CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro carlos@comp.ita.br
Ramal: 5895 Sala: 106

AULA 1

Orientações gerais (estrutura do curso)
Strings, alfabetos e linguagens
Grafos e árvores
Definições recursivas
Provas indutivas
Expressões regulares

Motivação

Objetivo do curso:
Apresentar os fundamentos da Teoria da Computação.

- Como definir o processo computacional de maneira formal?
- Quais as capacidades / limitações da computação?
- Como definir uma máquina abstrata que realiza computações?

Em suma: quais as capacidades e limitações de um computador?

Veremos que existem vários modelos, e que a capacidade computacional de uma máquina é caracterizada pela linguagem formal que ela pode reconhecer.

Quanto mais "poderosa" uma máquina, maior a complexidade da linguagem que ela pode reconhecer.

Bibliografia Básica

- **Introduction to Automata Theory, Languages and Computation** - J.E. Hopcroft e J.D. Ullman. Addison-Wesley. *Um clássico na área, razoavelmente completo e didático. Provas sumárias de teoremas. Apresenta soluções comentadas para alguns exercícios.*
- **Introdução à Teoria de Autômatos, Linguagens e Computação** – J.E. Hopcroft, J.D. Ullman e R. Motwami. Campus. *Tradução da 2a. edição americana de versão estendida do Hopcroft/Ullman. Vários erros de tradução. Bem completo.*
- **Elements of the Theory of Computation** - H.R. Lewis e C.H. Papadimitriou. Prentice-Hall. *Um pouco mais pesado e menos didático do que o anterior.Conciso e objetivo.*
- **Languages and Machines** - T. Sudkamp. Addison-Wesley. *Mais didático do que os anteriores. Ordenação não muito ortodoxa: primeiro discute linguagens e gramáticas e só então introduz a teoria de autômatos.*
- **Introduction to the Theory of Computation** – M. Sipser. PWS. *Conciso e didático.*
- **Notas de aula**

Orientações Gerais: Horários e Avaliação

Horários:

– 3 tempos semanais (teoria)

Avaliação:

- Dois exames escritos individuais em sala.
- Dois projetos (integrados) <u>relatados e apresentados</u>.
- Duas listas de exercícios realizadas individualmente

- **Nota (1° bimestre): N1 = (5*p1 + 4*e1 +l1)/10**
  p1 = nota do primeiro projeto
  e1 = nota do exame 1
  l1= nota da lista 1
- **Nota (2° bimestre): N2 = (5*p2 + 4*e2 +l2)/10**
  p2 = nota do segundo projeto
  e2 = nota do exame 2
  l2 = nota da lista 2
- **Nota final: (N1+N2)/2**

Critérios para avaliação dos projetos:

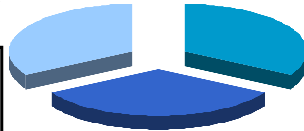
Qualidade do relatório (50%)
- **Adequação da metodologia**
- **Qualidade dos resultados**
- **Capacidade crítica**
- **Qualidade do texto**

Integração com demais projetos (30%)

Qualidade da apresentação (20%)
- **avaliação individual**

Cada projeto desenvolvido por grupo de <u>x alunos</u>

Teoria da Computação: Visão Geral



O QUE É UM COMPUTADOR?

Existem modelos gerais de computadores que não referenciam nenhuma implementação particular? Como verificar o que um computador faz?

Exemplos de modelos computacionais:

- Autômato finito: processamento de texto, compiladores e projeto de *hardware*.
- Gramática livre de contexto: linguagens de programação e Inteligência Artificial.

Há problemas que podem ser resolvidos por computadores, e outros que não podem.

Exemplo: Teorema de Gödel

O QUE TORNA ALGUNS PROBLEMAS COMPUTÁVEIS E OUTROS NÃO COMPUTÁVEIS? QUAIS AS CONSEQUÊNCIAS DISSO PARA O PROJETO DE COMPUTADORES?

Há problemas simples e problemas difíceis:

- Ordenamento: simples..
- Alocação de recursos: complexo.

O QUE TORNA ALGUNS PROBLEMAS FÁCEIS E OUTROS DIFÍCEIS? COMO CLASSIFICAR PROBLEMAS?

Kurt Gödel

- Nascido em 1906 (Brno, Rep. Tcheca), faleceu em 1978 (Princeton, EUA).
- Doutorado em 1929 (Univ. de Viena).
- Teorema provado em 1931: *qualquer sistema matemático baseado em axiomas tem proposições que não podem ser provadas*.
- Teorema de Gödel encerrou uma busca de mais de 100 anos por uma base axiomática completa para a Matemática (Hilbert, Russell e outros). É um dos dez mais importantes teoremas provados, considerando-se qualquer área da Ciência.

Mais info: http://www-groups.dcs.st-and.ac.uk/~history/Mathematicians/Godel.html

Relembrando Teoria dos Conjuntos

Conjuntos: letras maiúsculas, elementos: letras minúsculas
Def. de conjuntos: chaves+enumeração, chaves+condição.

Exemplos:

$A=\{1,2,3\}$, $B=\{a,b,c,...,z\}$, $C=\{x \in N \mid x<100$ e x é primo$\}$

Def.: Conjunto vazio ($\varnothing$): conjunto sem elementos.
Def.: $x \in X$ se x é um elemento do conjunto X.
Def.: $X=Y$ se X e Y têm os mesmos elementos.
Def.: $Y \subseteq X$ se todo elemento de Y é elemento de X.
Def.: $Y \subset X$ se todo elemento de Y é elemento de X e $Y \neq X$ (Y subconjunto **próprio** de X).

Da definição:

O conjunto vazio é subconjunto (próprio) de qualquer conjunto.

Qualquer conjunto X é subconjunto de si próprio.

Operações sobre Conjuntos

- $A \cup B = \{x \mid$ x está em A ou x está em B$\}$
- $A \cap B = \{x \mid$ x está em A e x está em B$\}$
- $A-B = \{x \mid$ x está em A e x não está em B$\}$
- $A \times B = \{(x,y) \mid$ x está em A e y está em B$\}$ (produto cartesiano de A e B)
- Uma relação binária de A e B é um subconjunto de $A \times B$
- 2^A, o conjunto das partes (*power set*) de A é o conjunto dos subconjuntos de A
  Exemplo: $A=\{1,2\} \Rightarrow 2^A = \{\varnothing, \{1\}, \{2\}, A\}$
- Dois conjuntos A e B têm a mesma cardinalidade se existir um mapeamento (função) de um-para-um dos elementos de A para os elementos de B.
- Um conjunto A tem cardinalidade menor do que um conjunto B se existir um mapeamento (função) de um-para-um dos elementos de A para apenas alguns dos elementos de B.
- A cardinalidade de um conjunto finito A é o número de elementos de A.
- Se A e B são finitos e $A \subset B$ então $card(A) < card(B)$.
- Se A e B são infinitos e $A \subset B$ então $card(A) < card(B)$ **ou** $card(A)=card(B)$.
- Um conjunto A é **contável** se $card(A) = card(Z)$.

Propriedades das Relações e Classes de Equivalência

- reflexiva se aRa para todo $a \in S$;
- transitiva se aRb e bRc implica aRc para todo $a,b,c \in S$;
- simétrica se aRb implica bRa para todo $a,b \in S$.

Um relação que satisfaz as três propriedades acima é dita **relação de equivalência**.

Seja $\equiv$ uma relação de equivalência em um conjunto X. A **classe de equivalência** de um elemento $a \in X$ (definida pela relação $\equiv$) é o conjunto $[a] = \{b \in X \mid a \equiv b\}$.

Ou seja: a classe de equivalência do elemento *a* é o subconjunto formado por aqueles elementos de X que se relacionam com *a* pela relação de equivalência em questão.

Exemplos

- Seja X um conjunto qualquer, e R a igualdade. R é relação de equivalência, pois:

  a) É reflexiva (a = a).

  b) É transitiva (se a=b e b=c, então a=c).

  c) É simétrica (se a=b, então b=a).

  Se X = N, qual é a classe de equivalência do elemento 2?

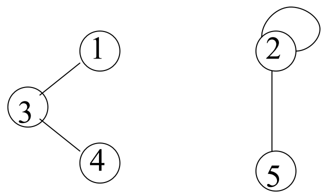
- Seja X o conjunto dos inteiros. Definamos uma relação R_n tal que xR_ny sss (x - y) é divisível por n.

  a) Mostre que R_n é uma relação de equivalência.

  b) Qual é classe de equivalência sobre R_2 do elemento 10?

Grafos

Um grafo G=(V,E) é um conjunto finito de vértices (nós) V e um conjunto de pares (ligações) de vértices E.



V={1,2,3,4,5}

E={(n,m)| n+m=4 ou n+m=7}

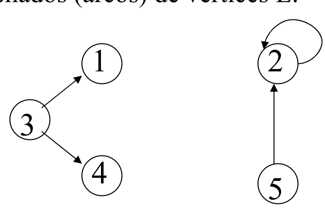
Um **caminho** (*path*) em um grafo G=(V,E) é uma sequência de vértices $v_1,v_2,\ldots,v_k$, $k \geq 1$, tais que existem ligações (v_i, v_{i+1}), $1 \leq i \leq k$.

O **comprimento do caminho** é k-1. Se $v_1 = v_k$, o caminho é um **ciclo**.

O **grau** de um nó é o número de ligações àquele nó.

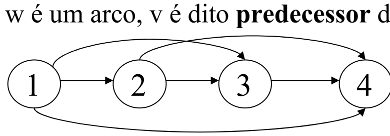
Grafos Direcionados

Um grafo direcionado G=(V,E) é um conjunto finito de vértices (nós) V e um conjunto de pares ordenados (arcos) de vértices E.



Um **caminho** (*path*) em um grafo direcionado G=(V,E) é uma sequência de vértices $v_1,v_2,\ldots,v_k$, $k \geq 1$, tais que existem arcos (v_i, v_{i+1}), $1 \leq i \leq k$.

Se $v \rightarrow w$ é um arco, v é dito **predecessor** de w e w é dito **sucessor** de v.



$\{\{1,2,3,4\},\{i \rightarrow j \mid i<j\}\}$

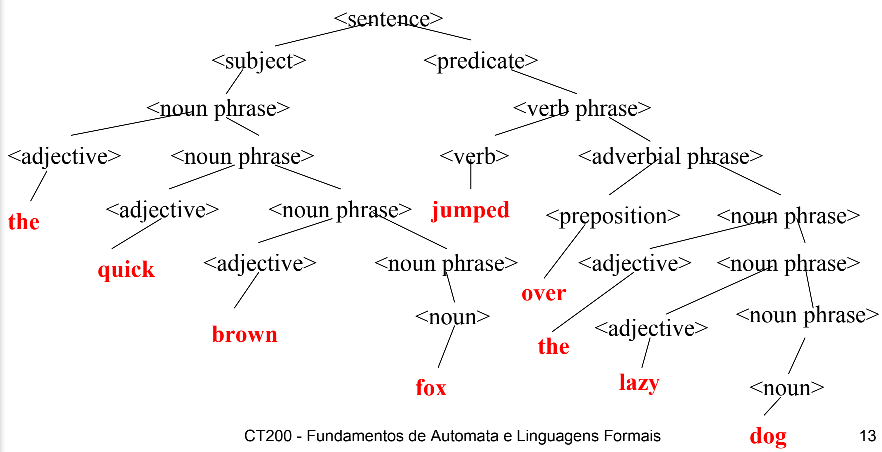
Árvores

Uma árvore é um grafo direcionado com as seguintes propriedades:

• Um nó *raiz*, sem predecessores, a partir do qual há um caminho para todo nó.

• Todos os outros nós têm exatamente um predecessor.

•Os sucessores de cada nó são ordenados a partir da esquerda.



Provas

Três mecanismos básicos:

1. Prova por construção

   Para provas envolvendo teoremas que afirmam a existência de um objeto. Consiste em produzir uma demonstração de como construir o objeto.

2. Prova por contradição (ou por absurdo)

   Consiste em assumir que o teorema é falso e mostrar que isto leva a uma contradição.

3. Prova por indução

   Utiliza o princípio da indução matemática: uma propriedade P(n) é resultado de P(0) e da implicação P(n-1) → P(n).

Exemplo de prova por construção

Teorema: Para qualquer número par *n* > 2 existe um grafo formado apenas por *n* vértices de grau 3.

Prova: *Construa um grafo G=(V,E) da seguinte maneira:*

$V = \{0,1,\ldots,n\text{-}1\}$

$E = \{\{i, i+1\} \text{ para } 0<i \leq n\text{-}1\} \cup \{\{n\text{-}1,0\}\} \cup \{\{i, i+n/2\} \text{ para } 0\leq i \leq n/2\text{-}1\}$

Desenhe o grafo e verifique.

Exemplo de prova por contradição

Teorema: √2 é irracional

Prova: *Suponha* **√2** *racional. Então* **√2** *= m/n, onde m e n são inteiros. Divida agora m e n pelo MDC(m,n), de modo que a fração não tem seu valor alterado, mas agora* <u>*m e n não podem mais ser ambos pares*</u>.

Multiplicando os dois lados por n e elevando ao quadrado: $2n^2 = m^2$. *Portanto,* m^2 *é par. Logo, m é par, e então m=2k. Substituindo:*

$2n^2=(2k)^2=4k^2 \Rightarrow n^2 = 2k^2 \Rightarrow n$ *é par (contradição)*

Prova por Indução: Exemplo 1

Prove que: $$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

i) Base para n=0: $$\sum_{i=0}^{0} i^2 = 0, \quad \frac{0(0+1)(2.0+1)}{6} = 0$$

ii) Passo indutivo: $$\sum_{i=0}^{n-1} i^2 = \frac{(n-1)(n)(2n-1)}{6}$$

$$\sum_{i=0}^{n} i^2 = \sum_{i=0}^{n-1} i^2 + n^2 = \frac{(n-1)(n)(2n-1)}{6} + n^2$$

$$= \frac{n(n+1)(2n+1)}{6}$$

QED

Provas por Indução: Exemplo 2

Prove que *3* é um fator de n^3-n+3

i) Base para $n=0$: $0^3-0+3 = 0+3=1.3$

ii) Passo indutivo: $n^3-n+3=k.3$

$$
\begin{aligned}
(n+1)^3-(n+1)+3 &= n^3+3n^2+2n+3 \\
&= n^3-n+3+(n^2+n)3 \\
&= k.3+(n^2+n)3 \\
&= k'.3
\end{aligned}
$$

QED

Linguagens: Conceitos Básicos

- Um **símbolo** é um símbolo...
- Um **alfabeto** Σ é um conjunto finito de símbolos.
- Uma **cadeia** ***(string)*** é uma sequência finita de símbolos.
  - Sequência de **zero** símbolos: ε (cadeia vazia)
  - Conjunto de **todas** as cadeias de um alfabeto Σ: Σ^*
  - Operação de **concatenação** de duas cadeias *w* e *v*: *wv*
- Uma **linguagem** é um subconjunto de Σ^*.
- A **sintaxe** da linguagem é o conjunto de propriedades satisfeitas por suas cadeias.

Exemplos:

- O conjunto vazio $\varnothing$ é uma linguagem. O conjunto formado pela cadeia vazia $\{\varepsilon\}$ é uma linguagem.
- O conjunto de palíndromos sobre $\{0,1\}$.
  - Símbolos: 0 1, Alfabeto: $\{0,1\}$
  - Linguagem: $\{\varepsilon, 0, 1, 00, 11, 010, 1101011, ...\}$
  - $u = 00$, $v = 010 \rightarrow uv = 00010$
- Σ^* é uma linguagem sobre um alfabeto Σ.
- $\Sigma=\{a\}$. $\Sigma^*=?$
- $\Sigma=\{0,1\}$. $\Sigma^*=?$
- Exercício: Uma linguagem natural encaixa-se na definição formal de linguagem? Em caso positivo, que símbolos seriam mais adequados: letras ou palavras?

Representações Finitas de Linguagens

Vantagens:
- evitar enumeração exaustiva
- explicitar a sintaxe da linguagem de modo não-ambíguo

Técnicas:
- Representação recursiva
  1. Defino um conjunto de elementos básicos da linguagem.
  2. Defino um passo recursivo que permite obter qualquer string que obedeça à sintaxe da linguagem.
  3. Imponho o fechamento: passos 1 e 2 definem todos os elementos da linguagem, e somente estes.
- Representação com uso de operadores
  - Defino operadores sobre linguagens
  - Represento a linguagem através de um conjunto de operações sobre linguagens mais simples.

Representação Recursiva: Exemplos

L = cadeias sobre {a,b} começando com a e com comprimento par.

• aa, ab $\in$ L (elementos básicos)

• $u \in$ L $\Rightarrow$ $u$aa, $u$ab, $u$ba, $u$bb $\in$ L (passo recursivo)

• $u \in$ L apenas se obtido dos elementos básicos por um número finito de aplicações do passo recursivo (fechamento)

L = cadeias sobre {a,b} em que cada b é imediatamente precedido por um a.

• $\varepsilon \in$ L (elemento básico)

• $u \in$ L $\Rightarrow$ ua, $u$ab $\in$ L (passo recursivo)

• $u \in$ L apenas se obtido dos elementos básicos por um número finito de aplicações do passo recursivo (fechamento)

Alguns Operadores sobre Linguagens

Concatenação de linguagens X e Y: $XY = \{uv \mid u \in X \textbf{ e } v \in Y\}$

Exemplo: X = {a,b,c}, Y = {abb,ba}

XY = {aabb, aba, babb, bba, cabb, cba}

$X^0 = \{\varepsilon\}$

$X^1 = X = \{a,b,c\}$

$X^2 = XX = \{aa, ab, ac, ba, bb, bc , ca, cb, cc\}$

Fechamento de Kleene da linguagem X: $X^* = \bigcup_{i=0}^{\infty} X^i$

União de linguagens X e Y: $X \cup Y$

Exemplo: X = {a,b,c}, Y = {abb,ba}

$X \cup Y = \{a, b, c, abb, ba\}$

$X^+ = \bigcup_{i=1}^{\infty} X^i$ (Note que $X^+ = XX^*$)

Reversão de linguagem X: $X^R = \{u^R \mid u \in X\}$

Representação com Operadores: Exemplos

Exemplo 1: L = cadeias sobre {a,b} com comprimento par.

$\{aa, bb, ab, ba\}^*$

Exemplo 2: L = cadeias que começam e terminam com a e que contém pelo menos um b.

$\{a\}\{a, b\}^*\{b\}\{a, b\}^*\{a\}$

Exemplo 3: L = cadeias sobre {a,b} começando com aa ou terminando com bb.

$\{aa\}\{a, b\}^* \cup \{a, b\}^*\{bb\}$

Linguagens Regulares: Definição

- $\varnothing$, $\{\varepsilon\}$ e $\{a\}$, para todo $a \in \Sigma$, são linguagens regulares.
- **Passo recursivo:** X e Y linguagens regulares $\Rightarrow$ $X \cup Y$, XY e X^* são linguagens regulares.
- **Fechamento:** X é linguagem regular apenas se obtido dos elementos básicos por um número finito de aplicações do passo recursivo.

Uma notação simplificada: **expressões regulares**

$\{a\} \rightarrow$ **a**, $\{a, b\} \rightarrow$ **a$\cup$b**, precedência * > concatenação > $\cup$

Exemplo: O conjunto de cadeias contendo a sub-cadeia bb.

$\{a\}, \{b\} \rightarrow \{a, b\}^*$	união, fechamento de Kleene
$\{b\}\{b\} \rightarrow \{bb\}$	concatenação
$\{a, b\}^*\{bb\}\{a, b\}^*$	concatenação (duas vezes)
$\mathbf{(a \cup b)^*\ bb\ (a \cup b)^*}$	**expressão regular correspondente**

Exemplo: O conjunto de cadeias contendo dois ou mais b's.

$\mathbf{a^*\ ba^*\ b\ (a \cup b)^*}$

$\mathbf{(a \cup b)^*\ ba^*\ ba^*}$

$\mathbf{(a \cup b)^*\ b(a \cup b)^*\ b(a \cup b)^*}$ expressões **equivalentes**

Algumas identidades envolvendo expressões regulares:

$\varnothing\mathbf{a} = \mathbf{a}\varnothing = \varnothing$	$\varepsilon\mathbf{a} = \mathbf{a}\varepsilon = \mathbf{a}$
$\varnothing^* = \varepsilon$	$\varepsilon^* = \varepsilon$
$\mathbf{a} \cup \varnothing = \mathbf{a}$	$\mathbf{a^* = (a^*)^*}$
$\mathbf{a(b \cup c) = ab \cup ac}$	$\mathbf{(b \cup c)a = ba \cup ca}$
$\mathbf{(ab)^*a = a(ba)^*}$	$\mathbf{(a \cup b)^* = (a^* \cup b)^* = a^*(a \cup b)^* = (a \cup ba^*)^* = (a^*b^*)^* = a^*(ba^*)^*}$

CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro carlos@comp.ita.br
Ramal: 5895 Sala: 106

AULA 2

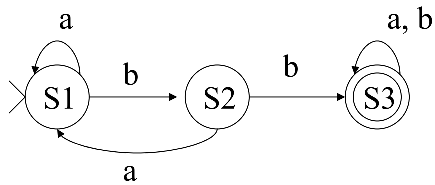
Automata finitos
Automata finitos não-determinísticos
Automata finitos com movimentos-ε
Teorema de Kleene

Autômato Finito

- Modelo de sistema dinâmico em que:
  - Existe um número finito de **estados** do sistema.
  - Existem transições entre estados, mediadas por entradas provenientes de um conjunto discreto finito (alfabeto Σ).
  - Para cada estado, todas as transições produzidas por cada um dos símbolos do alfabeto estão definidas.

Estados: S1 (inicial), S2, S3 (final)

Entradas: {a,b}



Representação por grafo direcionado (**diagrama de transição**)

Automata Finitos: Definição Formal

Um autômato finito é uma **quíntupla**

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

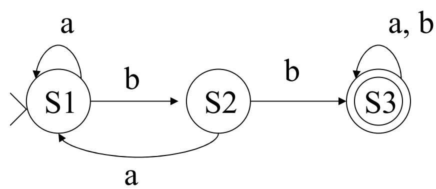
- Q = conjunto finito de estados
- Σ = alfabeto
- δ = função de transição $Q \times \Sigma \rightarrow Q$
- q_0 = estado inicial
- $F \subseteq Q$ = conjunto de estados finais (aceitadores)

Automata Finitos: Exemplos

1. $Q = \{S1, S2, S3\}$, $\Sigma = \{a,b\}$, $q_0 = S1$, $F = \{S3\}$

δ:

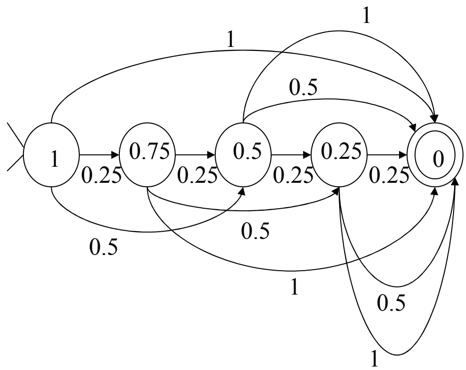
	a	b
S1	S1	S2
S2	S1	S3
S3	S3	S3



2. Máquina para vender bilhetes de metrô (R$ 1,00 cada). Não dá troco.

Q = {S1=1, S2=0.75, S3=0.5,S4=0.25,S5=0}, cada estado identificando o quanto falta para que se libere um bilhete de metrô.

Σ = {1, 0.5, 0.25, 0}, δ = ... , q_0 = S1 = 1.0, F = {S5=0}

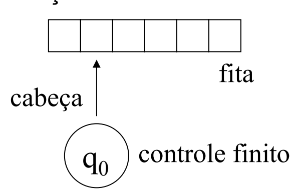


Observe que o estado codifica o que é relevante (memória implícita).

Automata Finitos e Máquinas de Estados

Um automato finito é uma **máquina de estados**

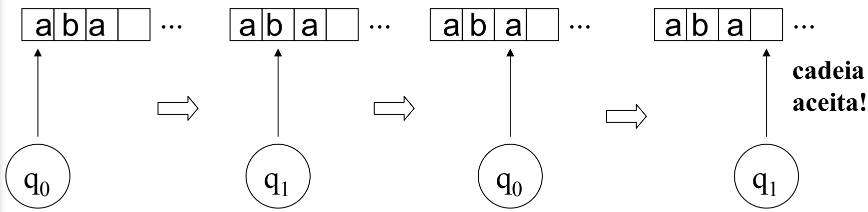
- um registrador de estado interno (controle finito) + uma fita segmentada + cabeça de leitura



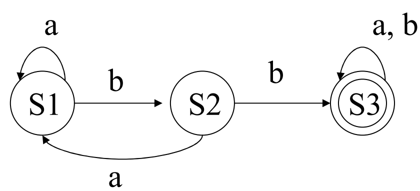
- fita: armazena uma cadeia de Σ (1 símbolo/segmento).
- cabeça: lê segmento da fita (um símbolo da cadeia).
- registrador: altera estado de acordo com δ e move a fita um segmento para a esquerda (computação).
- uma computação **termina** quando a cadeia "acaba".
- Cadeia é **aceita** pelo AF se a computação termina em um estado $q \in F$.
- Cadeia é **rejeitada** pelo AF se a computação termina em um estado $q \notin F$.

