# Capítulo 0: Conjuntos, funções, relações

***Notação.*** Usaremos **Nat** para representar o conjunto dos números naturais; **Int** para representar o conjunto dos números inteiros. Para cada $n \in$ **Nat**, [n] representa o conjunto dos naturais menores ou iguais a n:

$$[n] = \{ i \in \mathbf{Nat} \mid 0 < i \leq n \}.$$

Este conjunto [n] é às vezes representado por {1, 2, …, n}, convencionando-se que nos casos especiais n = 0 e n = 1, essa notação indica, respectivamente, o conjunto vazio $\varnothing$ e o conjunto unitário {1}.

***Produto Cartesiano.*** O produto cartesiano de dois conjuntos A e B é o conjunto $A \times B$ de pares ordenados de elementos de A e B:

$$A \times B = \{ (x, y) \mid x \in A \text{ e } y \in B \}.$$

Esse conceito pode ser estendido, usando n-tuplas, para definir o produto cartesiano de n conjuntos:

$$A_1 \times A_2 \times \ldots A_n = \{ (x_1, x_2, \ldots x_n ) \mid \text{para cada } i \in [n], x_i \in A_i \}$$

Podemos definir potências de um conjunto, a partir da definição de produto Cartesiano:

$$A^n = A \times A \times \ldots A \text{ (n vezes)} = \{ (x_1, x_2, \ldots x_n) \mid \text{para } i \in [n], x_i \in A\}.$$

Naturalmente, $A^1 = A$.

**Exemplo**: Sejam A = { a, b, c }, B = { d, e }. Então,

$A \times B$ = { (a, d), (a, e), (b, d), (b, e), (c, d), (c, e) }

$B \times A$ = { (d, a), (d, b), (d, c), (e, a), (e, b), (e, c) }

$A^1 = A$ = { a, b, c }

$A^2 = A \times A$ = { a, b, c } × { a, b, c } =

= { (a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c) }

□

***Relações.*** Podemos agora definir relação: dados n conjuntos $A_1$, $A_2$, …, $A_n$, uma relação em $A_1$, $A_2$, …, $A_n$ é um conjunto qualquer de tuplas de elementos de $A_1$, $A_2$, …, $A_n$. Portanto, usando a definição acima, R é uma *relação* em $A_1$, $A_2$, …, $A_n$ se

$$R \subseteq A_1 \times A_2 \times \ldots \times A_n.$$

Um caso especial que será muito importante no que se segue é o caso n=2, com $A_1=A_2=A$. R é uma *relação binária* em um conjunto A, se $R \subseteq A \times A$.

***Funções***. Outro caso especial é o das funções: uma relação f em $A \times B$, ou seja, um conjunto $f \subseteq A \times B$, é uma *função*, com *domínio* A e *codomínio* B, se para cada $x \in A$ existe em f um único $y \in B$ tal que $(x, y) \in f$. Essa unicidade pode também ser expressa por

$(x, y) \in f$ e $(x, z) \in f$ implicam em $y = z$.

Naturalmente, esse valor único de y que f faz corresponder a x é indicado pela notação habitual f(x), e podemos escrever também $f: x \mapsto y$. Escrevemos $f: A \to B$, para indicar que f é uma função com domínio A e codomínio B.

Definimos o *contradomínio* de $f: A \to B$ como sendo o conjunto

$\{ y \in B \mid (\exists x \in A) \ (f(x) = y) \}$.

**Exemplo:** Se considerarmos o conjunto **Int** dos números inteiros, e a função *suc*: **Int** $\to$ **Int** que a cada valor em **Int** associa seu sucessor, poderemos escrever

para cada $i \in$ **Int**, *suc*(i) = i + 1,

ou

*suc*: $i \mapsto i + 1$

ou ainda

*suc* = { …, (-2, -1), (-1, 0), (0, 1), (1, 2), … }

□

***Injeção, sobrejeção, bijeção***. Dizemos que uma função $f: A \to B$ é uma *injeção* se para cada $b \in B$ existe no máximo um $a \in A$ tal que f(a) = b; dizemos que $f: A \to B$ é uma *sobrejeção* se para cada $b \in B$ existe no mínimo um $a \in A$ tal que f(a) = b; dizemos que f é uma *bijeção* se f é ao mesmo tempo, uma injeção e uma sobrejeção.

No caso de sobrejeções (e bijeções), codomínio e contradomínio são iguais.

Alternativamente, podemos falar em funções *injetoras*, *sobrejetoras* ou *"sobre"*, e *bijetoras*.

***Conjuntos enumeráveis.*** Um conjunto A é *enumerável* se é vazio, ou se existe uma função sobrejetora $f:$ **Nat** $\to A$.

O nome *enumerável* se deve ao fato de que, se A não é vazio, a sequência f(0), f(1), f(2), f(3), … é uma lista infinita da qual fazem parte todos os elementos de A, ou seja, uma *enumeração* de A. Em particular, como não estão proibidas repetições em uma enumeração, temos:

**Fato:** Todos os conjunto finitos são enumeráveis.
Dem.: Exercício.

□

No que se segue, estaremos interessados principalmente em conjuntos enumeráveis infinitos. Neste caso, podemos usar uma *numeração*, em vez de uma *enumeração*. Por numeração entendemos aqui uma função como a função g mencionada na propriedade abaixo, que associa a cada elemento de A um número natural distinto.

**Fato:** Um conjunto infinito é enumerável, se e somente se existe uma função injetora g: A $\rightarrow$ **Nat**.
Dem. ($\Rightarrow$) Seja A um conjunto enumerável infinito. Pela definição, existe uma função sobrejetora f: **Nat** $\rightarrow$ A. Podemos definir a injeção g:A $\rightarrow$ **Nat** fazendo, para cada a $\in$ A, g(a) ser igual ao menor valor de i tal que f(i) = a. Assim, a função g é definida para qualquer valor de a, porque f é sobrejetora. Além disso, g é injetora, porque, pela própria definição, g(a) = g(b) implica em f(g(a)) = f(g(b)).

($\Leftarrow$) Seja A um conjunto tal que existe uma injeção g:A $\rightarrow$ **Nat**. Uma vez que A não é vazio, seja q um elemento qualquer de A. Defina agora a sobrejeção f: **Nat** $\rightarrow$ A por

$$f(i) = \begin{cases} a, & \text{se existir um a tal que } g(a) = i \\ q, & \text{se não existir} \end{cases}$$

Note que f é bem definida para todos os valores de i, porque g é uma injeção, e, para cada i, pode haver, no máximo, um a tal que g(a) = i; f é uma sobrejeção, porque g é definida para todos os elementos de A.

□

**Fato:** Um conjunto infinito A é enumerável se e somente se existe uma bijeção
f: A $\rightarrow$ **Nat**.
Dem.: Exercício.

□

**Fato:** Entre dois conjuntos infinitos enumeráveis A e B existe sempre uma bijeção
f: A$\rightarrow$B.
Dem.: Exercício.

□

**Exemplo:** O conjunto **Nat** é enumerável.
Basta tomar f como sendo a função identidade I: **Nat** $\rightarrow$ **Nat**, que é, claramente, uma bijeção.

