UNIVERSIDADE FEDERAL DE JUIZ DE FORA – UFJF
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Linguagens Formais e Autômatos

Prof. Itamar Leite de Oliveira
Curso de Ciência da Computação

Juiz de Fora – MG

1. Teoria de Linguagens Formais

1.1 Conceitos Iniciais

Nesta seção serão apresentados alguns conceitos gerais sobre gramáticas e linguagens que servirão para fundamentar o estudo de todos os tipos de linguagens que virão a seguir.

1.1.1 Introdução

A teoria de Linguagens Formais e a teoria de Máquinas (e autômatos) são tópicos abrangentes que inserem no estudo da Teoria da Computação em geral. A Teoria da Computação é uma ciência que procura organizar o conhecimento formal relativo aos processos de computação, como complexidade de algoritmos, linguagens formais, problemas intratáveis, etc.

Atualmente, o termo "computar" está associado ao conceito de fazer cálculos ou aritmética usando para isso, máquinas computadoras. Entretanto, a noção de computabilidade pode ser dissociada da implementação física da máquina. Originalmente, a palavra latina *putare* significa "pensar", mas no caso da teoria da computação o sentido mais adequado seria algo como "manipulação de símbolos", o que não é, necessariamente, pensamento. Essa manipulação de símbolos, envolvendo, computadores, segundo Alan Turing (1969) realiza qualquer operação formal de que o ser humano seria capaz.

Mesmo assim, alguns filósofos sustentam que os computadores não podem computar (fazer aritmética), porque não compreendem a noção de número (apenas manipulam sinais elétricos). Todavia, pode-se falar, sem entrar no mérito da questão, que os computadores realizam uma "computação perceptível" já que transformam uma entrada em saída como se estivessem realmente computando. Neste caso, o que importa é o efeito final do processo e não a maneira como ele é feito.

1.1.2 Símbolo

Um *símbolo* é uma entidade abstrata que não precisa ser definida formalmente, assim como "ponto" e "linha" não são definidos na geometria. Letras e dígitos são exemplos de símbolos freqüentemente usados.

Símbolos são ordenáveis lixicograficamente, e, portanto podem ser comparados quanto à igualdade ou precedência. A principal utilidade dos símbolos está na possibilidade de usá-los como elementos atômicos em definições de linguagens.

Exemplo: a, b, U, P, >, /,A, +, *, etc.

1.1.3 Cadeia

Uma *cadeia* (ou *string*, ou *palavra*) é uma seqüência finita de símbolos justapostos (isto é, sem vírgulas separando os caracteres.) Por exemplo, se *a*, *b*, e *c* são símbolos, então *abcb* é um exemplo utilizando estes símbolos.

1.1.4 Alfabeto

Um *alfabeto* (ou vocabulário) é definido simplesmente como um conjunto finito,não vazio, de símbolos (elementos). Representaremos um alfabeto por.

Exemplo:

V = {a, b, c,...,z }
V = {0, 1}
V = {a, e, i, o ,u}

Tamanho: O tamanho de uma cadeia é o comprimento da seqüência de símbolos que a forma. O tamanho de uma cadeia w será denotado por |w|.

Exemplo: Seja V = { a , b , c }
se x = aba, então |x| = 3
se x = c, então |x| = 1

Cadeia vazia: A cadeia vazia é denotada por ε (épslon), e tem tamanho igual a zero, isto é, não possui nenhum símbolo; assim |ε| = 0.

Prefixo e sufixo: Um prefixo de uma cadeia é um número qualquer de símbolos tomados de seu início e *sufixo* é um número qualquer de símbolos tomados de seu fim.

Exemplo: Seja a cadeia *abc*
Prefixos: ε, *a*, *ab*, *abc*
Sufixos: ε, *c*, *bc*, *abc*.

Um prefixo ou sufixo que não seja a própria cadeia é chamado de *prefixo próprio* ou *sufixo próprio* respectivamente.

Concatenação: A concatenação de duas cadeias é a cadeia formada pela escrita da primeira cadeia seguida da segunda, sem nenhum espaço no meio.

Exemplo: a concatenação de *horta* e *liça* é *hortaliça*.

O operador de concatenação é a justaposição. Isto é, se *w* e *x* são variáveis que denotam cadeias, então *wx* é a cadeia formada pela concatenação de *w* e *x*.

Potência: A *n-ésima potência* de uma cadeia *x*, denotada por x^n é a concatenação de *x* com ela mesma *n* vezes.

Exemplo: $(abc)^3 = abcabcabc$. $a^5 = aaaaa$ $a^2b^41^6 = aabbbb111111$

Fechamento: o fechamento de um alfabeto *V*, representado por V^*, é o conjunto todas as cadeias que podem ser formadas com os símbolos de *V*, inclusive a cadeia vazia.

Exemplo: V = {a, b}
V^* = {ε, a, b, ab, ba, aaa, aab, aba, abb, ...}

Fechamento positivo: O fechamento positivo de *V*, representado por V^+ é definido como V^*-{ε}
Exemplo: V = {a, b}
V^+ = {a, b, ab, ba, aaa, aab, aba, abb, ...}

1.1.5 Linguagem

Uma *linguagem* (formal) é um conjunto de cadeias de símbolos tomados de algum alfabeto. Isto é, uma linguagem sobre um alfabeto *V* é um subconjunto de V^*. Note-se que esta definição é meramente extensional, isto é, não considera os mecanismos formadores da linguagem, mas apenas a sua extensão. Assim, por exemplo, o conjunto de sentenças válidas da língua portuguesa poderia ser definido extensionalmente como um subconjunto de $\{a, b, c, ..., z, A, B, C, ..., Z\}^*$.

Exemplos: Seja V = {0, 1}
A linguagem $\{0^n1^n \mid n > 0\}$ é um subconjunto de V^*.
A linguagem $\{0^n1^m \mid n > 0, m > 0\}$ é um subconjunto de V^*.
A linguagem {w | possui um número par de 1's} é um subconjunto de V^*.

Uma linguagem é *finita* se suas sentenças formam um conjunto finito. Caso contrário, a linguagem é *infinita*. Uma linguagem infinita precisa ser definida através de uma representação finita.

Um *reconhecedor* para uma linguagem é um dispositivo formal usado para verificar se uma determinada sentença pertence ou não a uma determinada linguagem. São exemplos de reconhecedores de linguagens a *Máquina de Turing*, o *Autômato finito* e o *Autômato de Pilha*. Cada um destes mecanismos reconhece um conjunto bastante particular de linguagens. Entretanto, os modelos são inclusivos: todas as linguagens reconhecíveis por um autômato finito podem ser reconhecidas por um autômato de pilha e todas as linguagens reconhecíveis por um autômato de pilha são reconhecíveis por máquinas de Turing. As recíprocas, porém não são verdadeiras.

Um *sistema gerador* é um dispositivo formal através do qual as sentenças de uma linguagem podem ser sistematicamente geradas. Exemplos de sistemas geradores são as gramáticas gerativas, definidas pela primeira vez por Noam Chomsky, em seus estudos para sistematizar a gramática da Língua Inglesa.

1.2 Gramáticas

Uma *Gramática gerativa* é um instrumento formal capaz de construir (gerar) conjuntos de cadeias de uma determinada linguagem. As gramáticas são instrumentos que facilitam muito a definição das características das linguagens.

Usar um método de definição de conjuntos particular para definir uma linguagem pode ser um passo importante no projeto de um reconhecedor para a linguagem, principalmente se existirem métodos sistemáticos para converter a descrição do conjunto em programa que processa o conjunto.

Exemplo intuitivo de uma Gramática: (um subconjunto da gramática da língua portuguesa)

<sentença> :: = **<sujeito>** **<predicado>**
<sujeito> :: = **<substantivo>**
| <artigo> **<substantivo>**
| <artigo> **<adjetivo>** **<substantivo>**
<predicado> :: = **<verbo>** **<objeto>**
<substantivo> :: = **João | Maria | cachorro | livro | pão**
<artigo> :: = **o | a**
<adjetivo> :: = **pequeno | bom | bela**
<verbo> :: = **morde | lê | olha**
<objeto> :: = **<substantivo>**
| <artigo> **<substantivo>**
| <artigo> **<adjetivo>** **<substantivo>**

1.2.1 Definição Formal

Uma *gramática* possui um conjunto finito de *variáveis* (também chamadas *não-terminais*). A linguagem gerada por uma gramática é definida recursivamente em termos das variáveis e dos símbolos primitivos chamados *terminais*.

As regras que relacionam terminais e variáveis são chamadas de *produções*. Uma regra de produção típica estabelece como uma determinada configuração de variáveis e terminais pode ser reescrita, gerando uma nova configuração.

Uma gramática é especificada por uma quádrupla (N, T, P, S), onde:

a) N é o conjunto de não-terminais ou variáveis.
b) T é o conjunto de terminais, ou alfabeto, sendo que N e T são conjuntos disjuntos.
c) P é um conjunto finito de produções, ou regras, sendo que $P \subseteq (N \cup T)^{+} \times (N \cup T)^{*}$
d) S é um símbolo não-terminal inicial, sendo que $S \in N$

Uma produção (n, p) pode ser escrita como $n \rightarrow p$, para facilitar a leitura. No caso de existir mais de uma alternativa para uma mesma configuração e símbolos do lado esquerdo da regra, como, por exemplo:

<*substantivo*> → coisa
<*substantivo*> → cão
<*substantivo*> → pedra

O símbolo → pode ser ido como: "é definido por", e o símbolo | pode ser lido como "ou".

Exemplo: Formalizando o subconjunto da gramática da língua portuguesa, apresentado na seção anterior, teríamos:

Gportugues = (*N*, *T*, *P*, *S*), onde:
N = { <sentença> , <sujeito> , <predicado> , <substantivo> ,
<artigo> , <adjetivo> , <predicado> , <verbo> , <objeto> }
T = {joão, maria, cachorro, livro, pão, o, a, pequeno, bom, bela, morde, lê, olha}
P = é o conjunto das regras gramaticais apresentado
S = <sentença>

1.2.2 Derivação

Uma gramática pode ser usada para gerar uma linguagem através de *reescrita* ou *derivação*. A derivação pode ser definida da seguinte maneira: dada uma sentença[1] a_1bb_1, com a_1 e a_2 sendo cadeias de terminais e não-terminais, se existir uma regra da forma $b \rightarrow c$, então a sentença inicial pode ser reescrita como a_1cb_1. A derivação, ou reescrita é denotada da seguinte forma:

$$a_1bb_1 \Rightarrow a_1cb_1$$

Se $a \Rightarrow b$, então se diz que *a* produz diretamente *b*. O fecho transitivo e reflexivo da relação $\Rightarrow$ é $\Rightarrow^*$. Portanto, se $a \Rightarrow b \Rightarrow ... \Rightarrow z$, então se diz que *a* produz *z*, simplesmente, anotando a produção por $\Rightarrow^*$. O fecho transitivo pode ser definido da seguinte forma:

a) $a \Rightarrow^0 a$
b) $a \Rightarrow^1 b$ se $a \Rightarrow b$
c) $a \Rightarrow^{n+1} z$ se $a \Rightarrow^n y$ e $y \Rightarrow^1 z$ para todo $n \geq 1$.
d) $a \Rightarrow^* z$ se $a \Rightarrow^n z$ para algum $n \geq 1$.

Seja, por exemplo, a seguinte gramática, onde as regras são numeradas para acompanhar o desenvolvimento da derivação:

1: <S> → <X>+<X>=
2,3: <X> → 1<X> | 1
4: <X>+1 → <X>1+
5: <X>1 → 1<X>
6: <X>+= → <X>
7: 1<X> → <X>1

A derivação de uma sentença a partir desta gramática consiste em uma seqüência de aplicações das produções a partir do símbolo inicial. Por exemplo:

<S>	⇒ <X>+<X>=	[1]
	⇒ 1<X>+<X>=	[2 no primeiro <X>]
	⇒ 1<X>+1=	[3 no segundo <X>]
	⇒ 1<X>1+=	[4]
	⇒ 11<X>+=	[5]

[1] Uma sentença neste texto é uma cadeia de terminais e não-terminais. Assim, se w é uma sentença, então $w \in (T \cup U)^*$.

$\Rightarrow$ 11<X> [6]
$\Rightarrow$ 111 [3]

O fato das regras terem sido aplicadas praticamente em seqüência foi mera coincidência. A princípio, qualquer regra de derivação pode ser aplicada a qualquer ponto da sentença desde que a configuração de símbolos do lado esquerdo da regra apareça na sentença sendo derivada.

Originalmente, a motivação do uso de gramáticas foi o de descrever estruturas sintáticas da língua natural. Poderia-se escrever regras (produções) como:

<*frase*> $\rightarrow$ <*artigo*><*substantivo*><*verbo*><*artigo*><*substantivo*>.|
<*artigo*><*substantivo*><*verbo*>.
<*verbo*> $\rightarrow$ dorme | escuta | ama
<*substantivo*> $\rightarrow$ gato | rádio | rato
<*artigo*> $\rightarrow$ o | a | os | as

Desta gramática pode-se obter derivações como:

<*frase*> $\Rightarrow$ <*artigo*><*substantivo*><*verbo*><*artigo*><*substantivo*>.
$\Rightarrow$ o<*substantivo*><*verbo*><*artigo*><*substantivo*>.
$\Rightarrow$ o gato <*verbo*><*artigo*><*substantivo*>.
$\Rightarrow$ o gato ama <*artigo*><*substantivo*>.
$\Rightarrow$ o gato ama o<*substantivo*>.
$\Rightarrow$ o gato ama o rato.

Entretanto, a mesma gramática também permite derivar “o gato dorme o rato”, frase que não faz sentido. Para evitar a geração de tais frases, pode-se usar uma gramática sensível ao contexto, estabelecendo que <*verbo*> só pode ser reescrito para “dorme” se for sucedido pelo ponto final da frase:

<*frase*> $\rightarrow$ <*artigo*><*substantivo*><*verbo*><*artigo*><*substantivo*>.|
<*artigo*><*substantivo*><*verbo*>.
<*verbo*>. $\rightarrow$ dorme
<*verbo*><*artigo*> $\rightarrow$ escuta | ama
<*substantivo*> $\rightarrow$ gato | rádio | rato
<*artigo*> $\rightarrow$ o | a | os | as

Para esta gramática é impossível derivar as cadeias “o gato dorme o rato”, “o gato ama” e “o gato escuta”. O verbo *dorme* só pode ser derivado se for sucedido por um ponto final e os verbos *escuta* e *ama* só podem ser derivados se forem sucedidos por um artigo.

1.2.3 Notação

Como convenção notacional, pode-se admitir que símbolos não terminais serão sempre representados por letras maiúsculas, e terminais por minúsculas. Assim, uma regra como <*a*> $\rightarrow$ <*a*>*b* | *b* poderia ser escrita como $A \rightarrow Ab \mid b$. Além disso, os conjuntos *T* e *N* podem ficar subentendidos, e não precisam ser expressos sempre.

