APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Proposta de Trabalho

Introdução aos autômatos finitos e às linguagens formais para descrição algébrica das máquinas de Turing.

Índice

Autômatos Finitos e Linguagens Formais

Os conceitos centrais da teoria de autômatos

As definições mais importantes que permeiam os termos de autômatos são:

a) alfabeto → um conjunto de símbolos;

b) string → uma lista de símbolos de um alfabeto;

c) linguagem → um conjunto de string de um mesmo alfabeto.

Alfabeto

Um alfabeto é um conjunto de símbolos finitos e não-vazio. Convencionalmente utiliza-se ∑ para um alfabeto.

1. ∑ = {0,1} alfabeto binário
2. ∑ = {a,b,c, ... , z} o conjunto de todas as letras minúsculas

Strings

Um string (ou as vezes palavra ou também cadeia) é uma sequência finita de símbolos de algum alfabeto.

Notas importantes sobre os strings:

a. O string vazio (ε) é o string com zero ocorrências de símbolos. Esse string pode ser escolhido de qualquer alfabeto;

b. O comprimento de um string é o número de símbolos no string;

- String: 01101
  - Existem dois símbolos {0,1} neste string, mas existem cinco posições para símbolos.
  - | ε | = 0
  - |01101| = 5

c. Se ∑ é um alfabeto, pode-se expressar o conjunto de todos os strings de um certo comprimento a partir deste alfabeto, utilizando uma notação exponencial, ou seja, $\sum^k$

$\sum^k \equiv$ conjunto de strings de comprimento ***k***

Se ∑ = {0,1} então,

- $\sum^0 \equiv$ { ε }
- $\sum^1 \equiv$ { 0,1 }
- $\sum^2 \equiv$ { 00,01,10,11 }

- $\sum^3 \equiv$ { 000,001,010,011,100,101,110,111 }
- ... etc

Nota Importante:

$\sum \equiv$ alfabeto com elementos 0 e 1

$\sum^1 \equiv$ conjunto de string 0 e 1, cada um dos quais com comprimento 1

d. Convenção: comumente se utiliza letras minúsculas do início do alfabeto (ou dígitos) para denotar símbolos, e letras minúsculas próximas ao fim do alfabeto, em geral w, x, y e z, para denotar strings.

$\sum = \{a,b\}$

w = aaba, x = aabbb e y = aabaa

e. O conjunto de todos os strings sobre um alfabeto $\sum$ é denotado convencionalmente por $\sum^*$.

Por exemplo:

$\{0,1\}^* = \{ \varepsilon,0,1,00,01,10,11,000, ...\}$. Como gerá-lo matematicamente?

$\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup ...$

$$\sum^* = \bigcup_{i=0}^{\infty} \sum^i \text{ (fechamento de Kleeme)}$$

f. Às vezes é desejado excluir o string vazio do conjunto de strings. Assim o conjunto de strings não vazio do alfabeto $\sum$ é denotado por $\sum^+$, valendo as seguintes propriedades:

$$\sum^+ = \sum^1 \cup \sum^2 \cup ... = \bigcup_{i=1}^{\infty} \sum^i$$

$\sum^* = \sum^+ \cup \{\varepsilon\}$

g. Concatenação de strings:

Sejam os strings x e y. Então, xy denota a concatenação de x e y, isto é, o string formado tomando-se uma cópia de x e acrescentando-se a ela uma cópia de y.

Se $x = a_1a_2...a_i$ e $y=b_1b_2...b_j$, então xy é o string de comprimento i+j: $xy = a_1a_2...a_i\ b_1b_2...b_j$

h. O string vazio (ε) é a identidade para a concatenação, isto é, εw = wε = w

Linguagens

Um conjunto de strings, todos escolhidos a partir de algum ∑*, onde ∑ é um alfabeto específico, e L ⊆ ∑*, então L é uma linguagem sobre ∑.

Exemplo de Linguagens:

1. A linguagem de todos os strings que consistem em ***n*** zeros seguidos por ***n*** uns, para algum n ≥ 0:

   { ε, 01, 0011, 000111...}

2. O conjunto de strings de zeros e uns com um número igual de cada um deles:

   {ε, 01, 10, 0011, 0101, 1001, 001011, ...}

3. O conjunto de números binários cujo valor é um número primo:

   { 10, 11, 101, 111, 1011, ...}
   2 3 5 7 11

4. ∑* é uma linguagem para qualquer alfabeto ∑.
5. ∅, a linguagem vazia, é uma linguagem sobre qualquer alfabeto.
6. {ε}, a linguagem que consiste apenas no string vazio, também é uma linguagem sobre qualquer alfabeto. Note que ∅ ≠ {ε}.

Nota:

A única restrição importante sobre o que pode ser uma linguagem é que todos os alfabetos são finitos.

Problema

Na teoria de autômatos, um problema é a questão de decidir se um dado string é elemento de alguma linguagem específica.

Mais precisamente, se ∑ é um alfabeto e ***L*** é uma linguagem sobre ∑, então o problema ***L*** é:

- Dado um string ***w*** em ∑*, definir se ***w*** está ou não em ***L***.

É uma linguagem ou um problema?

Na realidade, linguagens e problemas são a mesma coisa. O termo que preferimos usar depende de nosso ponto de vista.

Pode-se demonstrar por redução ao absurdo que:

"Se o teste de pertinência a linguagem L_x é difícil, então compilar programas na linguagem de programação ***x*** é difícil".

Termos Fundamentais

Alfabeto: um alfabeto é qualquer conjunto finito não vazio de símbolos.

Strings: um string é uma sequência de comprimento finito de símbolos.

Linguagens e problemas: Uma linguagem é um (possivelmente infinito) conjunto de strings, os quais escolhem seus símbolos a partir de algum alfabeto único. Quando os strings de uma linguagem têm de ser interpretados de algum modo, a questão de saber se um string pertence à linguagem às vezes se denomina um problema.

Autômatos Finitos: Os autômatos finitos envolvem estados e transições entre estados em resposta a entradas sobre o grafo que compõe os estados. Um autômato finito tem um conjunto de estados, e seu "controle" se desloca de estado para estado em resposta a entradas "externas".

Linguagens Regulares: Essas linguagens são exatamente aquelas que podem ser descritas por autômatos finitos.

Expressões Regulares: Essas expressões formam uma notação estrutural para descrever os mesmos padrões que podem ser representados por autômatos finitos.

Gramáticas livres de contexto: essas gramáticas constituem uma notação importante para descrever a estrutura de linguagens de programação e conjuntos de strings inter-relacionados.

Maquina de Turing: Essas máquinas são autômatos mais complexos que modelam o poder dos computadores reais. Ela nos permite estudar as questões de decibilidade.

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Autômatos Finitos, Linguagens e Gramáticas

Índice

Figuras

Autômatos Finitos, Linguagens e Gramáticas

Definição de um Autômato Finito Determinístico

A = (Q, ∑, δ, q_0, F) tupla de cinco elementos

Q ≡ conjunto finito de estados

∑ ≡ conjunto finito de símbolos de entrada

δ ≡ uma função de transição que toma como argumentos um estado e um símbolo de entrada e retorna um estado.

q_0 ≡ um estado inicial, um dos estados em ***Q***

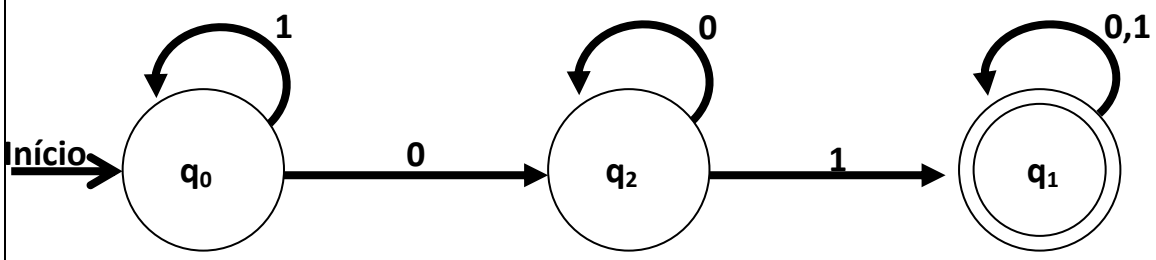
F ≡ um conjunto de estados finais ou de aceitação. ***F*** é um subconjunto de ***Q***

Como DFA Processa Strings

Suponha que $a_1a_2...a_n$ seja uma sequência de símbolos de entrada e que estão todos definidos pelo alfabeto ∑. A "linguagem" do DFA é um conjunto de todos os strings que o DFA aceita. Seja então, q_0 o estado inicial do DFA. Para processar esta sequência de símbolos é necessário consultar uma função de transição ***δ*** do tipo $\delta(q_{i-1}, a_i)=q_1$ para cada i = 0, ..., n. Se q_n é um elemento de ***F***, então a entrada $a_1a_2...a_n$ é "aceita" e, se não, ela é "rejeitada".

Exemplo: Para uma linguagem L = {w | w é formada por x01y para alguns strings que consistem somente em 0's e 1's}, tem-se:

A = ({q_0,q_1,q_2}, {0,1}, δ, q_0, {q_1})



δ	0	1
→ q_0	q_2	q_0
* q_1	q_1	q_1
q_2	q_2	q_1

→ estado inicial
* estado de aceitação

Figura 1 – diagrama de transições ou grafo de um DFA.

"O DFA define uma linguagem: o conjunto de todos os strings que resultam em uma sequência de transições de estado, desde o estado inicial até um estado de aceitação".

Para tornar esta a noção de "linguagem" de um DFA é necessário também definir uma "função de transição estendida" que descreve o que acontece quando se começa de qualquer estado e se segue por uma sequência de entradas.

Assim,

"Se δ é a função de transição, então $\hat{\delta}$ será a função de transição estendida".

A definição de $\hat{\delta}$ é feita por indução da seguinte forma:

Base: $\hat{\delta}(q, \varepsilon) = q$ "Se não lida nenhuma entrada, então permanece no mesmo estado q"

Indução: L = {w | w da forma ***xa***, onde ***a*** é o ultimo símbolo de w}

$\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x),a) = \delta(p,a)$ (q é o estado inicial)

Onde: p ≡ estado em que o autômato se encontra depois de processar tudo, exceto o último símbolo de w, ou seja a.

A linguagem de um DFA

Seja um DFA definido como $A = (Q, \sum, \delta, q_0, F)$, então a linguagem deste DFA será denotada por ***L(A)*** e definida na forma, $L(A) = \{w / \hat{\delta}(q_0, w)$ está em F$\}$.

$L(A) = \{w / \hat{\delta}(q_0, w)$ está em F$\}$

"A linguagem de A, L(A), é o conjunto de strings w que levam o estado inicial q_0, até um dos estados de aceitação".

"Se ***L*** é L(A) para algum DFA A, diz-se que ***L*** é uma linguagem regular".

Autômatos Finitos Não-Determinísticos – NFA

"Um autômato finito não-determinístico (NFA) tem o poder de estar em vários estados ao mesmo tempo. Essa habilidade é expressa com frequência como a capacidade de adivinhar algo sobre a sua entrada".

"Os NFAs aceitam exatamente as linguagens regulares da mesma maneira que fazem os DFAs".

"Frequentemente, os NFAs são mais sucintos e mais fáceis de projetar que os DFAs".

"Embora, sempre seja possível converter um NFA em um DFA, esse último pode ter exponencialmente mais estados que o NFA, felizmente, casos desse tipo são raros".

"A diferença entre um DFA e um NFA está no tipo de δ".

"Para um NFA a função δ pode retornar zero, um elemento de estado, ou mais de um estado".

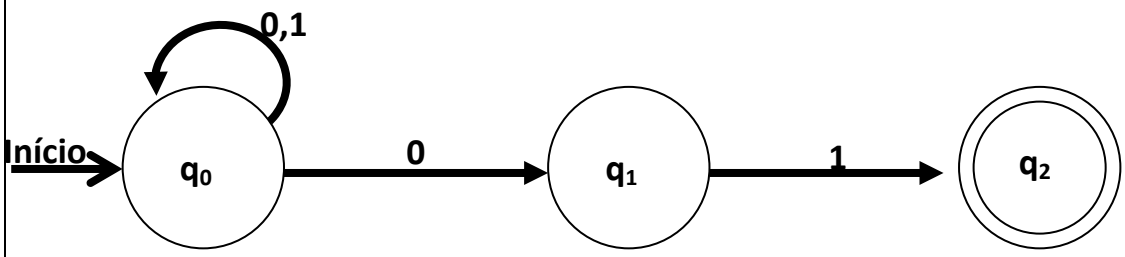
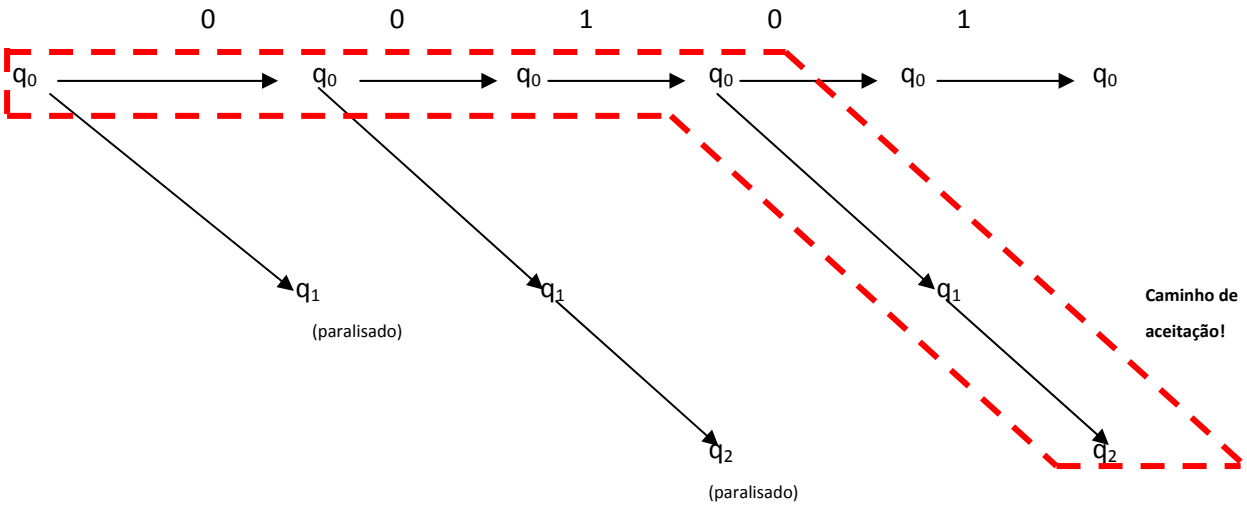


Figura 2 – um NFA que aceita todos os strings que terminar em 01 (x 01)

Sequência de entrada: 00101



Nota:

Como existe um caminho no grafo do NFA que leva o string 00101 para q_2, então este string é aceito.

Definição de Autômato Finito Não-Determinístico

Um NFA é essencialmente representado como um DFA:

A = (Q, ∑, δ, q_0, F) tupla de cinco elementos

Onde,

Q ≡ conjunto finito de estados

∑ ≡ conjunto finito de símbolos de entrada (alfabeto)

δ≡ a função de transição. Se δ retorna um conjunto de estados com mais de um elemento ou nenhum, tem-se um NFA, caso contrário, se δ sempre retorna um único elemento tem-se um DFA.

O NFA da figura 2 toma a seguinte forma algébrica:

A = ({q_0, q_1, q_2}, {0,1}, δ, q_0, {q_2})

Onde,

δ	0	1
→ q_0	{q_0, q_1}	{q_0}
q_1	∅	{q_2}
*q_2	∅	∅

A Função de Transição Estendida em um NFA

Base: $\hat{\delta}(q, \varepsilon) = \{q\}$. O NFA permanece no mesmo estado quando não lê nenhum símbolo.

Indução: Seja w um string da forma w = xa, onde ***a*** é o ultimo símbolo de w e ***x*** é o restante de w. Seja também que $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_R\}$. Seja,

$$\bigcup_{i=1}^{r} \delta\,(pi, a) = \{r_1, r_2, \ldots, r_m\}$$

Então, $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, r_m\}$, ou sejam pode-se calcular $\hat{\delta}(q, w)$ calculando primeiro $\hat{\delta}(q, x)$.

"Um NFA aceita um string ***w*** se é possível tomar qualquer sequência de escolhas do próximo estado, enquanto são lidos os caracteres de ***w***, e ir do estado inicial para algum estado de aceitação".

A Linguagem de um NFA

Formalmente, se A = (Q, ∑, δ, q_0, F) é um NFA, então L(A) = {w | $\hat{\delta}(q,w) \cap F \neq \varnothing$}

Isto é, L(A) é o conjunto de strings w em ∑* tais que $\hat{\delta}(q_0,w)$ contém pelo menos um estado de aceitação.

Equivalência entre DFA e NFA

- Embora existam muitas linguagens para as quais um NFA é mais fácil de construir que um DFA, é um fato surpreendente que toda linguagem que pode ser descrita por algum NFA também possa ser descrita por algum DFA.
- Entretanto, no pior caso, o menor DFA pode ter 2^n estados, enquanto o menor NFA para a mesma linguagem tem apenas ***n*** estados.
- A prova de que os DFAs podem fazer tudo que os NFAs podem fazer envolve uma "construção" importante, denominada construção de subconjuntos, porque inclui a **construção** de todos os subconjuntos do conjunto de estados do NFA.
- É importante observar a **construção de subconjuntos** como um exemplo de como se descreve formalmente um autômato em termos dos estados e transições de outro, sem conhecer os detalhes específicos desse último autômato.

  A **construção de subconjuntos** começa a partir de um NFA N = (Q_N, ∑, δ_D, {q_0}, F_D) tal que **L(D) = L(N)**. Os alfabetos de entrada dos dois autômatos são os mesmos e o estado **inicial** de ***D*** é o conjunto que contém apenas o estado inicial de ***N***, e
  - Q_D é o conjunto de subconjuntos de Q_N, isto é, Q_D é o conjunto potência de Q_N. Note que, se Q_N tem ***n*** estados, então Q_D terá 2^n estados;
  - F_D é o conjunto de subconjuntos ***S*** de Q_N tais que $S \cap F_N \neq 0$. Isto é, F_D representa todos os conjuntos de estados de ***N*** que incluem pelo menos um estado de aceitação de ***N***;
  - Para cada conjunto $S \subseteq Q_N$ e para cada símbolo de entrada ***a*** em ∑, $\delta_D(S,a) = \cup\, \delta_N(p,a)$ em S

- Frequentemente pode-se evitar a etapa de tempo exponencial da construção de entradas da tabela de transições para todos os subconjuntos de estados, se executarmos uma "avaliação ociosa" nos subconjuntos como segue.

  Base: Sabe-se com certeza que o conjunto que consiste apenas no estado inicial de ***N*** é acessível.

  Indução: Supondo que o conjunto ***S*** de estados é acessível, então para cada símbolo de entrada ***a***, determina-se o conjunto de estados $\delta_D(s,a)$, sabendo que esses conjuntos de estados também serão acessíveis.

Teorema 01: Se D = {Q_D, ∑, δ_D, {q_0}, F_D} é o DFA construído a partir do NFA N = {Q_N, ∑, δ_N, q_0, F_N} pela construção de subconjuntos, então L(D) = L(N).

Teorema 02: Uma linguagem ***L*** é aceita por algum DFA se e somente se ***L*** é aceita por algum NFA.

Exemplo:

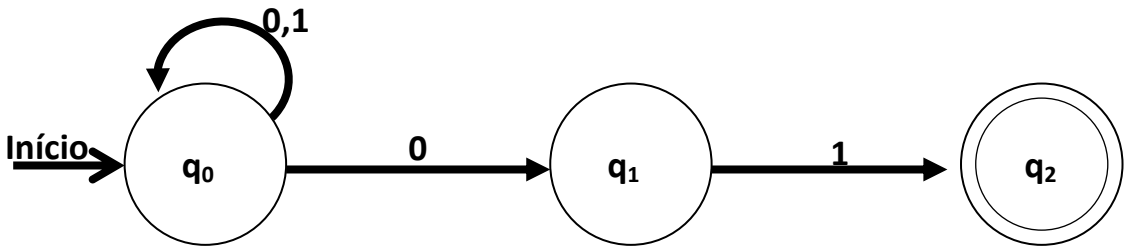


Figura 3 – Esquema de um NFA denominado A_N

$A_N = (Q_N, \sum, \delta_N, q_0, F_N) = (\{q_0, q_1, q_2\}, \{0,1\}, \delta_N, q_0, \{q_2\})$

δ	0	1
→ q_0	{q_0, q_1}	{q_0}
q_1	∅	{q_2}
*q_2	∅	∅

Convertendo o NFA para um DFA equivalente $A_D = (Q_D, \sum, \delta_D, q_0, F_D)$

∑ = {0,1} e $q_0 = q_0$

δ_D	0	1
∅		
→ q_0	{ q_0, q_1}	{q_0}
q_1		
* q_2		
{ q_0, q_1}	{ q_0, q_1}	{ q_0, q_2}
* { q_0, q_2}	{ q_0, q_1}	{q_0}
* { q_1, q_2}		
* { q_0, q_1, q_2}		

Não precisa mais de cálculos, as outras linhas podem ser desprezadas.

$\delta_D (\{q_0, q_1\}, 0) = \delta_N (\{q_0\}, 0) \cup \delta_N (\{q_1\}, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$

$\delta_D (\{q_0, q_1\}, 1) = \delta_N (\{q_0\}, 1) \cup \delta_N (\{q_1\}, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

$\delta_D (\{q_0, q_2\}, 0) = \delta_N (\{q_0\}, 0) \cup \delta_N (\{q_2\}, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$

$\delta_D (\{q_0, q_2\}, 1) = \delta_N (\{q_0\}, 1) \cup \delta_N (\{q_2\}, 1) = \{q_0\} \cup \varnothing = \{q_0\}$

Analisada a função de transição δ_D pode-se gerar o seguinte DFA,

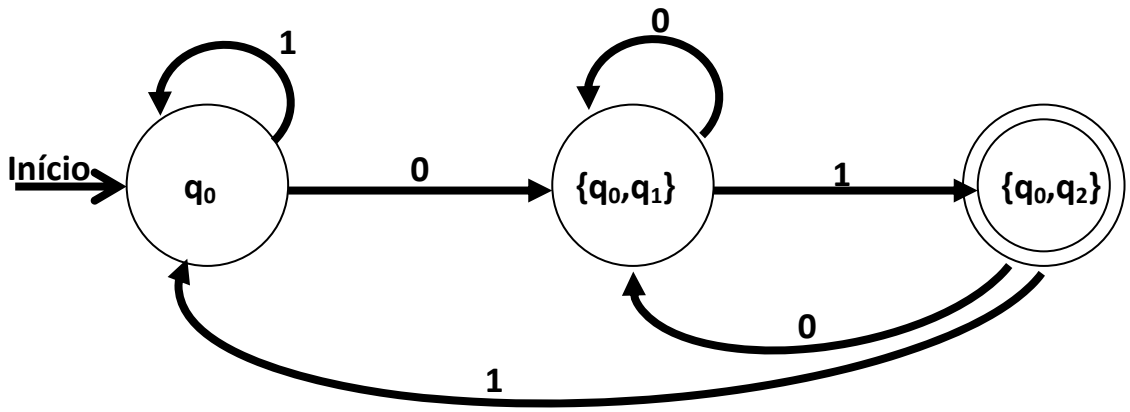


Figura 4 – Esquema de um DFA denominado A_D

- É surpreendente o fato de que o DFA A_D faz a mesma coisa que o NFA A_N no sentido de que $L(A_D) = L(A_N)$.

- Se você não gostou da notação acima, reorganize-a conforme esquematizado abaixo:

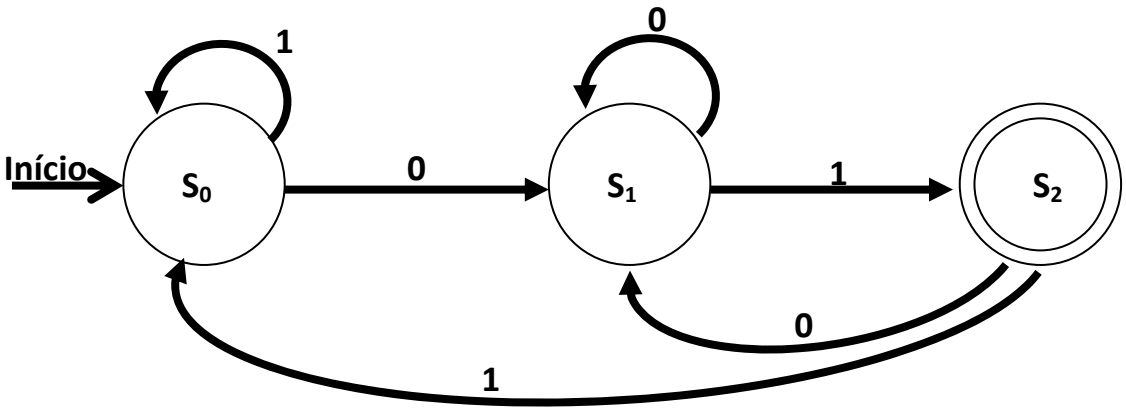
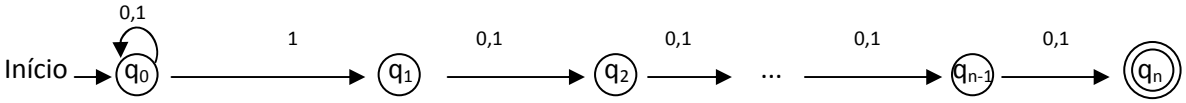
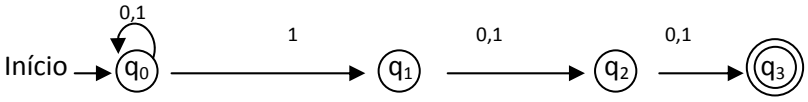


Figura 5 – Esquema de um DFA denominado A_D reorganizado

δ	0	1
→ S_0	S_1	S_0
S_1	S_1	S_2
*S_2	S_1	S_0



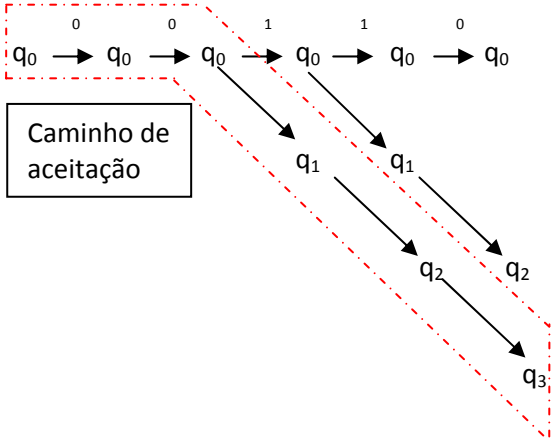
Esquema de um NFA para o conjunto de todos os strings de 0's e 1's tais que o n-ésimo símbolo a partir do fim é 1.



(n=3)

δ_N	0	1
→ q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\{q_3\}$	$\{q_3\}$
*q_3	$\varnothing$	$\varnothing$

String 00110 (n=3)



δ_D	0	1
$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_1\}$	$\{q_2\}$	$\{q_2\}$
…	…	…
$\{q_n\}$	$\varnothing$	$\varnothing$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q1, \ldots q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_3\}$	$\{q_0, q1, \ldots q_3\}$
…	…	…
$\{q_0, q_n\}$		
…	…	…
$\{q_0, q1, \ldots q_n\}$		

$\delta_D(\{q_0, q_1\}, 0) = \delta_N(\{q_0\}, 0) \cup \delta_N(\{q_1\}, 0)$

$= \{q_0\} \cup \{q_2\} = (\{q_0, q_2\}$

$\delta_D(\{q_0, q_1\}, 1) = \delta_N(\{q_0\}, 1) \cup \delta_N(\{q_1\}, 1)$

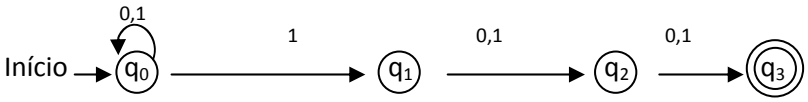
$= \{ q_0, q_1\} \cup \{q_2\} = (\{q_0, q_1, q_2\}$

Total de Linhas:
n + 1
2 linhas
Linhas Realmente utilizadas:
2^n linhas

Entretanto, δ_N tem apenas n+1 linhas!

Esse processo indutivo irá levá-lo a percorrer 2^n linhas da função de transição δ_D

Reconhecer o string: 00**1**10 (n = 3)



(Exemplo de um NFA)

δ_N	0	1
→ q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\{q_3\}$	$\{q_3\}$
*q_3	$\varnothing$	$\varnothing$

	δ_D	0	1
	∅	~~∅~~	~~∅~~ morre
1	→ {q0}	{q0}	{q0, q1}
	{q_1}	~~{q_2}~~	~~{q_2}~~ morre
	{q_2}	~~{q_3}~~	~~{q_3}~~ morre
	*{q_3}	~~∅~~	~~∅~~ morre
2	{q_0, q_1}	{q_0, q_2}	{q_0, q1, ... q_2}
3	{q_0, q_2}	{q_0, q_3}	{q_0, q1, ... q_3}
4	{q_0, q_3}	{q_0}	{q_0, q_1} (descartado)
	{q_1, q_2}		
	*{q_1, q_3}		
	*{q_2, q_3}		
5	{q_0, q_1, q_2}	{q_0, q_2, q_3}	{ q_0, q_1, q_2, q_3}
6	*{q_0, q_1, q_3}	{q_0, q_2}	{q_0, q_1, q_2} (descartado)
7	*{q_0, q_2, q_3}	{q_0, q_3}	{q_0, q_1, q_3} (descartado)
	*{q_1, q_2, q_3}		
8	*{ q_0, q_1, q_2, q_3}	{q_0, q_2, q_3}	{ q_0, q_1, q_2, q_3} (descartado)

$\delta_D(\{q_0, q_1\}, 0) = \delta_N(\{q_0\}, 0) \cup \delta_N(\{q_1\}, 0) = \{q_0\} \cup \{q_2\} = (\{q_0, q_2\}$

$\delta_D(\{q_0, q_1\}, 1) = \delta_N(\{q_0\}, 1) \cup \delta_N(\{q_1\}, 1) = \{ q_0, q_1\} \cup \{q_2\} = (\{q_0, q_1, q_2\}$

O DFA equivalente ao NFA da página anterior

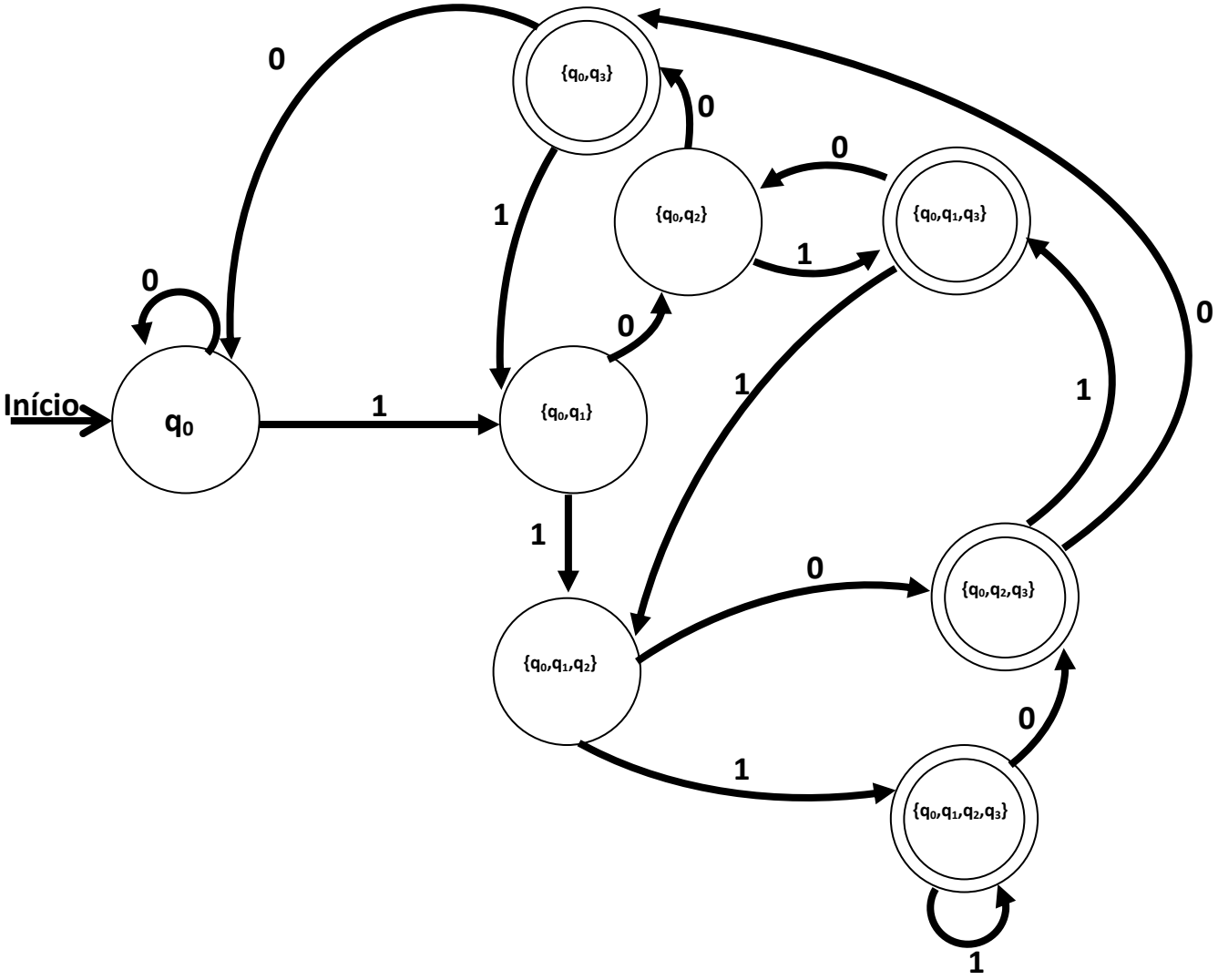
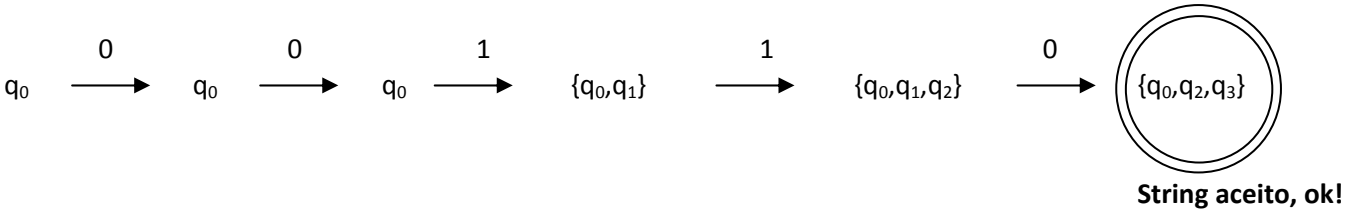


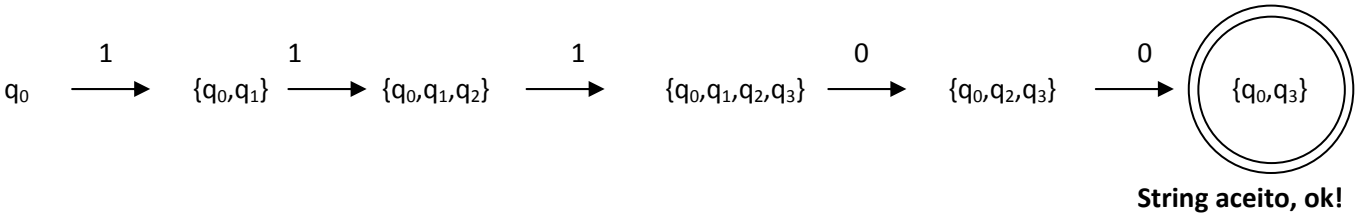
Figura 6 – DFA equivalente ao NFA

1. **Teste para o String**: 00**1**10



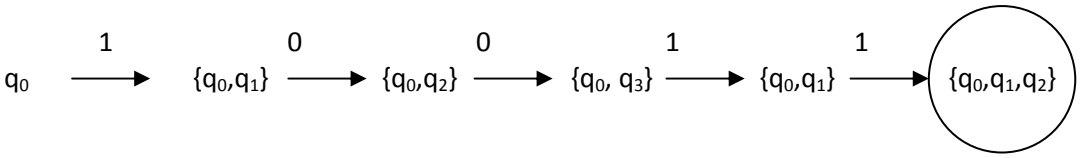
String aceito, ok!

2. **Teste para o string**: 11**1**00



String aceito, ok!

3. **Teste para o string**: 10**0**11



String rejeitado, ok!

Autômatos Finitos com Epsílon-transições ou ε-Transições

- Os NFA's com ε-transições em δ, que é denominado ε-NFA's, estão intimamente relacionados às expressões regulares, e são úteis para provar a equivalência entre as classes de linguagens por autômatos finitos e por expressões regulares.
- A linguagem de ***A***, L(A), é o conjunto de strings ***w*** que levam o estado inicial q_0, até um dos estados de aceitação. Em notação de conjunto, L(A) = {w | $\hat{\delta}$ (q_0,w) está em ***F***}
- Se ***L*** é L(A) para algum DFA A, diz-se que ***L*** é um linguagem regular.

Exemplo de um ε-NFA's:

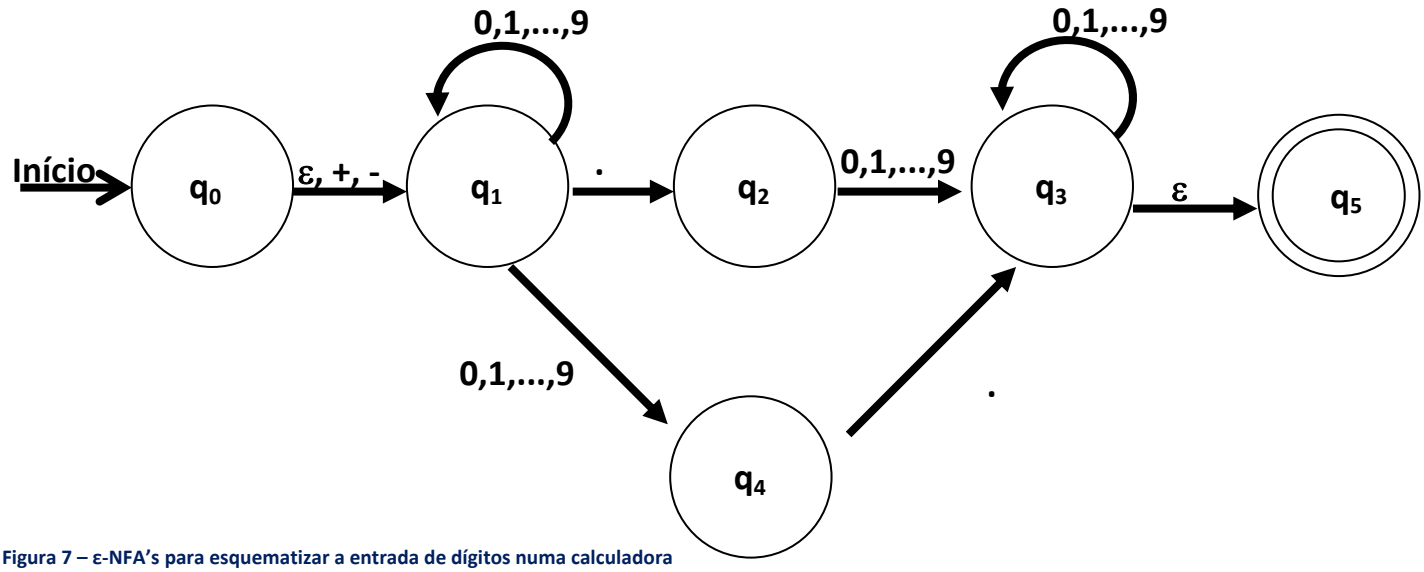


Figura 7 – ε-NFA's para esquematizar a entrada de dígitos numa calculadora

A Notação Formal para um ε-NFA

Um ε-NFA A por A = (Q,∑,δ,q_0,F) para δ projetado da seguinte forma,

Argumentos que δ pode receber

1. δ ← um estado de Q
2. δ ← um elemento de ∑ ∪ {ε}, ou seja, o string ε é separado do alfabeto ∑ para evitar confusões.

Assim, para o autômato ε-NFA da página anterior, tem-se δ da seguinte forma:

E = ({q_0, q_1, q_2, ..., q_5}, {.,+,-,0,1,...,9},δ,q_0,{q_5})

δ	ε	+,-	.	0,1,...,9
→ q_0	{q_1}	{q_1}	∅	{∅}
q_1	∅	∅	{q_2}	{q_1,q_4}
q_2	∅	∅	∅	{q_3}
q_3	{q_5}	∅	∅	{q_3}
q_4	∅	∅	{q_3}	∅
*q_5	∅	∅	∅	∅

ε-fechamento de um estado e função transição estendida ($\hat{\delta}$) num ε-NFA's

- Informalmente, o "ε-fechamento de um estado q" é utilizado para seguir as transições saindo de ***q*** e que são rotuladas por ε.
- Formalmente, o "ε-fechamento de um estado ***q***, abreviado por ***ECLOSE(q)*** pode ser definido recursivamente da seguinte forma:
  - **Base**: o estado q está em ECLOSE(q).
  - **Indução**: se o estado p está em ECLOSE(q), e existe uma transição do estado ***p*** para o estado ***r*** está em ECLOSE(q). Assim, se δ é a função de transição do ε-NFA envolvido, então ECLOSE(q) contém todos os estado em δ(p,ε).

  Para o exemplo da figura 7, tem-se:

  ECLOSE(q_0) = {q_0,q_1}

  ECLOSE(q_3) = {q_3,q_5}

  ECLOSE(q_i) = {q_i} para i = 1,2,4,5

- **Vantagem do ε-fechamento**: permite explicar facilmente qual será a aparência das transições de um ε-NFA quando é dada uma sequência de entrada "não vazia".

Definição de $\hat{\delta}$ para um ε-NFA

- Seja um ε-NFA do tipo E=(Q,∑,δ,q_0,F), então a definição apropriada para $\hat{\delta}$ é:
  - **Base**: $\hat{\delta}(q,\varepsilon)$ = ECLOSE(q)
  - **Indução**: Seja ***w*** da forma ***w=xa***, então $\hat{\delta}(q,w)$ será determinado como segue:
    1. Seja $\{p_1,p_2,..,p_k\}$ o valor de $\hat{\delta}(q,x)$
    2. Seja $\bigcup_{i=1}^{k} \delta(pi,a)$ o conjunto $\{R_1,r_2,...,r_m\}$
       Os $r_{j's}$ são apenas alguns dos estados que se pode alcançar a partir de q ao longo de caminhos rotulados por w. Os estados adicionais são determinados a partir dos $r_{j's}$ seguindo-se arcos rotulados por ε na etapa (3).
    3. $\hat{\delta}(q,w) = \bigcup_{j=1}^{m}$ ECLOSE(r_j). Está expressão considera a possibilidade de existirem arcos adicionais rotulados por ε após o ultimo símbolo "real", ***a***.

Nota: Para se compreender melhor a definição indutiva acima, considere a seguir, três exemplos para o calculo de $\hat{\delta}$, um exemplo para "DFA", outro para "NFA" e por fim um exemplo para "ε-NFA".

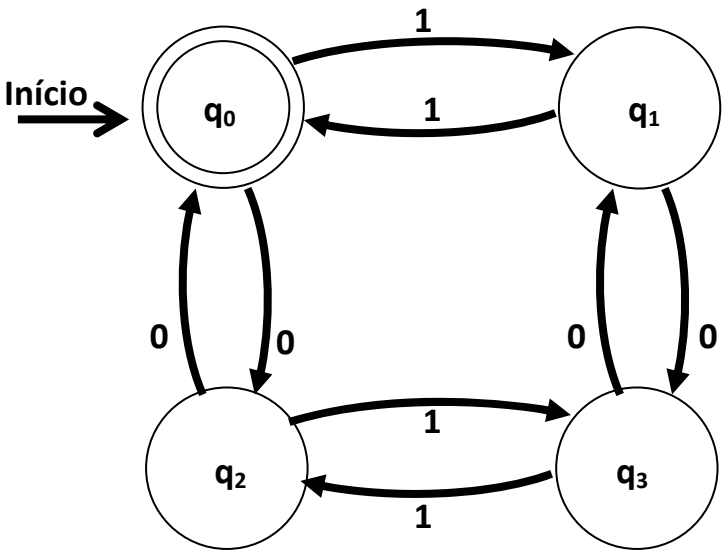
Funções de Transição Estendidas

DFA

- Base: $\hat{\delta}(q,\varepsilon) = q$
- Indução: $\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x),a)$ para w = xa

Exemplo:

L = {w/w tem ao mesmo tempo um número par de 0's e um número par de 1's}



δ	0	1
*→ q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figura 8 – DFA's para função de transição estendida

$\hat{\delta}(q_0,110101) = ?$

$\hat{\delta}(q_0,\varepsilon) = q_0$

$\hat{\delta}(q_0,1) = \delta(\hat{\delta}(q_0,\varepsilon),1) = \delta(q_0,1) = q_1$

$\hat{\delta}(q_0,11) = \delta(\hat{\delta}(q_0,1),1) = \delta(q_1,1) = q_0$

$\hat{\delta}(q_0,110) = \delta(\hat{\delta}(q_0,11),0) = \delta(q_0,0) = q_2$

$\hat{\delta}(q_0,1101) = \delta(\hat{\delta}(q_0,110),1) = \delta(q_2,1) = q_3$

$\hat{\delta}(q_0,11010) = \delta(\hat{\delta}(q_0,1101),0) = \delta(q_3,0) = q_1$

$\hat{\delta}(q_0,110101) = \delta(\hat{\delta}(q_0,11010),1) = \delta(q_1,1) = q_0$ (string aceito, ok!)