□

**Exemplo:** O conjunto $\mathbf{Nat}^2$ = **Nat** $\times$ **Nat** de pares de números naturais é enumerável.

Podemos fazer a caracterização de diversas maneiras:

1. através da injeção g:$\mathbf{Nat^2} \rightarrow$ **Nat** definida por g( (i,j) ) = $2^i\ 3^j$. Esta numeração dos pares de inteiros é às vezes chamada de *numeração de Goedel*. Esse processo pode ser estendido a potências superiores de **Nat**. Por exemplo, podemos associar à tripla (i, j, k) o número $2^i\ 3^j\ 5^k$. Para n-uplas, poderiam ser usados como bases os primeiros n números primos.
2. definindo diretamente a ordem de enumeração:

```
repita para cada k = 0, 1, 2, …
        enumere os pares (i, j) tais que i+j = k, na ordem crescente de i:
                (0, k), (1, k-1), …, (k-1, 1), (k, 0).
```

Isso corresponde a

(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), (0, 3) …

ou seja, a uma sobrejeção f: **Nat** $\rightarrow$ **Nat** dada por

f(0) = (0,0), f(1) = (0,1), f(2) = (1,0), f(3) = (0,2), …

□

**Exemplo:** O conjunto **Int** dos inteiros é enumerável.
Basta usar uma enumeração como 0, -1, +1, -2, +2, -3, +3, …

□

**Teorema:** O conjunto P(**Nat**) dos subconjuntos de **Nat** não é um conjunto enumerável.

Dem.: por "diagonalização".

Uma vez que a definição de conjunto enumerável se baseia na existência de uma função com certas propriedades, devemos mostrar que tal função não existe, e a demonstração será feita por contradição (ou redução ao absurdo).

Suponhamos que o conjunto P(**Nat**) é enumerável. Isto significa que existe uma enumeração de P(**Nat**), ou seja uma sobrejeção f: **Nat** $\rightarrow$ P(**Nat**). Assim, para cada elemento A de P(**Nat**) (um conjunto A de naturais), existe um número i tal que f(i) = A.

Vamos considerar o conjunto X definido a seguir:

$X = \{ j \in \mathbf{Nat} \mid j \notin f(j) \}$

Como X é um conjunto de naturais, $X \in$ P(**Nat**). Entretanto, veremos que X não faz parte da enumeração acima. Seja k qualquer. Duas possibilidades podem ocorrer:

- ou $k \in f(k)$, e neste caso $k \notin X$,
- ou $k \notin f(k)$, e neste caso $k \in X$.

Em qualquer das possibilidades, portanto, os conjuntos X e f(k) diferem em pelo menos um elemento. Assim, $X \neq f(k)$ para todos os k. Desta forma, X não faz parte da enumeração definida por f, caracterizando-se a contradição. Consequentemente, P(**Nat**) não é enumerável.

□

Esta técnica de demonstração recebeu o nome de *diagonalização.* Representamos um conjunto $A \subseteq$ **Nat** por uma sequência infinita de 0's e 1's: se $i \in A$, o i-ésimo símbolo da sequência será 1; caso contrário, será 0. Assim, se fizéssemos uma tabela infinita com uma linha correspondendo a cada conjunto f(k), $k \in$ **Nat**, o conjunto X seria definido invertendo o que se encontra na diagonal da tabela: se na posição (i,i) se encontra um 1, indicando que $i \in f(i)$, na linha correspondente a X teríamos um 0 na i-ésima coluna, indicando que $i \notin X$, e (vice-versa) se na posição i,i se encontra um 0, indicando que $i \notin f(i)$, na linha correspondente a X teríamos um 1 na i-ésima coluna, indicando que $i \in X$.

Desta forma, podemos ver que, para qualquer i, $f(i) \neq X$. Para isso, basta notar que i pertence a exatamente um dos dois conjuntos f(i) e X. Portanto, qualquer que fosse a enumeração de P(**Nat**), X não pertenceria a ela.

Esta técnica será usada neste curso em diversas ocasiões para demonstrações semelhantes à anterior; foi usada por Cantor, para mostrar que a cardinalidade de um conjunto P(A) é sempre superior à cardinalidade de A. O mesmo vale aqui: a cardinalidade de todos os conjuntos enumeráveis infinitos A é a mesma, equivalente à de **Nat**, mas a cardinalidade dos conjuntos potência P(A) é superior à de **Nat**, sendo equivalente à de P(**Nat**). Falando *informalmente*,

- "todo conjunto enumerável tem o mesmo número de elementos que **Nat**."
- "há mais elementos em P(**Nat**) do que em **Nat**."
- "para qualquer conjunto A enumerável, P(A) tem o mesmo número de elementos que P(**Nat**)."

**Fato:** Se um conjunto A é enumerável, e se B é um subconjunto de A, B também é enumerável.
Dem. Exercício.

□

Exercícios:
*(1)* Mostre que, se A e B são conjuntos enumeráveis, então A×B também é enumerável.
*Sugestão:* se A e B são enumeráveis, existem numerações $n_A$: A→**Nat** e $n_B$: B→**Nat**; seja então g: $\mathbf{Nat}^2 \rightarrow \mathbf{Nat}$ a mesma numeração de $\mathbf{Nat}^2$ vista anteriormente; considere então a função n: A×B → **Nat** definida por

$$n( (a, b) ) = g(n_A (a), n_B (b)).$$

*(2)* Uma das definições possíveis para par ordenado é a seguinte: definimos o par ordenado (a, b) como sendo o conjunto {{a, b}, {a}}. Mostre que, com esta definição, vale a propriedade fundamental:

(a, b) = (c, d) se e somente se a=c e b=d.

*(3)* Podemos definir uma tripla (ou 3-tupla) a partir da definição de par ordenado:

(a, b, c) = ((a, b), c).

Isto corresponde a definir $\mathbf{Nat}^3$ como $\mathbf{Nat}^2 \times \mathbf{Nat}$. Mostre que com esta definição, vale a propriedade fundamental:

(a, b, c) = (d, e, f) se e somente se (a=d) e (b=e) e (d=f).

*(4)* Para definir uma numeração dos elementos de **Nat**, podemos usar as funções $F_1$ e $F_2$ definidas a seguir:

$$F_1 ( (i,j,k) ) = 2^i 3^j 5^k$$

$$F_2 ( (i,j,k) ) = g( (i, g( (j,k) ) ) ),$$

onde g é a função definida anteriormente:

$$g( (i,j) ) = 2^i 3^j.$$

*Experimente* calcular $F_1 ( (5, 5, 5) )$ e $F_2 ( (5, 5, 5) )$.

□

***Relações binárias.*** Quando tratamos de relações binárias, normalmente usamos uma notação mais simples para indicar que (x, y) é um elemento de uma relação binária R em

A: escrevemos apenas x R y. Essa notação é semelhante à usada para relações comuns, como as relações de ordem $<$, $\leq$, etc.: não escrevemos $(x, y) \in \leq$, mas, mais simplesmente, $x \leq y$.

Vamos a seguir introduzir algumas propriedades de relações binárias. Seja R uma relação binária em um conjunto A ($R \subseteq A^2$ ). Então dizemos que

- R é reflexiva se para qualquer $x \in A$, x R x;
- R é simétrica se, para quaisquer $x, y \in A$, x R y implica y R x.
- R é transitiva se, para quaisquer $x, y, z \in A$, x R y e y R z implicam em x R z.

**Exemplos:** As relações $<$, $\leq$, $=$, $\neq$ são relações binárias definidas no conjunto **Nat**, e tem as propriedades indicadas a seguir:

| | *reflexiva* | *simétrica* | *transitiva* |
|---|---|---|---|
| $<$ | não | não | sim |
| $\leq$ | sim | não | sim |
| $=$ | sim | sim | sim |
| $\neq$ | não | sim | não |

□

***Equivalência.*** Uma relação R é uma *relação de equivalência* (ou simplesmente uma *equivalência*) se é reflexiva, simétrica, e transitiva.

**Exemplo:** A relação = no conjunto **Nat** é uma relação de equivalência; outros exemplos de relações de equivalência são as relações de paralelismo entre retas, de semelhança de triângulos, de congruência módulo n. (Dois naturais x e y são *congruentes módulo n* se o resto da divisão de x por n é igual ao resto da divisão de y por n.)