Exemplos

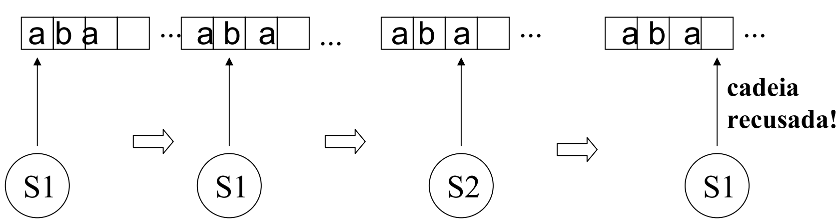
1. M: $Q = \{q_o, q_1\}$ $\delta(q_0, a) = q_1$
$\Sigma = \{a, b\}$ $\delta(q_0, b) = q_0$
$F = \{q_1\}$ $\delta(q_1, a) = q_1$
$\delta(q_1, b) = q_0$



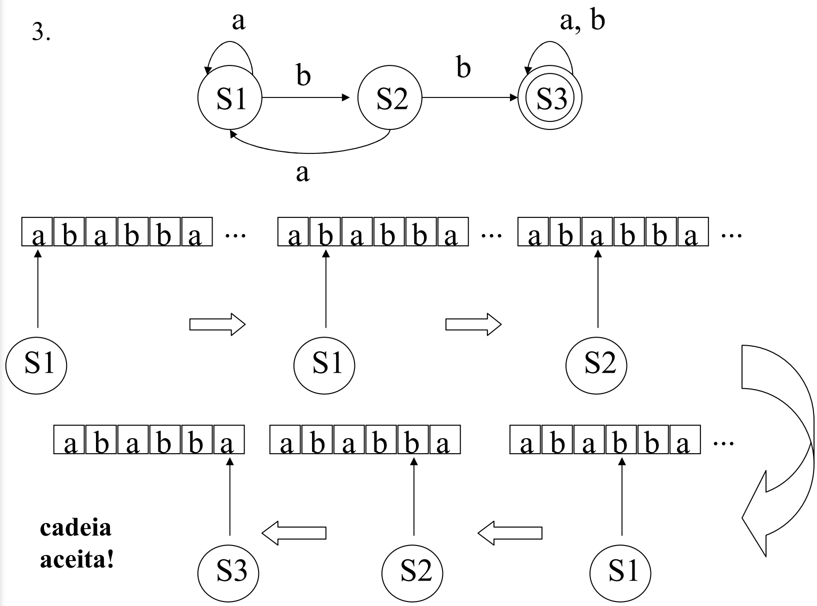
2.



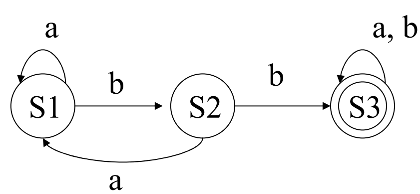
M: $Q = \{S1,S2,S3\}$ $\delta(S1, a) = S1$
$\Sigma = \{a, b\}$ $\delta(S1, b) = S2$
$F = \{S3\}$ $\delta(S2, a) = S1$
$q_o=S1$ $\delta(S2, b) = S3$



3.



4.



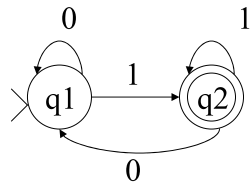
Este automato aceita qualquer cadeia que contenha dois b´s seguidos...

Diz-se que este automato finito **reconhece** a linguagem definida por cadeias com dois b´s seguidos.

5. M_5: $Q = \{q1, q2\}$, $\Sigma = \{0,1\}$, δ , $q_0 = q1$, $F = \{q2\}$

δ:

	0	1
q1	q1	q2
q2	q1	q2

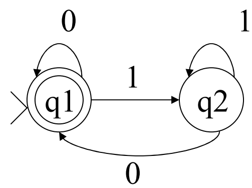


$L(M_5) = ?$

6. M_6 : $Q = \{q1, q2\}$, $\Sigma = \{0,1\}$, δ, $q_0 = q1$, $F = \{q1\}$

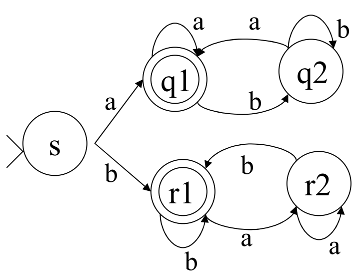
δ:

	0	1
q1	q1	q2
q2	q1	q2



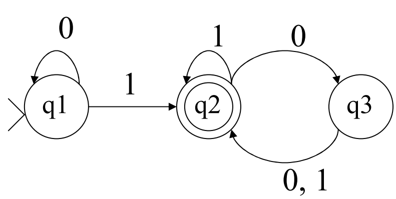
$L(M_6) = ?$

7. M_7:



$L(M_7) = ?$

8. M_8 :



$L(M_8) = ?$

AFs: Mais Definições

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

- A **configuração instantânea** $[q_i, \boldsymbol{w}]$ de um AF corresponde ao estado q_i e cadeia ***w*** ainda não processados num dado instante (um elemento de $Q \times \Sigma^*$). A configuração instantânea define o comportamento futuro do AF.
- A função $\vdash$ em $Q \times \Sigma^+$ é definida por $[q_i, a\boldsymbol{w}] \vdash [\delta(q_i, a), \boldsymbol{w}]$
- A notação $[q_i, \boldsymbol{u}] \overset{*}{\vdash} [q_j, \boldsymbol{v}]$ é usada para indicar que configuração $[q_j, v]$ pode ser obtida de $[q_i, u]$ por zero ou mais transições.
- A **função de transição estendida** $\hat{\delta}$ de $Q \times \Sigma^*$ em Q descreve formalmente a operação de um AF sobre uma cadeia ***w***, e é definida por: $\hat{\delta}(q, \varepsilon) = q, \hat{\delta}(q, wv) = \delta(\hat{\delta}(q, w), v)$
- Uma cadeia *x* é aceita (por um autômato *M)* se $\delta(q_0, x) = p \in F$.
- A **linguagem L(*M*) aceita por *M*** é o conjunto de cadeias em Σ^* aceitas por *M.*
- Dois AFs que aceitam a mesma linguagem são ditos **equivalentes**.

Exemplo

M:Q = $\{q_o, q_1, q_2\}$

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

$F = \{q_2\}$

δ	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

$[q_o, \text{abba}]$ $[q_o, \text{bba}]$ $[q_1, \text{ba}]$ $[q_2, \text{a}]$ $[q_2, \varepsilon]$

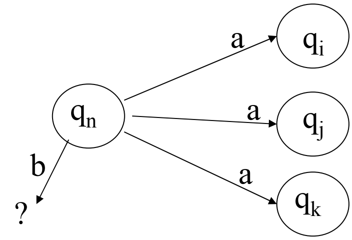
$[q_o, \text{abab}]$ $[q_o, \text{bab}]$ $[q_1, \text{ab}]$ $[q_0, \text{b}]$ $[q_1, \varepsilon]$

$\hat{\delta}(q_0, \text{abba}) = q_2$ **cadeia aceita!**

$\hat{\delta}(q_0, \text{abab}) \neq q_2$ **cadeia rejeitada!**

AFs Não-Determinísticos

- **Zero, um ou mais** próximos estados podem resultar de computação.



$\delta(q_n, a) = \{q_i, q_j, q_k\}$
$\delta(q_n, b) = \emptyset$

- Um dado AFN **equivale** a algum AF (embora mais "complicado").
- Vantagens:
  - simplifica prova de teoremas relacionados com AFs.
  - facilita projeto de autômatos reconhecedores de linguagens.

Em particular, AFNs serão usados para provar que:

Uma linguagem L é aceita por um AF $\Leftrightarrow$ L é uma linguagem regular

AFs Não-Determinísticos: Definição

Um AFND é uma **quíntupla** $M = (Q, \Sigma, \delta, q_0, F)$

- Q = conjunto finito de estados
- Σ = alfabeto
- δ = relação de transição $\subset Q \times \Sigma \times Q$
- q_0 = estado inicial
- $F \subseteq Q$ = conjunto de estados finais

M: $Q = \{q_o, q_1, q_2\}$
$\Sigma = \{a, b\}$
$F = \{q_2\}$

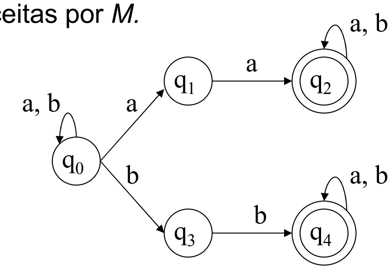
δ	a	b
q_0	$\{q_0\}$	$\{q_0,q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

$[q_0, ababb]$ $[q_0, babb]$ $[q_0, abb]$ $[q_0, bb]$ $[q_0, b]$ $[q_0, \varepsilon]$
$[q_0, ababb]$ $[q_0, babb]$ $[q_1, abb]$
$[q_0, ababb]$ $[q_0, babb]$ $[q_0, abb]$ $[q_0, bb]$ $[q_1, b]$ $[q_2, \varepsilon]$

AFNDs: Outras Definições

Seja $M = (Q, \Sigma, \delta, q_0, F)$ um AFND.

- Uma cadeia *w* é aceita por *M* se existir **ao menos uma** computação que processa a cadeia e termina em estado de F.
  No exemplo anterior: ababb é aceita por *M*.
- A **linguagem de *M***, denotada L(*M*), é o conjunto de cadeias em Σ^* aceitas por *M*.



aceita cadeias contendo subcadeias <u>aa</u> ou <u>bb</u>

Equivalência entre AFs e AFNDs

Teorema: Seja L uma linguagem aceita por um AFND. Então existe um AF que aceita L.

Prova:

Seja *M* = (Q, Σ, δ, q_0, F) um AFND.

Idéia: Construir um AF M' = (Q', Σ, δ', q_0', F') que simula computações realizadas por M. O AF manterá como "estado" o conjunto de todos os estados que podem resultar após uma dada computação de M.

Definamos:

a) Q' = Conjunto de todos os subconjuntos de Q ($Q' = 2^Q$). Note portanto que os estados de M' são então **conjuntos** de estados de M.

Notação para um elemento de Q': $[q_1, q_2, \ldots q_i]$, onde $q_1, q_2, \ldots q_i \in Q$.

b) $q_0' = [q_0]$

c) F' = estados de Q' contendo um estado de F.

d) $\delta'([q_1, q_2, \ldots q_i], a) = [p_1, p_2, \ldots p_j] \Leftrightarrow \delta(\{q_1, \ldots, q_i\}, a) = \{p_1, \ldots p_j\}$,

onde $\delta(\{q_1, \ldots, q_i\}, a)$ representa a aplicação de δ a cada q_k seguida da união dos conjuntos resultantes. Uma transição em M' produz portanto um conjunto que contém todos os possíveis sucessores para cada possível estado de M.

Provemos agora (por indução) que uma computação de uma *string* em M' "imita" o comportamento de M, isto é:

$\delta'(q_0', v) = [p_1, \ldots p_j] \Leftrightarrow \delta(q_0, v) = \{p_1, \ldots p_j\}$

i) para $|v| = 0$: $v = \varepsilon$ e $\delta'(q_0', \varepsilon) = q_0' = [q_0]$

ii) suponha válido para $|v| \leq m$: $\delta'(q_0', v) = [p_1, \ldots p_j] \Leftrightarrow \delta(q_0, v) = \{p_1, \ldots p_j\}$

iii) prova para $|u| = m+1$: $\delta'(q_0', u) = \delta'(q_0', va) = \delta'(\delta'(q_0', v), a)$

Mas $\delta'(q_0', v) = [p_1, \ldots, p_j] \Leftrightarrow \delta(q_0, v) = \{p_1, \ldots p_j\}$ (por hipótese)

e $\delta'([p_1, \ldots p_j], a) = [r_1, r_2, \ldots r_k] \Leftrightarrow \delta(\{p_1, \ldots, p_j\}, a) = \{r_1, \ldots r_k\}$ (definição)

$\therefore \delta'(\delta'(q_0', v), a) = [r_1, r_2, \ldots r_k] \Leftrightarrow \delta(\delta(q_0, v), a) = \{r_1, \ldots r_k\}$

$\therefore \delta'(q_0', va) = [r_1, r_2, \ldots r_k] \Leftrightarrow \delta(q_0, va) = \{r_1, \ldots r_k\}$

Finalmente, observe que $\delta'(q_0', v) = [p_1, \ldots p_j] \Leftrightarrow \delta(q_0, v) = \{p_1, \ldots p_j\}$

implica $\delta'(q_0', v) = [p_1, \ldots p_j] \in F' \Leftrightarrow \delta(q_0, v) = \{p_1, \ldots p_j\}$ contém um estado de F.

Portanto, L(M) = L(M').

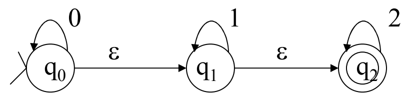
A volta é trivial.

AFNDs com Transições ε

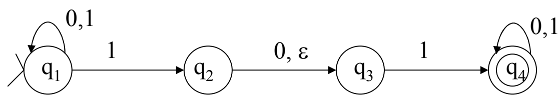
Um AFND com transições ε é um AFND em que
δ = relação de transição $\subset Q \times (\Sigma \cup \varepsilon) \times Q$

Ou seja: um AFND-ε aceita transições "não-forçadas".

Exemplo 1. Um AFND- ε que aceita um número de zeros (inclusive 0), seguidos por 1´s e 2´s.



Exemplo 2. Q={q1,q2,q3,q4}, Σ={0,1}, q_0 = q1, F={q4}.



$\delta = ?$

Fechamento- ε (ε-*CLOSURE*)

Def.: O **fechamento-ε** (ε-*CLOSURE*) de um estado q é o conjunto de todos os estados p tais que existe um caminho de q a p apenas com rótulos ε.

No exemplo 1:

No exemplo 2:

ε-*CLOSURE* $(q_0) = \{q_0, q_1, q_2\}$

ε-*CLOSURE* $(q_0) = \{q_1, q_2\}$

ε-*CLOSURE* $(q_0) = \{q_2\}$

- Para AFs, tínhamos: $\hat{\delta}(\mathrm{q}, \varepsilon) = \mathrm{q}$ $\quad \hat{\delta}(\mathrm{q}, wv) = \delta(\hat{\delta}(\mathrm{q}, w), v)$
- Para AFNDs com transições ε:

$$\hat{\delta}(\mathrm{q}, \varepsilon) = \varepsilon - CLOSURE(\mathrm{q}) \qquad \hat{\delta}(\mathrm{q}, wa) = \varepsilon - CLOSURE(\mathrm{P})$$

onde P={p | para algum r em $\hat{\delta}(q, w)$, p está em $\delta(r, a)$}

Equivalência entre AFNDs e AFNDs com Transições ε

Teorema: Uma linguagem L é aceita por um AFN-ε se e somente se L é aceita por um AFN

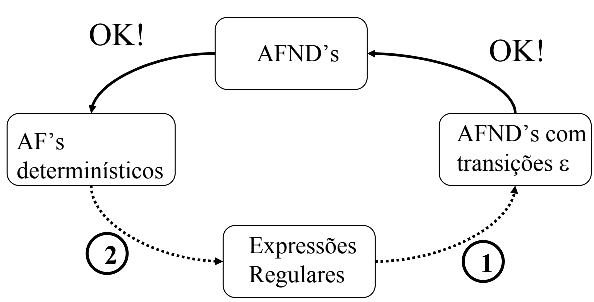
Prova: Hopcroft/Ullman, pág. 26.

Observe que a volta também vale, trivialmente.

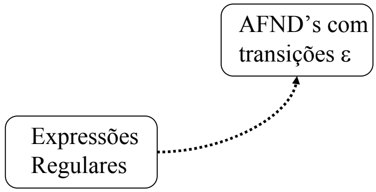
Relação entre AFs e Linguagens Regulares

Vamos agora provar que as linguagens aceitas pelos automata finitos são aquelas definidas pelas expressões regulares (ou seja, as linguagens regulares).

Idéia:



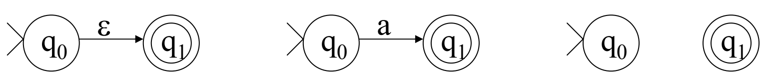
1



Teorema: Seja r uma expressão regular. Então existe um AFND-ε que aceita L(r).

Prova: Usaremos PIF sobre o número de operadores na expressão r.

1. Base (nenhum operador). Neste caso, r é ε (cadeia vazia), um símbolo a $\in\Sigma$ ou $\varnothing$ (nenhuma cadeia aceita). Os seguintes AFND-ε servem, respectivamente:

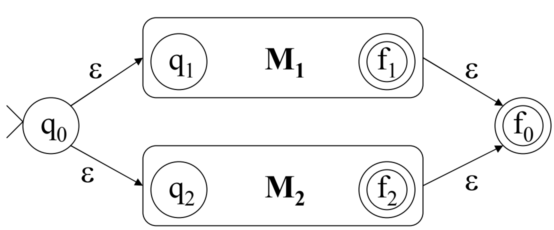


2. Indução. Suponha que o teorema seja válido para expressões regulares r com i-1 operadores.

3. Provemos a validade para i operadores. Por definição, expressões regulares são formadas via $\cup$, concatenação ou estrela de Kleene. Temos portanto três casos a analisar:

3.1 $r = r_1 \cup r_2$, onde r_1 e r_2 tem até i-1 operadores.

Neste caso, serve o seguinte AFND-ε:



Formalmente definido por:

$$M = \left(Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta, q_0, \{f_0\}\right)$$

onde

$$M_1 = \left(Q_1, \Sigma_1, \delta_1, q_1, \{f_1\}\right)$$

$$M_2 = \left(Q_2, \Sigma_2, \delta_2, q_2, \{f_2\}\right)$$

e

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

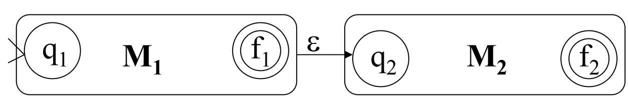
$$\delta(q, a) = \delta_1(q, a) \,\text{se}\, q \in Q_1 - \{f_1\}, a \in \Sigma_1 \cup \{\varepsilon\}$$

$$\delta(q, a) = \delta_2(q, a) \,\text{se}\, q \in Q_2 - \{f_2\}, a \in \Sigma_2 \cup \{\varepsilon\}$$

$$\delta(f_1, \varepsilon) = \delta(f_2, \varepsilon) = \{f_0\}$$

3.2 r = r_1 r_2 , onde r_1 e r_2 tem até i-1 operadores.

Neste caso, serve o seguinte AFND-ε:



Formalmente definido por:

$$M = \left(Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta, q_1, \{f_2\}\right)$$

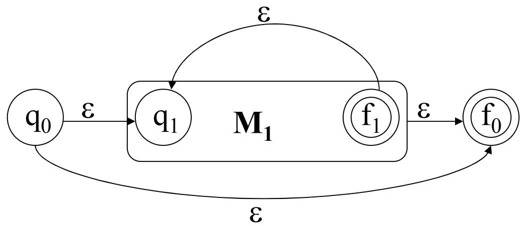
onde

$$\delta(q, a) = \delta_1(q, a) \,\text{se}\, q \in Q_1 - \{f_1\}, a \in \Sigma_1 \cup \{\varepsilon\}$$

$$\delta(q, a) = \delta_2(q, a) \,\text{se}\, q \in Q_2 - \{f_2\}, a \in \Sigma_2 \cup \{\varepsilon\}$$

$$\delta(f_1, \varepsilon) = \{q_2\}$$

3.3 $r = r_1^*$, onde r_1 tem até i-1 operadores.



$$M = \left(Q_1 \cup \{q_0, f_0\}, \Sigma_1, \delta, q_0, \{f_0\}\right)$$

$$\delta(q,a) = \delta_1(q,a) \text{ se } q \in Q_1 - \{f_1\}, a \in \Sigma_1 \cup \{\varepsilon\}$$
$$\delta(q_0\ ,\varepsilon) = \delta(f_1,\varepsilon) = \{q_1, f_0\}$$

Exercícios

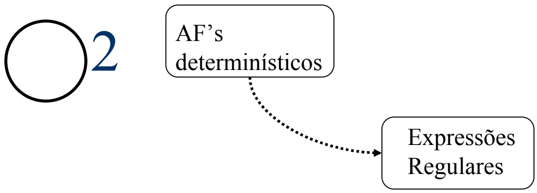
1. Construir AFND-ε's para as seguintes linguagens regulares:

1.1. $\mathbf{01^* \cup 1}$

1.2. $\mathbf{10+(0+11)0^*1}$

2. Construir um AF equiv. ao AFND ({p,q,r,s},{0,1},δ,p,{s}), sendo a função de transição

	0	1
p	p,q	p
q	r	r
r	s	-
s	s	s



Teorema: Seja L uma linguagem aceita por um AF. Então L é uma linguagem regular.

Prova (informal): Um mecanismo que gera uma linguagem regular a partir de qualquer AF. A idéia é "reduzir" passo a passo o autômato original, mostrando que o que ele produz ao final é uma expressão regular.

Existe uma prova formal em Hopcroft/Ullman (pág. 33).

Um algoritmo para gerar expressões regulares a partir de um AF

Seja M um automato com $Q=\{q_1,q_2,...,q_n\}$, e seja m o número de estados finais de M (ou seja, card(F)=m). Seja w_{ij} o rótulo (símbolo do alfabeto) associado à transição do estado q_i ao estado q_j.

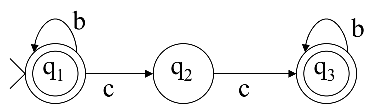
Inicialmente, faça m cópias $M_1,M_2,...,M_m$ de M, cada uma com um estado final diferente.

- Para cada M_t faça
  - Escolha um q_i em M_t que não seja inicial ou final em M_t ;
  - Para todo j,k $\neq$ i (mas possivelmente j=k), faça:
    - Se $w_{ji} \neq \emptyset$, $w_{i\,k} \neq \emptyset$, $w_{ii} = \emptyset$ então adicione um arco de q_j a $q_{\,k}$ com rótulo $w_{ji}w_{jk}$.
    - Se $w_{ji} \neq \emptyset$, $w_{i\,k} \neq \emptyset$, $w_{ii} \neq \emptyset$ então adicione um arco de q_j a $q_{\,k}$ com rótulo $w_{ji}(w_{ii})^* w_{j\,k}$.
    - Se j e k têm arcos w_1, w_2, ..., w_s conectando-os, então substitua todos os arcos por um único $w_1 \cup w_2 \cup \ldots \cup w_s$.
    - Remova o nó q_i e todos os seus arcos incidentes em M_t.

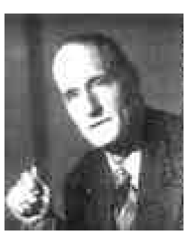
  Até que os únicos nós em M_t sejam o estado inicial e o (único) estado final. Determine então a expressão L_t aceita por M_t

A expressão regular aceita por M é formada pela união: $L_1 \cup L_1 \cup \ldots \cup L_m$

Exemplo:



Terminamos então com um Teorema fundamental da Teoria da Computação:



Stephen Kleene
(1909-1994)

Teorema de Kleene:

Uma linguagem L é aceita por um AF com alfabeto Σ sss L é uma linguagem regular sobre Σ

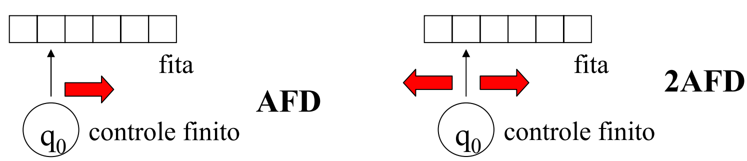
CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro carlos@comp.ita.br
Ramal: 5895 Sala: 106

AULA 3

Automata finitos bidirecionais
Automata finitos com saída
Propr. de fechamento p/ conjuntos regulares

Automata Bidirecionais



Será que isto <u>aumenta</u> o poder do automato?

Um 2AFD é uma **quíntupla** $M = (Q, \Sigma, \delta, q_0, F)$

- Q = conjunto finito de estados, Σ = alfabeto
- δ = função de transição Q x $\Sigma \rightarrow$ Q x {L, R}
  - δ(q,a)=(p,L): lê a em q, vai p/ estado p e move p/ esquerda,
  - δ(q,a)=(p,R): lê a em q, vai p/ estado p e move p/ direita
- q_0 = estado inicial, F $\subseteq$Q = conjunto de estados finais

Para AFs, podemos definir operação do autômato para uma dada cadeia. Isto não serve para um 2AFD: preciso indicar a **direção** de movimento da fita **para cada novo símbolo** da cadeia que é lido...

2AFD: Operação sobre Cadeias

Def.: Uma <u>descrição instantânea</u> de um 2AFD *M* é uma cadeia $wqx \in \Sigma^* Q \, \Sigma^*$ tal que:

- *wx* é a Cadeia de entrada;
- *q* é o estado atual do autômato;
- a cabeça está lendo o 1o. símbolo da subcadeia *x*.

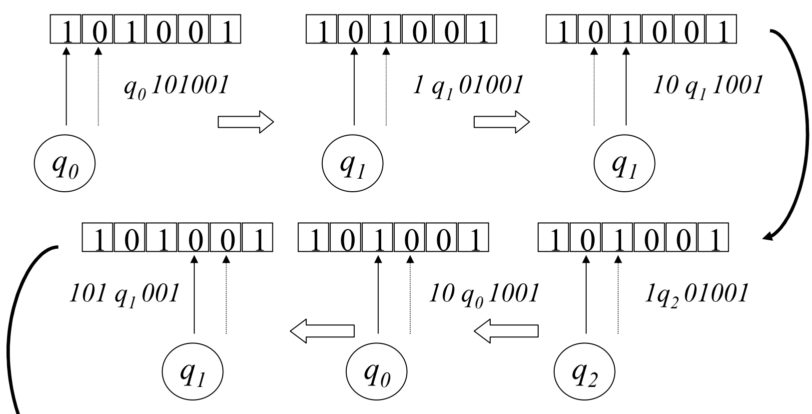
Def.: Uma transição $\overset{M}{\succ}$ em um 2AFD é:

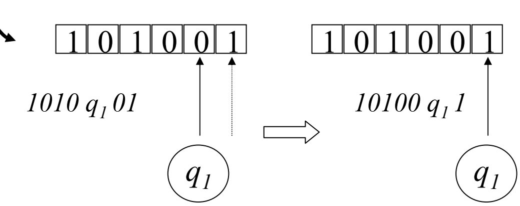
$$a_1 a_2 \ldots a_{i-1} q a_i \ldots a_n \succ a_1 a_2 \ldots a_{i-1} a_i p a_{i+1} \ldots a_n \text{ se } \delta(q, a_i) = (p, R)$$

$$a_1 a_2 \ldots a_{i-1} q a_i \ldots a_n \succ a_1 a_2 \ldots a_{i-2} p a_{i-1} a_i \ldots a_n \text{ se } \delta(q, a_i) = (p, L) \text{ e } i > 1$$

Def.: Uma linguagem *L(M)* aceita por um 2AFD é formada por cadeias *w* tais que

$$q_0 w \overset{*}{\succ} wp, \text{ para algum } p \in F$$





	0	1
q_0	(q_0,R)	(q_1,R)
q_1	(q_1,R)	(q_2,L)
q_2	(q_0,R)	(q_2,L)

2AFD Só Aceitam Linguagens Regulares

Teorema: Se *L* é aceita por um 2AFD, então L é uma linguagem regular.

Prova: Hopcroft/Ullman, págs. 40-41

Ou seja:

Um 2AFD é equivalente a um AFD...

Autômata Finitos com Saída

Até agora, só vimos AFs que não produzem saída (apenas indicam se aceitam ou não uma Cadeia...)