NFA

Base: $\hat{\delta}(q,\varepsilon) = \{q\}$

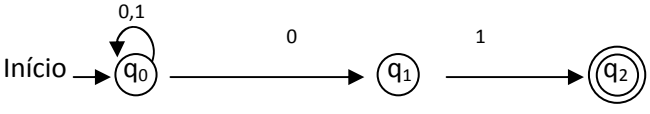
Indução: para w=xa e $\hat{\delta}(q,x) = \{p_1,p_2,...,p_k\}$ e seja

$\bigcup_{i=1}^{k} \delta(p_i,a) = \{r_1,r_2,...,r_m\}$

Então, $\hat{\delta}(q,w) = \{r_1,r_2,...,r_m\}$, ou seja, $\hat{\delta}(q,w) \leftarrow f[\hat{\delta}(q,x)]$

Exemplo:

L = {w/ w=x01}



δ_N	0	1
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	∅	$\{q_2\}$
*q_2	∅	∅

$\hat{\delta}(q_0,00101) = ?$

$\hat{\delta}(q_0,\varepsilon) = q_0$

$\hat{\delta}(q_0,0) = \delta(\hat{\delta}(q_0,\varepsilon),0) = \delta(q_0,0) = \{q_0,q_1\}$

$\hat{\delta}(q_0,00) = \delta(q_0,0) \cup \delta(q_1,0) = \{q_0,q_1\} \cup \varnothing = \{q_0,q_1\}$ *

$\hat{\delta}(q_0,001) = \delta(q_0,1) \cup \delta(q_1,1) = \{q_0\} \cup \{q_2\} = \{q_0,q_2\}$

$\hat{\delta}(q_0,0010) = \delta(q_0,0) \cup \delta(q_2,0) = \{q_0,q_1\} \cup \varnothing = \{q_0,q_1\}$

$\hat{\delta}(q_0,00101) = \delta(q_0,1) \cup \delta(q_0,1) = \{q_0\} \cup \{q_2\} = \{q_0,q_2\}$ (string aceito, ok!)

$\hat{\delta}(q_0,00) =$

$\delta(\hat{\delta}(q_0,0),0) =$

$\delta(\{q_0,q_1\},0) =$

$\delta(q_0,0) \cup \delta(q_1,0) =$

$\{q_0,q_1\} \cup \varnothing = \{q_0,q_1\}$

ε-NFA

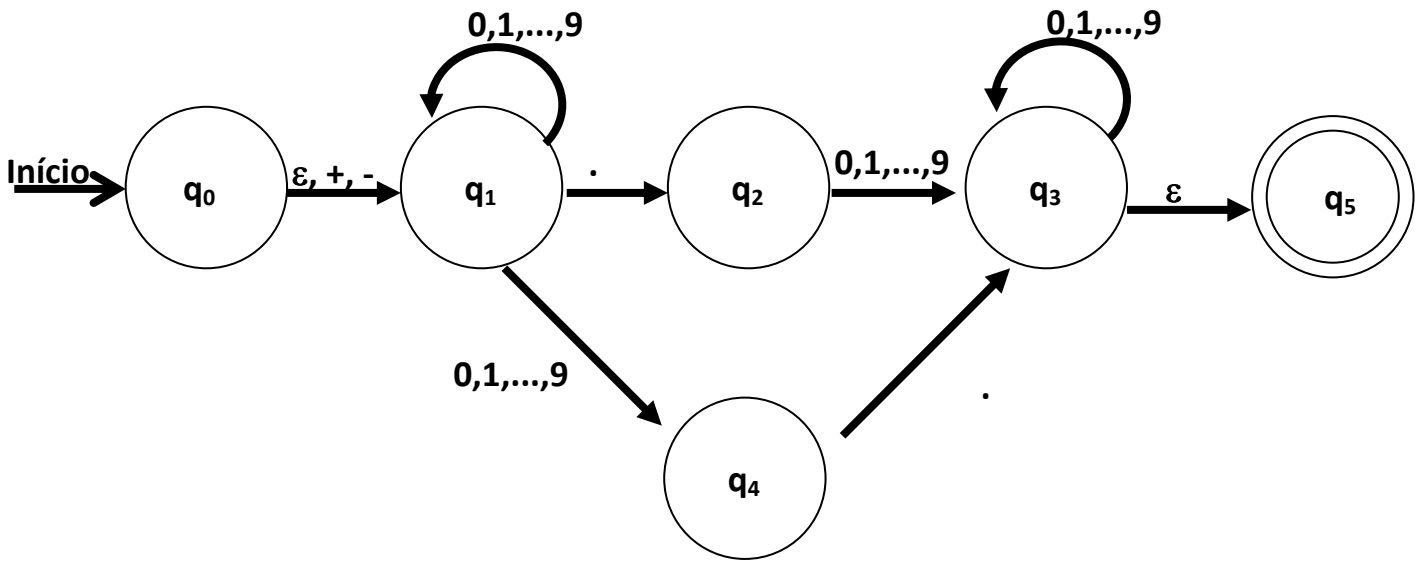


Figura 9 – ε-NFA's para função de transição estendida

δ	ε	+,-	.	0,1,...,9
→ q_0	$\{q_1\}$	$\{q_1\}$	∅	$\{\varnothing\}$
q_1	∅	∅	$\{q_2\}$	$\{q_1,q_4\}$
q_2	∅	∅	∅	$\{q_3\}$
q_3	$\{q_5\}$	∅	∅	$\{q_3\}$
q_4	∅	∅	$\{q_3\}$	∅
*q_5	∅	∅	∅	∅

$\hat{\delta}(q_0,5.6) = ?$

$\hat{\delta}(q_0,\varepsilon) = \text{ECLOSE}(q_0) = \{q_0,q_1\}$

1) $\delta(q_0,5) \cup \delta(q_1,5) = \varnothing \cup \{q_1,q_4\} = \{q_1,q_4\}$

$\hat{\delta}(q_0,5) = \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup (q_4) = \{q_1,q_4\}$

2) $\delta(q_1,.) \cup \delta(q_4,.) = \{q_2\} \cup \{q_3\} = \{q_2,q_3\}$

$\hat{\delta}(q_0,5.) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup (q_3,q_5) = \{q_2,q_3,q_5\}$

3) $\delta(q_2,6) \cup \delta(q_3,6) \cup \delta(q_5,6) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{ q_3\}$

$\hat{\delta}(q_0,5.6) = \text{ECLOSE}(q_3) = \{q_3,q_5\}$

Logo, $\boxed{\hat{\delta}(q_0,5.6) = \{q_3,q_5\}}$ (5.6 string aceito, ok!)

Eliminação de ε-transições

- Dado qualquer ε-NFA ***E***, então pode-se encontrar um DFA ***D*** que aceita a mesma linguagem que ***E***. Isto é feito introduzindo as ε transições de ***E*** em ***D*** através do mecanismo de ε-fechamento.

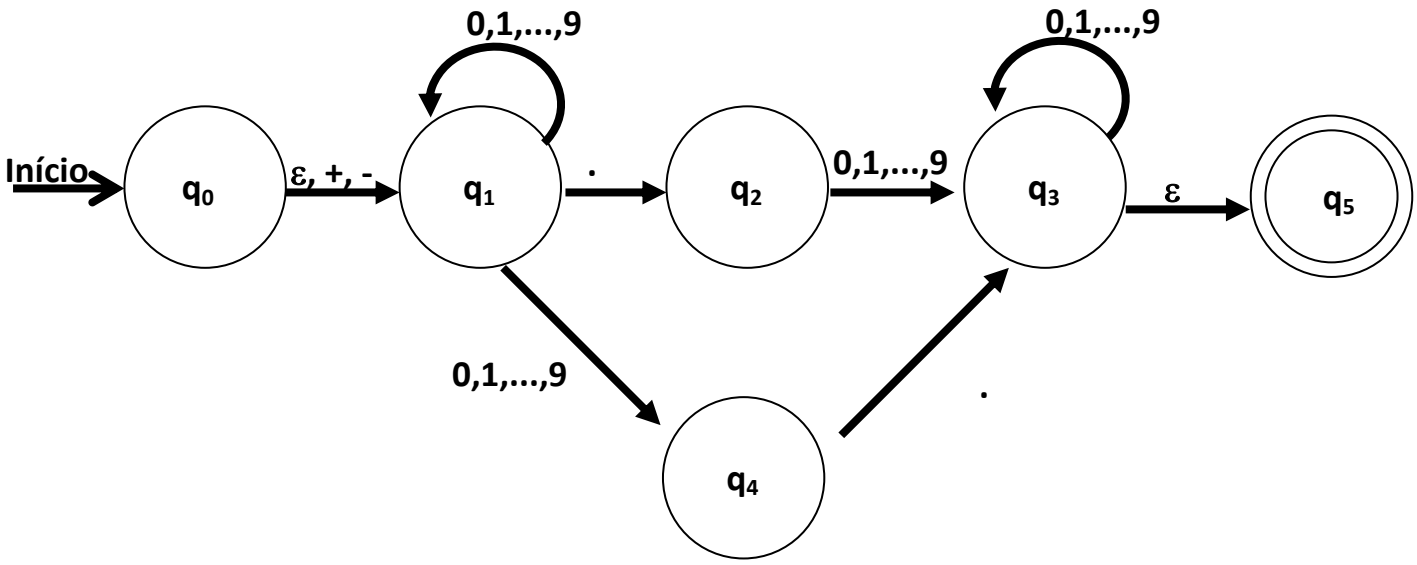
Seja $E = (Q_E,\sum,\delta_E,q_0,F_E)$. Então o DFA equivalente $D = (Q_D,\sum,\delta_D,q_D,F_D)$ é definido como a seguir:

1. Q_D é o conjunto de subconjuntos de Q_E.
   "Todos os estados acessíveis de ***D*** são subconjuntos com ε-fechamento de Q_E".
2. $Q_D = \text{ECLOSE}(q_0)$
3. $F_D = \{S \,/\, S \text{ está em } Q_D \text{ e } S \cap F_E \neq \varnothing\}$

   "F_D apresenta os conjuntos de estados que contém pelo menos um estado de aceitação de ***E***."
   "$S \subseteq Q_E$ tais que $S = \text{ECLOSE}(S)$".

4. δ_D = (S,a) é calculado, para todo ***a*** em ∑ e todos os conjuntos ***S*** em Q_D por:
   a) Seja S = $\{p_1,p_2,...,p_r\}$
   b) Calcule $\bigcup_{i=1}^{k} \delta_E (p_i,a)$; seja esse conjunto $\{r_1,r_2,...,r_m\}$
   c) Então δ_D (S,a) = $\bigcup_{j=1}^{m} \text{ECOLOSE}(rj)$

Teorema: Uma linguagem ***L*** é aceita por algum ε-NFA se e somente se ***L*** é aceita por algum DFA.



c	ε	+,-	.	0,1,...,9
→ q_0	$\{q_1\}$	$\{q_1\}$	∅	{∅}
q_1	∅	∅	$\{q_2\}$	$\{q_1,q_4\}$
q_2	∅	∅	∅	$\{q_3\}$
q_3	$\{q_5\}$	∅	∅	$\{q_3\}$
q_4	∅	∅	$\{q_3\}$	∅
*q_5	∅	∅	∅	∅

Objetivo: de $E = (Q_E,\sum,\delta_E,q_0,F_E)$ deseja-se obter $D = (Q_D,\sum,\delta_D,q_0,F_D)$.

Solução: Estado Inicial (→)

$q_D = ECLOSE(q_0) = \{q_0,q_1\}$

$\delta_E(q_0,\{+,-\}) \cup \delta_E(q_1,\{+,-\}) = \{q1\} \cup \{\varnothing\} = \{q_1\}$

$\delta_D(\{q_0,q_1\},\{+,-\}) = ECLOSE(q_1) = \{q_1\}$

$\delta_E(q_0,\{.\}) \cup \delta_E(q_1,\{.\}) = \{\varnothing\} \cup \{q_2\} = \{q_2\}$

$\delta_D(\{q_0,q_1\},\{.\}) = ECLOSE(q_2) = \{q_2\}$

$\delta_E(q_0,\{0,1,...,9\}) \cup \delta_E(q_1,\{0,1,...,9\}) = \varnothing \cup \{q_1,q_4\} = \{q_1,q_4\}$

$\delta_D(\{q_0,q_1\},\{0,1,...,9\}) = ECLOSE(q_1,q_4) = \delta_D(\{q_0,q_1\},\{0,1,...,9\}) = \{q_1,q_4\}$

...

E assim por diante! Observe todos os valores mais relevantes na tabela abaixo.

δ	ε	+,-	.	0,1,...,9
∅	-	∅	∅	∅
q_0	-	∅	∅	∅
q_1	-	∅	$\{q_2\}$	$\{q_1,q_4\}$
q_2	-	∅	∅	$\{q_3,q_5\}$
...	...	...	...	...
→ $\{q_0, q_1\}$	-	$\{q_1\}$	$\{q_2\}$	$\{q_1,q_4\}$
...	-	...	...	...
$\{q_1, q_4\}$	-	∅	$\{q_2,q_3,q_5\}$	$\{q_1,q_4\}$
...	-	...	...	...
*$\{q_3, q_5\}$	-	∅	∅	$\{q_3,q_5\}$
...	-	...	...	...
*$\{q_2,q_3, q_5\}$	-	∅	∅	$\{q_3,q_5\}$
...	...	...	...	...

são dispensados

Tecnicamente esta tabela para δ_D deveria ter $2^6 = 64$ linhas. Entretanto para este caso basta computar 7 linhas.

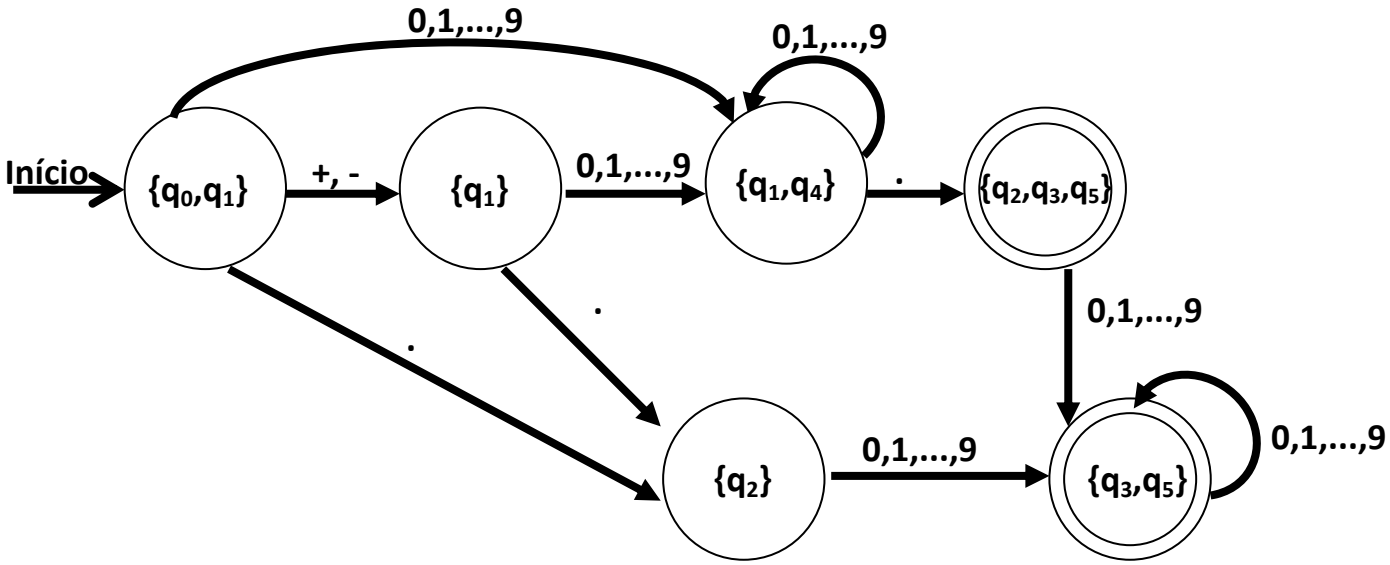


Figura 10 – O DFA D que elimina ε-transições no ε-NFA's da figura 9

Este DFA está incompleto!

Notas:

$\delta_E(q_2),\{+,-\} = \varnothing$

$\delta_D(q_2),\{+,-\} = \text{ECLOSE}(\varnothing)$

$\delta_D(q_2),\{+,-\} = \varnothing$

$\delta_E(q_2),\{.\} = \varnothing$

$\delta_D(q_2),\{.\} = \text{ECLOSE}(\varnothing)$

$\delta_D(q_2),\{.\} = \varnothing$

De maneira análoga:

$\delta_D(q_2),\{0,1,...,9\} = \{q_3,q_5\}$

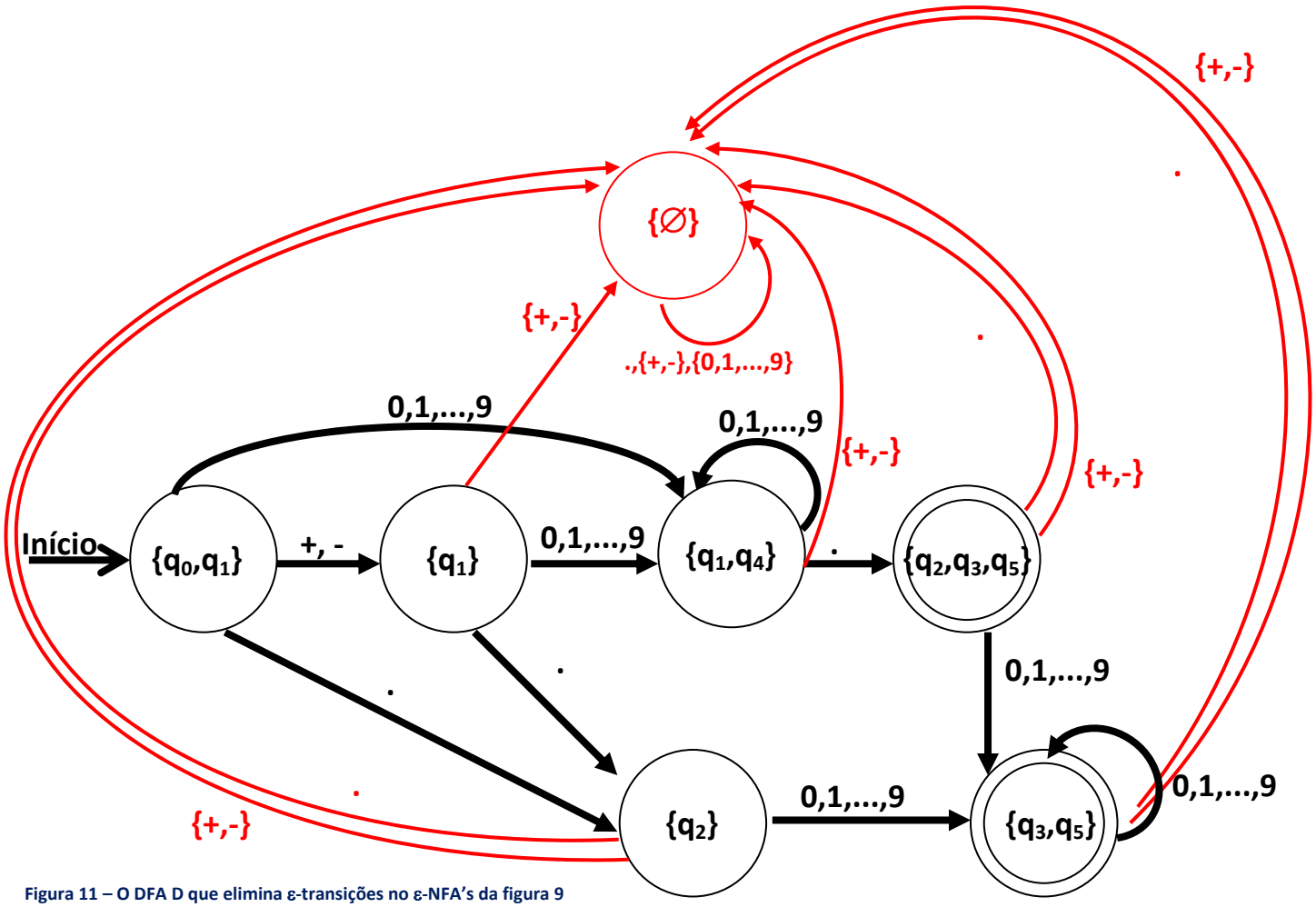


Figura 11 – O DFA D que elimina ε-transições no ε-NFA's da figura 9

δ	ε	+,-	.	0,1,...,9
∅	-	∅	∅	∅
q_1	-	∅	$\{q_2\}$	$\{q_1,q_4\}$
q_2	-	∅	∅	$\{q_3,q_5\}$
→ $\{q_0, q_1\}$	-	$\{q_1\}$	$\{q_2\}$	$\{q_1,q_4\}$
$\{q_1, q_4\}$	-	∅	$\{q_2,q_3,q_5\}$	$\{q_1,q_4\}$
*$\{q_3, q_5\}$	-	∅	∅	$\{q_3,q_5\}$
*$\{q_2,q_3, q_5\}$	-	∅	∅	$\{q_3,q_5\}$

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Expressões Regulares e Autômatos Finitos

Índice

Figuras

Expressões Regulares e Autômatos Finitos

Os operadores de Expressões Regulares

i. **A união de duas linguagens *L* e *M***, denotadas por L ∪ M é o conjunto de strings que estão em ***L*** ou ***M***, ou em ambas.

Exemplo: Se L = {001,10,111} e M = {ε,001} então L ∪ M = {ε,10,001,111}.

ii. **A concatenação de linguagens *L* e *M***, é o conjunto de strings que podem ser formados tomando-se qualquer string em L e concatenando-se esse string com qualquer string em M.

Exemplo: Se L = {001,10,111} e M = {ε,001} então L . M, ou simplesmente LM é {001, 10, 111, 001001, 10001, 111001}, observando que ε é a identidade para a concatenação.

iii. **O fechamento (ou estrelam ou fechamento de kleeme) de uma linguagem** L: é denotadas ***L**** e representa o conjunto dos strings que podem ser formando tomando-se qualquer numero de strings de ***L***, possivelmente comm repetições. Mais formalmente, ***L**** é a união infinita $\bigcup_{i\geq 0} L^i$, onde $L^0 = \{\varepsilon\}$, L^1=L e L^i para i≥1 é LL...L (a concatenação de i cópias de L).

i vezes

Para calcular ***L****, devemos calcular L^i para cada ***i***, e tomar 2.

Exemplo: união de todas essas linguagens. L^i tem 2^i elementos (para alfabetos binários).

Embora cada L^i seja finito, a união de número infinito de termos L^i é em geral linguagem infinita.

SE L = {0,1} então L^0={ε}, L^1={0,1}, L^2={00,01,10,11, ..., L*=$L^0 \cup L^1 \cup$...}

Observações sobre Expressões Regulares e Autômatos Finitos

a) As expressões regulares podem definir todas e somente as linguagens regulares;

b) As expressões regulares podem definir exatamente as mesmas linguagens que as diversas formas de autômatos finitos (determinísticos ou não-determinísticos) descrevem, ou sejam as linguagens regulares;

c) As expressões regulares oferecem algo que os autômatos não oferecem: Um modo declarativo de expressar somente os strings que queremos aceitar;

d) As expressões regulares podem ser descritas recursivamente;

Uma expressão regular ***E*** é apenas uma expressão, não uma linguagem. Devemos usar L(E) quando quisermos nos referir à linguagem que ***E*** denota. Porém, é prática comum fazer referência a, digamos, "E" quando realmente queremos nos referir a "L(E)";

e) Suponha que ***L*** seja a linguagem que contém strings de comprimento 1 e para cada símbolo a em ∑, existe um string a em ***L***. Neste caso, ***L*** é dito um conjunto de strings e ∑ é dito um conjunto de símbolos. Por outro lado, L* denota a mesma linguagem de ∑*;

f) **Precedência de operadores**: * > concatenação > União { 01*+1 é / (0(1*))+1

g) ∅* = {ε} e ∅i = {∅} **Nota**: O fechamento para o conjunto vazio não é infinito.

Descrição Recursiva para as expressões regulares

Base:

1. As constantes ε e ∅ são expressões regulares, denotando as linguagens {ε} e ∅, respectivamente. Isto é, L(ε) = {ε} e L(∅) = ∅.
2. Se ***a*** é qualquer símbolo, então ***a*** é uma expressão regular. Essa expressão denota a linguagem {a}. Isto é, L(a)={a}.
3. Uma variável, em geral escrita em maiúscula e em itálico, como ***L***, é uma variável que representa qualquer linguagem.

Indução: Há quatro partes para a etapa indutiva, uma para cada um dos três operadores e uma para a introdução de parênteses.

1. Se E e F são expressões regulares, então E+F é uma expressão regular denotando a união de L(E) e L(F). Isto é, L(E+F) = L(E)+L(F).
2. Se E e F são expressões regulares, então EF é uma expressão regular denotando a concatenação de L(E) e L(F). Isto é, L(EF) = L(E) L(F).
3. Se E é expressão regular, então E* é uma expressão regular denotando o fechamento de L(E). Isto é, L(E*) = (L(E))*.
4. Se E é uma expressão regular, então (E), também é uma expressão regular. Formalmente, L((E)) = L(E).

Exemplo 01:

L = {conjunto de strings em 0's e 1's alternados}

01 ... começa com "0" e termina com "1"

10 ... começa com "1" e termina com "0"

010 ... começa com "0" e termina com "0"

101 ... começa com "1" e termina com "1"

...

010101010101

01010101

Nota: ***L*** pode ser infinito!

Solução: (01)* + (10)* + 0(10)* + 1(01)*

Ou

(ε+1)(01)*(ε+0)

Agrupamento de Expressões Regulares

i. 01* + 1 é equivalente a (0(1*))+1

ii. $\underbrace{(01)^*}$ + 1 ≠ $\underbrace{01^*}$ + 1

Repetição sucessivas de "01" "0" seguido de repetições sucessivas de "1"

(01)* + (10)* + 0(10)* + 1(01)*	=	(ε+1) (01)* (ε+0) ?

{ε,1} (01)* {ε,0}

ε(01)* ε (01)* {ε,0}

ε(01)* ε (01)*0 + 1(01)*ε + 1(01)*0

(01)* + (01)*0 + 1(01)* + 1(01)*0

(01)* + 0(10)* + 1(01)* + (10)*10

$\underbrace{(01)^*}$ + $(10)^+$ + 0(10)* + 1(01)*

$(\varepsilon+(01)^+)$

Pela propriedade de Idempotência (L + L = L), tem-se:

$(\varepsilon+(01)^+) = (\varepsilon+\varepsilon+(01)^+) = (\varepsilon+(01)^+) + \varepsilon = (01)^* + \varepsilon$

Assim, (01)* = (01)* + ε

Logo,

(01)* + (ε + $(10)^+$) + 0(10)* + 1(01)*

(01)* + (10)* + 0(10)* + 1(01)*

Portanto,

(01)* + (10)* + 0(10)* + 1(01)*	=	(ε+1) (01)* (ε+0)

Autômatos Finitos e Expressões Regulares

- Embora a abordagem de expressões regulares para descrever linguagens seja fundamentalmente distinta da abordagem de autômatos finitos, essas duas notações representam exatamente o mesmo conjunto de linguagens, que denominamos “linguagens regulares”.
- Para mostrar que as expressões regulares definem a mesma classe, devemos mostrar que:
    1. Toda linguagem definida por um desses autômatos também PE definida por uma expressão regular. Para essa prova, podemos supor que a linguagem é aceita por algum DFA.
    2. Toda linguagem definida por uma expressão regular é definida por um desses autômatos. Para essa parte da prova, é mais fácil mostrar que existe um NFA com ε-transições que aceita a mesma linguagem.

De Autômatos Finitos para Expressões Regulares

Passo 1: Compreender as leis algébricas para expressões regulares.

Passo2: Aplicar o teorema de equivalência (prova por construção).

“Se L=L(A) para algum DFA A, então existe uma expressão regular R tal que L = L(R).”

Passo 1: as leis algébricas parra expressões regulares::

Comutatividade	$L + M = M + L$ $LM \neq ML$
Associatividade	$(L + M) + N = L + (M + N)$ $(LM)N = L(MN)$
Identidade	$\emptyset + L = L + \emptyset = L$ $\varepsilon L = L\varepsilon = L$
Aniquilador	$\emptyset L = L\emptyset = \emptyset$
Distributiva	$L(M+N) = LM + LN$ $(M+N)L = ML + NL$
Idempotência	$L + L = L$
Fechamentos	$(L^*)^* = L$ $\emptyset^* = \varepsilon$ $\varepsilon^* = \varepsilon$ $L^+ - LL^* = L^*L$ $L^* = L^+ + \varepsilon$ $(L+M)^* = (L^*M^*)^*$

Extras
$(\varepsilon + R)^* = R^*$
$(\varepsilon + R)\, R^* = R^*$
$R^* (\varepsilon + R) = R^*$

Exemplo de Simplificação:

$1^*0 + 1^*0\ (\varepsilon + 0 + 1)^*\ (\varepsilon + 0 + 1) =$

$1^*0 + 1^*0\ (0 + 1)^*\ \ (\varepsilon + (0 + 1)) =$

$1^*0\varepsilon + 1^*0\ (0 + 1)^*\ \ (\varepsilon + (0 + 1)) =$

$1^*0\ (\varepsilon + (0 + 1)^*\ (\varepsilon + (0 + 1)\ )\ ) =$

$1^*0\ (\varepsilon + (0 + 1)^*\ \varepsilon + (0 + 1)^*\ (0 + 1)\ ) =$

$1^*0\ (\varepsilon + (0 + 1)^* + (0 + 1)^+\ ) =$

$1^*0\ \underbrace{(\varepsilon + 0 + 1)^+}_{(0+1)^*} + (0 + 1)^*) =$ ➔ $(L^* = L^+ + \varepsilon)$

$1^*0\ \underbrace{(0 + 1)^* + (0 + 1)^*)}_{(0+1)^*} =$ ➔ $(L + L = L)$

$1^*0\ (0+1)^*$

Propriedades particulares para alfabeto binário:

$(ab)^*a = a(ba)^*$

$(a+b)^* = (a^*+b)^* = a^*(a+b)^* = (a^*b^*)^*$

$a^*(ba^*)^*$

Logo,

$1^*0 + 1^*0\ (\varepsilon + 0 + 1)^*\ (\varepsilon + 0 + 1) = 1^*0(0+1)^*$

Passo 2: Aplicar o teorema de equivalência, que estabelece uma prova por construção.

Teorema: Se L=L(A) para algum DFA A, então existe uma expressão regular R tal que L=L(R).

Prova por construção

$R_{ij}^{(k)}$

O nome de uma expressão regular cuja linguagem é o conjunto de strings ***w*** tais que ***w*** é o rótulo de um caminho do estado ***i*** ao estado ***j*** em A+{1,2,...,n}, e esse caminho não tem nenhum nó intermediário cujo número seja maior que ***k***.

Base: A base é k=0, onde não existe nenhum caminho intermediário entre os estados i e j. Isto pode acontecer se:

1. Apenas um arco do nó ***i*** ao nó ***j***;
2. Um caminho de comprimento zero (de ***i*** para ***i***).

No caso 1 tem-se

a. $R_{ij}^{(0)} = \varnothing$;
b. então $R_{ij}^{(0)} = a$ (um símbolo);
c. então $R_{ij}^{(0)} = R_{ij}^{(0)} = a_1 + a_2 + ... + a_k$ (vários símbolos).

No caso 2 tem-se

a. $R_{ij}^{(0)} = \varepsilon$;
b. ou $R_{ij}^{(0)} = \varepsilon + a$;
c. ou $R_{ij}^{(0)} = \varepsilon + a_1 + ... + a_k$.

Indução: Suponha que exista um caminho do estado ***i*** para o estado ***j*** que não passe por nenhum estado maiss alto que ***k***. Há dois casos:

a. O caminho não passa por ***k*** e assim tem-se no máximo $R_{ij}^{(k-1)}$;
b. O caminho passa por ***k*** pelo menos uma vez, neste caso $R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$;

Exemplo:

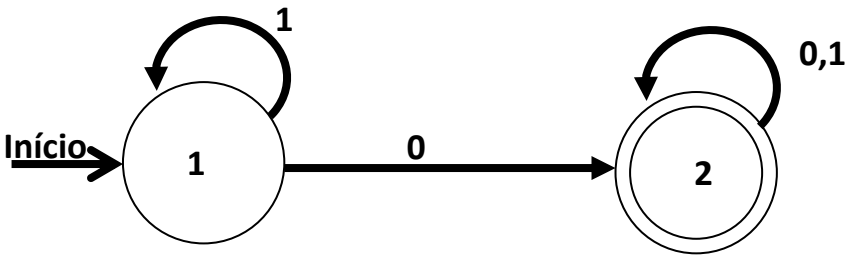


Figura 1 – Automato do conjunto de todos os strings que tem pelo menos um zero

L(DFA) = {conjunto de todos os strings que tem pelo menos um zero}

Base	
$R_{11}^{(0)}$	$\varepsilon + 1$
$R_{12}^{(0)}$	0
$R_{21}^{(0)}$	$\varnothing$
$R_{22}^{(0)}$	$(\varepsilon + 0 + 1)$

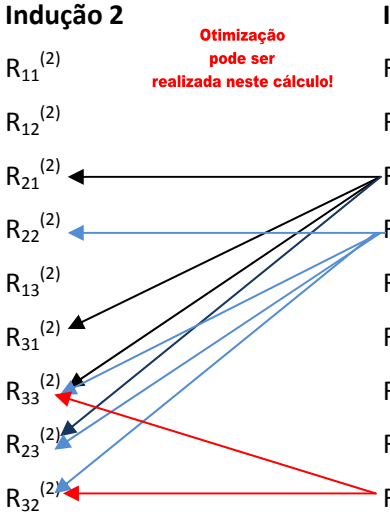
Processo Indutivo
$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$

$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^*R_{1j}^{(0)}$		
	Substituição Direta	Simplificada
$R_{11}^{(1)}$	ε+1 + (ε+1)(ε+1)* (ε+1)	1*
$R_{12}^{(1)}$	0 + (ε+1)(ε+1)* 0	1*0
$R_{21}^{(1)}$	∅+∅(ε+1)*(ε+1)	∅
$R_{22}^{(1)}$	ε+0+1+∅(ε+1)* ∅	ε + 0 + 1

$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^*R_{2j}^{(1)}$		
$R_{11}^{(2)}$	1* + 1*0(ε+0+1) ∅	1*
$R_{12}^{(2)}$	1*0 + 1*0 (ε+0+1)*(ε+0+1)	1*0(0+1)*
$R_{21}^{(2)}$	∅+(ε+0+1) (ε+0+1)*∅	∅
$R_{22}^{(2)}$	ε+0+1+(ε+0+1) (ε+0+1)* (ε+0+1)	(0 + 1)*

A = {1,2,3}

Base	**Indução 1**	**Indução 2**	**Indução 3**
$R_{11}^{(0)}$	$R_{11}^{(1)}$	$R_{11}^{(2)}$	$R_{11}^{(3)}$
$R_{12}^{(0)}$	$R_{12}^{(1)}$	$R_{12}^{(2)}$	$R_{12}^{(3)}$
$R_{21}^{(0)}$	$R_{21}^{(1)}$	$R_{21}^{(2)}$	$R_{21}^{(3)}$
$R_{22}^{(0)}$	$R_{22}^{(1)}$	$R_{22}^{(2)}$	$R_{22}^{(3)}$
$R_{13}^{(0)}$	$R_{13}^{(1)}$	$R_{13}^{(2)}$	$R_{13}^{(3)}$
$R_{31}^{(0)}$	$R_{31}^{(1)}$	$R_{31}^{(2)}$	$R_{31}^{(3)}$
$R_{33}^{(0)}$	$R_{33}^{(1)}$	$R_{33}^{(2)}$	$R_{33}^{(3)}$
$R_{23}^{(0)}$	$R_{23}^{(1)}$	$R_{23}^{(2)}$	$R_{23}^{(3)}$
$R_{32}^{(0)}$	$R_{32}^{(1)}$	$R_{32}^{(2)}$	$R_{32}^{(3)}$



$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$$

Assim,

i. Se 1 é estado inicial e {2,3} são terminais então, $L(R) = R_{12}^{(3)} \cup R_{13}^{(3)}$

ii. Se 3 é estado inicial e {2} é terminal então, $L(R) = R_{32}^{(3)}$

iii. Se 2 é estado inicial e {1,2,3} são terminais então, $L(R) = R_{21}^{(3)} \cup R_{22}^{(3)} \cup R_{32}^{(3)}$

"A construção de uma expressão regular para definir a linguagem de qualquer DFA é surpreendente complicada"

(*Hopcraff, p.98*)

Indução:

a. O caminho não passa por ***k*** e assim tem-se no máximo $R_{ij}^{(k-1)}$;

b. O caminho passa por ***k*** pelo menos uma vez, neste:

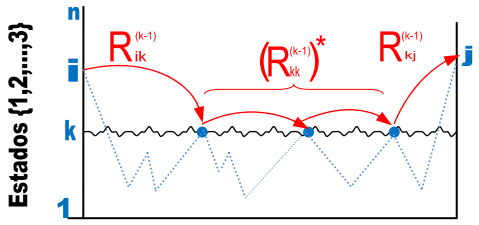


Figura 2 – Um caminho cujo rótulo está na linguagem da expressao regular $R_{ij}^{(k)}$

Combinando os itens (a) e (b), tem se:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^*R_{kj}^{(k-1)}$$ Método de construção Indutiva (primeiro método)

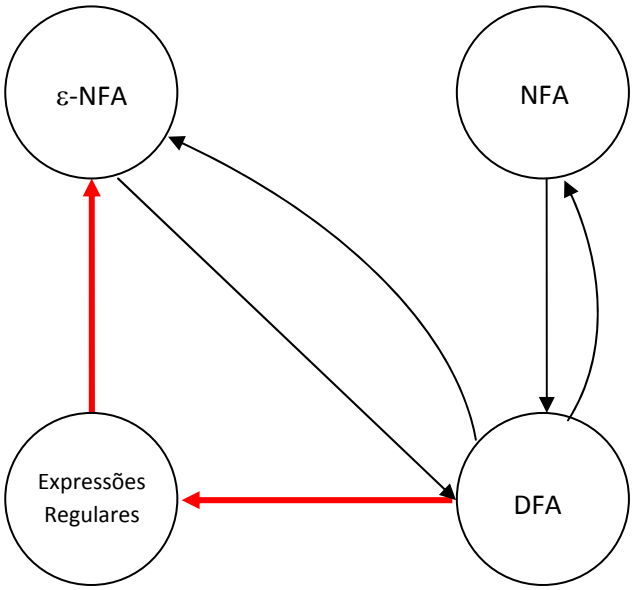


Figura 3 – Plano para mostrar a equivalencia entre ass quatros notações diferentes para linguagens regulares

- (Segundo Método) Procedimento de "eliminação de estados". Em primeira vista, este método parece ser mais eficiente e mais fácil que o método anterior.
- O método de Construção Indutiva para converter um DFA em uma expressão regular sempre funciona. Perceba também que ele realmente não depende de o autômato ser determinístico, e poderia ter sido igualmente aplicado a um NFA ou mesmo a um ε-NFA. Porém, a construção de "expressão regular" é dispendiosa.

Segundo Método: procedimento de "Eliminação de Estados"

- A abordagem de construção de "expressões regulares" que vamos aprender agora envolve a "eliminação de estados".
- Quando eliminamos um estado ***s***, todos os caminhos que passam por ***s*** não mais existem no autômato.
- Quando eliminamos um estado ***s*** do autômato original sua linguagem não deve mudar. Seja então o estado ***q*** que antecede o estado ***s***, e seja o estado p o estado que sucede o estado s. Para a linguagem não mudar após a eliminação do estado s, deveremos introduzir um "arco apropriado" que sai de ***q*** e vá até ***p***.
- Tendo em vista que o rótulo desse novo arco pode agora envolver strings, em vez de símbolos isolados, e pode haver até mesmo um numero infinito de tais strings, não podemos simplesmente listar os strings como um rótulo. Felizmente, há um modo simples e finito de representar todos esses strings: usar uma expressão regular.
- Desse modo, o método de "Eliminação de Estados" vai transformando os "símbolos isolados" que cada arco do autômato original progressivamente em "expressões regulares" mais complexas. Assim, somos levados a considerar que têm expressões regulares como rótulos.
- A linguagem do autômato original será então, a união sobre todos os caminhos desde o estado inicial até um estado de aceitação da linguagem formada por "concatenação" das linguagens das "expressões regulares ao longo desse caminho".

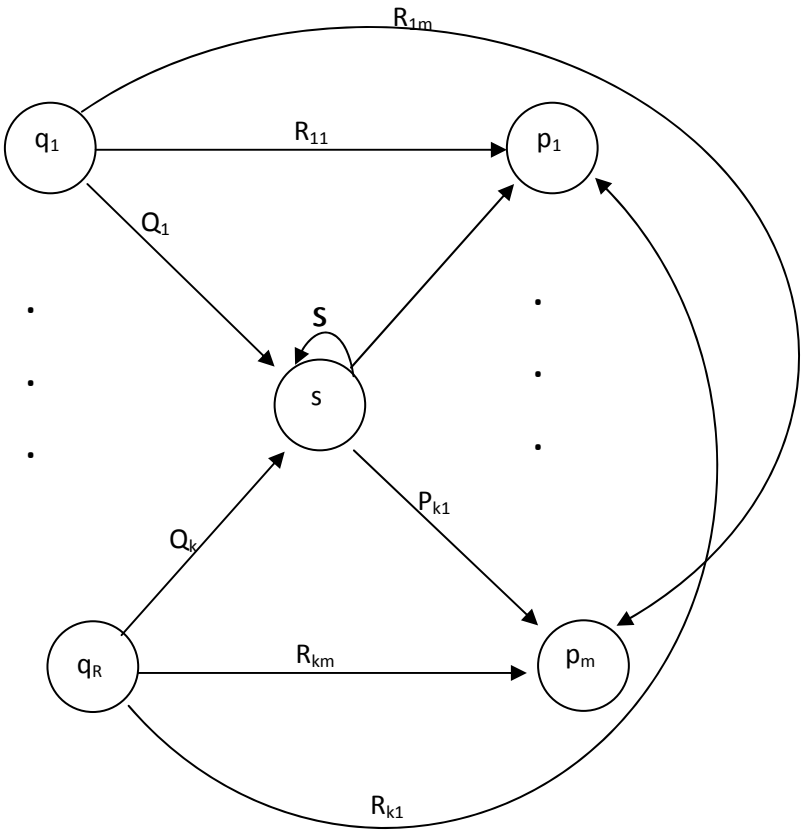


Figura 4 – Um estado S prestes a ser eliminado

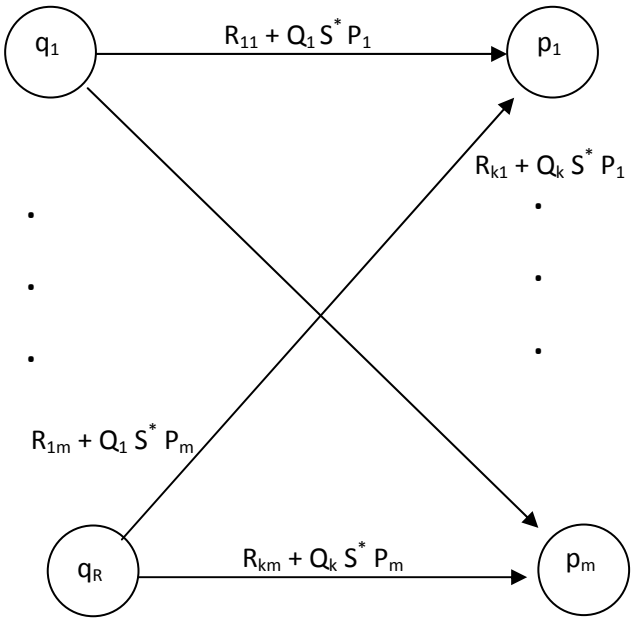


Figura 5 – Resultado daa eliminação do estado s da figura 4

Onde,

Q_i ... "expressão regular" que rotula ao arco q_1;

P_i ... rótulo de um loop em ***s*** (***S*** também é uma expressão regular);

R_{ij} ... "expressão regular" no arco de q_i até p_i para todo ***i*** e todo ***j***. Note que alguns desses arcos podem não existir no autômato e, nesse caso, tomaremos a expressão nesse arco como ∅.

Assim, a estratégia para construir uma expressão regular a partir de um autômato finito é:

1. Para cada "estado de aceitação" ***q***, aplique o processo de redução anterior para produzir um autômato equivalente com "rótulos de expressões regulares" nos arcos. Elimine todos os estados, exceto ***q*** e o estado inicial q_0.
2. Se $q \neq q_0$, ficaremos com um autômato de dois estados semelhante ao da figura 6. A expressão regular para os strings aceitos pode ser descrita de várias maneiras. Uma delas é (R + S U* T)* SU*.

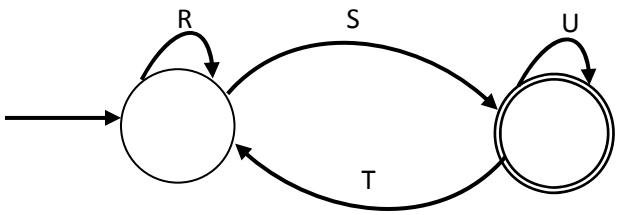


Figura 6 – Um automato genérico de dois estados

(R + S U* T)* SU*

Figura 7 – Uma poossível "expressão regular" para um autômato genérico de dois estados

3. Se o estado inicial também for um estado de aceitação, então também devemos executar uma eliminação de estados no autômato original que nos livre de todos os estados, exceto o estado inicial. A expressão regular que denota os strings que ele aceita é R*.

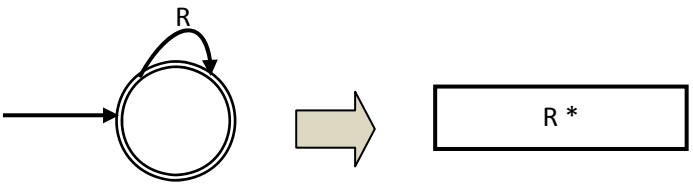


Figura 8 – Um automato genérico de um estados

4. A expressão regular desejada é a "soma (união)" de todas as expressões derivadas dos autômatos reduzidos corespondentes a cada estado de aceitação, pelas regras (2) e (3).

Exemplo:

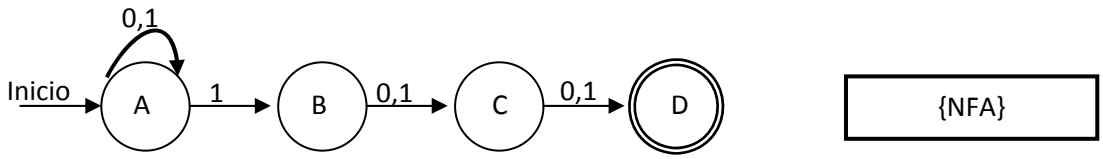


Figura 9 – Um automato que ainda não ocorreu a eliminação estados

Passo 1: Substituir os rótulos "0,1" pela "expressão regular" equivalente "0+1". Observe que ainda não ocorreu a eliminação de estados.