□

***Composição de relações:*** definimos a composição de relações da forma a seguir: se $R \subseteq A \times B$ e $S \subseteq B \times C$ são relações, definimos a relação $R \circ S \subseteq A \times C$, a composição de R e S, por

$$R \circ S = \{ (x, z) \in A \times C \mid \exists\, y \in B, (x, y) \in R \text{ e } (y, z) \in S \}.$$

Se as relações R e S são funções, a composição R∘S se reduz exatamente à composição de funções: se $(x, y) \in R$ e $(y, z) \in S$, temos $y = R(x)$, $z = S(y) = S(R(x))$, e portanto $(R \circ S)(x) = S(R(x))$, como era de se esperar[1].

**Exemplo:** Sejam as relações

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

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

Temos:

$R \circ S = \{ (1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3) \}$

$S \circ R = \{ (2, 2), (2, 3), (2, 4), (3, 3), (3, 4), (4, 4) \}$

□

[1]Alguns autores preferem a ordem inversa: $(R \circ S)(x) = R(S(x))$. A diferença é apenas de notação.

**Exemplo:** Sejam as relações

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

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

Já que R e S são funções, o mesmo vale para as composições:

$R \circ S = \{ (1, 1), (2, 1), (3, 2), (4, 1) \}$

$S \circ R = \{ (1, 2), (2, 2), (3, 2), (4, 3) \}$

□

***Operações com relações binárias.*** Se R é uma relação binária num conjunto A (isto é, $R \subseteq A \times A$), podemos definir as potências $R^i$ de R, para $i \in$ **Nat** de forma recursiva:

$$R^0 = I_A = \{ (x, x) \mid x \in A \}$$

$$R^{i+1} = R^i \circ R, \text{ para } i \in \mathbf{Nat}$$

Fato:

1. A relação $I_A$ é a identidade para a composição de relações, associada ao conjunto A, ou seja, para qualquer $R \subseteq A^2$, $R \circ I_A = I_A \circ R = R$.
2. Para qualquer $R \subseteq A^2$, $R^1 = R$.
3. Para quaisquer $R \subseteq A^2$, $i, j \in$ **Nat**, $R^i \circ R^j = R^j \circ R^i$, ou seja, potências da mesma relação sempre comutam.

Dem.: Exercício.

□

**Exemplo:** Sejam $A = \{ 1, 2, 3, 4 \}$ e $R = \{ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) \}$.As potências de R são:

$R^0 = I = \{ (1,1), (2,2), (3,3), (4,4) \}$.

$R^1 = R = \{ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) \}$

$R^2 = R^1 \circ R = R \circ R = \{ (1,3), (1,4), (2,4) \}$

$R^3 = R^2 \circ R = \{ (1,4) \}$

$R^4 = R^5 = \ldots = \varnothing$.

No caso do exemplo, podemos provar que $(x, y) \in R$ se $y\text{-}x \geq 1$. Assim, em geral, $(x, y) \in R^i$ se $y\text{-}x \geq i$. Naturalmente, no conjunto A, a maior diferença possível é 3, e todas as potências além da terceira são relações vazias: nunca podem ser satisfeitas.

□

***Fechamento.*** Definimos o *fechamento reflexivo-transitivo* $R^*$ de uma relação binária R em um conjunto A através de

$x\ R^*\ y$ se e somente se para algum $i \in$ **Nat**, $x\ R^i\ y$,

ou, equivalentemente,

$$R^* = \bigcup_{i=0}^{\infty} R^i = R^0 \cup R^1 \cup R^2 \cup R^3 \cup \cdots$$

**Exemplo:** Seja a relação R, no conjunto **Nat** definida por

$x\ R\ y$ se e somente se $y = x + 1$.

Temos $x\ R^i\ y$ se e somente se $y = x + i$, de forma que $x\ R^*\ y$ se e somente $y \geq x$.

□

O nome de fechamento reflexivo-transitivo de R dado à relação R* se deve ao fato de que R* é a menor relação (no sentido da inclusão de conjuntos) que contém R e é reflexiva e transitiva. Ou seja, qualquer relação S

*(1)* que satisfaça x R y implica x S y (isto é, $S \supseteq R$) e
*(2)* que seja reflexiva e transitiva

satisfaz também $S \supseteq R^*$.

De forma semelhante, a notação $R^+$ é frequentemente utilizada para descrever o fechamento transitivo da relação R:

$$R^+ = \bigcup_{i=1}^{\infty} R^i = R^1 \cup R^2 \cup R^3 \cup \cdots$$

ou seja, x $R^+$ y se e somente se para algum $i>0$, x $R^i$ y.

**Exemplo:** Seja a mesma relação R do exemplo anterior. Neste caso, temos
x $R^+$ y se e somente se $y > x$.

□

***Partições.*** Dado um conjunto A, definimos uma *partição* de A como sendo uma família de conjuntos (chamados de *blocos da partição*) $\Pi = \{ B_i \mid i \in I \}$ com as seguintes propriedades:

(1) para cada $i \in I$, $B_i \neq \emptyset$. — *nenhum bloco é vazio*

(2) $\cup\Pi = \bigcup_{i \in I} B_i = A$ — *a união dos blocos é A*

(3) se $i \neq j$, $B_i \cap B_j = \emptyset$. — *blocos são disjuntos dois a dois*

Dessa maneira, cada elemento a de A pertence a exatamente um bloco da partição P.

**Observação:** Na maioria das vezes o conjunto I usado para indexar os elementos da família $\Pi$ será um conjunto enumerável, um subconjunto dos naturais.

**Exemplo:** Seja o conjunto A = { a, b, c, d, e }. Temos a seguir alguns exemplos de partições de A:

{ { a, b, c, d, e } }
{ { a }, { b }, { c }, { d }, { e } }
{ { a, b }, { c, d, e } }
{ { a, e }, { b, c, d } }

□

**Exercício:** Escreva todas as partições de { a, b, c, d, e }.

□

***Classes de equivalência.*** Seja R uma equivalência em um conjunto A. Definimos a classe de equivalência [*a*] de $a \in A$ da seguinte maneira:

$[a] = \{ x \in A \mid x \, R \, a \}$,

ou seja, a classe de equivalência de $a \in$ A é o conjunto dos elementos de A que são equivalentes a *a*. Note que como R é uma equivalência, $a \in [a]$, para qualquer *a*.

**Exemplo:** Seja a equivalência R em A = {a, b, c, d, e, f}, dada pelas seguintes propriedades:

(1) R é uma equivalência

(2) a R b, b R c, d R e.

(3) x R y somente se isto decorre de (1) e (2).

Temos então, examinando todos os casos possíveis:

| | |
|---|---|
| a R a, b R b, c R c, d R d, e R e, f R f | *(reflexividade)* |
| b R a, c R b, e R d | *(simetria)* |
| a R c, c R a | *(transitividade)* |

e R é composta dos pares: (a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c), (d, d), (d, e), (e, d), (e, e), (f, f).

Assim podemos ver diretamente que [a] = [b] = [c] = { a, b, c }, que [d] = [e] = { d, e } e que [f] = { f }.

□

***Conjunto quociente.*** Definimos o conjunto quociente A/R de A por uma equivalência R em A, através de

A/R = { [x] | x ∈ A },

ou seja, A/R é o conjunto das classes de equivalência de R em A.

**Exemplo:** Sejam A e R como no exemplo anterior. As classes de equivalência de R formam uma partição de A, que é exatamente o conjunto quociente A/R:

A/R = { { a, b, c }, { d, e }, { f } }

□

**Fato:** Seja R uma equivalência em um conjunto A. Então A/R é uma partição de A.
Dem.:

*(1)* note que as classes de equivalência não são vazias: à classe [*a*] pertence pelo menos o elemento *a*;

*(2)* a união das classes de equivalência é A, porque cada elemento *a* de A pertence a pelo menos uma classe de equivalência: $a \in [a]$.

*(3)* Classes de equivalência diferentes são disjuntas. Com efeito, suponha que duas classes [*a*] e [*b*] tem sua interseção não vazia, com um elemento *c* em comum: $c \in [a]$ e $c \in [b]$. Neste caso, usando o fato de que R é simétrica e transitiva, temos *c* R *a*, *c* R *b*, e, portanto, *a* R *b*. Assim, pela propriedade transitiva, *x* R *a* se e somente se *x* R *b*, e [*a*] = [*b*]. Consequentemente, as classes de equivalência são disjuntas duas a duas, e formam uma partição de A.