Pode-se ainda arbitrar que o símbolo inicial *S* será o não-terminal que aparecer primeiro do lado esquerdo da primeira produção da gramática. Assim, bastaria listar as regras de produção para se ter a definição completa da gramática.

1.3 Técnicas para Definição de Gramáticas

Nesta seção serão introduzidas algumas técnicas padrão para definição de produções para fins específicos. São elas: listas simples, lista com separadores, opção, concatenação, aninhamento e relações quantitativas.

Lista simples: xxxxx...x

$A \rightarrow xA \mid x$ ou

$A \rightarrow Ax \mid x$

Exemplo: números naturais

$N \rightarrow DN \mid D$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid ... \mid 9$

Lista com separadores: xSxSxSxS...Sx

$A \rightarrow xSA \mid x$

Exemplo:cadeias de a's separadas por $.

$S \rightarrow A\$S$

$A \rightarrow aA \mid a$

Opção: G1 ou G2

$A \rightarrow B \mid C$

Exemplo1: comentário em pascal

$C \rightarrow H \mid P$

$H \rightarrow \{M\}$

$P \rightarrow (*M*)$

$M \rightarrow LM \mid M$

$M \rightarrow a \mid b \mid c \mid ... \mid z \mid A \mid B \mid ... \mid Z \mid \varepsilon$

Exemplo2: conj. números inteiros.

$Z \rightarrow P \mid N$

$P \rightarrow U \mid +U$

$N \rightarrow -U$

$U \rightarrow DU \mid D$

$D \rightarrow 0 \mid 1 \mid ... \mid 9$

Concatenação: G1G2

$C \rightarrow AB$

Exemplo: número em ponto flutuante

$R \rightarrow NPN \mid N \mid PN \mid NP$

$P \rightarrow .$

$N \rightarrow DN \mid D$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid ... \mid 9$

Aninhamento:

$A \rightarrow aAf \mid c$

Exemplo: expressões aritméticas com + e *

$E \rightarrow N \mid E+E \mid E*E \mid (E)$

$N \rightarrow DN \mid D$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid ... \mid 9$

Relações quantitativas: Exemplos:

1) $\{a^i b^{2i} \mid i \geq 0\} = \{\varepsilon, abb, aabbbb, ...\}$

$A \rightarrow aAbb \mid \varepsilon$

2) $\{a^i b^j \mid j \geq 0 \text{ e } i > j\} = \{\varepsilon, aab, aaaa, aaaabb ...\}$

$L \rightarrow aLb \mid S$

$S \rightarrow a \mid aS$

3) $\{a^i b^j \mid j = 2i \text{ ou } i > j\} = \{\varepsilon, aab, aaaa, aaaabb, abb ...\}$

$P \rightarrow A \mid L$

$A \rightarrow aAbb \mid \varepsilon$

$L \rightarrow aLb \mid S$

$S \rightarrow a \mid aS$

4) $\{a^i b^j c^k d^m \mid i = m \text{ e } j = 3k\}$ = {abbbcd, aabbbbbbccdd,...}

$L \rightarrow aLd \mid S$
$S \rightarrow bbbSc \mid \varepsilon$

5) $\{a^i b^j \mid i \neq j \text{ e } i,j > 0\}$ = {abbb, aab, aaabbbb, aaaab...}

$S \rightarrow aSb \mid L$
$L \rightarrow A \mid B$
$A \rightarrow a \mid aA$
$B \rightarrow b \mid bB$

6) $\{a^i b^j c^{i+j} \mid i,j > 0\}$ = {abcc, aabccc, aabbbccccc, ...}

$S \rightarrow aSc \mid B$
$B \rightarrow bBc \mid bc$

1.4 Exercícios

1) Uma palíndrome é uma cadeia que pode ser lida da mesma maneira da direita para a esquerda e da esquerda para a direita. Exemplos: *a*, *aba*, *osso*, *asddsa*. Defina uma gramática para a linguagem palíndrome (conjunto de todas as palíndromes sobre o alfabeto A = {*a, b, c, ..., z*}).

2) Seja G = ({S}, {*a, b*}, P, S). Dê definições para o conjunto de produções P, de maneira a gerar as seguintes linguagens:

a) Cadeias que começam e terminam com *a*.
b) Cadeias com tamanho ímpar.
c) Cadeias onde todos os *a*'s aparecem consecutivos.
d) Cadeias onde todos os *b*'s precedem todos os *a*'s.
e) Cadeias onde o número de *b*'s é par.

Observe que a gramática deve ter apenas um não terminal.

3) Defina uma gramática que gere as expressões numéricas com o seguinte alfabeto dígitos: 0 a 9; sinais: +, - , * e /; parênteses: (e). Teste sua definição mostrando a derivação das seguintes cadeias:

a) 1+(3*534)
b) ((8/2-(25+3))

1.5 Linguagens definidas por gramáticas

A linguagem definida por uma gramática consiste no conjunto de cadeias de terminais que esta gramática pode potencialmente gerar. Este conjunto será denotado por um L(G). A linguagem definida por uma gramática G = (N, T, P, S) é dada por:

$$L(G) = \{\omega \mid \omega \in T^* \ \ e \ \ S \rightarrow^* \omega\}$$

Em outras palavras, uma cadeia pertence a L(G) se e semente se ela consiste somente de terminais e pode ser produzida a partir de S.

Uma cadeia em $(T \cup N)^*$ é chamada de *forma sentencia l*(ou *sentença*) se ela pode ser produzida a partir de S. Assim, pode caracterizar L(G) como o conjunto de todas as formas sentenciais que contém apenas terminais.

1.6 Tipos de Gramáticas

Impondo restrições na forma das produções, pode-se identificar quatro tipos diferentes de gramáticas. Esta classificação foi feita por Chomsky e é conhecida como *hierarquia de Chomsky*:

a) Tipo 0. Não há restrição na forma das produções. Este é o tipo mais geral.

b) Tipo 1. Seja G = (N, T, P, S). Se (i) O símbolo inicial S não aparece no lado direito de nenhuma produção, e (ii) para cada produção $\sigma_1 \rightarrow \sigma_2$, é verdadeiro que $|\sigma_1| \leq |\sigma_2|$ (com exceção para a regra $S \rightarrow \varepsilon$), então se diz que G é uma gramática do tipo 1, u *Gramática Sensível ao Contexto* (GSC).

c) Tipo 2. Uma gramática G = (N, T, P, S) é do tipo 2 ou *Livre de Contexto* (GLC) se cada produção é livre de contexto, ou seja, cada produção é da forma $A \rightarrow \sigma$, com $A \in N$ e $\sigma \in (T \cup N)^*$.

d) Tipo 3. Uma gramática G é de tipo 3, ou *regular*, se cada produção é da forma $A \rightarrow aB$, $A \rightarrow a$ ou $A \rightarrow \varepsilon$, onde A e B são não-terminais ($A \in N$ e $B \in N$) e *a* é um terminal ($a \in T$).

Isto é claramente, uma restrição de tipo 2. A importância deste tipo simples de gramática será estudada mais adiante.

1.7 Tipos de Linguagens

Uma linguagem L é de tipo *i* se existe uma gramática G de tipo i tal que L = L(G), para i igual a 0, 1, 2 ou 3. Pode-se ver que cada linguagem de tipo 3 é também de tipo 2, cada linguagem de tipo 2 é também de tipo 1, e cada linguagem de tipo 1 é também de tipo 0. Estas inclusões são estritas, isto é, existem linguagens de tipo 0 que não são de tipo 1, existem linguagens de tipo 1 que não são de tipo 2 e existem linguagens de tipo 2 que não são de tipo 3.

Pode-se relacionar cada tipo de gramática com uma máquina reconhecedora da seguinte maneira: a gramática de tipo 0 gera linguagens reconhecíveis por máquinas de Turing; a gramática de tipo 2 gera linguagens reconhecidas por autômatos de pilha; a gramática de tipo 3 gera linguagens reconhecidas por autômatos finitos.

Não há uma classe de reconhecedores para apenas linguagens de tipo 1, porque qualquer máquina capaz de reconhecer linguagens de tipo 1 é poderosa o suficiente para reconhecer linguagens de tipo 0. A principal razão para identificar as linguagens de tipo 1 como um tipo separado é porque toda a linguagem de tipo 1 é decidível, isto é, se G = (N, T, P, S) então existe um algoritmo tal que, para qualquer cadeia *a*, responda "sim" se *a* pertence à L(G) e "não" caso contrário.

1.8 Exercícios

a) Seja a gramática G = (N, T, P, S), com N = {<sentença>, A, B}, T = {Este, é, o, gato, que, matou, rato, comeu, malte, estava, no, porão, da, casa, João, constuiu}, S = <sentença> e P dado por:

<senteça> → AB
A → Este é o | A gato que matou
matou B → matou o rato que comeu B
comeu B → comeu o malte que estava no B
B → porão da casa que João construiu

Derive duas cadeias diferentes a partir desta gramática, indicando as derivações.

b) Seja G a seguinte GLC (gramática livre de contexto):

$S \rightarrow aSbSa \mid \varepsilon$

Mostre quatro cadeias de L(G) e a derivação de cada uma delas.

c) Escreva uma gramática para cada uma das seguintes linguagens:

a) $\{a^{i+3}b^{2i+1} \mid i \geq 0\} \cup \{a^{2i+1}b^{3i} \mid i \geq 0\}$
b) $\{a^i b^j c^j d^i e^3 \mid i, j \geq 0\}$
c) $\{a^i b^j \mid i, j \geq 0 \ \ com \ \ i = 2j \ \ ou \ \ 2i = j\}$
d) $\{a^i b^j c^k d^l \mid i, j, k, l \geq 0, i \leq l \ \ e \ \ j \neq k\}$
e) $\{a^i b^j b^i a^j \mid i, j \geq 0\}$

Identifique o tipo de gramática que foi definida para cada um dos casos.

d) Escreva uma gramática livre de contexto para gerar a linguagem: conjunto de todas as cadeias sobre o alfabeto $\{a, b, c, (,)\}$ com os parênteses corretamente aninhados.

e) Caracterize e explique as relações entre linguagem, gerador e reconhecedor.

f) Defina uma gramática livre de contexto para os cabeçalhos de declaração de procedimentos e funções da linguagem Pascal.

g) Indique, entre os parênteses, o tipo de cada gramática:

( ) $S \rightarrow aS \mid b$
( ) $S \rightarrow \varepsilon \mid aB$ $\quad B \rightarrow Ab \mid a$
( ) $S \rightarrow AAA \mid a$ $\quad A \rightarrow AbcA \mid \varepsilon$
( ) $S \rightarrow (S) \mid S+S$ $\quad (S+S) \rightarrow [id+id]$
( ) $S \rightarrow aA \mid Aba$ $\quad aA \rightarrow a \mid baA$

2. Linguagens Regulares e Autômatos Finitos

As linguagens regulares constituem um conjunto de linguagens decidíveis bastante simples e com propriedades bem definidas e compreendidas. Essas linguagens podem ser reconhecidas por autômatos finitos e são facilmente descritas por expressões simples, chamadas *expressões regulares* (ER).

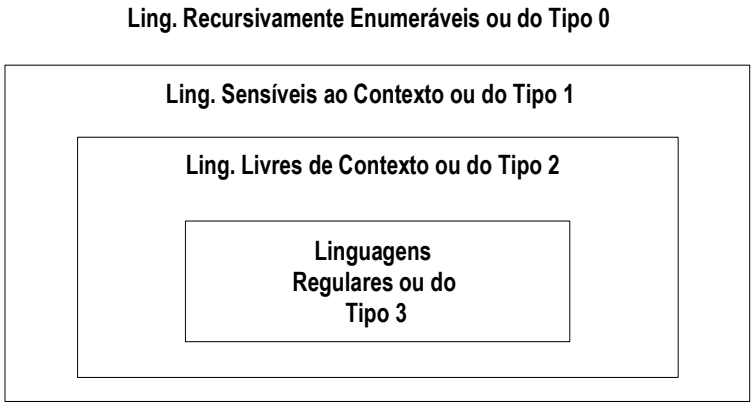


Figura 2-1: Hierarquia de Chomsky.

O estudo das linguagens regulares (ou linguagens do tipo 3 na Hierarquia de Chomsky, veja Figura 2-1, as mais simples, permitindo abordagens de pequena complexidade, grande eficiência e fácil implementação) pode ser abordado através de 3 diferentes formalismos:

- *operacional ou reconhecedor*: Autômato Finito, que pode ser determinístico, não determinístico ou com movimento vazio (com ε-transição).
- *axiomático ou gerador*: Gramática Regular
- *denotacional*: Expressão Regular (também pode ser considerado gerador).

2.1 Autômatos Finitos

Um autômato finito, pode ser vista como uma máquina composta basicamente por três partes (Figura 2-2):

- *Fita*: Dispositivo de entrada que contém a informação a ser processada. A fita é finita à esquerda e à direita. É dividida em células onde cada uma armazena um símbolo. Os símbolos pertencem a um alfabeto de entrada. Não é possível gravar sobre a fita. Não existe memória auxiliar. Inicialmente a palavra a ser processada, isto é, a informação de entrada ocupa toda a fita.
- *Unidade de Controle*: Reflete o estado corrente da máquina. Possui uma unidade de leitura (cabeça de leitura), que acessa uma unidade da fita de cada vez. Pode assumir um número finito e pré-definido de estados. Após cada leitura a cabeça move-se uma célula para a direita.
- *Programa* ou *Função de Transição*: Função que comanda as leituras e define o estado da máquina. Dependendo do estado corrente e do símbolo lido determina o novo estado do autômato. Usa-se o conceito de estado para armazenar as informações necessárias à determinação do próximo estado, uma vez que não há memória auxiliar.

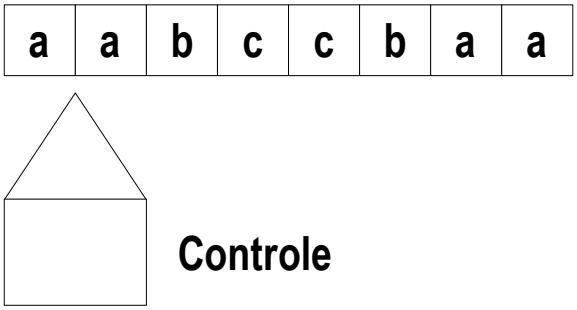


Figura 2-2: Autômato Finito como uma máquina com controle finito.

2.1.1 Autômato Finito Determinístico (AFD)

Um *autômato finito determinístico* (AFD), ou simplesmente *autômato finito*, M é uma quíntupla:

$$M = (\Sigma, Q, \delta, q0, F),$$

onde:

- Σ - Alfabeto de símbolos de entrada
- Q - Conjunto finito de estados possíveis do autômato
- δ - Função programa ou função de transição $\delta: Q \times \Sigma \rightarrow Q$
- q_0 - Estado inicial tal que $q_0 \in Q$
- F - Conjunto de estados finais, tais que $F \subseteq Q$.