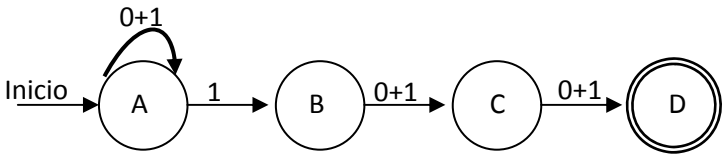


Figura 10 – Um automato que substitituiu os rótulos por expressão regular

Passo 2: Eliminar o estado B. $R_{AC} + Q_{AB} + S^* P_{BC} = \varnothing + 1\varnothing^* (0+1) = 1\varepsilon(0+1) = 1(0+1)$

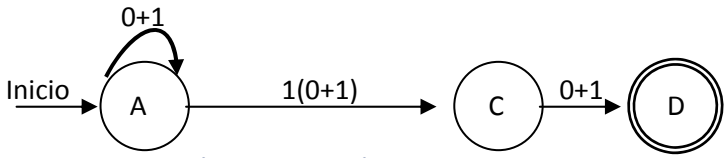


Figura 11 – Um automato que eliminnou um estado

Passo 3: Eliminar os estados C e D em reduções separadas.

Passo 3a: Eliminar o estado C. $R_{AD} + Q_{AC} + S^* P_{CD} = \varnothing + 1\ (0+1)\ \varnothing^*(0+1) = 1(0+1)(0+1)$

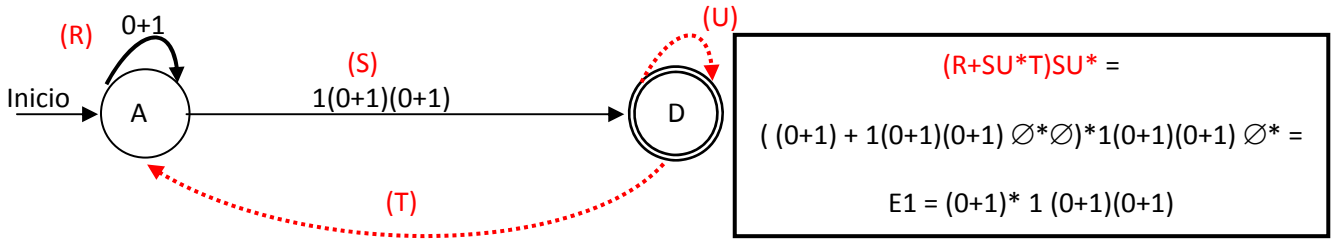


Figura 12 – Um automato que eliminnou o segundo estado

Passo 3b: Eliminar o estado D. $R_{CE} + Q_{CD} + S^* P_{CE} = \varnothing + (0+1)\ \varnothing^*\varnothing = \varnothing$

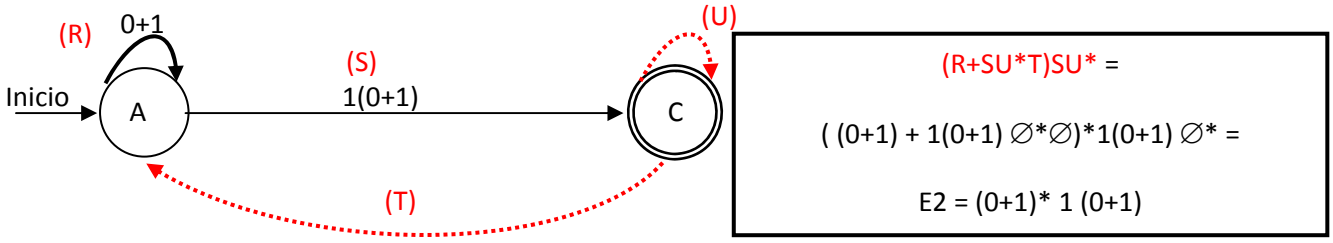


Figura 13 – Um automato que eliminnou o terceiro estado

Passo 4: A solução será $E_1 + E_2$ ou $E_1 \cup E_2$. Logo,

$$(0+1)^*\ 1\ (0+1)(0+1) + (0+1)^*\ 1\ (0+1)$$

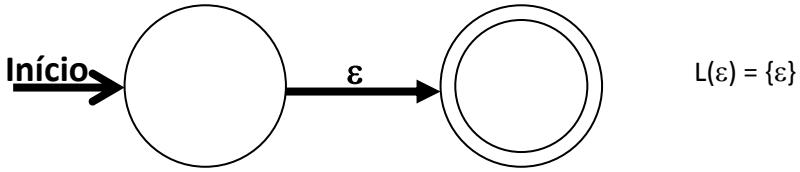
(Solução Final)

Conversão de Expressões Regulares em Autômatos

- Definição recursiva para Expressões Regulares

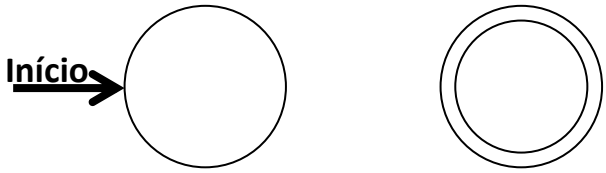
Base:

1. L(ε) = {ε} e L(∅) = ∅
2. L(a)= {a}
3. L … qualquer linguagem



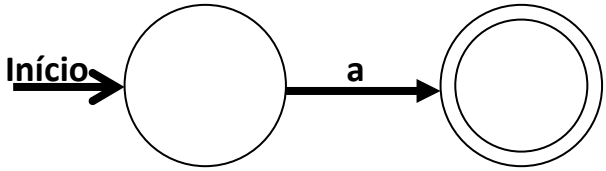
L(ε) = {ε}

Figura 14 –Automato L(ε) = {ε}



L(∅) = ∅

Figura 15 –Automato L(∅) = {∅}

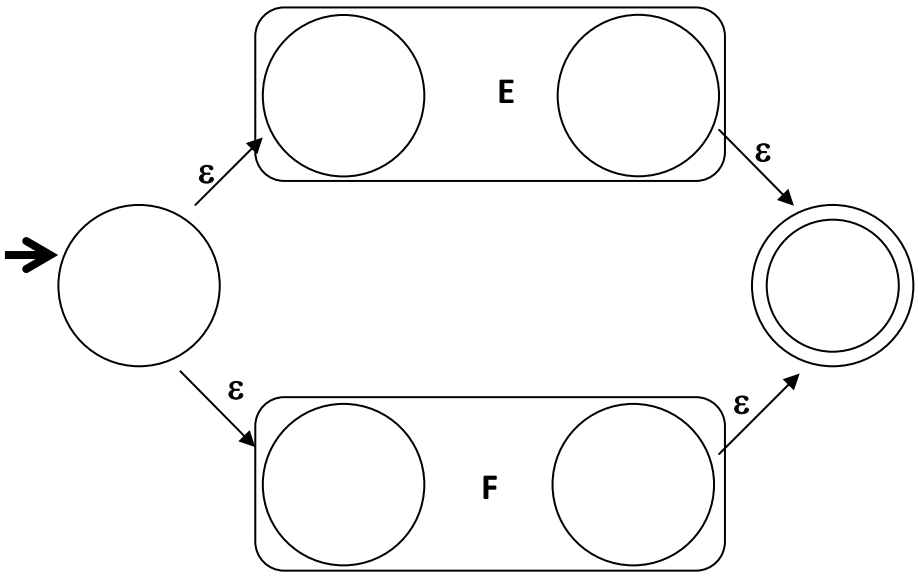


L(a) = {a}

Figura 16 –Automato L(a) = {a}

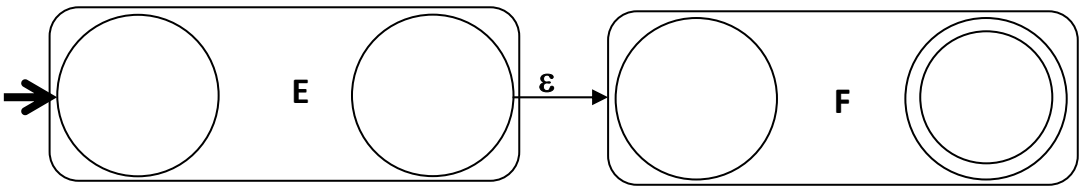
Indução:

1. L(E+F) = L(E) + L(F)
2. L(EF) = L(E) L(F)
3. L(E*) = (L(E))*
4. L((E)) = L(E)



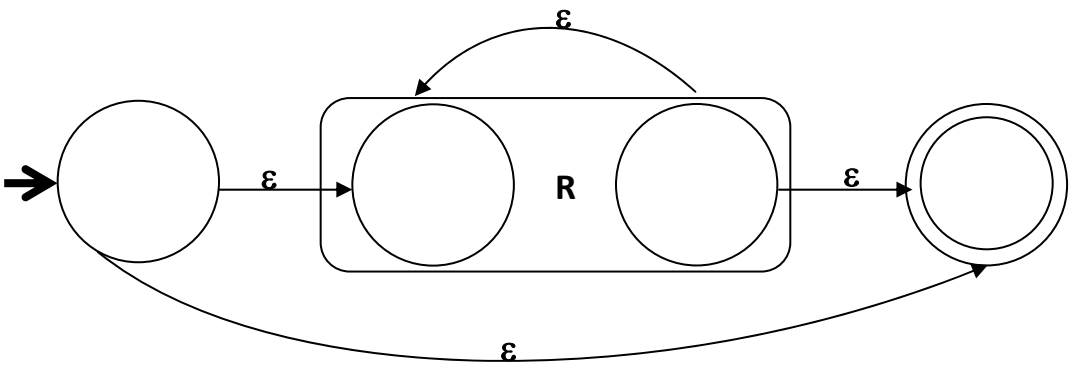
L(E+F) = L(E) + L(F)

Figura 17 –Automato L(E+F) = L(E) + L(F)



L(EF) = L(E)L(F)

Figura 18 –Automato L(EF) = L(E)L(F)



L(E*) = (L(E))*

Figura 19 –Automato L(E*) = (L(E))*

Exemplo:

0+1

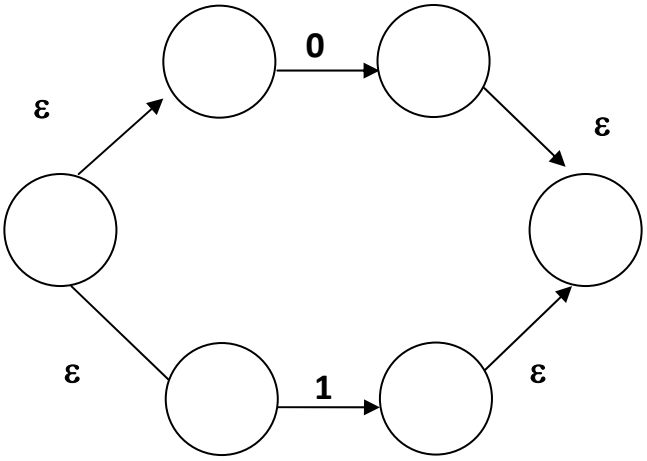


Figura 20 –Automato 0+1

(0+1)*

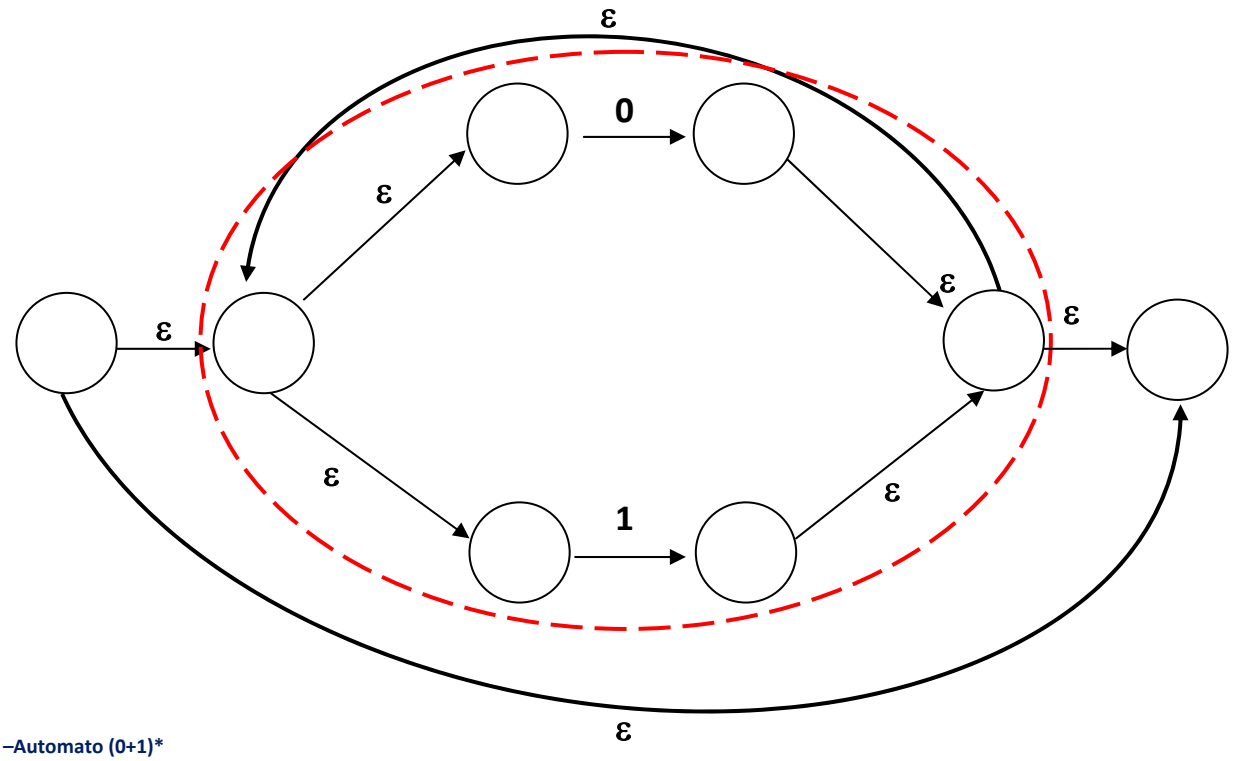


Figura 21 –Automato (0+1)*

(0+1)*1(0+1)

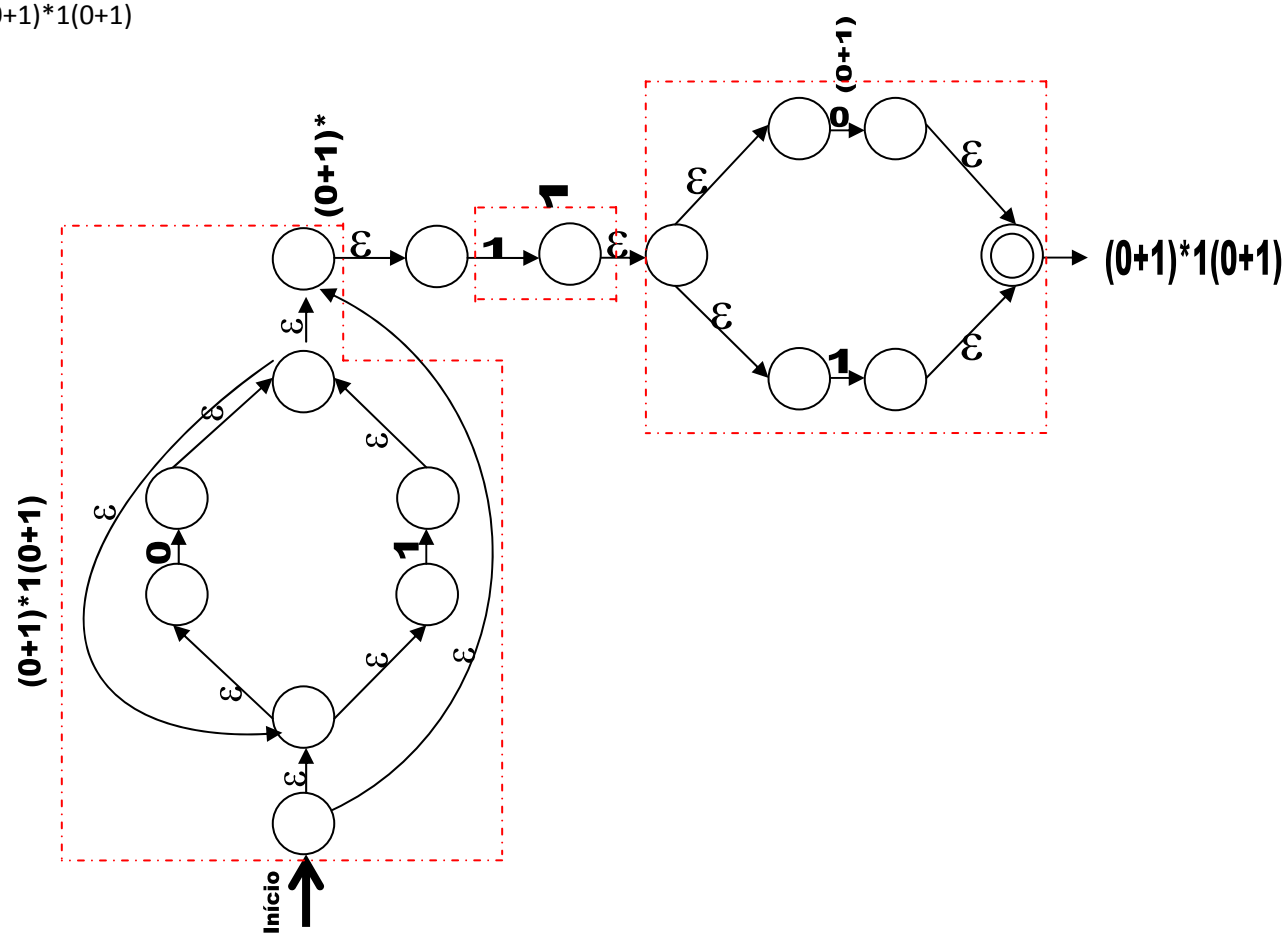


Figura 22 –Automato (0+1)*1(0+1)

O Teste de uma Lei Algébrica para Expressões Regulares

Agora, podemos enunciar o teste para saber se uma lei para expressões regulares é ou não verdadeira. O teste para saber se ***E=F*** é verdadeira, onde ***E*** e ***F*** são duas expressões regulares com o mesmo conjunto de variáveis, é:

1. Converter ***E*** e ***F*** nas expressões regulares concretas ***C*** e ***D***, respectivamente, substituindo cada variável por um símbolo concreto.
2. Teste se L(C) = L(D). Em caso afirmativo então E=F é uma lei verdadeira e, se não, a "lei" é falsa.

Teorema: O teste anterior identifica corretamente as leis verdadeiras para expressões regulares (e somente estas). (pgs.. 128 até 138, livro Hapcroft)

Exemplo 01 (lei lógica verdadeira): Iremos demonstrar a lei lógica (L+M)* = (L*M*)*. Se substituirmos as variáveis ***L*** e ***M*** por símbolos concretos (ou constantes) ***a*** e ***b***, respectivamente, obteremos as expressões regulares (a+b)* e (a*b*)*. É fácil verificar que essas duas expressões denotam a linguagem com todos os strings de ***a'***s e ***b***'s. Desse modo, as duas expressões concretas denotam a mesma linguagem, e a lei é válida.

Exemplo 02 (lei lógica falsa): Tentemos demonstrar que ***L+ML = (L+M)***L é uma lei lógica. Se escolhermos os símbolos ***a*** e ***b*** para as variáveis ***L*** e ***M***, respectivamente, teremos as duas expressões regulares concretas ***a+BA*** e ***(a+b)a***. Porém, as linguagens dessas expressões não são iguais. Por exemplo, o string ***aa*** está na segunda, mas não na primeira. Desse modo, esta lei é falsa.

Nota Importante: Na demonstração de leis lógicas para "expressões regulares" (e somente estas) as demonstrações que partem do "particular" para o "geral" são sempre válidas. Observe que na aritmética isto já não acontece, por exemplo, partindo-se de que 1+2 = 2+1 não se pode concluir, a partir dai, que x+y = y+x.

Observação: Considere uma álgebra estendida de expressões regulares que alem dos operadores da união (+), da concatenação (.) e do fechamento de kleene (*) contenha também o operador (∩). De modo bastante interessante, a adição de ∩ aos três operadores de expressões regulares não aumenta o conjunto de linguagens que podemos descrever, ou seja, a linguagem regular é "fechada" para o operador interseção. Entretanto, este operador torna inválido o teste de leis algébricas como descrito pelo teorema da página anterior.

Resumo do Capítulo 3

- **Expressões Regulares**: Essa notação algébrica descreve exatamente as mesmas linguagens que os autômatos finitos: as linguagens regulares. Os operadores de expressões regulares são união, concatenação (ou "ponto") e fechamento (ou "estrela").
- **Equivalência de Expressões Regulares e Autômatos Finitos**: Podemos converter um DFA em uma expressão regular por meio de uma "construção indutiva" na qual as expressões corespondentes aos rótulos de caminhos que podem passar por conjuntos cada vez maiores de estados são construídos. De modo alternativo, podemos usar um procedimento de "eliminação de estados" para elaborar a expressão regular correspondente a um DFA. No outro sentido, podemos construir recursivamente um ε-NFA a partir de expressões regulares, e depois converter o ε-NFA em um DFA, se desejar-mos. Veremos também mais adiante que existe um **algoritmo geral** que permite minimizar autômatos finitos.
- **A álgebra dde expressões regulares**: As expressões regulares obedecem a muitas leis algébricas da aritmética, embora existam diferenças. A união e a concatenação são associativas, mas só a união é comutativa. A concatenação se distribui sobre a união; A união é idempotente.

Referencias para o capítulo 3

- A ideia de expressões regulares e a prova de sua equivalência para autômatos finitos é o trabalho de S.C.kleene (1956).
- S.C.kleene, “Representation of events in nerve nets and finit automata”, em C.E.Shannon e J. McCarthy, Autmata Studies, Princetan Univ. Press,, 1956, pp. 3-42.
- O teste para a identidade de expressões regulares que trata variáveis como constantes foi desenvolvido por J.Gischer (1984).
- J.L. Gischer, STAN-CS-TR-84-1033(1984).
- O DFA convenccional foi proposto independentemente, em diversas variações semelhantes, por D.A. Huffman (1954), poor G.H.Mealy (11955) e por E.F.Moore(1956).

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Linguagens Regulares

Índice

Figuras

Linguagens Regulares

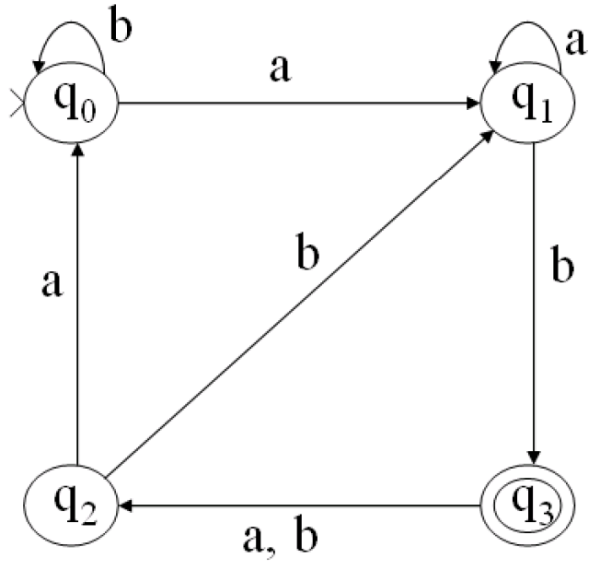
Propriedades das Linguagens Regulares

Explorar as propriedades das linguagens regulares nos conduzirá a algumas propriedades interessantes:

1. Estabelecer um modo de provar que certas linguagens não são regulares (tema do bombeamento);
2. Estabelecer as chamadas propriedades de fechamento, por exemplo, a interseção de duas linguagens regulares também é regular;
3. Estabelecer as propriedades de decisão, ou seja, um algoritmos para decidir se dois autômatos definem a mesma linguagem. Uma consequência de nossa habilidade para resolver essa questão é que podemos "minimizar autômatos", isto é, encontrar um equivalente a um dado autômato que tenha o mínimo de estados possível.

O *Pumping Lemma* para Conjuntos Regulares

Análise Informal:



ababb**aaab** ✓

ababbabbaaab ✓

…

a **(bab)**^{i} baaab ✓

u **v** **w**

Subcadeia "bombeada" (pumped)

Principio da Casa de Pombo

Princípio da Casa de Pombo: Coloquialmente, se você tem mais pombos que compartimentos em um pombal e cada pombo voa para algum compartimento, então deve haver pelo menos um compartimento com mais de um pombo. Em nosso exemplo, os "pombos" são as sequencias de ***n*** bits, e os "compartimentos" são os estados. Tendo em vista que existem menos estados que sequências, um estado deve ser atribuído a duas sequências.

Teorema 1: (O Lema de Bombeamento para linguagens regulares). Seja ***L*** uma linguagem regular. Então, existe uma constante ***n*** (que depende de ***L***) tal que, para todo string ***w*** em ***L*** tal que $|w| \geq n$, podemos dividir ***w*** em três strings, w = xyz, tais que:

1. $y \neq \varepsilon$
2. $|xy| \leq n$

Para todo $k \geq 0$, o string xy^kz também está em L.

Interpretação: Sempre poderemos encontrar um string não vazio ***y*** não muito longe do início de ***w*** que pode ser "bombardeado", ou seja, poderemos repetir ***y*** qualquer numero de vezes, ou excluí-lo (o caso de k=0), mantendo o string resultante na linguagem ***L***.

Exemplo 1: Se $1010 \in L$ então,

1010, 101010, 101010, etc $\in$ L, para ***L*** uma linguagem regular.

Prova: (por construção)

H_1: suponha que ***L*** seja regular, ou seja, L=L(A) parra algum DFA A;

H_2: suponha que ***A*** tenha n estados;

H_3: seja ***w*** qualquer string de aceitação de comprimento ***n*** ou maior, por exemplo, $w = a_1 a_2 \ldots a_m$, onde m $m \geq n$ e cada a_i é um símbolo de entrada;

H_4: para i = 1,2,...,n+1, define-se estado P_i como $\hat{\delta}(q_0, a_1 a_2 \ldots a_i)$, onde δ é a função de transição de ***A***, q_0 é o estado inicial de ***A*** e P_i o estado em que ***A*** se encontra depois de ler os primeiros ***i*** símbolos de ***w***. Nesta convenção observe que $P_0 = q_0$.

- Assim, aplicando o princípio da casa de Pombo, não é possível que os ***n+1*** diferentes $P_{i's}$ de H_4 para i=1,2,...,n+1 sejam distintos, pois só existem ***n*** estados diferentes (há mais pombos que compartimentos).

Deste modo, pode-se encontrar dois inteiros diferentes ***i*** e ***j***, com $1 < i < j \leq n+1$, tais que $P_i = P_j$. Agora, pode-se dividir w = xyz como a seguir:

1. $x = a_1\, a_2 \ldots a_{i-1}$
2. $y = a_i\, a_{i+1}\, a_{i+2} \ldots a_j$
3. $z = a_{j+1}\, a_{j+2} \ldots a_m$

- ou seja, ***x*** nos leva a P_i uma vez; ***y*** nos leva de P_i de volta a P_i (pois P_i também é P_j) e ***z*** é o restante de ***w***.
- Note que ***x*** e ***z*** podem ser vazios, mas ***y*** não pode ser vazio, pois ***i*** é estritamente menor que ***j***.
- Agora considere o que acontece se o autômato ***A*** recebe a entrada xy^kz para qualquer $k \geq 0$:
  - Se k = 0 então w = xz é aceita sem repetir nenhum estado;
  - Se k > 0 então $w = xy^kz$ é aceita repetindo P_i k vezes. (c.q.d.)

Teorema de bombeamento para Conjunto Regulares

Lema: Seja ***G*** grafo de estados de um AFD M com ***k*** estados. Então qualquer caminho de comprimento ***k*** em ***G*** contém um ciclo.

Corolário: Seja ***G*** o grafo de estados de um DFA M com ***k*** estados, e seja ***p*** um caminho de comprimento maior ou igual a ***k***. Então ***p*** pode ser decomposto em subtrajetórias ***x***, ***y*** e ***z*** (p=xyz), em que o comprimento de xy é menor ou igual a ***k*** e ***y*** é um ciclo.

Lema do Bombeamento: Seja *L* uma linguagem regular aceita por um DFA M com ***k*** estados. Seja ***w*** uma cadeia em ***L*** com comprimento maior ou igual a ***k***. Então ***w*** pode ser escrito como xyz, onde:

$$|xy| \leq k,\ |y| > 0 \text{ E } xy^iz \in L,\ \forall\, i \geq 0$$

$$\text{Se } w \in L \text{ e } |w| \geq k \text{ então, } \underbrace{w = xyz}_{1},\ \underbrace{|wy| \leq k}_{2},\ \underbrace{y \neq \varepsilon}_{3} \text{ e } \underbrace{xy^iz \in L}_{4}$$

Se uma linguagem é realmente uma linguagem regular então, não há outra saída senão ter que "engolir" o lema do bombeamento.

Isto implica também a todo DFA M tem que "engolir" o mesmo lema!

$L = \{a^ib^i \mid i \geq 0\}$

$L = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$

Demonstração: Assuma, por absurdo, que L é regular. Sendo assim então, ***l*** deverá ser aceita por algum AFD de ***k*** estados. O teorema de bombeamento, para este caso, toma a seguinte forma:

Se $w \in L$ e $|w| \geq k$ então, $\underbrace{w = xyz}_{1}$, $\underbrace{|wy| \leq k}_{2}$, $\underbrace{y \neq \varepsilon}_{3}$ e $\underbrace{xy^iz \in L}_{4}$

Desta forma, se for encontrado um string ***w*** que decomposto de acordo com o teorema de bombeamento não pertencer a ***L***, chegar-se-á a um absurdo – que por exclusão – fica assim demonstrado que ***L*** não é regular!

1. **Escolher o string (deve estar contida em *L*)**: $w = a^kb^k$ ($w \in L$ e $|w|=2k$)
2. **Desenvolver algebricamente o string escolhido**: $w = xyz = a^kb^k = a^ia^ja^{k-i-j}b^k$ para $i + j \leq k$ e $j > 0$

Exemplo 2: i = 2, j = 3 e k = 6 ($i+j = 2+3 = 5 < k = 6$ e $j > 0$)

$a^6b^6 = a^2a^3a^1b^6 = a^6b^6$ (ok!)

Se $w = \underbrace{a^i}_{x} \quad \underbrace{a^j}_{y} \quad \underbrace{a^{k-i-j}b^k}_{z}$ então, $xy^2z = a^ia^ja^ja^{k-i-j}b^k = \underbrace{a^ia^ja^{k-i-j}}_{a^k}a^ja^kb^k = a^ka^ja^kb^k$

Logo, $xy^2z = a^ka^ja^kb^k \notin L$ e L não é regular.

$L = \{z \in \{a,b\}^* \mid |z|$ seja um quadrado perfeito$\}$

Demonstração:

Se $w \in L$ e $|w| \geq k$ então, $\underbrace{w = xyz}_{1}$, $\underbrace{|wy| \leq k}_{2}$, $\underbrace{y \neq \varepsilon}_{3}$ e $\underbrace{xy^iz \in L}_{4}$

3 é um ciclo

Teorema do Bombeamento

1. **Escolher o String**: $w = xyz \mid |w| = k^2$ (quadrado perfeito)
2. **Manipular**: Se $w = xyz$ então $|w| = k^2$

Teorema bombeamento: 2 $|xy| \leq k$ e 3 $y \neq \varepsilon \Rightarrow 0 < |u| \leq k$

Logo,

$xy^2z \leq |xyz| + |z| \leq k^2 + k$

$xy^2z \leq \underbrace{k^2+k}_{\text{Resultado aplicado ao possível DFA que poderia representar a linguagem } \boldsymbol{L} \text{ considerada se esta fosse realmente uma linguagem regular.}} < \underbrace{\text{quadrado perfeito } (k^2 + 2k + 1)}_{\text{Resultado exigido pela linguagem considerada, quem em princípio (antes da demonstração ter sido considerada) não sabemos se ela é realmente regular ou não.}}$

Exemplo 3: $L_{pr} = \{w = 1^n \mid n \text{ é primo}\}$. Seja então, provar que esta linguagem não é regular.

- Se $w \in L_{pr}$ e $|w| \geq k+2$ então vale o lema de bombeamento, ou seja,

$$\underbrace{w = xyz}_{1}, \underbrace{|wy| \leq k}_{2}, \underbrace{y \neq \varepsilon}_{3} \text{ e } \underbrace{xy^iz \in L_{pr}}_{4}$$

Onde,

k ... número de estados do autômato que "aceitaria" tal linguagem;

$|w|=p \geq k+2$ ... aqui "p" é um número primo (tem que haver tal primo p, pois existe uma infinidade deles).

Logo, $w = 1^p$. **(a)**

Seja $|y| = m$ **(b)** {lembre-se que y não pode ser ε}

Então, $|xz| = |xyz| - |y| = p - m$ **(c)**

Se L_{pr} é regular deve valer também o lema do bombeamento, ou seja $xy^{p-m}z$ também deve pertencer a esta linguagem. Porém,

$|xt^{p-m}z| = |xz| + (p-m)|y| = (p-m)m(1+m)$ **(d)**

Conclusão: Parece que $|xt^{p-m}z|$ não é primo, pois tem dois fatores (m+1) e (p-1). Entretanto devemos ainda provar que nenhum dos fatores (p-m) ou (1+m) é igual a 1, pois nestes casos teríamos um número primo. Entretanto,

i. Para linguagens regulares necessariamente temos $y \neq \varepsilon$ e assim $|y| = n \geq 1$ ou $1 + m \geq 2$;

ii. $p-m > 1$ é realmente verdadeiro pois, $p \geq k+2$ (foi escolhido assim) e $m \leq k$, ou seja, $m = |y| \leq |xy| \leq k$. Assim, para $\{p \geq k+2$ no pior caso $p = k+2$ e $m=k$, $m \leq k\}$, ou seja $p-m = (k+2)-k$ ou $p-m>1$.

Máquinas de Moore

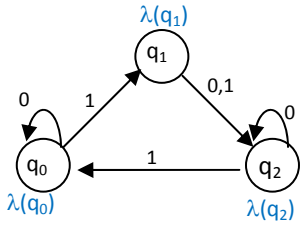
- Máquina de Moore M é uma sêxtupla $(Q, \sum, \Delta, \delta, \lambda, q_0)$

  $\sum$: alfabeto de entrada {0,1}

  Δ: alfabeto de saída {a,b}

Entrada 0110

Saída aa



δ	0	1
→ q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_0

λ	Valor
$\lambda(q_0)$	a
$\lambda(q_1)$	a
$\lambda(q_2)$	b

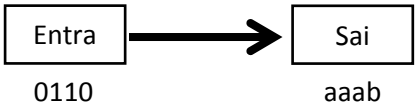
q_0 0110

0 q_0 110 $\lambda(q_0)$

01 q_1 10 $\lambda(q_0)$ $\lambda(q_0)$

011 q_2 0 $\lambda(q_0)$ $\lambda(q_0)$ $\lambda(q_1)$

0110 q_2 $\lambda(q_0)$ $\lambda(q_0)$ $\lambda(q_1)$ $\lambda(q_2)$
a a a b



Máquinas de Mealy

q_0 0110

0 q_0 110 $\lambda(q_0,0)$

01 q_1 10 $\lambda(q_0,0)$ $\lambda(q_0,1)$

011 q_2 0 $\lambda(q_0,0)$ $\lambda(q_0,1)$ $\lambda(q_1,1)$

0110 q_2 $\lambda(q_0,0)$ $\lambda(q_0,1)$ $\lambda(q_1,1)$ $\lambda(q_2,0)$

a b b b

λ	0	1
q_0	a	b
q_1	a	b
q_2	b	a

0110 abbb

Automata finitos bidirecionais

Teorema 2: Se L é aceita por um 2AFD, então L PE uma linguagem regular.
Ou seja:
Um 2 AFD é equivalente a um AFD ...

Máquinas de Moore: saídas associadas ao estado.

Máquinas de Mealy: saídas associadas à transição.

Teorema 3: Se $M_1 = (Q, \sum, \Delta, \delta, \lambda, q_0)$, é uma máquina de Moore, então existe uma máquina de Mealy M_2 equivalente, desde que ignoremos a primeira saída da máquina de Moore.

Teorema 4: **A classe das linguagens regulares é fechada sob complementação.** Se ***L*** é uma linguagem regular sobre $\sum$, então $\bar{L} = \sum^* - L$ é regular.

Teorema 5: **A classe das linguagens regulares é fechada sob intersecção.** Se L_1 e L_2 são linguagens regulares, então $L_1 \cap L_2$ é linguagem regular.

Substituições: Exemplos e Fechamento

Exemplo 1: $\sum = \{0,1\}$ $\Delta = \{a,b\}$

$f(0) = a$, $f(1) = b^* \Rightarrow f(010) = f(0)\, f(1)\, f(0) =$ **ab*a**

Teorema 6: **A classe das linguagens regulares é fechada sob substituições.** Se ***L*** é uma linguagem regular sobre $\sum$, então *f(L)* é regular (sobre Δ).

Y. Bar-Hilel, M. Perles e E. Shamir [1961]

Teorema 7: **A classe das linguagens regulares é fechada sob homomorfismos.**

Y. Bar-Hilel, M. Perles e E. Shamir [1961]

Teorema 8: **A classe das linguagens regulares é fechada sob homomorfismos inversos.**

S. Ginsburg e G. Rose [1963]

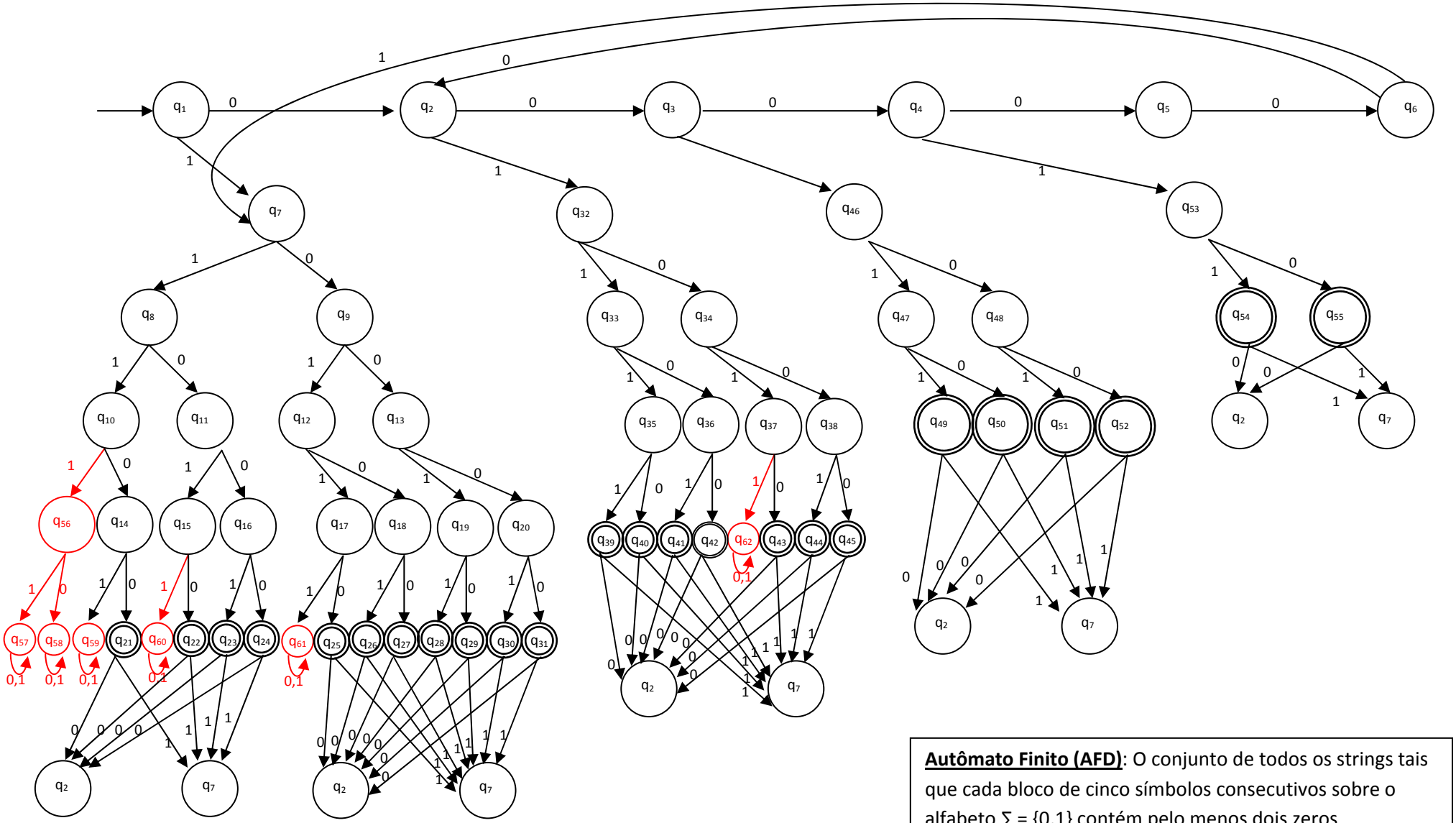


Figura 1 – AFD – sobre o alfabeto ∑ = {0,1}

Autômato Finito (AFD): O conjunto de todos os strings tais que cada bloco de cinco símbolos consecutivos sobre o alfabeto ∑ = {0,1} contém pelo menos dois zeros

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Gramáticas Livre de Contexto

Índice

Figuras

Gramáticas

Definição Formal: Uma gramática consiste em uma ou mais variáveis que representam classes de strings, isto é, linguagens. Há vários tipos de gramáticas e apenas para citar algumas tem-se a gramática regular e a gramática livre de contexto.

Por isto, há três maneiras, por exemplo, de se expressar uma linguagem regular:

- Através de um autômato finito determinístico (máquina de estados);
- Através de expressões regulares;
- Através de uma gramática regular.

Objetivo: estudar as gramáticas regulares e em seguida as gramáticas livres de contexto.

$G = (V, \sum, P, S)$

- S → Símbolo inicial S e V
- P → Conjunto de produções
- $\sum$ → Conjunto finito de símbolos terminais para $V \cap \sum = \varnothing$
- V → Conjunto finito V de símbolos não terminais

"A linguagem L(G) gerada por G consiste de todas as cadeias sobre ∑ derivadas de S"

Exemplo: uma gramática para gerar palíndromos sobre {a,b}

$G = (V, \sum, P, S)$

$V = \{S\}$

$\sum = \{a,b\}$

$P = \{S \rightarrow a, S \rightarrow b, S \rightarrow \varepsilon, S \rightarrow aSa, S \rightarrow bSb\}$

$S = S$ (S: cabeça; ε: campo)

Gerando palíndromos: {ε, a, b, aεa, aaa, aba, bεb, bab, bbb, ababa, ...}

Observações:

1. $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$ (simplificação de notação)

   $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, …, $A \rightarrow \alpha_n$

2. Uma "produção unitária" é uma produção da forma $A \rightarrow B$, onde tanto A quanto B são variáveis.

Definições dos Termos de uma Gramática

Como vimos há 4 componentes básicos importantes para a descrição gramatical de uma linguagem:

G = (V, ∑, P, S)

V ≡ Conjunto finito de variáveis, também chamada às vezes não terminais ou categorias sintáticas. Cada variável representa uma linguagem, isto é, um conjunto de strings (símbolos em letras maiúsculas);

∑ ≡ Conjunto finito de símbolos que formam strings da linguagem que está sendo definida. Chamamos este alfabeto de terminais ou de símbolos terminais (símbolos em letras minúsculas);

S ≡ Uma das variáveis (S ∈ V) que representa a linguagem que está sendo definida; ela é chamada símbolo de início;

P ≡ Conjunto finito de produções ou regras que representam a definição recursiva de uma linguagem. Casa produção consiste em:

Variável	→	string (símbolos + variáveis)
Cabeça da produção	símbolo de produção	corpo da produção

(a) Cabeça da produção – uma variável que sendo (parcialmente) definida pela produção;

(b) O símbolo da produção →

(c) Corpo da produção – Um string de zero ou mais terminais e variáveis. O corpo da produção representa um modo de formar strings na linguagem da variável da cabeça. Para fazê-lo, deixam-se os terminais inalterados e substitui-se cada variável do corpo por qualquer string conhecido por estar na linguagem variável.

Derivações que utilizam uma Gramática

- Aplicam-se as **produções** de uma Gramática Livre de Contexto (CFG[1]) para inferir que certos strings estão na linguagem de uma certa variável. Há duas abordagens para essa inferência:
  a. Por inferência recursiva, considerada também a abordagem mais convencional, que é utilizar as regras do corpo para a cabeça;
  b. Por derivação, na qual se utiliza as produções da cabeça para o corpo. É nesta abordagem que utiliza-se o símbolo de derivação (⇒).

Exemplo: E → I | E + E | E * E | (E) (quatro operações)

I → a |b | Ia | Ib | I0 | I1 (seis operações)

Por inferência recursiva	Por derivação
I → a | b Tornam-se os strings que sabe-se estar na linguagem a b I → I0 b0 I → I0 b00	I ⇒ I0 ⇒ I00 ⇒ b00 I → a | b E ⇒ E * E ⇒ I * E ⇒ a * E ⇒ a * (E) ⇒ a * (E + E) ⇒ a*(I+E)⇒ a*(a + E) ⇒ a * (a+1) ⇒ a*(a+I0) ⇒ a*(a+I00) ⇒ a*(a+b00) $E \overset{*}{\Rightarrow}$ a*(a+b00)

Inferência por Derivação

- Suponha que G = (V, ∑, P, S) seja gramática livre de contexto (CFG).
- Seja αAβ um string de terminais e variáveis, sendo A uma variável. Assim, α e β são strings em (VUT)* e A está em V.
- Seja A → j uma produção de G

Assim dizemos que:

$\alpha A\beta \overset{\Rightarrow}{G} \alpha j\beta$ ou $\alpha A\beta \Rightarrow \alpha j\beta$

Neste caso, G está subtendida

[1] Context-free language

- Observe que uma <u>etapa de derivação</u> substitui qualquer variável em qualquer lugar no string pelo corpo de uma de suas produções.
- Pode-se estender o relacionamento $\Rightarrow$ para representar zero, uma ou muitas etapas de derivação, de maneira muito semelhante ao modo como a função de transição δ de um autômato finito foi estendido para $\hat{\delta}$. No caso de derivações, usamos um * parra denotar "zero ou mais etapas", como a seguir.

