar a função. Contudo, como a noção intuitiva de procedimentos não é matematicamente precisa, é impossível demonstrar formalmente se a Máquina de Turing é, efetivamente, o mais genérico dispositivo de computação. Entretanto, foi mostrado que todos os demais modelos propostos possuem, no máximo, a mesma capacidade computacional de Turing, o que reforça a Hipótese de Church.

As *Linguagens Enumeráveis Recursivamente* ou *Tipo 0* são aquelas que podem ser reconhecidas por uma Máquina de Turing. Considerando que, segundo a Hipótese de Church, a Máquina de Turing é o mais geral dispositivo de computação, então a Classe das Linguagens Enumeráveis Recursivamente representa o conjunto de todas as linguagens que podem ser reconhecidas mecanicamente e em um tempo finito.

Analogamente às demais classes de linguagens, é possível representar as Linguagens Enumeráveis Recursivamente usando um formalismo axiomático ou gerador, na forma de gramática, denominado *Gramática Irrestrita*. Como o próprio nome indica, uma Gramática Irrestrita não possui qualquer restrição sobre a forma das produções.

As *Linguagens Sensíveis ao Contexto* ou Tipo 1 estão contidas propriamente nas Enumeráveis Recursivamente (bem como nas Recursivas, introduzidas adiante). Entretanto, incluem praticamente todas as linguagens aplicadas.

Como nas demais classes de linguagens, o estudo da Classe das Linguagens Sensíveis ao Contexto pode ser desenvolvido a partir de formalismos gerador e reconhecedor, como segue:

a) *Gramática Sensível ao Contexto*. O termo "sensível ao contexto" deriva do fato de que o lado esquerdo das produções da gramática pode ser uma palavra de variáveis ou terminais, definindo um "contexto" de derivação;

b) *Máquina de Turing com Fita Limitada*. Trata-se de uma Máquina de Turing com limitação no tamanho da fita a qual, portanto, é finita. Deve-se destacar o fato de que não é conhecido se a facilidade de não-determinismo aumenta ou não o poder computacional das Máquinas de Turing com Fita Limitada.

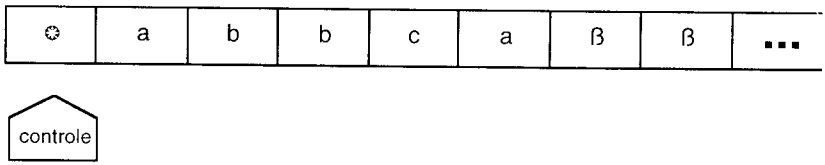


Figura 4.1 Fita e unidade de controle de uma Máquina de Turing

4.1 Máquina de Turing

A Máquina de Turing para ser considerada como um modelo formal de *procedimento efetivo*, *algoritmo* ou *função computável* deve satisfazer às seguintes propriedades, entre outras:

a) A descrição do algoritmo deve ser finita;

b) Deve consistir de passos:

- discretos;
- executáveis mecanicamente;
- em um tempo finito.

O modelo proposto por Alan Turing em 1936, conhecido como Máquina de Turing, consiste basicamente de 3 partes:

a) *Fita*. Usada simultaneamente como dispositivo de entrada, saída e memória de trabalho;

b) *Unidade de Controle*. Reflete o estado corrente da máquina. Possui uma unidade de leitura e gravação (cabeça da fita) a qual acessa uma célula da fita de cada vez e movimenta-se para a esquerda ou direita;

c) *Programa* ou *Função de Transição*. Função que comanda as leituras e gravações, o sentido de movimento da cabeça e define o estado da máquina.

A fita é finita à esquerda e infinita à direita, sendo dividida em células, onde cada uma armazena um símbolo. Os símbolos podem pertencer ao alfabeto de entrada, ao alfabeto auxiliar ou ainda, ser "branco" ou "marcador de início de fita".

Inicialmente, a palavra a ser processada (ou seja, a informação de entrada para a máquina) ocupa as células mais à esquerda, após o marcador de início de fita, ficando as demais com "branco", como ilustrado na Figura 4.1, onde ß e ⊙ representam "branco" e "marcador de início de fita", respectivamente.

A unidade de controle possui um número finito e predefinido de estados. A *cabeça da fita* lê o símbolo de uma célula de cada vez e grava um novo símbolo. Após a leitura/gravação, a cabeça move uma célula para a direita ou esquerda. O símbolo gravado e o sentido do movimento são definidos pelo programa.

O programa é uma função que, dependendo do estado corrente da máquina e do símbolo lido, determina o símbolo a ser gravado, o sentido do movimento da cabeça e o novo estado.

Definição 4.1 Máquina de Turing.

Uma *Máquina de Turing* é uma 8-upla:

$$M = (\Sigma, Q, \delta, q_0, F, V, \beta, \circledcirc)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- Q *conjunto de estados* possíveis da máquina o qual é finito;
- δ *programa* ou *função de transição*:

  $$\delta: Q \times (\Sigma \cup V \cup \{\beta, \circledcirc\}) \rightarrow Q \times (\Sigma \cup V \cup \{\beta, \circledcirc\}) \times \{E, D\}$$

  a qual é uma função parcial;
- q_0 *estado inicial* da máquina tal que q_0 é elemento de Q;
- F *conjunto de estados finais* tal que F está contido em Q;
- V *alfabeto auxiliar* (pode ser vazio);
- β símbolo especial *branco*;
- $\circledcirc$ *símbolo* ou *marcador de início* da fita. ❑

O símbolo de início de fita sempre ocorre exatamente uma vez e na célula mais à esquerda da fita, auxiliando na identificação de que a cabeça da fita se encontra na célula mais à esquerda da fita.

A função programa, considera o estado corrente e o símbolo lido da fita para determinar o novo estado, o símbolo a ser gravado e sentido de movimento da cabeça onde esquerda e direita são representados por E e D, respectivamente. A função programa pode ser interpretada como um grafo finito direto, como ilustrado na Figura 4.2. Neste caso, os estados inicial e finais são representados como nos Autômatos Finitos.

O processamento de uma Máquina de Turing $M = (\Sigma, Q, \delta, q_0, F, V, \beta, \circledcirc)$, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa a partir do estado inicial q_0 e da cabeça posicionada na célula mais à esquerda da fita até ocorrer uma condição de parada. O processamento de M para a entrada w, pode parar ou ficar em *loop* infinito. A parada pode ser de duas maneiras: aceitando ou rejeitando a entrada w. As condições de parada são as seguintes:

a) A máquina assume um estado final: a máquina pára e a palavra de entrada é aceita;

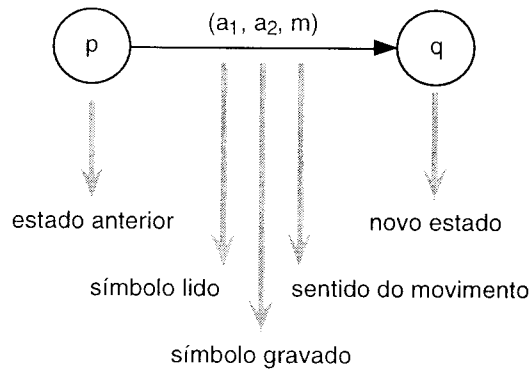


Figura 4.2 Representação da função programa como um grafo

b) A função programa é indefinida para o argumento (símbolo lido e estado corrente): a máquina pára e a palavra de entrada é rejeitada;

c) O argumento corrente da função programa define um movimento à esquerda e a cabeça da fita já se encontra na célula mais à esquerda: a máquina pára e a palavra de entrada é rejeitada.

Para definir formalmente o comportamento de uma Máquina de Turing, é necessário estender a definição da função programa usando como argumento um estado e uma palavra. Esta extensão é sugerida como exercício.

As seguintes notações são adotadas, as quais são análogas às usadas para Autômatos com Pilha (suponha que M é uma Máquina de Turing):

a) ACEITA(M) ou L(M): conjunto de todas as palavras de Σ^* aceitas por M;

b) REJEITA(M): conjunto de todas as palavras de Σ^* rejeitadas por M;

c) LOOP(M): conjunto de todas as palavras de Σ^* para as quais M fica processando indefinidamente.