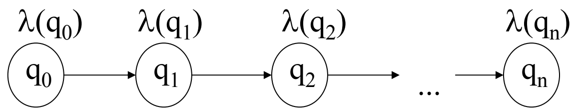
Máquinas de Moore: saídas associadas ao estado.

Máquinas de Mealy: saídas associadas à transição.

Máquinas de Moore

Def.: Uma máquina de Moore *M* é uma sextupla $(Q,\Sigma,\Delta,\delta,\lambda,q_o)$, onde Δ é o alfabeto da saída e $\lambda:Q\rightarrow\Delta$ define a saída associada a cada estado.

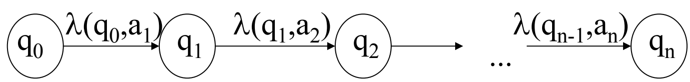
Entrada $a_1a_2...a_n$ **Estados** $q_0q_1...q_n$ **Saída** $\lambda(q_0)\lambda(q_1)...\lambda(q_n)$



Máquinas de Mealy

Def.: Uma máquina de Mealy *M* é uma sextupla $(Q,\Sigma,\Delta,\delta,\lambda,q_o)$, onde Δ é o alfabeto da saída e $\lambda:Q \times \Sigma\rightarrow\Delta$ define a saída associada a cada transição.

Entrada $a_1a_2...a_n$ **Estados** $q_0q_1...q_n$ **Saída** $\lambda(q_0,a_1)\lambda(q_1,a_2)...\lambda(q_{n-1},a_n)$



Equivalência: Moore → Mealy

Teorema 1 : Se $M_1 = (Q,\Sigma,\Delta,\delta,\lambda,q_o)$, é uma máquina de Moore, então existe uma máquina de Mealy M_2 equivalente, <u>desde que</u> ignoremos a primeira saída da máquina de Moore.

Prova: Seja $M_2 = (Q,\Sigma,\Delta,\delta,\lambda',q_o)$ uma máquina de Mealy tal que $\lambda'(q,a) = \lambda(\delta(q,a))$. Então:

a) M_2 realiza as mesmas transições de M_1, sob o mesmo conj. de estados e alfabeto.

b) M_2 produz as mesmas saídas de M_1, a menos da primeira:

M_1 produz $\lambda(q_0) = \lambda(\delta(q_0,\varepsilon))$. M_2 produz $\lambda'(q_0, \varepsilon)= \varepsilon$ (ignoro)

M_1 produz $\lambda(q_1) = \lambda(\delta(q_0,a_1))$. M_2 produz $\lambda'(q_0, a_1)$ ok!

M_1 produz $\lambda(q_2) = \lambda(\delta(q_1,a_2))$. M_2 produz $\lambda'(q_1, a_2)$ ok!

Equivalência: Mealy → Moore

Teorema 2 : Se $M_1 = (Q,\Sigma,\Delta,\delta,\lambda,q_o)$, é uma máquina de Mealy, então existe uma máquina de Moore M_2 equivalente.

Prova: Seja $M_2 = (Q\text{x } \Delta,\Sigma,\Delta,\delta',\lambda',[q_o,b_0])$ uma máquina de Moore tal que:

i) b_0 é um elemento qualquer de Δ.

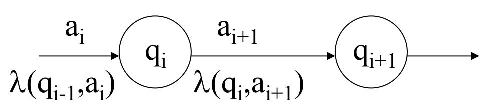
ii) $\delta'([q,b],a)=[\delta(q,a), \lambda(q,a)]$

iii) $\lambda'([q,b])=b$

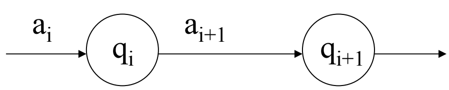
M_2 produz as mesmas saídas de M_1, sobre um conjunto de estados univocamente mapeado a partir dos estados de M_1.

Equivalência: Mealy → Moore

Mealy:



Moore:



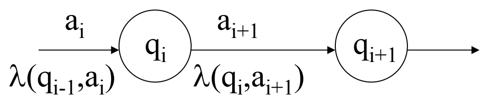
$\lambda'(q_i)=\lambda(q_{i-1},a_i)$

Não funciona! Por def., λ' deveria mapear estados em saídas ($Q\to\Delta$), não dependendo de estados e entradas anteriores... E o que fazer se existirem 2 transições distintas p/ um estado?

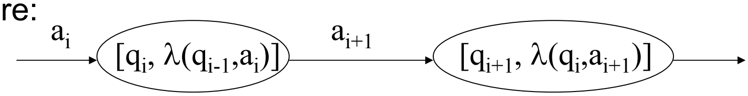
Equivalência: Mealy → Moore

Solução: representar a saída produzida pela última transição Mealy como parte do estado Moore.

Mealy:



Moore:



$\lambda'([q_i, \lambda(q_{i-1},a_i)])=\lambda(q_{i-1},a_i)$

λ' mapeia "estados" $Q \times \Delta \rightarrow \Delta$, indep. de estados/entradas ant.

Linguagens Regulares: Propriedades de Fechamento

Relembrando a definição de linguagem regular:

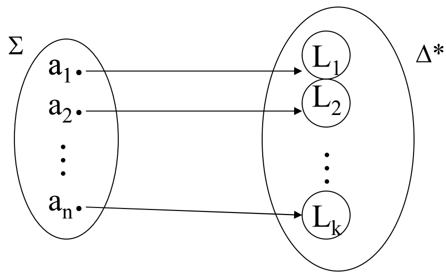
- Ø, {ε} e {a}, para todo $a \in \Sigma$, são linguagens regulares.
- **Passo recursivo:** X e Y linguagens regulares ⇒ $X \cup Y$, XY e X^* são linguagens regulares.
- **Fechamento:** X é linguagem regular apenas se obtido dos elementos básicos por um número finito de aplicações do passo recursivo.

Uma linguagem é regular sss é reconhecida por um AFD.

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

Uma substituição f é um mapeamento de um alfabeto Σ em subconjuntos de Δ^*, para algum alfabeto Δ:



Estendendo a definição para Cadeias: $f(\varepsilon)= \varepsilon$, $f(xa)=f(x)f(a)$

E para linguagens: $f(L) = \bigcup_{x \in L} f(x)$

Substituições: Exemplos e Fechamento

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

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

Exemplo 2: $\Sigma=\{0, 1\}$ $\Delta=\{a, b\}$

$f(0)=\mathbf{a}$, $f(1)=\mathbf{b}^*$, $L=\mathbf{0^*(0+1)1^*} \Rightarrow f(L)=\mathbf{a^*(a+b^*)(b^*)^*} = \mathbf{a^*b^*}$

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

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

Homomorfismos e Homomorfismos Inversos

Def.: Um homomorfismo h é uma substituição que mapeia um alfabeto Σ em subconjuntos unitários de Δ^*, para algum alfabeto Δ.

Homomorfismos mapeiam símbolos em cadeias.

Def.: Um homomorfismo inverso h^{-1} da linguagem L é $h^{-1}(L)=\{x \mid h(x)\in L\}$
Def.: Um homomorfismo inverso h^{-1} da cadeia w é $h^{-1}(w)=\{x \mid h(x)=w\}$

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

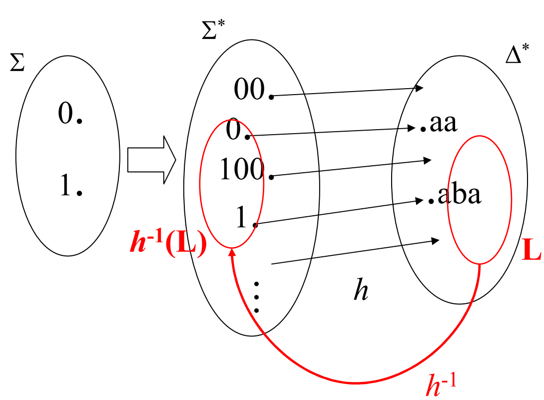
$h(0) = aa$, $h(1) = aba \Rightarrow h(010) = h(0)h(1)h(0) = aaabaaa$

Exemplo 2: $\Sigma=\{0, 1\}$, $\Delta=\{a, b\}$

$h(0) = aa$, $h(1) = aba$, $L = \mathbf{(01)^*} \Rightarrow h(L) = \mathbf{(aaaba)^*}$

Homomorfismos Inversos: Exemplo

$\Sigma=\{0, 1\}$ $\Delta=\{a, b\}$
$h(0) = aa$, $h(1) = aba$, $L = \mathbf{(ab+ba)^*a}$ $h^{-1}(L) = ?$



Homomorfismos Inversos: Exemplo

Σ={0, 1} Δ={a, b} *h*(0) = aa, *h*(1) = aba, L = **(ab+ba)*a** $h^{-1}(L)$ = ?

i) Cadeia w em L começando com b: não pode ser *h*(x) para nenhuma cadeia x de 0's e 1's! (pois *h*(0) e *h*(1) começam com a).

Logo, cadeias w em L que <u>podem</u> ser imagem de cadeias x em Σ^* devem começar com a.

ii) Caso 1: w=a Impossível (*h* sobre alfabeto Σ é Cadeia de pelo menos dois símbolos).

Caso 2: w=abw', para algum w' em **(ab+ba)*a**

Neste caso, devo ter h^{-1}(w) começando com 1 (aba), de modo a garantir o b na segunda posição. Mas aí, tenho 2 possibilidades:

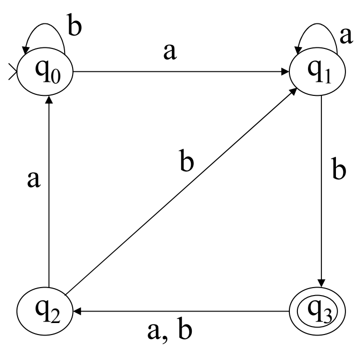
Caso 2.1: w'=a. Neste caso: w=aba, e portanto $\mathbf{h^{-1}(w) = 1}$.

Caso 2.2: w'=abw'' $\Rightarrow$ w=ababw''. Impossível de ser gerado a partir de *h*(0) e *h*(1).

- **Teorema:** ***A classe das linguagens regulares é fechada sob homomorfismos.*** *Y. Bar-Hilel, M. Perles e E. Shamir [1961]*
- **Teorema:** ***A classe das linguagens regulares é fechada sob homomorfismos inversos.*** *S.Ginsburg e G. Rose [1963]*

O *Pumping Lemma* para Conjuntos Regulares

Análise Informal:



a**bab**baaab ✓

a**babbab**baaab ✓

...

a **(bab)^i** baaab ✓

u **v** **w**

Subcadeia "bombeada" (pumped)

O *Pumping Lemma* para Conjuntos Regulares

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

Prova: Caminho de comprimento k: k+1 estados.

Mas existem k nós em M: existirá um nó q_i que ocorre pelo menos duas vezes. A subtrajetória entre a primeira e a segunda ocorrências de q_i é o ciclo.

Corolário: Seja *G* o grafo de estados de um AFD *M* com k estados, e seja p um caminho de comprimento maior igual a k. Então p pode ser decomposto em subtrajetórias q, r e s (p=qrs), em que o comprimento de qr é menor ou igual a k e r é um ciclo.

O *Pumping Lemma* para Conjuntos Regulares

Lema (*Pumping Lemma*): Seja L uma linguagem regular aceita por um AFD M com k estados. Seja *z* uma Cadeia em L com comprimento maior ou igual a k. Então *z* pode ser escrito como *uvw*, onde:

$$length(uv) \leq k, length(v) > 0 \text{ e } uv^i w \in L, \forall i \geq 0$$

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

Muito útil para provar que uma dada linguagem **não é** regular: Basta mostrar que não existe uma decomposição *uvw* de uma Cadeia da linguagem que satisfaça as condições do lema.

Exemplo 1: L={$z \in$ {a, b}* | length(z) é quadrado perfeito}

Prova por contradição

Assuma, por absurdo, que L é regular.

Então L é aceita por algum AFD de k estados.

Pumping lemma: $z \in L$ e length(z) $\geq$ k $\Rightarrow$ $z = uvw$, length (uv) $\leq$ k, $v \neq \varepsilon$ e $uv^i w \in L$.

Vou mostrar que existe uma Cadeia decomposta de acordo com o *Pumping Lemma* cujo comprimento não é quadrado perfeito:

Seja z tal que length(z)=k^2 (um quadrado perfeito)

Pumping Lemma: $z=uvw$, 0 < length(v) $\leq$ k, v é ciclo $\Rightarrow$ $uv^2w \in L$

length(uv^2w) = length(uvw) + length (v) $\leq$ $k^2 + k < k^2 + 2k + 1 = (k+1)^2$

Ou seja: k^2 < length(uv^2w) < $(k+1)^2 \Rightarrow uv^2w \in L$ não é quadrado perfeito (não pertence a L)!

Exemplo 2: L={$a^i b^i$ | $i \geq 0$}

Prova por contradição

Assuma, por absurdo, que L é regular.

Então L é aceita por algum AFD de k estados.

Pumping lemma: $z \in L$ e length(z) $\geq$ k $\Rightarrow$ $z = uvw$, length (uv) $\leq$ k, $v \neq \varepsilon$ e $uv^i w \in L$.

Vou mostrar que existe uma cadeia decomposta de acordo com o *Pumping Lemma* que não pertence à L:

Seja $z=a^k b^k$ (pertence a L, tem comprimento 2k)

Pumping Lemma: $z=uvw=a^i \mathbf{a^j} a^{k-i-j} b^k$, $i+j \leq$ k e j > 0.

Como v é o ciclo, posso realizar o bombeamento:

$$uv^2w = a^i \mathbf{a^j a^j} a^{k-i-j} b^k = a^k a^j b^k \notin L\ !$$

Exemplo 3: L={Cadeias de 0´s e 1´s começando com 1, que são representações binárias de números primos}

Lema A: existe uma quantidade infinita de números primos.

Lema B (Fermat): $2^{p-1} \equiv 1$ mod p, para qualquer primo p (ou seja: $2^{p-1} - 1$ é divisível por p).

Prova por contradição

Assuma, por absurdo, que L é regular.

Então L é aceita por algum AFD de k estados.

Pumping lemma: $z \in L$ e length(z) $\geq$ k $\Rightarrow$ $z = uvw$, length (uv) $\leq$ k, $v \neq \varepsilon$ e $uv^i w \in L$.

Vou mostrar que existe uma Cadeia decomposta de acordo com o *Pumping Lemma* que não pertence à L:

Seja z=rep. binária de um primo p tal que p>2^k (existe - **Lema A**)

Pumping lemma: $z \in L$ e length(z)=| z | $\geq$ k $\Rightarrow$ $z = uvw$, length (v) = | v | > 0 e $uv^i w \in L$ (ou seja, $uv^i w$ é representação binária de um primo). Em particular, $uv^p w$ é um primo q tal que:

$$q_{10} = n_u 2^{|w|+p|v|} + n_v 2^{|w|}\left(1 + 2^{|v|} + \ldots + 2^{(p-1)|v|}\right) + n_w$$

Onde n_u, n_v e n_w são os valores na base 10 de u, v e w.

Lema B: $2^{(p-1)} \equiv 1 \bmod p \therefore 2^{(p-1)|v|} \equiv 1 \bmod p$

e portanto $2^{p|v|} = 2^{(p-1)|v|}2^{|v|} \equiv 2^{|v|} \bmod p$

Seja $s = \left(1 + 2^{|v|} + \ldots + 2^{(p-1)|v|}\right)$

então $\left(2^{|v|} - 1\right)s = 2^{p|v|} - 1 = 2^{|v|} - 1 \bmod p$

e logo $\left(2^{|v|} - 1\right)(s - 1)$ é div. por p. Mas $1 \leq |v| \leq k \Rightarrow 2 \leq 2^{|v|} \leq 2^k < p$,

ou seja: p não pode dividir $\left(2^{|v|} - 1\right)$. Logo, p divide $(s - 1)$, ou seja:

$s \equiv 1 \bmod p$

Mas: $q_{10} = n_u 2^{|w|+p|v|} + n_v 2^{|w|} s + n_w$

e portanto

$$q_{10} \equiv n_u 2^{|w|+|v|} + n_v 2^{|w|} + n_w \bmod p$$

Só que $p_{10} \equiv n_u 2^{|w|+|v|} + n_v 2^{|w|} + n_w$

e portanto concluo que $q \equiv p \bmod p$

Ou seja: um primo q é divisível por p...

CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro carlos@comp.ita.br
Ramal: 5895 Sala: 106

AULA 4

Algoritmos de decisão p/ AFs
Minimização de automata finitos
Teorema Myhill-Nerode

Algoritmos de Decisão para AFDs

Seria ótimo se tivéssemos algoritmos para:

- Determinar se uma dada linguagem é vazia (nenhuma cadeia);
- Determinar se uma dada linguagem é finita (no. finito de cadeia);
- Determinar se uma dada linguagem é infinita (no. infinito de cadeia);
- Determinar se dois autômatos aceitam a mesma linguagem.

Felizmente, estes algoritmos **existem**! Vamos então estudá-los...

Teorema: Seja M um AFD com k estados.

Então, L(M) é não-vazio $\Leftrightarrow$ M aceita uma cadeia z com length(z) < k.

Prova:

i) $\Leftarrow$ Trivial!

ii) $\Rightarrow$ L(M) não-vazio. Seja x a menor cadeia de L(M). Assuma (**por absurdo**) que length(x) > k-1.

Pelo *Pumping Lemma*: x=uvw, com $uv^iw \in L$. Em particular, $uv^0w = uw \in L$. Mas length(uw)<length(x), ou seja, a menor cadeia de L (x) é maior do que uma outra cadeia uw de L (absurdo).

Teorema: Seja M um AFD com k estados.

Então, L(M) tem um número infinito de cadeias $\Leftrightarrow$ M aceita uma cadeia x com k $\leq$ length(z) < 2k.

Prova:

- $\Leftarrow$ M aceita uma cadeia x com k $\leq$ length(x) < 2k.

Pelo *Pumping Lemma*: x=uvw, com $uv^iw \in L$, qualquer que seja i: número infinito de cadeias (diferindo apenas no número de "ciclos" realizados).

- $\Rightarrow$ L(M) tem um número infinito de cadeias .

O número de cadeias com length(.) < k é limitado pelo no. de símbolos do alfabeto e no. de estados. Logo, teremos alguma cadeia x tal que length(x) $\geq$ k.

Assuma (**absurdo**) que não existe z tal que k $\leq$ length(z) < 2k.

Seja x então a menor cadeia tal que length(x) $\geq$ 2k.

Pumping Lemma: x=uvw, length(v) $\leq$ k e $uv^0w = uw \in L(M)$.

Se length(uw) $\geq$ 2k, então, x não seria a menor cadeia tal que length(x) $\geq$ 2k (pois length(uw) < length(x)).

Se length(uw) < 2k, então length(uw) $\geq$ k, pois senão teríamos:

$$\text{length}(x)=\underbrace{\text{length}(uw)}_{<k} + \underbrace{\text{length}(v)}_{\leq k}< 2k \text{ (contradição)}$$

Algoritmo para Determinar Cardinalidade de Linguagem Regular

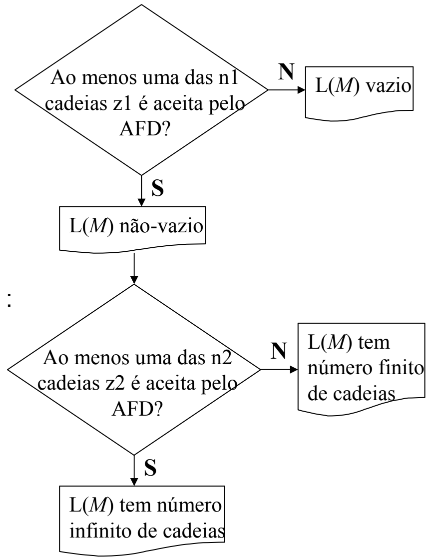
Sejam **k** o número de estados e **j** o tamanho do alfabeto de uma AFD *M*.

No. **n1** de cadeias z1 não-nulas, length(z1) < k :

$$j + j^2 + \ldots + j^{k-1} = j(j^k - 1) / (j-1)$$

No. **n2** de cadeias z2, $k \le$ length(z2) < 2k :

$$j^k + j^{k+1} + \ldots + j^{2k-1} = j^k(j^k - 1) / (j-1)$$



Algoritmo para Determinar Equivalência de AFDs

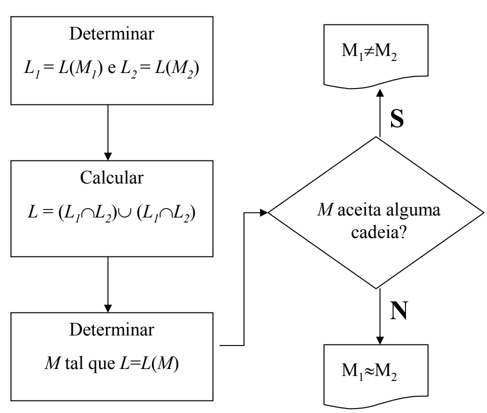
Def.: Dois automata finitos determinísticos M_1 e M_2 são equivalentes se aceitam a mesma linguagem (ou seja, se $L(M_1) = L(M_2)$).

Teorema: Sejam M_1 e M_2 AFDs. Existe um procedimento para determinar se M_1 e M_2 são equivalentes.

Prova: Sejam L_1 e L_2 as linguagens aceitas por M_1 e M_2. Considere então a linguagem

$$L = (L_1 \cap \overline{L}_2) \cup (\overline{L}_1 \cap L_2)$$

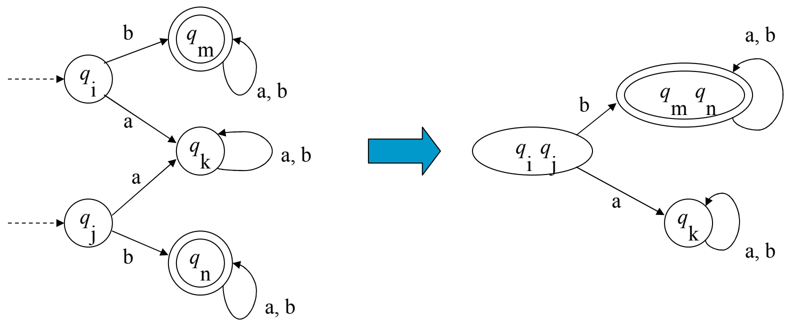
- *L é regular*
- $L = \varnothing$ sss L_1 e L_2 forem idênticos



Equivalência de Estados de um AFD

Def.: Seja M=(Q,Σ,δ,q_0,F) um AFD. Dois estados q_i e q_j são **equivalentes** se

$$\hat{\delta}(q_i, u) \in \mathrm{F} \Leftrightarrow \hat{\delta}(q_j, u) \in \mathrm{F}, \text{ para todo } u \in \Sigma^*$$



Estados equivalentes são também ditos **não-distinguíveis**

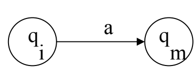
Estados não-equivalentes são também ditos **distinguíveis**

Algoritmo para Identificar Equivalências de Estados

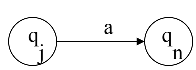
Associados a cada par q_i , q_j (i>j), dois vetores D (de 0´s e 1´s) e S (de conjuntos de pares de inteiros):

- D[i, j]: valor 1 quando q_i e q_j são distinguíveis, caso contrário valor 0.
- S[i, j]: conjunto de pares (m,n) tais que a não-equivalência de q_m e q_n pode ser determinada a partir daquela de q_i e q_j.

Idéia do algoritmo:



D[n,m]=1: q_m e q_n distinguíveis. Ao examinar q_i e q_j observo que S[m,n] contém o par [i,j], logo são distinguíveis também (e faço D[i,j]=1).



D[n,m] não definido: não sei se q_m e q_n são distinguíveis. Ao examinar q_i e q_j observo que S[m,n] contém o par [i,j], logo serão distinguíveis se, em algum momento, descobrir que D[n,m]=1 (e aí faço D[i,j]=1).

Algoritmo para Identificar Equivalências de Estados

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

- **Inicialização**

  Para cada par q_i, q_j com $i<j$ faça $D[i,j]=0$, $S[i,j]=\varnothing$
- **Para cada** par i, j com $i<j$

  Se q_i (q_j) é um estado final e q_j (q_i) não é um estado final **então** faça $D[i, j]=1$ (estados distinguíveis para a cadeia vazia).
- **Para cada** par i, j com $i<j$ e $D[i, ,j]=0$

  Se para algum $a\in\Sigma$, $\delta(q_i, a)=q_m$ e $\delta(q_j, a)=q_n$ e $D[n,m]=1$ (ou $D[m,n]=1$), então DIST(i, j)

  Senão para cada $a\in\Sigma$ faça

  $\delta(q_i, a)=q_x$ e $\delta(q_j, a)=q_y$

  Se $m<n$ e $[i, j] \neq [x, y]$ **então** adicione [i, j] a S[x, y]

  Senão se $m>n$ e $[i, j] \neq [x, y]$ **então** adicione [i, j] a S[y, x]

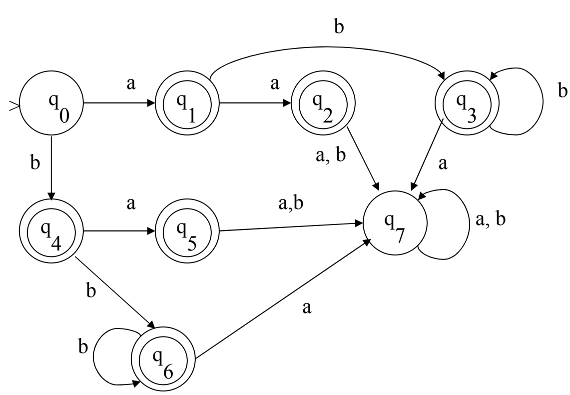
DIST(i, j):

D[i, j]=1

para todo $[m, n] \in S[i, j]$ DIST(m, n)

Exemplo

Determine o AFD mínimo equivalente a:



Relembrando: Relações de Equivalência

Uma relação binária $\equiv$ sobre um conjunto X é de equivalência se:

- a $\equiv$ a para todo a $\in$ X (propriedade reflexiva);
- a $\equiv$ b e b $\equiv$ c implica a $\equiv$ c para todo a,b,c $\in$ X (propriedade transitiva);
- a $\equiv$ b implica b $\equiv$ a para todo a,b $\in$ S (propriedade de simetria)

A **classe de equivalência** (c.e.) sobre $\equiv$ de um elemento $a \in$ X é o conjunto

$$[a]_{\equiv} = \{b \in X \mid a \equiv b\}.$$

Ou seja: a classe de equivalência do elemento a é o subconjunto formado por aqueles elementos de X que se relacionam com a pela relação de equivalência em questão.