A função programa pode ser representada como um grafo orientado finito conforme representado pela Figura 2-3 e Figura 2-4:

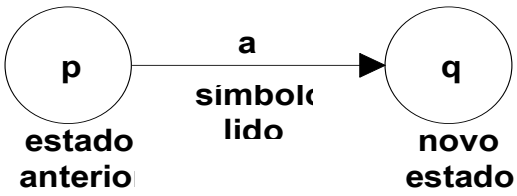


Figura 2-3: Representação da Função programa como um grafo.

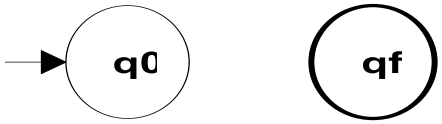


Figura 2-4: Representação dos estados inicial (com uma seta) e final (com linha grossa) como nodos de um grafo.

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 direita, até ocorrer uma condição de parada.

Exemplo: Autômato Finito

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 representada pela tabela abaixo e pelo grafo da Figura 2-5, reconhece a linguagem

L1 = {w | w possui aa ou bb como subpalavra}

	δ_1	a	b
→	q0	q1	q2
	q1	qf	q2
	q2	q1	qf
*	qf	qf	qf

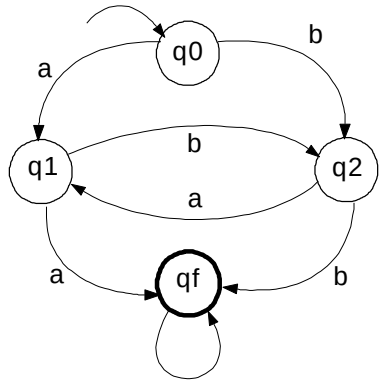


Figura 2-5: Grafo do autômato finito determinístico M1.

O algoritmo apresentado usa os estados q1 e q2 para "memorizar" o símbolo anterior. Assim q1 representa "o símbolo anterior é a" e q2 representa "o símbolo anterior é b". Após identificar dois aa ou dois bb consecutivos o autômato assume o estado qf (final) e varre o sufixo da palavra de entrada sem qualquer controle lógico, somente para terminar o processamento. A Figura 2-6 ilustra o processamento do autômato finito M1 para a palavra de entrada w = abba, a qual é aceita.

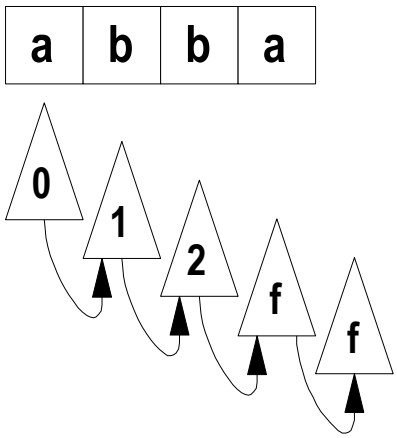


Figura 2-6: Seqüência de processamento.

Note-se que um autômato finito sempre pára ao processar qualquer entrada, pois como toda palavra é finita e como um novo símbolo de entrada é lido a cada aplicação da

função programa, não existe a possibilidade de ciclo (loop) infinito. A parada do processamento pode ocorrer 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 para e a entrada w é aceita.
b) Após processar o último símbolo da fita, o autômato finito assume um estado não final. O autômato para e a entrada w é rejeitada
c) A função programa é indefinida para o argumento (estado corrente e símbolo lido). O autômato para e a entrada w é rejeitada.

Definição: Linguagens Regulares ou do Tipo 3
Uma linguagem aceita por um autômato finito é uma *Linguagem Regular* ou do *Tipo 3*.

Exercício
Desenvolver AFDs que reconheçam as seguintes linguagens sobre $\Sigma = \{a, b\}$:
a) {w | w possui aaa como subpalavra}
b) {w | o sufixo de w é aa}
c) {w | w possui um número ímpar de a e b}
d) {w | w possui número par de a e ímpar de b ou vice-versa}
e) {w | o quinto símbolo da esquerda para a direita de w é a}

Exemplos: Autômato Finito
Os autômatos M2 = ({a,b}, {q0}, δ2, q0, ∅) e M3 = ({a, b}, {q0}, δ3, q0, {q0}), cujo diagramas estão na Figura 2-7, reconhecem respectivamente as linguagens L1 = ∅ e L2 = Σ*, onde δ2 e δ3 são representadas abaixo em forma de tabela.

M2

	δ2	a	b
→	q0	q0	q0

M3

	δ3	a	b
* →	q0	q0	q0

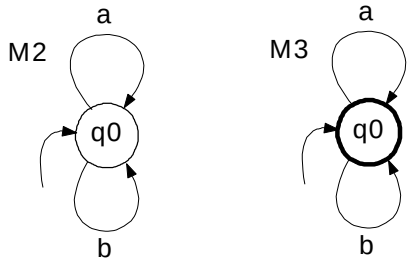


Figura 2-7: Exemplo de Autômato Finito.

O autômato M4 = ({a, b}, {q0, q1, qa2, q3}, δ, q0, {q0}), Figura 2-8, reconhece a linguagem:

L4 = {w | w possui um número par de a e b}.

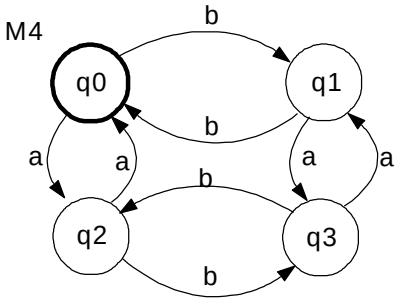


Figura 2-8: Diagrama de transições de estado para o autômato M4.

2.1.2 Autômato Finito Não-Determinístico (AFND)

- Não-determinismo é uma importante generalização dos AF's, essencial para a teoria da computação e para a teoria das linguagens formais.
- Qualquer AFND pode ser simulado por um autômato finito determinístico.

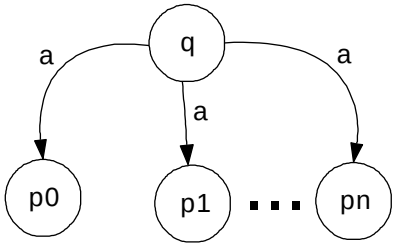


Figura 2-9: Exemplo de um AFND.

- Em AFNDs, a função programa leva de um par (estado, símbolo) a um conjunto de estados possíveis.
- Pode-se entender que o AFND assume simultaneamente todas as alternativas de estados possíveis {p0, p1, ..., pn} a partir do estado atual ($q \in Q$) e do símbolo recebido ($a \in \Sigma$), como se houvesse uma unidade de controle para processar cada alternativa independentemente, sem compartilhar recursos com as demais, Figura 2-9.
- Assim o processamento de um caminho não influi no estado, símbolo lido e posição da cabeça dos demais caminhos alternativos.

Definição: Autômato Finito Não-Determinístico (AFND)

Um AFND é uma quíntupla

$$M = (\Sigma, Q, \delta, q0, F),$$

onde:

- Σ - Alfabeto de símbolos de entrada
- Q - Conjunto finito de estados possíveis do autômato
- δ - Função programa ou função de transição $\delta: Q \times \Sigma \rightarrow 2^Q$, parcial.
- q_0 - Estado inicial tal que $q_0 \in Q$
- F - Conjunto de estados finais, tais que $F \subseteq Q$.

Portanto, os componentes do AFND são os mesmos do AFD, com exceção da função programa (ver Figura 2-9).

Exemplo: Autômato Finito Não-Determinístico
O AFND M5 = ({a, b}, {q0, q1, q2, qf}, δ5, q0, {qf}), veja Figura 2-10, reconhece a linguagem L5 = {w | w possui aa ou bb como sub-palavra}, onde δ5 é dada abaixo, na forma de tabela:

	δ_5	a	b
→	q0	{q0, q1}	{q0, q2}
	q1	{qf}	-
	q2	-	{qf}
*	qf	{qf}	{qf}

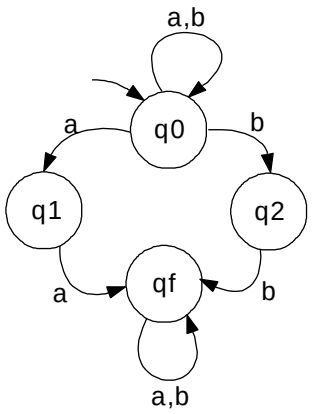


Figura 2-10: Diagrama de transições de estados do AFND M5.

Exemplo: Autômato Finito Não-Determinístico
O AFND M6 = ({a,b}, {q0, q1, q2, qf}, δ6, q0, {qf}), representado na Figura 2-11 reconhece a linguagem L6 = { w | w possui aaa como sufixo }

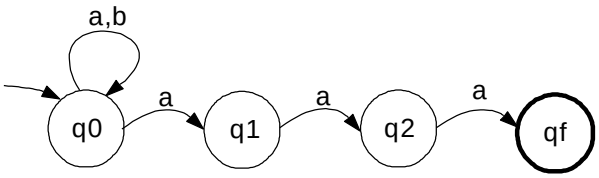


Figura 2-11: Diagrama de transições do AFND M6.

2.1.3 Autômato Finito com Movimentos vazios (ou com ε-Transição)

- *Movimentos vazios* constituem uma generalização dos AFND e são transições que ocorrem sem que haja a leitura de símbolo algum.
- Os movimentos vazios podem ser interpretados como um não-determinismo interno do autômato, que é encapsulado.
- A não ser por uma eventual mudança de estados, nada mais pode ser observado sobre um movimento vazio.
- Qualquer AFε pode ser simulado por um autômato finito não-determinístico.

Definição: Autômato Finito com Movimento Vazio (AFε)

Um autômato finito não-determinístico e com movimento vazio (AFNDε), ou simplesmente autômato finito com movimento vazio (AFε), é uma quíntupla:

$$M = (\Sigma, Q, \delta, q0, F),$$

onde:

- Σ - Alfabeto de símbolos de entrada
- Q - Conjunto finito de estados possíveis do autômato
- δ - Função programa ou função de transição δ: Q x $(\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$, parcial.
- q_0 - Estado inicial tal que $q_0 \in Q$
- F - Conjunto de estados finais, tais que $F \subseteq Q$.

Portanto, os componentes do AFε são os mesmos do AFND, com exceção da função programa, veja a Figura 2-12.

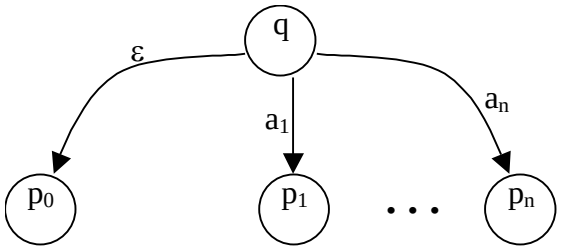


Figura 2-12: Esquema genérico de um AFNDε possuindo uma ε-transição.

O processamento dos AFε é similar ao dos AFND. Por analogia 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 de origem e destino da transição.

Exemplo: Autômato Finito com Movimento Vazio

O AFε M7 = ({a,b}, {q_0, q_f}, δ7, q0, {q_f}), representado na figura abaixo reconhece a linguagem L7 = {w | qualquer símbolo *a* antecede qualquer símbolo *b*}, onde δ7 é representada na forma da tabela:

	δ7	a	b	ε
→	q_0	{q_0}	-	{q_f}
*	q_f	-	{q_f}	-

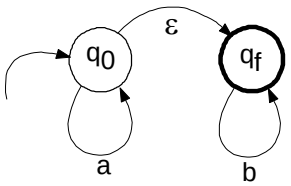


Figura 2-13: Diagrama do AFε M7.

2.2 Operações de Concatenação e Fechamento

Seja C um conjunto finito de símbolos (alfabeto), e sejam L, L_1 e L_2 subconjuntos de C^* (linguagens sobre o alfabeto C). A concatenação de L_1 e L_2, denotada por L_1L_2, é o conjunto $\{xy \mid x \in L_1 \ e \ y \in L_2\}$. Em outras palavras, as cadeias da linguagem L_1L_2 são formadas por uma cadeia de L_1 concatenada a uma cadeia de L_2, nesta ordem, incluindo aí todas as combinações possíveis.

Define-se $L^0 = \{\varepsilon\}$ e $L^n = L\ L^{n-1}$ para $n \geq 1$. O *fecho de Kleene* (ou simplesmente *fecho*) de uma linguagem L, denotado por L^*, é o conjunto:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

e o *fecho positivo* da linguagem L, denotado por L^+, é o conjunto:

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

Em outras palavras, L^* denota as cadeias construídas pela concatenação de qualquer número de cadeias tomadas de L. O conjunto L^+ é semelhante, mas neste caso, as cadeias de zero palavras, cuja concatenação é definida como ε, são excluídas. Note-se, porém que L^+ contém ε se e somente se L a contém. Esta definição difere da definição do fechamento de alfabetos, onde A^+ era definido como A^*- {ε}. Note-se que no caso de linguagens podem ocorrer dois casos:

a) Se $\varepsilon \in L$, então $L^+ = L^*$
b) Se $\varepsilon \notin L$, então $L^+ = L^* - \{\varepsilon\}$

Exemplo 1:
Se $L_1 = \{10,1\}$ e $L_2 = \{0011,11\}$. Então $L_1L_2 = \{100011,1011,10011,111\}$

Exemplo 2:
Se $L_1 = \{10,11\}$ e $L_2 = \{0011,11\}$. Então $L_1L_2 = \{100011,1011,110011,1111\}$

2.3 Expressões Regulares

Uma expressão regular (ER) sobre um alfabeto Σ é indutivamente definida como se segue:

a) ∅ é uma ER que denota a linguagem vazia.
b) ε é uma ER que denota a linguagem contendo exclusivamente a cadeia vazia, ou seja {ε}.
c) Qualquer símbolo x pertencente ao alfabeto Σ é uma ER e denota a linguagem contendo a cadeia unitária x, ou seja {x}.
d) Se r e s são ERs e denotam respectivamente as linguagens R e S, respectivamente, então:
   i) (r+s) é ER e denota a linguagem $R \cup S$
   ii) (rs) é ER e denota a linguagem $\{uv \mid u \in R \ e \ v \in S\}$
   iii) (r*) é ER e denota a linguagem R*

Observações:
1. Os parênteses podem ser omitidos, respeitando-se as seguintes prioridades de operações:
   - A concatenação sucessiva tem precedência sobre a concatenação e a união, e
   - A concatenação tem precedência sobre a união.
2. Uma linguagem gerada por um expressão regular r é representada por L(r) ou GERA(r).