Da mesma forma, as seguintes afirmações são verdadeiras:

- $ACEITA(M) \cap REJEITA(M) \cap LOOP(M) = \varnothing$
- $ACEITA(M) \cup REJEITA(M) \cup LOOP(M) = \Sigma^*$
- o complemento de:

  ACEITA(M) é $REJEITA(M) \cup LOOP(M)$
  REJEITA(M) é $ACEITA(M) \cup LOOP(M)$
  LOOP(M) é $ACEITA(M) \cup REJEITA(M)$

EXEMPLO 1 Máquina de Turing - Duplo Balanceamento.

Considere a linguagem:

$$L_1 = \{ a^n b^n \mid n \geq 0 \}$$

A Máquina de Turing:

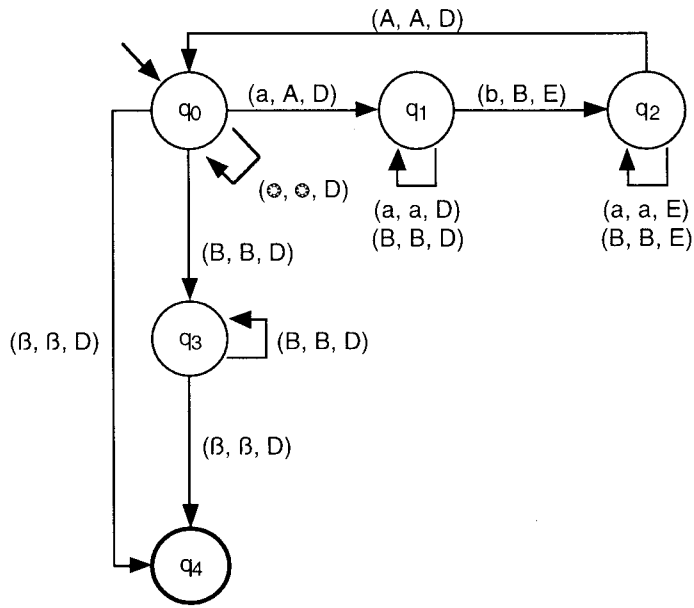


Figura 4.3 Máquina de Turing

$$M_1 = (\{a, b\}, \{q_0, q_1, q_2, q_3, q_4\}, \delta_1, q_0, \{q_4\}, \{A, B\}, ß, ⊙)$$

ilustrada na Figura 4.3 e onde δ_1 é como na tabela abaixo, é tal que:

$ACEITA(M_1) = L_1$

$REJEITA(M_1) = \Sigma^* - L_1$

e, portanto, $LOOP(M_1) = \varnothing$.

δ_1	⊙	a	b	A	B	ß
q_0	$(q_0, ⊙, D)$	(q_1, A, D)			(q_3, B, D)	$(q_4, ß, D)$
q_1		(q_1, a, D)	(q_2, B, E)		(q_1, B, D)	
q_2		(q_2, a, E)		(q_0, A, D)	(q_2, B, E)	
q_3					(q_3, B, D)	$(q_4, ß, D)$
q_4						

O algoritmo apresentado reconhece o primeiro símbolo a, o qual é marcado como A, e movimenta a cabeça da fita à direita, procurando o b correspondente, o qual é marcado como B. Este ciclo é repetido sucessivamente até identificar para cada a o seu correspondente b.

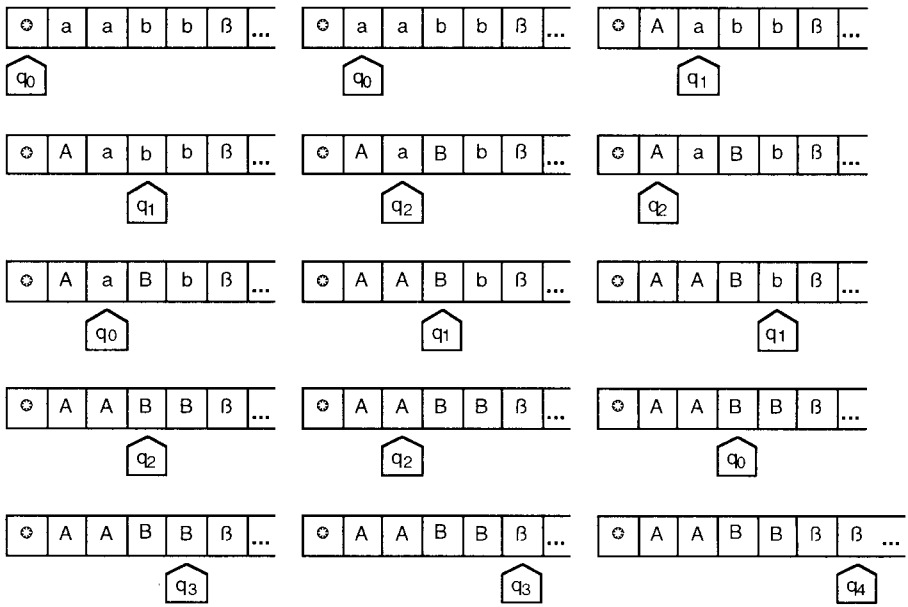


Figura 4.4 Seqüência de processamento de uma Máquina de Turing

Adicionalmente, o algoritmo garante que qualquer outra palavra que não esteja na forma $a^n b^n$ é rejeitada. Note-se que o símbolo de início de fita não tem influência na solução proposta.

A Figura 4.4 ilustra a seqüência do processamento da Máquina de Turing M_1 para a entrada w = aabb. ❑

4.2 Modelos Equivalentes à Máquina de Turing

Uma das razões para considerar a Máquina de Turing como o mais geral dispositivo de computação é o fato de que todos os demais modelos e máquinas propostos, bem como diversas modificações da Máquina de Turing, possuem, no máximo, o mesmo poder computacional da Máquina de Turing. As modificações apresentadas a seguir são freqüentemente usadas.

a) *Autômato com Múltiplas Pilhas.* Conforme citado no estudo das Linguagens Livres do Contexto, o poder computacional do Autômato com Duas Pilhas (A2P) é equivalente ao da Máquina de Turing. Adicionalmente, um maior número de pilhas (AnP, n > 2) também não

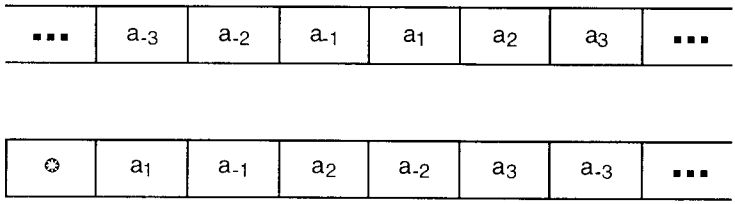


Figura 4.5 Simulação de uma fita infinita à esquerda e à direita

aumenta a capacidade computacional. A definição formal do A2P e AnP e a equivalência ao modelo da Máquina de Turing é sugerida como exercício;

b) *Máquina de Turing Não-Determinística.* A facilidade de não-determinismo não aumenta o poder computacional da Máquina de Turing;

c) *Máquina de Turing com Fita Infinita à Esquerda e à Direita.* A modificação da definição básica da Máquina de Turing permitindo que a fita seja infinita dos dois lados não aumenta o seu poder computacional. Na realidade, a fita infinita à esquerda e à direita pode ser facilmente simulável por uma fita tradicional, como ilustrado na Figura 4.5 onde as células pares representam a parte direita da fita e as ímpares a parte esquerda;

d) *Máquina de Turing com Múltiplas Fitas.* A Máquina de Turing com múltiplas fitas possui k fitas infinitas à esquerda e à direita e k cabeças de fita. A função programa é como segue:
   - dependendo do estado corrente da máquina e do símbolo lido em cada uma das fitas;
   - grava um novo símbolo em cada uma das fitas;
   - move cada uma das cabeças independentemente;
   - a máquina assume um (único) novo estado.