Propriedade 1: Seja $\equiv$ uma relação de equivalência sobre X, e sejam a e b elementos de X. Então, $[a]_{\equiv} = [b]_{\equiv}$ ou $[a]_{\equiv} \cap [b]_{\equiv} = \varnothing$.

Ou seja: classes de equivalência sobre X formam uma família disjunta de subconjuntos de X.

Propriedade 2: Seja $\equiv$ uma relação de equivalência sobre X. As c.e.s de $\equiv$ formam uma partição sobre X.

Ou seja: a união de classes de equivalência sobre X formam o conjunto X, e as classes de equivalência são disjuntas entre si.

Equivalência de cadeias

- **Def.:** Seja L uma linguagem sobre Σ. Duas cadeias **u** e **v** são não-distingüíveis (ou equivalentes) em L (**u** $\equiv_L$ **v**) se, para todo **w** $\in \Sigma^*$:
  - **uw** e **vw** estão em L; ou
  - **uw** e **vw** não estão em L.

*Ou seja: se existir <u>ao menos uma cadeia **w**</u> tal que **uw** esteja em L e **vw** não esteja em L (ou vice-versa), então u e v são distingüíveis.*

- **Teorema:** Para qualquer linguagem L, a relação $\equiv_L$ é uma relação de equivalência.

Prova:

1. $\equiv_L$ é reflexiva, pois **u** $\equiv_L$ **u**: para todo **w** $\in \Sigma^*$, **uw** e **uw** estão (ou não) ambos em L.
2. $\equiv_L$ é simétrica, pois se **u** $\equiv_L$**v** então **v** $\equiv_L$**u** : para todo **w** $\in \Sigma^*$,
   Se **uw** e **vw** estão ambos em L, então **vw** e **uw** estão ambos em L.
   Se **uw** e **vw** não estão em L, então **vw** e **uw** não estão em L.
3. $\equiv_L$ é transitiva, pois se **u** $\equiv_L$**v** e **v** $\equiv_L$**y**, então **u** $\equiv_L$**y** : para todo **w** $\in \Sigma^*$,
   Se **uw** e **vw** estão em L (para todo w) e **vw** e **yw** estão em L (para todo w), então **uw** e **yw** estão em L (para todo w).
   Se **uw** e **vw** não estão em L (para todo w) e **vw** e **yw** não estão em L (para todo w), então **uw** e **yw** não estão em L (para todo w).

Equivalência de cadeias: Exemplo 1

L = **a** (**a** ∪ **b**) (**bb**)*

aa e ab são indistingüíveis:

aa**w** e ab**w** estão ambos em L se **w** é um número par de b's.

aa**w** e ab**w** não estão em L para qualquer outro **w**.

b e ba são indistingüíveis:

b**w** e ba**w** não estão em L para nenhum **w**.

a e ab são distingüíveis:

a**w** está em L para **w**=a

ab**w** não está em L para **w**=a

Classes de equivalência de $\equiv_L$:

Elemento representativo	***Classe de equivalência***
$[\varepsilon]_L$	ε
[**b**]$_L$	**b**(**a**∪**b**)* ∪ **a**(**a**∪**b**)**a**(**a**∪**b**)* ∪ **a**(**a**∪**b**)**ba**(**a**∪**b**)*
[**a**]$_L$	**a**
[**aa**]$_L$	**a** (**a**∪**b**)(**bb**)***a**
[**aab**]$_L$	**a** (**a**∪**b**)**b**(**bb**)*

Equivalência de cadeias: Exemplo 2

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

Cadeias a^i e a^j para todo $i \neq j$: distingüíveis.

Classes de equivalência: a, aa, aaa, aaaa, ...

(cada valor de i define uma classe de equivalência com um único elemento)

Um número infinito de classes de equivalência, cada uma com um único elemento...

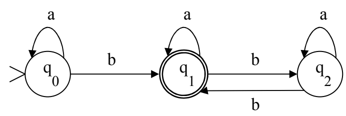
Equivalência de cadeias: uma nova definição

Def.: Seja $M=(Q,\Sigma,\delta,q_0,F)$ um AFD. Duas cadeias ***u*** e ***v*** são **equivalentes** (***u*** $\equiv_M$ ***v***) se

$$\hat{\delta}(q_0,u)=\hat{\delta}(q_0,v)$$

Teorema: Para qualquer autômato M, $\equiv_M$ é uma relação de equivalência.

Exemplo:



aceita $\mathbf{a^*ba^*(ba^*ba^*)^*}$

Estado	*Classe de equivalência associada*
q_0	$\mathbf{a^*}$
q_1	$\mathbf{a^*\,ba^*\,(ba^*ba^*)^*}$
q_2	$\mathbf{a^*\,ba^*\,ba^*\,(ba^*ba^*)^*}$

Teorema Myhill-Nerode

Def.: Uma relação de equivalência $\equiv$ é <u>invariante à direita</u> em Σ^* se **u** $\equiv$ **v** implica **uw** $\equiv$ **vw** para todo **w** $\in \Sigma^*$.

Teorema Myhill-Nerode
As seguintes afirmativas são equivalentes:

i) L é regular sobre Σ
ii) Existe uma relação de equivalência $\equiv$ invariante à direita em Σ^* com um conjunto finito de classes de equivalência de tal modo que L é a união de um subconjunto das classes de equivalência de $\equiv$.
iii) $\equiv_L$ tem conjunto finito de classes de equivalência.

Prova

i) $\rightarrow$ ***ii)*** *: L é regular* $\rightarrow$ *L aceita por um AFD M=(Q, Σ, δ, q_0, F).*

Seja então a relação de equivalência $\equiv_M$ definida para este AFD.

Esta relação é invariante à direita, pois u $\equiv_M$ v implica $\delta(q_0,u)=\delta(q_0,v) = q_x$, e portanto $\delta(q_0,uw) =\delta(q_x,w)= \delta(q_0,vw)$.

Além disso, o número de classes de equivalência de $\equiv_M$ é finito, pois cada cadeia u relaciona-se com cadeias que terminam no mesmo estado que u, a partir do estado q_0. E como um AFD tem um número finito N de estados, concluímos que o número de classes de equivalência de $\equiv_M$ é no máximo N.

Finalmente, note que L é o conjunto de cadeias que terminam em algum estado em F. Para cada estado q_i, existirá uma c.e. (possivelmente com um único elemento ou mesmo vazia) formada por cadeias cuja computação termina em q_i. A linguagem L é a união das cadeias que terminam em um estado de F, ou seja: a união dos conjuntos de cadeias cujas classes de equivalência estão associadas a estados de F.

Teorema Myhill-Nerode

Prova (cont.)

ii) → iii) : *Seja* ≡ a relação de equivalência que satisfaz (ii). Seja [u] uma c.e. desta relação, cujo elemento representativo é a cadeia u. Seja v uma segunda cadeia em [u], ou seja: u ≡ v.

Como ≡ é invariante à direita, então uw ≡ vw, para todo w. Logo, uw e vw estão numa mesma c.e. na relação de equivalência ≡.

Como por (ii) L é a união de algumas das c.e. de ≡, toda cadeia em uma dada c.e. ou está em L (se pertencer à alguma c.e. que forma L) ou não está em L (se pertencer a alguma c.e. que não forma em L).

Logo, ou uw e vw estão em L, ou uw e vw não estão em L. Pela definição de $\equiv_L$, u $\equiv_L$ v.

O que isto mostra é que: para toda cadeia u, se u e v estão na mesma c.e. [u] ≡, então estão também na mesma c.e. [u] $\equiv_L$. Portanto, cada c.e. de ≡ é um subconjunto de alguma c.e. de $\equiv_L$.

E não será possível termos alguma c.e. de $\equiv_L$ para a qual não existam elementos de alguma c.e. de ≡. Se assim fosse não teríamos ≡ definido um conjunto de c.e. que particionam Σ^* .

Conclusão: como o número de c.e.s de ≡ é finito, o conjunto de c.e.s de $\equiv_L$ também é finito.

Teorema Myhill-Nerode

Prova (cont.)

iii) → i) : Construiremos um AFD que aceita L (prova por construção):

O alfabeto é Σ e os estados serão as c.e.s de $\equiv_L$. O estado inicial será a c.e. que contém ε.

Provemos que cada c.e. de fato é um estado, ou seja, as transições não dependem de estados passados ou do particular elemento da c.e. escolhido como elemento representativo. Ou seja, definindo

$$\delta([u]_L, a) = [ua]_L$$

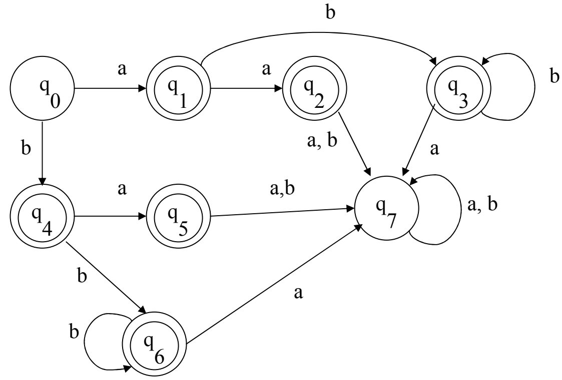
devemos mostrar que esta definição independe do valor de u.

Sejam então u e v duas cadeias em $[u]_L$.
Como $[u]_L = [v]_L$, devemos ter $\delta([u]_L, a) = [ua]_L = [va]_L = \delta([v]_L, a)$.
Ou seja, preciso mostrar que ua $\equiv_L$ va. Pela definição de $\equiv_L$ isto equivale a mostrar que uax e vax estão ambos em L ou estão ambos fora de L, para qualquer cadeia $x \in \Sigma^*$. Mas como u $\equiv_L$ v, então:

uw e vw estão ambos em L, para toda cadeia w. Se w = ax, então uax e vax estão ambos em L, para toda cadeia x **ou** uw e vw não estão em L, para toda cadeia w. Se w = ax, então uax e vax estão ambos fora de L, para toda cadeia x.

Falta só mostrar que a linguagem aceita por este AFD construído é L. Isto pode ser feito construindo o AFD e definindo que se $u \notin L$ antão $[u]_L$ não é um estado de aceitação. *(Prova como exercício)*

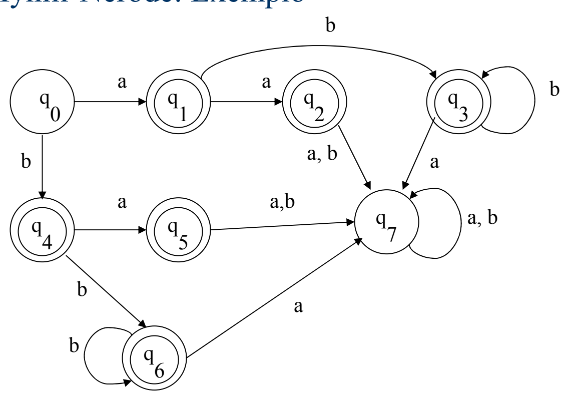
Teorema Myhill-Nerode: Exemplo



Classes de $\equiv_M$

Estado	Classe de equivalência	Estado	Classe de equivalência
q_o	ε	q_4	b
q_1	a	q_5	ba
q_2	aa	q_6	bbb^*
q_3	abb^*	q_7	$(aa(a\cup b) \cup abb^*a \cup ba(a\cup b) \cup bbb^*a)(a\cup b)^*$

Teorema Myhill-Nerode: Exemplo



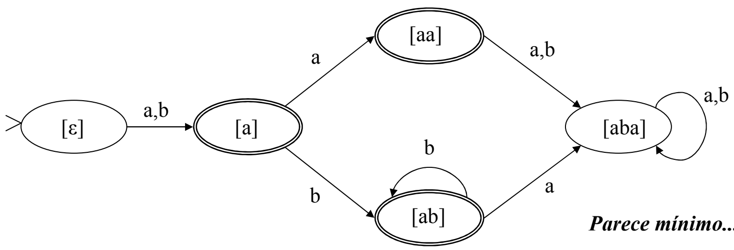
Classes de $\equiv_L$

Elemento representativo	Classe de equivalência
[a]	$a \cup b$
[aa]	$aa \cup ba$
[ab]	$abb^* \cup bbb^*$
[ε]	ε
[aba]	$(aa(a\cup b) \cup abb^*a \cup ba(a\cup b) \cup bbb^*a)(a\cup b)^*$

Teorema Myhill-Nerode: Exemplo

Relação entre $\equiv_M$ e $\equiv_L$

Elemento representativo	Classe de equivalência
[a]	$a \cup b$ **(classe $q_1 \cup$ classe q_4)**
[aa]	$aa \cup ba$ **(classe $q_2 \cup$ classe q_5)**
[ab]	$abb^* \cup bbb^*$ **(classe $q_3 \cup$ classe q_6)**
[ε]	ε **(classe q_0)**
[aba]	$(aa(a \cup b) \cup abb^*a \cup ba(a \cup b) \cup bbb^*a)(a \cup b)^*$ **(classe q_7)**



Parece mínimo...

Teorema Myhill-Nerode: Corolário

Seja L uma linguagem regular e $\equiv_L$ a relação de equivalência de cadeias sobre L. O AFD mínimo que aceita L é a máquina de estados M_L definida a partir das classes de equivalência sobre $\equiv_L$, tal como especificada na prova do teorema Myhill-Nerode.

Prova: Sudkamp, pág. 221

Ou seja: o teorema Myhill-Nerode também permite que se obtenha o AFD mínimo para uma dada linguagem L.

E também permite a identificação de linguagens não-regulares (iii).

Exemplo: Prove que L={ a^{2^i} | $i \geq 0$} não é regular}.

É um teorema de mil e uma utilidades!

CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro carlos@comp.ita.br
Ramal: 5895 Sala: 106

AULA 5

Gramáticas e linguagens
A Hierarquia de Chomsky
AFDs e gramáticas regulares

Gramática: Definição

- Uma **gramática** G = (V, Σ, P, S) consiste de:
  - um conjunto finito V de símbolos não-terminais;
  - um conjunto finito Σ de símbolos terminais, onde $V \cap \Sigma = \emptyset$;
  - um subconjunto P de $[(V \cup \Sigma)^* - \Sigma^*] \times (V \cup \Sigma)^*$, chamado conjunto de ***produções*** ou ***regras;***
  - um símbolo inicial $S \in V$.
- Uma produção $(A,B) \in P$ é escrita $A \rightarrow B$, onde $A \in [(V \cup \Sigma)^* - \Sigma^*]$ e $B \in (V \cup \Sigma)^*$.

  Assim, A deve conter ao menos um símbolo não-terminal e B pode conter qualquer combinação de símbolos terminais e não-terminais.

Uma gramática *gera (produz)* as cadeias de uma linguagem.

Exemplos de Gramáticas

Seja a gramática $G = (V, \Sigma, P, S)$, com:

a) $V=\{A,B\}$, $\Sigma=\{0,1,\#\}$, $P=\{A\rightarrow 0A1, A\rightarrow B, B\rightarrow \#\}$, $S=A$.

b) $V=\{S\}$, $\Sigma=\{a,b\}$, $P=\{S\rightarrow Sa,\ S\rightarrow b\}$, $S=S$.

c) $V=\{A,B\}$, $\Sigma=\{0,1\}$, $P=\{B\rightarrow \varepsilon, B\rightarrow A, A\rightarrow 1A, A\rightarrow 0A, A\rightarrow 1, A\rightarrow 0\}$, $S=B$.

d) Uma gramática para gerar palíndromos sobre {a,b}:

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

Derivações

- ***Def.:*** Seja a gramática $G = (V, \Sigma, P, S)$. Se $\alpha\rightarrow\beta$ é uma produção e $x\alpha y\in(V\cup \Sigma)^*$, dizemos que $x\beta y$ é **diretamente derivável** de $x\alpha y$ e escrevemos: $x\alpha y \Rightarrow x\beta y$.

  Se $\alpha_i\in(V\cup \Sigma)^*$ para $i=1,..,n$ e α_{i+1} é diretamente derivável de α_i para $i=1,..,n-1$, dizemos que **α_n é derivável de α_1** e escrevemos: $\alpha_1 \stackrel{*}{\Rightarrow} \alpha_n$

- Derivação de α_n a partir de α_1: $\alpha_1 \stackrel{*}{\Rightarrow} \alpha_n : \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$

Por convenção, qualquer elemento de $(V\cup \Sigma)^*$ é derivável de si mesmo.

- **A linguagem L(G) gerada por G** consiste de todas as cadeias sobre Σ derivadas de S.

As gramáticas G e G' são equivalentes se $L(G) = L(G')$.

Derivações: Exemplos, Formas Sentenciais e Sentenças

- Seja a gramática $G = (V, \Sigma, P, S)$, com

$V=\{S, A\}$, $\Sigma =\{a,b\}$, $P=\{S\rightarrow bS,\ S\rightarrow aA,\ A\rightarrow bA,\ A\rightarrow b\}$

(a) A cadeia abAbb é diretamente derivável de aAbb, escrita como aAbb $\Rightarrow$ abAbb, usando a produção $A\rightarrow bA$.

(b) A cadeia bbab é derivável de S, escrita $S \overset{*}{\Rightarrow}$ bbab. A derivação é: $S\Rightarrow bS\Rightarrow bbS\Rightarrow bbaA\Rightarrow bbab$

- ***Def.:*** Seja a gramática $G = (V, \Sigma, P, S)$. Diz-se que $w\in(V\cup \Sigma)^*$ é uma **forma sentencial** de G se existir derivação $S \overset{*}{\Rightarrow} w$ em G.
- ***Def.:*** Uma cadeia $w \in \Sigma^*$ é uma **sentença** de G se existir uma derivação $S \overset{*}{\Rightarrow} w$ em G.

Forma sentencial: qualquer cadeia (incluindo símbolos não-terminais) deriváveis a partir do símbolo inicial S.

Sentença: forma sentencial apenas com símbolos terminais.

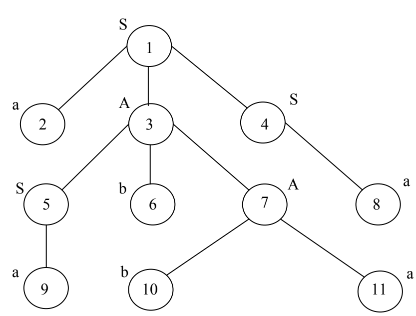
Árvores de Derivação

Forma de representar derivações em uma gramática.

- Nós: símbolos terminais ou não-terminais
- Se existir um vértice A com sucessores $X_1, X_2, \ldots, X_k$, então $A\rightarrow X_1X_2\ldots X_k$ é uma produção da gramática. A recíproca não é necessariamente verdadeira.
- O nó-raiz corresponde a um símbolo inicial S.

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

P: $S \rightarrow aAS \mid a$, $A \rightarrow SbA \mid SS \mid ba$



O produto ***(yield)*** de uma árvore de derivação é a string formada pela leitura sequencial das folhas da árvore.

Neste exemplo*: produto*= aabbaa

Relação entre Derivações e Árvores de Derivação

Teorema: Seja G = (V, Σ,P,S) uma GLC. Então $S \overset{*}{\Rightarrow} \alpha$ sss existir uma árvore de derivação em G com produto α.

Ou seja:

a) Dada uma derivação, existe uma árvore correspondente.

b) Dada uma árvore de derivação, seu produto corresponde a alguma derivação em G.

Prova: Hopcroft / Ullman, pp. 85-86

Derivações à Direita e à Esquerda

<u>Derivação à Direita:</u> cada passo de derivação aplicado à variável mais à direita.

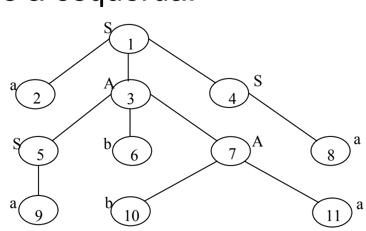
<u>Derivação à Esquerda:</u> cada passo de derivação aplicado à variável mais à esquerda.

<u>Exemplo</u>

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

P: $S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$



À direita: $S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa$

À esquerda: $S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$

Forma Backus-Naur

- BNF ("**B**ackus-**N**aur **F**orm"): modo alternativo de descrever gramática G.

  – símbolos não terminais: inclusos em <..>
  – produção $A \rightarrow T$, escrita: A::=T
  – produções da forma: $A::=T_1$, $A::=T_2$, …, $A::=T_n$ podem ser combinadas em
  $A::= T_1 | T_2 | \ldots | T_n$. *(lê-se "ou" para "|")*

<u>Um exemplo:</u>

Gramática para inteiros - um inteiro é definido como uma cadeia contendo um sinal opcional (+ ou -), seguido por uma cadeia de dígitos (0 a 9).

Símbolo inicial: <inteiro>

<digito>::= 0 |1| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<inteiro>::= <inteiro com sinal> | <inteiro sem sinal>

<inteiro com sinal>::= + < inteiro sem sinal> | - <inteiro sem sinal>

<inteiro sem sinal>::= <digito> | <digito> <inteiro sem sinal>

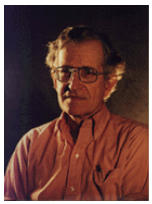
BNF: Exemplo

derivação do inteiro -901

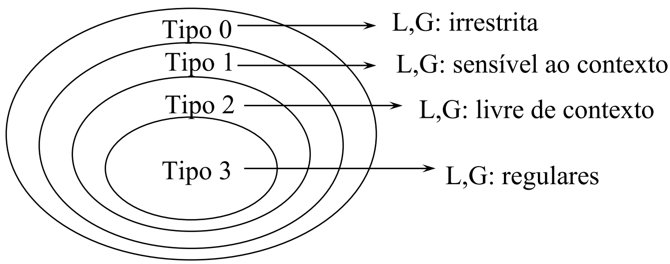
<inteiro> ⇒ <inteiro com sinal>
⇒ - <inteiro sem sinal>
⇒ - <digito> <inteiro sem sinal>
⇒ - <digito> <digito> <inteiro sem sinal>
⇒ - <digito> <digito> <digito>
⇒ - 9 <digito> <digito>
⇒ - 90 <digito>
⇒ - 901.

A Hierarquia de Chomsky

- Conforme as restrições impostas ao formato das produções de uma gramática, varia-se a classe de linguagens que tal gramática gera.
- Existem 4 classes de gramáticas, capazes de gerar 4 classes correspondentes de linguagens, de acordo com a denominada ***Hierarquia de Chomsky***, que estabelece uma relação de inclusão entre as gramáticas.



Noam Chomsky



Linguagens e Gramáticas

- *Def.:* Uma linguagem L é sensível ao contexto (respectivamente livre de contexto, regular) se existe uma gramática sensível ao contexto G (respectivamente livre de contexto, regular) com L=L(G).
- <u>Obs: Uma convenção para a produção nula:</u> Se uma gramática permitir produção da cadeia nula, ela deverá ser da forma $S \rightarrow \varepsilon$, onde S é o símbolo inicial e não pertence ao lado direito de qualquer produção e ε é a cadeia nula. Assim, pode-se tratar esta produção como um caso especial.

É sempre possível transformar uma gramática G em uma gramática equivalente G' que satisfaz a convenção acima (veremos isso mais tarde - slide 22).

Gramáticas Regulares

- ***Definição (inclui itens a,b,c e d):*** *Seja G uma gramática e ε a cadeia nula.*

(a) Se toda produção estiver na forma $A \rightarrow a$ ou $A \rightarrow aB$ ou $A \rightarrow \varepsilon$, com $A, B \in V$, $a \in \Sigma$, G é uma **gramática regular** (ou **tipo 3**).

- Nesta gramática, pode-se substituir um símbolo não terminal por: (i) um símbolo terminal, (ii) um símbolo terminal seguido por um não terminal ou (iii) pela cadeia nula.

<u>Exemplo:</u>

G = (V, Σ, P, S), com Σ ={a,b}, V={S, X}, P={S$\rightarrow$bS, S$\rightarrow$aX, X$\rightarrow$bX, X$\rightarrow$b} é **regular.**

Derivações possíveis: ab, abb, bbbabb,…

L(G)={$b^n a b^m \mid n \geq 0, m \geq 1$} $\Rightarrow$ G é uma gramática regular, portanto, a linguagem L(G) que ela gera é uma linguagem regular!

Gramáticas Livres de Contexto

(b) Se toda produção estiver na forma $A \rightarrow \delta$, com $A \in V$, $\delta \in (V \cup \Sigma)^*$, G é uma **gramática livre de contexto** (ou **tipo 2**).

- Nesta gramática, pode-se substituir A (um não terminal isolado) por δ sempre que se queira, <u>independentemente do contexto</u> em que A esteja inserido.

<u>Exemplo:</u> A gramática G = (V, Σ, P, S), com Σ ={a,b}, V={S}, P={S$\rightarrow$aSb, S$\rightarrow$ab} é **livre de contexto.**

Derivações possíveis: ab, aabb, aaabbb,…

L(G)={$a^n b^n \mid n=1,2,\ldots$} $\Rightarrow$ L é uma linguagem livre de contexto e não é uma linguagem regular!

Linguagens livres de contexto permitem, além das operações permitidas para uma linguagem regular, <u>operações de aninhamento</u>.

A gramática para inteiros dada (slide 9) é livre de contexto. Se mudarmos as produções para:

<digitos>::= 0<digitos> |1<digitos> |... | 9<digitos> | ε

<inteiro>::= + <inteiro sem sinal> | - <inteiro sem sinal> | 0<digitos> |1<digitos> |... | 9 <digitos>

<inteiro sem sinal>::= 0<digitos> |1<digitos> |... | 9<digitos>

resultará numa gramática G regular. Como a linguagem L=L(G) gerada não foi modificada, concluímos que L é uma linguagem regular.

Gramáticas Sensíveis ao Contexto

(c) Se toda produção estiver na forma $\alpha A\beta \rightarrow \alpha\delta\beta$, com $\alpha,\beta \in (V\cup \Sigma)^*$, $A \in V$, $\delta \in (V\cup \Sigma)^* - \{\varepsilon\}$, G é uma **gramática sensível ao contexto** (ou **tipo 1**).

- Nesta gramática, pode-se substituir A por δ se A estiver <u>dentro do contexto</u> α e β.
- Na gramática do tipo 1, $|\alpha A\beta| \leq |\alpha\delta\beta|$, exceto para a produção $S\rightarrow\varepsilon$, com S sendo o símbolo inicial.