Exemplos:

Expressão Regular	**Linguagem Representada**
aa	Somente a cadeia aa.
ba*	Todas as cadeias que iniciam por b, seguido de zero ou mais a.
(a+b)*	Todas as cadeias sobre o alfabeto {a, b}
(a+b)*aa(a+b)*	Todas as cadeias contendo aa como subpalavra.
a*ba*ba*	Todas as cadeias contendo exatamente dois b
(a+b)*(aa+bb)	Todas as cadeias que terminam com aa ou bb.
(a+ε)(b+ba)*	Todas as cadeias que não possuem dois a consecutivos
0*1*2*	Todas as cadeias de 0 seguidas de 1's seguidas d 2's

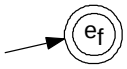
Exercícios:
Desenvolva expressões regulares que gerem as seguintes linguagens sobre Σ = {a, b}:
1. {w | w tem no máximo um par de a como subcadeia e no máximo um par de b como subcadeia}.
2. {w | qualquer par de a antecede qualquer par de b}
3. {w | w não possui aba como subpalavra}

2.4 Equivalência entre AF's e ER's

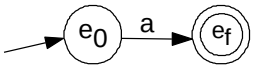
Pode-se facilmente construir um AF com ε-transições a partir de qualquer expressão regular. O método deve ser aplicado para a base (casos (a), (b) e (c)) e indutivamente para cada tipo de construção das ER´s.

a) Para expressão ∅, construa o seguinte AF:
b)

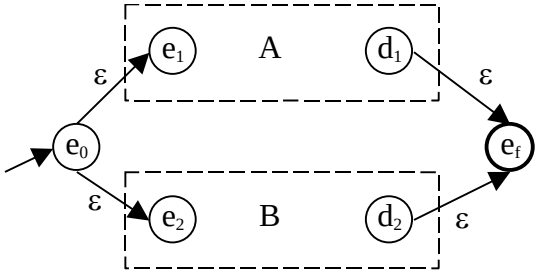
c) Para ER ε, construa o seguinte AF:
d)



c) Para ER *a* (para algum a∈Σ), construa o seguinte AF:
d)

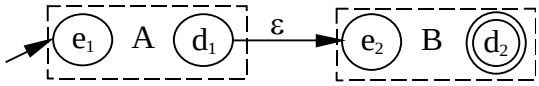


d) Para ER *(A + B)*, construa a seguinte estrutura:



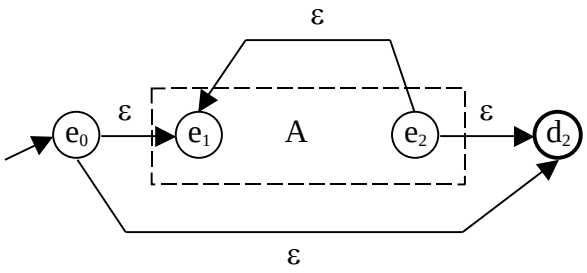
onde e_1 e d_1 são, respectivamente, os estados inicial e final da ER A, e e_2 e d_2 são, respectivamente, os estados inicial e final da ER B.

e) Para ER *AB*, construa a seguinte estrutura:



onde e_1 e d_1 são, respectivamente, os estados inicial e final da ER A, e e_2 e d_2 são,respectivamente, os estados inicial e final da ER B.

f) Para ER *A**, construa a seguinte estrutura:



onde e_1 e e_2 são, respectivamente, os estados inicial e final da ER A.

2.5 Transformação de GR em AF

Para transformar uma gramática regular G = (N, T, P, S) em um autômato finito M=(E,A,t,e_0,F), proceda como segue:

Algoritmo: Transformação de GR em AF
Entrada: Uma Gramática Regular G = (N, T, P, S)
Saída: Um Autômato Finito M=(E, A, t, e_0, F)

E:= N ∪ {e_{novo}, S}, onde e_{novo} é um símbolo que não pertence a N;
A := T;
e_0 := S;
se S ::= ε ∈ P então
    F := { e_{novo}, S}
senão
    F := { e_{novo}};

fimSe;
Construa *t* de acordo com as seguintes regras:

a) Para cada produção da forma B::=a∈P c/ a∈{T∪{ε}}, crie uma transição t(B,a) = e_{novo};

b) Para cada produção da forma B::=aC ∈ P, crie uma transição t(B,a) = C;

c) Para todo *a*∈ T, deixe *t*(e_{novo},a) indefinida (ou use o estado ERRO).

2.6 Transformação de AF em GR

Para transformar um AF em uma gramática regular equivalente, proceda como segue:

Algoritmo: Transformação de AF em GR
Entrada: Um Autômato Finito M=(E, A, t, e_0, F)
Saída: Uma Gramática Regular G = (N, T, P, S)

N := E;
T := A;
S := e_0;
Defina *P* de acordo com as seguintes regras:

a) Se t(B, a) = C então adicione B::=aC em P;

b) Se t(B, a) = C e C∈F, então adicione B::=a em P;

c) Se e_0∈F, então adicione S::= ε em P.

2.7 Determinização de AFND

Por definição, todo AFD é um caso especial de AFND no qual a relação de transição é uma função. Assim, a classe de linguagens reconhecidas por um AFND inclui as linguagens regulares (aquelas que são reconhecidas por AFD's). Entretanto, pode-se provar que as linguagens regulares são as únicas linguagens reconhecidas por um AFND. Para isto, basta mostrar que para qualquer AFND pode-se construir um AFD que reconhece a mesma linguagem. Um método de transformação é dado a seguir.

Algoritmo: Determinização de Autômato Finito
Entrada: Um AFND MN = (E, A, t, e_0, F)
Saída: Um AFD MD = (E', A, t', e_0, F')

1. Rotule a primeira linha da tabela de transições t' para MD com um conjunto unitário contendo apenas o estado inicial e_0 de MN. Aplique o passo (2) a este conjunto.

2. Dado um conjunto de estados S, rotulando uma linha da tabela t' para MD, para a qual as transições ainda não foram computadas, encontre os estados de MN que podem ser alcançados a partir dos estados contidos em S para cada símbolo de entrada; coloque estes estados na coluna correspondente ao símbolo de entrada na tabela para MD.

3. Para cada novo conjunto gerado pelas transições do passo (2), determine se ele já é usado como rótulo para alguma linha de MD. Se o conjunto ainda não foi usado, então crie uma nova linha com o conjunto como rótulo. Se o conjunto já foi usado, não é necessário fazer nada com ele.

4. Se existe alguma linha em MD para a qual as transições não forma computadas, volte e aplique o passo (2) àquela linha. Se todas as transições já forma computadas, vá pra o passo (5).

5. Para todas as linhas de MD rotuladas com conjuntos que contenham pelo menos um estado final de MN, marque esta linha como estado final de MD. Fim.

Exemplo: Construção de um AFD a partir do AFND M6.

Determinizar o AFND M6 = ({a,b}, {q_0, q_1, q_2, q_f}, t, q_0, {q_f}), dado na Figura 2-11 e repetido a seguir.

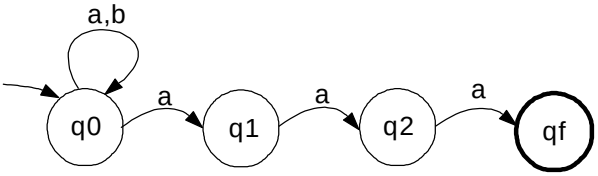


Tabela de *t* para o autômato M6 mostrado acima.

	t	a	b
→	q_0	q_0, q_1	q_0
	q_1	q_2	-
	q_2	q_f	-
*	q_f	-	-

Executando o algoritmo:

t'	a	b
q_0	q_{01}	q_0
q_{01}	q_{01}, q_2	q_0

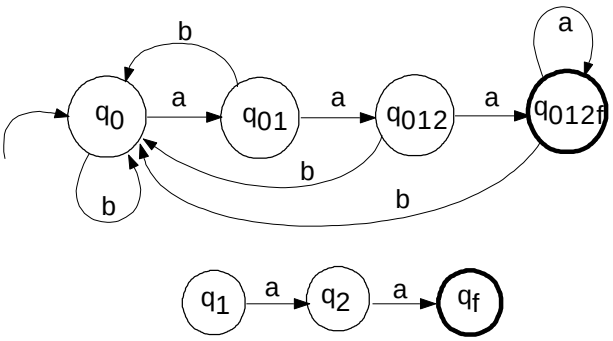
t'	a	b
q_0	q_{01}	q_0
q_{01}	q_{012}	q_0
q_{012}	q_{012}, q_f	q_0

t'	a	b
q_0	q_{01}	q_0
q_{01}	q_{012}	q_0
q_{012}	q_{012f}	q_0
q_{012f}	q_{012f}	q_0

Adicionando os estados restantes e determinando os estados finais, tem-se a tabela do autômato M6 determinizado (M6D):

	t'	a	b
→	q_0	q_{01}	q_0
	q_{01}	q_{012}	q_0
	q_{012}	q_{012f}	q_0
*	q_{012f}	q_{012f}	q_0
	q_1	q_2	-
	q_2	q_f	-
*	q_f	-	-

O AFD M6D = ({a, b}, Q', t', q_0, F'), construído conforme o algoritmo dado é:



onde:
Q' = $\{q_0, q_1, q_2, q_f, q_{01}, q_{012}, q_{012f}\}$
F' = $\{q_f, q_{012f}\}$.
t' = Exibida acima.

2.8 Equivalência entre AF's com e sem ε-transições

Para um autômato finito com ε-transições (ou com movimento vazio) define-se o ε-fechamento de um estado *e* como o conjunto de estados alcançáveis a partir de *e* utilizando-se somente ε-transições. Para o exemplo a seguir tem-se o seguinte:

		0	1	2	ε
→	e_0	e_0	∅	∅	e_1
	e_1	∅	e_1	∅	e_2
*	e_2	∅	∅	e_2	∅

ε-fecho(e_0) = $\{e_0, e_1, e_2\}$
ε-fecho(e_1) = $\{e_1, e_2\}$
ε-fecho(e_2) = $\{e_2\}$

Seja M = (E, A, t, e_0, F), um AF com ε-transições, pode-se construir M' = (E, A, t', e_0, F') sem as ε-transições da seguinte maneira:

Algoritmo: Eliminação de ε-transições
Entrada: Um AFND M = (E, A, t, e_0, F)
Saída: Um AFND sem ε-transições M' = (E, A, t', e_0, F')
1. F' será definido por $F \cup \{e \mid e \in E$ e ε-fecho(e) contém um estado de F};

2. t' deve conter todas as transições de t que não são ε-transições e mais aquelas que são obtidas pela composição de transições de t' com as ε-transições de t, da seguinte maneira:
    a) se $(e_1, a, e_2) \in$ t' e $e_3 \in$ ε-fecho(e_2) então coloque $(e_1, a, e_3) \in$ t';
    a) se $(e_1, a, e_2) \in$ t' e $e_1 \in$ ε-fecho(e_3) então coloque $(e_3, a, e_2) \in$ t';

2.9 Minimização de Autômatos Finitos

Diz-se que um AF é *mínimo* se ele não possui estados inatingíveis ou mortos e se não existem estados equivalentes entre si. Se um AF não é mínimo, então um AF equivalente com menos estados pode ser obtido pela eliminação dos estados inatingíveis e mortos e pela combinação dos estados equivalentes. Este processo de redução pode ser repetido até que um AF mínimo seja encontrado. Assim, todo AF tem um AF mínimo correspondente.

Realizando estas reduções por caminhos diferentes, ou iniciando com máquinas equivalentes, mas diferentes, pode-se obter máquinas mínimas que pareçam ser diferentes. Entretanto, estas máquinas serão, de fato, idênticas em tudo, com exceção dos nomes usados para os estados.

Pode-se concluir que, exceto pelos nomes dos estados, existe apenas uma máquina mínima para um dado problema de reconhecimento. Isto significa que não importa a forma do AF inicialmente encontrado para reconhecer uma determinada linguagem e não importa a forma com que as reduções são feitas: existe apenas uma máquina mínima que pode ser encontrada pra este problema.

2.9.1 Estados Inatingíveis

Pode haver alguns estados no AF que nunca serão atingidos com nenhuma seqüência de símbolos a partir do estado inicial.

Tais estados são chamados de *estados inatingíveis*. As linhas que correspondem a estados inatingíveis podem ser simplesmente removidas da tabela de transição para deixar a máquina com menos estados.

Algoritmo: Eliminação de Estados Inatingíveis
Entrada: Um Autômato Finito M = (E, A, t, e_0, F)
Saída: Um Autômato Finito sem estados Inatingíveis M' = (E', A, t', e_0, F')

1. Inicialize o conjunto de estados atingíveis E' com o estado inicial e_0.
2. Adicione, em E', para cada estado *e*, presente no conjunto E', todos os estados *d* que podem ser alcançados por uma transição a partir de *e*, ou seja, todos os estados *d* tal que $t(e,x) = d$ para algum $x \in A$.
3. Se nenhum novo estado pode ser adicionado ao conjunto E' com o uso destas regras, já foi obtido o conjunto de estados atingíveis.
4. Coloque em t' todas as transições $t(x,y)=z$ tal que $x,z \in$ E' e $y \in$ A. Coloque em F' todos os estados que pertencem simultaneamente a F e a E'.

2.9.2 Estados Mortos

Um estado *e* de um AF é *morto* se ele não é final e se a partir dele não se pode atingir nenhum estado final. Para eliminar os estados mortos de um AF M proceda como segue:

Algoritmo: Eliminação de Estados Mortos
Entrada: Um AF M = (E, A, t, e_0, F)
Saída: Um AF sem estados mortos M' = (E', A, t', e_0', F)
1. Inicialize o conjunto de estados não-mortos E' com os estados finais F de M.
2. Para cada estado *e* em E', adicione os estados de E que levam para *e* com uma transição de *t* para algum símbolo de entrada. Ou seja, adicione em E' o conjunto $\{x| \, t(x,a) = e, e \in E' \text{ e } a \in A\}$
3. Repita o passo (2) até que mais nenhum estado possa ser adicionado ao conjunto E'.
4. Coloque em t' todas as transições $t(x,y) = z$ tal que $x, z \in E'$ e $y \in A$. Se $e_0 \in E'$ então faça $e_0' = e_0$; caso contrário, e_0' é um novo símbolo de estado e a linguagem reconhecida pelo autômato é vazia, uma vez que o estado inicial é morto.

2.9.3 Estados Equivalentes

Dois estados *e* e *d* são equivalentes se e somente se as duas condições seguintes forem satisfeitas:

a) Condição de compatibilidade: os estados *e* e *d* devem ser ambos finais ou ambos não-finais.
b) Condição de propagação: para qualquer símbolo de entrada, os estados *e* e *d* devem levar a estados equivalentes.