   Inicialmente, a palavra de entrada é armazenada na primeira fita, ficando as demais com valor branco;

e) *Máquina de Turing Multidimensional.* Neste modelo, a fita tradicional é substituída por uma estrutura do tipo arranjo k-dimensional, infinita em todas as 2k direções;

f) *Máquina de Turing com Múltiplas Cabeças.* A Máquina de Turing com esta modificação possui k cabeças de leitura e gravação sobre a mesma fita. Cada cabeça possui movimento independente. Assim, o processamento depende do estado corrente e do símbolo lido em cada uma das cabeças;

g) *Combinações de Modificações sobre a Máquina de Turing.* A combinação de algumas ou todas as modificações apresentadas não aumenta o poder computacional da Máquina de Turing. Por exemplo, uma Máquina de Turing não-determinística com múltiplas fitas e múltiplas cabeças pode ser simulada por uma Máquina de Turing tradicional.

4.3 Hipótese de Church

A. M. Turing propôs, em 1936, um modelo abstrato de computação, conhecido como Máquina de Turing, com o objetivo de explorar os limites da capacidade de expressar soluções de problemas. Trata-se, portanto, de uma proposta de definição formal da noção intuitiva de algoritmo. Diversos outros trabalhos, como *Máquina de Post* (Post - 1936) e *Funções Recursivas* (Kleene - 1936), resultaram em conceitos equivalentes ao de Turing. O fato de todos estes trabalhos independentes gerarem o mesmo resultado em termos de capacidade de expressar computabilidade é um forte reforço no que é conhecido como *Hipótese de Church* ou *Hipótese de Turing-Church*:

> *"A capacidade de computação representada pela Máquina de Turing é o limite máximo que pode ser atingido por qualquer dispositivo de computação"*

Em outras palavras, a Hipótese de Church afirma que qualquer outra forma de expressar algoritmos terá no máximo a mesma capacidade computacional da Máquina de Turing. Como a noção de algoritmo ou Função Computável é intuitiva, a Hipótese de Church não é demonstrável.

4.4 Máquinas de Turing como Reconhecedores

Uma linguagem aceita por uma Máquina de Turing é dita Enumerável Recursivamente ou Tipo 0. Historicamente, o termo "enumerável" deriva do fato de que as palavras de qualquer Linguagem Enumerável Recursivamente podem ser enumeradas ("listadas") por uma Máquina de Turing; e "recursivamente" é um termo matemático anterior ao computador, com significado similar ao de "recursão", usado em informática.

A Classe das Linguagens Enumerável Recursivamente inclui algumas para as quais é impossível determinar mecanicamente se uma palavra não pertence à linguagem. Se L é uma destas linguagens, então para qualquer

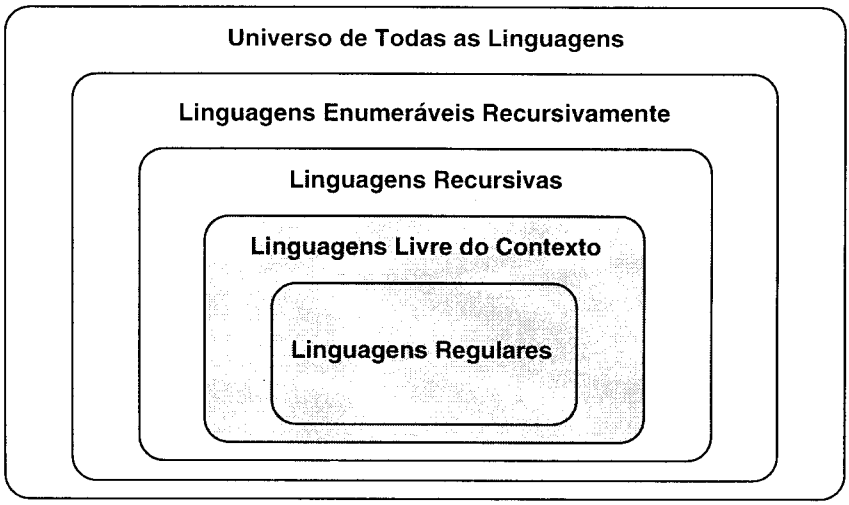


Figura 4.6 Relação entre as classes de linguagens

máquina M que aceita L, existe pelo menos uma palavra w não pertencente a L que, ao ser processada por M, a máquina entra em *loop* infinito. Assim, pode-se afirmar que:

- se w pertence a L, M pára e aceita a entrada;
- se w não pertence a L, M pode parar, rejeitando a palavra ou permanecer processando indefinidamente.

Portanto, é conveniente definir uma subclasse da Classe das Linguagens Enumerável Recursivamente, denominada Classe das Linguagens Recursivas, composta pelas linguagens para as quais existe pelo menos uma Máquina de Turing que pára para qualquer entrada, aceitando ou rejeitando.

O diagrama ilustrado na Figura 4.6 representa a relação entre as classes de linguagens.

4.4.1 Linguagens Enumeráveis Recursivamente

As Linguagens Enumeráveis Recursivamente são definidas a partir das Máquinas de Turing.

Definição 4.2 Linguagem Enumerável Recursivamente ou Tipo 0.

Uma linguagem aceita por uma Máquina de Turing é dita *Linguagem Enumerável Recursivamente* ou *Tipo 0*. ❑

EXEMPLO 2 Linguagem Enumerável Recursivamente.

As seguintes linguagens são exemplos de Linguagens Enumeráveis Recursivamente sendo que, para a primeira, a correspondente Máquina de Turing foi construída no Exemplo 1 (as demais são sugeridas como exercício):

a) $\{a^n b^n \mid n \geq 0\}$

b) $\{w \mid w$ tem o mesmo número de símbolos a e b$\}$

c) $\{a^i b^j c^k \mid i = j$ ou $j = k\}$ ❑

Considerando que, segundo a Hipótese de Church, a Máquina de Turing é o mais geral dispositivo de computação, pode-se afirmar que a Classe das Linguagens Enumeráveis Recursivamente representa todas as linguagens que podem ser reconhecidas ("compiladas") mecanicamente. Trata-se, portanto, de uma classe de linguagens muito rica. Existem, entretanto, conjuntos que não são Enumeráveis Recursivamente, ou seja, linguagens para as quais não é possível desenvolver uma Máquina de Turing que as reconheça.

No teorema a seguir, mostra-se que existe pelo menos uma linguagem que não é Enumeráveis Recursivamente.

Teorema 4.3 Linguagem Não-Enumerável Recursivamente.

Suponha $\Sigma = \{a, b\}$.

a) Suponha que X_i representa o i-ésimo elemento na ordenação lexicográfica de Σ^*, ou seja, $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$;

b) Conforme é proposto nos exercícios, é possível codificar todas as Máquinas de Turing como uma palavra sobre Σ de tal forma que cada código represente uma única Máquinas de Turing. Suponha o conjunto dos códigos ordenados lexicograficamente e suponha que T_i representa o i-ésimo código nesta ordenação.

A linguagem que segue não é Enumerável Recursivamente:

$$L = \{X_i \mid X_i \text{ não é aceita por } T_i\}$$

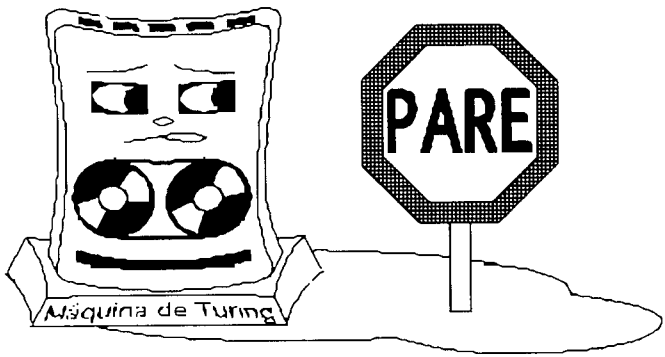
Prova: A prova que segue é por redução ao absurdo. Suponha que L é Enumerável Recursivamente. Então, por definição, existe uma Máquina de Turing que aceita L. Seja T_k a codificação desta Máquina de Turing, ou seja, $ACEITA(T_k) = L$. Assim, as seguintes afirmações são válidas:

a) Por definição de L, X_k pertence a L se, e somente se, X_k não é aceita por T_k;

b) Entretanto, como T_k aceita a linguagem L, X_k pertence a L se, e somente se, X_k é aceita por T_k.