Inferência por Derivação

<u>Base</u>: Para qualquer string α de terminais e variáveis, dizemos que $\alpha \overset{*}{\Rightarrow}_G \alpha$. Isto é, qualquer string deriva de si próprio.

<u>Indução</u>: Se $\alpha \overset{*}{\Rightarrow}_G \beta$ e $\beta \Rightarrow_G \gamma$, então $\alpha \overset{*}{\Rightarrow}_G \gamma$. Isto é, se α pode se tornar β por meio de zero ou mais etapas, e se mais uma etapa nos leva de β para γ, então α pode se tornar γ. Em outras palavras, $\alpha \overset{*}{\Rightarrow} \beta$ significa que existe uma sequencia de strings $\gamma_1, \gamma_2, ..., \gamma_n$, parra algum $n \geq 1$, tal que

1. $\alpha = \gamma_1$,
2. $\beta = \gamma_n$, e
3. Para I – 1, 2, …., n-1, temos $\gamma_i \Rightarrow \gamma_{i+1}$.

Se a gramática G é subentendida, então usamos $\overset{*}{\Rightarrow}$ em lugar de $\overset{*}{\Rightarrow}_G$.

A linguagem de uma Gramática

Se G = (V, ∑, P, S) é uma gramática livre de contexto (CFG), a linguagem de G, denotada por LG=(G), é o conjunto de strings terminais que têm derivações desde o símbolo de início, ou seja,

$$L(G) = \{w \text{ em } T^* \mid S \overset{*}{\Rightarrow}_G w\}$$

Se uma linguagem L é a linguagem de alguma gramática livre de contexto, então L é dita uma linguagem livre de contexto (CFL).

Derivações mais à esquerda e mais à direita

- Para restringir o numero de escolhas que temos na derivação de um string, muitas vezes é útil exigir quem em cada etapa, a variável mais à esquerda seja substituída por um de seus corpos de produção. Tal derivação é chamada derivação mais à esquerda, e indicaremos que uma derivação é mais à esquerda as relações $\overset{*}{\Rightarrow}_{lm}$ e $\Rightarrow_{lm}$ w para uma ou muitas etapas, respectivamente.

Exemplo:

E → I | E + E | E * E | (E) (quatro produções)

I → a |b | Ia | Ib | I0 | I1 (seis produções)

$E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E \Rightarrow_{lm} a * (E) \Rightarrow_{lm} a * (E + E) \Rightarrow_{lm} a * (I + E) \Rightarrow_{lm} a * (a * E) \Rightarrow_{lm} a * (a + I) \Rightarrow_{lm} a * (a + I0) \Rightarrow_{lm}$ a * (a + I00) $\Rightarrow_{lm}$ a * (a + b00)

ou

$E \overset{*}{\Rightarrow}_{lm} a * (a + b00)$

- De modo semelhante, é possível exigir que em cada etapa a variável mais à direita seja substituída por um de seus corpos. Nesse caso chamamos a derivação mais à direita e utilizamos os símbolos $\overset{*}{\Rightarrow}_{rm}$ e $\Rightarrow_{rm}$ para indicar uma muitas etapas de derivação mais à direita, respectivamente.

$E \Rightarrow_{rm} E * E \Rightarrow_{rm} E * (E) \Rightarrow_{rm} E * (E + E) \Rightarrow_{rm} E * (E + I) \Rightarrow_{rm} E * (E + I0) \Rightarrow_{rm} E * (E + I00) \Rightarrow_{rm} E * (E + b00) \Rightarrow_{rm} E * (I + b00) \Rightarrow_{rm}$ E * (a + I00) $\Rightarrow_{rm}$ E * (a + b00) $\Rightarrow_{rm}$ a * (a + b00)

ou

$E \overset{*}{\Rightarrow}_{rm} a * (a + b00)$

Árvores de Análise Sintática

- Há uma representação em árvore para derivações que se mostrou extremamente útil. Essa árvore nos mostra claramente como os símbolos de um string de terminal estão agrupados em substrings, cada um dos quais pertence à linguagem de uma das variáveis da gramática.
- **Utilidade das árvores de Análise Sintática**: em um compilador, a estrutura em árvore do programa-fonte facilita a conversão desse programa-fonte em código executável, permitindo que funções recursivas simples executem esse processo de conversão.
- Pode ser demonstrado matematicamente que a árvore de análise sintática e sua existência está intimamente ligada à existência de derivações (de todos os tipos) e de inferências recursivas.

Construindo árvores de Análise Sintática

Dada uma gramática livre de contexto G=(V,T,P,S). As árvores de análise sintática para G são árvores com as seguintes condições:

1. Cada nó interior é rotulado por uma variável em V.
2. Cada folha é rotulada por variável, um terminal ou ε. No entanto, se a folha for rotulada por ε, ela deve ser o único filho de seu pai.
3. Se um nó interior é rotulado por A e seus filhos são rotulados por X_1, X_2, ..., X_k, respectivamente, a partir da esquerda, então A → $X_1X_2...X_k$ é uma produção em P. Observe que o único momento em que um dos X's pode ser ε é quando esse é o rotulo do único filho, e se A → ε é uma produção de G.

O Resultado de uma Árvore de Análise Sintática

- Se for analisada as folha de qualquer árvore de análise sintática e se estas folhas forem concatenados a partir da esquerda, ter-se-á um string chamado resultado da árvores, que é sempre um string derivado da variável raiz. O resultado da raiz também denominado de produto pode ser da raiz possui sustentação de demonstração matemática.

Exemplo:

E → I | E + E || E * E | (E) (quatro produções)

I → a |b | Ia | Ib | I0 | I1 (seis produções)

Árvore de análise sintática mostrando que a*(a+b00) está na linguagem livre de contexto considerada.

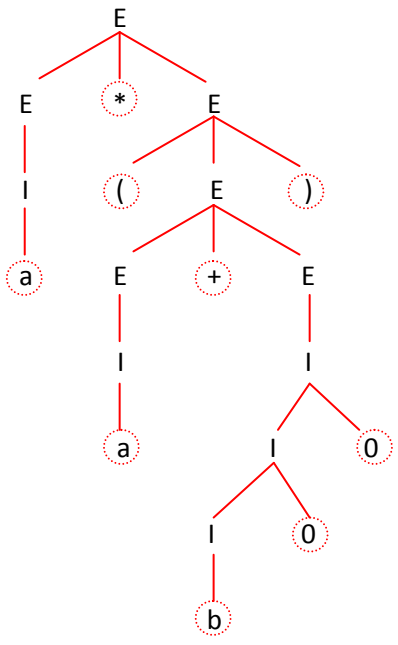


Figura 1 – Árvore de análise sintática de a*(a+b00)

Inferência, Derivações e Árvores de Análise Sintática

- Cada uma das ideias introduzidas até agora para descrever o funcionamento de uma gramática livre de contexto fornece essencialmente os mesmos fatos sobre as produções de strings. Ou seja, dada uma gramática G = (V,T,P,S), mostram-se que os seguintes itens são equivalentes.

1. O procedimento de inferência recursiva determina que o string de terminais w está na linguagem da variável A.
2. $A \overset{*}{\Rightarrow} w$, ou seja, inferência por derivação.
3. $A \overset{*}{\underset{lm}{\Rightarrow}} w$, ou seja, inferência por derivação mais à esquerda.
4. $A \overset{*}{\underset{rm}{\Rightarrow}} w$, ou seja, inferência por derivação mais à direita.
5. Existe uma árvore de análise sintática com raiz A e resultado w.

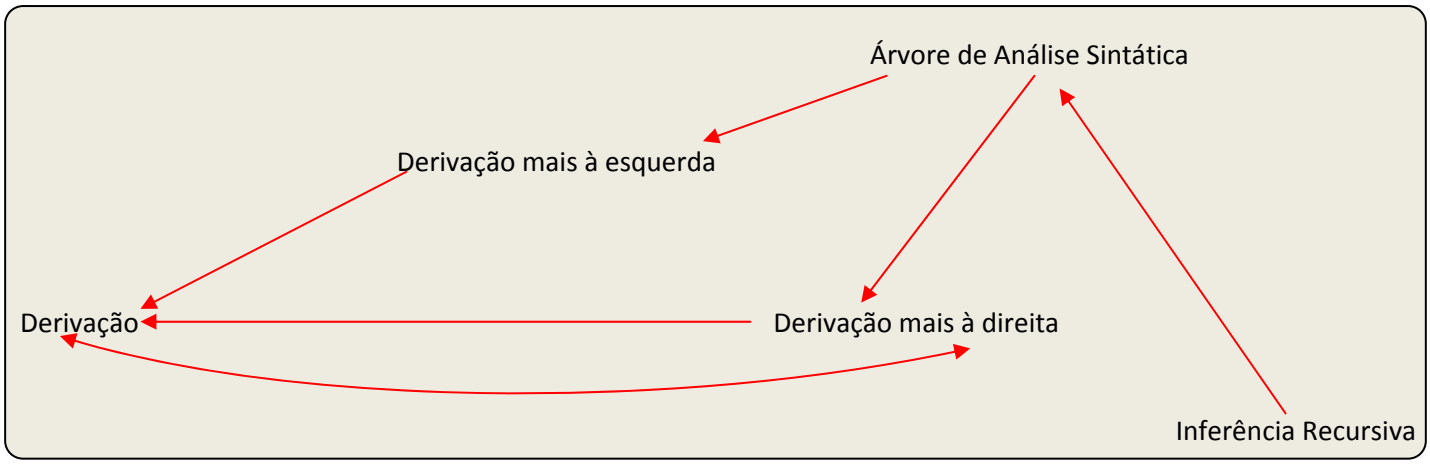


Figura 2 – Provando a equivalência de certas declarações sobre gramáticas

Teorema 1:	Seja G = (V,T,P,S) uma gramática livre de contexto (CFG). Se o procedimento de inferência recursiva nos diz que o string de terminal w está na linguagem da variável A, então existe uma árvore de análise sintática com raiz A e resultado w. (pg. 198-19, Hopcraft).
Teorema 2:	Seja G = (V,T,P,S) uma gramática livre de contexto (CFG), e suponha que exista uma árvore de análise sintática com raiz rotulada pela variável A e com resultado w, onde w está em T*. Então, existe uma derivação mais à esquerda $A \overset{*}{\underset{lm}{\Rightarrow}}$ na gramática G.
Teorema 3:	Seja G = (V,T,P,S) uma gramática livre de contexto (CFG), e suponha que exista uma árvore de análise sintática com raiz rotulada pela variável A e com resultado w, onde w está em T*. Então, existe uma derivação mais à direita $A \overset{*}{\underset{rm}{\Rightarrow}}$ na gramática G.
Teorema 4:	Seja G = (V,T,P,S) uma gramática livre de contexto (CFG), e suponha que exista uma derivação $A \overset{*}{\Rightarrow} w$, onde w está em T*. Então, o procedimento de inferência recursiva aplicado a G determina que w está na linguagem da variável A.

Aplicações das Gramáticas Livres de Contexto

1. A gramática livre de contexto foi proposta primeiro como um método de descrição para linguagens naturais por Chomsky.

   N. Chomsky, “Three models for the description of language”, IRE Trans. On Information Theory 2:3 (1956), pp. 113-124.

2. Contudo, à medida que os usos de conceitos definidos recursivamente se multiplicaram em Ciência da Computação, se acentuou a necessidade de se usar as CFG’s como um meio para descrever instâncias desses conceitos.
3. As gramáticas são usadas para descrever linguagens de programação. Mais importante ainda, existe um modo mecânico de transformar a descrição da linguagem na forma de uma gramática CFG em um analisador sintático, o componente do compilador que descobre a estrutura do programa-fonte e representa esta estrutura por uma árvore de análise sintática.

Ambiguidade em Gramáticas e Linguagens

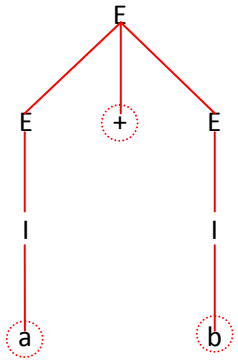
- Como as gramáticas podem ser utilizadas para dar estruturas a programas e documentos, supõe-se facilmente que uma gramática deva então, determinar de forma única uma estrutura para cada string em sua linguagem. Entretanto, nem toda gramática fornece estruturas únicas.

Exemplo 01:

E → I | E + E || E * E | (E) (quatro produções)

I → a |b | Ia | Ib | I0 | I1 (seis produções)

1. $E \underset{lm}{\Rightarrow} E + E \underset{lm}{\Rightarrow} I + E \underset{lm}{\Rightarrow} a + E \underset{lm}{\Rightarrow} a + I \underset{lm}{\Rightarrow} a + b$
2. $E \underset{rm}{\Rightarrow} E + E \underset{rm}{\Rightarrow} E + I \underset{rm}{\Rightarrow} E + b \underset{rm}{\Rightarrow} I + b \underset{rm}{\Rightarrow} a + b$



Nota 01: De fato, ambas as derivações produzem a mesma árvore de analise sintática, se for aplicada a construção dos teoremas 02 e 03 enunciados na página 9.

Exemplo 02: aplicado sobre a mesma gramática do exemplo 01:

1. $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow I + E * E \Rightarrow a + E * E \Rightarrow a + I * E \Rightarrow a + a * E \Rightarrow a + a * I \Rightarrow a + a * a$
2. $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow E + E * I \Rightarrow E + E * a \Rightarrow E + I * a \Rightarrow E + a * a \Rightarrow I + a * a \Rightarrow a + a * a$

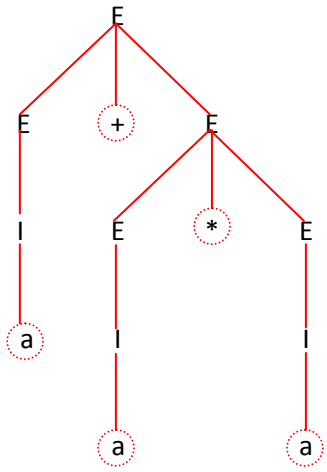
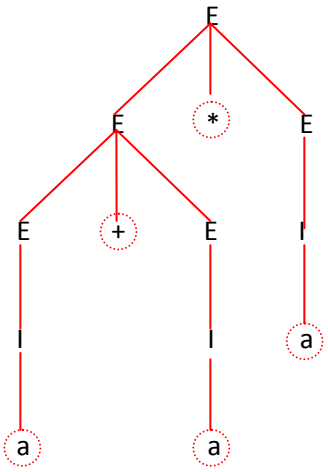
1 – Derivação mais a esquerda	2 – Derivação mais a direita
	

Figura 3 – Duas arvorres de análise sintatica com resultado a+a*a, demonstrando a ambiguidade de nossa gramática de expressões

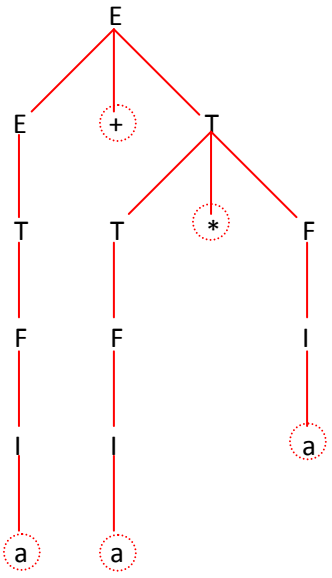
- Os dois exemplos anteriores sugerem que não é uma multiplicidade de derivações que causa ambiguidades, mas sim a existência de duas ou mais árvores de análise sintática.

"Uma gramática livre de contexto G=(V,T,P,S) é ambígua se existe pelo menos um string ***w*** em ***T**** para o qual podemos encontrar duas árvores de análise sintática diferentes, cada qual com uma raiz identificada como ***S*** e um resultado ***w***.
Se cada string tiver no máximo uma árvore de análise sintática na gramática, a gramática é não-ambígua."

Teorema 5:	(Sobre a indecibilidade de gramáticas ambíguas): é um fato surpreendente que não existe absolutamente nenhum algoritmo geral que possa nos informar se uma gramática livre de contexto (CFG) é ambígua. (Livro do Hopcraft, cap. 9 sobre Indecibilidade)
Teorema 6:	Existem linguagem livres de contexto (CFG) ambíguas, cuja a remoção da ambiguidade é impossível.
Teorema 7:	Para cada gramática G=(V,T,P,S) e string ***w*** em ***T****, ***w*** tem duas árvores de análise sintática distintas se e somente se ***w*** tem duas derivações mais à esquerda distintas a partir de ***S***.

Teorema 8:	A mesma validade imposta pelo teorema 07, mas consideranddo0se agora duas derivações mais à direita.
Heurística:	Felizmente, a situação não é tão horrível na prática. Para os tipos de construções que aparecem nas linguagens de programação comuns, existem técnicas bem conhecidas para eliminar ambiguidade.

Gramática Ambígua		Gramática Não-Ambígua
E → I | E + E || E * E | (E) I → a |b | Ia | Ib | I0 | I1	Gramáticas Equivalentes No sentido de que representam a mesma linguagem	F → I | (E) T → F | T * F E → T | E + T I → a | b | Ia | Ib | I0 | I1



1. A linguagem de expressões geradas por esta gramática é realmente não ambígua.
2. Embora essa gramática seja ambígua, existe outra gramática para a mesma linguagem que é não-ambígua.
3. Uma linguagem livre de contexto ***L*** é dita inerente ambígua se todas as suas gramáticas são ambíguas.
4. Se até mesmo uma única gramática de ***L*** for não ambígua, ***L*** será uma linguagem não ambígua.

Figura 4 – A única árvore de análise sintática para a + a * a

O fato dessa gramática ser não ambígua talvez esteja longe de ser óbvio. Entretanto, sempre é possível encontrar uma explicação porque de uma gramática não ser ambígua, se ela realmente não for ambígua (Hopcraft, pg. 225 e 226).

Aplicação Prática do Teorema 07

Teorema 7:	Para cada gramática G=(V,T,P,S) e string ***w*** em ***T****, ***w*** tem duas árvores de análise sintática distintas se e somente se ***w*** tem duas derivações mais à esquerda distintas a partir de ***S***.

Conclusão: Se uma gramática é ambígua sempre é possível encontrar um "string exemplo" para confirmar este fato.

Exemplo:

$$L = \{a^n b^n c^m d^n \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$$

Gramática Possível para a linguagem L:

S → AB |C
A → aAb | ab
B → cBd | cd
C → aCd | aDd
D → bDc | bc

Assim, por exemplo, o string aabbccdd tem duas derivações mais à esquerda:

1. $S \underset{lm}{\Rightarrow} AB \underset{lm}{\Rightarrow} aAbB \underset{lm}{\Rightarrow} aabbB \underset{lm}{\Rightarrow} aabbccdd$
2. $S \underset{lm}{\Rightarrow} C \underset{lm}{\Rightarrow} aCd \underset{lm}{\Rightarrow} aaDdd \underset{lm}{\Rightarrow} aabDcdd \underset{lm}{\Rightarrow} aabbccdd$

Logo, esta gramática é ambígua.

Entretanto, a prova de que todas as gramáticas para ***L*** devem ser ambíguas é bastante complexa e foge ao escopo de nosso estudo.

Resumo

Gramáticas Livres de Contexto: Uma CFG é um modo descrever linguagens por regras recursivas chamadas produções. Uma CFG consiste em um conjunto de variáveis, um conjunto de símbolos terminais, uma variável de inicio e produções. Cada produção consiste em uma variável de cabeça e um corpo que consiste em um string de zero ou mais variáveis e/ou terminais.

Derivações e Linguagens: Começando com o símbolo de início, derivamos strings de terminais substituindo repetidamente uma variável pelo corpo de alguma produção com essa variável na cabeça.

Derivações mais à esquerda e mais à direita: Todo o string na linguagem de uma gramática livre de contexto (CFG) tem pelo menos uma derivação mais à esquerda e pelo menos uma mais à direita.

Árvore de Análise Sintática: Uma árvore sintática é uma árvore que mostra os aspectos essenciais de uma derivação. Os nós internos são rotulados por variáveis, e as folhas são rotuladas por terminais ou por ε. Para cada nó interno, deve haver uma produção tal que a cabeça da produção seja o rótulo do nó, e que os rótulos de seus filhos, lidos da esquerda para a direita, fornecem o corpo desta produção.

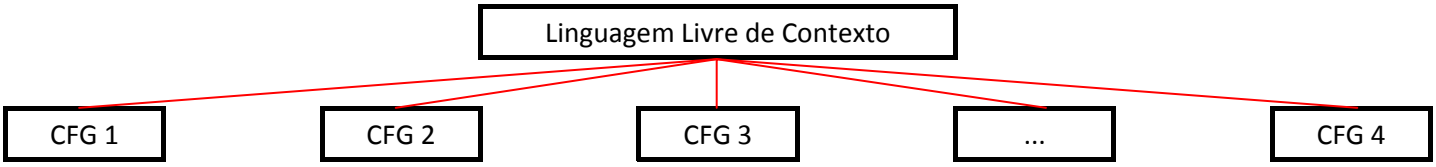
Equivalência entre árvores de análise sintática e derivações: Um string de terminais está na linguagem de uma gramática livre de contexto se e somente se ele é o resultado de pelo menos uma árvore de análise sintática. Desse modo, a existência de derivações mais à esquerda, derivações mais à direita e árvores de análise sintática são condições equivalentes, e cada uma define exatamente os strings na linguagem de uma CFG.

Gramáticas Ambíguas: Para algumas CFG's, é possível encontrar um string de terminais com mais de uma árvore de analise sintática ou, de forma equivalente, mais de uma derivação mais à esquerda ou mais de uma derivação mais à direita. Tal gramática é chamada ambígua.

Problema de Indecibilidade: É um fato surpreendente que não existe absolutamente nenhum algoritmo que possa nos informar se uma CFC é ambígua.
Existem linguagens livre de contexto que não tem nada além de CFG's ambíguas, para essas linguagens, a remoção da ambiguidade é impossível.
Felizmente, a situação não é tão horrível na pratica. Para os tipos de construção que aparecem nas linguagens de programação comuns, existem técnicas bem conhecidas para eliminar a ambiguidade.

Eliminação de Ambiguidades: para muitas gramáticas úteis é possível encontrar uma gramática não-ambígua que gere a mesma linguagem. Infelizmente, a gramática não-ambígua com frequência é mais complexa que a gramática ambígua mais simples para a linguagem.

Analisadores Sintáticos: A gramática livre de contexto é um conceito essencial para a implementação de compiladores e outros processadores de linguagens de programação.



Notas:

1. Dada uma linguagem livre de contexto(L) poder-se-á ter várias gramáticas livres de contexto (CFG's) para esta mesma linguagem. Isto é algo muito parecido ao fato de existir vários autômatos finitos para uma mesma linguagem regular. Digamos então, que para nossa linguagem livre de contexto existam N CFG's associadas. Neste caso, há duas situações a serem consideradas:
   (i) Todas as N CFG's são ambíguas, ou seja, esta linguagem é meramente ambígua**;**
   (ii) Há pelo menos 1 CFG não-ambígua. Neste caso, se até mesmo uma única gramática de ***L*** não ambígua, ***L*** será uma linguagem não-ambígua.

2. Não existe absolutamente nenhum algoritmo geral que possa nos informar se uma gramática livre de contexto é ambígua (demonstração matemática envolvendo o conceito de indecibilidade).

3. Dada uma CFG I, i=1,2,…,N, ou seja, uma gramática qualquer da linguagem ***L*** considerada, pode-se verificar, se ela é ambígua, caso sejam encontrados duas derivações à esquerda distantes (ou duas derivações à direita distintas) para um mesmo string da aceitação.

4. Entretanto, provar que todas as ***N*** gramáticas livres de contexto de uma dada linguagem ***L*** é ambígua é uma tarefa complexa e foge do escopo de nossas aulas, porém possível.

5. Felizmente é possível utilizar heurísticas para gramáticas livre de contexto ambíguas, para encontrar uma equivalente não-ambígua, caso esta exista. Entretanto, as gramáticas não-ambíguas ser mais complexas que as não-ambíguas.

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Segunda Parte do Curso de CT-200

Índice

Figuras

Tabelas

Segunda Parte do Curso de CT-200

- **Revisão sobre gramáticas e linguagens.**
  - A hierarquia de Chomsky e especial atenção às gramáticas livres de contexto
  - Simplificação de gramáticas livres de contexto e "Formas Normais de Chomsky"
- **Autômatos de Pilha e Linguagens Livres de Contexto (LLC)**
  1. O lema do bombeamento para LLC
  2. Propriedades de fechamento para LLC e Linguagens Regulares
  3. Propriedades de decisão para LLC e Linguagens Regulares (minimização)

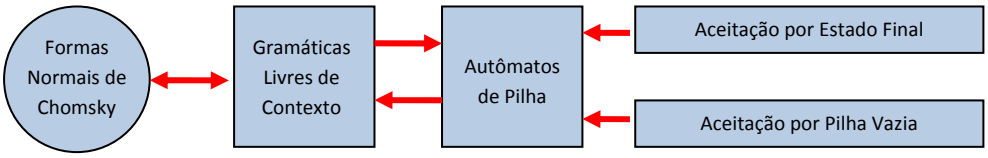


Figura 1 – Propriedades das GLC e Automatos de Pilha

- Máquinas de Turing (os autômatos são essenciais para o estudo dos limites da computação)
  1. O que o computador pode fazer, afinal?
     i. Este estudo é chamado de "decibilidade"
  2. O que o computador pode fazer de forma eficiente?
     i. Este estudo é chamado de "tratabilidade"

Essas máquinas são autômatos que modelam o poder dos computadores reais. Elas nos permitem estudar os problemas de "decibilidade" e os problemas "tratáveis"

A máquina de Turing é uma tupla de 7 valores semelhantes aos Autômatos de Pilha (4) Indecibilidade.

Resumo sobre Gramáticas

1. Gramática

$G = (V, \sum, P, S)$

- S → Símbolo inicial S e V
- P → Conjunto de produções
- $\sum$ → Conjunto finito de símbolos terminais para $V \cap \sum = \varnothing$
- V → Conjunto finito V de símbolos não terminais

Exemplo: gerador de palíndromos sobre {a,b}

G = (V, ∑, P, S)

V = {S} ..."Conjunto de variáveis" ou "Símbolos Não -Terminais"

∑ = {a,b} ... "Alfabeto de Terminais" ou "Símbolos Terminais"

P = ...{S → a, S → b, S → ε, S → aSa, S → bSb}

S = S ... Símbolo de início (sempre uma das variáveis)

2. $A \rightarrow \alpha_1, A \rightarrow \alpha_2, ..., A \rightarrow \alpha_n \Leftrightarrow A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

3. Se $\alpha \rightarrow \beta$ e $x\alpha y \in (V \cup \sum)^*$, então $x\beta y$ é <u>diretamente derivável</u> de $x\alpha y$, ou seja, $x\alpha y \Rightarrow x\beta y$.
   Se $\alpha_i \in (V \cup \sum)^*$ para i =1, ..., n e $\alpha_i \Rightarrow \alpha_{i+1}$ para i=1, ..., (n-1), então α_n é <u>derivável</u> de α_1, ou seja, $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$.
   $\alpha_1 \overset{*}{\Rightarrow} \alpha_n : \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$

4. Uma "produção unitária" é uma produção da forma A → B, onde tanto A quanto B são <u>variáveis</u>.

<u>Contra exemplo</u>: A → a

5. "A linguagem L(G) gerada por G consiste de todas as cadeias sobre ∑ derivados de S".

Propriedades de Linguagem Livres de Contexto

- Nossa tarefa na aula de hoje será **<u>simplificar</u>** as gramáticas Livres de Contexto (GLCs).
- Essas simplificações tornam mais fácil provar fatos sobre GLCs.

"Se uma linguagem é uma GLC, então ela tem uma gramática em alguma forma especial"

"O objetivo desta aula é mostrar que toda GLC (sem ε) é gerada por uma GLC na qual todas as produções têm a forma A → BC ou A → α, onde A, B e C são variáveis, e α é um terminal. Essa forma é chamada Forma Normal de Chomsky".

- São necessárias as seguintes simplificações preliminares:

A ordem delas influencia
- Eliminação de Recursão sobre símbolo inicial S;
- Eliminação de Regras ε;
- Eliminação de "Regras Encadeadas" ou "Produções Unitárias";
- Eliminação de Símbolos inúteis.

Eliminação da Recursão sobre S

Exemplo numérico:

G: S → aS | AB | AC
A → aA | ε
B → bB | bS
C → cC | ε

} G =(V,∑,P,S)

G: S' → S
S → aS | AB | AC
A → aA | ε
B → bB | bS
C → cC | ε

} G =(V',∑,P',S')

Algoritmo para Eliminar ε-produções

- Embora ε-produções sejam uma conveniência em muitos problemas de projetos de gramáticas, não são essenciais.
- É claro que sem uma produção que tenha um corpo ε, é impossível gerar o string vazio como um elemento da linguagem.
- Se a linguagem L tem uma GLC, então L-{ε} tem uma GLV sem ε-produções.
- Quais variáveis são anuláveis? Uma variável A é anulável se $A \overset{*}{\Rightarrow} \varepsilon$.

Algoritmo para Detectar as Variáveis Anuláveis

Seja G(V,∑,P,S) uma GLC, então pode-se encontrar todos os símbolos anuláveis de G pelo algoritmo iterativo a seguir:

Base: Se A → ε é uma produção de G, então A é anulável.

Indução: Se existe uma produção $B \rightarrow C_1 C_2 \ldots C_k$, onde cada C_i é anulável, então B é anulável. Observe que C_i deve ser uma variável para ser anulável, e assim só se deve considerar produções com "corpos" que contêm apenas variáveis.

Teorema 1:	Em qualquer gramática G, os únicos símbolos anuláveis são as variáveis encontradas pelo algoritmo anterior.

Construção de uma gramática sem ε-produções

Seja G=(V,∑,P,S) uma GLC, então para a construção de uma gramática G_L faça:

(i) Determine todos os símbolos anuláveis de G;

(ii) Coinstrua uma nova gramática $G_L = (V,\sum,P_L,S)$, cujo conjunto de produções P_L é determinada a seguir: "Para cada produção $A \rightarrow X_1 X_2 \ldots X_k$ de P, onde $k \geq 1$, suponha que m dos k valores X_i sejam símbolos anuláveis. A nova gramática G_L terá 2^m versões dessa produção, onde os X_i's anuláveis, em todas as combinações possíveis, estão presentes ou ausentes. Há uma única exceção: Se m = k, isto é, se todos os símbolos são anuláveis, então não incluímos o caso no qual todos os X_i's estão ausentes. Observe também que, se uma produção da forma $A \rightarrow \varepsilon$ está em P, não inserimos essa produção em P_L".

Teorema 2:	Se a gramática G_L é construída a partir de G pela construção anterior para eliminar ε-produções, então $L(G_L) = L(G)-\{\varepsilon\}$.

Eliminação de ε-produções

Exemplo numérico:

S → AB

A → aAA | ε

B → bBB | ε

(i) Símbolos Anuláveis

A e B são diretamente anuláveis (**Base**)

S é anulável, porque a produção S → AB possui corpo que consiste apenas em símbolos anuláveis (**Indução**)

{A,B,S} são anuláveis

(ii) Produções da G_L

S → AB S → AB | A | B	A → aAA (são iguais) A → aAA | aA | aA | a A → aAA | aA | a	B → bBB B → bBB | bB | b

Constituição de G_L

S → AB | A | B

A → aAA | aA | a

B → bBB | bB | b

Algoritmo para Eliminação de "Regras Encadeadas" ou "Produções Unitárias"

- Uma "produção unitária" é uma produção da forma A → B, onde tanto A quanto B são variáveis.
- Produções unitárias, por exemplo, são úteis para criar uma gramática não-ambígua para expressões aritméticas simples. Entretanto, produções unitárias podem complicar certas provas, e também introduzem etapas extras em derivações que tecnicamente não precisam estar lá.

(i) A técnica para a eliminação de produções unitárias que oferece a garantia de funcionar envolve encontrar primeiro todos os pares de variáveis A e B tais que "$A \overset{*}{\Rightarrow} B$" é obtida através somente de uma sequencia de produções unitárias.

(ii) Uma vez determinados todos esses pares, podemos substituir qualquer sequencia de etapas de derivações em que $A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow ... \Rightarrow B_n \Rightarrow \alpha$ por uma produção que utilize a "produção não-unitária" $B_n \Rightarrow \alpha$ por uma produção que utilize a "produção não-unitária" $B_n \rightarrow \alpha$ diretamente a partir de A, isto é, $A \rightarrow \alpha$

ou

"Devemos criar um novo conjunto de produções usando o primeiro elemento de um par unitário como a cabeça de uma produção em G_L e todos os corpos não unitários para o segundo elemento do par unitário como os corpos de produções em G_L".

Construção Indutiva dos pares (A,B) tais que $A \overset{*}{\Rightarrow} B$ através somente de produções unitárias

Base: (A,A) é um par unitário para qualquer variável A. Isto é, $A \overset{*}{\Rightarrow} A$ em zero etapas.

Indução: Suponha que descobrimos que (A,B) é um par unitário, e B → C é uma produção, onde C é uma variável. Então (A,C) é um par unitário.

Teorema 3: O algoritmo acima encontra exatamente os pares unitários para uma gramática livre de contexto G.

Para eliminar produções unitárias, prosseguimos como a seguir. Dada uma gramática livre de contexto G = (V,T,P,S), construa a G_1 = (V,T,P_1,S):

1. Encontre todos os pares unitários de G;
2. Para cada par unitário (A,B), adicione a P_1 todas as produções A → α, onde B → α é uma produção não-unitária em P. Observe que A = B é possível, desse modo, P_1 contém todas as produções não unitárias em P.

B → α é uma produção não-unitária em P. Observe que A = B é possível, desse modo, P_1 contém todas as produções não-unitárias em P.

Teorema 4: Se a gramática G_1 é construída a partir da gramática G pelo algoritmo descrito acima para eliminação de produções unitárias, então $L(G_1) = L(G)$.

Exemplo numérico:

Eliminação de "Regras Encadeadas" ou "Produções Unitárias"

G: I → a | b | Ia | Ib | I0 | I1

F → I | (E)

T → F | T*F

E → T | E+T

(i) A **Base** dos pares são **(E,E)**$_1$, **(T,T)**$_2$, **(F,F)**$_3$, **(I,I)**$_4$

A etapa **indutiva**

1. (E,E) e a produção E → T nos dão o par unitário (E,T) 5
2. (E,T) e a produção T → F nos dão o par unitário (E,F) 6
3. (E,F) e a produção F → I nos dão o par unitário (E,I) 7
4. (T,T) e a produção T → F nos dão o par unitário (T,F) 8
5. (T,F) e a produção F → I nos dão o par unitário (T,I) 9
6. (F,F) e a produção F → I nos dão o par unitário (F,I) 10

De (i) encontramos 10 pares unitários para um GLC G!!

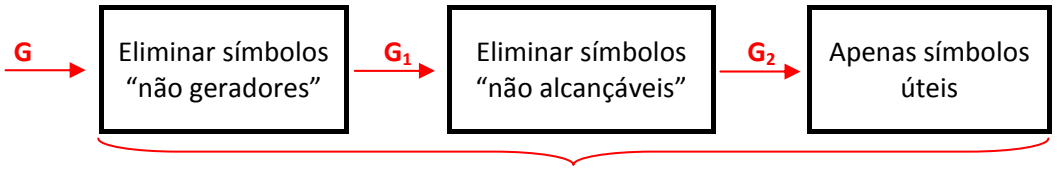
(ii)

n	Par	Produções
1.	(E,E)	E → E+T
2.	(T,T)	T → T*F
3.	(F,F)	F → (E)
4.	(I,I)	I → a | b | Ia | Ib | I0 | I1
5.	(E,T)	E → T*F
6.	(E,F)	E → (E)
7.	(E,I)	E → a | b | Ia | Ib | I0 | I1
8.	(T,F)	T → (E)
9.	(T,I)	T → a | b | Ia | Ib | I0 | I1
10.	(F,I)	F → a | b | Ia | Ib | I0 | I1

Tabela 1 – Pares unitários para um GLC G

Algoritmo para Eliminação de Símbolos Inúteis

- Dizemos que um símbolo X é "útil" para uma gramática G=(V,∑,P,S) se existe alguma derivação da forma $S \overset{*}{\Rightarrow} \alpha X \beta \overset{*}{\Rightarrow} w$, onde w está em ∑*. Observe que X pode estar em V ou ∑, e que a forma sentencial $\alpha X \beta$ poderá ser a primeira ou a ultima da derivação.
- Se X não é útil, dizemos que ele é inútil.
- As duas "ações necessárias" que um símbolo deve ser capaz de realizar para ser útil: "X gerador" e "X alcançável".
    1. Dizemos que X é "gerador" se $X \overset{*}{\Rightarrow} w$ para algum string de terminais w. Observe que todo terminal é gerador, pois w pode ser esse terminal propriamente dito, que é derivado por zero etapas.
    2. Dizemos que X é "alcançável" se existe uma derivação $S \overset{*}{\Rightarrow} \alpha X \beta$ para algum α e β.
- Um símbolo "útil" será ao mesmo tempo "gerador" e "alcançável".



Importantíssimo: A ordem não pode ser violada!

Teorema 5: Seja G=(V,∑,P,S) uma GLC, e suponha que L(G) ≠ ∅, isto é, G gera pelo menos um string. Seja $G_U=(V_U,\sum_U,P_U,S)$ a gramática que obtemos pelas seguintes etapas:

1. Primeiro, eliminamos símbolos que não são geradores e todas as produções que envolvem um ou mais desses símbolos. Seja $G_1 = (V_1,\sum_1,P_1,S)$ essa nova gramática. Observe que S deve ser gerador, pois supomos que L(G) tem pelo menos um string, e assim S não é eliminado.
2. Em segundo lugar, eliminamos todos os símbolos que não são alcançáveis na gramática G_1, produzindo G_U. Então G_U não tem símbolos inúteis, e $L(G_U) = L(G)$.

Cálculo de Símbolos Geradores e Alcançáveis

- Seja G=(V,∑,P,S) uma gramática. Para calcular os símbolos geradores de G, executamos a seguinte indução

  Base: Todo símbolo de T é sem dúvida gerador; ele gera a si mesmo.

  Indução: Suponha que exista uma produção A → α, e todo símbolo de α já seja conhecido como gerador. Então, A é gerador. Observe que essa regra inclui o caso em que α=ε; todas as variáveis que têm ε como corpo de uma produção seguramente são geradoras.

Teorema 6: O algoritmo anterior encontra todos os símbolos geradores de G e somente eles.

- Seja G=(V,∑,P,S) uma gramática. Para calcular os símbolos alcançáveis de G, executamos a seguinte indução:

  Base: S é sem dúvida alcançável.

  Indução: Suponha que descobrimos que alguma variável A é alcançável. Então para todas as produções com A na "cabeça", os símbolos dos "corpos" dessas produções são alcançáveis.

Teorema 7: O algoritmo anterior encontra todos os símbolos alcançáveis de G e somente eles.

Considere a gramática:

S → AB | a
A → b

$\alpha \Rightarrow \beta$
β é diretamente derivável de α
$\alpha \overset{*}{\Rightarrow} \beta$
β é diretamente derivável de α

(i) **a** e **b** geram a si mesmos, portanto **a** e **b** são geradores;

(ii) S gera **a** e **A** gera **b**, portanto S e A também são geradores;

(iii) B não gera a si mesmo nem os outros, portanto B **não é gerador**.

Se eliminarmos B, devemos eliminar a produção S → AB.

Assim,

S → a
A → b
} Primeiro eliminamos os símbolos que não são geradores

(iv) Apenas **S** e **a** são alcançáveis a partir de S.

Assim,

S → A } Só depois eliminamos os símbolos que não são alcançáveis.

Nota: A invenção da ordem "gerador → alcançável" dera problema" Veja exemplo:

S → AB | a

A → b

Base: S é alcançável

Indução: S → AB
S → a
} Tanto **AB** e **a** são alcançáveis

Resumo Geral sobre Simplificações de GLC

- Para converter qualquer GLC G em uma GLC equivalente que não tenha símbolos inúteis, ε-produções ou produções unitárias, é preciso ter um certo cuidado na ordem de aplicação das construções. Uma ordem segura é:
    1. Eliminar ε-produções.
    2. Eliminar produções unitárias.
    3. Eliminar símbolos inúteis.

Teorema 8:	Se G é uma GLC que gera uma linguagem que contém pelo menos um string de ε, então existe outra GLC G_F tal que $L(G_F) = L(G)-\{\varepsilon\}$ e G_F não tem ε-produções, produções unitárias ou símbolos inúteis. {A demonstração deste teorema, exige a sequencia 1 – 2 – 3}

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Forma Normal de Chomsky

Índice

Forma Normal de Chomsky

- Toda GLC não vazia sem ε tem uma gramática G em que todas as produções estão em uma dentre duas formas simples:
    1. A → BC, onde A, B e C, são todas variáveis, ou
    2. A → a, onde A é uma variável e **a** é um terminal.

Além disso, G não tem nenhum símbolo inútil. Dizemos que tal gramática está na forma normal de Chomsky, ou FNC.

- Para converter uma GLC numa FNC, o ponto de partida é o teorema 8, ou seja, a gramática não pode ter ε-produções, nem produções unitárias e nem símbolos inúteis, ou seja, tudo que vimos até o momento, nesta aula, será utilizado agora. Entretanto, os passos anteriores não são eficientes. É necessário, depois disto:

a) Organizar todos os corpos de comprimento 2 ou mais que consistem apenas em variáveis.
b) Desmembrar os corpos de comprimento 3 ou mais em uma cascata de produções, cada uma com um corpo consistindo em duas variáveis.
c) Além disso, obviamente, toda forma do tipo A → α é aceita diretamente.

- **Como construir o passo (a)?**

"Para todo terminal "a" que aparecer em um corpo de comprimento 2 ou mais, crie uma nova variável, digamos A. Essa variável terá apenas uma produção A → a. Agora, usamos A em lugar de **a** em todo lugar em que **a** aparecer em um corpo de comprimento 2 ou mais. Neste ponto, toda produção terá um corpo que será um úunicco terminal ou pelo menos duas variáveis e nenhum terminal".

- **Como construir o passo (b)?**

"Devemos desmembrar as produções $A \rightarrow B_1 B_2 \ldots B_k$, para $k \geq 3$, em um grupo de produções com duas variáveis em cada corpo. Introduziremos (k-2) novas variáveis, $C_1, C_2, \ldots, C_{k-2}$. A produção original é substituída pelas k-1 produções.

$A \rightarrow B_1C_1$
$C_1 \rightarrow B_2C_2,\ldots$
$C_{k-3} \rightarrow B_{k-2}C_{k-2}$
$C_{k-2} \rightarrow B_{k-1}C_k$

- **Como construir o passo (c)?**

Imediato!!!

1

E → I | E + E | E * E | (E)
I → a | b | Ia | Ib | I0 | I1

G = ({E,I}, T, P, E)
Símbolo Inicial

T = {+, *, (,), a, b, 0, 1}

Gramática Ambígua

Equivalência Gramatical

2

I → a | b | Ia | Ib | I0 | I1
F → I | (E)
T → F | T * F
E → T | E + T

Gramática Não–Ambígua
(Em geral, nesta gramática, há produções unitárias**).**

1. Eliminar ε-produções;
2. Eliminar produções unitárias;
3. Eliminar símbolos inúteis;
4. Forma Normal de Chomsky

Partiremos de 2

3 **Eliminação de Produções Unitárias**

Par	Produções
(E,E)	E → E+T
(T,T)	T → T*F
(F,F)	F → (E)
(I,I)	I → a | b | Ia | Ib | I0 | I1
(E,T)	E → T*F
(E,F)	E → (E)
(E,I)	E → a | b | Ia | Ib | I0 | I1
(T,F)	T → (E)
(T,I)	T → a | b | Ia | Ib | I0 | I1
(F,I)	F → a | b | Ia | Ib | I0 | I1

Esta gramática contém símbolos inúteis?

Não, sendo assim, eles não precisam ser eliminados.

3

E → E+T | T*F |(E) | a | b | Ia | Ib | I0 | I1
T → T*F |(E) | a | b | Ia | Ib | I0 | I1
F → (E) | a | b | Ia | Ib | I0 | I1
I → a | b | Ia | Ib | I0 | I1

Passo (a):

Oito terminais {+, *, (,), a, b, 0, 1} → cada um destes terminais aparece num corpo que não é único terminal

Consequência: introduzir 8 novas variáveis:

A → a

B → b

Z → 0

θ → 1

P → +

M → *

L → (

R →)

E → EPT | TMF |LER | a | b | IA | IB | IZ | Iθ

T → TMF |LER | a | b | IA | IB | IZ | Iθ

F → LER | a | b | IA | IB | IZ | Iθ

I → a | b | IA | IB | IZ | Iθ

A → a

B → b

Z → 0

θ → 1

P → +

M → *

L → (

R →)

Passo (b):

EPT, TMF e LER (são os críticos)

1. Em "EPT" introduzimos C_1:

E → EPT

- E → EC_1
- C_1 → PT

2. Em "TMF" introduzimos C_2:

E → TMF	E → EC_2
T → TMF	T → TC_2

C_2 → MF

3. Em "LER" introduzimos C_3:

E → TMF	E → LC_3
T → TMF	T → LC_3
E → TMF	F → LC_3

C_3 → ER

Resultado Final

E → EC_1 | TC_2 | LC_3 | a | b | IA | IB | IZ | Iθ

T → TC_2 | LC_3 | a | b | IA | IB | IZ | Iθ

F → LC_3 | a | b | IA | IB | IZ | Iθ

I → a | b | IA | IB | IZ | Iθ

A → a

B → b

Z → 0

θ → 1

P → +

M → *

L → (

R →)

C_1 → PT

C_2 → MF

C_3 → ER

Forma Normal de Chomsky!

Pergunta: Esta gramática volta a ser ambígua?

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Autômatos de Pilha

Índice

Figuras

Tabelas

Autômatos de Pilha

- "Autômatos de Pilha" é uma extensão do autômato finito não determinístico com ε-transições.
- O "Autômato de pilha" é essencialmente um ε-NFA com a inclusão de uma pilha.
- Há duas versões deferentes do "Autômato de Pilha":
  - **a)** Uma que aceita "strings" pela entrada se esta operação resultar em um estado de aceitação do conjunto F;
  - **b)** Uma outra versão que aceita pelo esvaziamento de sua pilha, independentemente do estado em que se encontra.

Teorema: Essas duas variações aceitam exatamente as linguagens livres de contexto, isto é, as gramáticas podem ser convertidas em autômatos de pilha equivalentes e vice-versa.