Exemplo: A gramática G = (V, Σ, P, S), com Σ ={a,b,c}, V={S,A,B,C,D,E}, P={S$\rightarrow$aAB, S$\rightarrow$aB, A$\rightarrow$aAC, A$\rightarrow$aC, B$\rightarrow$Dc, D$\rightarrow$b, CD$\rightarrow$CE, CE$\rightarrow$DE, DE$\rightarrow$DC, Cc$\rightarrow$Dcc} é **sensível ao contexto**

(ex.: CE$\rightarrow$DE diz que C pode ser substituído por D <u>se</u> C for seguido por E)

Derivações possíveis: abc, aabbcc, aaabbbccc,…

L(G)=$\{a^n b^n c^n \mid n=1,2,\ldots\} \Rightarrow$ não existe uma gramática livre de contexto G com L=L(G); assim, L não é uma linguagem livre de contexto!

Gramáticas Irrestritas

(d) Se toda produção de G estiver na forma $\alpha \rightarrow \beta$, com $\alpha \in [(V \cup \Sigma)^* - \Sigma^*]$ e $\beta \in (V \cup \Sigma)^*$, G é uma **gramática irrestrita** (ou **tipo 0**).

– Nesta gramática, nenhuma limitação é imposta.

<u>Exemplo:</u> A gramática G = (V, Σ, P, S), com Σ ={a,b}, V={S,B,C},

P={S$\rightarrow$BC, BC$\rightarrow$CB, B$\rightarrow$b, C$\rightarrow$a} é **irrestrita.**

L(G)={ab, ba}

Esta produção BC$\rightarrow$CB só é permitida em gramáticas irrestritas.

- ***Outros exemplos***:

1. <A>::= 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
   <A>::= <A> <operador> <A>
   <operador>::= + | - | * | /

   é uma gramática que define uma linguagem regular

2. Se acrescentarmos a produção: <A>::= (<A>), transformamos numa gramática que define uma linguagem *livre de contexto*.

3. <X1>::= { <declaração> <X> }
   <declaração>::= integer <A> | boolean <A>
   <A>::= x | y | z

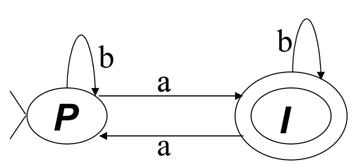
   é uma gramática sensível ao contexto, desde que as variáveis utilizadas em <X> sejam: i) as mesmas que em <declaração>; ii) do mesmo tipo que em <declaração>.

4. Vários exemplos de GLCs (Sudkamp, págs. 67 a 70)

- Uma gramática regular é uma gramática livre de contexto.
- Uma gramática livre de contexto, sem produções do tipo $A \to \varepsilon$, é uma gramática sensível ao contexto.
- Uma gramática sensível ao contexto é uma gramática irrestrita.

Gramáticas Regulares e Automata Finitos

- Nesta seção mostraremos que gramáticas regulares e automata finitos são essencialmente equivalentes, no sentido em que ambos são especificações de uma linguagem regular:
  - A gramática é **geradora** da linguagem
  - O autômato é **reconhecedor** da linguagem (já vimos)
- Seja o autômato finito abaixo, o qual aceita cadeias sobre {a,b} que contêm um número ímpar de a's.



Como determinar a gramática regular equivalente?

Gramáticas Regulares a partir de AFs

Teorema *(Chomsky e Miller, 1958)*:

Seja *M* um AF de estado inicial S. Seja Σ o conjunto dos símbolos de entrada e V o conjunto de estados de M. Defina produções A→*x*A' se existir um arco rotulado *x* de A para A' e A→ε se A for um estado de aceitação. Então a gramática regular G=(V,Σ,P,S) é tal que L(G)=L(M).

Para o exemplo:

- **AF ⇨ G:** os símbolos de entrada {a,b} do AF são os símbolos terminais de G. Os estados **P** e **I** são os símbolos não-terminais. O estado inicial **P** é o símbolo inicial. Os arcos do AF correspondem às produções de G. Se existir um arco rotulado por *x* de A para A', escreve-se a produção: A→*x*A'.

  Temos então: ***P****→ b****P****,* ***P****→ a****I****,* ***I****→ a****P****,* ***I****→ b****I***
- Além disso, se A for estado de aceitação, inclui-se A→ε. *No exemplo:* ***I****→ε.*
- Assim, a gramática G=(V,Σ,P,**P**), com V={**I**,**P**}, Σ={a,b} e P={***P****→ b****P****,* ***P****→ a****I****,* ***I****→ a****P****,****I****→ b****I****,* ***I****→ε*} gera a linguagem L(G), que corresponde ao conjunto de cadeias aceitas pelo autômato finito.

AFs a Partir de Gramáticas Regulares

Teorema *(Chomsky e Miller, 1958)*:

Seja G = (V,Σ,P,S) uma gramática regular. Seja I = Σ, X = V ∪ {F}, onde F ∉ V∪ Σ, f(X,x) = {X' | X→xX' ∈ P} ∪ {F | X →x ∈ P}, A = {F} ∪ {X | X→ε ∈ P}. O autômato finito não-determinístico *M* = (I, X, f, A, S) aceita precisamente as cadeias de L(G).

Exemplo:

- Seja a gramática regular G=(V,T,P,S), com V={S, C}, T={a,b}, P={S→bS, S→aC, C→bC, C→b}. Determinar o AF correspondente.
- **G ⇨ AF:** Os símbolos não terminais serão os estados. Para cada produção da forma A→*x*A', desenhar uma ligação de A a A', com rótulo *x* (produções S→bS, S→aC, C→bC). A produção C→b equivale a: C→bF, F→ε, sendo F um símbolo não-terminal adicional. A produção F→ε indica que F é um estado de aceitação.

Equivalência AFs ⇔ Gramáticas Regulares

- Vimos então que, se A é um autômato finito, existe uma gramática regular G, com L(G)=L(*M*). Vimos também que, se G é uma gramática regular, existe um autômato finito não-determinístico *M*, com L(G)=L(*M*).
- Como já vimos (aula 2) que existe um AFD equivalente a qualquer AFN, concluímos que:

Se G é gramática regular, existe um autômato finito determinístico M, com L(G)=L(M).

CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro carlos@comp.ita.br
Ramal: 5895 Sala: 106

AULA 6

Simplificação de GLCs
Formas normais de Chomsky e Greibach

Normalização de GLCs

- Para GLCs, é possível restringir a quantidade e forma das produções, sem reduzirmos seu poder gerador de linguagem. Isto garante propriedades interessantes:
  - Garantia de terminação do processo de *parsing* (análise sintática);
  - Facilitação da caracterização das linguagens geradas pela gramática.
- Passos preliminares para normalização:
  - Eliminação de Recursão sobre Símbolo Inicial S
  - Eliminação de Regras ε
  - Eliminação de Regras Encadeadas
  - Eliminação de Símbolos Inúteis

Eliminação de Recursão sobre S

- Força o símbolo inicial S a atuar apenas como um iniciador de derivações.

Seja $G=(V,\Sigma,P,S)$ uma GLC. Existe uma GLC $G'=(V',\Sigma,P',S')$ (e um algoritmo correspondente para produzí-la) que satisfaz:

i) $L(G')=L(G)$.

ii) As regras de P' são da forma $A \rightarrow w$, onde $A \in V'$ e $w \in ((V-\{S'\}) \cup \Sigma)^*$.

Exemplo.

Eliminação de Regras ε

- Elimina regras que produzem cadeias nulas (a menos que seja parte da linguagem e a única regra ε seja $S \rightarrow \varepsilon$).

Seja $G=(V,\Sigma,P,S)$ uma GLC. Existe uma GLC $G_L=(V_L,\Sigma,P_L,S_L)$ (e um algoritmo correspondente para produzí-la) que satisfaz:

i) $L(G_L)=L(G)$.

ii) S_L é variável não-recursiva.

ii) $A \rightarrow \varepsilon \in P_L$ sss $\varepsilon \in L(G)$ e $A=S_L$.

Uma gramática G_L satisfazendo estas condições é dita essencialmente não-contrátil.

Exemplo.

Eliminação de Regras em Cadeia

- Elimina regras do tipo $A \rightarrow B$, que não são nada mais do que uma renomeação de variáveis.

Seja $G=(V,\Sigma,P,S)$ uma GLC essencialmente não-contrátil. Existe uma GLC $G_C=(V_C,\Sigma,P_C,S)$ (e um algoritmo correspondente para produzí-la) que satisfaz:

i) $L(G_C)=L(G)$.
ii) G_C não tem regras em cadeia.

Exemplo.

Eliminação de Símbolos Inúteis

- Elimina variáveis que não contribuem para a geração de cadeias da linguagem gerada pela gramática.
- Def.: Seja G uma GLC. Um símbolo $x \in (V \cup \Sigma)$ é <u>útil</u> se existir derivação $S \overset{*}{\Rightarrow} uxv \overset{*}{\Rightarrow} w$, *onde* $u, v \in (V \cup \Sigma)^*$ e $w \in \Sigma^*$. Caso contrário, o símbolo é dito <u>inútil</u>.

Seja $G=(V,\Sigma,P,S)$ uma GLC. Existe uma GLC $G_U=(V_U,\Sigma,P_U,S)$ (e um algoritmo correspondente para produzí-la) que satisfaz:

i) $L(G_U)=L(G)$.
ii) G_U não tem símbolos inúteis

Exemplo.

Formas Normais de Chomsky e Greibach

- *Def.:* Uma GLC G está na forma normal de Chomsky se cada regra estiver em uma das seguintes formas: $A \to BC$, $A \to a$, $A \to \varepsilon$
- *Def.:* Uma GLC G está na forma normal de Greibach se cada regra estiver em uma das seguintes formas: $A \to aA_1A_2\ldots A_n$, $A \to a$, $A \to \varepsilon$

Seja G=(V,Σ,P,S) uma GLC. Existem GLCs G' e G'' (e algoritmos correspondente para produzí-las) que satisfazem:

i) L(G'')=L(G')=L(G).
ii) G' está na forma normal de Chomsky.
iii) G'' está na forma normal de Greibach

Exemplos.

CT200

Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro   carlos@comp.ita.br
Ramal: 5895   Sala: 106

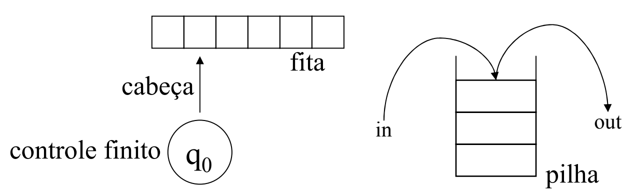
AULA 7

Automata de pilha e LLCs
O lema do bombeamento para LLCs
Propriedades de fechamento para LLCs
Construção de GLCs a partir de APs
Construção de APs a partir de GLCs

Automata de Pilha e Máquinas de Estados

Um autômato de pilha é uma **máquina de estados com uma pilha LIFO.**

- um registrador de estado interno (controle finito) + uma fita segmentada + cabeça de leitura + pilha LIFO

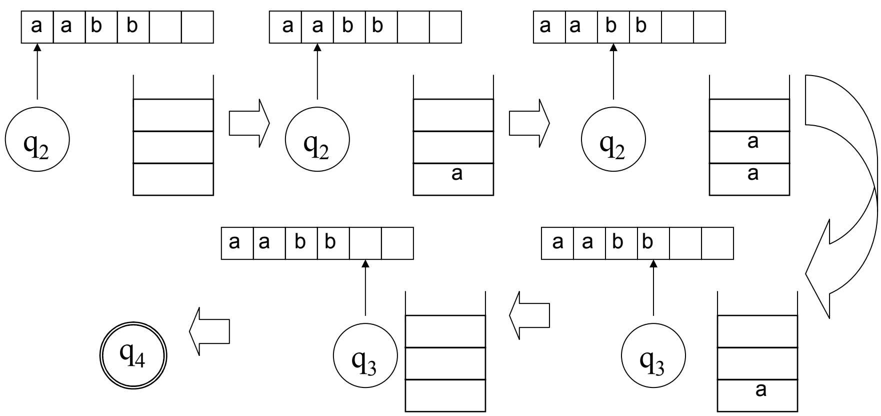


- fita: armazena uma cadeia de Σ (1 símbolo/segmento).
- pilha: armazena uma cadeia de Γ (1 símbolo/segmento)
- cabeça: lê segmento da fita
- registrador: altera estado e conteúdo do topo da pilha de acordo com δ, move fita um segmento para a esquerda.
- uma computação **termina** quando a cadeia da fita “acaba”.

Um Exemplo

AP para reconhecer $\{a^n b^n \mid n \geq 0\}$

- Ao ler a na fita, copio-o para a pilha. Ao ler b na fita, retiro o a do topo da pilha.
- Se a cadeia termina exatamente quando a pilha esvazia, aceito a entrada.



Autômato de Pilha: Definição Formal

A pilha adiciona memória ao autômato finito determinístico usual.
Também chamado **<u>autômato *pushdown*</u>**.

Formalmente: $M=(Q,\Sigma,\Gamma,\delta,q_0,F)$, onde

Q = conjunto de estados
Σ=alfabeto da fita
Γ=alfabeto da pilha (inclui um símbolo especial \$, indicador de pilha vazia)
$q_0 \in Q$ = estado inicial
δ=função de transição $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ em subconjuntos de $Q \times (\Gamma \cup \{\varepsilon\})$
F=conjunto de estados finais

- Uma **computação** em um AP é uma sequência de transições a partir da situação inicial.
- Cadeia é **aceita** pelo AP se a computação termina em um estado $q \in F$.
- Cadeia é **rejeitada** pelo AP se a computação termina em um estado $q \notin F$.
- Situação inicial: estado inicial q_0 e pilha vazia.
- Primeiro passo: transição ε para colocar \$ na pilha, indicando pilha vazia.

A Função de Transição δ

Q X (Σ∪{ε}) X (Γ∪{ε}) → subconjuntos de Q X (Γ∪{ε})

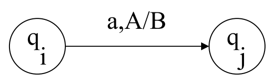
estado

símbolo lido na fita

símbolo lido no topo da pilha e retirado

possíveis configurações (estado, símbolo colocado no topo da pilha) resultantes

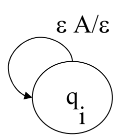
$\delta(q_i, a, A) = \{[q_j, B]\}$:



Estado q_i, símbolo a lido na fita, símbolo A no topo da pilha lido e retirado. Estado resultante q_j, símbolo B colocado no topo da pilha.

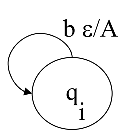
Obs: Poderia ter, por exemplo: $\delta(q_i, a, A) = \{[q_j, B], [q_k, C]\}$ (**Um AP admite transições não-determinísticas!)**

Representação por Diagrama de Estados: Exemplos



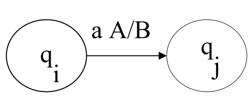
Nenhum símbolo lido na fita, *A* é lido e retirado da pilha, nada é colocado no topo da pilha.

$$\delta(q_i, \varepsilon, A) = \{[q_i, \varepsilon]\}$$



Símbolo *b* lido na fita, nada lido da pilha, *A* é colocado no topo da pilha.

$$\delta(q_i, b, \varepsilon) = \{[q_i, A]\}$$



Símbolo *a* lido na fita, símbolo *A* lido e retirado do topo da pilha, símbolo *B* colocado na pilha.

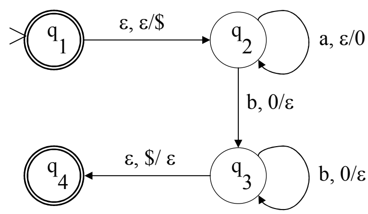
$$\delta(q_i, a, A) = \{[q_j, B]\}$$

Exemplo 1: AP para $L=\{a^n b^n \mid n \geq 0\}$

$Q=\{q_1,q_2,q_3,q_4\}$, $\Sigma=\{a,b\}$, $\Gamma=\{0,\$\}$, $F=\{q_1,q_4\}$

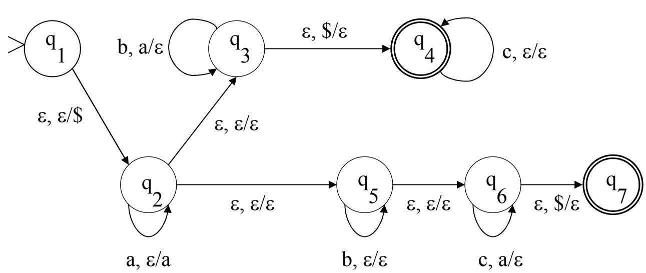
δ:

Fita	0			1			ε		
Pilha	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2,\$)\}$
q_2			$\{(q_2,0)\}$	$\{(q_3,\varepsilon)\}$					
q_3				$\{(q_3,\varepsilon)\}$				$\{(q_3,\varepsilon)\}$	
q_4									



Exemplo 2: AP para $L=\{a^i b^j c^k \mid i,j,k \geq 0 \text{ e } i=j \text{ ou } i=k\}$

$Q=\{q_1,q_2,q_3,q_4,q_5,q_6,q_7\}$, $\Sigma=\{a,b\}$, $\Gamma=\{a,\$\}$, $F=\{q_4,q_7\}$

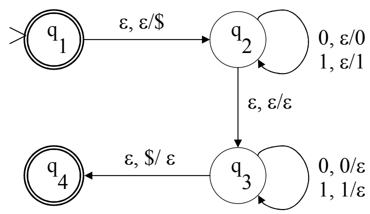


Observe que este é não-determinístico...

Pode ser provado que o não determinismo deste AP é *<u>essencial</u>* *para o reconhecimento de L*

Exemplo 3: AP para L={ww^R | $w \in \{0,1\}^*$ }

Q={q_1,q_2,q_3,q_4}, Σ={0,1}, Γ={0,1,\$}, F={q_1,q_4}



- Copia símbolos lidos para pilha.
- A cada passo, tenta "adivinhar" se o meio da cadeia foi atingido, e retira um símbolo da pilha verificando se é o mesmo lido na fita.
- Para cadeias aceitas, a pilha vai se esvaziar no momento que termina a leitura da cadeia termina.

Configurações em APDs

- **Def.** Uma **configuração instantânea** de um APD é um tripla [q_i, *w*, α], em que q_i é o estado, *w* é a cadeia não processada e α é o conteúdo da pilha (símbolo mais à esquerda no topo).
- **Def.** A função $\vdash_M$ é definida por $[q_i, w, \alpha] \vdash_M [q_j, v, \beta]$ e indica que a configuração $[q_j, v, \beta]$ pode ser obtida a partir da configuração $[q_i, w, \alpha]$ por uma transição do APD *M*.

  A notação $[q_i, w, \alpha] \overset{*}{\vdash}_M [q_j, v, \beta]$ indica que a configuração $[q_j, v, \beta]$ pode ser obtida a partir da configuração $[q_i, w, \alpha]$ por zero ou mais transições do APD *M*.

Configurações em APDs: Exemplos

M: $Q=\{q_0, q_1, q_2, q_3\}$ $\Sigma=\{a,b\}$

$\Gamma=\{A,\$\}$ $F=\{q_3\}$

$\delta(q_0,\varepsilon,\varepsilon)=\{[q_1,\$]\}$ $\delta(q_1,\varepsilon,\varepsilon)=\{[q_2,A]\}$

$\delta(q_2,a,A)=\{[q_2,A]\}$

$\delta(q_2,b,A)=\{[q_3,A]\}$

$\delta(q_3,b,A)=\{[q_3,A]\}$

$[q_0,aabb,\varepsilon] \vdash [q_1, aabb, \$] \vdash [q_2, aabb, A\$] \vdash [q_2, abb, A\$] \vdash [q_2, bb, A\$] \vdash [q_3, b, A\$] \vdash [q_3, \varepsilon, A\$]$

Computação para *aabb* terminou em estado final → cadeia **aceita** pelo AP.

M: $Q=\{q_0, q_1, q_2, q_3, q_4\}$ $\Sigma=\{a,b\}$

$\Gamma=\{A,\$\}$ $F=\{q_4\}$

$\delta(q_0,\varepsilon,\varepsilon)=\{[q_1,\$]\}$ $\delta(q_1,\varepsilon,\varepsilon)=\{[q_2,A]\}$

$\delta(q_2,a,A)=\{[q_3,A]\}$ $\delta(q_3,\varepsilon,\varepsilon)=\{[q_3,A]\}$

$\delta(q_2,b,A)=\{[q_3,A]\}$ $\delta(q_3,b,A)=\{[q_4,\varepsilon]\}$

$[q_0,abb,\varepsilon] \vdash [q_1,abb,\$] \vdash [q_2,abb,A\$] \vdash [q_3,bb,A\$] \vdash [q_3,bb,AA\$] \vdash [q_4,b,A\$]$

Computação para *abbb* não terminou → cadeia não **aceita** pelo AP.

APDs e Linguagens Livres de Contexto

Teorema (*Chomsky, 1962*): L é uma linguagem livre de contexto sss L é aceita por algum APD *M* que aceita por pilha vazia (ou por estado final).

Análogo ao teorema de Kleene para AFs e linguagens regulares!

Prova: Sipser, págs. 107/114

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 $\text{length}(z) \geq n$, podemos escrever z=uvwxy de modo que:

- $\text{length}(vx) \geq 1$
- $\text{length}(vwx) \leq n$
- $u\, v^i\, w\, x^i\, y \in L$, para todo $i \geq 0$

Corresponde ao bombeamento de <u>duas</u> subcadeias separadas por w.

Como no caso de linguagens regulares, posso usar o *Pumping Lemma* para mostrar que uma linguagem <u>não é</u> livre de contexto. A idéia 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 $L=\{a^i b^i c^i \mid i \geq 1\}$ 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 $\text{length}(z) \geq n$, então posso escrever:

$z = uvwxy$, com $\text{length}(vx) \geq 1$, $\text{length}(vwx) \leq n$, e $uv^i wx^i y \in L$

i) vx não pode conter a's <u>e</u> c's, pois teria que colocar n b's entre eles (impossível, pois length $(vwx) \leq n$).

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 $\text{length}(vx) \geq 1$). Assim, não poderia escrever $uwy \in L$ como $a^k b^k c^k$.

iii) vx não pode conter apenas a's <u>e</u> 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 $\text{length}(vx) \geq 1$). Assim, não poderia escrever $uwy \in L$ 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, $z=uvwxy \in L$ (por hipótese), mas v e x não podem ser formados por nenhuma combinação de símbolos do alfabeto da linguagem L: **contradição $\Rightarrow$ L não é livre de contexto.**

Exemplo 2

Prove que $L=\{a^i b^k c^i d^k \mid i, k \geq 1\}$ não é livre de contexto.

Suponha que L seja L.C. Então, vale o *pumping lemma* e existe um n correspondente. Seja $z = a^n b^n c^n d^n \in L$. 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 mais do que dois símbolos diferentes e consecutivos, pois senão teríamos length(vwx)≤n.

ii) Se vx tivesse apenas a's, então uwy teria menos a's do que c's e não poderíamos escrever uwy ∈ L como $a^k b^{k'} c^k d^{k'}$.

iii) Se vx tivesse apenas a's e b's, então ainda assim uwy teria menos a's do que c's e não poderíamos escrever uwy ∈ L como $a^k b^{k'} c^k d^{k'}$.

Por raciocínio análogo a ii), v e x não podem ter só b's, c's ou d's.

Por raciocínio análogo a iii), vx não pode conter apenas b's e c's ou c's e d'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 $L=\{w \in \mathbf{a}^* \mid \text{length}(w) \text{ é primo}\}$ 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:

$\text{length}(uwy) + \text{length}(v^i x^i) = \text{length}(uwy) + i\,\text{length}(vx) = m + i\,(k-m)$

Em particular, $\text{length}(uv^{k+1} wx^{k+1} y) = m + (k+1)(k-m) = k(k-m+1)$, que é divisível por k! **Contradição...**

LLCs : Propriedades de Fechamento

Teorema: Linguagens Livres de Contexto são fechadas sob as operações de:

União / Concatenação / Fechamento de Kleene / Homomorfismo / Homomorfismo inverso

Teorema: Seja L_R uma linguagem regular e L_L uma linguagem livre de contexto. Então $L = L_R \cap L_L$ é livre de contexto.

Obs: Linguagens Livres de Contexto **não** são fechadas sob as operações de:

Intersecção / Complementação

CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro
carlos@comp.ita.br
Ramal: 5895   Sala: 106

AULA 8

Análise Sintática (*Parsing)*
GLCs ambíguas
Grafos de GLCs
Estratégias para *parsing*
Exemplos de *parsers*

Análise Sintática (*Parsing)*

Derivações em uma GLC: mecanismo para gerar cadeias de uma linguagem livre de contexto. Ok, mas...

Como determinar se uma dada cadeia pode ser gerada por uma dada gramática?

Em outras palavras:

Como determinar se uma dada cadeia é sintaticamente correta (ou seja, se está de acordo com a sintaxe definida pela gramática)?

Nosso objetivo: determinar um algoritmo para produzir derivações das cadeias se uma linguagem de uma gramática dada. Se a cadeia não estiver na linguagem, o algoritmo deve descobrir que não existe derivação capaz de produzí-la.

Este algoritmo é conhecido como **analisador sintático** ou ***parser.***

Parsers são variações de algoritmos de percurso em grafos.

Derivações à Direita e à Esquerda (revisão)

Derivação à Direita: cada passo de derivação aplicado à variável mais à direita.

Derivação à Esquerda: cada passo de derivação aplicado à variável mais à esquerda.

Exemplo:

G = ({S,A}, {a,b}, P,S}

P: S→aAS |a | b

A→SbA |SS| ba

À direita: S =>aAS=>aAb=> aSbAb=> aSbbab=> aabbab

À esquerda: S =>aAS=>aSbAS=>aabAS=> aabbaS=>aabbab

Um Teorema Óbvio

Seja G=(V, Σ, P,S) uma gramática livre de contexto.Uma sentença *w* está em L(G) sss existe uma derivação à esquerda de *w* a partir de S.

a) Para qualquer sentença *w*, existe uma derivação à esquerda. **Óbvio.**

b) Se existe derivação à esquerda de *w*, então *w* está em L(G). **Óbvio ululante**

Exemplo:

G: S→ AB

A→ aA | ε

B→ bB | ε

Gera L(G)=a^*b^*. Para qualquer sentença de l(G), existe uma derivação à esquerda.

Por outro lado, **não há derivação à esquerda** para a forma sentencial A... Mas **existe derivação à direita!**

Existe um teorema idêntico para derivações à direita. Dada a dualidade, **vamos a partir de agora nos** concentrar apenas em derivações à esquerda.