□

**Fato:** Dada uma partição P de um conjunto A, a relação R definida por

x R y se e somente se x e y fazem parte do mesmo bloco de P

é uma relação de equivalência em A, e A/R = P.
Dem.: Exercício.

□

***Indução finita.*** Muitas das demonstrações que veremos nas seções seguintes utilizam uma técnica conhecida por *indução finita*. A idéia fundamental é simples: suponha que desejamos provar que a propriedade P vale para todos os elementos de **Nat**, isto é, que queremos provar que, para todo $x \in$ **Nat**, P(x).

Uma propriedade fundamental de **Nat** é que **Nat** é composto por um elemento especial, 0, e por seus sucessores. Dito de outra forma, **Nat** é o menor conjunto que contém 0 e é fechado para a função *sucessor s*. Esquematicamente,

**Nat** = { 0, *s*(0), *s*(*s*(0)), *s*(*s*(*s*(0))), *s*(*s*(*s*(*s*(0))))… }.

Assim, se provarmos

I. *(base da indução)*

P(0)

II. *(passo de indução)*

Para qualquer $i \in$ **Nat**, P(i) implica P(s(i)).

estaremos provando P para todos os naturais, pois teremos

*(0)* P(0) (I)

*(1)* P(0) $\Rightarrow$ P(1) (II)

*(2)* P(1) $\Rightarrow$ P(2) (II)

*(3)* P(2) $\Rightarrow$ P(3) (II)

…

e, portanto, P(0), P(1), P(2), P(3), …

**Exemplo:** Suponhamos que queremos demonstrar a fórmula da soma dos elementos de uma progressão geométrica de razão $q \neq 1$,

$a_0 , a_1 , a_2 , a_3 , \ldots,$

com $a_{i+1} = a_i\, q$.

A fórmula da soma é

$$S_n = f(n) = \frac{(a_n q - a_0)}{(q-1)}$$

Devemos provar inicialmente a *base de indução* (para n=0): $S_0 = f(0)$. A demonstração se resume à verificação de que

$$f(0) = \frac{(a_n q - a_0)}{(q-1)} = a_0$$

Para provar o passo de indução, devemos assumir a *hipótese de indução* $S_i = f(i)$ e provar a *tese de indução* $S_{i+1} = f(i+1)$. Temos $a_{i+1} = a_i\, q$, e $S_{i+1} = S_i + a_{i+1}$.
Portanto,

$$S_{i+1} = S_i + a_{i+1} = f(i) + a_{i+1} = \frac{(a_i q - a_0)}{(q-1)} + a_{i+1} = \frac{(a_{i+1} - a_0)}{(q-1)} + a_{i+1} =$$
$$= \frac{(a_{i+1} - a_0 + a_{i+1} q - a_{i+1})}{(q-1)} = \frac{(a_{i+1} q - a_0)}{(q-1)} = f(i+1).$$

□

Uma forma alternativa de indução, que pode facilitar as demonstrações, em vez de usar apenas o último resultado anterior P(i) para provar P(i+1), usa todos os resultados anteriores, ou seja, P(0), P(1), …, P(i).

Assim, para mostrar P(i) para todos os naturais i, mostramos

I. P(0)

II. $\forall j \leq i\ P(j) \Rightarrow P(i+1)$.

***Indução em estrutura.*** Quando trabalhamos com estruturas que apresentam uma lei de formação bem definida, tais como cadeias, árvores, expressões, podemos usar para a indução um número natural, como, por exemplo, o tamanho da estrutura considerada; muitas vezes, entretanto, isso não é necessário, ou não é conveniente, e podemos fazer a indução de outra forma, baseada na própria estrutura.

Por exemplo, dados um conjunto I e uma propriedade Q, suponha um conjunto X definido como o menor conjunto, no sentido da inclusão, que satisfaz 1 e 2 a seguir:

1. todo $x \in I$ pertence a X, ou seja, $I \subseteq X$.
2. se $x \in X$ e Q(x,y), então $y \in X$.

Ou seja, um elemento x de X ou pertence a um conjunto inicial I, ou satisfaz a propriedade Q, que liga x a um (outro) elemento y de X. Para provarmos uma propriedade P(x) para todos os elementos de X, basta provar:

I. *(base da indução)*
se $x \in I$, P(x)

II. *(passo de indução)*
se $x \in X$, P(x) e Q(x,y), então P(y).

Este esquema pode ser generalizado para permitir várias propriedades Q, e para incluir a possibilidade que essas propriedades relacionem vários elementos de X a um (novo) elemento. Este caso mais geral de indução em estrutura está ilustrado a seguir.

**Exemplo:** Suponha que definimos uma expressão da seguinte maneira:

1. a, b, c são expressões.
2. Se $\alpha$ e $\beta$ são expressões, então $\alpha+\beta$ é uma expressão.
3. Se $\alpha$ e $\beta$ são expressões, então $\alpha*\beta$ é uma expressão.
4. Se $\alpha$ é uma expressão, $[\alpha]$ é uma expressão.

Suponha adicionalmente que queremos provar a propriedade: *"toda expressão tem comprimento (número de símbolos) ímpar"*. Vamos indicar *"$a$ tem comprimento ímpar"* por P($\alpha$). Devemos então, para provar *"para qualquer expressão $a$, $P(a)$"*, provar:

1. P(a), P(b), P(c).
2. Se P($\alpha$) e P($\beta$), então P($\alpha$+$\beta$).
3. Se P($\alpha$) e P($\beta$), então P($\alpha$*$\beta$).
4. Se P($\alpha$), então P([$\alpha$]).

Neste caso, (1) é a base da indução; (2)..(4) são passos de indução. Naturalmente, para mostrar (1), basta observar que

$|a| = |b| = |c| = 1$; $\alpha\beta$

para mostrar os demais, basta observar que

$|\alpha+\beta| = |\alpha| + |\beta| + 1$,
$|\alpha*\beta| = |\alpha| + |\beta| + 1$, e
$|[\alpha]| = |\alpha| + 2$.

□

*(revisão de 27fev97)*

# Capítulo 1: Alfabetos, cadeias, linguagens

***Símbolos e alfabetos.*** Um *alfabeto* é, para os nossos fins, um conjunto finito não vazio cujos elementos são chamados de símbolos. Dessa maneira, os conceitos de símbolo e alfabeto são introduzidos de forma interdependente: um alfabeto é um conjunto de símbolos, e um símbolo é um elemento qualquer de um alfabeto.

Note que consideramos aqui apenas *alfabetos finitos*: isso é feito por simplicidade, naturalmente, e também porque são raros os casos em que a consideração de um alfabeto infinito seria desejável.[1]

Qual o alfabeto que devemos considerar, ou seja, quais são os símbolos do alfabeto considerado depende do contexto em que pretendemos trabalhar. Como exemplos de alfabetos, citamos {0, 1} ou {*a*, *b*}, o alfabeto da língua portuguesa { a, b, c, ç, … z}, o conjunto de caracteres ASCII, etc.

Até certo ponto, podemos arbitrar os símbolos que nos interessam, e incluir apenas esses símbolos no alfabeto. Para cada aplicação específica, o usuário deve escolher o alfabeto que pretende utilizar: para exemplos no quadro negro, alfabetos como {0, 1}, e {a, b} são boas escolhas; para ensinar a linguagem Pascal, o alfabeto escolhido deverá conter símbolos como *program*, *begin*, *end*, *if*, *then*, *else*; para implementar a linguagem Pascal, provavelmente o alfabeto adequado não conterá símbolos como os vistos acima, mas sim caracteres como os do conjunto ASCII (letras, dígitos, +, *, etc.), uma vez que são estes os componentes básicos de um arquivo fonte.