Claramente b) contradiz a). Portanto, é absurdo supor que L é Enumerável Recursivamente. Logo, L não é Enumerável Recursivamente. ❑

Note-se que, na prova do Teorema 4.3, afirma-se que existe uma codificação de todas as Máquinas de Turing. Portanto, o conjunto destas codificações é isomorfo a um subconjunto infinito dos números naturais. Logo, o conjunto de todas as Máquinas de Turing (ou de todos os problemas solucionáveis) é contável. Em contra-partida, é interessante destacar que o cardinal do conjunto das linguagens que não são Enumeráveis Recursivamente (ou dos problemas não-solucionáveis) é infinito e não-contável.



4.4.2 Linguagens Recursivas

As Linguagens Recursivas também são definidas a partir das Máquinas de Turing.

Definição 4.4 Linguagem Recursiva.

Uma linguagem L é dita *Linguagem Recursiva* se existe uma Máquina de Turing M tal que:

a) ACEITA(M) = L

b) REJEITA(M) = Σ^* - L ❑

Ou seja, uma linguagem é Recursiva se existe pelo menos uma Máquina de Turing que aceita a linguagem e sempre pára para qualquer entrada.

EXEMPLO 3 Linguagem Recursiva.

É fácil verificar que as seguintes linguagens são Recursivas:

a) $\{a^n b^n \mid n \geq 0\}$

b) $\{a^n b^n c^n \mid n \geq 0\}$

c) $\{w \mid w \in \{a, b\}^*$ e tem o dobro de símbolos a que b$\}$ ❑

4.4.3 Propriedades das Linguagem Enumeráveis Recursivamente e Recursivas

A seguir são apresentadas algumas das principais propriedades das Linguagens Enumeráveis Recursivamente e das Recursivas.

Teorema 4.5 Complemento de uma Linguagem Recursiva é Recursiva.

Se uma linguagem L sobre um alfabeto Σ qualquer é Recursiva, então o seu complemento $\Sigma^* - L$ também é uma Linguagem Recursiva.

Prova: Suponha L uma Linguagem Recursiva sobre Σ. Então existe M, Máquina de Turing, que aceita a linguagem e sempre pára para qualquer entrada. Ou seja:

$ACEITA(M) = L$
$REJEITA(M) = \Sigma^* - L$
$LOOP(M) = \varnothing$

Seja M' uma Máquina de Turing construída a partir de M, mas invertendo-se as condições de ACEITA por REJEITA e vice-versa (como a inversão pode ser implementada?). Portanto, M' aceita $\Sigma^* - L$ e sempre pára para qualquer entrada. Ou seja:

$ACEITA(M') = \Sigma^* - L$
$REJEITA(M') = L$
$LOOP(M') = \varnothing$

Logo, $\Sigma^* - L$ é uma Linguagem Recursiva. ❑

Teorema 4.6 Linguagem Recursiva × Enumerável Recursivamente.

Uma linguagem L sobre um alfabeto Σ qualquer é Recursiva se, e somente se, L e $\Sigma^* - L$ são Enumeráveis Recursivamente.

Prova:

a) Suponha L uma Linguagem Recursiva sobre Σ. Então, como foi mostrado no Teorema 4.5, $\Sigma^* - L$ é Recursiva. Como toda Linguagem Recursiva também é Enumerável Recursivamente , então L e $\Sigma^* - L$ são Enumeráveis Recursivamente;

b) Suponha L uma linguagem sobre Σ tal que L e $\Sigma^* - L$ são Enumeráveis Recursivamente. Então existem M_1 e M_2, Máquinas de Turing tais que:

$ACEITA(M_1) = L$
$ACEITA(M_2) = \Sigma^* - L$

Seja M Máquina de Turing não-determinística definida conforme esquema ilustrado na Figura 4.7 (como seria, detalhadamente, a definição de M?). Para qualquer palavra de entrada, M aceita se M_1 aceita e M rejeita se M_2 aceita. Portanto, claramente M sempre pára. Logo, L é Recursiva. ❑

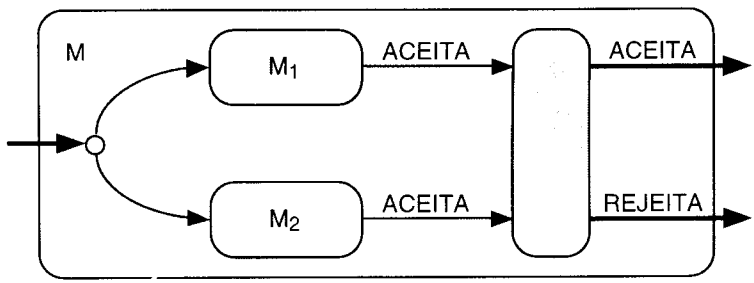


Figura 4.7 Máquina de Turing

Teorema 4.7 Linguagem Recursiva x Enumerável Recursivamente.

A Classe das Linguagens Recursivas está contida propriamente na Classe das Linguagens Enumeráveis Recursivamente.

Prova: Para mostrar que a inclusão é própria, é suficiente mostrar que existe pelos menos uma Linguagem Enumerável Recursivamente que não é Recursiva. A Linguagem Enumerável Recursivamente que segue não é Recursiva, onde X_i e T_i são como definidas em Teorema 4.3 (cuidado para não confundir com a linguagem $\{X_i \mid X_i$ *não* é aceita por $T_i\}$):

$$L = \{X_i \mid X_i \text{ é aceita por } T_i\}$$

a) *L é Enumerável Recursivamente.* Segue um esboço da construção de uma Máquina de Turing M que aceita uma palavra w qualquer pertencente a L:

   a.1) M gera as palavras $X_1, X_2, \ldots$ em ordem lexicográfica, comparando com w. Quando $X_i = w$, M sabe que w é a i-ésima palavra na enumeração;

   a.2) M gera T_i, a i-ésima Máquina de Turing (conforme algoritmo proposto nos exercícios);

   a.3) M simula T_i para a entrada $w = X_i$ e, se w pertence a ACEITA(T_i), então w pertence a ACEITA(M) (o algoritmo do simulador também é proposto como exercício);

   Portanto, M aceita w se, e somente se, $X_i = w$ é aceita por T_i. Logo, L é Enumerável Recursivamente;

b) *L não é Recursiva.* Conforme o Teorema 4.6, L é Recursiva se, e somente se, L e seu complemento são Enumeráveis Recursivamente. Como o complemento de L, conforme o Teorema 4.3, não é Enumerável Recursivamente, então L não é Recursiva. ❑

4.5 Gramática Irrestrita

Contrariamente às demais gramáticas introduzidas anteriormente, quais sejam, Regulares e Livres do Contexto, uma Gramática Irrestrita é simplesmente uma gramática, sem qualquer restrição nas produções.

Definição 4.8 Gramática Irrestrita.

Qualquer gramática G = (V, T, P, S) é uma *Gramática Irrestrita* ou *Tipo 0.* ❑

Portanto, o termo irrestrita é somente para enfatizar que não existe qualquer restrição sobre as produções da gramática.

EXEMPLO 4 Gramática Irrestrita.

A linguagem $L = \{ a^n b^n c^n \mid n \geq 0 \}$ é gerada pela seguinte Gramática Irrestrita:

G = ({S, C}, {a, b, c}, P, S), onde:

P = { $S \rightarrow abc \mid \varepsilon$,
$ab \rightarrow aabbC$,
$Cb \rightarrow bC$,
$Cc \rightarrow cc$ }

A palavra aaabbbccc pode ser derivada como segue (existe alguma outra seqüência de derivação que gera a mesma palavra?):

$$S \Rightarrow abc \Rightarrow aabbCc \Rightarrow aaabbCbCc \Rightarrow aaabbCbcc \Rightarrow aaabbbCcc \Rightarrow aaabbbccc$$

Deve-se reparar que a variável C "caminha" na palavra até a posição correta para gerar um terminal c. ❑

Teorema 4.9 Ling. Enumerável Recursivamente x Gramática Irrestrita.

L é uma Linguagem Enumerável Recursivamente se, e somente se, L é gerada por uma Gramática Irrestrita. ❑

O teorema não será demonstrado.

4.6 Linguagem Sensível ao Contexto

As Linguagens Sensíveis ao Contexto são definidas a partir das Gramáticas Sensíveis ao Contexto.