Ambigüidade em GLCs

Restringir a atenção a derivações à esquerda é suficiente para estabelecer uma <u>derivação canônica para qualquer cadeia de linguagem</u> de uma gramática?

Infelizmente, não...

Exemplo

G:

$S \rightarrow AA$

$A \rightarrow AAA \mid bA \mid Ab \mid a$

$$\mathbf{S} \underset{L}{\Rightarrow} \mathbf{AA} \underset{L}{\Rightarrow} \mathbf{aA} \underset{L}{\Rightarrow} aAAA \underset{L}{\Rightarrow} abAAA \underset{L}{\Rightarrow} abaAA \underset{L}{\Rightarrow} ababAA \underset{L}{\Rightarrow} ababaA \underset{L}{\Rightarrow} ababaa$$

$$\mathbf{S} \underset{L}{\Rightarrow} \mathbf{AA} \underset{L}{\Rightarrow} \mathbf{AAAA} \underset{L}{\Rightarrow} aAAA \underset{L}{\Rightarrow} abAAA \underset{L}{\Rightarrow} abaAA \underset{L}{\Rightarrow} ababAA \underset{L}{\Rightarrow} ababaA \underset{L}{\Rightarrow} ababaa$$

Ao menos duas derivações à esquerda para uma dada cadeia: **gramática ambígua.** É o mesmo conceito de ambigüidade em linguagens naturais.

Exemplo: *João ganhou um livro de Jorge Amado.*

Ambigüidade em GLCs

Def: Uma GLC é ambígua se existir alguma cadeia *w* em L (G) derivada por duas derivações à esquerda distintas.

Exemplo

G: $S \rightarrow aS \mid Sa \mid a$

Tem $L(G) = a^+$ e duas derivações à esquerda distintas para aa (quais?).

G: $S \rightarrow aS \mid a$

Tem L(G) = a^+ e é não-ambígua.

Observe portanto que a ambigüidade é propriedade da gramática e não da linguagem.

Pergunta: Dada uma linguagem livre de contexto, é sempre possível achar uma GLC não-ambígua que a gere?

Resposta: Não!

Grafos de GLCs e *Parsing*

Seja G = (V,Σ,P,S) uma GLC. O grafo da gramática G, g(G), é o grafo direcionado (N,P,A) onde:

- **N = {$w \in (V \cup \Sigma)^* \mid S \overset{*}{\underset{L}{\Rightarrow}} w$}**
- **A = {$[v, w, r] \in N \times N \times P \mid v \underset{L}{\Rightarrow} w$ por aplicação da regra r}**

Cada caminho de S a *w* em g(G) representa uma derivação à esquerda para *w.*

O rótulo no arco de v a *w* indica a regra utilizada para obter *w* a partir de v.

Decidir se uma dada cadeia pode ser gerada por G (ou seja, se está de acordo com a sintaxe da linguagem de G) equivale a <u>achar um caminho de S a *w*</u> no grafo g (G).

Exemplos: págs.95 e 96, Sudkamp

Idéia para a análise Sintática: Expansão do grafo da GLC.

Estratégias:

- ***Top-down:*** começo com nó S e tento achar caminho para cadeia w.
- ***Bottom-up:*** começo com cadeia w e tento chegar no nó inicial S.

Processo não-determinístico: para construir uma derivação, em geral existem várias possíveis regras a serem aplicadas à forma sentencial. Nunca se sabe qual a "melhor"regra a ser aplicada

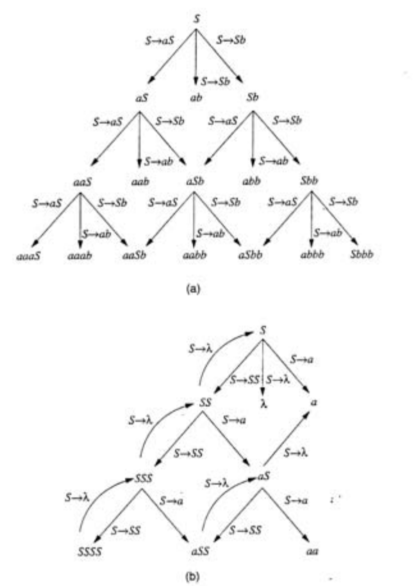
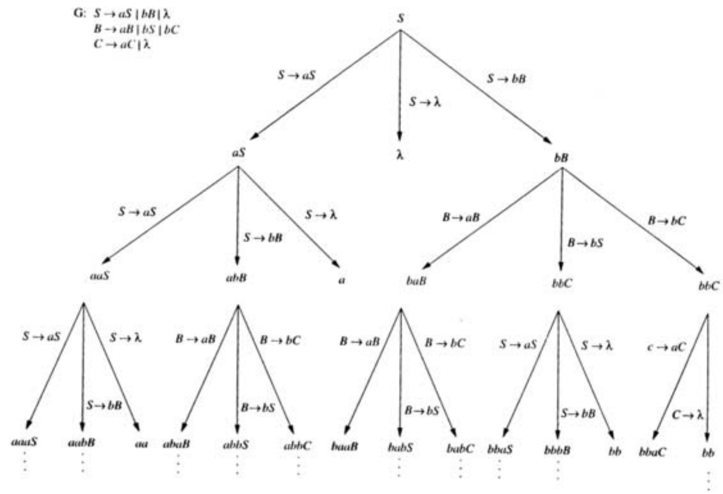


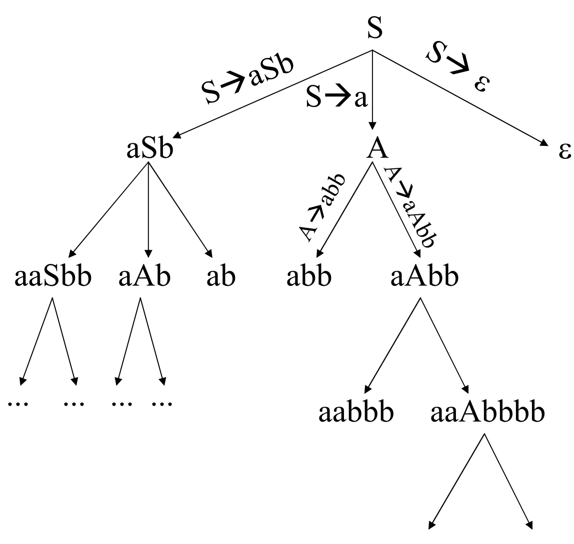
FIGURE 4.2 Graphs (a) $S \to aS \mid Sb \mid ab$. (b) $S \to SS \mid a \mid \lambda$.



Uma gramática não-ambígua:

S → aSb | A | ε

A → aAbb | abb



Um Analisador Sintático *Breadth-First, Top-Down*

Top-down: derivações à esquerda a partir de S.

Prefixo da foram sentencial *uVw:u.*

*Breadth-first:*busca completa (garante *parsing* caso a cadeia seja da gramática).

Algoritmo

- Inicializo uma fila com S.
- Loop: **Remove nó do início da fila. Tem prefixo consistente com cadeia a ser analisada?**
  - Sim: Expando o nó do inicio da fila. Cada filho é uma derivação à esquerda possível, rotulado pela cadeia resultante da derivação. Coloco os nós correspondentes no **final da fila.**
  - Não: Apenas removo o nó do início da fila.

  Até que o nó do início da fila seja igual a cadeia analisada ou fila fique vazia.

Exemplo: parsing de (b+b) para gramática

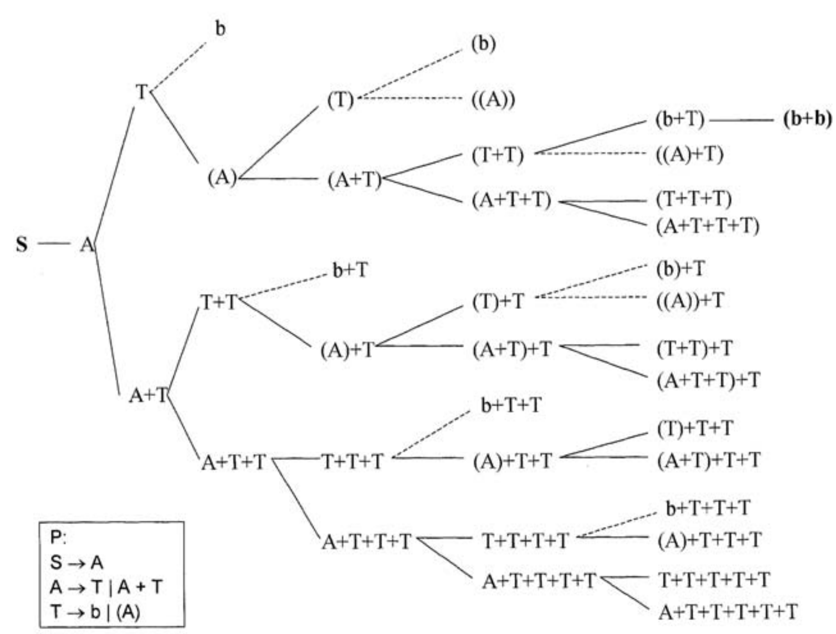
G: V= {S, A, T}

Σ={ b,+, (,)}

P: S → A

A → T | A + T

T → b | (A)



Um Analisador Sintático *Depth-first, Top-down*

Top-down: derivações à esquerda a partir de S.

Algoritmo

- Inicializo uma pilha com S.
- Loop:

  - Nó do topo da pilha tem prefixo consistente com cadeia a ser analisada?
    - Sim: Removo o espaço o nó do topo da pilha. Cada filho é uma derivação à esquerda possível, rotulado pela cadeia resultante da derivação. Coloco os nós correspondentes no **topo da pilha**.
    - Não: Removo o nó do topo da pilha.

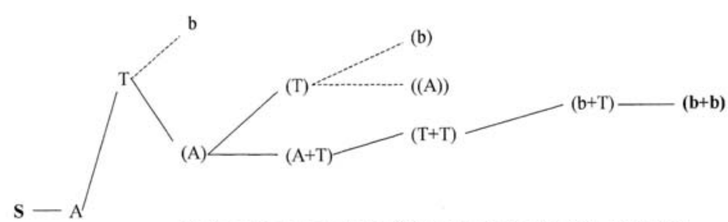
  Até que nó do topo da pilha seja igual a cadeia analisada ou pilha fique vazia.

Exemplo: parsing de (b+b) para a gramática

G: V = {S, A, T}

Σ = {b, +, (,)}

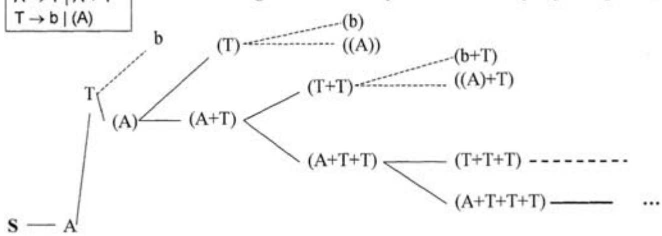
P:
S → A
A → T | A + T
T → b | (A)



Vantagem: muito mais econômico do que *breadth-first* → mantém menos nós na memória.

P:
S → A
A → T | A + T
T → b | (A)

Desvantagem: risco de loops infinitos. Exemplo: *parsing* de (b) + b



Parsing *Bottom-Up*

Idéia: realizar busca no grafo a partir da cadeia a ser analisada (geração de caminhos na direção da raiz). Como as únicas derivações consideradas são as que podem gerar a cadeia, o tamanho da árvore expandida tende a ser menor.

O processo de geração das formas sentenciais em níveis cada vez mais altos da árvore é conhecido como <u>redução</u>.

Exemplo: Redução de (b) + b para gramática

G: V = { S, A, T}

Σ = {b, +, (,)}

P: S → A

A → T | A + T

T → b | (A)

Redução	Regra
(b) + b	
(T)+ b	T→b
(A)+ b	A→T
T+ b	T→(A)
A+ b	A→T
A+ T	T→b
A	A→A+T
S	S→A

Parsing *Bottom-Up:* Geração Automática das Reduções

- **Algoritmo:**

1. Escrevo $w=uv$ (na primeira interação, $u=\varepsilon$, $v=w$).
   Para cada regra:
2. Comparo lado direito das regras com sufixos de u:
   $u=u_1 q$ e $A \rightarrow q \in P$? Reduzo w a u_1Av
3. Volto a 1, fazendo w=u'v' tal que u'é u concatenado ao primeiro elemento de v, e v'é v sem o seu primeiro elemento (processo de ***shift)***

Exemplo: Redução de (A+T) para gramática

G: V = {S,A,T}

Σ = {b,+, (,)}

P: S → A

A → T | A + T

T → b | (A)

	u	**v**	**regra**	**redução**
	ε	(A+T)		
Shift	(	A+T)		
Shift	(A	+T)	S→A	(S+T)
Shift	(A+	T)		
Shift	(A+T	)	A→A+T	(A)
			A→T	(A+A)
shift	(A+T)	ε		

Um Analisador Sintático *Breadth-First, Bottom-Up*

Bottom-up: derivações à **direita** a partir de S, pois as reduções são feitas **à esquerda** a partir da cadeia terminal..

*Breadth- first:*busca completa (garante *parsing* caso a cadeia seja da gramática).

Algoritmo

- Inicializo uma fila com cadeia terminal.
- Loop:
  - **Removo nó do início da fila. Existe redução de uwv deste nó para uAv com v formado só por símbolos terminais?**
    - Sim: Expando o nó do inicio da fila. Cada filho é uma redução à esquerda possível, rotulando pela cadeia resultante da redução. Coloco os nós correspondentes no **final da fila.**
    - Não: Apenas removo o nó.

  Até que o nó do início da fila seja igual a S ou fila fique fazia.

<u>Exemplo</u>: parsing de (b+b) para gramática

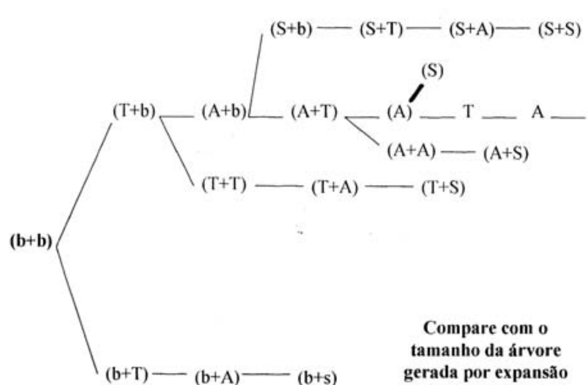
G: V= {S, A, T}

Σ={ b,+, (,)}

P: S→A

A→T | A + T

T →b | (A)



Compare com o tamanho da árvore gerada por expansão *top-down*...

P:
S → A
A → T | A + T
T → b | (A)

Variações

- Técnicas de busca:Aumento da eficiência computacional
  - aprofundamento iterativo (depth-first com aumento gradual do nível de profundidade).
  - busca bidirecional (top-down e bottom-up simultâneos)
  - etc,etc,...
- Heurísticas: Diminuem o espaço de busca.
  - Análise a priori da cadeia (uso de contadores, etc.): por exemplo, formas sentenciais com mais símbolos terminais do que a sentença analisada certamente não levam a *parsing.*
- Normalização de GLCs:Normalização Greibach garante completeza da análise sintática *depth-first top-down..*
- Particularização do processo de *parsing* para classes particulares de GLCs: por exemplo, a classe LL (k) permite *parsing top-down* **determinístico,** desde que se utilize o conceito de *look-ahead.*

  Exemplo: *uAv* obtido durante o *parsing de p=uaw.* Ao invés de considerar apenas o prefixo *u,* olho mais adiante (*look-ahead)* e analizo regras *A.* Aquelas cuja produção não começa com a podem ser consideradas.

  O determinismo surge naturalmente: a classe LL (1) obriga uma única regra aplicável se for usado look-ahead de 1 símbolo, a classe LL(2) obriga uma única regra se for usado look-ahead de 2 símbolos, etc.

CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro
carlos@comp.ita.br
Ramal: 5895   Sala: 106

Aula 9

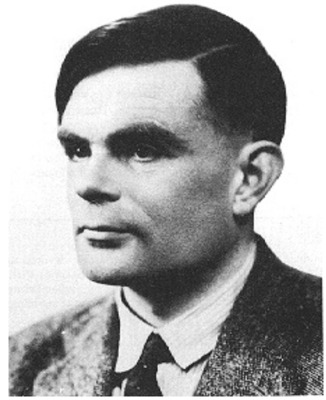
Máquinas de Turing
MTs e Funções Inteiras
Linguagens recursivas
Artifícios para o projeto de MTs
Variações de MTs
Máquinas equivalentes a MTs

A Máquina de Turing

- Proposta pelo inglês Alan M. Turing, em 1936.
- Baseada em sequências de operações elementares.
- Não apresenta limitações de memória ou tempo.
- Várias possíveis realizações.
- Realização computacional de um **procedimento**.
- É a máquina computacional abstrata <u>mais poderosa</u>. Isto <u>não significa</u> ser capaz de computar qualquer função...

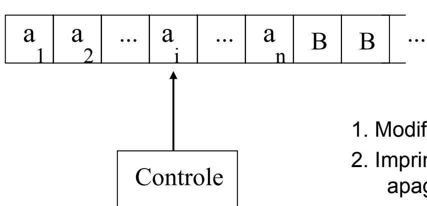
Alan Mathison Turing
(23/06/1912 – 07/06/1954)

Matemático, lógico, criptógrafo e herói de guerra britânico. Considerado o pai da ciência da computação. Fez previsões acerca da Inteligência Artificial e propôs o Teste de Turing, contribuindo para o debate sobre a consciência das máquinas e suas capacidades de pensar. Formalizou o conceito de algoritmo e computação com a Máquina de Turing, gerando sua versão da Tese de Church-Turing. Responsável pela quebra do código alemão *Enigma* durante a II Guerra Mundial. Depois da guerra, projetou um pioneiro computador digital programável eletronicamente. Foi processado e condenado por ser homossexual (em 1952). Morreu envenenado (provável suicídio). O Turing Award foi criado em sua homenagem.



Um Modelo da Máquina de Turing

- Fita dividida em células, finita à esquerda, infinita à direita.
- Cabeça de leitura + controle da cabeça (registrador de estados).
- Cadeias escritas a partir da célula mais à esquerda.



1. Modifica-se o estado do controle
2. Imprime-se um símbolo na célula lida, apagando-se o que estava.
3. Move-se a cabeça para direita ou esquerda.

O poder computacional adicional da MT em relação a um 2-AF está em sua capacidade de <u>escrever</u> na fita.

Máquina de Turing: Definições

- Uma Máquina de Turing (modelo básico) é denotada $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$ onde:
  - Q é o conjunto finito de **estados**;
  - Γ é o alfabeto finito de **símbolos da fita**;
  - $B \in \Gamma$ é o **símbolo "em branco"** (*blank*);
  - $\Sigma \subset \Gamma - \{B\}$ é o conjunto de **símbolos de entrada**;
  - δ é a **função de transição** de $Q\times\Gamma$ em $Q\times\Gamma\times\{L,R\}$ (pode ser uma função não definida para alguns argumentos);
  - $q_0 \in Q$ é o **estado inicial**;
  - $F \subseteq Q$ é o **conjunto de estados finais**.

- Uma **configuração instantânea** de uma MT é um tripla $\alpha_1 q \alpha_2$ em que q é o estado e $\alpha_1\alpha_2$ é a string de Γ^* correspondente ao conteúdo da fita até o símbolo diferente de B mais à direita ou até o símbolo à esquerda da cabeça de leitura (o que estiver mais à direita). A cabeça está sobre o símbolo mais à esquerda de α_2 (se $\alpha_2 = \varepsilon$ a cabeça está sobre o símbolo B).

Máquina de Turing: Mais Definições

- Seja $X_1X_2 \ldots X_{i-1}\, q\, X_i \ldots X_n$ uma configuração instantânea. Assume-se que se $i=n+1$, entao $X_i = B$. Então:
  - Caso 1: **movimento para a esquerda** $\delta(q, X_i)=(p,Y,L)$.
    - Se i=1, não existe uma próxima C.I.
    - Se $i>1$, $X_1X_2 \ldots X_{i-1}\, q\, X_i \ldots X_n \vdash X_1X_2 \ldots X_{i-2}\, p\, X_{i-1}\, Y \ldots X_n$, e se qualquer sufixo de $X_{i-1}\, Y \ldots X_n$ é "em branco", este é apagado.
  - Caso 2: **movimento para a direita** $\delta(q, X_i)=(p,R,L)$.
    - $X_1X_2 \ldots X_{i-1}\, q\, X_i \ldots X_n \vdash X_1X_2 \ldots X_{i-1}\, Y\, p\, X_{i+1} \ldots X_n$.
- A notação $\overset{*}{\vdash}$ indica que uma configuração instantânea pode ser obtida a partir de outra por zero ou mais movimentos da máquina M.
- **Def.** A **linguagem L(M)** aceita por uma MT M é o subconjunto de Σ^* que leva M a um estado final a partir da configuração inicial definida por:
  - Cadeia posicionada a partir da posição mais à esquerda da fita;
  - Cabeça de leitura posicionada sobre célula mais à esquerda da fita.
  - M no estado q_0 ;

  Formalmente: $L(M)=\{w \mid w \in\Sigma^* \text{ e } q_0w \overset{*}{\vdash} \alpha_1 p \alpha_2 \text{ para algum } p\in F, \alpha_1,\alpha_2 \in \Gamma^*\}$

MTs como Decisores

- Ao iniciar uma MT com uma entrada, pode-se:
  - Aceitar a entrada;
  - Rejeitar a entrada;
  - Não parar (MT em *loop*).
- Uma MT pode não aceitar uma cadeia de entrada: (i) ao entrar em $q \notin F$ e parar (ou seja, **rejeitar** a cadeia) ou (ii) ao entrar em *loop* (não parar).
  - Pode ser muito difícil distinguir uma MT que entrou em *loop* de outra que está demorando para computar.
- Uma MT **decide uma linguagem** quando pára para todas as entradas:
  - se $w \notin L(M)$, MT pára em $q \notin F$;
  - se $w \in L(M)$, MT pára em $q \in F$.
- A classe das linguagens aceitas por MTs decisoras é a classe das **linguagens recursivas**.

Exemplo: MT para reconhecer $L=\{0^n1^n \mid n\geq 1\}$

- *Inicialmente na fita: 0^n1^n*
- *Operação: substitui 0 mais à esquerda por X, a seguir move-se para direita e substitui 1 mais à esquerda por Y. Move-se então para a esquerda, até se encontrar o X mais à direita. Move-se então uma célula para a direita (onde deve estar agora o 0 mais à esquerda) e repete-se o ciclo.*
- *Aceitação: a) Se ao se procurar um 1 encontra-se um B, realiza-se uma parada sem aceitação. b) Se, após mudar um 1 para Y, M não encontrar mais 0's, tenta-se achar 1's. Se não se encontra nenhum, ocorre aceitação.*

$Q=\{q_0,q_1,q_2,q_3,q_4\}$ $\Sigma=\{0,1\}$ $\Gamma=\{0,1,X,Y,B\}$ $F=\{q_4\}$

- q_0: Estado inicial; revisitado imediatamente antes da substituição do 0 mais à esquerda por um X. Ao ler um Y, muda para estado q_3.
- q_1: Usado para se buscar para a direita, ignorando-se 0's e Y's até se encontrar o 1 mais à esquerda. Ao se encontrar o 1, muda-se para Y e entra-se no estado q_2. Caso se encontre um B ou X antes do 1, a string é rejeitada.
- q_2: Usado para se fazer uma busca para a esquerda por um X, muda então para estado q_0 e move cabeça para direita, até se encontrar o 0 mais à esquerda.
- q_3: Varre sequencia de Y's, e checa se não existem 1's remanescentes. Se os Y's são seguidos por um B, entra-se estado q_4 e aceita, caso contrário rejeita.

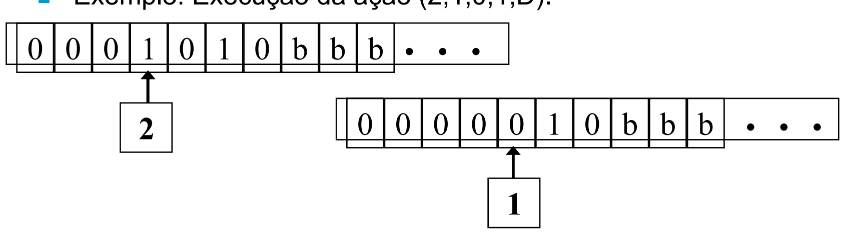
MT para reconhecer $L=\{0^n1^n \mid n\geq 1\}$

	Símbolo				
Estado	**0**	**1**	**X**	**Y**	**B**
q_0	(q_1,X,R)	-	-	(q_3,Y,R)	-
q_1	$(q_1,0,R)$	(q_2,Y,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	-	-	-	-	-

Exemplos de computação

Forma alternativa de descrever uma MT

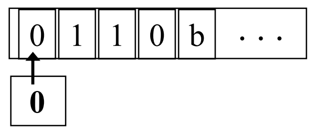
- Uma MT também pode ser descrita por um conjunto de ações.
- **Ações**: **(s, i, i', s', d)** sendo:
  - s: estado atual da MT ($s \in S$)
  - i: símbolo que está sendo lido na fita ($i \in \Gamma$)
  - i': símbolo que é impresso na fita ($i' \in \Gamma$)
  - s': novo estado da MT ($s' \in S$)
  - $d \in \{D,E\}$.
- Exemplo: Execução da ação (2,1,0,1,D).