As condições (a) e (b) podem ser incorporadas em um teste geral de equivalência entre estados. O método é chamado método da separação, uma vez que seu objetivo é separar, ou particionar, o conjunto de estados em subconjuntos disjuntos, ou blocos, tal que estados não-equivalentes fiquem em blocos separados. O método é ilustrado por sua aplicação no AF da tabela seguinte:

	t	a	b
→	e_1	e_6	e_3
	e_2	e_7	e_3
	e_3	e_1	e_5
	e_4	e_4	e_6
*	e_5	e_7	e_3
*	e_6	e_4	e_1
*	e_7	e_4	e_2

Os estados são inicialmente separados em dois blocos; um contendo os estados finais e outro contendo os estados não-finais. Para o exemplo, a partição inicial L_0 é dada pela seguinte lista:

$$L_0 = [\{e_1, e_2, e_3, e_4\}, \{e_5, e_6, e_7\}]$$

já que e_1, e_2, e_3 e e_4 são estados não finais e e_5, e_6 e e_7 são estados finais. Nenhum dos estados no primeiro bloco é equivalente a nenhum dos estados no segundo bloco, porque tal para violaria a condição de compatibilidade.

Agora, observe o que acontece aos estados no bloco $\{e_1, e_2, e_3, e_4\}$ com entrada *a*. Os estados e_3 e e_4 vão para estados contidos no primeiro bloco (e_1 e e_4 respectivamente), enquanto os estados e_1 e e_2 vão para estados no segundo bloco (e_6 e e_7 respectivamente). Isto significa que para qualquer estado no conjunto $\{e_1, e_2\}$ e qualquer estado em $\{e_3, e_4\}$ os estados seguintes correspondendo a entrada *a* não serão equivalentes. Isto é uma violação da condição de propagação, e assim se pode concluir que nenhum estado em $\{e_1, e_2\}$ será equivalente a nenhum estado em $\{e_3, e_4\}$. Isto habilita a construção de uma nova partição:

$$L_1 = [\{e_1, e_2\},\{e_3, e_4\},\{e_5, e_6, e_7\}]$$

Com a propriedade de que estados tomados de blocos diferentes serão sempre não equivalentes.

Agora se tentará encontrar um bloco em L_1 e uma entrada tal que o bloco possa ser separado com respeito a entrada e uma nova partição seja assim obtida. Esta nova partição também terá a propriedade de que os estados tomados de diferentes blocos serão garantidamente não-equivalentes. Este processo deve ser repetido até que nenhuma nova separação seja possível. No exemplo, a continuação seria a seguinte:

Separando $\{e_3, e_4\}$ de L_1 com respeito a *a*:

$$L_2 = [\{e_1, e_2\},\{e_3\},\{e_4\},\{e_5, e_6, e_7\}]$$

Separando $\{e_5, e_6, e_7\}$ de L_2 com respeito a *a* (ou *b*):

$$L_3 = [\{e_1, e_2\},\{e_3\},\{e_4\},\{e_5\},\{e_6, e_7\}]$$

A partição L_3 não pode ser mais separada. Para verificar isto, observe que todos os estados no bloco $\{e_1, e_2\}$ vão para estados no bloco$\{e_6, e_7\}$ com entrada *a*, e para o bloco $\{e_3\}$ com entrada *b*. Similarmente $\{e_6, e_7\}$ vão para os blocos $\{e_4\}$ e $\{e_1, e_2\}$ com entrada *a* e *b*, respectivamente. Os outros blocos têm apenas um elemento, e assim não podem mais ser separados.

Quando o procedimento termina, os estados dentro de um mesmo bloco são equivalentes. No exemplo, os estados e_1 e e_2 são equivalentes e os estados e_6 e e_7 também são equivalentes.

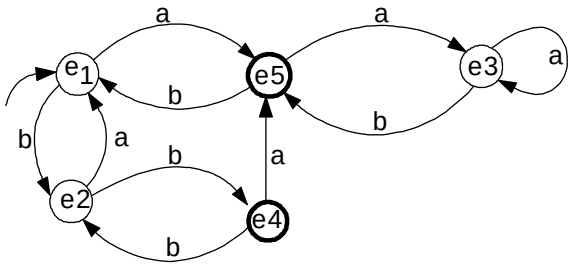
Os blocos da partição final podem ser usados para construir uma nova máquina, a qual é equivalente à original, porém sem possuir estados equivalentes. Para o exemplo a máquina mínima resultante seria:

	t	a	b
→	$\{e_1, e_2\}$	$\{e_6, e_7\}$	$\{e_3\}$
	$\{e_3\}$	$\{e_1, e_2\}$	$\{e_5\}$
	$\{e_4\}$	$\{e_4\}$	$\{e_6, e_7\}$
*	$\{e_5\}$	$\{e_6, e_7\}$	$\{e_3\}$
*	$\{e_6, e_7\}$	$\{e_4\}$	$\{e_1, e_2\}$

ou

	t	a	b
→	e_1	e_5	e_2
	e_2	e_1	e_4
	e_3	e_3	e_5
*	e_4	e_5	e_2
*	e_5	e_3	e_1

ou em forma de diagrama de transição (com os estados renomeados):



O conjunto de estados da nova máquina é o conjunto de blocos da partição final. As transições para a nova máquina são obtidas a partir da antiga observando quais blocos seguem quais para cada entrada. Assim, na tabela acima, a transição do estado $\{e_1, e_2\}$ com entrada *a* é definida para o estado $\{e_6, e_7\}$, porque os estados contidos no primeiro bloco vão para os estados do segundo bloco com entrada *a*. O estado inicial da nova máquina é simplesmente o bloco que contém o estado inicial da máquina original e os estados finais são aqueles blocos que contêm estados finais da máquina original.

Algoritmo: Eliminação de Estados Equivalentes
Entrada: Um Autômato finito M = (E, A, t, e_0, F)
Saída: Um Autômato finito sem estados equivalentes M' = (E', A, t', e_0', F')

Crie uma lista de partições L_0 = [E – F, F].
i :=0;
Repita:
    L_{i+1} := [];
    Para cada elemento *p* de L_i faça:
        Se | *p* | = 1 então:
            Coloque *p* em L_{i+1}
        Senão
            p' := ∅;
            e := "um estado qualquer de *p*";
            coloque *e* em *p*';
            para todo símbolo $d \in p-\{e\}$ faça:
                se *d* e *e* são equivalentes então:
                    adicione *d* em *p*'
                fim se;
            fim para;
            coloque *p*' em L_{i+1};
            se *p*' ≠ *p* então
                coloque *p* – *p*' em L_{i+1}
            fim se;
        fim se;
    fim para;
    i := i + 1;
até $L_i = L_{i+1}$;
Crie um símbolo de estado e_j para cada elemento p_j da partição L_i;
Coloque estes estados e_j no conjunto E';
Para cada transição t(*e*, *a*) = *d* faça
    Crie a transição t'(e_j, *a*) = e_l onde $e \in p_j$ e $d \in p_l$, sendo e_l símbolo de p_j e *e* de *p*.

Fim para.
Coloque em F' todos os estados e_j de E' originados a partir de algum p_j tal que $x \in p_j$ e $x \in$ F.
e_0' = e_j, tal que e_j se originou de p_j e e_0 de p_j.

2.9.4 Estado de Erro

Para não deixar transições indefinidas em M, pode-se substituir todas as indefinições por um estado de erro "ERRO" e adicionar este estado à tabela. Para cada símbolo de entrada o estado "ERRO" tem transição para si próprio. O estado de erro não deve jamais ser definido como estado final.

2.9.5 Autômato Finito Mínimo

Para encontrar um AF mínimo a partir de um AF qualquer preceda como segue:

Algoritmo: Minimização de Autômato Finito
Entrada: um Autômato Finito M = (E, A, t, e_0, F).
Saída: um AF mínimo M' = (E', A, t', e_0', F').
1. Elimine os estados inatingíveis (ou inalcançáveis);
2. Elimine os estados mortos;
3. Adicione o estado de erro;
4. Elimine os estados redundantes

3. Linguagens Livres de Contexto

A importância das LLC's reside no fato de que praticamente todas as linguagens de programação podem ser descritas por este formalismo, e que um conjunto bastante significativo destas linguagens pode ser analisado por algoritmos muito eficientes. A maioria das linguagens de programação pertence a um subconjunto das LLC's que pode ser analisado por algoritmos que executam *n.log(n)* passos para cada símbolo da sentença de entrada. Outro subconjunto, que corresponde às linguagens não ambíguas pode ser analisado em tempo n^2. No pior caso, uma LLC ambígua pode ser analisada em tempo n^3. O que ainda é bastante eficiente, se comparado às linguagens sensíveis ao contexto que pode requerer complexidade exponencial (2^n), sendo, portanto, computacionalmente intratáveis, e as linguagens de tipo 0, que podem ser inclusive indecidíveis, isto é, o algoritmo de análise pode não parar nunca.

Relembrando, uma *gramática livre de contexto* (GLC) é denotada por G = (N, T, P, S), onde N e T são conjuntos disjuntos de variáveis e terminais, respectivamente, P é um conjunto finito de produções, cada uma da forma A ::= α, onde A é uma variável do conjunto N e α é uma cadeia de símbolos de $(T \cup N)^*$. Finalmente, S é uma variável especial denominada símbolo inicial.

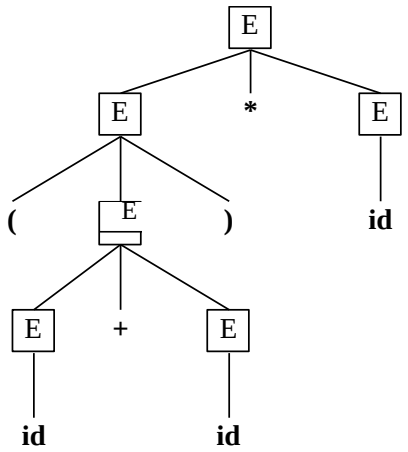
3.1 Árvore de Derivação e Ambigüidade

Muitas vezes é útil mostrar as derivações de uma GLC através de árvores. Estas figuras, chamadas *árvores de derivação* ou *árvores sintáticas*, impõem certas estruturas úteis às cadeias das linguagens geradas, principalmente as de programação.

Os vértices de uma árvore de derivação são rotulados com terminais ou variáveis, ou mesmo com a cadeia vazia. Se um vértice *n* é rotulado com A, e os filhos de *n* são rotulados com X_1, X_2,..., X_k, da esquerda para a direita, então A ::= $X_1X_2...X_k$ deve ser uma produção. Considere, como exemplo, a gramática:

E ::= E+E | E*E|(E)|**id**

Uma árvore de derivação para a sentença "(id+id)*id" gerada por esta gramática poderia ser:



Mais formalmente, seja G = (N, T, P, S) uma GLC. Então uma árvore é uma árvore de derivação para G, se:

a) Cada vértice tem um rótulo, que é um símbolo de $N \cup T \cup \{\varepsilon\}$.
b) O rótulo da raiz é o símbolo S.
c) Os vértices interiores são rotulados apenas com símbolos de N.
d) Se o vértice *n* tem rótulo A, e os vértices n_1, n_2,..., n_k são filhos de *n*, da esquerda para a direita, com rótulos X_1, X_2,..., X_k, respectivamente, então:

$$A ::= X_1X_2...X_k$$

deve ser uma produção em P.

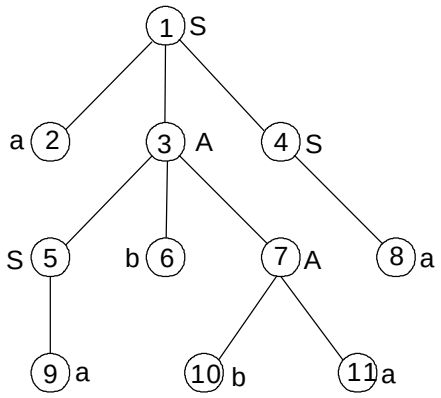
e) Se o vértice *n* tem rótulo ε, então *n* é uma folha e ε é o único filho de seu pai.

Exemplo

Considere a gramática G = ({S, A}, {a, b}, P, S), onde P consiste de

S ::= aAS | a
A ::= SbA | SS | ba

A figura seguinte é uma árvore de derivação para uma sentença desta gramática, na qual os nodos foram numerados para facilitar a explicação:



Os vértices interiores são 1, 3, 4, 5 e 7. O vértice 1 tem rótulo S, e seus filhos, da esquerda para a direita têm rótulos a, A e S. Note que S ::= aAS é uma produção. Da mesma forma, o vértice 3 tem rótulo A, e os rótulos de seus filhos são S, b e A, da esquerda para a direita; A :: SbA é também uma produção. Os vértices 4 e 5 têm ambos o rótulo S. O único filho deles tem rótulo a; e S ::= a é uma produção. Finalmente, o vértice 7 tem rótulo A, e seus filhos, da esquerda para a direita, tem rótulos b e a; e A ::= ba é também uma produção. Assim as condições para que esta árvore seja uma árvore de derivação para G foram cumpridas.

Pode-se estender a ordem "da esquerda para a direita" dos filhos para produzir uma ordem da esquerda para a direita de todas as folhas. No exemplo acima, o caminhamento da esquerda para a direita nas folhas da árvore produziria a seguinte seqüência: 2, 9, 6, 10, 11 e 8.

Pode-se ver que a árvore de derivação é uma descrição natural para a derivação de uma forma sentencial particular da gramática G.

Uma sub-árvore de uma árvore de derivação é composta por um vértice particular da árvore, juntamente com seus descendentes.

Se uma gramática G tem mais de uma árvore de derivação para uma mesma cadeia, então G é chamada de *gramática ambígua*. Uma linguagem livre de contexto para a qual toda gramática livre de contexto é ambígua é denominada *linguagem livre de contexto inerentemente ambígua*.

3.2 Derivação mais à Esquerda e mais à Direita

Se cada passo na produção de uma derivação é aplicado na variável mais à esquerda, então a derivação é chamada *derivação mais à esquerda*. Similarmente, uma derivação onde a cada passo a variável à direita é substituída, é chamada de *derivação mais à direita*.

Se *w* está em L(G), então *w* tem ao menos uma árvore de derivação. Além disso, em relação a uma árvore de derivação particular, *w* tem uma única derivação mais à esquerda e uma única derivação mais à direita.

Evidentemente, *w* pode ter várias derivações mais à esquerda e várias derivações mais à direita, já que pode haver mais de uma árvore de derivação para *w*. Entretanto, é fácil mostrar que para cada árvore de derivação apenas uma derivação mais à esquerda e uma derivação mais à direita pode ser obtida.

Exemplo:

derivação mais à esquerda correspondendo à árvore do exemplo anterior

S $\rightarrow$ aAS
$\rightarrow$ aSbAS
$\rightarrow$ aabAS
$\rightarrow$ aabbaS
$\rightarrow$ aabbaa

derivação mais à direita correspondente

S $\rightarrow$ aAS
$\rightarrow$ aAa
$\rightarrow$ aSbAa
$\rightarrow$ aSbbaa
$\rightarrow$ aabbaa

3.3 Simplificações de Gramáticas Livres de Contexto

Existem várias maneiras de restringir as produções de uma gramática livre de contexto sem reduzir seu poder expressivo. Se L é uma linguagem livre de contexto não vazia, então L pode ser gerada por uma gramática livre de contexto G com as seguintes propriedades:

a) Cada variável e cada terminal de G aparecem na derivação de alguma palavra de L.
b) Não há produções da forma A::=B, onde A e B são variáveis.
c) Se ε não está em L, então não há necessidade de produções da forma A::=ε.