***Cadeias.*** Formalmente, uma *cadeia de símbolos* em um alfabeto $\Sigma$ pode ser definida como uma função: uma sequência $s$ de comprimento $n$ no alfabeto $\Sigma$, é uma função $s{:}[n] \rightarrow \Sigma$, com domínio $[n]$, e com contradomínio $\Sigma$. O número natural $n$ é o *comprimento* de $s$, e é representado por $|s|$.

Por exemplo, se o alfabeto considerado for $\Sigma$ = {a, b, c}, a sequência de comprimento 4 (composta por quatro ocorrências de símbolos) $s$ = cbba pode ser vista como a função $s{:}[4] \rightarrow \Sigma$, definida por $s(1)$ = c, $s(2)$ = b, $s(3)$ = b, $s(4)$ = a.

***Concatenação.*** A principal operação sobre sequências é a operação de concatenação. Informalmente, o resultado da concatenação das sequências $x$ e $y$ é a sequência $xy$, ou $x \circ y$, composta pelos símbolos de $x$, seguidos pelos símbolos de $y$, nessa ordem. Mais formalmente, dadas duas sequências (funções) $x{:}[m] \rightarrow \Sigma$ e $y{:}[n] \rightarrow \Sigma$, de comprimentos $m$ e $n$, no mesmo alfabeto $\Sigma$, definimos a sequência (função) $x \circ y{:}[m+n] \rightarrow \Sigma$, de comprimento $m+n$, por

$$x \circ y(i) = \begin{cases} x(i), \text{ se } i \leq m \\ y(i-m), \text{ se } i > m \end{cases}$$

[1] Em geral, é possível usar alguma forma de *codificação*, e representar cada símbolo de um alfabeto infinito *enumerável* através de uma sequência de símbolos de um alfabeto finito.

Assim, se tivermos Σ = {a, b, c}, $x$ = cbba, e $y$ = ac, teremos $x \circ y$ = cbbaac. Representando as funções através de tabelas, temos:

| $i$ | $x(i)$ |
|---|---|
| 1 | c |
| 2 | b |
| 3 | b |
| 4 | a |

| $i$ | $y(i)$ |
|---|---|
| 1 | a |
| 2 | c |

| $i$ | $x \circ y(i)$ |
|---|---|
| 1 | c |
| 2 | b |
| 3 | b |
| 4 | a |
| 5 | a |
| 6 | c |

Naturalmente, $|x \circ y| = |x| + |y|$.

No que se segue, em geral não faremos referência ao fato de que sequências são funções. Se considerarmos símbolos $x_1$ , $x_2$ , … $x_n$ , de um alfabeto Σ, representaremos a sequência formada por ocorrências desses símbolos, nessa ordem, por $x_1x_2\ldots x_n$. Note que no caso especial $n$ = 1, a notação acima confunde a sequência a de comprimento 1 com o símbolo a. Esta ambiguidade não causa maiores problemas.

Um outro caso especial importante é o caso $n$ = 0, em que falamos da *sequência vazia*, indicada por ε. Usamos um nome "ε" para a sequência vazia simplesmente porque não é possível usar a mesma notação das outras sequências. Afinal, se escrevermos zero símbolos, como convencer alguém de que alguma coisa foi escrita? A sequência vazia ε é o elemento neutro (identidade) da concatenação: qualquer que seja a sequência x, temos

$$x \circ \varepsilon = \varepsilon \circ x = x$$

***Linguagens.*** Dado um alfabeto Γ, uma *linguagem* em Γ é um conjunto de sequências de símbolos de Γ.

O conjunto de todas as sequências de símbolos de um alfabeto Γ é uma linguagem, indicada por Γ*. A linguagem Γ* inclui todas as sequências de símbolos de Γ, incluindo também a sequência vazia ε. Com essa notação, uma linguagem L em Γ é um subconjunto de Γ*, ou seja, L ⊆ Γ*.

Note que todas essas sequências satisfazem a definição anterior, e tem como comprimento um número natural finito. Podemos assim ter linguagens infinitas, mesmo sem considerar sequências infinitas.

**Exemplo:** Os conjuntos a seguir são linguagens em Γ = {a, b}.

∅
{ε}
{a, aa, aaa}
{a, b}*
{x | |x| é par}
{a, b} — *notação ambígua*

**Nota:** Já observamos que a notação aqui usada é ambígua; essa ambiguidade se torna aparente neste último exemplo: {a, b} pode ser uma linguagem (conjunto de sequências) ou um alfabeto (conjunto de símbolos). Isto vale para qualquer alfabeto Γ. Se isso fosse necessário, a ambiguidade poderia ser evitada usando-se uma notação apropriada: em uma das notações possíveis, representamos *sequências entre aspas*, e *símbolos entre plicas*, de forma que "a" fica sendo a sequência, 'a' o símbolo, {"a", "b"} a linguagem e {'a', 'b'} o alfabeto. Para nós, entretanto, tais distinções não serão necessárias.

***Operações com linguagens.*** Linguagens são conjuntos, de forma que as operações de conjuntos podem ser diretamente usadas com linguagens. Assim, não há necessidade de definir *união*, *interseção* ou *diferença* de linguagens; no caso do *complemento*, podemos usar como *universo* o conjunto Γ* de todas as sequências no alfabeto considerado Γ.

Se L e M são linguagens em Γ, temos:

$$\begin{aligned} \textit{união:} \quad & L \cup M = \{ x \mid x \in L \text{ ou } x \in M \} \\ \textit{interseção:} \quad & L \cap M = \{ x \mid x \in L \text{ e } x \in M \} \\ \textit{diferença:} \quad & L - M = \{ x \mid x \in L \text{ e } x \notin M \} \\ \textit{complemento:} \quad & \overline{L} = \Gamma^* - L = \{ x \in \Gamma^* \mid x \notin L \} \end{aligned}$$

**Exemplo:** Seja o alfabeto Γ = {a, b, c} , e sejam as linguagens L = {a, bc, cb} e M = {aa, bb, cc, bc, cb} , em Γ. Temos:

$L \cup M = \{a, bc, cb, aa, bb, cc\}$
$L \cap M = \{bc, cb\}$
$L - M = \{a\}$
$M - L = \{aa, bb, cc\}$
$\overline{L} = \{ x \in \Gamma^* \mid x \neq a, x \neq bc \text{ e } x \neq cb \} =$
$= \{\varepsilon, b, aa, ab, ac, ba, bb, ca, cb, cc, aaa, \ldots \}$

□

***Concatenação de linguagens.*** A operação de concatenação, que foi definida para sequências, pode ser estendida a linguagens:

$$L \circ M = \{ x \circ y \mid x \in L \text{ e } y \in M \} .$$

**Exemplo:** Sendo L e M como no exemplo anterior,

$L \circ M = \{aaa, abb, acc, abc, acb, bcaa, bcbb, bccc, bcbc,$
$bccb, cbaa, cbbb, cbcc, cbbc, cbcb\}$
$M \circ L = \{aaa, aabc, aacb, bba, bbbc, bbcb, cca, ccbc,$
$cccb, bca, bcbc, bccb, cba, cbbc, cbcb\}$

□

**Fato:** A linguagem I = { ε } é o elemento neutro (identidade) da concatenação de linguagens, ou seja, para qualquer linguagem L,

$L \circ I = I \circ L = L.$

Dem.: Exercício.

□

***Potências.*** Podemos introduzir as potências $L^i$ (para *i* natural) de uma linguagem L através de uma definição recursiva:

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

$L^{i+1} = L \circ L^i$, para qualquer $i \in$ **Nat**.