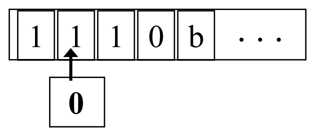
Ações da Máquina de Turing

- Ex: uma MT é definida pelo conjunto de quíntuplas: (0,0,1,0,D), (0,1,0,0,D), (0,b,1,1,E), (1,0,0,1,D), (1,1,0,1,D), (1,b,b,2,D). O estado de aceitação é 2. Verificar sua computação:

Início:

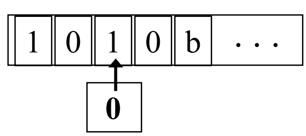


(0,0,1,0,D)

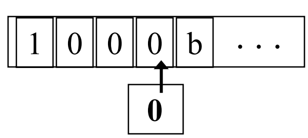


(0,1,0,0,D)

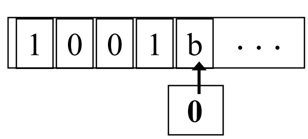
Ações da Máquina de Turing



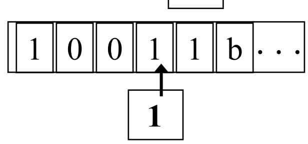
(0,1,0,0,D)



(0,0,1,0,D)

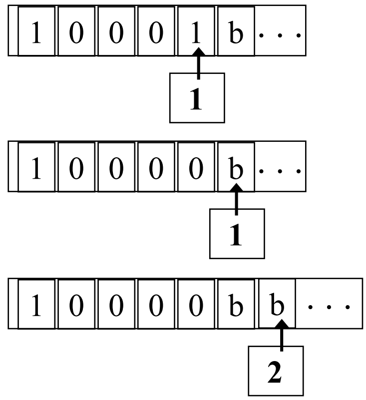


(0,b,1,1,E)



(1,1,0,1,D)

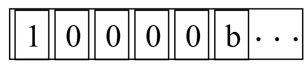
Ações da Máquina de Turing



(1,1,0,1,D)

(1,b,b,2,D)

Como a máquina pára no estado 2 (estado de aceitação), a fita terá:



Ações da Máquina de Turing

- **Exercício:** Considere a seguinte MT: (0,0,0,1,D), (0,1,0,0,D), (0,b,b,0,D), (1,0,1,0,D), (1,1,1,0,E). Qual o comportamento da MT quando iniciada com:

(a)

1	0	b	. . .

(b)

0	1	b	. . .

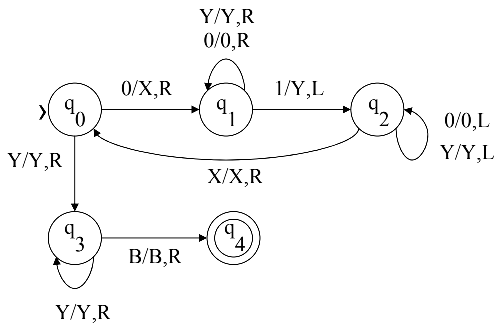
(c)

0	0	b	. . .

Diagramas de Transição para MTs

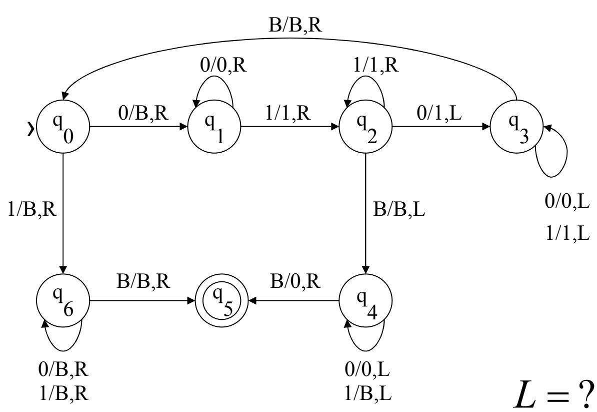
- Máquina de Turing $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$:
  - Cada nó = um estado
  - Arco X/Y R ligando q a p $\leftrightarrow$ $\delta(q,X)=(p,Y,R)$
  - Arco X/Y L ligando q a p $\leftrightarrow$ $\delta(q,X)=(p,Y,L)$
  - Estado inicial e estados de aceitação: como em AFDs.

Exemplo 1:



$$L = 0^n 1^n$$

Exemplo 2



$$L = ?$$

Computação de Funções Inteiras

Uma MT pode também ser vista como um computador para funções inteiras.

Representação:

- Inteiro $i \geq 0$: string 0^i
- Separação entre inteiros como argumentos de uma função:
  $(i_1,i_2,\ldots,i_k) \rightarrow 0^{i_1},1, 0^{i_2},1,\ldots,1, 0^{i_k}$
- Parada em uma fita com conteúdo 0^m: $f(i_1,i_2,\ldots,i_k) = m$, onde f e a função calculada (computada) pela MT.
- **OBS1:** Uma mesma MT pode computar uma função de um argumento, uma outra função de dois argumentos, etc.
- **OBS2:** Domínio da função pode ser um **subconjunto** de Z^k.

Computação de Funções Inteiras: Exemplo

Uma MT para realizar subtração de inteiros de acordo com a seguinte definição: m-n=m-n se m≥n, e m-n=0 se m<n.

Considere a MT $M=(\{q_0,q_1,q_2,q_3,q_4,q_5,q_6\},\{0,1\},\{0,1,B\},\delta,q_0,B,\emptyset)$, definida da seguinte maneira:

- Inicialmente na fita: 0^m10^n
- Computação: parada com 0^{m-n} na fita.
- Operação: substitui 0 inicial por B, a seguir inicia busca de 1 seguido de 0, substitui o 0 por 1. M entao volta para esquerda até encontrar um B, e recomeça ciclo.
- Fim da operação:
  - Ao procurar um 0 quando avançando a cabeca, M encontra um B. Nesse caso, os n 0's em 0^m10^n foram mudados para 1, e n+1 dos m 0's foram mudados para B. M entao substitui os n+1 1's por um 0 e n B's, deixando m-n 0's na fita.
  - Ao iniciar um ciclo, M não consegue encontrar um 0 a ser transformado em B, pois os primeiros m 0's já foram modificados. Neste caso, $n \geq m$ e portanto m-n = 0. M então substitui todos os 0's e 1's restantes por B's.

Um exemplo de computação.

Artifícios para o Projeto de MTs

Não existem "receitas" simples para o projeto de uma MT. Alguns artifícios, porém, podem ser úteis:

- Armazenamento da informação nos estados.
- Utilização de múltiplas fitas.
- Checagem de símbolos.
- Deslocamento de símbolos.
- Programação estruturada (utilização de subrotinas).

Armazenamento da Informação

- Estado pode ser escrito como um par de elementos:
  - Primeiro elemento: controle
  - Segundo elemento: informação

Não há modificação sobre o modelo básico: um par de elementos é um "estado"...

Exemplo: Uma MT que aceita a linguagem regular ($01^* + 10^*$):

- Lê o primeiro símbolo da fita (0 ou 1).
- Registra o símbolo como parte do "estado"
- Verifica se o símbolo não aparece mais na fita.

A idéia é:

- Primeiro componente do "estado" controla avanço da cabeça.
- Segundo componente do "estado" lembra-se do primeiro símbolo lido.

M=(Q, {0,1}, {0,1,B}, δ, [q_0, B], B, F}
Q={[q_0, 0], [q_0, 1], [q_0, B], [q_1, 0], [q_1, 1], [q_1, B]}

1) A partir do estado inicial ([q_0, B]): avança para estado seguinte e armazena símbolo lido como segundo componente do estado:

$\delta([q_0,B],0) = ([q_1,0],0,R)$ $\delta([q_0,B],1) = ([q_1,1],1,R)$

2) A partir do estado ([q_1, X]): se o símbolo armazenado no estado for diferente do símbolo da fita, simplesmente avança:

$\delta([q_1,0],1) = ([q_1,0],1,R)$ $\delta([q_1,1],0) = ([q_1,1],0,R)$

3) M entra no estado final ([q_1, B]) se chegar a um B sem ter antes encontrado um símbolo igual àquele armazenado no estado. Escreve o símbolo repetido no lugar do B, para referência, e volta a cabeça:

$\delta([q_1,0],B) = ([q_1,B],0,L)$ $\delta([q_1,1],B) = ([q_1,B],1,L)$

4) Parada sem aceitação: [q_1,0] e símbolo 0 ou [q_1,1] e símbolo 1 (transições não definidas).

Armazenamento de Informação em Múltiplas Fitas

- Estado é escrito como uma n-upla de elementos:
  - Primeiro elemento: controle
  - Elementos restantes: informação (várias)

c	1	0	1	1	1	1	$	B
B	B	B	B	1	0	1	B	B
B	1	0	0	1	0	1	B	B

…

Controle

Exemplo: Verificador de Números Primos (> 2)

1. Número (base 2) escrito na primeira fita (cercado por c e $).

c	1	0	1	1	1	1	$	B	
B	B	B	B	B	B	B	B	B	···
B	B	B	B	B	B	B	B	B	

Controle

2. Copia conteúdo da primeira fita para terceira fita, escreve dois em binário na segunda:

c	1	0	1	1	1	1	$	B	
c	1	0	$	B	B	B	B	B	···
c	1	0	1	1	1	1	$	B	

Controle

3. Subtrai conteúdo da segunda do conteúdo da terceira, tantas vezes quantas forem possíveis. O que sobra na terceira fita é portanto o resto da divisão.

c	1	0	1	1	1	1	$	B	
c	1	0	$	B	B	B	B	B	···
c	1	0	1	1	0	1	$	B	

c	1	0	1	1	1	1	$	B	
c	1	0	$	B	B	B	B	B	…
c	1	0	1	0	1	1	$	B	

⋮

c	1	0	1	1	1	1	$	B	
c	1	0	1	1	1	1	$	B	…
c	1	$	B	B	B	B	B	B	

Controle

Sobrou 0 na terceira? Número não primo, parada da MT (s/ aceitação)

Senão...

4. Copio conteúdo da primeira fita para terceira fita, incremento o conteúdo da segunda de uma unidade:

c	1	0	1	1	1	1	$	B	
c	1	1	$	B	B	B	B	B	…
c	1	0	1	1	1	1	$	B	

Controle

5. **Conteúdo da segunda fita igual ao da primeira?** Número primo, pára com aceitação.
Senão volto ao passo 3.

Checagem de Símbolos

- Útil para visualizar reconhecimento de linguagens definidas por símbolos repetidos.
- Útil para comparar comprimentos de substrings em linguagens do tipo $a^i b^j c^k$.

"Truque": utilizar uma fita extra que marca se o símbolo correspondente da outra fita foi (✓) ou não (B) considerado.

Exemplo

- Uma MT que reconhece $\{wcw \mid w \in (a+b)^*\}$

Idéia:

1. Marca o primeiro símbolo e registra-o no "estado", avança sobre todos os símbolos não-marcados até achar o c.

1

B	B	B	B	B	B	B	B	B	
a	b	b	c	a	b	b	B	B	…

2

✓	B	B	B	B	B	B	B	B	
a	b	b	c	a	b	b	B	B	…

2. Após achar c, continua avançando, agora sobre símbolos marcados. Se o primeiro após c for igual ao registrado no estado, marca-o e volta para primeira posição após o símbolo marcado. Recomeça o processo a partir de 1. Caso contrário, pára sem aceitação.

3

✓	B	B	B	B	B	B	B	B	
a	b	b	c	a	b	b	B	B	…

4

✓	B	B	B	✓	B	B	B	B	
a	b	b	c	a	b	b	B	B	…

3. Quando terminar (ou seja, após marcar o último antes de c), tem que conferir se não sobrou nenhum não-marcado após o c.

5

✓	✓	✓	B	✓	✓	✓	B	B	
a	b	b	c	a	b	b	B	B	…

6

✓	✓	✓	B	✓	✓	✓	B	B	
a	b	b	c	a	b	b	B	B	…

⋮

10

✓	✓	✓	B	✓	✓	✓	B	B	
a	b	b	c	a	b	b	B	B	…

Formalmente, M=(Q, Σ,Γ, δ, q_0, B, F} onde

1. Q={[q_i, d] | q $\in$ {q_1,q_2,...,q_9} e d=a,b ou B}
2. Σ={[B,d] | d=a,b ou c}, ou seja: um símbolo de entrada é sempre um símbolo "comum" acompanhado do B (que fica na segunda fita).
3. Γ={[X,d] | X=B ou ✓ e d=a,b,c ou B}
4. q_0=[q_1,B]
5. F={[q_9,B]}

Função de Transição δ (d = a ou b; e = a ou b)

1. δ([q_1,B],[B,d])=([q_2,d],[✓,d],R)

Lê símbolo em q_1, vai p/ q_2 guardando símbolo, marca fita e avança p/ direita.

2. δ([q_2,d],[B,e])=([q_2,d],[B,e],R)

Em q_2 continua avançando p/ direita, enquanto não aparece o c.

3. δ([q_2,d],[B,c])=([q_3,d],[B,c],R)

Ao achar o c, muda para estado q_3. Continua avançando p/ direita.

4. δ([q_3,d],[✓,e])=([q_3,d],[✓,e],R)

Continua avançando p/ direita, ignorando os símbolos já marcados.

5. δ([q_3,d],[B,d])=([q_4,B],[✓,d],L)

Achou um símbolo não-marcado após o c. Muda p/ estado q_4, "limpa" a memória, marca a posição e inicia a volta.

6. $\delta([q_4,B],[\checkmark,d])=([q_4,B],[\checkmark,d],L)$
Em q_4 continua avançando p/ esquerda, ignorando os símbolos já marcados.

7. $\delta([q_4,B],[B,c])=([q_5,B],[B,c],L)$
Ao achar o c, muda para estado q_5. Continua avançando p/ esquerda.

8. $\delta([q_5,B],[B,d])=([q_6,B],[B,d],L)$
Símbolo à esquerda de c não marcado: muda estado, continua avanço p/ esquerda. Se estiver marcado, tenho que fazer teste (transição 11).

9. $\delta([q_6,B],[B,d])=([q_6,B],[B,d],L)$
Continua avançando p/ esquerda, ignorando os símbolos não marcados.

10. $\delta([q_6,B],[\checkmark,d])=([q_1,B],[\checkmark,d],R)$
Achou um símbolo marcado antes do c. Muda p/ estado q_1, "limpa" a memória, e reinicia a ida (transição 1)

11. $\delta([q_5,B],[\checkmark,d])=([q_7,B],[\checkmark,d],R)$
O estado q_5 indica que o c foi encontrado na última transição, após a qual a cabeça foi p/ esquerda (transição 7). Símbolo à esquerda de c marcado, começa o teste: muda p/ estado q_7, volta p/ direita

12. $\delta([q_7,B],[B,c])=([q_8,B],[B,c],R)$
Teste em andamento: ao achar c, muda p/ q_8 e continua avançando p/ direita.

13. $\delta([q_8,B],[\checkmark,d])=([q_8,B],[\checkmark,d],R)$
Teste em andamento: símbolo à direita de c marcado, continua avançando p/ direita.

14. $\delta([q_8,B],[B,B])=([q_9,B],[\checkmark,B],L)$
Termina o teste: achou [B,B], vai p/ estado final e pára (com aceitação). P/ qualquer outro [X,Y] na fita com estado q_8, ocorre parada sem aceitação.

Deslocamento de Símbolos

- Uma operação útil: deslocar símbolos diferentes de B para a direita.
- Idéia:
  – Lê o símbolo da fita, armazena-o no estado.
  – Substitui o símbolo da fita por aquele que havia sido lido quando da leitura realizada quando cabeça estava na posição imediatamente a esquerda.

Um exemplo: Ex. 7.6, Hopcroft/Ullman

Subrotinas

- A operação de MTs pode ser decomposta em um conjunto de “subrotinas”, que correspondem a diferentes partes de um programa.
- Idéia: uso estados específicos para controlar as chamadas e retornos da subrotina.

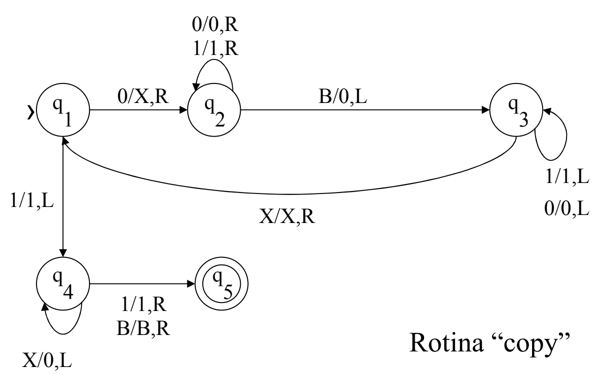
Um exemplo: Ex. 7.7, Hopcroft/Ullman

Visualização de sub-rotinas

- Uma MT para executar multiplicações:
  - Início: 0^m10^n1 na fita
  - Fim: 0^{mn} na fita

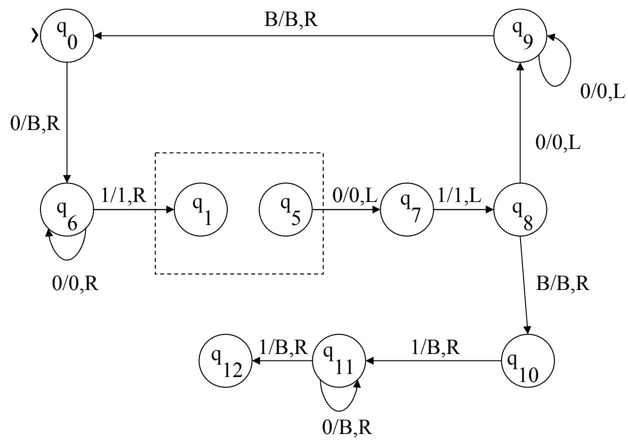
  - Estratégia:
    1. Em geral, a fita conterá $0^i10^n10^{kn}$ para algum k.
    2. Mudamos um 0 no primeiro grupo para B e **adicionamos n 0´s ao último grupo.** Resultado: $0^{i-1}10^n10^{(k+1)n}$
    3. O processo acima é executado m vezes, e a cada vez um 0 no primeiro grupo é trocado por B. Quando o primeiro grupo de 0´s tiver sido completamente trocado por B´s, teremos mn 0´s no último grupo.
    4. Etapa final: trocar os 10^n por B.

Visualização de sub-rotinas



Rotina “copy”

Programa de multiplicação usando a rotina "copy"



Modificações da Máquina de Turing (1)

- Fita infinita nas duas direções.
  - Similar à MT usual, mas admitindo deslocamento nas duas direções (parte à esquerda do símbolo inicial na fita é ocupado com B's).

Teorema: L é reconhecida por um MT com fita infinita nas duas direções sss L é reconhecida por uma MT usual. ☹

Modificações da Máquina de Turing (2)

- MT multifita.
  - Similar à MT usual, mas admitindo várias k fitas infinitas bidirecionais e k cabeças, comandadas por um único controle.
  - Para cada estado do controle e conjunto de símbolos lidos por cada uma das cabeças:
    - muda estado;
    - imprime um novo símbolo em cada uma das fitas;
    - move ou não cada uma das cabeças independentemente, para a direita ou esquerda.

Teorema: L é reconhecida por um MT multifita sss L é reconhecida por uma MT usual. ☹

Modificações da Máquina de Turing (3)

- MT não-determinística
  - Similar à MT usual, mas admitindo escolha não-determinística do trio <novo estado, novo símbolo a ser impresso, direção de movimento da cabeça>.

Teorema: L é reconhecida por um MT não-determinística sss L é reconhecida por uma MT usual. ☹

Modificações da Máquina de Turing (4)

- MT multidimensional
  - Similar à MT usual, mas admitindo fita como uma matriz infinita (no espaço n-dimensional). Inicialmente, a string a ser reconhecida é escrita em uma das direções, todo o resto da matriz sendo ocupado por B's.
  - Para cada estado do controle e símbolo lido:
    - muda estado;
    - imprime um novo símbolo na matriz;
    - move cabeça em uma das n direções, sentido positivo ou negativo.

Teorema: L é reconhecida por um MT multidimensional sss L é reconhecida por uma MT usual. ☹

Modificações da Máquina de Turing (5)

- MT multicabeças
  - Similar à MT usual, mas admitindo fita lida por k cabeças independentes. Para cada estado do controle e conjunto de símbolos lidos pelas cabeças:
    - muda estado;
    - cada cabeça imprime um novo símbolo;
    - move ou não cabeças independentemente, para direita ou esquerda

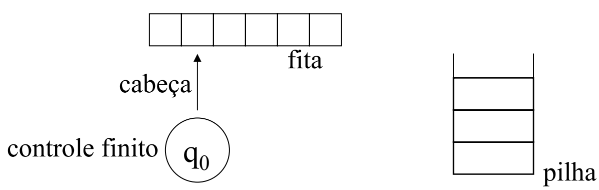
Teorema: L é reconhecida por um MT multicabeças sss L é reconhecida por uma MT usual. ☹

Máquinas equivalentes a MTs

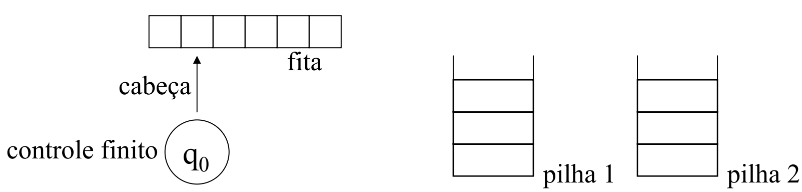
- Vimos que uma MT pode ter várias realizações equivalentes, aparentemente mais complexas:
  - várias fitas
  - memória no estado
  - não-determinismo
- Ok, mas... Existe máquinas aparentemente mais simples do que MTs com o mesmo poder computacional?

SIM!

Autômato de "Pilhas"



Reconhece LLCs ⇒ menos poderoso do que uma MT



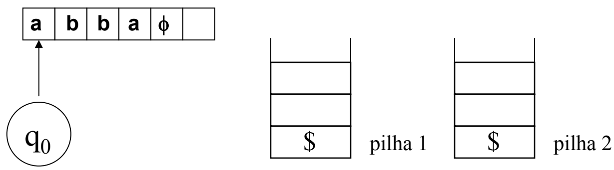
Reconhece LREs ⇒ equivalente a MTs !!!

Autômato de “Pilhas”

Início:

Pilhas vazias ($ no fundo)
Cadeia na fita com final marcado por um símbolo ϕ.

Regra de transição generalizada: $\delta(q,a,X_1,X_2,\ldots,X_k) = (p, \Gamma_1, \Gamma_2,\ldots, \Gamma_k)$
X_i: símbolos
Γ_i: cadeias



A.M. Turing Award

ACM's most prestigious technical award is accompanied by a prize of **US$100.000**. It is given to an individual selected for contributions of a technical nature made to the computing community. The contributions should be of lasting and major technical importance to the computer field.

Some Former Award Recipients

1968 Richard Hamming	1975 Herbert A. Simon
1969 Marvin Minsky	1983 Dennis M. Ritchie
1971 John McCarthy	1984 Niklaus Wirth
1972 E.W. Dijkstra	1986 John Hopcroft
1974 Donald E. Knuth	1994 Edward Feigenbaum
1975 Allen Newell	2003 Alan Kay

CT200
Fundamentos de Automata e Linguagens Formais

Prof. Carlos H. C. Ribeiro
carlos@comp.ita.br
Ramal: 5895   Sala: 106

Aula 10

Linguagens irrestritas e linguagens recursivas
Computabilidade
Problemas de decisão
A Tese de Church-Turing
O Problema da parada
Redução de Problemas
O Teorema de Rice
O Problema de Correspondência de Post

Linguagens Irrestritas

- As linguagens irrestritas (ou recursivamente enumeráveis) formam a classe de linguagens aceitas pelas Máquinas de Turing.
- Uma linguagem irrestrita corresponde à linguagem gerada por uma <u>Gramática Irrestrita</u> (ou seja, são as linguagens do tipo 0).
- Analogamente aos teoremas de Kleene e Chomsky:

Gramáticas Irrestritas $\xrightarrow{\text{produzem}}$ Linguagens Irrestritas $\xrightarrow{\text{reconhecidas por}}$ Máquinas de Turing

Linguagens Recursivas

- Problema: Para cadeias <u>não aceitas</u> por uma MT, pode não ocorrer a parada...
- Defino então:
  - Uma linguagem L é dita <u>recursiva</u> se for aceita por pelo menos uma MT que produz parada (com ou sem aceitação) para qualquer entrada.

Teorema: A classe das linguagens recursivas é um subconjunto da classe de linguagens irrestritas.

Linguagens Recursivas: Propriedades

- O complemento de uma linguagem recursiva é uma linguagem recursiva.

Prova: Seja L ling. recursiva e M a MT que a aceita com parada para qualquer entrada. Construa M' a partir de M como:
  - Se M entra em estado final com entrada w, M' pára sem aceitação.
  - Se M pára sem aceitação, M' entra em estado final.

As cadeias aceitas por M são exatamente aquelas não aceitas por M', e as cadeias aceitas por M' sao exatamente as não aceitas por M. Assim, L(M') é o complemento de L(M). Como M' produz parada p/ qualquer cadeia e aceita L(M'), então L(M') é recursiva.

Ilustração: Hopcroft/Ullman, pag. 180.

- A união de linguagens recursivas é uma linguagem recursiva.
- A união de linguagens irrestritas é uma linguagem irrestrita.

Linguagens Formais e Modelos Computacionais

- Existem dispositivos computacionais com capacidades intermediárias entre autômatos finitos e máquinas de Turing, que reconhecem linguagens do tipo 2 (livres de contexto) e tipo 1(sensíveis ao contexto).
- Os **autômatos de pilha** (já vimos) reconhecem linguagens do tipo 2 (livres de contexto).
- Os **autômatos limitados lineares** (máquina de Turing cujo cursor de leitura/gravação está limitado à parte da fita que contém a entrada original) reconhecem linguagens do tipo 1(sensíveis ao contexto).

Linguagens Formais e Modelos Computacionais

Linguagem	**Gramática**	**Disp. Computacional**
irrestrita	irrestrita	Máquina de Turing
sensível ao contexto	sensível ao contexto	Autômato limitado linear
livre de contexto	livre de contexto	Autômato de Pilha
regular	regular	Autômato Finito

Problemas de Decisão

- Um problema de decisão P é um conjunto de questões, cada uma com uma resposta S ou N.
- Uma <u>solução</u> para um problema de decisão P é um algoritmo que determina a resposta apropriada para toda questão $p \in P$.
- E o que é um algoritmo? Dificil de definir, mas algumas propriedades sao claras:
  - Um algoritmo é completo: produz resposta S ou N para cada questao do problema.
  - Um algoritmo é mecânico: consiste de uma sequência finita de instruções não-ambíguas.
  - Um algoritmo é determinístico: produz sempre a mesma resposta para uma dada entrada.

Decisão × Otimização

- Muitos problemas de interesse são **problemas de otimização**, onde cada solução tem um valor associado, e deseja-se encontrar a solução com o melhor valor.
  - Ex: caminho mais curto entre dois nós em um grafo.
- **Problemas de decisão** são aqueles cuja resposta é SIM / NÃO (0 ou 1).
- Problemas de otimização podem ser facilmente convertidos em problemas de decisão.
  - Ex.: Ao invés de procurar pelo caminho mais curto entre dois vértices em um grafo, pergunta-se se existe um caminho com comprimento < k e varia-se k.