No caso de (a) estes símbolos (variáveis ou terminais) são conhecidos como símbolos inúteis (símbolos improdutivos e símbolos inalcançáveis). Em (b) as produções A::=B são conhecidas como produções unitárias. As produções da forma A::=ε (item c) são conhecidas como ε-produções. Existem algoritmos que podem ser aplicados nos casos (a), (b) e (c) que produzem uma nova gramática equivalente à gramática original. Veremos apenas o algoritmo para eliminar as ε-produções.

3.3.1 Eliminação das ε-produções

Conforme vimos uma GLC pode ter produções do tipo ::=ε. Mas toda GLC pode ser transformada em uma GLC equivalente sem estes tipos de produções (chamadas ε-produções), com exceção da produção S::=ε (S é o símbolo inicial), se esta existir. Assim procedendo, é possível mostrar que toda GLC pode obedecer à restrição das GSC (tipo 1).

O método de eliminação das ε-produções consiste em determinar, para cada variável A em N, se $A \rightarrow^{*} \varepsilon$. Se isso ocorrer diz-se que a variável A é anulável. Pode-se, assim, substituir cada produção da forma $B::=X_1X_2X_3...X_n$ por todas as produções formadas pela retirada de uma ou mais variáveis X_i anuláveis.

Exemplo:

Dada a gramática	Neste caso as variáveis anuláveis são B e C, assim, a gramática pode ser transformada na seguinte gramática:
A ::= BCDe	A ::= BCDe | CDe | BDe | De
B ::= ε | e	B ::= e
C ::= ε | a	C ::= a
D ::= b | cC	D ::= b | c | cC

Os não-terminais que derivam a cadeia ε são chamados ε-não-teminais. Um não-terminal (variável) A é um ε-não-terminal se existir uma derivação $A \rightarrow^{*} \varepsilon$ em G. Note que ε está em L(G) se e somente se S é um ε-não-terminal. Se G não tem ε-não-terminal, ela é dita ε-livre.

Se G tem ε-produções, então em uma derivação sentencial da forma:

$$S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow ... \rightarrow \alpha_n$$

Os tamanhos das sentenças irão variar não-monotonicamente um em relação ao outro, isto é, ora aumentam, ora diminui. Entretanto, se G não tem ε-produções, então:

$$|S| \leq |\alpha_1| \leq |\alpha_2| \leq ... \leq |\alpha_n|$$

Isto é, os tamanhos das sentenças são monotonicamente crescentes. Esta propriedade é útil quando se quer testar se uma dada palavra é ou não gerada por uma GLC.

Antes de apresentar um algoritmo para eliminar as ε-produções, será apresentado um algoritmo para encontrar os ε-não-terminais.

Algoritmo: Encontrar o Conjunto dos ε-não-terminais
Entrada: Uma GLC G = (N, T, P, S)
Saída: O conjunto E de ε-não-terminais

E := {A | A∈N e A ::= ε};
Repita
    Q := {X | X∈N e X∉E e existe pelo menos uma produção da forma X ::= Y_1 Y_2...Y_n tal que Y_1∈E, Y_2∈E, ...,Y_n∈E};
    E := E∪Q;
Até Q = ∅;

Exemplo: Seja G a seguinte gramática:

S ::= aS | SS | bA
A ::= BB
B ::= CC | ab | aAbC
C ::= ε

Inicialmente temos E = {C} e se obtem:

B ::= ~~CC~~ | aAb~~C~~
~~C~~ ::= ε

Quando o laço é executado pela primeira vez deixa Q = {B} e

A ::= ~~BB~~
~~B~~ ::= ~~CC~~ | ab | aAb~~C~~
~~C~~ ::= ε

obtendo E = {B, C}

Como Q não é vazio, uma segunda iteração deixa Q = {A}

S ::= b~~A~~
~~A~~ ::= ~~BB~~
~~B~~ ::= ~~CC~~ | ab | a~~A~~b~~C~~
~~C~~ ::= ε

obtendo E = {A, B, C}.

Mais uma vez, Q é não-vazio, mas uma iteração subseqüente não acrescenta novos símbolos, assim o algoritmo termina. A, B e C são os únicos ε-não-terminais em G.

Para eliminar os ε-não-terminais de uma GLC, pode-se usar o seguinte algoritmo, que elimina ε-não-terminais sem introduzir novos. A estratégia é baseada na seguinte idéia. Seja A um ε-não-terminal em G. Então ele é dividido conceitualmente em dois não-terminais A' e A'', tal que A' gera todas as cadeias não vazias e A'' apenas gera ε. Agora, o lado direito de cada produção onde A aparece uma vez, por exemplo, B::=αAβ, é trocado por duas produções B::=αA'β e B::=αA''β. Já que A'' só gera ε,

ele pode ser trocado por ε, deixando B::=αβ. Depois disso, pode-se usar A em lugar de A'. A produção final fica: B::=αβ | B::=αAβ, onde A não é mais um ε-não-terminal. Se B::=αAβAγ, então são obtidas as quatro combinações possíveis pelas trocas de A por ε, e assim por diante.

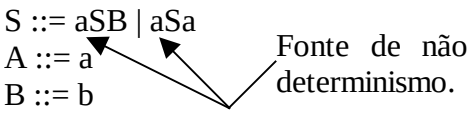
Algoritmo:**Eliminação de todos os ε-não-terminais**
Entrada: Uma GLC G = (N, T, P, S)
Saída: Uma GLC G' = (N', T, P', S') ε-livre

Construa o conjunto E;
P' := {p | p∈P e p não é ε-produção}
Repita
    Se P' tem uma produção da forma A::= αBβ, tal que B∈E, αβ∈$(N\cup T)^*$ e αβ ≠ ε,
    então inclua a produção A::= αβ em P';
até que nenhuma nova produção possa ser adicionada a P';

se S∈E então
    Adicione a P' as produções S'::=S | ε
    N' := N ∪ {S'};
senão
    S' := S;
    N' = N;
Fim-se;

3.3.2 Fatoração de GLC

Uma GLC está *fatorada* se ela é determinística, ou seja, se ela não possui produções para um mesmo não terminal no lado esquerdo cujo lado direito inicie com a mesma cadeia de símbolos ou com símbolos que derivam seqüências que iniciem com a mesma cadeia de símbolos. De acordo com esta definição, a seguinte gramática é não determinística:



Para fatorar uma GLC (retirar o não determinismo) deve-se alterar as produções envolvidas no não determinismo da seguinte maneira:

a) As produções com não determinismo direto da forma:

   A ::= αβ | αγ serão substituídas por:

   A :: αA'
   A' ::= β | γ

b) O não determinismo indireto é retirado através de sua transformação em determinismo direto (através de substituições sucessivas) e posterior eliminação segundo o item (a).

Exemplo: Fatorar a gramática

S ::= aSB | aSa		S ::= aSS’
A ::= b	Eliminando o não	S’::=B | a
B ::= a	determinismo direto	A ::= b
		B ::= a

Veja que na produção

S’ ::= B | a

tem um não-determinismo indireto, pois o lado direito da variável B começa com um **a**. Para fatorar esta gramática tem-se que tornar este não determinismo indireto em direto substituindo o lado direito de B ::= a na variável B da produção S’ ::= B. Assim procedendo tem-se:

S ::= aSS’		S ::= aSS’
S’::= a | a	Eliminando o não	S’::= aS’’
A ::= b	determinismo direto	S’’ ::= ε
B ::= a		A ::= b
		B ::= a

3.3.3 Eliminação da Recursão à Esquerda

Um não-terminal A, em uma GLC G = (N, T, P, S) é *recursivo* se $A \rightarrow^{+} \alpha A \beta$, para α e $\beta \in (N \cup T)^{*}$.

Se $\alpha = \varepsilon$, então A é *recursivo à esquerda*; se $\beta = \varepsilon$, então A é recursivo à direita. Esta recursividade pode ser direta ou indireta.

Uma gramática com pelo menos um não-terminal recursivo à esquerda ou à direita é uma *gramática recursiva à esquerda* ou *à direita*, respectivamente.

Uma GLC G = (N, T, P, S) possui *recursão à esquerda direta*, se P contém pelo menos uma produção da forma $A ::= A\alpha$.

Uma GLC G = (N, T, P, S) possui recursão à esquerda *indireta*, se existe em G uma derivação da forma: $A \rightarrow^{n} A\beta$, para algum $n \geq 2$.

Para eliminar as recursões diretas à esquerda de um não terminal A, com produções

$$A ::= A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

onde nenhum β_i começa por A, deve-se substituir estas produções-A pelas seguintes:

$$A ::= \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$$
$$A' ::= \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \varepsilon$$

Algoritmo: **Eliminação da recursão á esquerda**
Entrada: Uma GLC G = (N, T, P, S);
Saída: Uma GLC G' = (N', T, P', S) sem recursão à esquerda;
Ordene os não-terminais em uma ordem qualquer (ex: A_1, A_2, ...,A_n);
Para i de 1 a n faça
    Para j de 1 a i-1 faça
        Substitua as produções $A_i::=A_j\gamma$ por $A_i::=\delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$, onde $A_j::=\delta_1 \mid \delta_2 \mid ... \mid \delta_k$ são todas as produções-A_j correntes;
    fim_para;
    Elimine as recursões diretas à esquerda entre as produções A_i;
fim_para;

3.4 Tipos Especiais de GLC's

A seguir serão identificados alguns tipos especiais de GLC.

a) *Gramática própria*: Uma GLC é própria se:
   a.1) Não possui produções cíclicas; (ver definição a seguir)
   a.2) É ε-livre; e
   a.3) Não possui símbolos inúteis.

b) *Gramática sem ciclos*: G = (N, T, P, S) é uma GLC sem ciclos (ou livre de ciclos) se não existe em G nenhuma derivação da forma $A \rightarrow^+ A$, para todo $A \in N$.

c) *Gramática reduzida*: Uma GLC G = (N, T, P, S) é uma GLC reduzida se
   c.1) L(G) não é vazia;
   c.2) Se $A ::= \alpha \in P$ então $A \neq \alpha$
   c.3) G não possui símbolos inúteis.

d) *Gramática de operadores*: Uma GLC G = (N, T, P, S) é de operadores se ela não possui produções cujo lado direito contenha não terminais consecutivos.

e) *Gramática unicamente inversível*: Uma GLC reduzida é unicamente inversível se ela não possui produções com lados direitos iguais.

f) *Gramática linear*: Uma GLC G = (N, T, P, S) é linear se todas as suas produções forem da forma A ::= xBw | x, onde A e B pertencem a N, e x e w pertencem a T^*.

g) *Forma normal de Chomsky*: Uma GLC está na forma normal de Chomsky se ela é ε-livre e todas as suas produções (exceto, possivelmente, S ::= ε) são da forma:
   g.1) A ::= BC, com a, b e percencentes a N, ou
   g.2) A ::= a, com $A \in N$ e $a \in T$.

h) *Forma normal de Greibach*: Uma GLC está na forma normal de Greibach se ela é ε-livre e todas as suas produções (exceto, possivelmente, S ::= ε) são da forma $A ::= a\alpha$ tal que $a \in T$, $\alpha \in N^*$ e $A \in N$.

3.5 Principais Notações de GLC

A BNF (Backus Naur Form) é uma notação utilizada na especificação formal de sintaxe de linguagens de programação. Esta é a forma de especificação de gramáticas que tem sido utilizada nesse texto. A gramática a seguir é um exemplo de gramática descrita na notação BNF:

1. <S> ::= a<S> | ε
2. <E> ::= <E>+id | id

A BNFE (BNF Extended) é equivalente à BNF, mas permite uma especificação mais compacta da sintaxe de uma linguagem de programação. Os símbolos entre chaves abreviam os fechos das cadeias que eles representam. A mesma gramática descrita acima em BNF pode ser representada em BNFE da seguinte forma:

1. <S> ::= {a}
2. <E> ::= id {+id}

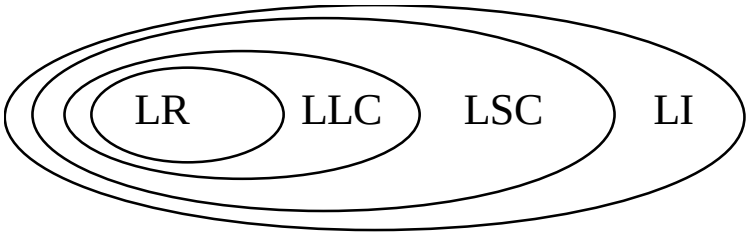
4. Autômato de Pilha

4.1 Introdução

Tipos de linguagens

Tipo 0 Linguagens Irrestritas
Tipo 1 Linguagens Sensíveis ao Contexto
Tipo 2 Linguagens Livres de Contexto
Tipo 3 Linguagens Regulares

Hierarquia de Chomsky



Reconhecedores decidem se uma cadeia w pertence ou não a uma dada linguagem L(G).

$$w \in L(G)? = \begin{cases} sim \\ não \end{cases}$$

Exemplo: L= $\{a^i b^i \mid i>0\}$

Cadeias com números iguais de a's e b's e os a's e b' são consecutivos

ab ∈ L?, aabb ∈ L?, aaabbb ∈ L?, aaab ∈ L?

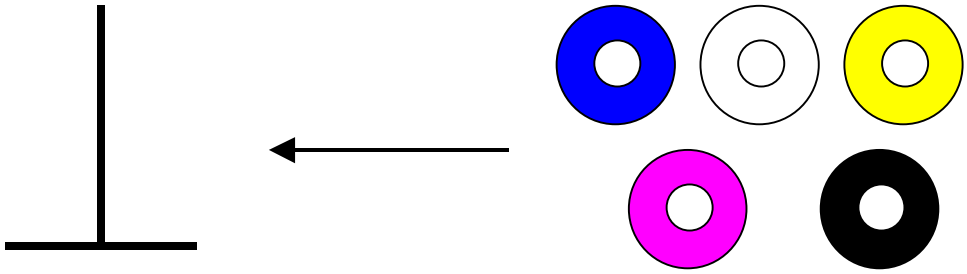
Reconhecedores x Linguagens:

Linguagem	**Reconhecedor**
Irrestrita Sensível ao Contexto	Máquina de Turing
Livre de contexto	**Autômato de Pilha**
Regular	Autômato finito

4.2 Conceito Intuitivo

Suponha L uma LLC sobre o alfabeto Σ. Verificar se uma cadeia w pertence a L.