**Exemplo:** Seja L = {0, 11}. Então temos:

$L^0 = \{ \varepsilon \}$
$L^1 = L \circ L^0 = \{ \varepsilon \} \circ \{0, 11\} = \{0, 11\}$
$L^2 = L \circ L^1 = \{0, 11\} \circ \{0, 11\} = \{00, 011, 110, 1111\}$
$L^3 = L \circ L^2 = \{0, 11\} \circ \{00, 011, 110, 1111\} =$
$= \{000, 0011, 0110, 01111, 1100, 11011, 11110, 111111\}$
e assim por diante.

□

**Fato:**

(1) Para qualquer linguagem L, $L^1 = L$.

(2) Para qualquer linguagem L, temos $L^i \circ L^j = L^{i+j}$ , para *i* e *j* quaisquer.

Dem.: Exercício.

□

***Fechamento.*** Podemos definir, para uma linguagem L qualquer, o seu fechamento L* como sendo a união de todas as potências de L:

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$$

Outra notação frequentemente utilizada é $L^+$, que indica a união de todas as potências de L, excluída a potência 0:

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \ldots$$

**Exemplo:** Para a linguagem L do exemplo anterior, temos:

$L^* = \{ \varepsilon, 0, 11, 00, 011, 110, 1111, 000, 0011, 0110,$
$01111, 1100, 11011, 11110, 111111, \ldots\}$

$L^+ = \{ 0, 11, 00, 011, 110, 1111, 000, 0011, 0110,$
$01111, 1100, 11011, 11110, 111111, \ldots\}$

**Exercício:** Caracterize a classe das linguagens L para as quais $L^* = L^+$.

□

**Fato:** Para qualquer alfabeto Γ, o conjunto Γ* de todas as sequências de símbolos de Γ é enumerável. (Note que Γ pode ser considerado um alfabeto ou uma linguagem, sem que isso altere o valor de Γ*.)

Dem. Considere (por exemplo) a seguinte enumeração:

1. Escolha uma ordem qualquer (ordem alfabética) para os elementos do alfabeto Γ.
2. Enumere as sequências por ordem crescente de comprimento, e, dentro de cada comprimento, por ordem alfabética. Por exemplo, se Γ = {a, b, c}, a enumeração pode ser

   ε, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, …

Observamos que a ordem alfabética simples não seria adequada para as cadeias, uma vez que teríamos, neste caso,

ε, a, aa, aaa, aaaa, aaaaa, ….

excluindo, portanto, da enumeração, todas as sequências que não pertencessem a $\{a\}^*$ Assim, a cadeia b, por exemplo, *nunca seria atingida*.

O esquema apresentado se baseia no fato de que, para cada comprimento, o número de sequências de comprimento n é finito. Assim, se Γ pudesse ser infinito, a enumeração não seria possível, como descrita, uma vez que não teríamos um número finito de sequências para cada comprimento.

□

No que se segue, usaremos a notação $x_i$ para representar a *i*-ésima sequência numa enumeração de Γ*, para um alfabeto Γ fixo, supondo uma ordenação fixada para Γ*.

**Fato:** Qualquer linguagem é enumerável.

Dem.: Toda linguagem em Γ é um subconjunto de Γ*.

□

**Fato:** A classe de todas as linguagens em um alfabeto Γ não é enumerável.

Dem.: A classe de todas as linguagens em Γ é P(Γ*), e já vimos que o conjunto formado por todos os subconjuntos de um conjunto enumerável infinito não é enumerável.

□

**Exercício:** Aplique a técnica da diagonalização diretamente, para mostrar que a classe P(Γ*) de todas as linguagens no alfabeto Γ não é enumerável.

□

**Exercício:** Suponha Γ = { a, b, c }.

(a) Escreva um programa que, quando recebe como entrada o natural i, determina a cadeia $x_i \in \Gamma^*$.

(b) Escreva um programa que, quando recebe como entrada uma cadeia $x \in \Gamma^*$, determina o natural i tal que $x = x_i$.

rev. 17/6/96

# Capítulo 2: Procedimentos e algoritmos

Para estudar o processo de computação de um ponto de vista teórico, com a finalidade de caracterizar o que é ou não é computável, é necessário introduzir um modelo matemático que represente o que se entende por computação. Entretanto, cada estudioso do assunto tem seu modelo matemático favorito, e por isso há diversas definições essencialmente distintas. (Por exemplo, um programador FORTRAN tenderia certamente a afirmar que computável é exatamente aquilo que pode ser feito por um programa FORTRAN).

Diversos modelos foram apresentados, e podem ser estudados na literatura (veja, por exemplo, [BrLa74][1] ). Neste curso veremos apenas dois destes modelos: as funções recursivas parciais e as máquinas de Turing. O modelo das funções recursivas parciais se baseia em idéias semelhantes às da programação funcional, enquanto o modelo das máquinas de Turing procura introduzir um computador elementar, com repertório de instruções e estrutura de memória com a maior simplicidade possível. Outros modelos, que não veremos aqui, são baseados em algoritmos de Markov (com algumas semelhanças com a linguagem SNOBOL), em linguagens de programação mais convencionais (como o FORTRAN citado acima), ou em outras arquiteturas de computadores ideais ou linguagens de programação.

O fato surpreendente a respeito disso é que (até hoje) todos os modelos usados (se não são completamente absurdos) concordam na definição do que quer dizer *"computável"*. Uma conjectura, enunciada por Alonzo Church, diz que *todos os modelos razoáveis do processo de computação, definidos e por definir, são equivalentes.* Essa conjectura é conhecida como a *tese de Church.* Por sua própria natureza, a tese de Church não admite nenhuma prova formal, mas até hoje todos os modelos propostos se mostraram equivalentes.

Neste capítulo, entretanto, não queremos ainda definir modelos matemáticos precisos, mas queremos apenas trabalhar com a idéia intuitiva do que quer dizer computável. Para isso, vamos introduzir, mais abaixo, *informalmente*, dois conceitos relacionados: o conceito de *procedimento*, e o conceito de *algoritmo*.

***Codificação:*** Na discussão que se segue, levaremos em consideração apenas conjuntos enumeráveis. A razão para isto é que nenhum modelo "razoável" proposto para o processo de computação admite o tratamento de conjuntos não enumeráveis. Precisamos ter alguma garantia de que podemos representar de forma finita todos os elementos do conjunto considerado, e isso não seria possível no caso de conjuntos não enumeráveis. Por exemplo, o conjunto dos números reais não é enumerável, mas o conjunto dos números racionais é enumerável, e, por essa razão, o tratamento computacional de números reais é feito através de aproximações racionais.

Adicionalmente, sem perda de generalidade, é conveniente considerar que todos os conjuntos considerados ou são conjuntos de números naturais (subconjuntos de **Nat**) ou são linguagens em algum alfabeto $\Gamma$ conhecido (subconjuntos de $\Gamma^*$). Sabemos que $\Gamma^*$, **Nat** e seus subconjuntos são enumeráveis, e que existe uma bijeção entre quaisquer

[1] *Walter S. Brainerd, Lawrence H. Landweber*, Theory of Computation, *John Wiley, 1974*

dois conjuntos enumeráveis infinitos. Isso significa que podemos, usando essa bijeção como codificação, usar valores em um conjunto enumerável qualquer para substituir valores em outro. Em particular, a codificação através de números naturais (*numeração*), e a descrição através de cadeias de símbolos (muitas vezes conhecidos como identificadores) são idéias familiares.

***Procedimentos.*** Vamos definir um *procedimento* como sendo uma sequência *finita* de instruções, e definir *instrução* como uma operação *claramente descrita*, que pode ser executada *mecanicamente*, em *tempo finito*.

- *"mecanicamente"* quer dizer que não há dúvidas sobre o que deve ser feito;

- *"em tempo finito"* quer dizer que não há dúvidas de que a tarefa correspondente à instrução pode, em qualquer caso, ser levada até sua conclusão.

Para descrever um procedimento podemos usar uma linguagem natural, uma linguagem de programação, ou a linguagem normalmente usada em matemática. Frequentemente, usamos uma mistura de todas estas. Sobre a forma de descrever instruções e procedimentos, suporemos apenas que existe uma linguagem, comum a todos os interessados, em que instruções e procedimentos podem ser descritos sem ambiguidades.