Definição 4.10 Gramática Sensível ao Contexto.

Uma *Gramática Sensível ao Contexto* G é uma gramática G = (V, T, P, S) com a restrição de que qualquer regra de produção de P é da forma $\alpha \rightarrow \beta$, onde:

a) α é uma palavra de $(V \cup T)^+$

b) β uma palavra de $(V \cup T)^*$

c) $|\alpha| \leq |\beta|$, excetuando-se, eventualmente, para $S \rightarrow \varepsilon$. Neste caso, S não pode estar presente no lado direito de qualquer produção. ❑

Ou seja, a cada etapa de derivação, o tamanho da palavra derivada não pode diminuir, excetuando-se para gerar a palavra vazia, se esta pertencer à linguagem. Note-se que nem toda gramática na forma livre do contexto é uma gramática na forma sensível ao contexto (por quê?).

Definição 4.11 Linguagem Sensível ao Contexto ou Tipo 1.

Uma linguagem gerada por uma Gramática Sensível ao Contexto é dita *Linguagem Sensível ao Contexto* ou *Tipo 1.* ❑

EXEMPLO 5 Linguagem Sensível ao Contexto.

A linguagem (introduzida no Capítulo 3 - Linguagens Livres do Contexto, como exemplo de linguagem que não é Livre do Contexto):

$$L = \{ ww \mid w \text{ é palavra de } \{a, b\}^* \}$$

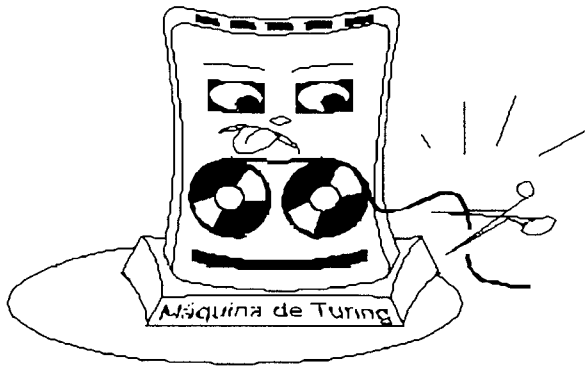
pode ser gerada por um Gramática Sensível ao Contexto como segue:

$G = (\{S, X, Y, A, B, \langle aa\rangle, \langle ab\rangle, \langle ba\rangle, \langle bb\rangle\}, \{a, b\}, P, S)$, onde:

$P = \{ S \rightarrow XY \mid aa \mid bb \mid \varepsilon,$
$X \rightarrow XaA \mid XbB \mid aa\langle aa\rangle \mid ab\langle ab\rangle \mid ba\langle ba\rangle \mid bb\langle bb\rangle,$
$Aa \rightarrow aA,\ Ab \rightarrow bA,\ AY \rightarrow Ya,$
$Ba \rightarrow aB,\ Bb \rightarrow bB,\ BY \rightarrow Yb,$
$\langle aa\rangle a \rightarrow a\langle aa\rangle,\ \langle aa\rangle b \rightarrow b\langle aa\rangle,\ \langle aa\rangle Y \rightarrow aa,$
$\langle ab\rangle a \rightarrow a\langle ab\rangle,\ \langle ab\rangle b \rightarrow b\langle ab\rangle,\ \langle ab\rangle Y \rightarrow ab,$
$\langle ba\rangle a \rightarrow a\langle ba\rangle,\ \langle ba\rangle b \rightarrow b\langle ba\rangle,\ \langle ba\rangle Y \rightarrow ba,$
$\langle bb\rangle a \rightarrow a\langle bb\rangle,\ \langle bb\rangle b \rightarrow b\langle bb\rangle,\ \langle bb\rangle Y \rightarrow bb \}$

Excetuando-se para $|w| \leq 1$, a gramática apresentada gera o primeiro w após X e o segundo w após Y, como segue:

- a cada símbolo terminal gerado após X, é gerada uma variável correspondente;
- esta variável "caminha" na palavra até passar por Y, quando deriva o correspondente terminal;
- para encerrar, X deriva uma subpalavra de dois terminais e uma correspondente variável a qual "caminha" até encontrar Y quando é derivada a mesma subpalavra de dois terminais. Note-se que, se a gramática fosse modificada para X derivar uma subpalavra de somente um terminal (e a correspondente variável) como por exemplo $X \rightarrow u\langle u\rangle$, ao encontrar Y, seria necessário derivar usando uma produção do tipo $\langle u\rangle Y \rightarrow u$ onde o lado direito possui comprimento menor que o lado esquerdo e, conseqüentemente, a gramática não seria sensível ao contexto. ❑



4.7 Máquina de Turing com Fita Limitada

Uma *Máquina de Turing com Fita Limitada*, também conhecida como *Autômato Limitado Linearmente* ou *Autômato de Fita Limitada*, é, basicamente, uma Máquina de Turing com a fita limitada ao tamanho da entrada mais duas células contendo marcadores de início e de fim de fita. Como não é conhecido se a facilidade de não-determinismo aumenta o poder computacional das Máquinas de Turing com Fita Limitada, a sua definição inclui a facilidade de não-determinismo.

Definição 4.12 Máquina de Turing com Fita Limitada.

Uma *Máquina de Turing com Fita Limitada* é uma 8-upla:

$$M = (\Sigma, Q, \delta, q_0, F, V, \circledcirc, \dagger)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- Q *conjunto de estados* possíveis da máquina o qual é finito;
- δ *programa* ou *função de transição*:

  $$\delta: Q \times (\Sigma \cup V \cup \{\circledcirc, \dagger\}) \rightarrow 2^{Q \times (\Sigma \cup V \cup \{\circledcirc, \dagger\}) \times \{E, D\}}$$

  a qual é uma função parcial;
- q_0 *estado inicial* da máquina tal que q_0 é elemento de Q;
- F *conjunto de estados finais* tal que F está contido em Q;
- V *alfabeto auxiliar* (pode ser vazio);
- $\circledcirc$ *símbolo* ou *marcador de início* o qual marca o início da fita;
- $\dagger$ *símbolo* ou *marcador de fim* o qual marca o fim da fita. ❑

EXEMPLO 6 Máquina de Turing com Fita Limitada.

Considere a linguagem:

$$L = \{ww \mid w \text{ é palavra de } \{a, b\}^*\}$$

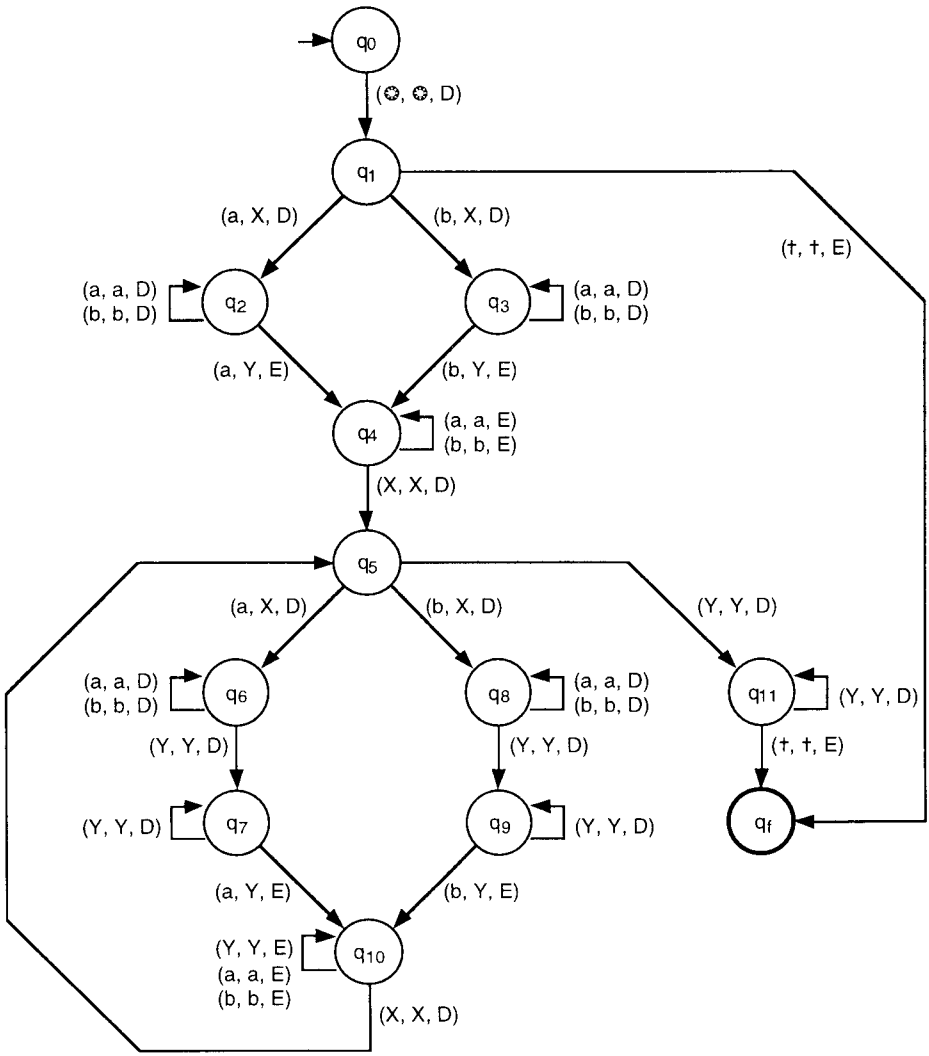


Figura 4.8 Máquina de Turing com Fita Limitada

A Máquina de Turing com Fita Limitada:

$$M = (\{a, b\}, \{q_0, q_1, ..., q_9, q_f\}, \delta, q_0, \{q_f\}, \{X, Y\}, ⊙, †)$$

ilustrada na Figura 4.8 é tal que:

ACEITA(M) = L

REJEITA(M) = Σ^* - L

e, portanto, LOOP(M) = ∅. A partir de q_1, o início do primeiro w é marcado com um X. Os estados q_2 e q_3 definem não-determinismos, com o objetivo de marcar

com um Y o início do segundo w. De q_5 a q_{11} é verificada a igualdade da primeira com a segunda metade da palavra. ❑

Teorema 4.13 Linguagem Sensível ao Contexto × Máquina de Turing com Fita Limitada.

L é uma Linguagem Sensível ao Contexto se, e somente se, L é reconhecida por uma Máquina de Turing com Fita Limitada. ❑

O teorema não será demonstrado.

4.8 Exercícios

4.1 Qual a importância do estudo da Máquina de Turing para a Ciência da Computação em geral e para as Linguagens Formais, em particular ?

4.2 Sobre a Hipótese de Church:

a) Por que não é demonstrável?

b) Qual o seu significado e importância?

4.3 Qual a diferença fundamental entre as Classes das Linguagens Recursivos e dos Enumeráveis Recursivamente? Qual a importância de se distinguir estas duas classes?

4.4 Para cada uma das linguagens abaixo, desenvolva uma Máquina de Turing que a reconheça. Sugere-se que, pelo menos três sejam do tipo com Fita Limitada.

a) $L_1 = \{a^n b^n c^n \mid n \geq 0\}$

b) $L_2 = \{w \mid w$ possui o mesmo número de símbolos a, b e c$\}$

c) $L_3 = \{a^n b^m a^{n+m} \mid n \geq 0$ e $m \geq 0\}$

d) $L_4 = \{wcw \mid w$ é palavra de $\{a, b\}^*\}$

e) $L_5 = \{ww \mid w$ é palavra de $\{a, b\}^*\}$

f) $L_6 = \{(awwa)^n \mid w$ é palavra de $\{a, b\}^*$ e $n \geq 0\}$

g) $L_7 = \{w \mid w = a^1 b^2 a^3 b^4 ... a^{n-1} b^n$ e n é par$\}$

4.5 Para cada uma das linguagens abaixo, desenvolva uma gramática que a gere (sugere-se que, pelo menos duas sejam do tipo Sensível ao Contexto):

a) $L_3 = \{a^n b^m a^{n+m} \mid n \geq 0$ e $m \geq 0\}$

b) $L_5 = \{ww \mid w$ é palavra de $\{a, b\}^*\}$

c) $L_8 = \{www \mid w$ é palavra de $\{a, b\}^*\}$

d) $L_9 = \{(a^2)^n \mid n \geq 0\}$

4.6 Na demonstração do teorema que prova que o complemento de uma Linguagem Recursiva também é Recursiva, é realizada a inversão das condições ACEITA por REJEITA e vice-versa de uma Máquina de Turing que sempre pára. Como esta inversão pode ser realizada?

4.7 Demonstre que a Classe das Linguagens Recursivas é fechada para as seguintes operações:

a) União;

b) Intersecção;

c) Diferença;

d) Concatenação sucessiva (L*, supondo L Recursiva).

4.8 Com relação à Classe das Linguagens Enumeráveis Recursivamente, o que pode-se afirmar para as seguintes operações:

a) União?

b) Intersecção?

c) Diferença?

4.9 Sobre Autômato com Pilha e Máquina de Turing:

a) Modifique a definição do Autômato com Pilha, prevendo duas pilhas (A2P);

b) Demonstre que a Classe dos A2P é equivalente, em termos de poder computacional, à Máquina de Turing;

c) Repita os itens a) e b) para AnP (Autômato com $n \geq 2$ Pilhas).

4.10 Modifique a definição do Autômato Finito para *Autômato Infinito* (ou seja, onde o conjunto de estados pode ser infinito). Demonstre que a Classe dos Autômatos Infinitos é equivalente à Classe das Máquinas de Turing.

4.11 Uma Máquina de Turing com Fita Limitada possui uma fita finita, um conjunto de estados finitos e alfabetos finitos. Portanto, pode assumir um conjunto finito de estados. Então por que o seu poder computacional não é equivalente ao de um Autômato Finito? Fundamente a sua resposta.

4.12 Estenda a função programa da Máquina de Turing usando como argumento um estado e uma palavra, de forma similar à realizada para os Autômatos Finitos.

4.13 Conforme proposto em Teorema 4.3, descreva um algoritmo para nomear (codificar) uma Máquina de Turing qualquer sobre $\Sigma = \{a, b\}$ como uma palavra em binário (do mesmo alfabeto Σ).

Sugestão: expresse a função programa como uma 5-upla (estado atual, símbolo lido, símbolo gravado, novo estado e sentido do movimento da cabeça) no alfabeto Σ. Após, codifique a Máquina de Turing como uma seqüência de 5-uplas justapostas, adicionando informações como qual o estado inicial e o conjunto de estados finais.

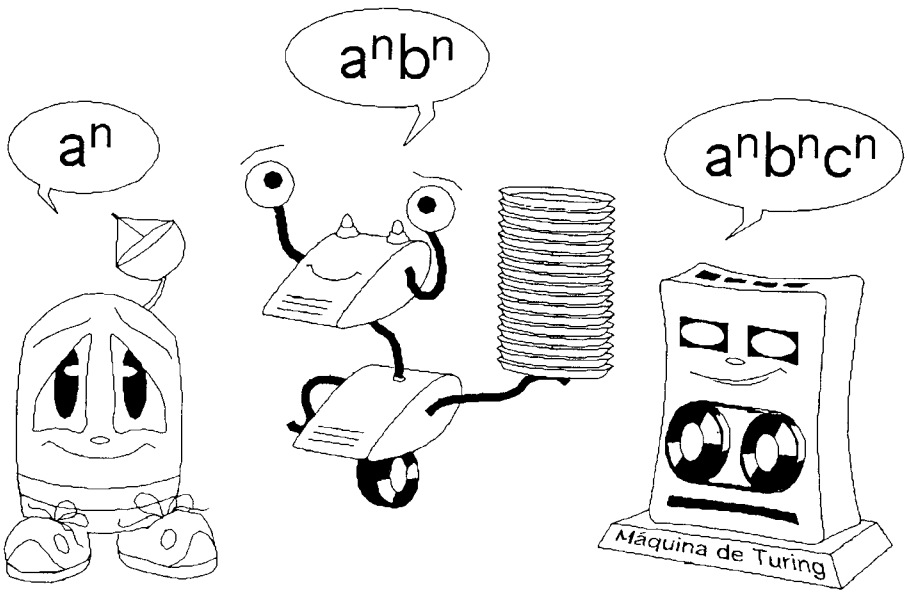
Repare que o código que representa cada Máquina de Turing é um número natural expresso em binário.