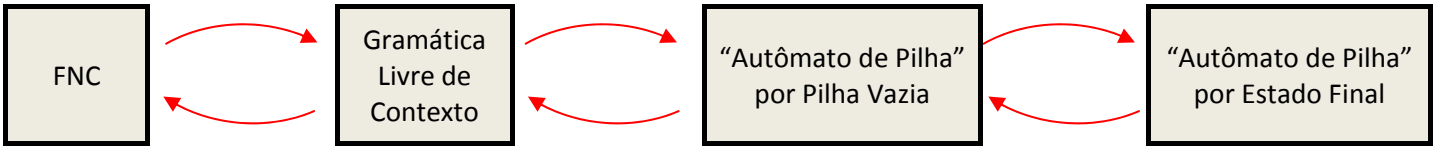


Figura 1 – Associação entre "Autômatos de Pilha" e Gramática Livre de Contexto

A Definição Formal de Autômatos de Pilha

Um "Autômato de Pilha" P pode ser definido a partir de uma tupla de 7 elementos, como a seguir:

$$P = (Q, \sum, \Gamma, \delta, q_0, Z_0, F)$$

Onde,

Q	um conjunto finito de estados, semelhantes aos estados de um autômato finito;
∑	um conjunto finito de símbolos de entrada, também semelhante ao de um autômato finito;
Γ	um alfabeto da pilha finito (conjunto "novo" de símbolos que se tem permissão de inserir na pilha), em geral ∑ ≠ Γ;
δ	A função de transição na forma δ(q,a,X), q ≡ é um estado em Q a ≡ um símbolo de entrada, onde a ∈ {∑, ε} X ≡ é um símbolo da pilha, isto é, um elemento de Γ
q_0	O estado inicial. O "Autômato de Pilha" está nesse estado antes de fazer quaisquer transições.
Z_0	O símbolo de inicio da pilha;
F	O conjunto de estados de aceitação ou estados finais.

Descrições Instantâneas de um ADP

ADP ≡ Autômato de Pilha

- Representa-se a configuração de um ADP por uma tripla (q,w,γ), onde:
    a) q é o estado;
    b) w é a parte restante da entrada;
    c) γ é o conteúdo da pilha.

Tal tripa, é denominada descrição instantânea, ou DI, do autômato de pilha.

- Seja P = (Q, $\sum$, Γ, δ, q_0, Z_0, F) um ADP. Define-se por $\vdash_P$ ou $\vdash$ o momento em que P é reconhecido, como segue. Supondo que δ(q,a,X) contém (p,α). Então, para todos os strings w em $\sum^*$ e β em Γ^*:

$$(q, aw, X\beta) \vdash (p, w, \alpha\beta)$$

"Esse movimento reflete a ideia de que, consumindo a (que pode ser ε) da entrada e substituindo X no topo da pilha por α, pode-se ir do estado q para o estado p".

- Tem-se também:

- $\vdash_P^*$ ou $\vdash^*$: quando ADP P está subentendido, para representar zero ou mais movimento ADP. Isto é:

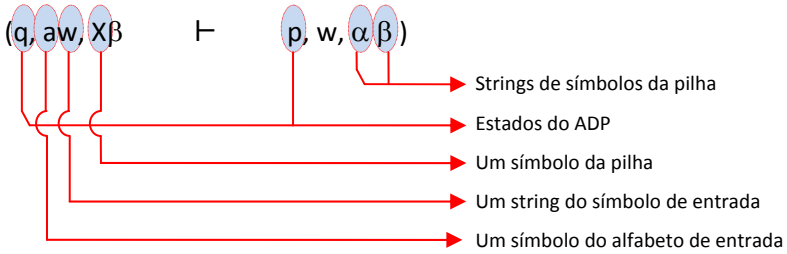
Base: I $\vdash^*$ I para qualquer DI I.

Indução: I $\vdash^*$ J se existe alguma DI K tal que I $\vdash$ K e K $\vdash^*$ J, ou seja, I $\vdash^*$ J se existe uma sequencia de DI's K_1, K_2, ..., K_n tal que I = K_1, J = K_n e, para todo i = 1,2,n, ..., n-1, tem-se $K_i \vdash K_{i+1}$.

Convenções de Notação para ADP's

1. Os símbolos do alfabeto de entrada serão representados por letras minúsculas próximas ao início do alfabeto, como a, b, c, ...
2. Os estados serão representados de modo geral por q e p ou por outras letras que estejam próximas a essas em ordem alfabética.
3. Os strings de símbolos de entrada serão representados por letras minúsculas próximas ao fim do alfabeto, como w ou z.
4. Os símbolos da pilha serão representados por letras maiúsculas próximas ao fim do alfabeto, como X ou Y.
5. Os strings de símbolos da pilha serão representados por letras gregas como δ ou γ.

Exemplo:



(q, aw, Xβ ⊢ p, w, α β)

"consumindo a (que pode ser ε) da entrada e substituindo X no topo da pilha por α, pode-se ir ao estado q para o estado p".

Exemplo: $L_{ww}{}^{R}$ = {ww^R | w está em (0+1)*} denominada "w-w-reverso" ou dos palíndromos de comprimento par.

Autômato de Pilha Equivalente

P = ({q_0, q_1, q_2}, {0,1}, {0,1,Z_0}, δ, q_0, Z_0, {q_2})

Σ	0				1				ε			
Γ	0	1	Z_0	ε	0	1	Z_0	ε	0	1	Z_0	ε
q_0	(q_0,00)	(q_0,01)	(q_0,0Z_0)		(q_0,10)	(q_0,11)	(q_0,1Z_0)		(q_1,0)	(q_1, 1)	(q_1, Z_0)	
q_1	(q_1, ε)					(q_1, ε)				(q_2, ε)		
q_2												

Tabela 1 – Função Transição do ADP

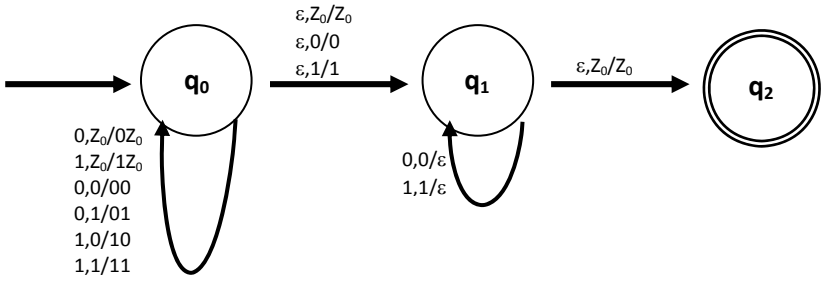


Figura 2 – Representação de um ADP como um diagrama de transição generalizado

As “Descrições Instantâneas” para o string “1111” passando pelo ADP da figura 002.

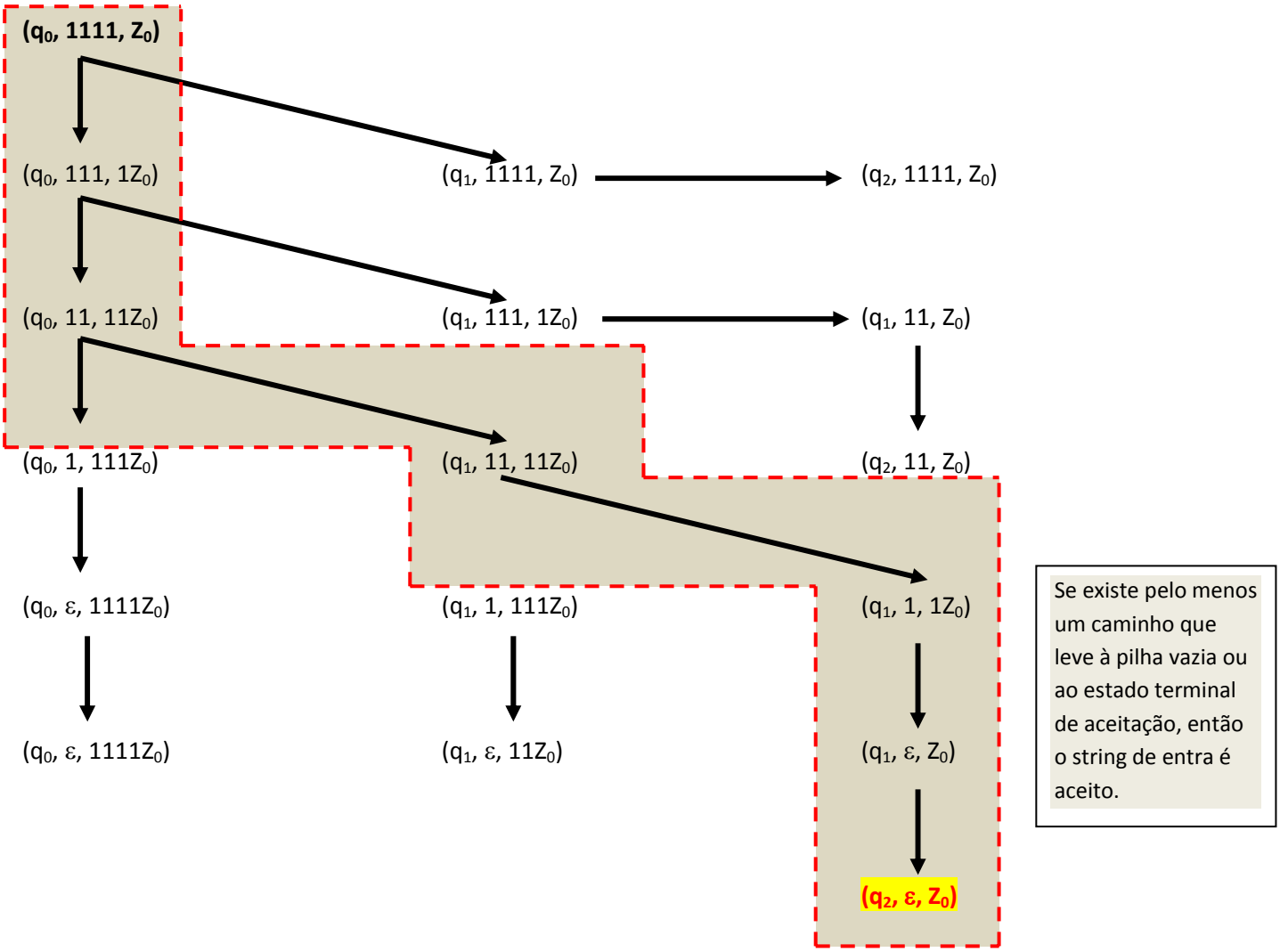


Figura 3 – DI para o string “1111”

- Em uma transição, autômato de pilha:
  a) Consome da entrada o símbolo que ele utiliza na transição. A única exceção é ε;
  b) Vá para um novo estado, podendo ou não podendo ser o anterior;
  c) Substitui o símbolo no topo da pilha por qualquer string. Se ε → operação de extração (pop), A/A (nenhuma mudança é feita), A/B (substituição do topo) e o topo ser substituído por dois ou mais símbolos.

Teorema sobre as Descrições Instantâneas

Teorema 01: Se P = (Q, ∑, Γ, δ, q_0, Z_0, F) é um ADP, e se (q,x,α) $\vdash^*$ (p,y,β), então para quaisquer strings w em Σ* e γ em Γ*, também são verdades,

(q, xw, αγ) $\vdash^*$ (p, yw, βγ) Exercício para casa!

2. (q, x, αγ) $\vdash^*$ (p, y, βγ)

1. (q, xw, α) $\vdash^*$ (p, yw, β)

Teorema 02: Se P = (Q, ∑, Γ, δ, q_0, Z_0, F) é um ADP, e se (q,xw,α)$\vdash_p^*$ (p,yw,β), então também é verdade que:

3. (q,x,α)$\vdash_p^*$ (p,y,β)

1. Se uma sequencia de DI's é válida para um ADP P, então a computação formada pela inclusão do mesmo string de entrada adicional no final da entrada (segunda componente) em cada DI também é válida.
2. Se uma computação é válida para um ADP P, então a computação formada pela inclusão dos mesmos símbolos de pilha adicionais abaixo da pilha em cada DI também é válida.
3. Se uma computação é válida para um ADP P, e se algum final da entrada não é consumido, então deve-se remover esse final da entrada em cada DI, e a computação resultante ainda permanecerá válida.

As Linguagens de um ADP

- **Aceitação por Estado Final**: quando um ADP aceita sua entrada consumindo-a e entrando em um estado de aceitação.
- **Aceitação por pilha vazia**: quando um ADP aceita todas as strings que fazem sua pilha, a partir da DI inicial.

Uma linguagem L tem um ADP que aceita pelo estado final	SSE	L tem um ADP que a aceita por pilha vazia.

- Entretanto, para um dado ADP, as linguagens que P aceita por estado final e por pilha vazia em geral são "diferentes". Assim,

> "Faz-se necessário **converter** em ADP que aceita L por estado final e outra ADP que aceita L por pilha vazia e vice-versa"

Aceitação pelo estado Final

Seja P = (Q, ∑, Γ, δ, q_0, Z_0, F) um ADP. Então **L(P)**, a linguagem aceita por P pelo estado final, é:

{w|(q_0,w,Z_0) $\vdash_P^*$ (q,ε,α)} para algum estado q em F e qualquer string de pilha α.

Aceitação por pilha vazia

Seja P = (Q, ∑, Γ, δ, q_0, Z_0, F) e define-se N(P) = **{w|(q_0,w,Z_0) $\vdash_P^*$ $(q,\varepsilon,\varepsilon)$}** para qualquer estado q. Isto é, N(P) é o conjunto de entradas w que p pode consumir e que ao mesmo tempo esvazia a sua pilha.

Prova de Equivalência de pilha vazia para estado final

"Deve-se mostrar que as classes de linguagens que são L(P) para algum ADP P é igual à classe de linguagens que são N(P) para algum ADP P. Essa classe também é exatamente a de linguagens livres de contexto".

Primeira construção: Como tomar um ADP P_N que aceita uma linguagem L por pilha vazia e construir um ADP P_F que aceita L pelo estado final.

Teorema: Se L=N(P_N) para algum ADP P_N=(Q, ∑, Γ, δ, q_0, Z_0) existe um ADP P_F tal que L = L(P_F).

Seja P = (Q, ∑, Γ, δ, q_0, Z_0, F) e define-se N(P) = **{w|(q_0,w,Z_0) $\vdash_P^*$ $(q,\varepsilon,\varepsilon)$}** para qualquer estado q. Isto é, N(P) é o conjunto de entradas w que p pode consumir e que ao mesmo tempo esvazia a sua pilha.

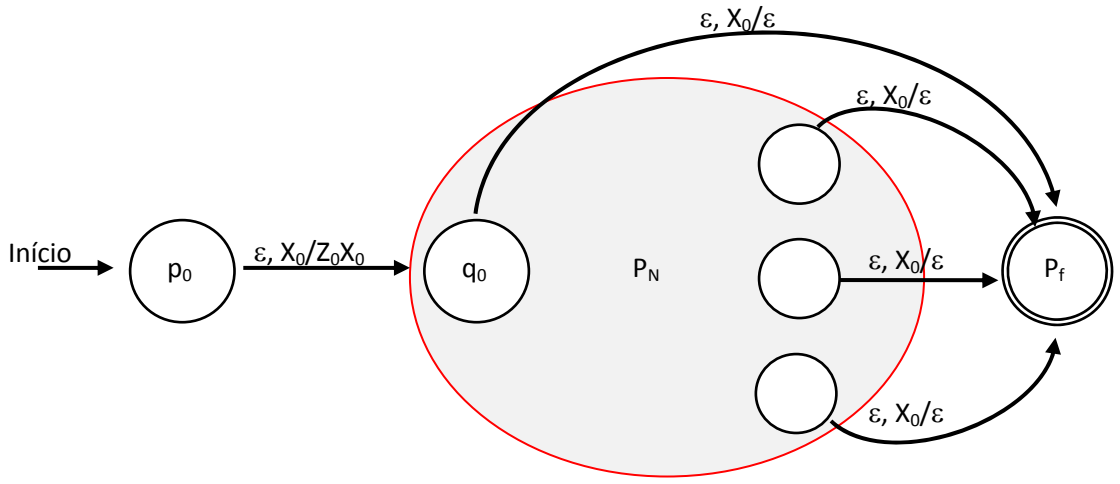


Figura 4 – Representação de um ADP de pilha vazia convertido em um ADP de estado final

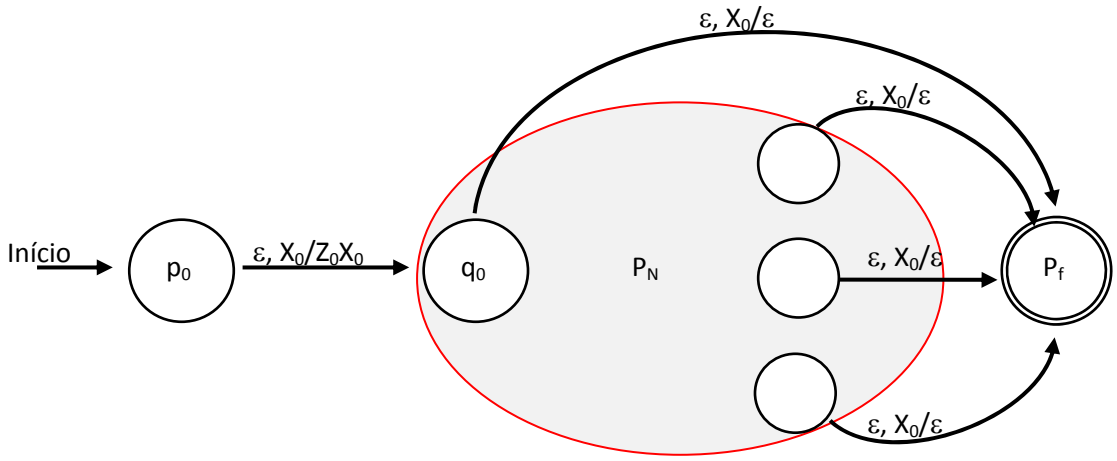
$X_0 \notin \Gamma$ É ao mesmo tempo o símbolo de início de P_F e um marcador na parte inferior da pilha que informa quando P_N alcança uma pilha vazia. Se P_F ver X_0 no topo da pilha então, P_N terá esvaziado sua pilha.

P_0 Novo estado inicial com única função de empilhar Z_0, o símbolo de início de P_N, sobre o topo da pilha e entrar no estado q_0, o estado inicial de P_N.

P_F Novo estado de aceitação de P_F (esse P_F se transfere para o estado p_f sempre que descobre que P_N esvaziou sua pilha).

$$P_F = (Q \cup \{p_0,p_F\}, \sum, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_F\})$$

Seja P = (Q, ∑, Γ, δ, q_0, Z_0, F) um ADP. Define-se por $\vdash_P$ ou $\vdash$ o momento em que P é reconhecido, como segue. Supondo que δ(q,a,X) contém (p,α). Então, para todos os strings w em ∑* e β em Γ*:



$P_F = (Q \cup \{p_0,p_F\}, \sum, \Gamma \cup \{X_0\}, \delta_F, p_0, X_0, \{p_F\})$

e

$P_N = (Q , \sum, \Gamma, \delta_N, q_0, Z_0)$

Onde δ_F é dado por:

1. $\delta_F (p_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$;
2. Para todos os estados q em Q, entradas a em ∑ ou a = ε é símbolos de pilha Y em Γ, δ_F (q,a,Y) contém todos os pares em δ_N(q,a,Y);
3. Além da regra (2), δ_F (q,ε,X_0) contém (p_F,ε) para todo estado q em Q.

Com todas estas definições deve-se então provar que:

Demonstração: "w está em $L(P_F)$ se e somente se w está em $N(P_N)$".

1. $N(P_N) \rightarrow L(P_F)$, se w está em $N(P_N)$, ou seja, é aceito por $N(P_N)$, então:

(i) De P_N sabe-se que $(q_0,w,Z_0) \vdash^*_{PN} (q,\varepsilon,\varepsilon)$ para algum estado q;

(ii) $(q_0,w,Z_0W_0) \vdash^*_{PF} (q,\varepsilon,X_0)$ [Teorema 01 (2) pq: 71];

(iii) Pela regra (2) acima $(q_0,w,Z_0X_0) \vdash^*_{PF} (q,\varepsilon,X_0)$;

(iv) Unindo a regra (1), a regra (3) e o passo (iii), tem-se $(p_0,w,X_0) \vdash^*_{PF} (q_0,w,Z_0X_0) \vdash^*_{PF} (q,\varepsilon,X_0) \vdash^*_{PF} (p_f,\varepsilon,\varepsilon)$

2. $L(P_F) \rightarrow N(P_N)$. Deixado como exercício para casa.

Exemplo Simples ADP por pilha vazia p/ ADP por estado final

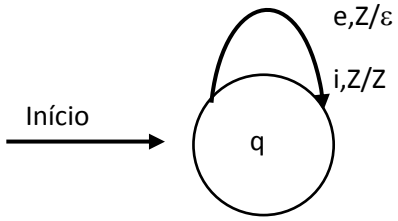


Figura 5 – Um ADP que aceita por "Pilha Vazia"

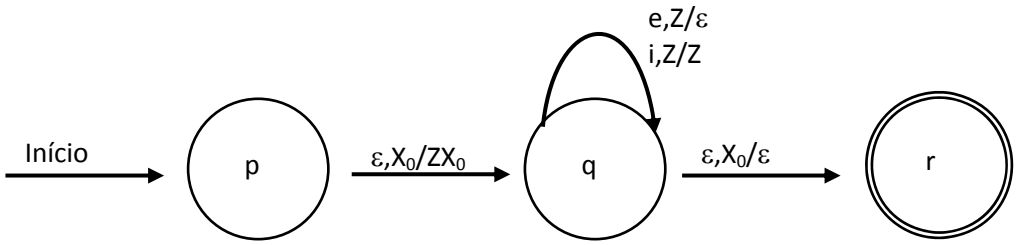


Figura 6 – Construção de um ADP que aceita pelo "Estado Final" a partir do ADP da figura 05

$P_N = (\{q\}, \{i,e\}, \{Z\}, \delta_N, q, Z)$	$P_F = (\{p,q,r\}, \{i,e\}, \{Z,X_0\}, \delta_F, p, X_0, \{r\})$
1. $\delta_N (q,i,Z) = \{q,ZZ\}$ 2. $\delta_N (q,e,Z) = \{q,\varepsilon\}$	1. $\delta_F (q,\varepsilon,X_0) = \{(q,ZX_0)\}$ 2. $\delta_F (P,i,Z) = \{(q,ZZ)\}$ 3. $\delta_F (q,e,Z) = \{(q, \varepsilon)\}$ 4. $\delta_F (q,\varepsilon,X_0) = \{(r, \varepsilon)\}$

Prova da Equivalência de estado final para Pilha Vazia

Caminhando no sentido oposto: a partir de um ADP P_F que aceite uma linguagem L pelo "estado final", construir outro ADP P_N que aceite L por pilha vazia.

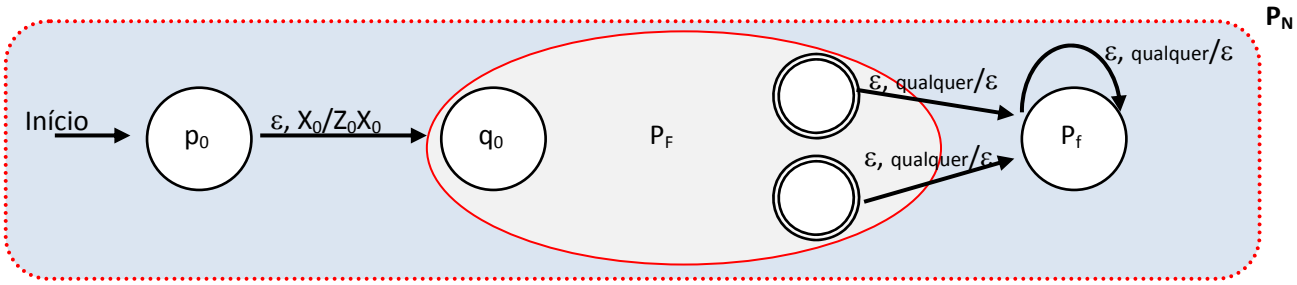


Figura 7 – P_N simula P_F e esvazia sua pilha quando e somente quando P_F entra em um estado de aceitação

Teorema: Seja L a $L(P_F)$ para algum ADP $P_F=(Q, \sum, \Gamma, \delta, q_0, Z_0, F)$.. Então existe um ADP P_N tal que $L=N(P_N)$.

Construção de P_N: Sugerida pela figura 7 acima, ou seja, $P_N = (Q \cup\{p_0,p\}, \Sigma, \Gamma\cup\{X_0\}, \delta_N, p_0, X_0)$

Onde, δ_N é definida por:

1. $\delta_n(p_0,\varepsilon,X_0)=\{(q_0,Z_0X_0)\}$;
2. Para todos os estados q em Q, símbolos de entrada a em Σ ou $a=\varepsilon$ e Y em Γ, $\delta_N(q,a,Y)$ contém todo par que esta $\delta_F(q,a,Y)$. Isto é, P_N simula P_F;
3. Para todos os estados de aceitação q em F e símbolos de pilha em Y em Γ ou $Y=X_0$, $\delta_N(q,\varepsilon,Y)$ contém (p,ε).
4. Para todos os símbolos de pilha Y em Γ ou $Y = X_0$, $\delta_N(p,\varepsilon,Y)=(p,\varepsilon)$.

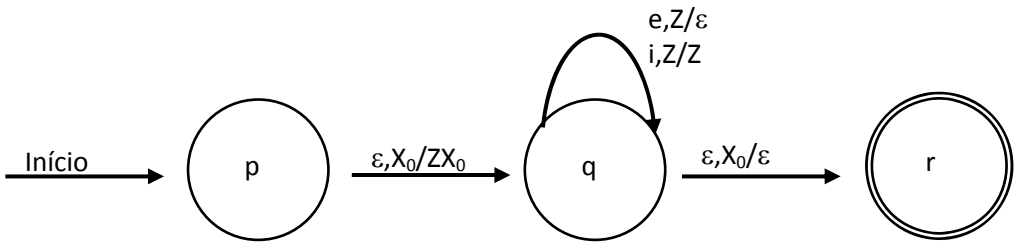


Figura 8 – Mesmo ADP que aceita pelo "estado final" da figura 6

$P_F = (\{p,q,r\}, \{i,e\}, \{Z,X_0\}, \delta_F, p, X_0, \{r\})$

1. $\delta_F (p,\varepsilon,X_0) = \{(q,ZX_0)\}$
2. $\delta_F (p,i,Z) = \{(q,ZZ)\}$
3. $\delta_F (q,e,Z) = \{(q, \varepsilon)\}$
4. $\delta_F (q,\varepsilon,X_0) = \{(r, \varepsilon)\}$

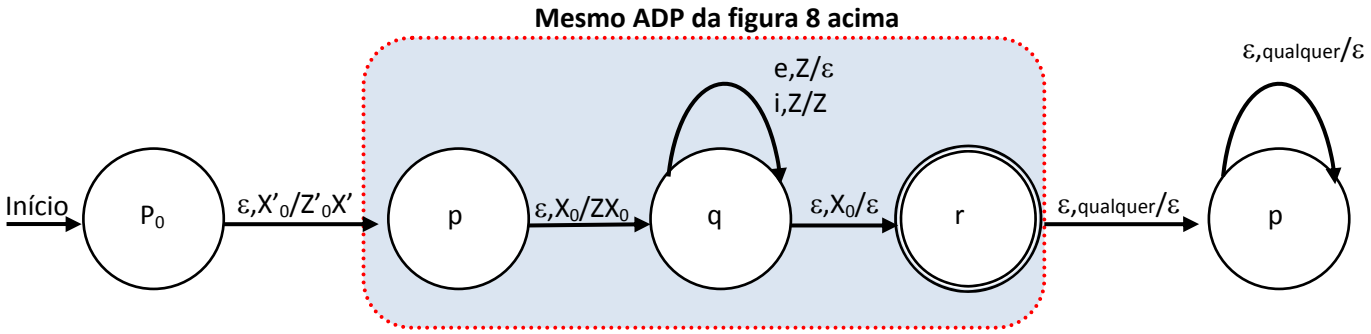


Figura 9 - Construção de um ADP que aceita por "pilha vazia" a partir do ADP por "estado final" da figura 8

$P_N = (\{p,q,r\} \cup \{p_0,p\}, \{i,e\}, \{Z,X_0\} \cup \{X'_0\}, \delta_N, p_0, X'_0,)$

δ_N:

1. $\delta_N(p_0,\varepsilon,X'_0) = \{(q_0,Z_0X'_0)\}$
2. $\delta_N = \delta_F$
3. $\delta_N(r,\varepsilon,$ qualquer$) = (p,\varepsilon)$
4. $\delta_N(p,\varepsilon,$ qualquer$) = (p,\varepsilon)$

1. $\delta_F\ (p,\varepsilon,X_0) = \{(q,ZX_0)\}$
2. $\delta_F\ (p,i,Z) = \{(q,ZZ)\}$
3. $\delta_F\ (q,e,Z) = \{(q,\varepsilon)\}$
4. $\delta_F\ (q,\varepsilon,X_0) = \{(r,\varepsilon)\}$

Nota: O ADP que aceita por "pilha vazia" da figura 9 é equivalente ao ADP que aceita por "pilha vazia" da figura 5, no sentido de representarem a mesma linguagem, mas não são iguais.

Resumo da Gramática

G = (V, ∑, P, S)

- S: Símbolo inicial S ∈ V
- P: Conjunto de produções
- ∑: Conjunto finito de símbolos terminais para V ∩ ∑ = ∅
- V: Conjunto finito V de símbolos não terminais

Exemplo: gerador de palíndromos sobre {a,b}

1. G = (V, ∑, P, S)

V = {S} ... "alfabeto de terminais" ou "símbolos terminais"

∑ = {a,b} ... "conjunto finito de variáveis"

P = {S → a, S → b, S → ε, S → aSa, S → bSb}

S = S ... Símbolo de inicio (sempre uma das variáveis)

2. $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n \Leftrightarrow A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_n$ (simplificação de notação)

3. Se $\alpha \rightarrow \beta$ e $x\alpha y \in (V \cup \Sigma)^*$, então $x\beta y$ é diretamente deriváveis de $x\alpha y$, ou seja $x\alpha y \Rightarrow x\beta y$.

   Se $\alpha i \in (V \cup \Sigma)^*$ para i=1, ..., n e $\alpha_{i+1} \Rightarrow \alpha_i$ para i=1,..,(n-1), então α_n é derivável de α_1, ou seja, $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$.

4. Uma "produção unitária" é uma produção da forma A → B, onde tanto A quanto B são variáveis. Contra exemplo: A→a.
5. "A linguagem L(G) gerada por G consiste de todas as cadeias sobre Σ derivados de S"

De gramáticas para Autômatos de Pilha

Derivação mais à esquerda: quando, em cada etapa, a variável mais a esquerda é substituída por um de seus corpos de produção.

$$\overrightarrow{ln} \overset{*}{\underset{ln}{\Rightarrow}}$$

Ex.: $E \underset{ln}{\Rightarrow} E*E \underset{ln}{\Rightarrow} I*E \underset{ln}{\Rightarrow} a*E \underset{ln}{\Rightarrow} a*(E) \underset{ln}{\Rightarrow} a*(E+E) \underset{ln}{\Rightarrow} a*(I+E)$

"A ideia que rege a construção de um ADP a partir de uma gramática é fazer o ADP simular a sequencia de formas sentenciais à esquerda que a gramática utiliza para gerar um dado string de terminais w".

Seja G=(V,∑,P,S) uma gramática livre de contexto. Construa o ADP P que aceite L(G) por pilha vazia, da seguinte forma:

$$P=(\{q\}, \Sigma, \overbrace{V \cup \Sigma}^{\Gamma}, \delta, S)$$

Onde a função de transição δ é definida por:

1. Para cada variável A, $\delta(q,\varepsilon,A)=\{(q,\beta) | A \rightarrow \beta$ é uma produção de G$\}$
2. Para cada terminal a, $\delta(q,a,a)=\{(q,\varepsilon)\}$

Teorema: Se o ADP P é construído a partir da GLC G pela construção anterior, então N(P)=L(G).

Para cada ADP $P=(Q,\sum,\Gamma,\delta,q_0,Z_0,F)$ define-se:

$$N(P) = \{w | (q_0,w,Z_0) \overset{*}{\vdash_P} (q,\varepsilon,\varepsilon)\}$$

Para qualquer estado q. Isto é, N(P) é o conjunto de entradas w que P pode consumir e que ao mesmo tempo esvazia a pilha.

EX.: I → a|b|Ia|Ib|I0|I1

E → I|E*E|E+E|(E)

Terminais para o ADP P = {a,b,0,1,(,),+,*}

$\Gamma = V \cup \sum$ = {a,b,0,1,(,),+,*,I,E}

Função de transição δ para o ADP P

a) $\delta(q,\varepsilon,I)$ = {(q,a), (q,b), (q,Ia), (q,Ib), (q,I0), (q,I1)}

b) $\delta(q,\varepsilon,E)$ = {(q,I), (q,E+E), (q,E*E), (q,(E))}

c) $\delta(q,a,a) = \{(q,\varepsilon)\}$

$\delta(q,b,b) = \{(q,\varepsilon)\}$

$\delta(q,0,0) = \{(q,\varepsilon)\}$

$\delta(q,1,1) = \{(q,\varepsilon)\}$

$\delta(q,(,() = \{(q,\varepsilon)\}$

$\delta(q,),)) = \{(q,\varepsilon)\}$

$\delta(q,+,+) = \{(q,\varepsilon)\}$

$\delta(q,*,*) = \{(q,\varepsilon)\}$

De autômatos de pilha para Gramáticas Livre de Contexto

Teorema 01:	Para todo ADP P que aceita por pilha vazia, pode-se encontrar uma linguagem Livre de Contexto G cuja linguagem é a mesma linguagem que P aceita.
Teorema 02:	Seja P = $(Q,\sum,\Gamma,\delta,q_0,Z_0,F)$. Então, existe uma gramática livre de Contexto G tal que L(G) = N(P), onde a gramática G=(V,∑,R,S) pode ser obtida da seguinte forma: (i) Conjunto de variáveis V 1. No símbolo especial S, que é o símbolo de início. 2. Em todos os símbolos da forma [pxq], onde p e q são estados em Q e X é um símbolo de pilha, em Γ. (ii) As produções em G são: 1. Para todos os estados p, G tem a produção S → $[q_0Z_0p]$ $(q_0,w,Z_0) \overset{*}{\vdash} (p,\varepsilon,\varepsilon)$ 2. Seja δ (q,a,X) contendo o par $(r,Y_1Y_2\ldots Y_R)$, onde: • a é um símbolo em Σ ou a = ε • k pode ser qualquer numero, inclusive 0, e nesse caso o par é (r,ε). Então, para todas as listas de estados $r_1,r_2,\ldots,r_k$, G tem a produção $[qXr_k] \rightarrow a[rY_1r_1]\, a[r_1Y_2r_2] \ldots a[r_{k-1}Y_kr_k]$ Pg: 259 e 260 do Hopcroft (português)

Exemplo:

Converter o

ADP $P_N=(\{q\}, \{i,e\}, \{Z\}, \delta_N, q, Z)$
$\delta_N\ (q,i,Z) = \{(q,ZZ)\}$
$\delta_N\ (q,e,Z) = \{(q,\varepsilon)\}$

Solução:

1. Como P_N tem apenas o estado {q} e apenas um símbolo de pilha {Z}, a construção da gramática é bastante simples para este caso. Assim, só existem duas variáveis na gramática G:
   a. S … símbolo de início;
   b. [qZq] … a única tripla que pode ser montada a partir de P_N.
2. A única produção para S é S → [qZq]
3. Do fato que δ_N (q,i,Z) ={(q,ZZ)} então, [qZq] → **i[qZq][qZq]**
4. Do fato que δ_N (q,e,Z) ={(q,ε)} então, [qZq] → **e**

Assim, se A ≡ [qZq] então,

S → A
A → iAA|e

} Note que tanto S como A derivam exatamente os mesmos strings!!

G = ({S}, {i,e}, {S→iSS|e}, S)

Exemplo:

$P = (\{q_1,q_2,q_3,q_4,q_5,q_6,q_7\}, \sum=\{a,b,c\}, \Gamma=\{a, \$\}, \delta, q, S, F=\{q_4,q_7\})$

O autômato de pilha P, como esquematizado abaixo e definido acima, representa a linguagem $L=\{a^i b^j c^k \mid i, j, k \geq 0 \text{ e } i=j \text{ ou } i=k\}$. Para este autômato P determine:

a) A tabela de função δ a partir da interpretação do grafo abaixo;
b) O autômato de Pilha equivalente por aceitação de **pilha vazia***;
c) A gramática livre de contexto equivalente.

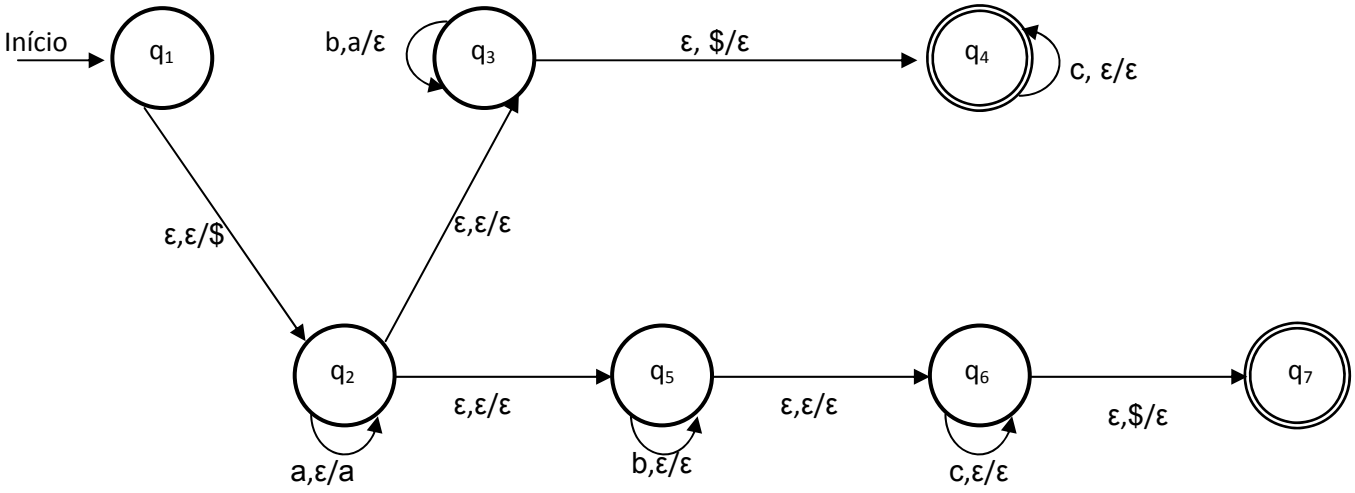


Figura 10 – Grafo Direcionado do Autômato P da Questão Acima

Nota: No item (b) construa também a tabela δ de "descrição instantâneas" a partir do "novo" grafo direcionado a ser obtido.

a) A tabela de função δ a partir da interpretação do grafo anterior.

δ_F	Autômato por Aceitação de **Estado Final** (Não Determinístico)											
Fita	a			b			c			ε		
Pilha	a	$	ε	a	$	ε	a	$	ε	a	$	ε
q_1	-	-	-	-	-	-	-	-	-	-	-	(q_2,ε)
q_2	-	-	(q_2,a)	-	-	-	-	-	-	-	-	(q_3,q_5,ε)
q_3	-	-	-	(q_3,ε)	-	-	-	-	-	-	(q_4,ε)	-
q_4	-	-	-	-	-	-	-	-	(q_4,ε)	-	-	-
q_5	-	-	-	-	-	(q_5,ε)	-	-	-	-	-	(q_6,ε)
q_6	-	-	-	-	-	-	(q_6,ε)	-	-	-	(q_7,ε)	-
q_7	-	-	-	-	-	-	-	-	-	-	-	-

Tabela 2 – Função δ a partir do grafo anterior

b) O autômato de Pilha equivalente por aceitação de **pilha vazia**;

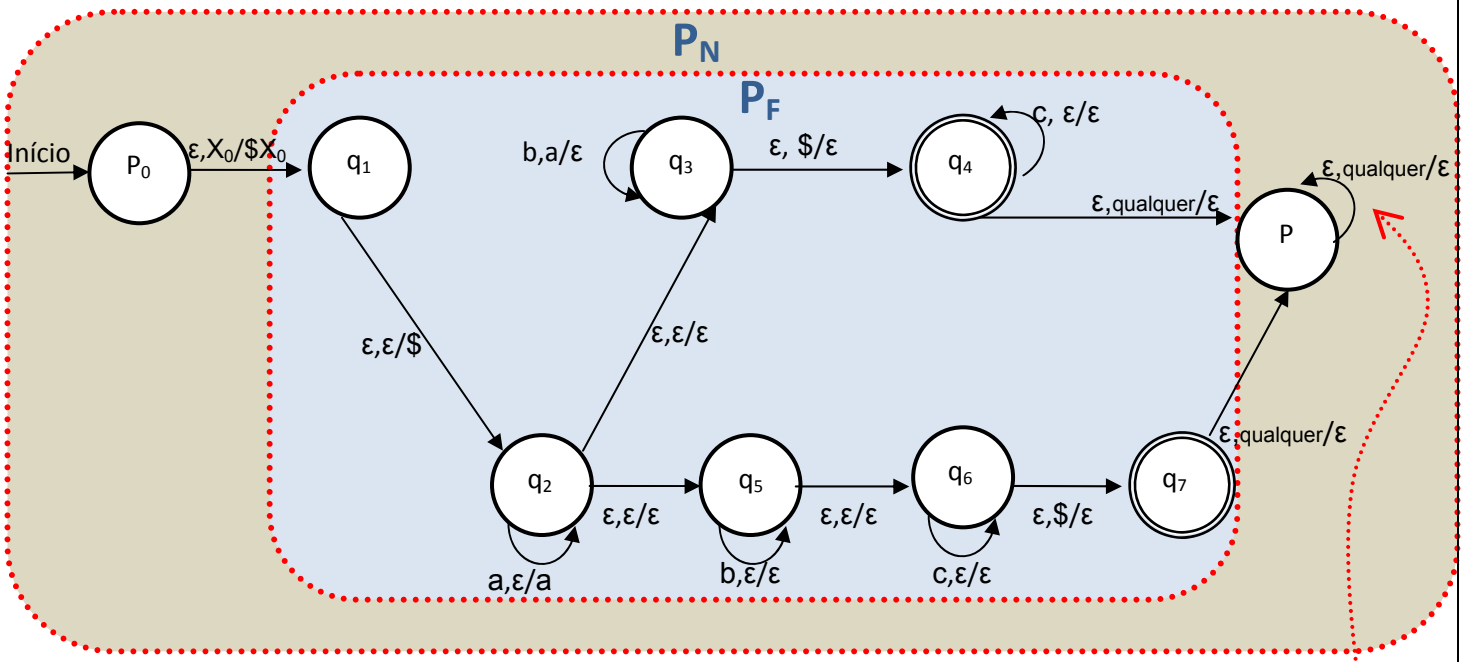


Figura 11 – Grafo Direcionado do Autômato de Pilha por Aceitação por Pilha Vazia

Grafo Direcionado do Autômato de Pilha por Aceitação por Pilha Vazia. Equivalente ao Autômato de Pilha P por Aceitação por Estado Final da Página Anterior.

Por que este é necessário?

Porque num autômato de pilha só é possível excluir da pilha um símbolo de cada vez!

b) A tabela de "descrições instantâneas" δ para o autômato P por aceitação de pilha vazia da página anterior.

δ_F	Autômato por Aceitação de **Estado Final** (Não Determinístico)													
Fita	a			b			c			ε				
Pilha	a	$	ε	a	$	ε	a	$	ε	a	$	ε	X_0	qualquer
P_0	-	-	-	-	-	-	-	-	-	-	-		$(q_1,\$X_0)$	(q_2,ε)
q_1	-	-	-	-	-	-					-	$(q_2,\$)$	-	-
q_2	-	-	(q_2,a)	-	-	-	-	-	-	-	-	(q_3,q_5,ε)	-	-
q_3	-	-	-	(q_3,ε)	-	-	-	-	-	-	(q_4,ε)	-	-	-
q_4	-	-	-	-	-	-	-	-	(q_4,ε)	-	-	-	-	(p,ε)
q_5	-	-	-	-	-	(q_5,ε)	-	-	-	-	-	(q_6,ε)	-	-
q_6	-	-	-	-	-	-	(q_6,ε)	-	-	-	(q_7,ε)	-	-	-
q_7	-	-	-	-	-	-	-	-	-	-	-	-	-	(p,ε)
P	-	-	-	-	-	-	-	-	-	-	-	-	-	(p,ε)

Tabela 3 – Função δ a partir do grafo anterior

c) A gramática livre de contexto equivalente.
O autômato de pilha a ser considerado é:

$$P_N=(\{\underbrace{p_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, p}_{Q}\}, \underbrace{\{a,b,c\}}_{\Sigma}, \underbrace{\{a,\$,X_0\}}_{\Gamma}, \delta, p_0, \$)$$

Onde,
δ ≡ função de "descrições instantâneas" como expressa na tabela anterior.

O objetivo deste exercício é obter a gramática livre de contexto G=(V,∑,P,$) equivalente ao autômato por aceitação por pilha vazia esquematizado na figura 11.

Passo 1: Conjunto de variáveis V (símbolos não terminais de G)

a) S … sempre existirá um novo símbolo, denominado **símbolo de início**;

b) Q = $\{ p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p\}$, Γ = $\{a,\$,X_0\}$

Devemos obter todo símbolo na forma [pXq].