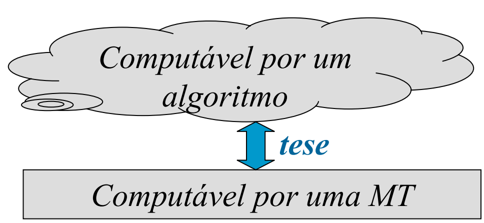
MTs e Algoritmos

- Uma MT:
  - É mecânica: consiste de uma sequência finita de instruções não ambíguas.
  - É determinística: produz sempre a mesma resposta para uma dada entrada.
  - Se o problema de decisão a ser resolvido puder ser expresso como uma linguagem recursiva, a MT correspondente é completa: produz resposta S (equiv. aceitação com parada) ou N (equiv. a rejeição com parada) para cada questão do problema.
- Nestas condições, **uma MT seria uma representação computacional formal de um algoritmo**...

Tese de Church-Turing

- Em 1936, com os artigos publicados por Turing e Alonzo Church, é que se definiu claramente o que de fato seria um **algoritmo**: **é uma MT que pára em respostas a todas as entradas.**
- Tese de Church-Turing:

"Qualquer problema de decisão é computável por um algoritmo se, e somente se, é computável por uma Máquina de Turing."



A Tese Church-Turing Estendida

Def.: Um problema de decisão P é parcialmente computável se existir um procedimento mecânico e determinista que produz a resposta S para cada questão de P de resposta S, mas que pode não produzir resposta para questões cuja resposta é N.

Tese: Um problema de decisão P é parcialmente computável se existe uma MT que aceita exatamente as questões de P cuja resposta é S.

Alonzo Church (14/06/1903 – 11/08/1995)

Matemático americano, responsável por alguns dos fundamentos da CC. Nasceu em Washington, DC, recebeu BA e PhD na Universidade de Princeton, USA, em 1924 e 1927, respectivamente. Tornou-se professor em Princeton em 1929. Foi orientador de Stephen C. Kleene (1909-1994), entre outros. Também orientou Alan Turing de 1936 a1938. Seu trabalho mais conhecido é o desenvolvimento do cálculo-λ no seu famoso artigo de 1936, mostrando a existência de problemas indecidíveis. Ele e Turing mostraram que o cálculo- λ e a MT são equivalentes em capacidades, resultando na tese de Church-Turing. Como há disputa sobre "quem foi o primeiro", esta tese também é conhecida como tese de Church e tese de Turing.

Problemas e Linguagens

- Um problema de decisão pode ser visto como um problema de reconhecimento de linguagem.
- Seja U o conjunto de todas as possíveis entradas para um problema de decisão (cujas respostas podem ser sim ou não).
- Seja $L \subseteq U$ o conjunto de todas as entradas que têm sim como resposta.
- Assim, L é a **linguagem correspondente ao problema de decisão**.
- Um problema é dito **decidível** se existir uma representação na qual o conjunto de cadeias que representam suas questões forma uma linguagem recursiva.
- Naturalmente, nem sempre conseguiremos representar um problema de decisão como uma linguagem recursiva: neste caso temos um problema **não-decidível**.
- **Para problemas não-decidíveis, não existe MT que páre para qualquer cadeia lida.**

Suficiência das MTs: Um Exemplo

Argumento principal em favor da suficiência de MTs: nenhum modelo computacional desenvolvido mostrou-se mais poderoso do que a MT.

Exemplo típico: RAM

- Conjunto infinito de palavras de memória (endereços) numeradas, cada uma armazenando um número inteiro; e
- Conjunto finito de registradores aritméticos, capazes de armazenar qualquer número inteiro.
- Cada número inteiro pode codificar instruções, variáveis, etc.

É possível se projetar uma MT equivalente a uma RAM...

Computabilidade

- Preocupação: classificar problemas como **computáveis** ou não.
- Como a capacidade de uma MT para resolver um problema (embora não muito eficientemente) **excede** a dos computadores atuais, se encontrarmos problemas que as MTs não podem fazer, saberemos que os computadores também não o poderão.
- Um problema é **incomputável (ou insolúvel)** quando não existe um algoritmo que possa executá-lo.

Exemplo de Problema Insolúvel

- Suponha que seja dado um programa computacional e uma especificação precisa do que este programa deve fazer (ex.: ordenar uma lista de números).
  – Note: o programa e a especificação são objetos matemáticos precisos, neste exemplo.
- Pode-se automatizar o processo de verificar se o programa executa o especificado por meio de um outro programa de computador?
- Infelizmente, o problema geral de **verificação de software** é **insolúvel** por computador!

Codificando Máquinas de Turing

- Como codificar uma MT sobre um alfabeto (por exemplo, {0,1})?
  - $M=(Q,\{0,1\},\{0,1,B\},\delta,q_1,B,\{q_2\})$
  - Faço: $X_1=0$, $X_2=1$, $X_3=B$, $D_1=L$, $D_2=R$. Assim, uma transição genérica $\delta(q_i,X_j)=(q_k,X_l,D_m)$ é representada por $0^i10^j10^k10^l10^m1$.
  - A máquina de Turing M é então representada como:
    $111c_111c_211...11c_r111$
    onde cada c_i é uma transição genérica da máquina.

Máquina × Cadeia

- Como a descrição ⟨T⟩ de uma MT é finita, podemos transformá-la em uma cadeia finita (eventualmente longa) descrevendo a MT.
- T(⟨T⟩) : significa que a MT T executa sua própria descrição ⟨T⟩ como cadeia de entrada.

Linguagem indecidível

- Seja a linguagem composta pelas cadeias $\langle T,\alpha\rangle$, correspondentes à concatenação das descrições $\langle T\rangle$ de MTs com cadeias α:

$$\mathbf{A_{MT} = \{ \langle T,\alpha\rangle \mid T \text{ é uma MT e } T \text{ aceita } \alpha\}}$$

- A linguagem A_{MT} é decidível? Este problema consiste em saber se existe uma MT H que **decide** A_{MT}.
- Provaremos, por contradição, que A_{MT} é indecidível, considerando como hipótese a existência de H.

Linguagem Indecidível

$$H(\langle T,\alpha\rangle): \begin{cases} \text{H aceita } \langle T,\alpha\rangle \text{ se T aceita } \alpha \\ \text{H rejeita } \langle T,\alpha\rangle \text{ se T não aceita } \alpha \end{cases}$$

- Considere a máquina D, definida em função de H:

D = "Para a entrada $\langle M\rangle$, sendo M uma MT:

1. Execute H com a entrada $\langle M, \langle M\rangle\rangle$.
2. Se H rejeitar, <u>aceite</u>.
3. Se H aceitar, <u>rejeite</u>."

$$\text{Assim, } D(\langle M\rangle): \begin{cases} \text{D aceita } \langle M\rangle \text{ se M não aceita } \langle M\rangle \\ \text{D rejeita } \langle M\rangle \text{ se M aceita } \langle M\rangle \end{cases}$$

Linguagem Indecidível

Hipótese: $H(\langle T,\alpha\rangle): \begin{cases} \text{H aceita } \langle T,\alpha\rangle \text{ se } \boxed{\text{T aceita } \alpha} \\ \text{H rejeita } \langle T,\alpha\rangle \text{ se } \boxed{\text{T não aceita } \alpha} \end{cases}$ **(1)**

- Seja a MT D:

$$D(\langle M\rangle): \begin{cases} \text{D aceita } \langle M\rangle \text{ se H rejeita } \langle M,\langle M\rangle\rangle \\ \text{D rejeita } \langle M\rangle \text{ se H aceita } \langle M,\langle M\rangle\rangle \end{cases}$$

- Se ⟨M⟩ = ⟨D⟩, vem:

$$D(\langle D\rangle): \begin{cases} \text{D aceita } \langle D\rangle \text{ se H rejeita } \langle D, \langle D\rangle\rangle \\ \text{D rejeita } \langle D\rangle \text{ se H aceita } \langle D, \langle D\rangle\rangle \end{cases}$$

- Mas, segundo (1), o que significa H(⟨D, ⟨D⟩⟩)?

$$D(\langle D\rangle): \begin{cases} \text{D aceita } \langle D\rangle \text{ se } \boxed{\text{D não aceita } \langle D\rangle} \\ \text{D rejeita } \langle D\rangle \text{ se } \boxed{\text{D aceita } \langle D\rangle} \end{cases}$$ **Contradição!!!**

- **Assim, D e H não podem existir e A_{MT} é indecidível**!

Problema da Parada

- Uma MT T que começa com uma cadeia α ou pára ou nunca pára.
- 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**.

Problema da Parada

- Seja a linguagem composta pelas cadeias $\langle T,\alpha\rangle$, correspondentes à concatenação das descrições $\langle T\rangle$ de MTs com cadeias α:

$HALT_{MT}$ = { $\langle T,\alpha\rangle$ | T é uma MT e T pára com a entrada α}

- O Problema da Parada corresponde a saber se existe uma MT R que **decide** $HALT_{MT}$:

$$R(\langle T,\alpha\rangle)\ \begin{cases} \text{R aceita } \langle T,\alpha\rangle \text{ se T pára com } \alpha \\ \text{R rejeita } \langle T,\alpha\rangle \text{ se T não pára com } \alpha \end{cases}$$

Problema da Parada

- A MT R pode ser usada na construção da MT H que decide a linguagem A_{MT}.

H = "Para a entrada $\langle$**T**,$\alpha\rangle$**:**

1. **Execute R com a entrada $\langle$T,$\alpha\rangle$.**
2. **Se R rejeitar, <u>rejeite</u>.** *// T não pára com α*
3. **Se R aceitar, simule T com α até T parar** *// T pára com α*
4. **Se T aceitar, <u>aceite</u>. Se T rejeitar, <u>rejeite</u>."**

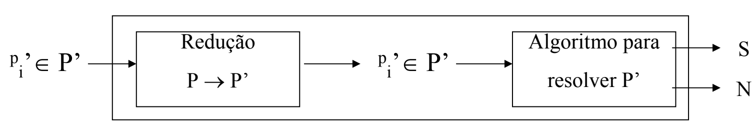
- Assim, se R decide $HALT_{MT}$, então H decide A_{MT}.
- Como vimos que H não pode existir e que A_{MT} é indecidível, então R também não existe e $HALT_{MT}$ é indecidível.

Redução de Problemas

- Problemas “difíceis” freqüentemente podem ser transformados (reduzidos) em outros, para os quais uma solução já foi encontrada.
- Formalmente:
  - Um problema de decisão P e redutível a P’ se existir uma MT que, a partir de uma entrada que representa uma questão $p_i \in P$, produz um problema $p_i' \in P'$ que tem a mesma resposta de p_i. O mapeamento pode ser “muitos para um”, ou seja: varias questões de P podem ser reduzidas para uma mesma questão de P’.

Solução de Problemas via Redução

- Se um problema P’ é decidível e P é redutível a P’, então P é também decidível. Uma solução para P pode ser portanto obtida a partir de:



Obs: Se P é não-decidível e P é redutível a P’, então P’ também é não-decidível.

Solução via Redução: Exemplo 1

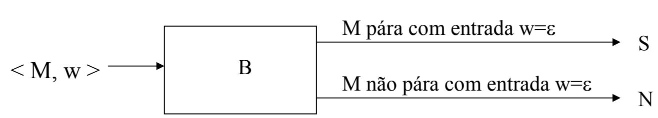
- Redução do problema **P: Determinar se uma dada cadeia pertence à linguagem L={uu | u=$a^i b^i c^i$ para algum i ≥ 0}** ao problema **P': Determinar se uma dada cadeia pertence à linguagem L'={u | u =$a^i b^i c^i$ para algum i ≥ 0}** .
  - Construo a maquina M' que aceita L'.
  - Copio cadeia de entrada w.
  - Uso esta cópia para determinar se w=uu, para algum u ∈ $\{a, b, c\}^*$:
    - Se w = uu, limpo a fita e deixo u na entrada de M'. Se M' agora aceitar u, é porque a entrada de M era uu, com u = $a^i b^i c^i$.
    - Se w ≠ uu, limpo a fita e deixo um a na entrada, que não será aceito por M'.

Solução via Redução: Exemplo 2

- Mostrar que não existe algoritmo que determina se uma MT qualquer pára quando uma computação é iniciada com a fita em branco.

Idéia: Reduzir o problema da parada em MTs a este problema

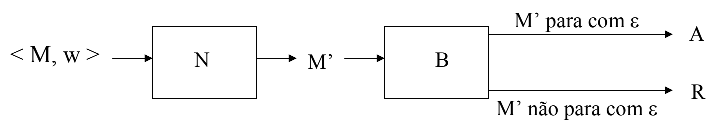
Assuma que exista uma máquina B que resolve o problema, ou seja, que determina se uma dada máquina M pára com entrada em branco:



Solução via Redução: Exemplo 2

Reduzo o problema de parada a este problema usando uma máquina N que le <M, w> e produz na saída uma representação de M' que:

1. Escreve w em uma fita em branco.
2. Retorna a cabeça para a posição inicial e no estado inicial de M.
3. Executa M.



N reduz o problema de parada ao problema da fita em branco.
Logo, problema da fita em branco é não-decidível

O Teorema de Rice: Motivação

Idéia: Ao invés de problemas ligados a decidabilidade envolvendo MTs e uma entrada particular

- Máquina M_i pára com entrada w?
- Máquina M_i pára com entrada em branco?

Etc, etc.

Podemos estar interessados em problemas mais gerais, tais como:

- A cadeia vazia pertence à $L(M_i)$?
- $L(M_i)$ é $\varnothing$?
- $L(M_i)$ é regular?
- $L(M_i) = \Sigma^*$?

Etc, etc.

O Teorema de Rice: Codificação

- Qualquer MT pode ser codificada como uma cadeia sobre {0,1}.
- Um conjunto de MTs portanto define uma linguagem sobre $\Sigma = \{0,1\}$.
- As questões anteriores podem ser reformuladas como:
  - M_i pertence a linguagem $L_\varepsilon = \{ M \mid \varepsilon \in L(M)\}$?
  - M_i pertence a linguagem $L_\varnothing = \{M \mid L(M) = \varnothing\}$?
  - M_i pertence a linguagem $L_{reg} = \{M \mid L(M)$ e regular$\}$?
  - M_i pertence a linguagem $L_\Sigma = \{M \mid L(M) = \Sigma^*\}$?

Etc, etc.

O Teorema de Rice

Def.: Uma <u>propriedade</u> de uma l.r.e. é uma condição que a l.r.e. pode satisfazer. Exemplos: “conter cadeia nula”, “ser regular”, etc.

Def.: Uma <u>linguagem de uma propriedade $\wp$</u> de uma l.r.e. é o conjunto $L_\wp = \{M \mid M$ satisfaz $\wp\}$. Exemplo: $L_\varnothing$ é a linguagem formada por todas as cadeias que representam as MTs que não aceitam nenhuma cadeia.

Def.: Uma propriedade $\wp$ de uma l.r.e. e dita <u>trivial</u> se a) não existirem linguagens que a satisfaçam ou b) todas as l.r.e.’s a satisfazem.

Teorema (Rice, 1953): Se $\wp$ é uma propriedade não trivial de linguagens recursivamente enumeráveis, então $L_\wp$ não é recursiva.

O Teorema de Rice: Algumas Conseqüências

- A propriedade "a linguagem é regular" é não-trivial: existem l.r.e.'s que a satisfazem, e existem l.r.e.'s que não a satisfazem. Pelo Teorema de Rice, então L_{reg} = {M | L(M) é regular} não é recursiva. Logo, não existe MT capaz de ter como entrada <u>qualquer</u> máquina M e parar com aceitação caso esta aceite uma linguagem regular, e parar com rejeição caso esta não aceite uma linguagem regular.
- Não existe MT capaz de ter como entrada <u>qualquer</u> máquina M e parar com aceitação caso esta aceite a cadeia vazia, e parar com rejeição caso esta não aceite a cadeia vazia.
- Não existe MT capaz de ter como entrada <u>qualquer</u> máquina M e parar com aceitação caso esta não aceite nenhuma cadeia, e parar com rejeição caso esta aceite alguma cadeia.

Etc, etc...

O Problema de Correspondência de Post

- Trata-se de um problema de dominós...
- Cada dominó: primeiro e segunda metades são cadeias de um mesmo alfabeto.
- Assumo um conjunto infinito de dominós de um dado tipo: jogar um dominó não limita o numero de escolhas futuras.
- Dominós são posicionados lado a lado, formando cadeias completas na metade superior e na metade inferior.
- **O problema: posicionar os dominós de modo que a cadeia da metade superior fique igual à cadeia da metade inferior.**

Exemplo 1: Alfabeto {a,b}, dominós [aaa, aa] e [baa, abaaa]
Solução:

aaa	*baa*	*aaa*	aaabaaaaa
aa	*abaaa*	*aa*	aaabaaaaa

O Problema de Correspondência de Post: Exemplo 2

Alfabeto {a,b}, dominós [ab, aba], [bba, aa], [aba,bab]

Solução:

Tenho que começar com:

ab
aba

(única maneira de ter os mesmo prefixos)

Dominó seguinte não pode ser [bba,aa]. Tenho então duas possibilidades:

ab	*ab*
aba	*aba*

Não serve: quarta posição

ab	*aba*
aba	*bab*

Agora, tenho que começar com b na metade de cima...

O Problema de Correspondência de Post: Exemplo 2

ab	*aba*	*bba*
aba	*bab*	*aa*

Com problema na sétima posição... Problema sem solução !!!

Vemos portanto que o Problema de Correspondência de Post pode ou não ter solução, dependendo do alfabeto e dominós.

A questão é: existe um algoritmo para determinar se uma dado P.C.P. tem solução?

Em outras palavras: **O P.C.P. e decidivel?**

O P.C.P. é base para vários teoremas envolvendo decidabilidade (especialmente p/ ling. livres de contexto, como veremos...)

Formalização do P.C.P.

Seja um alfabeto Σ e um conjunto de pares ordenados $[u_i, v_i]$, i=1,2,...,n onde $u_i, v_i \in \Sigma^+$. Uma solução para o sistema de correspondência de Post é uma seqüência i1, i2, ..., ik tal que

$$u_{i1}\, u_{i2} \ldots u_{ik} = v_{i1}\, v_{i2} \ldots v_{ik}$$

O problema de se determinar se um sistema de correspondência de Post tem alguma solução é chamado de

Problema de Correspondência de Post

Teorema (Post, 1946): Não existe algoritmo para se determinar se um dado P.C.P. tem uma solução, ou seja: **o P.C.P. é um problema não-decidível**.

Relembrando: Gramáticas Livres de Contexto

Uma G.L.C. é uma gramática G=(V,T,P,S) em que cada produção é da forma $A \rightarrow w$, em que $A \in V$ e $w \in (V \cup \Sigma)^*$.

Uma dada G.L.C. (gramática tipo 2 segunda a hierarquia de Chomsky) produz cadeias de uma linguagem livres de contexto, que é por sua vez reconhecida por um autômato *Pushdown*.

Uma derivação de uma sentença em uma gramática em que a variável transformada é sempre aquela mais à esquerda é dita derivação à esquerda. Qualquer cadeia terminal de uma G.L.C. pode ser obtida via derivações à esquerda.

Uma G.L.C. é dita ambígua se existir alguma cadeia $w \in L(G)$ que pode ser obtida por duas derivações à esquerda distintas.

Gramáticas Livres de Contexto Ambíguas

Uma G.L.C. é dita ambígua se existir alguma cadeia w $\in$ L(G) que pode ser obtida por duas derivações à esquerda distintas.

Exemplo:

G_1: $S \rightarrow aS \mid Sa \mid a$ (ambígua, pois aa tem duas derivações à esquerda).

G_2: $S \rightarrow aS \mid a$ (não ambígua, e equivalente a G_1).

A ambigüidade é uma propriedade da gramática, e não da linguagem. Porem , podemos definir:

Uma linguagem livre de contexto é dita ser inerentemente ambígua se toda G.L.C. que a gera for ambígua.

P.C.P. e Gramáticas Livres de Contexto

Seja (Σ, $[u_1, v_1]$, $[u_2, v_2]$,..., $[u_n, v_n]$) um sistema de correspondência de Post. Considere as seguintes gramáticas livres de contexto:

G_U: $V_U = \{S_U\}$

$\Sigma_U = \Sigma \cup \{1, 2, ..., n\}$

$P_U = \{S_U \rightarrow u_i\, S_U\, i,\ S_U \rightarrow u_i\, i\}$ (i = 1, 2, ..., n)

G_V: $V_V = \{S_V\}$

$\Sigma_V = \Sigma \cup \{1, 2, ..., n\}$

$P_V = \{S_V \rightarrow v_i\, S_V\, i,\ S_V \rightarrow v_i\, i\}$ (i = 1, 2, ..., n)

G_U gera as cadeias da metade superior dos dominós.
G_V gera as cadeias da metade inferior dos dominós.
Os dígitos (i) registram a seqüência de dominós que gera a seqüência.

P.C.P. e Gramáticas Livres de Contexto

Uma solução do problema de correspondência de Post corresponde a uma seqüência $i_1 \, i_2 \ldots i_{k-1} \, i_k$ tal que $u_{i1} \, u_{i2} \ldots u_{ik-1} \, u_{ik} = v_{i1} \, v_{i2} \ldots v_{ik-1} \, v_{ik}$ Nesse caso, as gramáticas produzem as derivações:

$S_U \Rightarrow u_{i1} \, u_{i2} \ldots u_{ik-1} \, u_{ik} \, i_k \, i_{k-1} \ldots i_2 \, i_1$

$S_V \Rightarrow v_{i1} \, v_{i2} \ldots v_{ik-1} \, v_{ik} \, i_k \, i_{k-1} \ldots i_2 \, i_1$

com $u_{i1} \, u_{i2} \ldots u_{ik-1} \, u_{ik} \, i_k \, i_{k-1} \ldots i_2 \, i_1 = v_{i1} \, v_{i2} \ldots v_{ik-1} \, v_{ik} \, i_k \, i_{k-1} \ldots i_2 \, i_1$

Ou seja: $L(G_U) \cap L(G_V) \neq \emptyset$. Por outro lado, se $w \in L(G_U) \cap L(G_V)$ então $w = w' \, i_k \, i_{k-1} \ldots i_2 \, i_1$. A cadeia $w' = u_{i1} \, u_{i2} \ldots u_{ik-1} \, u_{ik} = v_{i1} \, v_{i2} \ldots v_{ik-1} \, v_{i\,k}$ é uma solução para o problema de correspondência de Post.

P.C.P. e Gramáticas Livres de Contexto

Teorema: Determinar se duas linguagens livres de contexto são disjuntas é um problema não-decidivel.

Prova: Assuma que o problema seja decidivel. Então existe um algoritmo para resolvê-lo, e o problema de correspondência de Post poderia ser resolvido da seguinte maneira:

1. Construa as gramáticas G_U e G_V como indicado no slide anterior;
2. Use o algoritmo para determinar se $L(G_U) \cap L(G_V) = \emptyset$.
3. Se $L(G_U) \cap L(G_V) \neq \emptyset$, tenho a solução (w') para o P.C.P. em questão. Se $L(G_U) \cap L(G_V) = \emptyset$, não tenho a solução (w') para o P.C.P. em questão.

Mas já vimos (teorema anterior) que o P.C.P. é não-decidivel! Contradição...

P.C.P. e Gramáticas Livres de Contexto

Exemplo: Gramáticas G_U e G_V para o sistema de Post [aaa,aa], [baa,abaaa]:

G_U: $S_U \rightarrow$ aaa S_U 1 | aaa1 | baaS_U 2 | baa2
G_V: $S_V \rightarrow$ aa S_V 1 | aa1 | abaaaS_V 2 | abaaa2

Derivações que produzem solução para o problema de correspondência:

$S_U \Rightarrow$ aaa S_U 1 $\Rightarrow$ aaabaa S_U 21 $\Rightarrow$ aaabaaaaa 121
$S_V \Rightarrow$ aa S_V 1 $\Rightarrow$ aaabaaa S_V 21 $\Rightarrow$ aaabaaaaa 121

Outros teoremas provados a partir da construção das gramáticas G_U e G_V :

Um Teorema...

- Não existe algoritmo para determinar se a linguagem de uma G.L.C. é Σ^*.

Prova: Um algoritmo para $L = \Sigma^*$ *equivale a um algoritmo para* $\overline{L} = \emptyset$
Seja portanto C um PCP. Como $\overline{L(G_V)}$ e $\overline{L(G_U)}$ *são GLCs, então*

$$L = \overline{L(G_U)} \cup \overline{L(G_V)}$$ *também é GLC. Pela Lei de De Morgan,*

$$\overline{L} = \overline{\overline{L(G_U)} \cup \overline{L(G_V)}} = L(G_U) \cap L(G_V)$$

Assim, ter um algoritmo para determinar se $\overline{L} = \emptyset$ *equivale a ter um algoritmo para determinar se* $L(G_U) \cap L(G_V)$

Mas este último é um problema não-decidível. Assim, não existe algoritmo para determinar se $\overline{L} = \emptyset$ *(ou* $L = \Sigma^*$*).*

Observação

Mas há algum tempo vimos que existe um algoritmo de decisão para linguagens regulares:

"Seja M um AFD com k estados. Então, L(M) é não-vazio $\Leftrightarrow$ M aceita uma cadeia z com length(z) < k."

O que mudou agora?

- Não existe mais um AFD de k estados que aceita L, pois L é GLC.
- Logo, não tenho como limitar a inspeção de cadeias z àquelas com length(z)<k.
- Assim, teria que verificar todas as infinitas cadeias em Σ^* ao APD que reconhece a linguagem L, para determinar se a linguagem de L é mesmo $\varnothing$. Mas um algoritmo deve terminar, e tentar operar sobre um número infinito de possíveis argumentos é naturalmente um processo que não termina!

Mais um Teorema...

- Não existe algoritmo para determinar se uma G.L.C. é ambígua.

Prova: Sejam G_U e G_V as GLCs associadas a um PCP C. Suponha que combinemos estas gramáticas para gerar uma terceira gramática da forma:

$$G: V = \{S, S_U, S_V\};\ \Sigma = \Sigma_C;\ P = P_U \cup P_V \cup \{S \rightarrow S_U,\ S \rightarrow S_V\}$$

Note que:

1. *Todas as derivações de G são á esquerda, já que qualquer forma sentencial contém no máximo um símbolo não-terminal.*
2. *G_U e G_V são não-ambíguas, pois derivações distintas produzem diferentes sequências de inteiros.*

Assim, G será ambígua apenas se e somente se existir alguma cadeia em $L(G_U) \cap L(G_V)$. Assim, um algoritmo para determinar se G é ambígua corresponde a uma algoritmo para determinar se $L(G_U) \cap L(G_V)$. Mas já vimos que determinar se duas LLC são disjuntas é não-decídivel!