4.14 Esboce um algoritmo que transforme qualquer Máquina de Turing em uma Máquina de Turing equivalente com somente dois estados.

Sugestão: a redução do número de estados obriga o aumento do número de símbolos do alfabeto auxiliar.

Repare que este resultado mostra que a fita é suficiente como única "memória", não sendo necessário usar os estados para "memorizar" informações passadas.

4.15 Desenvolva um programa em computador que simule qualquer Máquina de Turing. A entrada para o simulador deve ser a função programa e a saída o estado final da máquina simulada, o conteúdo da fita e o número de movimentos da cabeça da fita.



5 Hierarquia de Classes de Linguagens e Conclusões

As seguintes classes básicas de linguagens estudadas:

- Linguagens Regulares ou Tipos 3;
- Livres do Contexto ou Tipo 2;
- Sensíveis ao Contexto ou Tipo 1;
- Enumeráveis Recursivamente ou Tipo 0;

e as suas inclusões próprias como ilustrado na Figura 5.1, constituem o que normalmente é conhecido como a *Hierarquia de Chomsky*. Noam Chomsky definiu estas classes como (potenciais) modelos para linguagens naturais.

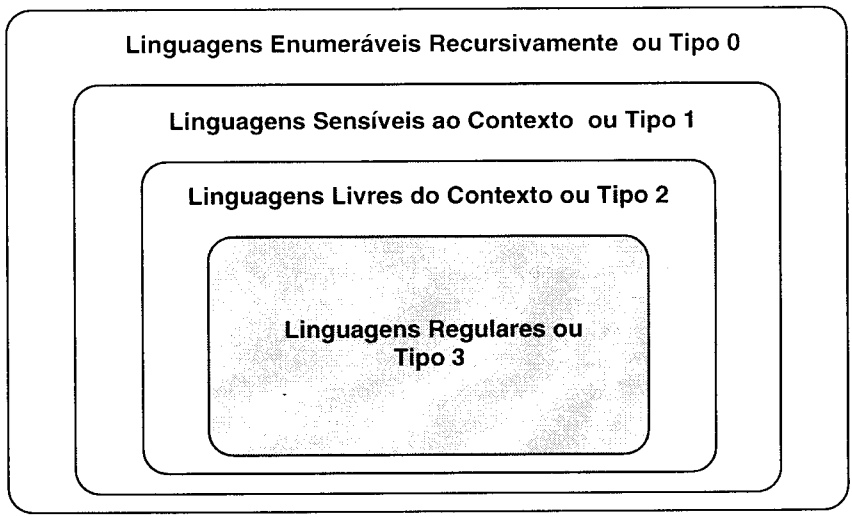


Figura 5.1 Hierarquia de Chomsky

No caso específico das linguagens naturais, o estudo das Linguagens Livres do Contexto tem sido de especial interesse, pois elas permitem uma representação simples da sintaxe, adequada tanto para a estruturação formal, como para a análise computacional. Entretanto, esse estudo tem mostrado problemas não-solucionáveis, como determinar se uma gramática é ambígua (ou seja, se existem duas ou mais árvores de derivação distintas para uma mesma palavra).

No que se refere às linguagens de programação, a Teoria das Linguagens Formais oferece meios para modelar e desenvolver ferramentas que descrevam sintaticamente a linguagem e seus processos de análise, bem como suas propriedades e limitações algorítmicas.

Entretanto, nem sempre as linguagens de programação são tratadas adequadamente na Hierarquia de Chomsky. Existem linguagens que não são Livres do Contexto, para as quais o poder dos formalismos Sensíveis ao Contexto é excessivo, sendo inadequados principalmente no que se refere à complexidade computacional (eficiência). Adicionalmente o conhecimento das Linguagens Sensíveis ao Contexto é relativamente limitado, o que dificulta o seu tratamento. Alguns exemplos de problemas que não podem ser tratados como Livres do Contexto são os seguintes:

- múltiplas ocorrências de um mesmo trecho de programa, como a declaração de um identificador e suas referências de uso (problema análogo à linguagem $\{wcw \mid w$ é palavra de $\{a, b\}^*\}$, a qual não é linguagem Livre do Contexto);

- alguns casos de validação de expressões com variáveis de tipos diferentes;
- a associação de um significado (semântica) de um trecho de programa, que dependeria da análise de um conjunto de informações como identificadores, ambientes, tipos de dados, localização, seqüências de operações, etc.

Nestes e em outros casos, a quantidade e o tipo de "memória" dos formalismos Livres do Contexto não são suficientes para manipular as informações necessárias para a adequada validação.

Por outro lado, para algumas linguagens de programação, a Classe das Linguagens Livres do Contexto é excessiva, e a das Regulares, insuficiente. Alguns exemplos de problemas para este tipo de linguagem são os seguintes:

- não existe um algoritmo que verifique a igualdade de duas linguagens Livres do Contexto, o que dificulta a otimização e o teste de processadores de linguagens;
- o Autômato com Pilha, em sua definição básica, possui a facilidade de não-determinismo. Entretanto, se a linguagem pode ser representada por um Autômato com Pilha Determinístico (ou seja, se é uma Linguagem Livre do Contexto Determinística), então é possível implementar (facilmente) um reconhecedor com tempo de processamento proporcional a $2n$ (onde n é o tamanho da entrada), o que é muito mais eficiente que o melhor algoritmo conhecido para as Linguagens Livres do Contexto.

Alguns problemas referentes à Teoria das Linguagens Formais, embora atuais, possuem muitas questões em aberto, como:

- o tratamento de linguagens n-dimensionais, com destaque para as bi e as tri-dimensionais, com aplicações no processamento de imagens no plano e no espaço;
- a tradução de linguagens, tanto naturais como de programação.

Como ilustração de uma abordagem das Linguagens Formais às linguagens n-dimensionais, no que segue é introduzida a noção de *Gramática de Grafos*.

Gramáticas de Grafos podem ser usadas para definir linguagens baseadas em grafos, bem como para descrever e analisar sistemas seqüenciais, concorrentes e distribuídos, avaliar expressões funcionais, definir mecanismos de sincronização, modelar sistemas biológicos, gerar imagens (como, por exemplo, fractais), etc.

A idéia básica das Gramáticas de Grafos é análoga à das de Gramáticas de Chomsky, ou seja:

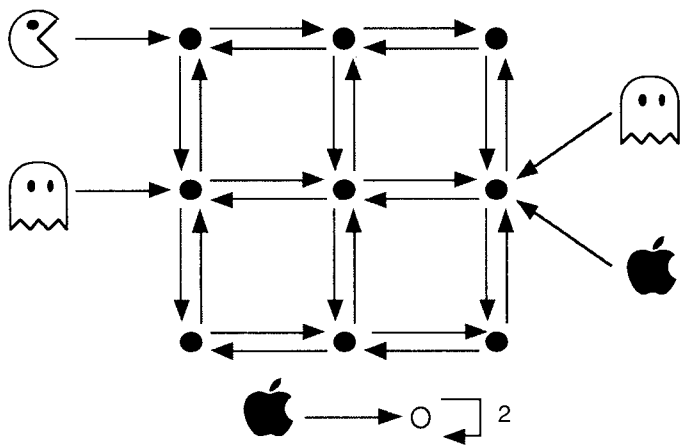


Figura 5.2 Grafo do jogo PacMan

- regras de produção são pares (mas de grafos);
- uma derivação é a substituição de um subgrafo de acordo com uma regra de produção.

As Gramáticas de Grafos são normalmente tratadas em um contexto categorial (nenhum conceito de Teoria das Categorias é formalmente introduzido nesta publicação). Informalmente, alguns conceitos podem ser visualizados no seguinte exemplo o qual ilustra o jogo de videogame PacMan.