Como exemplos, citamos:

- o algoritmo de Euclides para cálculo do máximo divisor comum de dois números naturais;
- um programa em FORTRAN que calcula a soma de dois números;
- a fórmula que calcula as raízes da equação do segundo grau.

**Exemplo:** O procedimento a seguir pára e diz *"sim"* se o número inteiro i, dado como entrada, for par e não negativo.

1. se i = 0, pare e diga *"sim"*.
2. diminua o valor de i de duas unidades.
3. vá para 1.

Note que se i for originalmente ímpar ou negativo, o procedimento não pára. Na definição de procedimento, dada acima, exigimos que cada instrução possa ser executada em tempo finito, mas não exigimos nada de semelhante para os procedimentos. Isso significa, em particular, que não poderíamos usar como instrução em algum outro procedimento uma chamada do procedimento do exemplo acima, uma vez que não podemos garantir sua parada em tempo finito, para qualquer valor da entrada.

□

**Exemplo:** O procedimento a seguir pára e diz *"sim"* se o número inteiro i for par e não negativo; pára e diz *"não"* nos demais casos.

1. se i = 0, pare e diga *"sim"*.
2. se i < 0, pare e diga *"não"*.
3. diminua o valor de i de duas unidades
4. vá para 1.

□

***Algoritmo.*** Definimos um algoritmo como sendo um procedimento que sempre pára, quaisquer que sejam os valores de suas entradas.

O segundo exemplo de procedimento dado acima é, portanto, um algoritmo. Programas corretos normalmente são algoritmos; em geral, um programa que não pára para alguns valores de suas entradas é um programa incorreto, um "programa com loop", um "programa com erro de lógica". Em alguns casos, entretanto, isso não é verdade, e o desejado é, exatamente, que a execução do programa se estenda por um tempo ilimitado. O exemplo mais característico desse tipo de procedimento parece ser o de um sistema operacional: uma vez iniciada sua execução, ela continua (em condições normais) até que a máquina seja desligada.

A seguir temos outro exemplo de procedimento que não tem parada prevista:

**Exemplo:** O procedimento a seguir enumera as sequências de {a, b}*, emitindo (imprimindo, por exemplo) todas essas sequências.

1. Faça X = {ε}.
2. Emita todas as sequências de X.
3. Para cada $y \in$ X, acrescente a X as sequências *y*a e *y*b.
4. Vá para 2.

O procedimento acima emite todas as sequências formadas por a's e b's.

ε, ε, a, b, ε, a, b, aa, ab, ba, bb, ε, a, b, …

Cada sequência é emitida uma infinidade de vezes, como permitido numa enumeração.

□

**Exemplo:** Seja uma função f: **Nat** → **Nat**. Suporemos que está disponível um algoritmo para calcular f(i), a partir de qualquer i ∈ **Nat**. Considere o procedimento a seguir:

1. faça i = 0.
2. calcule j = f(i), usando o algoritmo dado
3. se j = k, emita i, e pare
4. incremente o valor de i de 1
5. vá para 2.

Note que o passo 2 só pode ser considerado uma instrução por causa da disponibilidade de um *algoritmo* para cálculo dos valores da função. O procedimento do exemplo aceita como entrada um valor k ∈ **Nat**, e só pára se existir um valor de i tal que f(i) = k. Em particular, o valor de i emitido é o menor possível.

□

**Conjuntos recursivamente enumeráveis.** Dizemos que um conjunto A é recursivamente enumerável (r.e.) se existe um procedimento que enumera os elementos

de A. Isto quer dizer que existe um procedimento que emite todos os elementos de A, possivelmente com repetições.

**Exemplo:** Em um dos exemplos anteriores vimos que existe um procedimento que enumera as sequências pertencentes ao conjunto A = { a, b }$^*$. Logo, o conjunto A é recursivamente enumerável.

□

**Exemplo:** Seja f: **Nat** → **Nat**. O procedimento a seguir mostra que o contradomínio de f é recursivamente enumerável, supondo que existe um algoritmo que calcula o valor de f(i), a partir de i.

1. i := 0;
2. emita f(i);
3. i := i + 1;
4. vá para 2.

□

***Conjuntos recursivos.*** Dizemos que um conjunto A é *recursivo* se existe um algoritmo que determina, para um valor arbitrário de sua entrada x, se x ∈ A ou se x ∉ A. Embora isso não seja estritamente necessário, podemos, para fixar as idéias, especificar que o algoritmo deve parar e dizer *"sim"* ou *"não"*, respondendo à pergunta "x ∈ A?".

**Exemplo:** O conjunto dos naturais pares é recursivo. Basta examinar o algoritmo a seguir, cuja entrada é um natural i:

1. Se i = 0, pare e diga *"Sim"*.
2. Se i = 1, pare e diga *"Não"*.
3. faça i := i - 2.
4. vá para 1.

□

**Fato:** Todo conjunto recursivo é recursivamente enumerável.

Dem.: Se o conjunto A ⊆ **Nat** é recursivo, existe um algoritmo α que, para cada entrada i determina se i ∈ A. Considere o procedimento a seguir:

1. i := 0.
2. execute α com entrada i.
3. se α respondeu *"Sim"*, emita i.
4. i := i + 1.
5. vá para 2.

Note que:

- i assume todos os valores naturais
- α sempre pára, qualquer que seja sua entrada i;
- α responde "Sim" exatamente para os valores de i que pertencem a A.

Portanto, os valores de i emitidos são exatamente aqueles pertencentes a A.

□

**Fato:** A classe dos conjuntos recursivos é fechada para as operações de união, interseção e complementação.
Dem.: *(união)* Sejam A e B conjuntos recursivos. Sejam $\alpha$ e $\beta$ algoritmos que determinam pertinência em A e em B, respectivamente. Podemos construir um algoritmo $\gamma$ que determina se $x \in A \cup B$, da seguinte forma:

1. execute $\alpha$ com entrada x.
2. se $\alpha$ respondeu *"Sim"*, responda *"Sim"* e pare.
3. execute $\beta$ com entrada x.
4. pare e responda o que $\beta$ respondeu.

*(interseção)* Sejam A, B, $\alpha$, $\beta$ como acima. Construa um algoritmo $\gamma$ que determina se $x \in A \cap B$ da seguinte forma:

1. execute $\alpha$ com entrada x.
2. se $\alpha$ respondeu *"Não"*, responda *"Não"* e pare.
3. execute $\beta$ com entrada x.
4. pare e responda o que $\beta$ respondeu.

*(complemento)* Sejam A e $\alpha$ como acima. Construa um algoritmo $\gamma$ que determina se $x \in \overline{A}$ = **Nat** - A da seguinte forma:

1. execute $\alpha$ com entrada x.
2. se $\alpha$ respondeu *"Sim"*, responda *"Não"* e pare.
3. se $\alpha$ respondeu *"Não"*, responda *"Sim"* e pare.

□

**Fato:** Um conjunto A é recursivamente enumerável se e somente se existe um procedimento que, com entrada x, pára e diz *"Sim"*, se $x \in A$, o que não acontece se $x \notin A$. Isto quer dizer que, se $x \notin A$, ou *(1)* o procedimento não pára, ou *(2)* pára, mas não diz *"Sim"*.

Dem.: ($\Rightarrow$) Se A é r.e., existe um procedimento $\alpha$ que enumera seus elementos. Construa um procedimento $\beta$ que aceita um elemento x qualquer como entrada, modificando $\alpha$ da seguinte maneira: quando $\alpha$ emite um valor y, $\beta$ testa se y = x. Se isso acontecer, $\beta$ pára e diz *"Sim"*. Portanto, $\beta$ dirá *"Sim"* exatamente quando sua entrada for emitida por $\alpha$, ou seja quando for um elemento de A.