Uso de Pinos e Discos



Exemplo: $L= \{vcv^R \mid v \in (a,b)^*$, v^R é a cadeia reversa de v$\}$

Assim $\Sigma = \{a, b, c\}$. Queremos saber: abaacaaba $\in$ L? , abaca $\in$ L?

Usaremos uma pilha e dois discos: preto e branco. Para cada símbolo de entrada procederemos da seguinte forma:

- **Etapa 1**: Se achar **a** empilhe um disco preto, e se achar **b**, um disco branco.
- **Etapa 2**: Se achar **c** mude de atitude, e prepare-se para desempilhar discos.
- **Etapa 3**: Compare o símbolo de entrada com o que está no topo da pilha:
  - Se for **a** e o topo da pilha é ocupado por um disco preto desempilhe o disco; ou
  - Se for **b** e o topo da pilha é ocupado por um disco branco, desempilhe o disco.

Se em alguma situação nenhuma das etapas acima puder ser aplicada, então pare: $w \notin L$.

Verificar se w = abcba pertence a L? Sim

Símbolo de entrada	**Faz**	**Pilha**	**Resta na entrada**
a	Empilha disco preto	●	bcba
b	Empilha disco branco	●○	cba
c	Muda de atitude	●○	ba
b	Compara e desempilha	●	a
a	Compara e desempilha		

Verificar se w = cab pertence a L? Não

Símbolo de entrada	**Faz**	**Pilha**	**Resta na entrada**
c	Muda de atitude		ab
a	Compara e desempilha		ab

Verificar se w = aabcba pertence a L? Não

Símbolo de entrada	Faz	Pilha	Resta na entrada
a	Empilha disco preto	●	abcba
a	Empilha disco preto	●●	bcba
b	Empilha disco branco	●●○	cba
c	Muda de atitude	●●○	ba
b	Compara e desempilha	●●	a
a	Compara e desempilha	●	

Conclusão: A entrada deverá se aceita se, e somente se, ao final da execução do processamento ela foi totalmente consumida e a pilha está vazia

4.3 Definição Formal

Um autômato de pilha não determinístico M é definido como:

$$M = (\Sigma, \Gamma, Q, q_1, F, \delta)$$

onde:

Σ : Alfabeto de entrada
Γ: Alfabeto da pilha
Q: Conjunto de estados: $(q_1, q_2, .., q_n\}$
q_1: Estado inicial: $q_1 \in Q$
F: Conjunto de estados finais: $F \subseteq Q$
δ: Função de transição: $\delta = Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow P_f(Q \times \Gamma^*)$

Entendendo a função de transição:

Seja M um AP e suponha que: $q \in Q$, $\sigma \in \Sigma \cap \{\varepsilon\}$ e $\gamma \in \Gamma \cap \{\varepsilon\}$

Dados $p \in Q$ e $u \in \Gamma^*$, a transição $(q, \sigma, \gamma) \rightarrow (p,u)$ significa que:

- M muda para o estado p.
- M troca γ por u no topo da pilha.

Casos onde aparecem o ε

Casos		Interpretação
$\sigma = \varepsilon$,	$(q, \varepsilon, \gamma) \rightarrow (p,u)$	A entrada não é consultada
$\gamma = \varepsilon$,	$(q, \sigma, \varepsilon) \rightarrow (p,u)$	A topo da pilha não é consultado
$\sigma = \gamma = \varepsilon$,	$(q, \varepsilon, \varepsilon) \rightarrow (p,u)$	Nem a entrada, nem o topo da pilha são consultados.
$\gamma \neq \varepsilon$ e $u = \varepsilon$,	$(q, \sigma, \gamma) \rightarrow (p, \varepsilon)$	Remove γ da pilha
$\gamma = \varepsilon$ e $u \neq \varepsilon$,	$(q, \sigma, \varepsilon) \rightarrow (p, u)$	Empilha u
$\gamma = \varepsilon$ e $u = \varepsilon$,	$(q, \sigma, \varepsilon) \rightarrow (p, \varepsilon)$	Não altera a pilha

Exemplo1: Criar um Autômato de Pilha M_1 para

$L_1 = \{vcv^R \mid v \in (a,b)^*$, v^R é a cadeia reversa de v$\}$

Σ : {a, b, c}
Γ: {a, b}

δ: função de transição:

$$\left.\begin{array}{ll}\delta(q_1,a,\varepsilon)=\{(q_1,a)\} & \\ \delta(q_1,b,\varepsilon)=\{(q_1,b)\} & \end{array}\right\rangle \textit{Empilha}$$

$$\delta(q_1,c,\varepsilon)=\{(q_2,\varepsilon)\} \quad \rangle \textit{Pilha permanece inalterada}$$

$$\left.\begin{array}{ll}\delta(q_2,a,a)=\{(q_2,\varepsilon)\} & \\ \delta(q_2,b,b)=\{(q_2,\varepsilon)\} & \end{array}\right\rangle \textit{Desempilha}$$

$Q = \{q_1, q_2\}$, $q_1 = q_1$ e $F = \{q_2\}$

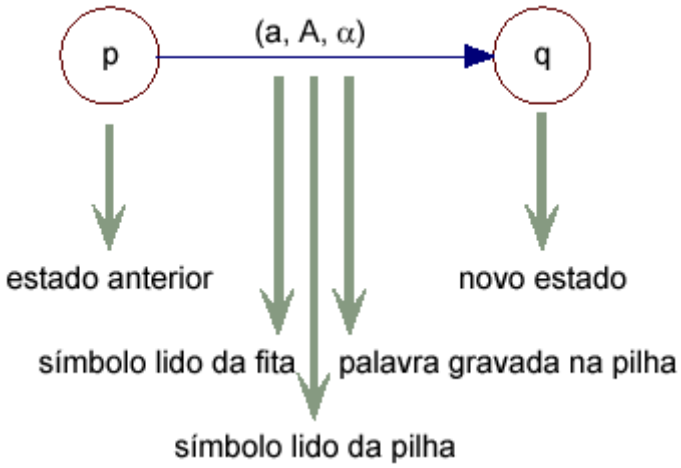
A função de transição pode ser colocada em forma de tabela.

Estado	Entrada	Topo da pilha	Transições
a	σ	γ	(p_1, u_1) ... (p_s, u_s)

Para o autômato M_1 acima

Estado	Entrada	Topo da pilha	Transições	Descrição
q_1	a	ε	(q_1, a)	Acha **a** e o empilha.
q_1	b	ε	(q_1, b)	Acha **b** e o empilha.
q_1	c	ε	$(q_2, ε)$	Acha **c** e muda de estado.
q_2	a	a	$(q_2, ε)$	Acha **a** na entrada e na pilha e o desempilha
q_2	b	b	$(q_2, ε)$	Acha **b** na entrada e na pilha e o desempilha

Forma gráfica de uma transição



Exemplo2: Criar um Autômato de Pilha M_2 para

$L_2 = \{vv^R \mid v \in (a,b)^*$, v^R é a cadeia reversa de v$\}$

Σ : {a, b}
Γ: {a, b}
δ: função de transição:

Estado	Entrada	Topo da pilha	Transições	Descrição
q_1	a	ε	(q_1, a)	Acha **a** e o empilha.
q_1	b	ε	(q_1, b)	Acha **b** e o empilha.
q_1	ε	ε	$(q_2, ε)$	Muda de estado.
q_2	a	a	$(q_2, ε)$	Acha **a** na entrada e no topo e o desempilha
q_2	b	b	$(q_2, ε)$	Acha **b** na entrada e no topo e o desempilha

$Q = \{q_1, q_2\}$, $q_1 = q_1$ e $F = \{q_2\}$

4.4 Exemplos e construções de AP

Exemplo 1: $L = \{a^i b^i \mid i \geq 0\}$

Procedimento com um disco:

- **Etapa 1**: Se achar um **a** na entrada, ponha um disco na pilha
- **Etapa 2**: Se achar um **b** mude de atitude e passe a comparar a entrada com a pilha, removendo um disco da pilha para cada **b** que achar na entrada (incluindo o primeiro!)

$M = (\{a,b\}, \{a\}, \{q_1, q_2\}, q_1, \{q_1, q_2\}, δ)$

Onde δ está na tabela

Estado	Entrada	Topo da pilha	Transições	Descrição
q_1	a	ε	(q_1, a)	Empilha **a**.
q_1	b	a	$(q_2, ε)$	Desempilha **a** e muda de estado.
q_2	b	a	$(q_2, ε)$	Desempilha **a**.

Exemplo 2: $L = \{w \in \{a, b\} \mid$ o número de a's é igual ao número de b's$\}$

Procedimento:

- **Etapa 1**: Empilhar **a** para cada **a** encontrado e **b** para cada **b** encontrado na entrada.
- **Etapa 2**: Empilhar um "**b** sobre um **a**" tem o efeito de desempilhar o **b** da pilha e vice-versa.

Tabela para δ

Estado	Entrada	Topo da pilha	Transições	Descrição
q_1	a	a	(q_1, aa)	Acha **a** e o empilha.
q_1	b	b	(q_1, bb)	Acha **b** e o empilha.
q_1	a	b	(q_1, ε)	Desempilha um **b**.
q_1	b	a	(q_1, ε)	Desempilha um **a**.

Veja que esta tabela não está bem especificada, pois em qualquer entrada (linha) da mesma a pilha tem que possuir um símbolo. E, inicialmente, não há símbolo na pilha. Se apenas adicionarmos as entradas:

q_1	a	ε	(q_1, a)	Empilha **a**.
q_1	b	ε	(q_1, b)	Empilha **b**

o autômato ficará errado.

A saída é inventar um novo símbolo β para o alfabeto da pilha. Este símbolo é acrescentado à pilha ainda vazia, no início da computação. Daí em diante, o autômato opera apenas com a's e b's na pilha. Assim, ao avistar um β no topo da pilha o autômato sabe que a pilha não contém mais nenhum a ou b. Para garantir que β só vai ser usado uma vez, convém reservar ao estado inicial apenas a ação de marcar o fundo da pilha. Portanto, se q_1 for o estado inicial teremos

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

Com isto se, ao final da computação, a pilha contém apenas o marcador β então a palavra está em L e é aceita; do contrário a palavra é rejeitada. Entretanto, pela definição formal o autômato só pode aceitar a palavra se a pilha estiver completamente vazia. Isto sugere que precisamos de mais uma transição para remover β do fundo da pilha.

Assim surge um novo problema. Se a palavra está em L, então será totalmente consumida deixando na pilha apenas β. Portanto, a transição que remove β ao final desta computação, e esvazia completamente a pilha, não tem nenhuma entrada para consultar. O problema é que se permitimos ao autômato remover sem consultar a entrada, ele pode realizar este movimento no momento errado, antes que a entrada tenha sido consumida. Como os nossos autômatos são não determinísticos, isto não apresenta nenhum problema conceitual.

Infelizmente o fato de ainda haver símbolos na entrada abre a possibilidade do autômato continuar a computação depois de ter retirado o marcador do fundo. Para evitar isto, o autômato deve, ao remover β, passar a um novo estado q_3 a partir do qual não há transições. Assim, se o autômato decidir remover o marcador antes da entrada ser totalmente consumida ele será obrigado a parar, e não poderá aceitar a entrada nesta computação. Como a pilha só vai poder se esvaziar em q_3 , é claro que este será o único estado final deste autômato.

Resumindo, temos um autômato que tem {a, b} como alfabeto de entrada; {a, b, β} como alfabeto da pilha; estados q_1 , q_2 e q_3 ; estado inicial q_1 ; estado final q_3 ; e cuja função de transição é definida na tabela abaixo.

Estado	Entrada	Topo da pilha	Transições	Descrição
q_1	ε	ε	(q_2, β)	Marca o fundo da pilha
q_2	a	β	$(q_2, a\beta)$	Acha **a** e empilha um **a**
q_2	b	β	$(q_2, b\beta)$	Acha **b** e empilha um **b**
q_2	a	a	(q_2, aa)	Acha **a** e empilha um **a**
q_2	b	b	(q_2, bb)	Acha **b** e empilha um **b**
q_2	b	a	(q_2, ε)	Desempilha um **a**
q_2	a	b	(q_2, ε)	Desempilha um **b**
q_2	ε	β	(q_3, ε)	Esvazia a pilha

4.5 Exercícios

1) Considere o autômato de pilha não determinístico M com alfabeto $\Sigma = \{a, b\}$ e $\Gamma = \{a\}$, estados q_1 e q_2, estado inicial q_1 e final q_2 e transições dadas pela tabela:

Estrado	Entrada	Topo da pilha	transições
q_1	a	ε	(q_1, a) (q_2, ε)
q_1	b	ε	(q_1, a)
q_2	a	a	(q_2, ε)
q_2	b	a	(q_2, ε)

a) Descreva todas as possíveis seqüências de transições de M na entrada *aba*.
b) Mostre que *aba*, *aa* e *abb* não pertencem a L(M) e que *baa*, *bab* e *baaaa* pertencem a L(M).

2) Ache um autômato de pilha não determinístico cuja linguagem aceita é L onde:
a) $L=\{a^n b^{n+1} | \ n \geq 0\}$;
b) $L=\{a^n b^{2n} | \ n \geq 0\}$;
c) $L=\{a^n b^m c^n | \ m,n \geq 1\}$;
d) $L=\{a^n b^m a^{n+m} | \ m,n \geq 1\}$;

3) Considere a linguagem dos parênteses balanceados.
a) Dê exemplo de uma gramática livre de contexto que gere esta linguagem.
b) Dê exemplo de um autômato de pilha não determinístico que aceita esta linguagem.

4.6 Gramáticas Livres de Contexto e Autômatos de Pilha

Nosso objetivo nesta seção é a demonstração do seguinte resultado fundamental: *Uma linguagem é livre de contexto se, e somente se, é aceita por algum autômato de pilha não determinístico*. Como uma linguagem é livre de contexto se for gerada por uma gramática livre de contexto, o que precisamos fazer é estabelecer um elo entre gramáticas livres de contexto e autômatos de pilha. Neste caso, iremos elaborar uma receita que, dada uma gramática livre de contexto G = (N, T, P, S), construiremos um autômato de pilha não determinístico M que aceita L(G). Não desenvolveremos a recíproca, isto é, dado M um autômato de pilha não determinístico, construir G a partir de M.

Seja G uma gramática livre de contexto. Nosso objetivo consiste em construir um autômato de pilha cujas computações simulem as derivações mais à esquerda em G . É claro que a pilha tem que desempenhar um papel fundamental nesta simulação. O que de fato acontece é que o papel dos estados é secundário, e a simulação se dá na pilha.