O jogo PacMan (simplificado) possui um tabuleiro juntamente com algumas entidades especiais como um PacMan e dois conjuntos, um de fantasmas e outro de maçãs. A Figura 5.2 ilustra um grafo ("palavra") da linguagem onde:

- os nodos pretos, representam os lugares do tabuleiro e as correspondentes arestas, os caminhos possíveis entre dois lugares;
- as entidades PacMan, fantasmas e maçãs são representadas por nodos com simbologia própria, onde os correspondentes arcos denotam o seu posicionamento no tabuleiro;
- o nodo branco descreve a fase em que se encontra o jogo (na Figura 5.2, a fase é 2). Adicionalmente, as maçãs cujos arcos são associados ao nodo branco, representam as maçãs já comidas pelo PacMan (na Figura 5.2, uma maçã foi comida).

A Figura 5.3 ilustra as regras de produção move, come e mata. Por fim, a Figura 5.4 ilustra o grafo resultante da aplicação da regra move ao grafo ilustrado na Figura 5.2. Note-se que, no grafo resultante, o PacMan encontra-se em um lugar onde existe um fantasma. Assim, a próxima produção pode ser mata ou move.

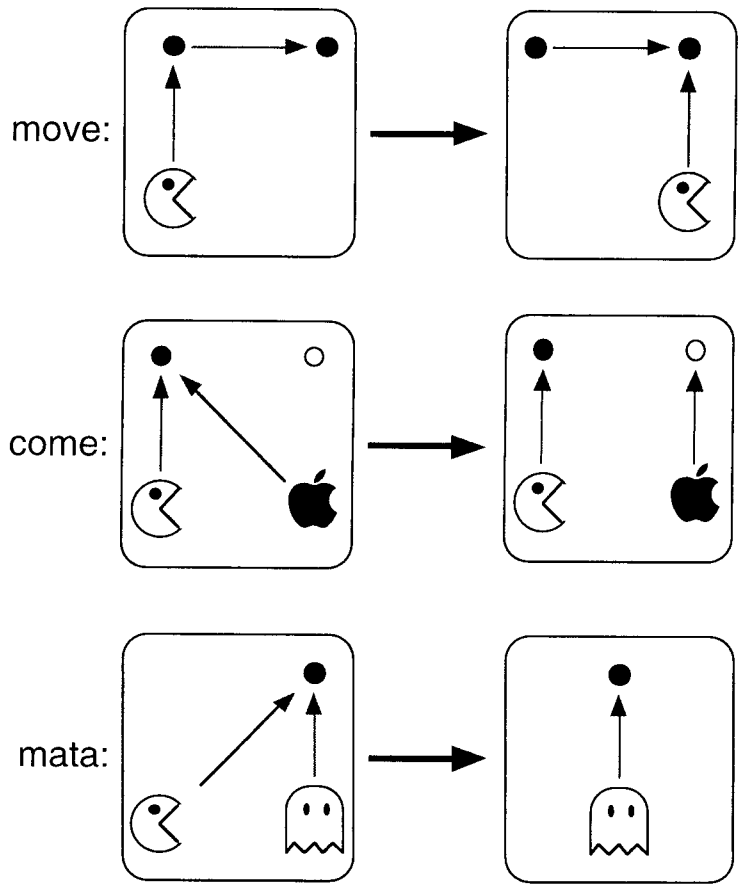


Figura 5.3 Regras de produção do PacMan

Assim, dependendo da linguagem e dos objetivos do trabalho, alguns estudos específicos, eventualmente fora da Hierarquia de Chomsky, são recomendados ou necessários. Estes transcendem o objetivo desta publicação e algumas referências sobre o assunto podem ser encontradas na bibliografia recomendada.

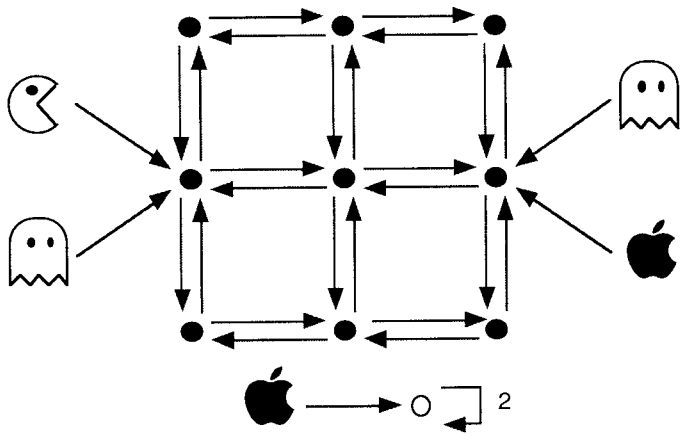


Figura 5.4 Grafo resultante da aplicação da regra move

6 Referências

Aho, A. V., *Currents in the Theory of Computing*, Prentice-Hall, 1973.

Aho, A. V. e Ullman, J. D., *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, 1972.

Barr, M. e Wells, C., *Category Theory for Computing Science*, Prentice Hall, 1990.

Beckman, F. S., *Mathematical Foundations of Programming*, Addison-Wesley, 1981.

Brookhear, J. G., *Theory of Computation, Formal Languages, Automata and Complexity*, Benjamin/Cummings, California, E.U.A., 1989.

Cleaveland, J. C. e Uzgali, R. C. S., *Grammars For Programming Languages*, Elsevier, 1977.

Erni, W. J., *Auxiliary Pushdown Acceptors and Regulated Rewriting Systems*, Universitat Karlsruhe, 1977.

Degano, P., Gorrieri, R. e Marchetti-Spaccamela A., *International Colloquium on Automata, Language and Programming*, LNCS 1256, Springer-Verlag, 1995.

Ferreira, A. B., *Novo Dicionário da Língua Portuguesa*, Nova Fronteira.

Fulop, Z. e Gesceg, F, *International Colloquium on Automata, Language and Programming*, LNCS 944, Springer-Verlag, 1997.

Gough, K. J., *Syntax Analysis and Software Tools*, Addison-Wesley, 1988.

Harrison, M. A., *Introduction to Formal Language Theory*, Addison-Wesley, 1978.

Hopcroft, J. E. e Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

Hopcroft, J. E. e Ullman, J. D., *Formal Languages and their Relation to Automata*, Addison-Wesley, 1969.

Löwe, M., *Algebraic Approach to Single-Pushout Graph Transformation.* TCS 109, pp. 181-224, 1993.

Manna, Z., *Mathematical Theory of Computation*, McGraw-Hill, 1974.

Maurer, H. A., *On Non Context-Free Grammar Forms*, Universitat Karlsruhe, 1977.

Menezes, P. B., *Teoria das Linguagens Formais*, UFRGS, 1991.

Menezes, P. B., Sernadas, A. e Costa, J. F., *Nonsequential Automata Semantics for a Concurrent Object-Based Language*, a ser publicado na ENTCS - Eletronic Notes in Theoretical Computer Science, Elsevier.

Menezes, P. B., *Reificação de Objetos Concorrentes*, Instituto Superior Técnico/Universidade Técnica de Lisboa, Lisboa, 1997.

Moll, R. N., Arbib, M. A. e Kfoury A. J., *An Introduction to Formal Language Theory*, Springer-Verlag, 1988.

Nijholt, A., *Context-Free Grammars: Covers, Normal Forms and Parsing*, LNCS 93, Springer-Verlag, 1980.

Paun, G. e Salomaa, A, *New Trends in Formal Languages*, LNCS 1218, Springer-Verlag, 1997.

Ribeiro, L., *Parallel Composition and Unfolding Semantics of Graph Grammars*, Technical University of Berlin, 1996.

Rozenberg, G. e Salomaa, A. (Eds.), *Handbook of Formal Languages.*, Vol.1 - *Word Language Grammar*, Vol.2 - *Linear Modeling: Background and application* e Vol. 3 - *Beyond Words*, Springer-Verlag, 1997.

Salomaa, A., *Formal Languages*, Academic Press, 1973.

Sernadas, C., *Introdução à Teoria da Computação*, Editora Presença, Portugal, 1992.

Sippu, S. Soisalon-Soininen, E., *Parsing Theory*, Vols. I e II, Springer-Verlag, 1988.

Stoy, F. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

Verson, J. A. e Noonan, R., *Heuristic Approach to Context Sensitive Parsing*, University of Maryland, College Park, MR, 1974.

Índice Remissivo

D

E

F

M

N

O

P

R

S