P_0 para os demais à direita		q_1 para os demais à direita		…	q_6 → direita	q_7 → direita	p → direita
$[p_0 a p_0]$	$[p_0 a q_6]$	$[q_1 a q_1]$	$[q_1 a q_7]$		$[q_6 a q_6]$	$[q_7 a q_7]$	[p a p]
$[p_0 a q_1]$	$[q_6 a p_0]$	$[q_1 a q_2]$	$[q_7 a q_1]$		$[q_6 a q_7]$	$[q_7 a p]$	
$[q_1 a p_0]$	$[p_0 a q_7]$	$[q_2 a q_1]$	$[q_1 a p]$		$[q_7 a q_6]$	$[p a q_7]$	
$[p_0 a q_2]$	$[q_7 a p_0]$	$[q_1 a q_3]$	$[p a q_1]$		$[q_6 a p]$		
$[q_2 a p_0]$	$[p_0 a p]$	$[q_3 a q_1]$		…	$[p a q_6]$		
$[p_0 a q_3]$	$[p a p_0]$	$[q_1 a q_4]$					
$[q_3 a p_0]$		$[q_4 a q_1]$					
$[p_0 a q_4]$		$[q_1 a q_5]$					
$[q_4 a p_0]$		$[q_5 a q_1]$					
$[p_0 a q_5]$		$[q_1 a q_6]$					
$[q_5 a p_0]$		$[q_6 a q_1]$					

Tabela 4 – Símbolo na forma [pXq]

Na verdade você não precisa desenvolver os 81 símbolos gerados a partir de Q e a mais os outros 81 símbolos gerados a partir de Q e $, …

Se $Q = \{p_1, p_2, \ldots, p_n\}$

e $\Gamma = \{ X_1, X_2, \ldots, X_m \}$

então pode-se utilizar a formula $(m.n^2)$ para determinar o numero total de variáveis V da gramática equivalente, ou seja, para nosso caso particular:

n=9 e m=3 ou seja 3.9^2=243 símbolos para V. Pode-se então adotar a seguinte convenção:

A_i^{α} para i=1, …, 81
$A_i^{\$}$ para i=1, …, 81
$A_i^{X_0}$ para i=1, …, 81

ou

$V = \{\$, A_i^{\alpha}, A_i^{\$}, A_i^{X_0} \mid i \in \}$

Passo 2: Determinar as produções de G.

a.

$S \rightarrow [p_0 X_0 q_i]$ p/ i=1,2,...,7
$S \rightarrow [p_0 X_0 p]$, $S \rightarrow [p_0 X_0 p_0]$

Se Q possui **n** estados distintos, então ter-se-á **n** produções em S

b. Da tabela 3, tem-se:

b.1) $\delta_N(q_2,a,\varepsilon)=\{(q_2,a)\}$

Daqui para frente tornou-se muito extenso determinar as produções. Sendo assim, nas páginas seguintes segue um exemplo mais Simples.

Exemplo mais Simples:

$P = (\{q,r\}, \{a,b\}, \{Z_1, Z_2, \$\}, \delta, q, \$)$

para $\delta_N(q,a,\$) = (q,Z_1Z_2)$

$\delta_N(q,b,\$) = (r,Z_1Z_2)$

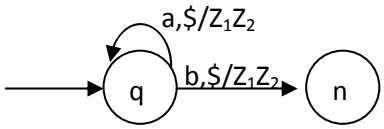


Figura 12 – Autômato de Pilha

Exemplo Infeliz: Se começarmos com pilha vazia, qualquer string nunca esvaziará a pilha.

a) Converter o autômato de pilha acima numa gramática livre de contexto.

Passo 1: Obter o conjunto de variáveis V (símbolos não terminais) de G=(V,∑,P,S)

a. S … sempre existira um novo símbolo, denominado **símbolo inicial**;

b. Q={q,r}, Γ={Z_1,Z_2,S}. Devemos obter todo símbolo na forma [pXq], logo

$$V=\left\{\begin{array}{llll}[qZ_1q], & [qZ_2q], & [q\$q], & \\ [qZ_1r], & [qZ_2r], & [q\$r], & \\ [rZ_1q], & [qZ_2q], & [r\$q], & \\ [rZ_1r], & [rZ_2r], & [r\$r], & S\end{array}\right.$$

$m.n^2=3.2^2=3.4=12$ símbolos ∈ V, mais o símbolo S

Total de símbolo em V ($m.n^2+1$)

Passo 2: Determinar as produções de G.

a. Para todos os estados p ∈ Q tem-se a produção S → [q$p]

S → [q$q]

S → [q$r]

b. Seja $\delta(q,a,X) \vdash (r,Y_1\ Y_2 \ldots Y_k)$, então as listas de estados $r_1,r_2, \ldots,r_k$, G tem a produção

$[qXr_k] \rightarrow a[rY_1r_1]\ [r_1Y_2r_2] \ldots [r_{k-1}Y_kr_k]$

Q = {q,r}	
$\delta_N(q, a, \$) = (q, Z_1, Z_2)$	$\delta_N(q, b, \$) = (r, Z_1, Z_2)$
[q$q] → a[qZ1q][qZ2q]	[q$q] → b[rZ1q][qZ2q]
[q$q] → a[qZ1r][rZ2q]	[q$q] → b[rZ1r][rZ2q]
[q$r] → a[qZ1q][qZ2r]	[q$r] → b[rZ1q][qZ2r]
[q$r] → a[qZ1r][rZ2r]	[q$r] → b[rZ1r][rZ2r]

Assim, a definição completa da gramática G = (V,∑,P,S) torna-se:

∑ = {a,b}

V = {S, [qZ$_1$q], [qZ$_2$q], [q$q], [qZ$_1$r], [qZ$_2$r], [q$r], [rZ$_1$q], [rZ$_2$q], [r$q], [rZ$_1$r], [rZ$_2$r], [r$r]}

Produções P:

S → [q$q]	[q$q] → a[qZ$_1$q][qZ$_2$q]	[q$q] → b[rZ$_1$q][qZ$_2$q]
S → [q$r]	[q$q] → a[qZ$_1$r][rZ$_2$q]	[q$q] → b[rZ$_1$r][rZ$_2$q]
	[q$r] → a[qZ$_1$q][qZ$_2$r]	[q$r] → b[rZ$_1$q][qZ$_2$r]
	[q$r] → a[qZ$_{1r}$][rZ$_2$r]	[q$r] → b[rZ$_1$r][rZ$_2$r]

Assim, tem-se finalmente que:

∑ = {a,b}

V = {S,A,B,C$_1$, C$_2$, C$_4$, C$_5$, C$_7$, C$_8$, C$_{10}$, C$_{11}$}

Produção P:

S → A	A → aC$_1$C$_2$	A → bC$_7$C$_2$
S → B	A → aC$_4$C$_8$	A → bC$_{10}$C$_8$
	B → aC$_1$C$_5$	B → bC$_7$C$_5$
	B → aC$_4$C$_{11}$	B → bC$_{10}$C$_{11}$

Nota: Na prova poderá cair um exemplo com até 3 estados em Q e 3 símbolos de pilha Γ.

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Autômatos de Pilha

Índice

Figuras

Tabelas

Autômatos de Pilha Determinísticos

- Embora por definição os Autômatos de Pilha (PDA's) possam ser não-determinísticos, o subcaso determinísticos é bastante importante.
- Intuitivamente, um PDA é determinístico se **nunca existe uma escolha** de movimento em qualquer situação.

- Desse modo, define-se um autômato de pilha PDA $P=(Q,\sum,\Gamma,\delta,Z_0,F)$ como **determinístico** (DPDA) se e somente se as seguintes condições são satisfeitas:

1. $\delta(q,a,X)$ tem no máximo um elemento para qualquer q em Q, a em $\sum$ ou a=ε e X em Γ.
2. Se $\delta(q,a,X)$ não é vazio para algum a em $\sum$, então $\delta(q,\varepsilon,X)$ deve ser vazio.

Exemplo de Autômato de Pilha Determinístico

$P=(\{q_0,q_1,q_2\}, \{0,1,c\}, \{0,1,Z_0\}, \delta, q_0, \{q_2\})$

∑	0			1			c			ε		
Γ	0	1	Z_0	0	1	Z_0	0	1	Z_0	0	1	Z_0
→q_0	$(q_0,00)$	$(q_0,01)$	$(q_0,0Z_0)$	$(q_0,10)$	$(q_0,11)$	$(q_0,1Z_0)$	$(q_1,0)$	$(q_1,1)$	(q_1,Z_0)	-	-	-
q_1	(q_1,ε)	-	-	-	(q_1,ε)	-	-	-	-	-	-	(q_2,Z_0)
*q_2	-	-	-	-	-	-	-	-	-	-	-	-

Tabela 1 – Função Transição do DADP

Um autômato de pilha será determinístico se e somente se as seguintes condições são satisfeitas:

1. $\delta(q,a,X)$ tem no máximo um elemento para qualquer q em Q, a em $\sum$ ou a=ε e X em Γ.
2. Se $\delta(q,a,X)$ não é vazio para algum a em $\sum$, então $\delta(q,\varepsilon,X)$ deve ser vazio.

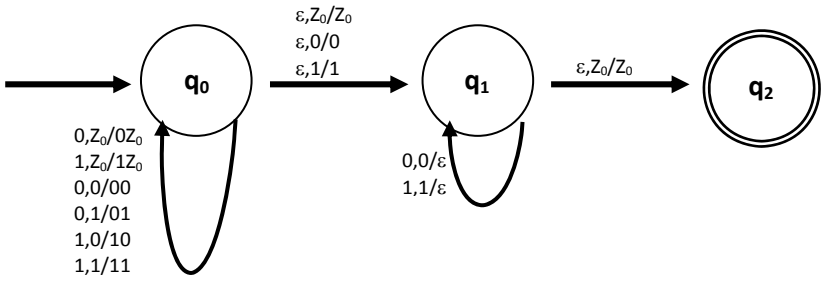


Figura 1 – Automato de Pilha Deterministico

Exemplo de Autômato de Pilha Não-Determinístico

$P=(\{q_0,q_1,q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$

∑	0				1				ε			
Γ	0	1	Z_0	ε	0	1	Z_0	ε	0	1	Z_0	ε
→q_0	$(q_0,00)$	$(q_0,01)$	$(q_0,0Z_0)$		$(q_0,10)$	$(q_0,11)$	$(q_0,1Z_0)$		$(q_1,0)$	$(q_1,1)$	(q_1,Z_0)	
q_1	(q_1,ε)	-	-		-	(q_1,ε)	-		-	-	(q_2,Z_0)	
*q_2	-	-	-		-	-	-		-	-	-	

Tabela 2 – Função Transição do Autômato de Pilha Não-Determinístico

Não é permitido – em nenhum caso – X=ε, em $\delta(q,a,X) \vdash (p,j)$

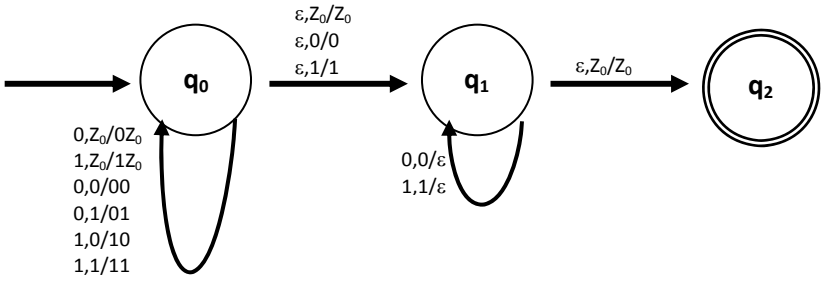


Figura 2 – Automato de Pilha Não-Deterministico

Além disso, um autômato de pilha não-determinístico poderia aceitar transições do tipo:

$$\delta(q_i,a,A) = \{(q_j,B), (q_R,C)\}$$

Teorema 1:	Se L PE uma linguagem regular, então L-L(P) para algum autômato de pilha DPDA P que aceita uma string por **estado final**.

Em essência, um autômato de pilha Determinístico DPDA pode simular um autômato finito determinístico DFA:

1. Formalmente, seja $A=(Q,\sum,\delta_A,q_0,F)$ um DFA;
2. Construa o DPDA $P=(Q,\sum,\{Z_0\},\delta p,q_0,F)$;
3. Defina $\delta_p(q,q,Z_0)=\{(p,Z_0)\}$ para todos os estados p e q em Q, tais que $\delta_A(q,a)=p$;
4. Neste caso, pode ser demonstrado por indução finita que

   $(q_0,w_0,Z_o) \overset{*}{\vdash_p} (p,\varepsilon,Z_0)$ se e somente se $\hat{\delta}_A(q_0,w)=p$. Isto é, P simula A usando seu estado.
5. Tendo em vista que tanto A quanto P aceitam pela entrada em um dos estados de aceitação de F, concluímos que suas linguagens são iguais.

Observação: Se desejarmos que um autômato de pilha determinístico DPDA **aceite por pilha vazia**, descobriremos que nossa capacidade de reconhecimento de linguagens é mais limitada que no caso anterior, ou seja, "não é nem mesmo verdade que **toda linguagem regular** é N(P) para algum DPDA P que **aceite por pilha vazia**".

- As linguagens aceitas por **DPA's pelo estado final** incluem propriamente as linguagens regulares, mas estão incluídas mais propriamente ainda nas gramáticas livres de contexto não-ambíguas.
- Não é mesmo nem verdade que **toda linguagem regular** é N(P) para algum DPDA P que **aceite por pilha vazia**.

Teorema 2:	Se L=N(P) para algum DPDA P, então L tem uma gramática livre de contexto não-ambígua. Aqui a notação L=N(P) se refere a um autômato de pilha que aceite por pilha vazia.
Teorema 3:	Se L=L(P) para algum DPDA P, então L tem uma gramática livre de contexto não-ambígua. Aqui a notação L=L(P) se refere a um autômato de pilha que aceite por estado final.

Propriedades de Fechamento e Propriedades de Decisão

1 Propriedades de Fechamento das Linguagens Regulares

- Esses teoremas são chamados com frequência **propriedades de fechamento** das linguagens regulares, pois mostram que a classe de linguagens regulares é fechada sobre a operação mencionada.

- As propriedades de fechamento expressam a ideia de que, quando uma ou várias linguagens são regulares, então certas linguagens relacionadas também são regulares.
- Nossas primeiras propriedades de fechamento são as três operações booleanas: união, interseção e complementação:

a. Sejam L e M linguagens sobre o alfabeto ∑. Então, L ∪ M é a linguagem que contém todos os strings que estão em L, em M ou em ambos;

b. Sejam L e M linguagens sobre o alfabeto ∑. Então L ∩ M é uma linguagem que contém todos os strings que estão em L e M;

c. Seja L uma linguagem sobre o alfabeto ∑. Então $\bar{L}$, o complemento de L, é o conjunto de strings em ∑* que não estão em L.

- Ocorre que as linguagens regulares são fechadas sob todas as três operações booleanas. Entretanto, as provas exigem abordagens bem diferentes.

Teorema 1:	Se L e M são linguagens regulares, e então L ∪ M também é.
Teorema 2:	Se L e M são linguagens regulares, e então L ∩ M também é.
Teorema 3:	Se L é uma linguagem regular sobre o alfabeto ∑, então $\bar{L}$=∑*-L também é uma linguagem regular.
Teorema 4:	Se L e M são linguagens regulares, e então L - M também é, onde L-M=L∩$\bar{M}$.

- **Reversão**: O reverso de um string $a_1a_2…a_n$ é o string escrito ao contrário, ou seja, $a_na_{n-1}…a_1$. Utilizamos w^r para representar o reverso do string w. Desse modo, 0010^R é 0100, e $\varepsilon^R = \varepsilon$.
- A reversão é outra operação que preserva as linguagens regulares, isto é, se L é uma linguagem regular, então L^R também é.

Teorema 5:	Se L é uma linguagem regular, então L^R também é

- **Homomorfismos**: Um homomorfismo de strings é uma função sobre strings que atua substituindo cada símbolo por um string específico.

Formalmente, se h é um homomorfismo sobre o alfabeto ∑, e w= $a_1a_2…$na é um string de símbolos em ∑, então h(w)=h(a_1)h(a_2)…h(na). Ou seja, aplicamos h a cada símbolo de w e concatenamos os resultados em ordem.

Além disso, podemos aplicar um homomorfismo a uma linguagem aplicando-o a cada um dos strings na linguagem. Isto é, se L é uma linguagem sobre o alfabeto ∑ e h é um homomorfismo sobre ∑, então h(L)={h()w)| w está em L}.

Exemplo: Se h(0)=ab e h(1)=ε então, para w=0011 ter-se-á h(w)=h(0)h(0)h(1)h(1)=(ab)(ab)(ε)(ε)=abab.

Teorema 6:	Se L é uma linguagem regular sobre o alfabeto ∑, e h é um homomorfismo sobre ∑, então h(L) também é regular.

Homomorfismo Inversos: Homomorfismos também podem ser aplicados "ao contrário" e, dessa maneira, também preservam linguagens regulares. Isto é, suponha que h seja um homomorfismo de algum alfabeto ∑ para strings em outro (possivelmente o mesmo) alfabeto T. Seja L uma linguagem sobre o alfabeto T. Então $h^{-1}(L)$;
Leia-se "h inverso de L", é o conjunto de strings w em ∑* tal que h(w) está em L.

Figura 3 – Um homomorfismmo aplicado no sentido direito e inverso

Teorema 7:	Se L é um homomorfismo do alfabeto ∑ para o alfabeto T, é L é uma linguagem regular sobre T, então $h^{-1}(L)$ também é uma linguagem regular.

2 Propriedades de Decisão das Linguagens Regulares

"As linguagens regulares, ao contrário das linguagens livres de contexto, não sofrem com problemas de **indecibilidades**, ou seja, em termos práticos as principais características que caracterizam as linguagens regulares são **decidíveis**".

- Para muitas das questões que formulamos, só existem algoritmos para a classe de linguagens regulares. As mesmas questões se tornam "indecidíveis" quando propostas com o uso de notações mais "expressivas" que as desenvolvidas para as linguagens regulares.
- A linguagem típica é infinita, e assim não é possível apresentar os strings da linguagem a alguém e formular uma pergunta que exija a inspeção do conjunto infinito de strings. Em vez disso, apresentaremos uma

linguagem fornecendo uma das representações finitas que desenvolvemos parra ela: um **DFA**, um **NFA**, um **ε-NFA** ou uma **expressão regular**.

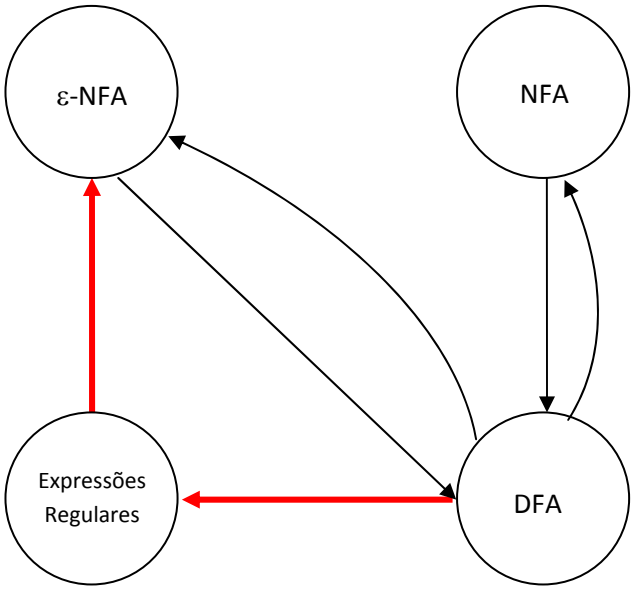


Figura 4 – Plano para mostrar a equivalencia entre ass quatros notações diferentes para linguagens regulares

- Embora existam algoritmos para quaisquer das conversões, às vezes não estamos interessados apenas na possibilidade de fazer uma conversão, mas no tempo que ela demora. Em particular, é importante distinguir entre algoritmos que demoram um tempo exponencial daqueles que demoram um tempo linear, quadrático ou algum polinômio de grau pequeno do tamanho de sua entrada.

Conversão de NFA's em DFA's

O tempo de execução da conversão de NFA em DFA, incluindo o caso em que o NFA tem ε-transições é $O(n^3 2^n)$. É claro que, na prática, é comum o número de estados criados ser muito menor que 2^n, e com frequência ser de somente **n** estados. Poderíamos enunciar o limite sobre o tempo de execução como $O(n^3 s)$, onde s é o número de estados que o DFA tem realmente.

Conversão de DFA's em NFA's

Essa conversão é simples e demora o tempo O(n) em um DFA de n estados.

Conversão de Autômatos em Expressões Regulares

Se primeiro convertermos um NFA em um DFA e depois convertermos o DFA em um expressão regular, isso pode demora um tempo de $O(n^3 4^{n^3 2^n})$, que é duplamente exponencial.

Conversão de Expressão Regular em Autômatos

A construção de um ε-NFA a partir de uma expressão regular leva um tempo linear no tamanho da expressão. Podemos eliminar ε-transições de um ε-NFA de n estados para criar um NFA ordinário em tempo $O(n^3)$, sem aumentar o numero de estados. Porém, a continuação até um DFA pode demorar um tempo exponencial.

- As três principais propriedades de decisões ou questões fundamentais sobre linguagens, sejam elas ou não regulares, são:
  1. A linguagem é vazia?
  2. Um determinado string w pertence à linguagem descrita?
  3. Duas descrições de uma linguagem realmente descrevem a mesma linguagem?
     Essa questão frequentemente é chamada "equivalência" de linguagens. Observe a profundeza desta pergunta, pois se é possível estabelecer a equivalência entre dois ou mais autômatos finitos será possível minimiza-lo.

Nota Importante: Para linguagens regulares há algoritmos para responder de forma positiva, sendo mais formal, há algoritmos decidíveis para responder as três questões anteriores. Veremos a seguir cada um deles.

2.1 Como testar o caráter vazio de linguagens regulares

- À primeira vista, a resposta à pergunta "a linguagem regular L é vazia?" é óbvia: Ø é vazia, e todas as outras linguagens regulares não o são. Porém, o problema não é enunciado como uma lista explícita dos strings de L. Em vez disso, temos alguma representação parra L e precisamos saber se essa representação denota a linguagem Ø.
- Se nossa representação é qualquer tipo de autômato finito, a questão do caráter vazio consiste em saber se existe um caminho qualquer desde o estado inicial até algum estado de aceitação. Neste caso, a linguagem é não vazia, por outro lado, se os estados de aceitação estão todos separados do estado inicial, então a linguagem é vazia.

O algoritmo anterior pode ser resumido por este processo recursivo:

Base: O estado inicial é certamente alcançável a partir do estado inicial.

Indução: Se o estado q é alcançável a partir do estado inicial, e se existe um arco desde q até p com qualquer rótulo (um símbolo de entrada ou ε, se o autômato é um ε-NFA), então p é alcançável.

Desta maneira, podemos calcular o conjunto de estados alcançáveis. Se qualquer estado de aceitação estiver entre eles, responderemos "não" (a linguagem do autômato não é vazia) e, caso contrário, responderemos "sim".

- As regras recursivas a seguir informam se uma expressão regular denota a linguagem vazia.

Base: Ø denota a linguagem vazia; **ε** e **a** para qualquer símbolo entrada **a** não denotam.

Indução: Suponha que R seja uma expressão regular. Há quatro casos a considerar, correspondentes aos possíveis modos de construção de R.

1. $R = R_1+R_2$. Então L(R) é vazia se e somente se $L(R_1)$ e $L(R_2)$ são ambas vazias.
2. $R = R_1R_2$. Então L(R) é vazia se e somente se $L(R_1)$ ou $L(R_2)$ é vazia.
3. $R = R^*_1$. Então L(R) não é vazia; ela sempre inclui pelo menos ε.
4. $R = (R_1)$. Então L(R) é vazia se e somente se $L(R_1)$ é vazia, pois elas são a mesma linguagem.

2.2 Como testar a Pertinência em uma Linguagem Regular

- A próxima questão importante é saber, dado um string w e uma linguagem regular L, se w está em L. Embora w seja representado explicitamente, L é representada por um autômato ou uma expressão regular.
- Se L é representado pó um DFA, o algoritmo é simples. Simule o DFA processando o string de símbolos de entrada w, começando no estado inicial. Se o DFA terminar em um estado de aceitação, a resposta é "sim"; caso contrário, a resposta é "não".
- Esse algoritmo é extremamente rápido. Se |w|=n, e se o DFA é representado por uma estrutura de dados adequada, como uma matriz bidimensional que seja a tabela de transições, então cada transição exigirá um tempo constante, e o teste inteiro levará o tempo O(n).
- Se L tiver qualquer outra representação além de um DFA, poderemos convertê-la em um DFA e executar o teste anterior.

2.3 Como testar a Equivalência e a Minimização de Autômatos

- Em contraste com as questões anteriores – caráter **vazio** e **pertinência** – cujos os algoritmos eram bastante simples, a questão de saber se duas descrições de duas **linguagens regulares** realmente definem a mesma linguagem envolve uma considerável mecânica intelectual.

- Aqui, uma questão pertinente, é discutirmos como testar se dois descritores de linguagens regulares são equivalentes, no sentido de que definem a mesma linguagem. Uma consequência importante desse teste é que existe uma forma para minimizar um DFA.
- Podemos tomar qualquer DFA e encontrar um DFA equivalente que tenha o número mínimo de estados. Esse DFA é essencialmente único: dados dois DFA's quaisquer com o número mínimo de estados que sejam equivalentes, sempre podemos encontrar uma forma para renomear os estados de modo que os dois DFA's se tornem igual.
- **Como testar a distinção de estados**: dois estados de um DFA são distinguíveis se existe um string de entrada que leva exatamente um dos dois estados para um estado de aceitação. Começando apenas com o fato de que pares consistem em um estado de aceitação e um de não-aceitação são distinguíveis, e tentando descobrir pares de estados distinguíveis adicionais encontrando pares cujos sucessores sobre um símbolo de entrada são distinguíveis, podemos descobrir todos os pares de estados distinguíveis.
- **Minimização de autômatos finitos determinísticos**: Podemos particionar os estados de qualquer DFA em grupos de estados mutuamente indistinguíveis. Elementos de dois grupos diferentes são sempre distinguíveis. Se substituirmos cada grupo por um único estados, obteremos DFA equivalente que tem poucos estados quanto qualquer DFA para a mesma linguagem.
- **J**.E.Hopcroft, "An n Log n algorithm for minimizing the states in a finite automaton". Academic Press, New York, pp. 189-196.

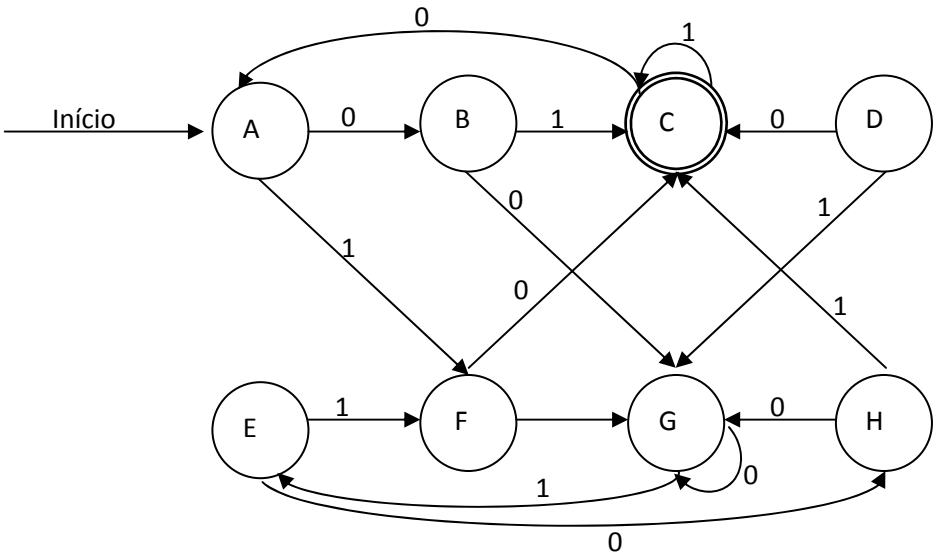


Figura 5 – Um autômato com estados equivalenntes

- Dizemos que os estados p e q são **equivalentes** se:
  - Para todos os strings de entrada w, $\hat{\delta}(p,w)$ é um estado de aceitação se e somente se $\hat{\delta}(q,w)$ é um estado de aceitação.

- Se dois estados <u>não são equivalentes</u>, então dizemos que eles são <u>distinguíveis</u>. Isto é, o estados p é <u>distinguível</u> do estado q se <u>existe pelo menos um string e</u> tal que um estados entre $\hat{\delta}(p,w)$ e $\hat{\delta}(q,w)$ é de aceitação, e o outro é de não-aceitação.
- <u>Qualquer string de entrada</u>, que leve os estado p e q a estados em que <u>somente um é de aceitação</u> é suficiente para provar que p e q <u>não são equivalentes</u>.

	A	B	C	D	E	F	G
B	X						
C	X	X					
D	X	X	X				
E		X	X	X			
F	X	X	X		X		
G	X	X	X	X	X	X	
H	X		X	X	X	X	X

Tabela 3 – Tabela de não-equivalência de estados

Nossa **meta** é entender quando dois estados distintos p e q podem ser substituídos por um único estado que se comporte como p e q.

- Para encontrar estados que sejam **equivalentes**, dedicaremos o melhor de nosso esforço a encontrar pares de estados que sejam **distinguíveis**. Talvez seja surpreendente, mas é verdade que, se fizermos o melhor possível de acordo com o algoritmo descrito a seguir, qualquer par de estados que considerarmos **distinguíveis** serão **equivalentes**.
- O algoritmo, a que nos referimos como o **algoritmo de preenchimento de tabela**, é uma descoberta recursiva de **pares distinguíveis** em um DFA $A=(Q,\sum,\delta,q_0,F)$.

Base:	Se p é um estado de aceitação e q é de não aceitação, então o par {p,q} é **distinguível**.
Indução:	Sejam p e q estados tais que, para algum símbolo de entrada a, $r=\delta(p,a)$ e $s=\delta(q,a)$ formam um par de estados conhecidos por serem **distinguíveis**. Então, {p,q} é um par de **estados distinguíveis**.

Teorema 8:	Se dois estados não são distinguíveis pelo algoritmo de preenchimento de tabela, então os estados são equivalentes.

Demonstração por redução ao absurdo (Hopcroft, pags: 168 e 169)

Testando a Equivalência de Linguagens Regulares

O algoritmo de preenchimento de tabela nos dá um modo fácil de testar se duas linguagens são iguais. Parra isso, faça o seguinte:

1. Suponha que cada uma das linguagens L e M seja representada de algum modo; por exemplo, uma delas por uma expressão regular e a outra por um NFA. Converta cada representação em um DFA.
2. Agora, imagine um DFA cujos os estados sejam a união dos estados dos DFA's correspondentes a L e M. Tecnicamente, esse DFA tem dois estados iniciais mas, na realidade, o estado inicial é irrelevante no que se refere á prova de equivalência de estados, e assim fazemos de qualquer estado p único estado inicial.
3. Agora, teste se os estados inicias dos dois DFA's originais são equivalentes, usando o **algoritmo de preenchimento de tabela**.
4. Se estes dois estados iniciais forem equivalentes, então L=M e, em caso contrário, então L≠M.

Exemplo:

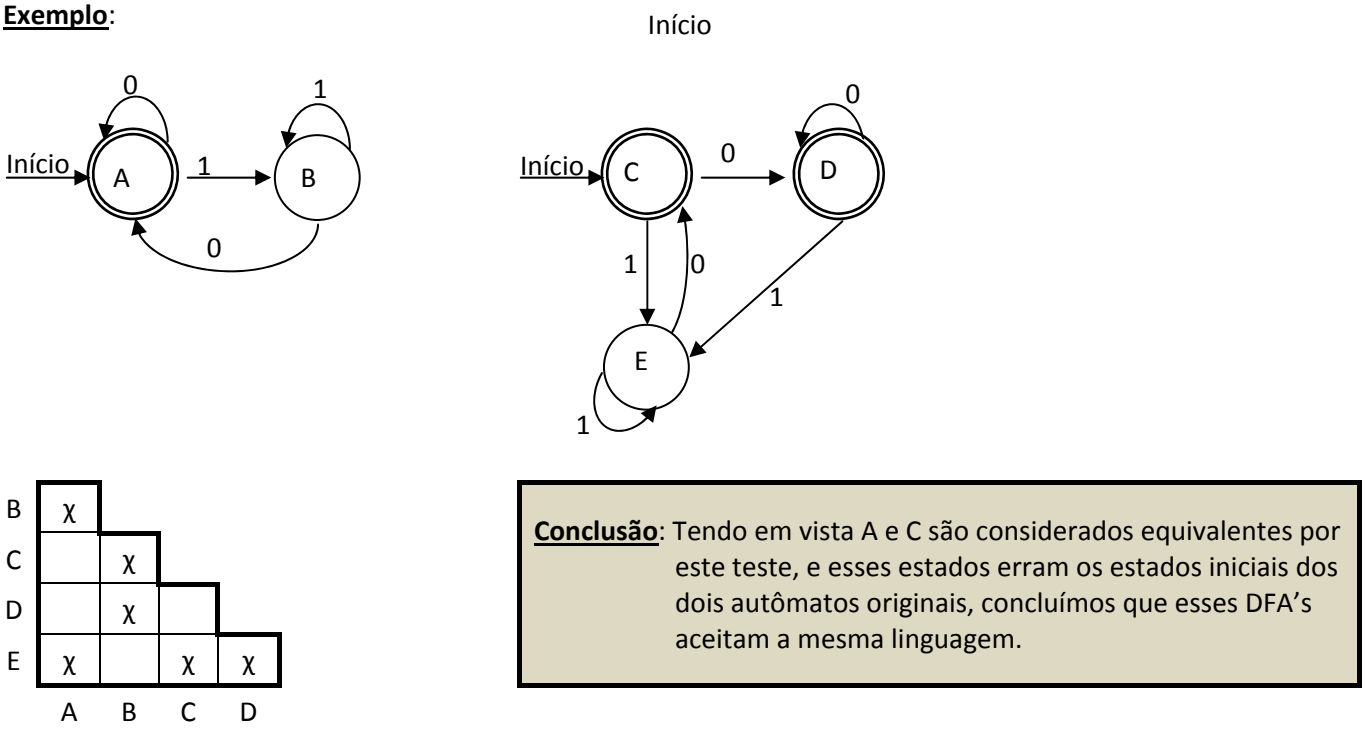


Figura 6 – Exemplo de Teste de Equivalência

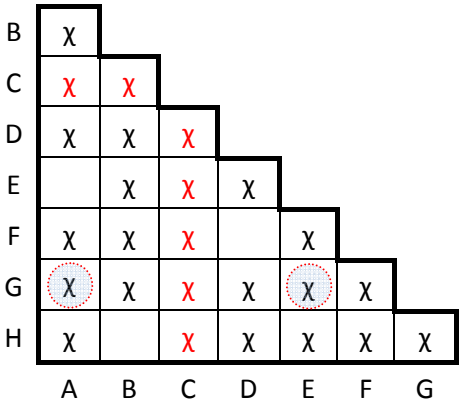
Conclusão: Tendo em vista A e C são considerados equivalentes por este teste, e esses estados erram os estados iniciais dos dois autômatos originais, concluímos que esses DFA's aceitam a mesma linguagem.

Minimização de DFA's

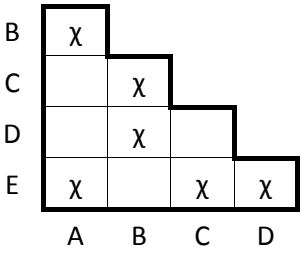
- Outra consequência importante de teste de equivalência de estados é que podemos "minimizar" DFA's. Isto é, para cada DFA podemos encontrar um DFA equivalente que tem tão poucos estados quanto qualquer DFA que aceita a mesma linguagem. Além disso, com exceção de nossa habilidade para denominar os estados com os nomes que preferirmos, esse DFA de número mínimo de estados é único para a linguagem. O algoritmo é dado a seguir:

1. Primeiro, elimine qualquer estado que não possa ser acessado a partir do estado inicial.

2. Em seguida, **particione** os estados restantes em **blocos**, de forma que todos os estados no mesmo bloco sejam equivalentes, e que nenhum par de estados de blocos diferentes seja equivalente. O **teorema 10** a seguir mostra que sempre podemos fazer tal partição.



Partição dos estados em blocos equivalentes = ({A,E},{B,H},{C},{DF},{G})



Partição dos estados em blocos equivalentes = ({A,C,D},{B,E})

Tabela 4 – Tabelas parra minimização

Teorema 9:	A equivalência de estados é transitiva. Isto é, se algum DFAA A-(Q,∑,δ,q_0,F) descobrimos que os p e q são equivalentes, e também descobrimos que q e r são equivalentes, então p e r também devem ser equivalentes. ***Demonstração por redução ao absurdo***

- Podemos utilizar o **teorema 9** para justificar o algoritmo óbvio de parcionamento de estados. Para cada estado q, construa um bloco que consiste em q e em todos os estados que são equivalentes a q. Devemos mostrar que os blocos resultantes formam uma **partição**, isto é, nenhum estado pertence a dois blocos distintos.
- A equivalência de estados **parciona** os estados, isto é, dois estados têm o mesmo conjunto de estados equivalentes (inclusive eles próprios) ou seus estados equivalentes são disjuntos.

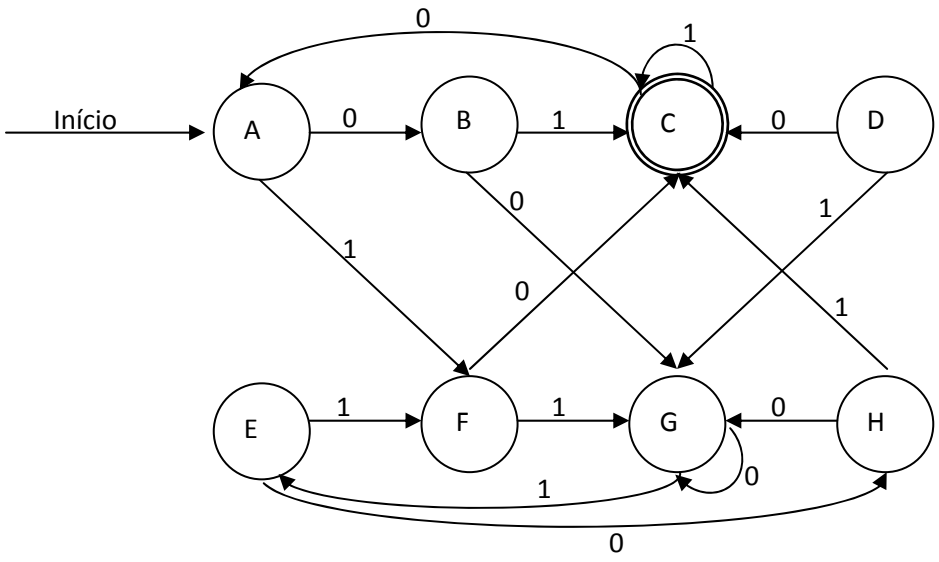
Teorema 10:	Se criarmos para cada estado q de um DFA um bloco consistindo em q e em todos os estado equivalentes a q, então os diferentes blocos de estados formaram uma **partição** do conjunto de estados. Isto é, cada estado está exatamente em um bloco. Todos os elementos de um bloco são equivalentes e nenhum par de estados escolhidos de diferentes blocos é equivalente.

- Agora, podemos enunciar de forma sucinta o algoritmo para minimizar um DFA $A=(Q,\sum,\delta,q_0,F)$.

1. Utilize o **algoritmo de preenchimento de tabela** para descobrir os pares de estados equivalentes.
2. Parcione o conjunto de estado Q em blocos de estados mutuamente equivalentes, pelo método descrito anteriormente (de acordo com os **teoremas 9 e 10**).
3. Construa o DFA número mínimo de estados equivalentes B utilizando os blocos como seus estados. Seja γ a função de transição de B. Suponha que S seja um conjunto de estados equivalentes de A, e que **a** seja um símbolo de entrada. Então, deve existir um bloco T de estados tais que, para todos os estados q em S, $\delta(q,a)$ é um elemento do bloco T. Caso contrário, o símbolo de entrada **a** tomará dois estados p e q de S como estados em diferentes blocos, e esses estados serão distinguíveis pelo teorema 10. Esse fato no leva a concluir que p e q não são equivalentes, e que eles não pertenciam ambos a S. Com consequência, podemos fazer $\gamma((S,a)=T$. Além disso:

   a. O estado inicial de B é o bloco que contém o estado inicial de A.
   b. O conjunto de estados de aceitação de B é o conjunto de blocos que contém estados de aceitação de A. Note que, se um estado de um bloco for de aceitação, então todos os estados desse bloco terão de ser de aceitação. A razão é que qualquer estado de aceitação é distinguível de qualquer estado de não-aceitação, e assim você não pode ter ao mesmo tempo estados de aceitação e de não-aceitação em um bloco de estados equivalentes.

Esquema Resumido de Minimização de DFA



	A	B	C	D	E	F	G
B	X						
C	X	X					
D	X	X	X				
E		X	X	X			
F	X	X	X		X		
G	X	X	X	X	X	X	
H	X		X	X	X	X	X

A partição dos estados em bloco equivalentes é ({A,E}, {B,H}, {C}, {D,F}, {G}).

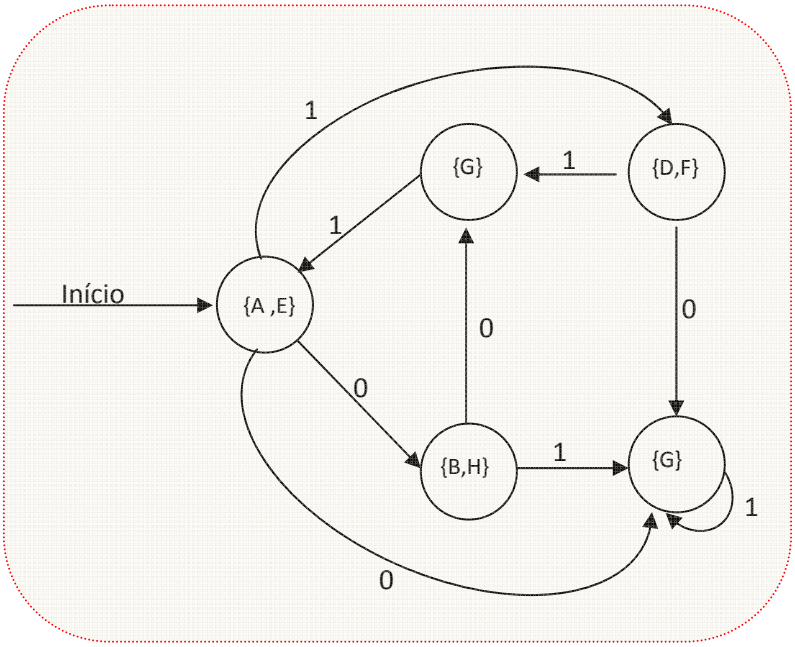


Figura 7 – Exemplo de DFA Minimizado

Por que o DFA Minimizado não pode ser Vencido

- Suponha que temos um DFA e o minimizamos para construir um DFA M, usando o método de parcionamento do teorema 10. Esse teorema mostra que não podemos agrupar os estados de A em grupos menos e ainda ter um DFA equivalente. Contudo, poderia haver outro DFA N, não relacionado a A, que aceitasse a mesma linguagem que A e M e que ainda tivesse menos estados que M? Podemos provar por contradição que N não existe.

Teorema 11:	Se A é um DFA e M é o DFA construído a partir de A pelo algoritmo descrito no enunciado do teorema 10, então M tem tão poucos estados quanto qualquer DFA equivalente a A.

- Portanto, o DFA de número mínimo de estado equivalente a A é único, exceto por uma possível mudança de nomes dos estados.

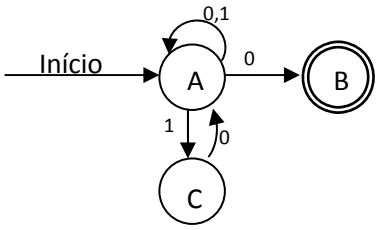


Figura 8 – Um NFA que não poode ser minnimizado por equivalência de estados

3 Propriedades de Fechamento das Linguagens Livres de Contexto

- Muitas dessas propriedades de fechamento terão paralelo com os teoremas vistos para linguagens regulares. Porém, há algumas diferenças.

Primeiro Teorema: Uma linguagem livre de contexto é fechada sobre a **operação de substituição**.

- Assim, a substituição de cada símbolo nos strings de uma linguagem por uma linguagem inteira é a operação chamada **substituição**. O teorema enunciado acima é fundamental para auxiliar na demonstração de algumas outras propriedades de fechamento para as Linguagens Livres de Contexto.
- Pode ser demonstrado que as Linguagens Livres de Contexto são fechadas sob **homomorfismos** e **homomorfismos inversos**. Entretanto, diferentemente das linguagens regulares, as Linguagens Livres de Contexto não são fechadas sob **intersecção** ou **diferença (complemento)**. Contudo, a intersecção ou a diferença entre uma linguagem Livre de Contexto e uma Linguagem Regular é sempre uma Linguagem Livre de Contexto.

Substituições: Seja ∑ um alfabeto e suponha que, para todo símbolo **a** em ∑, escolhemos uma linguagem L_a. Essas linguagens escolhidas podem estar sobre qualquer alfabeto, não necessariamente ∑ e não necessariamente iguais. Essa escolha de linguagens define uma função S (uma substituição) sobre ∑, e iremos nos referir a L_a como S(a) para cada símbolo **a**.

- Se $w=a_1\ a_2 \ldots a_n$ é um string em ∑*, então S(w) é uma linguagem de toso os strings $x_1\ x_2 \ldots x_n$, tais que o string x_1 está na linguagem $S(a_i)$, para i=1,2,…,n. Em outras palavras, S(w) é a concatenação das linguagens $s(a_1)s(a_n)$. Podemos estender ainda mais a definição de **s** para aplicá-la a linguagens: S(L) é a união de S(w) para todos os strings **w** em *L*.

Exemplo: Suponha que S(0)={a^nb^n|n≥1} e S(1)={aa,bb}. Assim, seja w=01. Então S(w) é a concatenação das linguagens S(0) S(1). Para ser exato, S(w) consiste em todos os strings das formas a^nb^naa e a^nb^{n+2}, onde n ≥ 1.

Teorema 01:	Se L é uma linguagem livre de contexto sobre o alfabeto ∑ e S é uma substituição em ∑ tal que S(a) é uma linguagem livre de contexto para cada **a** em um, então S(L) é uma linguagem livre de contexto.

- Existem diversas propriedades de fechamento familiares, que estudaremos para linguagens regulares e que podemos demonstrar para Linguagens livres de Contexto (CFL's) utilizando o teorema 011. Vamos listar todas elas em um único teorema.

Teorema 02:	As linguagens livres de contexto são fechadas sob as seguintes operações: 1. União 2. Concatenação 3. Fechamento (*) e Fechamento Positivo (+) 4. Homomorfismo

Prova: Cada uma destas quatro operações exige apenas uma configuração adequada para a substituição desejada. Casa uma dessas provas envolve a substituição de linguagens livres de contexto em outras linguagens livres de contexto, e assim produzem CFL's pelo teorema 01.

1. **União**: Sejam L_1 e L_2 CFL's. Então $L_1 \cup L_2$ é a linguagem S(L) onde L é a linguagem {1,2} e **S** é a substituição definida por $S(1)=L_1$ e $S(2)=L_2$.