Seja *G = (N, T, P, S)* uma gramática livre de contexto e $M = (\Sigma, \Gamma, Q, q_1, F, \delta)$ o autômato de pilha a ser construído a partir de G . Como M deve aceitar L(G), isto é GERA(G)=ACEITA(M), é claro que seu alfabeto de entrada Σ tem que ser igual a T. Continuando, temos:

$\Sigma = T$	(alfabeto de entrada é igual conjunto de terminais de G)
$q_1 = i$	(estado inicial i)
$\Gamma = T \cup N$	(alfabeto de pilha é a união de terminasi e não terminais de G)

Cada derivação de G corresponde a uma transição de M. Entretanto, como a derivação de palavras de L(G) é feita a partir do símbolo inicial S, o autômato deve começar pondo este símbolo no fundo da pilha. Observe que o autômato não pode consumir entrada ao marcar o fundo da pilha, já que precisamos da entrada para guiá-lo na busca da derivação correta. Nem adianta consultar a pilha, já que ainda está totalmente vazia. Entretanto, uma transição que não consulta a entrada nem a pilha pode ser executada em qualquer momento da computação. Por isso forçamos o autômato a mudar de estado depois desta transição, impedindo assim que volte a ser executada. Temos então que

$$\delta(i, \varepsilon, \varepsilon) = \{(f, S)\},$$

onde *f*, diferente de *i*, é um estado do autômato.

Daí em diante toda a ação vai se processar na pilha, e podemos simular a derivação sem nenhuma mudança de estado adicional. Portanto, f será o estado final de M . Assim, à regra X ::= α de G fazemos corresponder a transição

(1) $$(f, \varepsilon, X) \in (f, \alpha)$$

de M . Contudo, a construção do autômato ainda não está completa. O problema é que M só pode aplicar uma transição como (1) se a variável X estiver no topo da pilha. Infelizmente isto nem sempre acontece, como ilustra o exemplo a seguir.

Exemplo
Suponhamos G1 = (N, T, P, S) com

T = {a, b, c}
N = { S }
S = S
P = { S ::= Sc | aSb | ε}.

Logo, de acordo com as discussões acima, $M1 = (\Sigma, \Gamma, Q, q_1, F, \delta)$, deve ter

Σ = {a, b, c}
Γ = {a, b, c, S}
Q = {i, f}
q_1 = i
F = {f}

E δ está representado na tabela abaixo

Estado	Entrada	Topo da pilha	Transição
i	ε	ε	(f, S)
f	ε	S	(f, Sc) (f, aSb) (f, ε)

A palavra abc^2 tem derivação mais à esquerda

$$S \to Sc \to Sc^2 \to aSbc^2 \to abc^2 \text{ em G1}$$

Agora, deve-se fazer uma computação de M1 que copie na pilha esta derivação abc^2. A computação deve começar marcando o fundo da pilha com S e deve prosseguir, a partir daí, executando, uma a uma, as transições de M1 que correspondem às regras aplicadas na derivação acima. Isto nos dá

(2) $(i, abc^2, \varepsilon) \vdash (f, abc^2, S) \vdash (f, abc^2, Sc) \vdash (f, abc^2, Sc^2) \vdash (f, abc^2, aSbc^2)$.

Até aqui tudo bem, mas a transição seguinte deveria substituir S por ε. Só que para isto ser possível a variável S tem que estar no topo da pilha, o que não acontece neste caso. Observe, contudo, que o *a* que apareceu na pilha acima do S na última configuração de (2) corresponde ao primeiro símbolo da palavra de entrada. Além disso, como se trata de uma derivação mais à esquerda, este símbolo não será mais alterado. Concluímos que, como o primeiro símbolo da palavra já foi corretamente derivado, podemos 'esquecê-lo' e partir para o símbolo seguinte. Para implementar isto no autômato basta apagar da pilha os terminais que precedem a variável mais à esquerda da pilha *e que já foram corretamente construído*s. Isto significa acrescentar à tabela acima transições que permitam apagar terminais que apareçam simultaneamente na entrada e na pilha:

Estado	Entrada	Topo da pilha	Transição
f	a	a	(f, ε)
f	b	b	(f, ε)
f	c	c	(f, ε)

Levando isto em conta a computação acima continua, a partir da última configuração de (2) da seguinte maneira

(3) (f, abc^2, aSbc^2) ⊢ (f, bc^2, Sbc^2) ⊢ (f, bc^2, bc^2) ⊢ (f, c^2, c^2) ⊢ (f, c, c) ⊢ (f, ε, ε).

Observe que a passagem da segunda para a terceira configuração em (3) corresponde à regra S ::= ε. Todas as outras etapas são aplicadas das transições da segunda tabela.

4.7 Construção de AP Através de GLC

Podemos agora descrever de maneira sistemática a receita usada para construir um autômato de pilha não determinístico M cuja linguagem aceita é L(G). Se a gramática livre de contexto G tem por elementos (N, T, P, S), então o autômato ficará completamente determinado pelos seguintes os elementos

- o alfabeto de entrada T;
- o alfabeto da pilha T∪N;
- o conjunto de estados {i, f};
- o estado inicial i;
- o conjunto de estados finais {f};
- a função de transição

$$\delta : \{i, f\} \times (T \cup \{\varepsilon\}) \times (T \cup N \cup \{\varepsilon\}) \to \{i, f\} \times (T \cup N)^*$$

que é definida por

$$\begin{cases} \delta(i, \varepsilon, \varepsilon) = (f, S) \\ \text{Para toda regra } X ::= \alpha \text{ de G fazer:} \\ \quad \delta(f, \varepsilon, X) = (f, \alpha) \text{ onde } X \in N \text{ e } \alpha \in (N \cup T)^* \\ \text{Para todo terminal } a \text{ de G fazer:} \\ \quad \delta(f, a, a) = (f, \varepsilon) \text{ onde } a \in T \end{cases}$$

Este autômato executa dois tipos diferentes de transição a partir do estado final *f*.

- *Transições de substituições*: substituem uma variável X no topo da pilha por α (pertence a T∪N*) quando X ::= α é uma regra de G.
- *Transições de remoção*: removem terminais que aparecem casados na pilha e na entrada.

Exemplo
Considere a gramática G2 cujas regras são

E ::= E + T | T
T ::= T * F | F
E ::= (E) | id

Assim, $M2 = (\Sigma, \Gamma, Q, q_1, F, \delta)$, deve ter

Σ = {id, +, *, (,)}
Γ = { id, +, *, (,), E, T, F}
Q = {i, f}

$q_1 = i$
$F = \{f\}$

E a função de transição é definida pela tabela

Estado	Entrada	Topo da pilha	Transição
i	ε	ε	(f, E)
f	ε	E	(f, E+T) (f, T)
f	ε	T	(f, T*F) (f, F)
f	ε	F	(f,(E)) (f, id)
f	(	(	(f, ε)
f	)	)	(f, ε)
f	+	+	(f, ε)
f	*	*	(f, ε)
f	id	id	(f, ε)

Uma computação para a cadeia id + id * id

(i, id+id*id, ε)⊢ (f, id+id*id, E)⊢ (f, id+id*id, E+T)⊢ (f, id+id*id, T+T)⊢
(f, id+id*id, F+T)⊢ (f, id+id*id, id+T)⊢ (f, +id*id, +T)⊢ (f, id*id, T)⊢ (f, id*id, T*F)⊢
(f, id*id, F*F) ⊢ (f, id*id, id*F)⊢ (f, *id, *F)⊢ (f, id, F)⊢ (f, id, id)⊢ (f, ε, ε).

5. Máquina de Turing

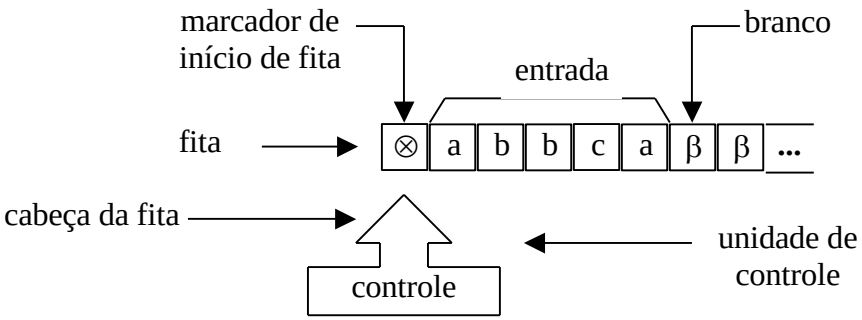
A máquina de Turing, proposta por Alan Turing em 1936, é universalmente conhecida e aceita como formalização de algoritmo. Trata-se de um mecanismo simples que formaliza a idéia de uma pessoa que realiza cálculos. Lembra, em muito, os computadores atuais, embora tenha sido proposta anos antes do primeiro computador digital. Apesar de sua simplicidade, o modelo Máquina de Turing possui, no mínimo, o mesmo poder computacional de qualquer computador geral.

A máquina de Turing pode ser vista como uma máquina constituída de três partes, como segue:

a) *Fita*. Usada simultaneamente como dispositivo de entrada, de saída e de 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 para a direita.
c) *Programa* ou *função de transição*. Função que define o estado da máquina e comanda as leituras, as gravações e o sentido de movimento da cabeça.

A fita é finita à esquerda e infinita (tão grande quanto necessário) à direita, sendo dividida em células, cada uma armazenando 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 abaixo, onde β 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 gravação é realizada na mesma célula de leitura), a cabeça move uma célula para a direita ou para a 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.

5.1 Modelo Formal

Uma *máquina de Turing* é uma 8-upla:

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

Onde:

Σ: alfabeto de símbolos de entrada;
Q: conjunto de estados possíveis da máquina, o qual é finito;
δ: programa ou função de transição:

$$\delta : Q \times (\Sigma \cup V \cup \{\beta, \otimes\}) \rightarrow Q \times (\Sigma \cup V \cup \{\beta, \otimes\}) \times \{E, D\}$$

q_0: estado inicial da máquina, tal que $q_0 \in Q$;
F: conjunto de estados finais, tal que $F \subset Q$;
V: alfabeto auxiliar;
β: símbolo especial *branco*
⊗: símbolo especial *marcador de início* ou *símbolo de início* da fita.

O símbolo de início da fita ocorre exatamente uma vez e sempre 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	*Para determinar*
• estado corrente • símbolo lido da fita	• novo estado • símbolo a ser gravado • sentido de movimento da cabeça, onde esquerda e direita são representados por E e D, respectivamente.

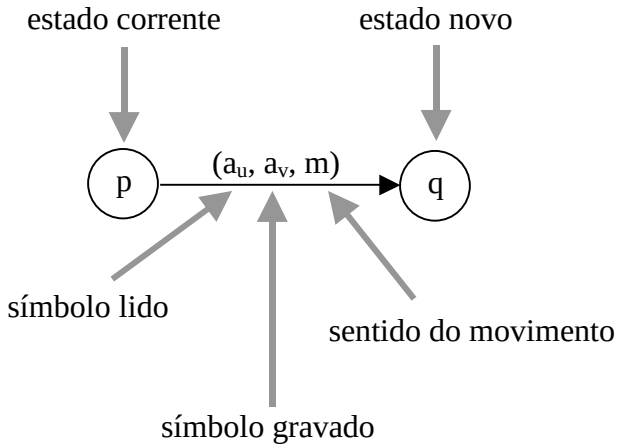
Assim, tem-se que:

δ(estadoCorrente, símboloLido) = (novoEstado, símboloGravado,sentidoDeMovimento)

ou seja, suponha que p, q $\in$ Q, a_u, $a_v \in (\Sigma \cup V \cup \{\beta, \otimes\})$ e m$\in${E, D}):

$$\delta(p, a_u) = (q, a_v, m)$$

A função programa pode ser interpretada em um diagrama de transição como é mostrado na figura abaixo.



Neste caso, os estados iniciais e finais são representados como:

A função programa pode ser representada na forma de tabela, veja a seguir. A tabela ilustra o caso para $\delta(p, a_u) = (q, a_v, m)$:

δ	⊗	...	$\mathbf{a_u}$	...	$\mathbf{a_v}$	...	β
p			(q, a_v, m)				
q							
...							

O processamento de uma máquina de Turing M = (Σ, Q, δ, q_0, F, V, β, ⊗) 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) *Estado final*. A máquina assume um estado final: a máquina pára e a palavra de entrada é aceita.
b) *Função indefinida*. A função programa é indefinida para o argumento (símbolo lido e estado corrente): a máquina pára e a palavra de entrada é rejeitada.
c) *Movimento inválido*. 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.

Exemplo: Máquina de Turing M1 para reconhecer a linguagem

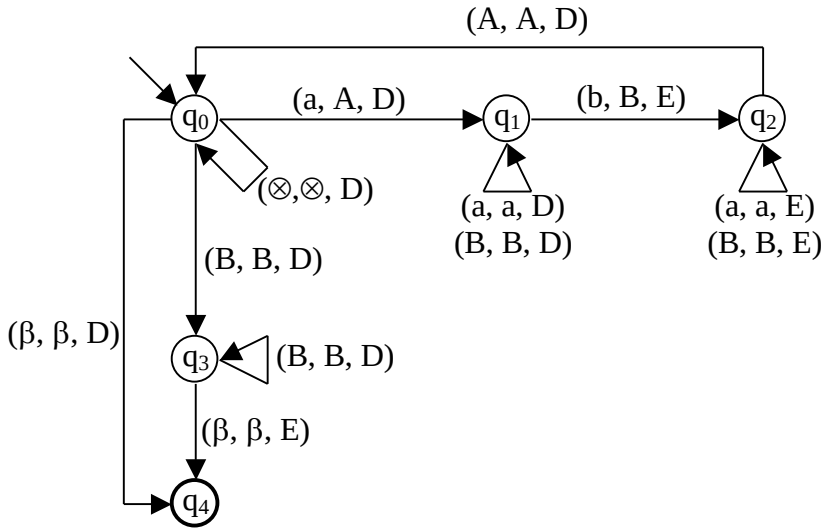
$$L = \{a^n b^n \mid n \geq 0\}$$

A máquina de Turing M1 = ({a,b}, {q_0, q_1, q_2, q_3, q_4}, δ, q_0, {q_4}, {A, B}, β,⊗) é ilustrada abaixo:

TABELA

δ	⊗	**a**	**b**	**A**	**B**	β
q_0	(q_0, ⊗, D)	(q_1, A, D)			(q_3, B, D)	(q_4, β, D)
q_1		(q_1, a, D)	(q_2, B, E)		(q_1, B, D)	
q_2		(q_2, a, E)		(q_2, A, D)	(q_2, B, E)	
q_3					(q_3, B, D)	(q_4, β, E)
q_4						

DIAGRAMA DE TRANSIÇÕES



O algoritmo apresentado reconhece o primeiro símbolo *a*, o qual é marcado como A, e movimenta a cabeça da fita para a 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*. Adicionalmente, o algoritmo garante que qualquer outra palavra que

não esteja na forma nab^n é rejeitada. Note-se que o símbolo de início de fita não tem influência na soução proposta.

A seguir é apresentada uma computação da máquina de Turing M1 para a entrada *w=aabb*.

COMPUTAÇÃO para a cadeia *aabb*.