   $L= L_1 \cup L_2 = \{L_1,L_2\}$ ou

   $L= 1 \cup 2 = \{1,2\}$ para $S(1)=L_1$ e $S(2)=L_2$

Assim, como a Linguagem $L_1 \cup L_2$ pode ser gerada pelas substituições $S(1)=L_1$ e $S(2)=L_2$, então pelo teorema 01 $L_1 \cup L_2$ é uma CFL.

Teorema 03:	Se L é uma Linguagem Livre de Contexto, então L^R também é.

- As linguagens livres de contexto não são fechadas sob a interseção. Por outro lado, existe uma afirmação mais fraca que podemos fazer a respeito da interseção.

Teorema 04: Se L é uma Linguagem Livre de Contexto e R é uma linguagem regular então L∩R é uma linguagem livre de contexto.

Teorema 05: As afirmativas a seguir são verdadeiras a respeito das linguagens livres de contexto L, L_1 e L_2, e para uma linguagem regular R.

1. L-R ≡ L ∩ $\overline{R}$ é uma linguagem livre de contexto.
2. $\overline{L}$ não é necessariamente uma linguagem livre de contexto ($\overline{L} \equiv \sum^* - L$).
3. $L_1 - L_2 \equiv L_1 \cap \overline{L}_2$ não é necessariamente livre de contexto.

- **Homomorfismo Inverso**: Se h é um homomorfismo e L é qualquer linguagem, então $h^{-1}(L)$ é o conjunto de strings w tais que h(w)está em L.

Teorema 06: Seja L uma linguagem livre de contexto e h um homomorfismo. Então $h^{-1}(L)$ é uma linguagem livre de contexto.

4 Propriedades de Fechamento das Linguagens Livres de Contexto

- Vamos considerar agora os tipos de perguntas que podemos responder sobre as linguagens livres de contexto.
- Em analogia com as linguagens regulares, nosso ponto de partida para uma pergunta é sempre alguma representação de uma linguagem livre de contexto: uma gramática ou um autômato de pilha (PDA).
- Vimos que podemos realizar a conversão entre gramáticas e PDA's, e assim podemos supor que temos uma ou outra representação de uma linguagem livre de contexto, usando a que for mais conveniente.
- Descobrimos que se pode decidir muito pouco sobre uma linguagem livre de contexto, os testes importantes que podemos realizar são para verificar se a linguagem é vazia e se um determinado string pertence a linguagem.
- Há vários tipos de problemas envolvendo as linguagens livres de contexto que são indecidíveis, isto é, que eles têm nenhum algoritmo. Os cinco principais problemas envolvendo linguagens livres de contexto e que sofrem de indecibilidade são:
  1. Uma das CFG G que representa a CFL é ambígua?
  2. Uma dada CFL é inerentemente ambígua?
  3. A interseção de duas CFL's é vazia?
  4. Duas CFL's são iguais?
  5. Uma dada CFL é igual a ∑*, onde ∑ é o alfabeto desta linguagem?

Todas estas perguntas sobre CFL's sofrem de indecibilidade!

- Normalmente aqui – em comparação com as linguagens regulares – embora também existam algoritmos para quaisquer das conversões de uma linguagem livre de contexto, às vezes não estamos interessados apenas na possibilidade de fazer uma conversão, mas no tempo que ela demora.

Complexidade da Conversão entre CFG's e PDA's

- Na discussão a seguir, seja n o comprimento da representação inteira de um PDA ou uma CFG. Várias conversões que vimos até agora são lineares no tamanho da entrada. Essas conversões são:
  1. Converter uma CFG em um PDA;
  2. Converter um PDA que aceita pelo estado final em um PDA que aceita por pilha vazia;
  3. Converter um PDA que aceita por pilha vazia em um PDA que aceita pelo estado final.
     - Por outro lado, o tempo de execução da conversão de um PDA em uma gramática é muito mais complexo.
  4. Existe um algoritmo de $O(n^3)$ que toma um PDA P cuja representação tem comprimento n e produz uma CFG de comprimento no máximo igual a $O(n^3)$. Essa CFG gera a mesma linguagem que P aceita por pilha vazia. Opcionalmente, podemos fazer G gerar a linguagem que P aceita por estado final.

Complexidade de Conversão para a Forma Normal de Chomsky

Considerando que os algoritmos de decisão podem depender da conversão inicial de uma CFG para a forma Normal de Chomsky, também devemos examinar o tempo de execução dos diversos algoritmos que usamos para converter uma gramática arbitrária em uma gramática CNF.

1. Usando-se o algoritmo adequado, a detecção dos símbolos alcançáveis e geradores de uma gramática podem ser feita no tempo O(n). A eliminação dos símbolos inúteis resultantes leva o tempo O(n) e não aumenta o tamanho da gramática.
2. A construção dos pares unitários e a eliminação das produções unitárias, demora o tempo $O(n^2)$, e a gramática resultante tem o comprimento $O(n^2)$.
3. A substituição de terminais por variáveis em corpos de produções, demora o tempo O(n) e resulta em uma gramática cujo comprimento é O(n).
4. O desmembramento dos corpos de produções de comprimento 3 ou mais em corpos de comprimento 2, também demora o tempo O(n) em uma gramática de comprimento O(n).

- A má noticia se refere a construção, em que eliminamos ε-produções, essa parte da construção levaria o tempo $O(2^n)$ e resultaria em uma gramática cujo comprimento é $O(2^n)$.
- Desse modo recomendamos, como uma etapa preliminar antes da eliminação de ε-produções, o desmembramento de todos os corpos de produções longos em uma sequencia de produções com corpo de comprimento 2.
- A construção explicada em aulas anteriores, para eliminar ε-produções, funcionará em corpos de comprimento no máximo 2, de tal modo que o tempo de execução seja O(n) e a gramática resultante tenha o comprimento O(n).
- A única etapa não-linear é a eliminação de produções unitárias. Como essa etapa é $O(n^2)$, concluímos o seguinte:

"Dada uma gramática G de comprimento n, podemos encontrar uma gramática equivalente na Forma Normal de Chomsky para G no tempo $O(n^2)$, a gramática resultante tem comprimento $O(n^2)$.

4.1 Como testar o caráter vazio de linguagens Livres de Contexto

- Já vimos o algoritmo para testar se uma Linguagem Livre de Contexto (CFL) L é vazia. Você saberia dizer qual desses algoritmos seria?

"Basta utilizar o algoritmo para decidir se o símbolo de início S de uma dada gramática livre de contexto G é gerador, isto é, se S deriva pelo menos um string. A linguagem L equivalente será vazia se e somente se S é não gerador".

Relembrando o algoritmo para detecção de todos os símbolos geradores:

- Seja G=(V,∑,P,S) uma gramática. Parra calcular os símbolos geradores de G, executamos a seguinte indução:

Base: todo símbolo de ∑ é sem duvida gerador, ele gera a si mesmo.
Indução: Suponha que exista uma produção A → α, e todo símbolo de α já seja conhecido como gerador. Então, A é gerador. Observe que esta regra inclui o caso em que α=ε, todas as variáveis que tem ε como corpo de uma produção seguramente são geradoras.

Teorema: O algoritmo anterior encontra todos os símbolos geradores de G e somente eles.

Nota: X é gerador se $X \overset{*}{\Rightarrow} w$ para algum string de terminais w. observe que o próprio string terminal w é gerador por zero etapas.

- Uma implementação ingênua do teste de símbolos geradores é $O(n^2)$.
- Contudo, existe um algoritmo mais cuidadoso que define uma estrutura de dados antecipadamente, a fim de fazer nossa descoberta de símbolos geradores levar apenas o tempo O(n).

Exemplo:

A → BCD
B → CD
D → A
S → ABCDE

Esta gramática é ou não vazia? Pela aplicação do algoritmo O(n) proposto por Hopcroft pg: 318-320, tem-se:

Esta é uma linguagem vazia! Então vejamos:

0º Passo: Gerar conjunto de variáveis distintas = {A,B,C,D,E,S}

1º Passo: É geradora?

A	?
B	?
C	?
D	Sim
E	?
S	?

Separar todas as variáveis que são geradoras diretamente de um terminal

{D}

Variável	Contador	Lista
A	2	BC
B	1	C
C	0	*fim*
D	0	*fim*
E	*indefinido*	*Indefinido*
S	4	ABCE

2º Passo: É geradora?

A	?
B	?
C	Sim
D	Sim
E	?
S	?

{C}

Variável	Contador	Lista
A	1	B
B	0	*fim*
C	0	*fim*
D	0	*fim*
E	*indefinido*	*Indefinido*
S	3	ABE

3º Passo: É geradora?

A	Sim
B	Sim
C	Sim
D	Sim
E	?
S	?

Variável	Contador	Lista
A	0	*fim*
B	0	*fim*
C	0	*fim*
D	0	*fim*
E	*indefinido*	*Indefinido*
S	1	E

Variáveis que faltam ser processadas ≡ VFSP={E,S}

Variáveis que ainda não são geradas ≡ VANG {E,S}

Como VFSP = VANG então, parar ou encerrar o algoritmo. Os pontos de interrogação remanescentes são os símbolos não geradores.

"No final do algoritmo, cada ponto de interrogação se transformará em "não", pois qualquer variável não descoberta pelo algoritmo é de fato não geradora".

(Hopcroft, pg: 318)

4.2 Como testar o Pertinência em uma linguagem Livre de Contexto

- Também podemos decidir a pertinência de um string w a uma CFL L. Existem várias maneiras ineficientes de realizar o teste, elas demoram um tempo exponencial em |w|, supondo0se que uma gramática ou um autômato de pilha para a linguagem L sejam dados e que seus tamanhos sejam tratados como uma constante, independente de w.
- Existe uma técnica muito mais eficiente, baseada na ideia de "programação dinâmica", que você também deve conhecer como "algoritmo de preenchimento de tabela" ou "tabulação". Esse algoritmo, conhecido como o Algoritmo CYK, começa com uma gramática CNF G=(V,T,P,S) para uma linguagem L. A entrada para o algoritmo é um string $w=a_1a_2…$na em T*. No tempo $O(n^3)$, o algoritmo constrói uma tabela que informa se w está em L.
- O nome deste algoritmo se deve a três pessoas, cada uma das quais descobriu independentemente a mesma ideia essencial: J.**C**ocke, D.**Y**ounger, T.**K**asami.
- No algoritmo CYK, construímos uma tabela triangular, como sugere a figura abaixo. O eixo horizontal coresponde às posições do string $w=a_1a_2…a_n$, que supomos aqui ter comprimento 5. A entrada de tabela X_{ij} é o conjunto de variáveis A tais que $A \overset{*}{\Rightarrow} a_ia_{i+1}…a_j$. Observe em particular que estamos interessados em saber se S pertence ao conjunto X_{1n}, porque isso é o mesmo que dizer que $S \overset{*}{\Rightarrow} w$, isto é, w está em L.

A tabela construída pelo algoritmo CYK

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

- Para preencher a tabela, trabalhamos linha por linha, de baixo para cima. Observe que cada linha corresponde a um comprimento de substrings, a linha inferior corresponde a strings de comprimento 1, a segunda linha de baixo para cima corresponde a strings de comprimento 2 e assim por diante, até a linha superior, que corresponde ao único substring de comprimento n, que é o próprio w.
- Aqui está o algoritmo $O(n^3)$ para calcular os valores X_{ij}'s:

Base: Calculamos a primeira linha como a seguir. Como o string que se inicia e termina na posição i é apenas o terminal a_i, e como a gramática esta em CNF, a única maneira de derivar o string a_i é usar uma produção da forma $A \rightarrow a_i$. Desse modo, X_{ii} é o conjunto de variáveis A tais que $A \rightarrow a_i$ é uma produção de G.

Indução: Suponha que queremos calcular X_{ij}, que está na linha j-i+1, e que calculamos todos os de X's nas linhas inferiores. Ou seja, conhecemos todos os strings mais curtos que $a_i a_{i+1} \ldots a_j$ e, em particular, conhecemos todos os prefixos e sufixos próprios desse string como j-i > 0 pode ser pressuposto (pois o caso de i=j é a base), sabemos que qualquer derivação $A \overset{*}{\Rightarrow} a_i a_{i+1} \ldots a_j$ deve começar com alguma etapa $A \rightarrow BC$. Então B deriva algum prefixo de $a_i a_{i+1} \ldots a_j$, digamos $B \overset{*}{\Rightarrow} a_i a_{i+1} \ldots a_k$, para algum k<j. Alem disso, C deve derivar então do restante de $a_i a_{i+1} \ldots a_j$, isto é, $C \overset{*}{\Rightarrow} a_{k+1} a_{k+2} \ldots a_j$. Concluímos que, para A estar em X_{ij}, devemos encontrar variáveis B e C, em um inteiro k, tais que:

1. $i \leq k < j$
2. B está em X_{ik}
3. C está em $X_{k+1,j}$
4. $A \rightarrow BC$ é uma produção de G

Encontrar tais variáveis A exige a comparação de no mínimo n pares de conjuntos calculados anteriormente: $(X_{ii}, X_{i+1,j})$, $(X_{i,i+1}, X_{i+2,j})$ e assim por diante, até $(X_{i,j-1}, X_{jj})$.

Teorema 7:	O algoritmo descrito calcula corretamente X_{ij} para todo i e j, desse modo, w está em L(G) se e somente se S está em X11n. Além disso, o tempo de execução do algoritmo CYK é $O(n^3)$.

Exemplo: Aqui estão as produções de uma gramática em CNF G:

S → AB|BC
A → BA|a
B → CC|b
C → AB|a

Testaremos a pertinência do string "baaba" à gramática aqui presente e que gera a linguagem L(G).

Teorema 7:	O algoritmo descrito calcula corretamente X_{ij} para todo i e j, desse modo, w está em L(G) se e somente se S está em X_{11}n. Além disso, o tempo de execução do algoritmo CYK é $O(n^3)$.

Exemplo: Aqui estão as produções de uma gramática em CNF G:

S → AB|BC
A → BA|a
B → CC|b
C → AB|a

Testaremos a pertinência do string "baaba" à gramática aqui presente e que gera a linguagem L(G).

Como S pertence a X_{15} então, o string "baaba" pertence a linguagem L(G)

{S,A,C}				
-	{S,A,C}			
-	{B}	{B}		
{S,A}	{B}	{S,C}	{S,A}	
{B}	{A,C}	{A,C}	{B}	{A,C}
b	a	a	b	a

Figura 9 – A tabela para o string "baaba" construída pelo algoritmo CYK

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

{S,A,C}				
-	{S,A,C}			
-	{B}	{B}		
{S,A}	{B}	{S,C}	{S,A}	
{B}	{A,C}	{A,C}	{B}	{A,C}
b	a	a	b	a

S → AB|BC
A → BA|a Gramática G na FNC
B → CC|b
C → AB|a

Primeira linha (regra Base): "X_{ii} é o conjunto de A tais que A → a_i é uma produção de G".

Os b's são gerados somente pela produção B→b

Os a's são gerados somente pelas produções A→a e C→a

Logo: $X_{11} = X_{44} = \{B\}$ e $X_{22} = X_{33} = X_{55} = \{A,C\}$

Segunda linha (começa a indução):

X_{12} : (X_{11},X_{22}) Portanto, X_{11}={B} e X_{22}={A,C}. Os corpos só podem ser BA ou BC.
ba b a

Assim, as únicas produções com estes corpos são: A → BA e S → BC.

Conclusão: $X_{12} = \{A,S\}$

X_{23}: (X_{22},X_{33}) Portanto, X_{22}={A,C} e X_{33}={A,C}

Corpos = {AA,AC,CA,CC}

Produções possíveis: B → CC **Conclusão**: $X_{23} = \{B\}$

X_{34}: (X_{33},X_{44}) Portanto, X_{33}={A,C} e X_{44}={B}

Corpos = {AB,CB}

Produções possíveis: S → AB, C → AB **Conclusão**: $X_{34} = \{S,C\}$

Terceira linha:

X_{24} : (X_{22},X_{34}), (X_{23},X_{44}), Portanto, X_{22}={A,C} e X_{34}={S,C}.

Corpos = $X_{22}X_{34}$ = {AS,AC,CS,CC}

Portanto, X_{23}={B} e X_{44}={B}.

Corpos = $X_{23}X_{44}$ = {BB}

Total de corpos = $X_{23}X_{44}$ ∪ $X_{23}X_{44}$ = {AS,AC,CS,CC,BB}

Produções possíveis B → CC **Conclusão**: X_{24}={B}

X_{15}: (X_{11},X_{25}), (X_{12},X_{35}), (X_{13},X_{45}) e (X_{14},X_{55})

(X_{11},X_{25}): X_{11}={B} e X_{25}={S,A,C} Logo $X_{11}X_{25}$={BS,BA,BC}

(X_{12},X_{35}) : X_{12}={S,A} e X_{35}={B} Logo $X_{11}X_{35}$={SB,AB}

(X_{13},X_{45}) : X_{13}={∅} e X_{45}={S,A} Logo $X_{13}X_{45}$={∅}

(X_{14},X_{55}) : X_{14}={∅} e X_{45}={A,C} Logo $X_{14}X_{55}$={∅}

Total de Corpos = $X_{11}X_{25}$ ∪ $X_{12}X_{35}$ ∪ $X_{13}X_{45}$ ∪ $X_{14}X_{55}$ = {BS,BA,BC,SB,AB}

Produções Possíveis = A → BA, S → BC, S → AB e C → AB

Conclusão: X_{15} = {S,A,C}

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Automata de pilha e LLCs
O lema do bombeamento para LLCs

Índice

Figuras

Propriedades das Linguagens Regulares

Explorar as propriedades das linguagens regulares nos conduzirá à algumas propriedades interessantes:

1. Estabelecer um modo de provar que certas linguagens não são regulares (lema do bombeamento);
2. Estabelecer as chamadas propriedades de fechamento, por exemplo, a interseção de duas linguagens regulares também é regular;
3. Estabelecer as propriedades de decisão, ou seja, um algoritmo para decidir se dois autômatos definem a mesma linguagem. Uma consequência de nossa habilidade para resolver essa questão é que podemos "minimizar autômatos", isto é, encontrar um equivalente a um dado autômato que tenha o mínimo de estados possível.

O Pumping Lemma para Conjuntos Regulares

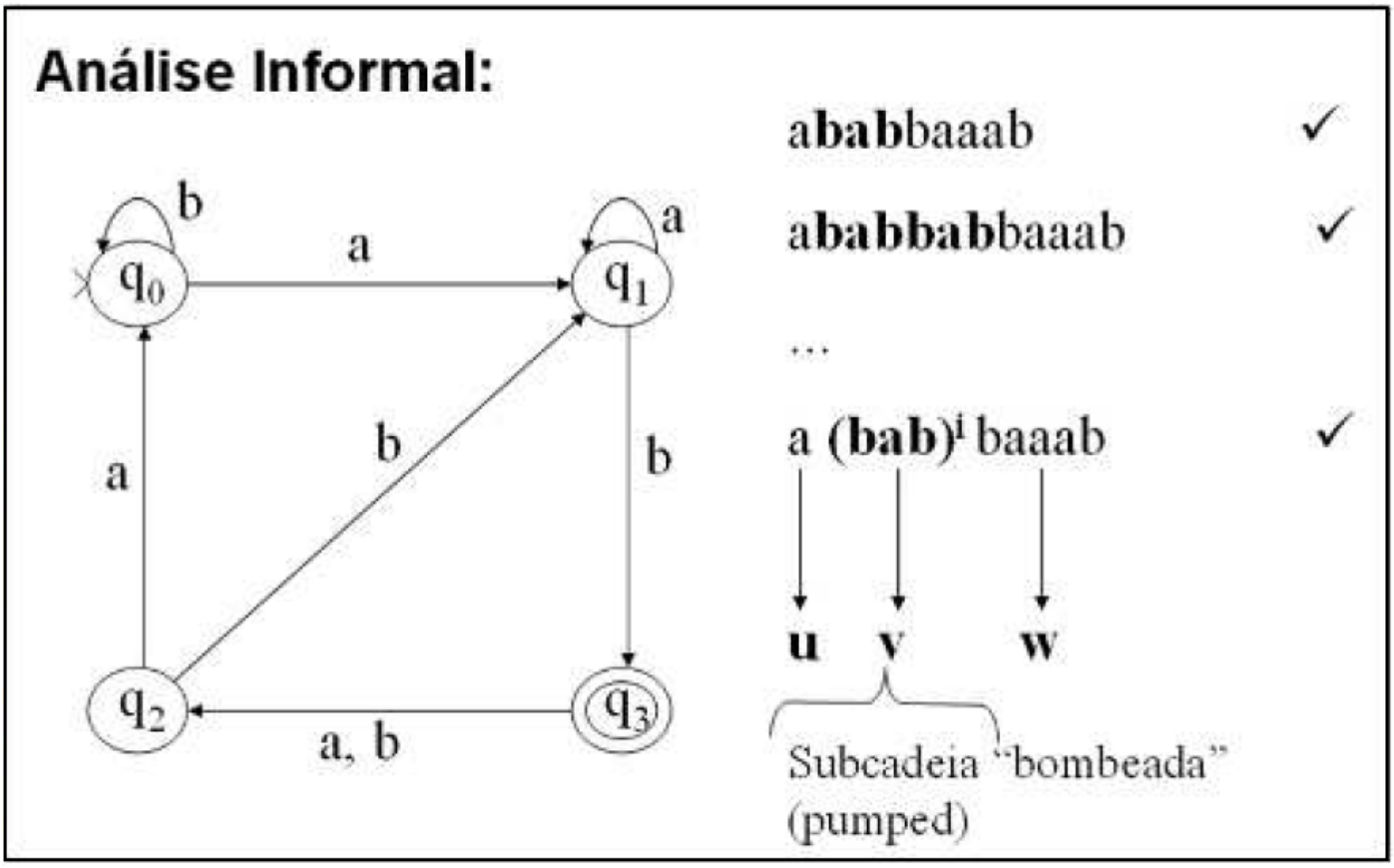


Figura 1 – Pumping Lemma para Conjuntos Regulares

Teorema 1: (O Lema de Bombeamento para linguagens regulares). Seja ***L*** uma linguagem regular. Então, existe uma constante ***n*** (que depende de ***L***) tal que, para todo string ***w*** em ***L*** tal que $|w| \geq n$, podemos dividir ***w*** em três strings, w = xyz, tais que:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. Para todo $k \geq 0$, o string xy^kz também está em L.

Interpretação: Sempre poderemos encontrar um string não vazio ***y*** não muito longe do início de ***w*** que pode ser "bombardeado", ou seja, poderemos repetir ***y*** qualquer numero de vezes, ou excluí-lo (o caso de k=0), mantendo o string resultante na linguagem ***L***.

Exemplo 1: Se $1010 \in L$ então,

1010, 101010, 101010, etc $\in$ L, para ***L*** uma linguagem regular.

Prova: (por construção)

H_1: suponha que ***L*** seja regular, ou seja, L=L(A) parra algum DFA A;

H_2: suponha que ***A*** tenha n estados;

H_3: seja ***w*** qualquer string de aceitação de comprimento ***n*** ou maior, por exemplo, $w = a_1 a_2 \ldots a_m$, onde m $m \geq n$ e cada a_i é um símbolo de entrada;

H_4: para i = 1,2,...,n+1, define-se estado P_i como $\hat{\delta}(q_0, a_1 a_2 \ldots a_i)$, onde δ é a função de transição de ***A***, q_0 é o estado inicial de ***A*** e P_i o estado em que ***A*** se encontra depois de ler os primeiros ***i*** símbolos de ***w***. Nesta convenção observe que $P_0 = q_0$.

- Assim, aplicando o princípio da casa de Pombo, não é possível que os ***n+1*** diferentes $P_{i's}$ de H_4 para i=1,2,...,n+1 sejam distintos, pois só existem ***n*** estados diferentes (há mais pombos que compartimentos). Deste modo, pode-se encontrar dois inteiros diferentes ***i*** e ***j***, com $1 < i < j \leq n+1$, tais que $P_i = P_j$. Agora, pode-se dividir w = xyz como a seguir:

1. $x = a_1 a_2 \ldots a_{i-1}$
2. $y = a_i a_{i+1} a_{i+2} \ldots a_j$
3. $z = a_{j+1} a_{j+2} \ldots a_m$

- ou seja, ***x*** nos leva a P_i uma vez; ***y*** nos leva de P_i de volta a P_i (pois P_i também é P_j) e ***z*** é o restante de ***w***.

- Note que ***x*** e ***z*** podem ser vazios, mas ***y*** não pode ser vazio, pois ***i*** é estritamente menor que ***j***.
- Agora considere o que acontece se o autômato ***A*** recebe a entrada xy^kz para qualquer $k \geq 0$:
  - Se k = 0 então w = xz é aceita sem repetir nenhum estado;
  - Se k > 0 então $w = xy^kz$ é aceita repetindo P_i k vezes. (c.q.d.)

Teorema de bombeamento para Conjunto Regulares

Lema: Seja ***G*** grafo de estados de um AFD M com ***k*** estados. Então qualquer caminho de comprimento ***k*** em ***G*** contém um ciclo.

Corolário: Seja ***G*** o grafo de estados de um DFA M com ***k*** estados, e seja ***p*** um caminho de comprimento maior ou igual a ***k***. Então ***p*** pode ser decomposto em subtrajetórias ***x***, ***y*** e ***z*** (p=xyz), em que o comprimento de xy é menor ou igual a ***k*** e ***y*** é um ciclo.

Lema do Bombeamento: Seja *L* uma linguagem regular aceita por um DFA M com ***k*** estados. Seja ***w*** uma cadeia em ***L*** com comprimento maior ou igual a ***k***. Então ***w*** pode ser escrito como xyz, onde:

$$|xy| \leq k,\ |y| > 0\ E\ xy^iz \in L,\ \forall\ i \geq 0$$

Se $w \in L$ e $|w| \geq k$ então, $\underbrace{w = xyz}_{1}$, $\underbrace{|wy| \leq k}_{2}$, $\underbrace{y \neq \varepsilon}_{3}$ e $\underbrace{xy^iz \in L}_{4}$

Se uma linguagem é realmente uma linguagem regular então, não há outra saída senão ter que "engolir" o lema do bombeamento. Isto implica também a todo DFA M tem que "engolir" o mesmo lema!

$L = \{a^ib^i \mid i \geq 0\}$

$L = \{\varepsilon, ab, aabb, aaabbb, ...\}$

Demonstração: Assuma, por absurdo, que L é regular. Sendo assim então, ***l*** deverá ser aceita por algum AFD de ***k*** estados. O teorema de bombeamento, para este caso, toma a seguinte forma:

Se $w \in L$ e $|w| \geq k$ então, $\underbrace{w = xyz}_{1}$, $\underbrace{|wy| \leq k}_{2}$, $\underbrace{y \neq \varepsilon}_{3}$ e $\underbrace{xy^iz \in L}_{4}$

Desta forma, se for encontrado um string ***w*** que decomposto de acordo com o teorema de bombeamento não pertencer a ***L***, chegar-se-á a um absurdo – que por exclusão – fica assim demonstrado que ***L*** não é regular!

1. **Escolher o string (deve estar contida em *L*)**: $w = a^kb^k$ ($w \in L$ e $|w|=2k$)
2. **Desenvolver algebricamente o string escolhido**: $w = xyz = a^kb^k = a^ia^ja^{k-i-j}b^k$ para $i + j \leq k$ e $j > 0$

Exemplo 2: i = 2, j = 3 e k = 6 (i+j = 2+3 = 5 < k = 6 e j > 0)

$a^6b^6 = a^2a^3a^1b^6 = a^6b^6$ (ok!)

Se $w = \underbrace{a^i}_{x} \underbrace{a^j}_{y} \underbrace{a^{k-i-j}b^k}_{z}$ então, $xy^2z = a^ia^ja^ja^{k-i-j}b^k = \underbrace{a^ia^ja^{k-i-j}}_{a^k}a^jb^k = a^ka^jb^k$

Logo, $xy^2z = a^ka^jb^k \notin L$ e L não é regular.

$L = \{z \in \{a,b\}^* \mid |z| \text{ seja um quadrado perfeito}\}$

Demonstração:

Se $w \in L$ e $|w| \geq k$ então, $\underbrace{w = xyz}_{1}$, $\underbrace{|wy| \leq k}_{2}$, $\underbrace{y \neq \varepsilon}_{3}$ e $\underbrace{xy^iz \in L}_{4}$

3 é um ciclo

Teorema do Bombeamento

1. **Escolher o String**: $w = xyz \mid |w| = k^2$ (quadrado perfeito)
2. **Manipular**: Se w = xyz então $|w| = k^2$ (pela própria hipótese do problema)

Teorema bombeamento: 2 $|xy| \leq k$ e 3 $y \neq \varepsilon \Rightarrow 0 < |u| \leq k$

Logo,

$xy^2z \leq |xyz| + |z| \leq k^2 + k$

$xy^2z \leq \underbrace{k^2+k}_{} < \underbrace{\text{quadrado perfeito } (k^2 + 2k + 1)}$

Exemplo 3: $L_{pr} = \{w = 1^n \mid n \text{ é primo}\}$. Seja então, provar que esta linguagem não é regular.

- Se $w \in L_{pr}$ e $|w| \geq k+2$ então vale o lema de bombeamento, ou seja,

$\underbrace{w = xyz}_{1}, \underbrace{|wy| \leq k}_{2}, \underbrace{y \neq \varepsilon}_{3}$ e $\underbrace{xy^iz \in L_{pr}}_{4}$

Onde,

k ... número de estados do autômato que "aceitaria" tal linguagem;

$|w|=p \geq k+2$ ... aqui "p" é um número primo (tem que haver tal primo p, pois existe uma infinidade deles).

Logo, $w = 1^p$. **(a)**

Seja $|y| = m$ **(b)** {lembre-se que y não pode ser ε}

Então, $|xz| = |xyz| - |y| = p - m$ **(c)**

Se L_{pr} é regular deve valer também o lema do bombeamento, ou seja $xy^{p-m}z$ também deve pertencer a esta linguagem. Porém,

$|xt^{p-m}z| = |xz| + (p-m)|y| = (p-m)m(1+m)$ **(d)**

Conclusão: Parece que $|xt^{p-m}z|$ não é primo, pois tem dois fatores (m+1) e (p-1). Entretanto devemos ainda provar que nenhum dos fatores (p-m) ou (1+m) é igual a 1, pois nestes casos teríamos um número primo. Entretanto,

i. Para linguagens regulares necessariamente temos $y \neq \varepsilon$ e assim $|y| = n \geq 1$ ou $1 + m \geq 2$;

ii. $p-m > 1$ é realmente verdadeiro pois, $p \geq k+2$ (foi escolhido assim) e $m \leq k$, ou seja, $m = |y| \leq |xy| \leq k$. Assim, para $\{p \geq k+2$ no pior caso $p = k+2$ e $m=k$, $m \leq k\}$, ou seja $p-m = (k+2)-k$ ou $p-m>1$.

O Pumping Lemma para Linguagens Livres de Contexto

Seja L uma linguagem livre de contexto. Então existe uma constante n (dependente apenas da linguagem L) tal que, se $z \in L$ e $length(z) \geq n$, podemos escrever z=uvwxy de modo que:

- $length(vx) \geq 1$
- $length(vwx) \leq n$
- $u\ v^i\ w\ x^i\ y \in L$, para todo $i \geq 0$

Corresponde ao bombeamento de duas subcadeias separadas por w.

Como no caso de linguagens regulares, posso usar o *Pumping Lemma* para mostrar que uma linguagem não é livre de contexto. A ideia é prestar atenção especial às subcadeias bombeadas v e x, e, assumindo que a linguagem seja LC, tentar chegar a alguma contradição.

Exemplo 1:

Prove que L={ai bi ci | i ≥ 1} não é livre de contexto.

Suponha que L seja LC. Então, vale o *pumping lemma* e existe um n correspondente. Seja $z = a^n\ b^n\ c^n \in L$. Como length(z)≥n, então posso escrever:

z = uvwxy, com length(vx)≥1, length(vwx)≤n, e $uv^i\ wx^i\ y \in L$

i. vx não pode conter a's e c's, pois teria que colocar n b's entre eles (impossível, pois length (vwx)≤n).

ii. v e x não podem ter apenas a's, pois se assim fosse teríamos $uv^0\ wx^0\ y$ = uwy com n b's e n c's, mas menos do que n a's, já que alguns destes estariam em vx (pois length(vx)≥1). Assim, não poderia escrever uwy ∈ L como $a^k\ b^k\ c^k$.

iii. vx não pode conter apenas a's e b's, pois se assim fosse teríamos $uv^0\ wx^0\ y$ = uwy com n c's, mas menos do que n a's ou b's, já que alguns destes estariam em vx (pois length(vx)≥1). Assim, não poderia escrever uwy ∈ L como $a^k\ b^k\ c^k$.

Por raciocínio análogo a ii), v e x não podem ter apenas b's ou c's.

Por raciocínio análogo a iii), vx não pode conter apenas a's e c's ou b's e c's.

Assim, z=uvwxy ∈ L (por hipótese), mas v e x não podem ser formados por nenhuma combinação de símbolos do alfabeto da linguagem L: **contradição ⇒ L não é livre de contexto.**

Exemplo 3:

Prove que L={w ∈ **a*** | length(w) é primo} não é livre de contexto.

Suponha que L seja L.C. Então, vale o *pumping lemma* e existe um n correspondente: Seja $z = a^k \in L$, com k um número primo maior do que n. Como length(z)=k≥n, então posso escrever:

z = uvwxy, com length(vx)≥1, length(vwx)≤n, e $uv^i wx^i y \in L$

Seja m = length(u) + length(w) + length(y). Então, o comprimento de qualquer cadeia $z = uv^i wx^i y$ é:

length(uwy)+ length($v^i x^i$) = length(uwy) + i length(vx) = m + i (k-m)

Em particular, length($uv^{k+1} wx^{k+1} y$) = m + (k+1)(k-m) = k(k-m+1), que é divisível por k! **Contradição...**

APOSTILA

CT-200 AUTÔMATOS FINITOS E LINGUAGENS FORMAIS

Prof. Dr. Paulo Marcelo Tasinaffo

Automata de pilha e LLCs
O lema do bombeamento para LLCs

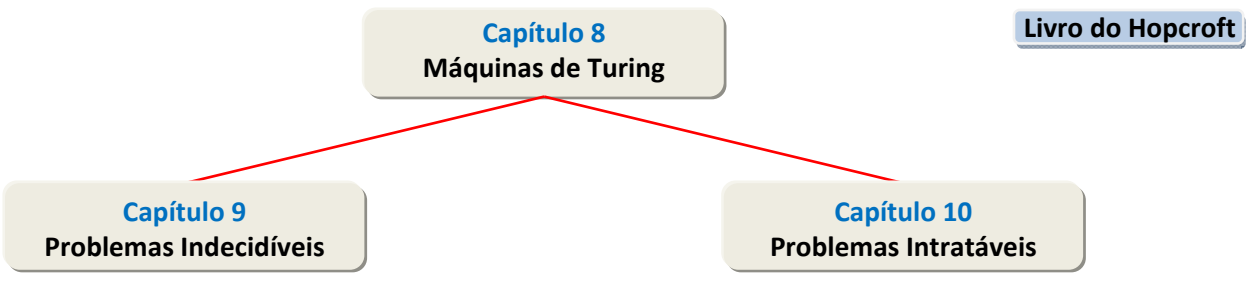
Índice

Figuras

Tabelas

Introdução às Máquinas de Turing

- Agora, começaremos a examinar a questão de quais linguagens podem ser definidas por qualquer dispositivo computacional. Essa questão é equivalente a pergunta o que os computadores podem fazer.
- Existem problemas específicos que não podemos resolver usando um computador. Esses problemas são chamados **indecidíveis**.
- A máquina de Turing é reconhecida há muito tempo como um modelo preciso daquilo que qualquer dispositivo físico de computação é capaz de fazer. Assim, é possível utilizarmos a máquina de Turing para desenvolver uma teoria de problemas "indecidíveis", isto é, problemas que nenhum computador pode resolver.
- Pode ser demonstrados que vários problemas fáceis de expressar são de fato indecidíveis. Um exemplo é determinar se uma dada gramática é ambígua, mas existem outros.
- Precisamos construir nossa teoria da indecibilidade a partir de um modelo muito simples de computador, chamado máquina de Turing. Esse dispositivo é essencialmente um autômato finito que tem uma única fita, na qual ele pode ler e gravar dados.
- A máquina de Turing é suficientemente simples para que possamos representar sua configuração de forma exata, usando uma notação simples muito semelhante às Descrições Instantâneas de um Autômato de Pilha PDA.
- Utilizando a notação da máquina de Turing, pode-se demonstrar que são indecidíveis certos problemas que parecem não estar relacionados à programação. Por exemplo, que o "Problema de Post" uma questão simples envolvendo duas listas de strings, é indecidível, e esse problema facilita a demonstração de que questões sobre gramáticas, como a ambiguidade, são indecidíveis.
- O propósito da teoria de problemas indecidíveis não é apenas estabelecer a existência de tais problemas, mas fornecer orientação aos programadores sobre o que eles poderiam ou não ser capazes de realizar através da programação.
- Esta teoria também tem grande impacto pragmático quando se discute problemas que, embora decidíveis, exigem grandes períodos de tempo para sua resolução. Esses problemas, chamados "problemas intratáveis", tendem a apresentar maior dificuldade para o programador e o projetista de sistemas do que os problemas indecidíveis.



Aspectos Históricos

- Na virada do século XX, o matemático D.Hilbert indagou se era possível encontrar um algoritmo para determinar a verdade ou a falsidade de qualquer proposição matemática, se existia um modo de descobrir se qualquer fórmula no cálculo de predicados de primeira ordem, aplicada a inteiros, era verdadeira.
- Entretanto, em 1931, K.Gödel publicou seu famoso teorema da incompletude. Ele construiu uma formula no cálculo de predicados aplicado a inteiros que afirmava que a própria fórmula nunca poderia ser provada nem contestada dentro do cálculo de predicados.
- Em 11936, A.M.Turing propôs a maquina de Turing como um modelo de qualquer computação possível.
- A hipótese improvável de que qualquer modo geral de computação nos permitirá calcular apenas as funções parcialmente recursivas é conhecido como hipótese de Church ou tese de Church-Turing.

Uma Notação para a Máquina de Turing

- A Máquina de Turing (MT) consiste em um controle finito, que pode se encontrar em qualquer estado de um conjunto finito de estados. Existe uma fita dividida em quadrado ou células, cada célula pode conter qualquer símbolo de um número finito de símbolos.

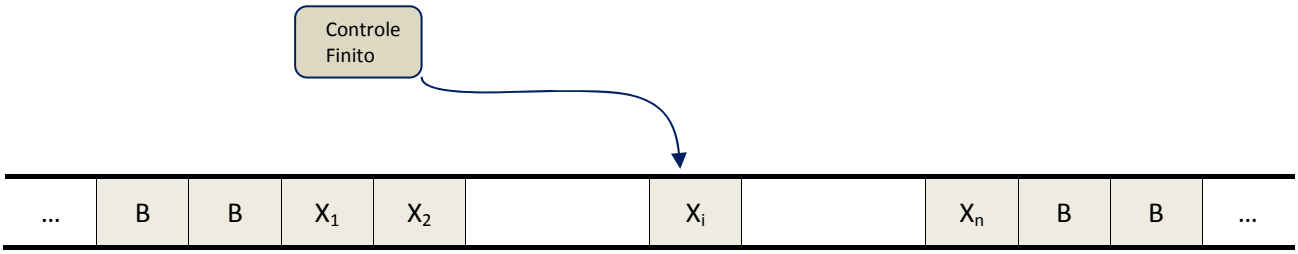


Figura 1 – Uma Máquina de Turing

1. Inicialmente, a entrada – um string de comprimento finito de símbolos escolhidos a partir do alfabeto de entrada – é colocado na fita.
2. O branco é um símbolo de fita, mas não um símbolo de entrada, e também pode haver outros símbolos de fita além dos símbolos de entrada e do branco.
3. Há uma cadeia da fita que fica sempre posicionada em uma das células da fita, Diz-se que a máquina de Turing está varrendo essa célula. Inicialmente, a cabeça da fita encontra-se na célula mais à esquerda que contém a entrada.
4. Um movimento da máquina de Turing é uma função do estado de controle finito e dos símbolo de fita varrido. Em um movimento, a máquina de Turing:

a. Mudará de Estados. O próximo estado poderá ser opcionalmente igual ao estado atual;
b. Gravará um símbolo de fita na célula varrida. Esse símbolo de fita substituíra qualquer símbolo que estava nessa célula. Opcionalmente, o símbolo poderá ser igual ao símbolo que ocupa a célula no momento;
c. Movimentará a cabeça da fita para a esquerda ou para a direita. Em nosso formalismo exige-se um movimento, e não permitindo que a cabeça permaneça estacionária.

- A notação formal que se utilize para uma Máquina de Turring (TM – Turing Machine) é semelhante ao que emprega para autômatos finitos ou autômatos de pilha. Assim, descreve-se uma TM pela tupla de sete valores:

$$M = (Q, \sum, \Gamma, q_0, B, F)$$

Cujos componentes têm os seguintes significados:

Q: O conjunto finito de estados do controle finito.

∑: O conjunto finito de símbolos de entrada.

Γ: O conjunto completo de símbolos de entrada.

δ: A função de transição.

Os argumentos δ(q,x) são um estado q e um símbolo de fita X. O valor de δ(q,X), se ele for definido, é uma tripla (p,Y,D), onde:

1. **P** é próximo estado em **Q**
2. **Y** é o símbolo, em **Γ**, gravado na célula que está sendo varrida, que substitui o símbolo que estava na célula.
3. **D** é uma direção ou sentido, seja **L** (esquerda) ou **R** (direita), para indicar respectivamente “esquerda” ou “direita”, informando-nos o sentido em que a cabeça se move.

q_0: O estado inicial, um elemento de **Q**, em que o controle finito se encontra inicialmente.

β: O símbolo branco. Esse símbolo está em **Γ**, isto é, não é um símbolo de entrada. O branco aparece inicialmente em todas as células da fita, exceto no número finito de células iniciais que contêm os símbolos de entrada.

F: O conjunto de estados finais ou estados de aceitação, um subconjunto de Q.

Descrições Instantâneas para Máquinas de Turing

- Para descrever formalmente o que uma máquina de Turing faz, precisamos desenvolver uma notação para configurações ou descrições instantâneas (ID's – instantaneous description), semelhante à notação que desenvolvemos para autômatos de pilha.
- Assim, além de representar a fita na descrição instantânea, devemos representar o controle finito e a posição da cabeça da fita. Para isso, incorporamos o estado à fita e o colocamos imediatamente à esquerda da célula varrida. Desse modo, usaremos o string $X_1X_2...X_{i-1}X_iX_{i+1}...X_n$ para representar uma descrição instantânea (ID) na qual:

  1. **Q** é o estado da máquina de Turing
  2. A cabeça da fita está varrendo o i-ésimo símbolo a partir da esquerda.
  3. $X_1X_2... X_n$ é a parte da fita entre não-branco mais à esquerda e o não-branco mais à direita.

Descreveremos movimentos de uma máquina de Turing $M=(Q,\sum,\Gamma,q_0,B,F)$ pela notação $\vdash_M$. Quando a TM M for submetida, usaremos apenas $\vdash$ para refletir movimentos. Como sempre, $\vdash_M^*$ ou simplesmente $\vdash^*$ será usado para indicar zero, um ou mais movimentos da TM M.

- Suponha que $\delta(q,X_i) = (p,Y,L)$, isto é, o próximo movimento é para a esquerda. Então,

  $X_1X_2...X_{i-1}X_iX_{i+1}...X_n \quad \vdash_M \quad X_1X_2...X_{i-2}pX_{i-1}YX_{i+1}...X_n$

Observe que este movimento reflete a mudança para o estado **p** e o fato de que a cabeça da fita fica posicionada agora na célula i-1. Há duas exceções importantes:

1. Se i=1, então M se move para o branco à esquerda de X_i. Nesse caso,

   $qX_1X_2...X_n \quad \vdash_M \quad pBYX_2...X_n$

2. Se i=n e Y=B, então o símbolo B gravado sobre X_n se junta à sequência infinita de brancos finais e não aparece na próxima ID. Desse modo,

   $X_1X_2...X_{n-1}qX_n \quad \vdash_M \quad X_1X_2...X_{n-2}pX_{n-1}$

- Aqui, o movimento reflete o fato de que a sua cabeça se moveu para a célula i+1. Mais umma vez, há duas exceções importantes:

1. Se i=n, então a (i+1)-enésima célula contém um branco, e essa célula não faz parte da ID anterior. Desse modo,

   $X_1X_2...X_{n-1}qX_n \quad \vdash_M \quad pX_2...X_n$

2. Se i=n e Y=B, então o símbolo B gravado sobre X_1 se junta à sequência infinita de brancos inicias e não aparece na próxima ID. Desse modo,

   $qX_1X_2...X_n \quad \vdash_M \quad pX_2...X_n$

Exemplo 1: Projete uma Máquina de Turing (TM) para reconhecer a linguagem $L=\{0^n1^n \mid n \geq 1\}$.

A especificação formal da (TM) é:

$$TM = (\overbrace{\{q_0, q_1, q_2, q_3, q_4\}}^{Q}, \overbrace{\{0,1\}}^{\Sigma}, \overbrace{\{0,1,X,Y,B\}}^{\Gamma}, \delta, B, \overbrace{\{q_4\}}^{F})$$

Onde δ é dado pela seguinte tabela,

Estado	0	1	X	Y	B
q_0	(q_1,X,R)	-	-	(q_3,Y,R)	-
q_1	$(q_1,0,R)$	$(q_2,0,L)$	-	(q_1,Y,R)	-
q_2	$(q_2,0,L)$	-	(q_0,X,R)	(q_2,Y,L)	-
q_3	-	-	-	(q_3,Y,R)	(q_4,B,R)
q_4	-	-	-	-	-

Tabela 1 – Tabela da Função Transição δ

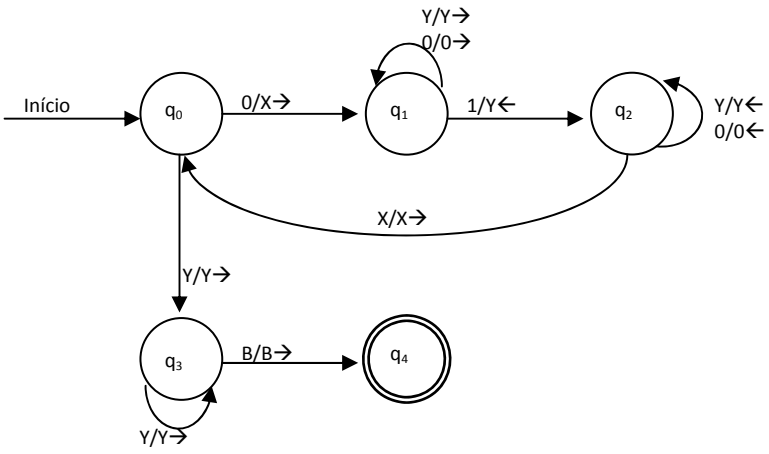


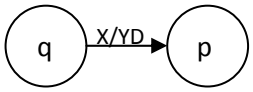
Figura 2 – Máquina de Turing

Um exemplo de computação de aceitação **Um exemplo de computação de não-aceitação**

$q_0$0011 ⊢ X$q_1$011 ⊢ X0$q_1$11 ⊢ X$q_2$0Y1 ⊢
q_2X0Y1 ⊢ X$q_0$0Y1 ⊢ XXq_1Y1 ⊢ XXY$q_1$1 ⊢
XXq_2YY ⊢ Xq_2XYY ⊢ XXq_0YY ⊢ XXYq_3Y ⊢
XXYYq_3B ⊢ XXYYq_4B

$q_0$0010 ⊢ X$q_1$010 ⊢ X0$q_1$10 ⊢ Xq_2Y0Y ⊢
q_2X0Y0 ⊢ X$q_0$0Y0 ⊢ XXq_1Y0 ⊢ XXY$q_1$0 ⊢
XXY0q_1B

Notas Adicionais sobre os Diagramas de Transição para Máquinas de Turing



- Um arco do estado q até o estado p é rotulado por um ou mais itens da forma X/YD, onde X e Y são símbolos de fita, e D é um sentido, L ou R. Ou seja, sempre que $\delta(q,X)=(p_1,Y_1,D)$, encontraremos o rótulo X/YD no arco de q para q.
- No entanto, em nossos diagramas, o sentido D é representado por indicando ← "esquerda" e → "direita".
- Como ocorre em outros tipos de diagramas de transição, representamos o estado inicial pela palavra "Início" e uma seta que entra nesse estado.
- Os estados de aceitação são indicados por círculos duplos.
- Desse modo, a única informação sobre a TM que não pode ser encontrada diretamente no diagrama é o símbolo usado para o branco. Devemos supor que o símbolo é B, a menos que haja alguma indicação em contrário.

Convenções de Notação para Máquina de Turing

- Os símbolos que utilizamos normalmente parra máquinas de Turing se assemelham aos de outros tipos de autômatos que vimos.

1. Letras maiúsculas no início do alfabeto representam símbolos de entrada.
2. Letras maiúsculas, em geral perto do fim do alfabeto, são usadas como símbolos de fita que podem ou não ser símbolos de entrada. Porém, B é geralmente usado para representar o símbolo branco.
3. Letras minúsculas perto do fim do alfabeto são strings de símbolos de entrada.
4. Letras gregas são strings de símbolos de fita.
5. Letras como q, p e letras vizinhas são estados.

A Linguagem de uma Máquina de Turing

- Sugerimos intuitivamente como uma máquina de Turing aceita uma linguagem. O string de entrada é colocado na fita, e a cabeça da fita começa no símbolo de entrada mais à esquerda. Se a TM entrar eventualmente em um estado de aceitação, a entrada será aceita e, caso contrário, não.
- De modo mais formal, seja M+(Q,∑,Γ,q_0,B,F) uma máquina de Turing. Então, L(M) é o conjunto de strings w em ∑* tais que q_0 w $\vdash^*$ αpβ para algum estado **p** em F e quaisquer strings de fita α e β.
- O conjunto de linguagens que podemos aceitar usando uma máquina de Turing é chamado com frequência de linguagens recursivamente enumeráveis ou linguagens RE.

Máquinas de Turing e Sua Parada

"Existe outra noção de aceitação usada comumente para máquinas de Turing: a aceitação por parada. Dizemos que uma TM para se ela entra em um estado **q**, varrendo um símbolo de fita X e não existe mais nenhum movimento nessa situação, isto é, δ(q,X) é indefinido".

- Sempre podemos supor que uma TM irá parar se ela aceitar. Isto é, sem mudar a linguagem aceita, podemos tornar δ(q,X) indefinido sempre que **q** for estado de aceitação. Em geral, afirmamos que:

  "Supomos que uma TM sempre para quando se encontra em um estado de aceitação".

- As linguagens que possuem máquinas de Turing que eventualmente param, independente de aceitação ou não, são chamadas recursivas, e consideraremos suas importantes propriedades mais tarde.
- As máquinas de Turing que sempre param, independente de aceitarem ou não, são um bom modelo de um algoritmo.

  "Se existir um algoritmo para resolver dado problema, diremos que o problema é decidível, e assim as TM's que sempre param desempenham um papel importante na teoria da decibilidade".

- Assim, podemos refinar a estrutura das linguagens recursivamente enumeráveis (RE) (aquelas que aceita por TMs) em duas classes.

A primeira classe: Corresponde ao que comumente imaginamos ser um algoritmo, tem uma TM que não apenas reconhece a linguagem, mas nos diz quando decide que o string de entrada não pertence à linguagem. Tal máquina de Turing sempre para independentemente do fato de alcançar ou não um estado de aceitação.

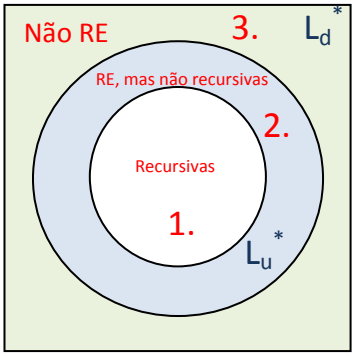
A segunda classe: Consiste nas linguagens RE que não são aceitas por nenhuma máquina de Turing com a garantia de parada. Essas linguagens são aceitas de uma forma inconveniente: Se a entrrada estiver na linguagem, eventualmente saberemos disso mas, se a entrada não estiver na linguagem então a máquina de Turing poderá continuar funcionando para sempre e nunca teremos certeza de que a entrada não será aceita mais tarde.

Linguagens Recursivas

Dizemos que uma linguagem L é recursiva se L=L(m) para alguma máquina de Turing M tal que:

1. Se w está em L, então M aceita (e portanto para).
2. Se w não está em L, então M para eventualmente, embora nunca entre em um estado de aceitação.

- Uma TM desse tipo corresponde à nossa noção informal de um algoritmo, uma sequencia bem definida de etapas que sempre termina e produz uma resposta.
- Imaginamos a linguagem K como um problema, como ocorrerá com frequência e então o problema L será chamado decidível se for c recursiva e será chamado indecidível se não for uma linguagem recursiva.



L_d ≡ A linguagem da diagramação

L_u ≡ A linguagem universal

1. As linguagens recursivas
2. As linguagens que são recursivamente enumeráveis, mas não-recursivas.
3. As linguagens não-recursivamente enumeráveis (não RE).

Exemplo 2:

Embora exista hoje em dia consideremos mais conveniente pensar em máquinas de Turing como reconhecedores de linguagens ou, de modo equivalente, solucionadores de problemas, a visão original de Turing sobre a máquina era de um computador de funções sobre inteiros.

Assim, apresentamos aqui uma Máquina de Turing para calcular a função $\dot{-}$, chamada menus ou subtração própria e definida por $m \dot{-} n = Max (m-n,0)$. Isto é, $m \dot{-} n$ é $m-n$ se $m \geq n$ e 0 se $m<n$.

- Uma TM que executa essa operação é especificada por
  $M=(\{q_0, q_1, \ldots, q_6\}, \{0,1\}, \{0,1,B\}, \delta, q_0, B)$

- Observe que, como essa TM não é usada para aceitar entradas, omitiremos o sétimo componente, o conjunto de estados de aceitação.
- M começará com uma fita que consiste em 0^m10^n cercada por brancos. M irá parar com $0^{m \dot{-} n}$ em sua fita, cercado por brancos.

A fita inicial (m=4 e n=2)

…	B	B	0	0	0	0	1	0	0	B	B	…

Terminará em,

…	B	B	0	0	B	B	B	B	B	B	B	…

A fita inicial (m=2 e n=4)

…	B	B	0	0	1	0	0	0	0	B	B	…

Terminará em,

…	B	B	B	B	B	B	B	B	B	B	B	…

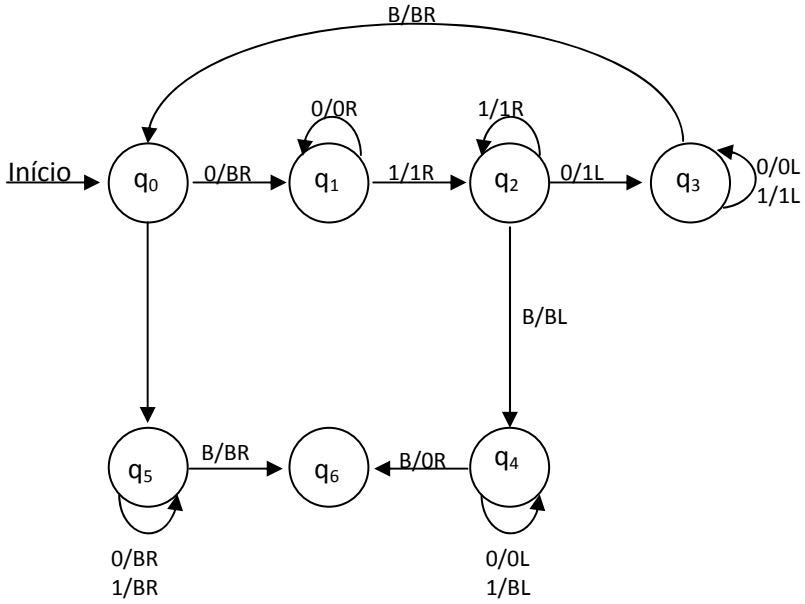


Figura 3 – Máquina de Turing para calcular a função :-

$q_0$0010 ⊢ B$q_1$010 ⊢ B0$q_1$10 ⊢ B01$q_2$0 ⊢ B0$q_3$11 ⊢ B$q_3$011 ⊢ q_3B011 ⊢ $q_0$011 ⊢ B$q_1$11 ⊢ B1$q_2$1 ⊢ B11q_2B ⊢ 1$q_4$1 ⊢ $q_4$1B ⊢ q_4BB ⊢ 0q_6B

q_0

...	B	B	0	0	1	0	B	B	...

Início

...	B	B	0	B	B	B	...

Término

$q_0$0100 ⊢ B$q_1$100 ⊢ B1$q_2$00 ⊢ B$q_3$110 ⊢ q_3B110 ⊢ B$q_0$110 ⊢ BB$q_5$10 ⊢ BBB$q_5$0 ⊢ BBBBq_5B ⊢ BBBBBq_2B

q_0

...	B	B	0	1	0	0	B	B	...

Início

...	B	B	B	B	B	B	...

Término

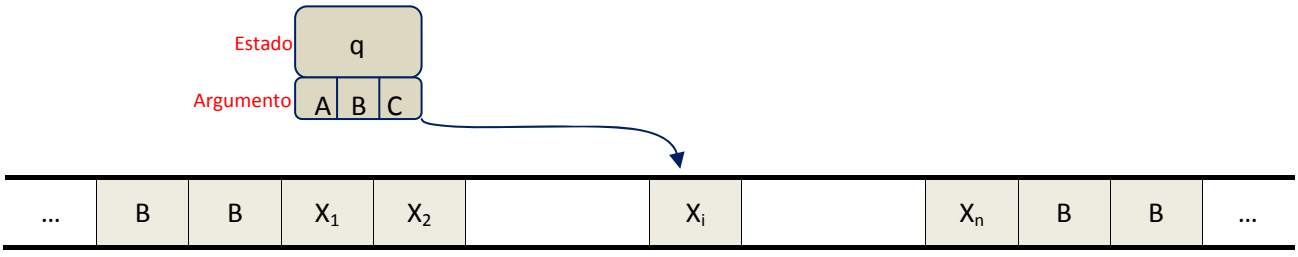
Técnicas de Programação para Máquinas de Turing

1. Nossa meta é lhe dar uma ideia de como uma máquina de Turing pode ser utilizada para calcular de maneira similar a um computador convencional.
2. Depois, queremos convencê-lo de que uma TM é exatamente tão poderosa quanto um computador convencional.
3. Essa habilidade "introspectiva" das máquinas de Turing e dos programas de computador é o que nos permite provar que existem problemas indecidíveis.
4. Para tornar mais clara a capacidade de uma TM, apresentaremos vários exemplos de como poderíamos imaginar a fita e o controle finito da máquina de Turing.
5. Nenhum desses artifícios estende o modelo básico da TM, eles são apenas conveniências da notação.

Exemplo 3:

Projetar uma máquina de Turing para representar a linguagem regular 01*+10*.

Nota: Podemos utilizar o **controle finito** não só para representar uma posição no "programa" da máquina de Turring, mas para conter uma quantidade finita de dados.



- Neste caso, o controle finito consiste não apenas em um estado de controle q, mas em três elemeentos de dados A, B, e C. Assim, devemos agora pensar no estado como um tupla. Para o exemplo acimma devemos pensar no estado como [q, A, B, C].

M=(Q, {0,1}, {0,1,B}. δ, $[q_0,B]$, $\{[q_1,B]\}$)

Q é $\{q_0,q_1\} \times \{0,1,B\}$

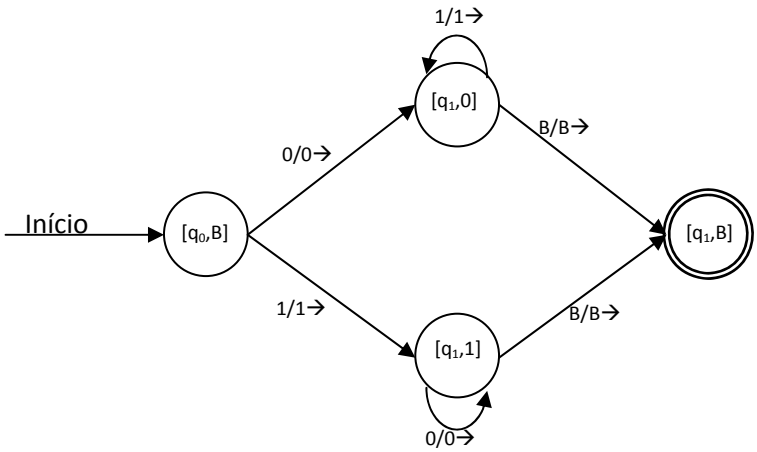
A função de transição δ de M é:

$\delta([q_0,B],a) = ([q_1,a],a,R)$ para a=0 ou a=1

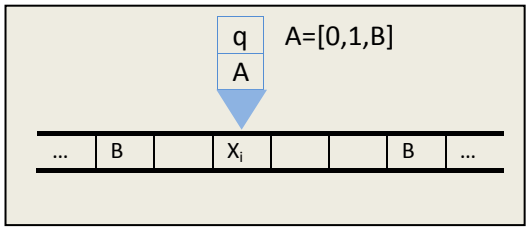
$\delta([q_1,a],a\text{-}) = ([q_1,a],\bar{a},R)$ para $\bar{a}$ o complemento de a

$\delta([q_1,a],B) = ([q_1,B],B,R)$ para a=0 ou a=1

L(01*+10*) = {0, 01, 011, 0111, ...} ∪ {1, 10, 100, 1000, ...}



Conjunto de Estados Q é:
$\{q_0,q_1\} \times \{0,1,B\}$



Estados	Símbolos		
	0	1	B
→[q0,B]	([q1,0],0,R)	([q1,1],1,R)	-
[q1,0]	-	([q1,0],1,R)	([q1,B],B,R)
[q1,1]	([q1,1],0,R)	-	([q1,B],B,R)
*[q1,B]	-	-	-

1. $[q_0,B]0111 \vdash 0[q_1,0]111 \vdash 01[q_1,0]11 \vdash 011[q_1,0]1 \vdash 0111[q_1,0]B \vdash 0111B[q_1,B]B$

   Para num estado de "Aceitação"

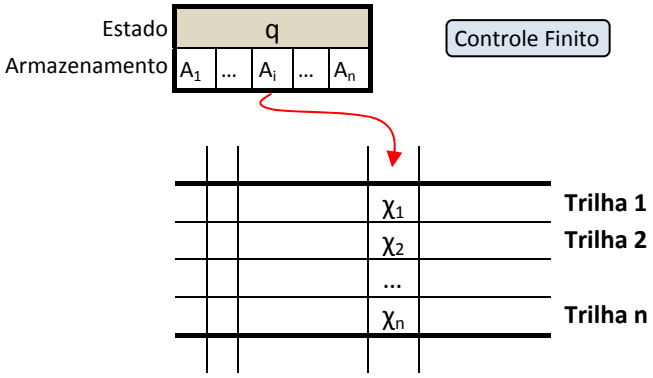
2. $[q_0,B]1000 \vdash 1[q_1,1]000 \vdash 10[q_1,1]00 \vdash 100[q_1,1]0 \vdash 1000[q_1,1]B \vdash 1000B[q_1,B]B$

   Para num estado de "Aceitação"

3. $[q_0,B]01011 \vdash 0[q_1,00]101 \vdash 01[q_1,0]01$

   Para num estado de "não-aceitação"

Máquinas de Turing com m dados no controle Finito e n trilhas



- Cada trilha pode conter um símbolo, e o alfabeto da fita da TM consiste em tuplas, com um componente para cada trilha.
- Com a técnica de armazenamento no controle finito, o uso de várias trilhas não amplia o que a máquina de Turing pode fazer. É simplesmente um modo de visualizar os símbolos de fita e imaginar que eles têm uma estrutura útil. Assim, seja a TM M, então:

$$\delta: \quad \delta([q_i,A'_1,\ldots,A'_m], [X'_1,X'_2,\ldots,X'_n]) = ([q_j,A''_1,A''_2,\ldots, A''_n], [X''_1,X''_2,\ldots,X''_n],D) \text{ para } D=\{L,R\}$$

No estado q_i, Se **M** estiver com os símbolos $A'_1, A'_2,\ldots, A'_m$ armazenamos no controle finito e os símbolos $X'_1,X'_2,\ldots,X'_n$ armazenamos nas **n** trilhas então, vá para o estado q_j, altere $A'_1, A'_2,\ldots, A'_m$ por $A''_1, A''_2,\ldots, A''_m$ nps dados do controle finito e altere $X'_1,X'_2,\ldots,X'_n$ por $X''_1,X''_2,\ldots,X''_n$, respectivamente, nas n trilhas. Obviamente poder-se-á ter $A'_i=A''_i$ ou $X'_j=X''_j$ para algum i, $1 \leq i \leq n$, ou para algum j, $1 \leq j \leq n$.

Exemplo 4: Projetaremos uma TM implementar a função multiplicação. Ou seja, nossa TM começara com 0^m10^n1 em sua fita e concluirá com 0^{nm} na fita.

Nota: Como ocorre com os programas em geral, é de grande ajuda pensar em máquinas de Turing como máquinas construídas a partir de uma coleção de componentes que interagem ou "sub-rotinas".

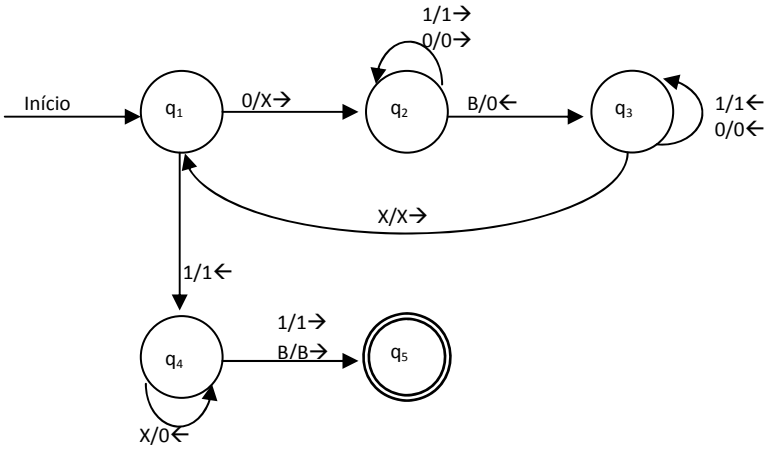


Figura 4 – A Sub-Rotina "Copy"

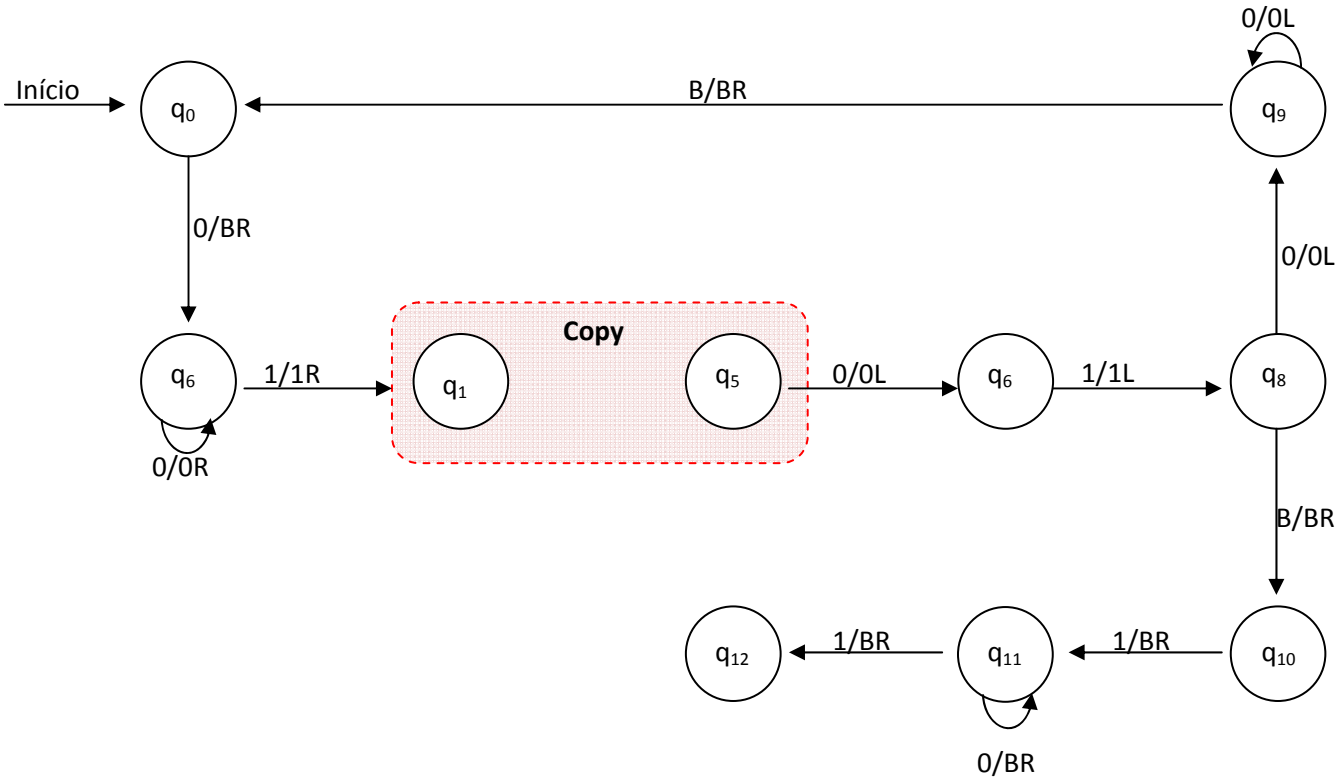


Figura 5 – Implementação da MT para função multiplicação

$q_0001001 \vdash q_601001 \vdash 0q_61001 \vdash 01q_1001 \vdash 01Xq_201 \vdash 01X0q_21 \vdash 01X01q_2B \vdash$
$01X0q_310 \vdash 01Xq_3010 \vdash 01q_3X010 \vdash 01Xq_1010 \vdash 01XXq_210 \vdash 01XX1q_20 \vdash 01XX10q_2B \vdash$
$01XX1q_300 \vdash 01XXq_3100 \vdash 01Xq_3100 \vdash 01XXq_1100 \vdash 01Xq_4X100 \vdash 01q_4X0100 \vdash 0q_4100100 \vdash$
$01q_50010 \vdash 0q_7100100 \vdash q_80100100 \vdash q_9B0100100 \vdash Bq_00100100 \vdash q_6100100 \vdash 1q_100100 \vdash$
$1q_100100 \vdash 1Xq_20100 \vdash 1X0q_2100 \vdash 1X01q_200 \vdash 1X010q_20 \vdash 1X0100q_2BB \vdash$
$1X010q_300 \vdash 1X01q_3000 \vdash 1X0q_31000 \vdash 1Xq_301000 \vdash 1q_3X01000 \vdash 1Xq_101000 \vdash$
$1XXq_21000 \vdash^* 1XX1000q_2B \vdash 1XX100q_300 \vdash^* 1Xq_3X10000 \vdash 1XXq_110000 \vdash$
$1Xq_4X10000 \vdash 1q_4X010000 \vdash q_410010000 \vdash 1q_50010000 \vdash q_710010000 \vdash$
$q_8B10010000 \vdash q_{10}10010000 \vdash Bq_{11}0010000 \vdash BBq_{11}010000 \vdash BBBq_710000 \vdash$
$BBBBq_20000BBB$

Para – não necessariamente num estado de aceitação – mas, mais do que isto, computa a multiplicação 2x2=4!

Entra em Copy

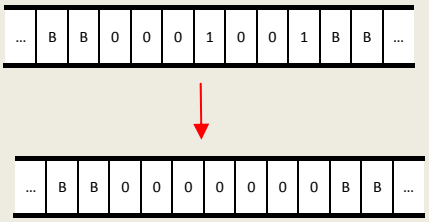
Sai de Copy

A estratégia é a seguinte:

Início:

0^m10^n1
$0^{m-1}10^n10^n$
$0^{m-2}10^n10^n0^n$
...
10^n10^{mn}
0

0001001
00100100
010010000
100100 00 00
00 00 00



Extensões para a Máquina de Turing Básica

1. **Máquinas de Turing de Várias Fitas**

- A máquina de Turing de várias fitas, é importante porque é muito fácil observar como uma (MT) de várias fitas pode simular computadores reais (ou outros tipos de máquinas de Turing), em comparação com o modelo de fita única.

- Ainda assim, as fitas extras nãoacrescentam nenhum poder ao modelo, pelo menos no que se refere à capacidade para aceitar linguagens.

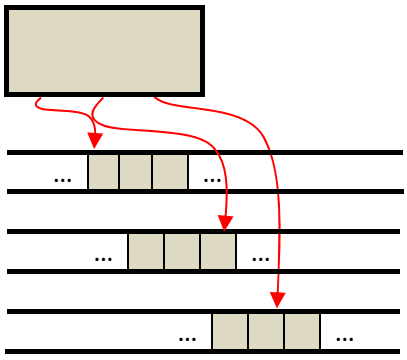


Figura 6 – Uma Máquina de Turing de Várias Fitas

- Uma Máquina de Turing de Várias Fitas é formada por: um controle finito (estado) e algum número finito de fitas. O conjunto de símbolos de fita inclui um branco e tem um subconjunto dos chamados símbolos de entrada, dos quais o branco não faz parte. O conjunto de estados inclui um estado inicial e alguns estados de aceitação.
- Inicialmente, uma (MT) com várias fitas possui a seguinte configuração:

1. A entrada, uma sequencia finita de símbolos de entrada, é colocada na primeira fita;
2. Todas as outras células de todas as fitas contêm brancos;
3. O controle finito está no estado inicial;
4. A cabeça da fita primeira está na extremidade esquerda da entrada;
5. Todas as outras cabeças de fitas estão em alguma célula arbitrária. Como fitas diferentes da primeira estão completamente em branco, não importa onde a cabeça está posicionada inicialmente, todas as células dessas fitas têm a mesma “aparência”.

- Em um movimento, a (MT) de várias fitas faz o seguinte:

1. O controle entre um novo estado, que pode ser igual ao estado anterior;
2. Em cada fita, um novo símbolo de fita é escrito na célula varrida. Quaisquer desses símbolos podem ser iguais ao símbolo que anteriormente estava lá;
3. Casa uma das cabeças de fita faz um movimento, que pode ser para a esquerda, para a direita ou estacionário. As cabeças se movem independentemente, assim, cabeças diferentes podem se movimentar em direções diferentes, e algumas podem não se movimentar de forma alguma. As direções são indicadas

agora por uma escolha de **L**, **R** ou **S**. No caso de máquina de uma fita, não permitimos que a cabeça ficasse estacionária, e assim a opção **S** não estava presente.

- As máquinas de Turing de várias fitas, como as MT's de uma fita, aceitam pela entrada em um estado de aceitação.

Teorema 1: Toda linguagem aceita por uma (MT) de várias fitas é recursivamente enumerável.

(Hopcroft, pg: 362)

- O tempo de execução da (MT) M sobre a entrada **w**, é o número de etapas que **M** faz antes de parar. Se **M** não parar em **w**, então o tempo de execução de **M** sobre **w** será infinito. A complexidade em tempo da (MT)**M** é a função T(n) que representa o valor máximo, para todas as entradas **w** com comprimento **n**, do tempo de execução de **M** sobre **w**. No caso de máquinas de Turing que não param em todas as entradas, T(n) pode ser infinito para alguns valores de **n** ou mesmo para todos eles.
- De fato, a (MT) de uma fita construída pode ter um tempo de execução muito maior que a (MT) de várias fitas.
- A (MT) de uma fita demora um tempo não maior que o quadrado do tempo tomado pela (MT) de várias fitas.

Teorema 2: O tempo tomado pela (MT) de uma fita para simular **n** movimentos da (MT) de **k** fitas – mencionado no teorema 1 – é $O(n^2)$

(Hopcroft, pg: 364)

2. Máquinas de Turing não-determinísticas

- Uma máquina de Turing não-determinística difere da variedade determinística que estudamos por ter uma função de transição δ tal que, para cada estado **q** e símbolo de fita **X**, δ(q,X) é um conjunto de triplas.

$$\{(q_1,Y_1,D_1), (q_2,Y_2,D_2), \ldots, (q_k,Y_k,D_k)\}$$

Onde **k** é qualquer inteiro finito.

- A máquina de Turing não-determinística pode escolher, em cada etapa, qualquer das tripas para ser o próximo movimento. Porém, ela não pode escolher um estado a partir de uma tripla, um símbolo de fita de outra e o sentido da terceira.
- A máquina de Turing **M** aceita uma entrada **w** se houver qualquer sequência de escolhas de movimentos que leve a "Descrição Instantânea" inicial com **w** como entrada a uma "Descrição Instantânea" com um estado de aceitação.

Teorema 3:	Se M_N é uma máquina de Turing não-determinística, então existe uma máquina de Tturring determinística M_D tal que $L(M_N)=L(M_D)$.

- Mote que a (MT) determinística construída pode demorar exponencialmente mais tempo que a (MT) não-determinística.
- A máquina de Turing não-determinística, uma extensão do modelo básico que pode fazer qualquer escolha em um conjunto finito de escolhas de movimentos em uma dada situação. Essa extensão também torna mais fácil "programar" máquinas de Turing em algumas circunstâncias, mas não acrescenta nenhum poder de definição de linguagens ao modelo básico.

Restrições para a Máquina de Turing Básica

- Vimos generalizações aparentes da máquina de Turing (Máquinas de Turing de várias fitas, Máquinas de Turing não-determinísticas) que não adicionam qualquer poder de reconhecimento de linguagens.
- Agora consideraremos alguns exemplos de restrições aparentes sobre a (MT) que também fornecem exatamente o mesmo poder de reconhecimento de linguagens.
- **Primeira Restrição**: Substituir a fita da (MT) que é infinita em ambos os sentidos por uma fita infinita somente à direita. Além disso proibi-se essa (MTT) restrita de imprimir um branco como símbolo de fita substituto.
- **Segunda Restrição**: Restringir as fitas da (MT) a se comportarem como pilhas.
- **Terceira Restrição**: Restringir mais ainda as fitas da (MT) a serem "contadores".
- O impacto dessa discussão é que existem vários tipos muito simples de autômatos que têm a capacidade total de qualquer computador.

1. **Máquinas de Turing com fitas semi-infinitas**

Teorema 4: Toda linguagem aceita por uma (MT) M_2 também é aceita por uma (MT) M_1 com as seguintes restrições:

1. A cabeça de M_1 nunca se move para a esquerda de sua posição inicial;
2. M_1 nunca grava um branco.

(Hopcroft, pg: 371)

2. **Máquinas de várias pilhas**

- Uma máquina de Turing pode aceitar linguagens que não são aceitas por nenhum Autômato de Pilha com uma única pilha. Ocorre que, se dermos duas pilhas ao Autômato de Pilha, ele poderá aceitar qualquer linguagem que uma (MT) pode aceitar.

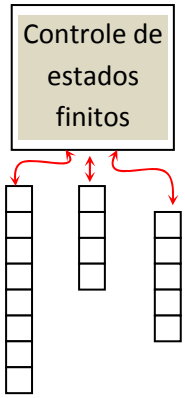


Figura 7 – Uma Máquina com três pilhas

- Uma máquina de k pilhas é um Autômato de Pilha determinístico com **k** pilhas. Ela obtém sua entrada, como um Autômato de Pilha, de uma fonte separada de entrada, em lugar de ter esta entrada colocada em uma fita ou na pilha, como a (MT).
- A máquina de várias pilhas tem um controle finito, que se encontra em um estado de um conjunto finito de estados. Ela tem um alfabeto de pilha finito, que utiliza parra todas as suas pilhas.
- Um movimento da máquina de várias pilhas se baseia:

1. No estado do controle finito.
2. No símbolo de entrada lido, que é escolhido no alfabeto finito de entrada. Alternativamente, a máquina de várias pilhas podem fazer um movimento usando a entrada ε mas, parra tornar a máquina determinística, não é possível existir a opção de um ε-movimento em qualquer situação.
3. No símbolo que ocupa o topo da pilha em cada uma de ssua pilhas.

- Ao se mover, a máquina de várias pilhas pode:

a) Mudar para um novo estado.

b) Substituir o símbolo do topo de cada pilha por um string de zero ou mais símbolos de pilha. Pode haver (e normalmente há) um string de substituição diferente para cada pilha.

- Desse modo, uma regra de transição típica para uma máquina de **k** pilha é semelhante a:
  $\delta(q,a,X_1,X_2,\ldots,X_k) = (p,\gamma_1, \gamma_2, \ldots, \gamma_k)$

- A interpretação dessa regra é que no estado **q**, comm X_i no topo da i-ésima pilha, para i=1,2,…,k, a máquina pode consumir **a** (um símbolo de entrada ou ε) de sua entrada, ir para o estado **p** e substituir X_i no topo da i-ésima pilha pelo string γ_1, para cada i=1,2,…,k. A máquina de várias aceita quando entra em um estado final.

- **Marcador de Fim**: Supõe-se que existem um símbolo especial **$**, chamado marcador de fim, que só aparece no fim da entrada (w$) e não faz parte dessa entrada. A presença do marcador de fim informa quando é consumida toda a entrada disponível. O marcador de fim facilita a simulação de uma máquina de Turing pela máquina de várias pilhas. Note que a (MT) convencional não precisa de nenhum marcador de fim especial, porque o primeiro branco serve para marcar o fim da entrada.

Teorema 5:	Se uma linguagem **L** é aceita por uma máquina de Turing, então l é aceita por uma máquina de duas pilhas.

- A ideia que está por traz deste teorema é que duas pilhas podem simular uma fita de máquina de Turing, comm uma pilha contendo o que está à esquerda da cabeça e a outra pilha contendo o que está à direita da cabeça.

3. **Máquinas de Contadores**

- A máquina de contadores tem a mesma estrutura da máquina de várias pilhas mas, em lugar de cada pilha, há um contador.
- Os contadores contêm qualquer inteiro não-negativo, mas só podemos distinguir entre contadores zero e diferentes de zero. Isto é, o movimento da máquina de contadores depende de seu estado, do símbolo de entrada e de quais dos contadores são iguais a zero, se houver algum.
- Em um movimento, a máquina de contadores pode:

a) Mudar de estado;

b) Somar ou subtrair 1 de qualquer de seus contadores, independentemente. Porém, um contador não pode se tornar negativo e, portanto não é possível subtrair q de um contador que seja atualmente 0.

Teorema 6:	Toda linguagem aceita por uma máquina de contadores é recursivamente enumerável.
Teorema 7:	Toda linguagem aceita por uma máquina de um contador é uma gramática livre de contexto.
Teorema 8:	Toda linguagem recursivamente enumerável é aceita por uma máquina de três contadores.
Teorema 9:	Toda linguagem recursivamente enumerável é aceita por uma máquina de dois contadores.

Resumo sobre Máquinas de Turing

- **A Máquina de Turing**: A máquina de Turing é uma máquina de computação abstrata com o poder dos computadores reais e de outras definições matemáticas do que pode ser calculado. A máquina de Turing (MT) consiste em um controle de estados finitos e em uma fita infinita dividida em celular. Cada célula contém um dentre um número finito de símbolos de fita, e uma célula está na posição atual da cabeça de fita. A (MT) executa movimento com base em seu estado atual e no símbolo de fita presente na célula varrida pela cabeça de fita. Em um movimento, ela muda de estado, sobregrava a célula varrida com algum símbolo de fita e move a cabeça uma célula para a esquerda ou direita.

- **Aceitação por uma Máquina de Turing**: A (MT) começa contendo em sua fita a entrada, um string de comprimento finito, e o restante da fita contém o símbolo de branco em cada célula. O branco é um dos símbolos de fita e a entrada é escolhida a partir de um subconjunto dos símbolos de fita – sem incluir o branco – chamados símbolos de entrada. A (MT) aceita sua entrada se entrar em um estado de aceitação.

- **Linguagens recursivamente enumeráveis**: As linguagens aceitas pelas Máquinas de Turing são chamadas linguagens recursivamente enumeráveis (RE). Desse modo, as linguagens (RE) são as linguagens que podem ser reconhecidas ou aceitas por qualquer tipo de dispositivo de computação.

Extensões para a Máquina de Turing Básica

1. **Armazenamento no controle finito**: às vezes, é útil projetar uma (MT) para uma determinada linguagem, se imaginarmos que o estado tem dois ou mais componentes, Um deles é o componente de controle, que normalmente funciona como um estado. Os outros componentes contêm dados que a (MT) precisa memorizar.

2. **Várias Trilhas**: Com frequência, também ajuda pensar nos símbolos de fita como vetores com um número fixo de componentes. Podemos visualizar cada componente como uma trilha separada da fita.
3. **Máquinas de Turing de Várias Fitas**: Um modelo de (MT) estendido tem um número fixo de fitas maior que um. Um movimento dessa (MT) se baseia no estado e no vetor de símbolos varridos pela cabeça em cada uma das fitas. Em um movimento, a (MT) de várias fitas muda de estado, sobregrava símbolos nas células varridas por cada uma das cabeças de fita, e move qualquer uma ou todas as cabeças de suas fitas uma célula em um dos sentidos ou estaciona. Embora seja capaz de reconhecer certas linguagens mais rápidas que a (MT) convencional, a (MT) de várias fitas não pode reconhecer nenhuma linguagem que não seja recursivamente enumerável (RE).
4. **Máquinas de Turing não-determinísticas**: Uma máquina de Turing não-determinística tem um número finito de escolhas parra o próximo movimento (estado, novo símbolo e cabeça) para cada estado e símbolo varrido. Ela aceita uma entrada se uma sequencia qualquer de escolhas a leve a uma Descrição Instantânea com um estado de aceitação. Embora aparentemente mais poderosa que a (MT) determinística, a (MT) não-determinística não é capaz de reconhecer nenhuma linnguagem que não seja (RE).

Restrições para Máquina de Turing Básica

1. **Máquinas de Turing de fita semi-infinita**: Podemos restringir uma (MT) a ter uma fita infinitta apenas à direita, ssem células à esqueda da posição inicial da babeça. Tal (MT) pode aceittar qualquerr linguagem (RE), mesmo também nçao podendo gravar brancos.
2. **Máquinas de Várias Pilhas**: Podemos restringir as fitas de uma (MT) de várias fitas a se ccompotarrem como uma pilha. A entrada fica em uma fita separada, que é lida uma vez da esquerrda para a direita, imitando o modo de entrada para um auttômato finito ou autômatto de pilha. Uma máquina de uma pilha é na realidade um autômato de pilha que aceita uma linguagem livre de contexto, enquanto umma máquina com dduas pilhas pode aceittar qualquer linguagem (RE).
3. **Máquinas de Contadores**: Podemos restringir ainda mais as pilha de uma máquina de várias pilhas, de forma que Lea só tenta um símbolo diferente de um marcador de fundo de pilha. Desse modo, cada pilha funciona como um contador, permitindo-nos armazenar um inteiro não negativo e testar se o inteiro armazenado é 0, mas além disso. Uma máquina com dois contadores é suficiente para aceitar qualquer linguagem (RE).

- **Simulação de uma Máquina de Turing por um computador real**: É possível, em principio, simular uma (MT) por um computador real, se aceitarmos que existe um suprimento potencialmente infinito de um dispositivo de armazenamento removível como um disco, a fim de simular a parte não branca da fita da (MT). Tendo em vista que os recursos físicos para prodduzir discos nçao são infinitos, esse argumetno é questionável. Porém, commo os liomites sobre a quantiddade de espaço de armazenamento existe no universo são desconhecidos e sem dúvida vastos, a hipótese de um recursso infinitto, como na fita da (MT), é reallista na prática e geralmente aceita.
- **Simulação de um computador por uma Máquina de Turing**: Uma (MT) pode simular o espaço de armazenamento e o controle de um computador real, usando uma única fita para armazenar todos os locais e seu conteúdo: registradores, memória principal, discos e outros dispositivos de armazenamento. Desse modo, podemos ter confiança em eu algo que não possa ser feito por uma (MT) não poderá ser realizado por um computador real.
- Uma Máquina de Turing Bidimensional tem o controle habitual de estados finitos, mas tem uma fita que é uma grade bidimensional de células, infinita em todas as direções. A entrada é colocada em uma linha de grade, com a cabeça na extremidade esquerda da entrada e o controle no estado inicial, como sempre. A aceitação é feita pela entrada em um estado final, também da maneira habitual. Assim, pode ser demonstrado o seguinte teorema.

"As linguagens aceitas por máquinas de Turing bidirecionais são as mesmas aceitas por (MT's) ordinárias"
(Hopcroft, pg: 370)

Linguagens Formais e Modelos Computacionais

Linguagem	Gramática	Dispositivo Computacional
Irrestrita	Irrestrita (Tipo 0)	Máquina de Turing
Sensível ao Contexto	Sensível ao Contexto (Tipo 1)	Autômato Limitado Linear
Livre de Contexto	Livre de Contexto (Tipo 2)	Autômato de Pilha
Regular	Regular (Tipo 4)	Autômato Finito

Tabela 2 – Linguagens Formais e Modelos Computacionais

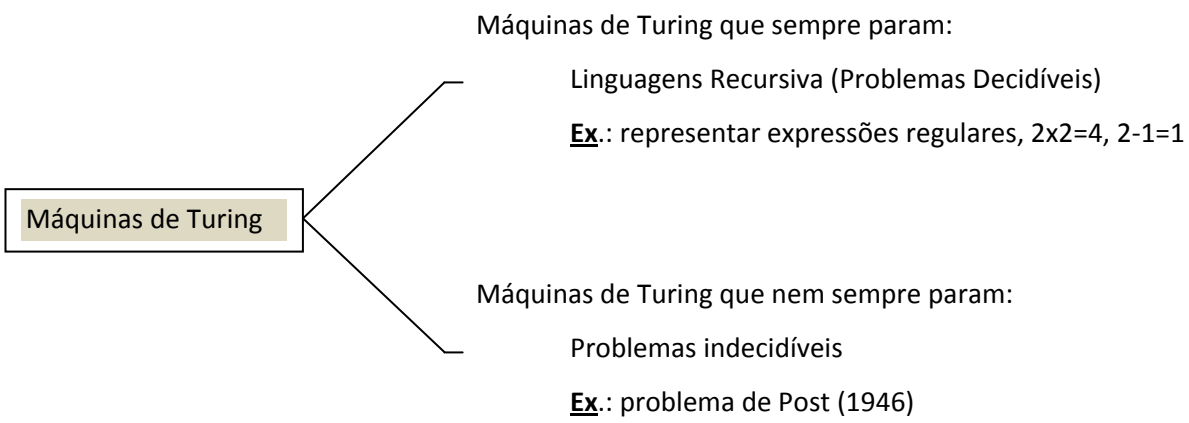
O que é um algoritmo?

1. Deve ser complete: Sempre produz uma resposta S ou N.
2. Deve ser mecânico: Sequência finita de instruções não-ambíguas.
3. Deve ser determinístico: Produz sempre a mesma resposta para cada entrada.

Tese de Church-Turing

"Qualquer problema de decisão é computável por um algoritmo se, e somente se, é computável por uma Máquina de Turing".

- **Problemas sobre especificações de Máquinas de Turing**: Por exemplo, os problemas a seguir indecidíveis:
  1. Se a linguagem aceita por uma (MT) é vazia;
  2. Se a linguagem aceita por uma (MT) é finita;
  3. Se a linguagem aceita por uma (MT) é uma linguagem regular;
  4. Se a linguagem aceita por uma (MT) é uma linguagem livre de contexto;



- Uma linguagem L é dita recursiva se for aceita por pelo menos uma (MT) que produz parada (com ou sem aceitação) para qualquer entrada.

Teorema 10:	A classe das linguagens recursivas é um subconjunto de classe de linguagens irrestritas.
Teorema 11:	O complemento de uma linguagem recursiva é uma linguagem recursiva.
Teorema 12:	A união de Linguagens recursivas é uma linguagem recursiva.
Teorema 13:	A união de Linguagens irrestritas é uma linguagem irrestrita.

Computabilidade

- Preocupação: Classificar problemas como computáveis ou não.

- Com a capacidade uma (MT) para resolver um problema (embora não muito eficiente) excede a dos computadores atuais, se encontrarmos problemas que as (MT's) não podem fazer, saberemos que os computadores também não poderão.
- Um problema é incomputável (ou insolúvel) quando não existe um algoritmo que possa executá-lo.
- Uma (MT)T que começa com uma cadeia α ou parra ou nunca para. Problema de decisão: "Existe um algoritmo que decide, dada qualquer (MT) e qualquer cadeia α, se a (MT) começando com α vai parar?". O interesse consiste num método geral que decida corretamente todos os casos.
- O problema da parada de uma (MT) é insolúvel.