($\Leftarrow$) Seja $\alpha$ um procedimento que pára e diz *"Sim"* quando sua entrada $x \in A$, e que ou não pára, ou não diz *"Sim"*, quando $x \notin A$. Um procedimento $\beta$ que enumera os elementos de A pode ser construído usando $\alpha$, da seguinte maneira:

1. faça k = 0;
2. para cada x = 0, …, k execute (2.1) e (2.2).
    2.1. execute um passo (adicional) de $\alpha$ com entrada x
    2.2. se $\alpha$ parou e disse *"Sim"*, emita x.
3. faça k = k + 1.
4. vá para 2.

Note que é necessário executar "em paralelo" $\alpha$ com as várias entradas x porque, se os diversos valores de x fossem tentados sequencialmente, e $\alpha$ não parasse para algum valor de x, $\alpha$ nunca chegaria a ser executado com os valores subsequentes de x.

□

**Fato:** A classe dos conjuntos recursivamente enumeráveis é fechada para as operações de união e de interseção.

Dem.: *(união)* Sejam A e B conjuntos r.e. e sejam $\alpha$ e $\beta$ procedimentos que enumeram os elementos de A e de B, respectivamente. Um procedimento *g* que enumera os elementos de $A \cup B$ executa $\alpha$ e $\beta$ em paralelo é:

1. execute um passo de $\alpha$.
2. se $\alpha$ emitiu i, emita i.
3. execute um passo de $\beta$.
4. se $\beta$ emitiu i, emita i.
5. vá para 1.

Os elementos emitidos por $\gamma$ são exatamente os elementos de A e os elementos de B, ou seja, os elementos de $A \cup B$.

*(interseção)* Sejam A, B, $\alpha$ e $\beta$ como acima. O procedimento $\gamma$ que enumera os elementos de $A \cap B$ executa em paralelo $\alpha$ e $\beta$, guardando os elementos já emitidos por $\alpha$ e por $\beta$ nos conjuntos X e Y, respectivamente:

0. Faça $X = \varnothing$ e $Y = \varnothing$.
1. execute um passo de $\alpha$.
2. se $\alpha$ emitiu i, acrescente i ao conjunto X.
3. execute um passo de $\beta$.
4. se $\beta$ emitiu i, acrescente i ao conjunto Y.
5. emita os elementos comuns a X e a Y.
6. vá para 1.

Os elementos emitidos por $\gamma$ são aqueles pertencentes aos conjuntos *finitos* X e Y, ou sejam aqueles já emitidos por $\alpha$ e por $\beta$.

□

**Fato:** Se um conjunto A e seu complemento $\overline{A}$ são ambos recursivamente enumeráveis, então A (e o complemento $\overline{A}$) são ambos recursivos.

Dem.: Já vimos que se A é r.e., existe um procedimento $\alpha$ que pára e diz *"Sim"*, exclusivamente quando sua entrada x é um elemento de A; como $\overline{A}$ também é r.e., existe um procedimento $\beta$ que pára e diz *"Sim"* exclusivamente quando sua entrada x é um elemento de $\overline{A}$, ou seja, exatamente quando x *não é* um elemento de A.

Podemos construir um algoritmo $\gamma$ que executa os procedimentos $\alpha$ e $\beta$ em paralelo, com entrada x, para determinar se x pertence ou não a A. O procedimento a seguir é um algoritmo porque eventualmente um dos dois passos (2) ou (4) será executado: ou $x \in A$ ou $x \notin A$, não havendo terceira possibilidade.

1. execute um passo (adicional) de $\alpha$ com entrada x.
2. se $\alpha$ parou e disse *"Sim"*, pare e diga *"Sim"*.
3. execute um passo (adicional) de $\beta$ com entrada x.
4. se $\beta$ parou e disse *"Sim"*, pare e diga *"Não"*.
5. vá para 1.

□

*(revisão de 27fev97)*

# Capítulo 3: Gramáticas

Já vimos que procedimentos podem ser usados para definir linguagens de duas maneiras essenciais: como *geradores*, procedimentos que enumeram os elementos da linguagem, e como *reconhecedores* (ou *aceitadores*), procedimentos que indicam quando uma sequência faz parte da linguagem. O tipo mais comum de gerador é a *gramática*. A idéia original de gramática vem do estudo de linguagens naturais, e as definições que apresentaremos aqui são essencialmente devidas ao linguista Noam Chomsky.

Fundamentalmente, uma gramática é composta por *regras de produção*, ou *regras de re-escrita*, através das quais é possível obter todos os elementos da linguagem a partir de um símbolo inicial, usando as regras para re-escrever (produzir) os elementos. Formalmente, definimos uma *gramática* G como sendo uma construção $<N, \Sigma, P, S>$, onde

- N é um alfabeto de símbolos auxiliares, chamados de símbolos não terminais, ou, simplesmente, de *nãoterminais*.
- $\Sigma$ é o alfabeto no qual a linguagem é definida, cujos elementos são os símbolos terminais, ou, simplesmente, *terminais*.
- P é o conjunto de regras de re-escrita, chamadas simplesmente de *regras* ou *produções*.
- S é o *símbolo inicial*.

Vamos definir o *vocabulário* de G, como sendo $V = N \cup \Sigma$, o alfabeto composto pelos símbolos terminais e não terminais. O conjunto de regras P é uma relação binária no conjunto $V^*$ de cadeias de símbolos quaisquer (terminais ou nãoterminais), isto é, $P \subseteq V^* \times V^*$, correspondendo cada regra individual a um par de cadeias $(\alpha, \beta)$. Entretanto, em vez de $(\alpha, \beta) \in P$, a notação habitual para a regra que permite a re-escrita de $\alpha$ como $\beta$ é simplesmente $\alpha \rightarrow \beta$. Além disso, reunimos regras com o mesmo lado esquerdo $\alpha$, tais como $\alpha \rightarrow \beta_1$, $\alpha \rightarrow \beta_2$, …, $\alpha \rightarrow \beta_n$, na abreviação

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

**Nota:** Em geral se procura usar as seguintes convenções:

| letras | representam |
|---|---|
| S, A, B, C, … | nãoterminais |
| *a*, *b*, *c*, … | terminais |
| X, Y, Z, … | símbolos quaisquer |
| $\alpha$, $\beta$, $\gamma$, … | cadeias quaisquer |
| x, y, z, … | cadeias de terminais |
| V, N, $\Sigma$, $\Gamma$, $\Delta$, … | alfabetos |

Como acontece com toda convenção, haverá alguns casos em que esta não será seguida.

□

**Nota:** Alguns autores preferem definir P como um subconjunto de $V^* N V^* \times V^*$, ou seja, exigem que, numa regra $\alpha \rightarrow \beta$, o lado esquerdo $\alpha$ contenha pelo menos um símbolo nãoterminal. A diferença entre as duas definições é irrelevante.

**Exemplo:** Vamos definir uma gramática

$$G = < N, \Sigma, P, S > = < \{ S \}, \{ 0, 1 \}, \{ (S, 0\ S\ 1), (S, \varepsilon) \}, S >$$

onde $N = \{ S \}$ é o conjunto de nãoterminais, $\Sigma = \{ 0, 1 \}$ é o conjunto de terminais, e $P = \{ (S, 0\ S\ 1), (S, \varepsilon) \} = \{ S \rightarrow 0S1, S \rightarrow \varepsilon \} = \{ S \rightarrow 0S1 \mid \varepsilon \}$ é o conjunto de regras.

Para mostrar que a cadeia 000111 faz parte da linguagem associada à gramática, seguimos, a partir de S, os seguintes passos intermediários:

S, 0S1, 00S11, 000S111, 000111.

Assim, por três vezes S é substituído por 0S1, e finalmente, S é substituído pela sequência vazia $\varepsilon$. Como veremos a seguir, aplicar uma regra $\alpha \rightarrow \varepsilon$ é equivalente a simplesmente remover $\alpha$.

□

***Relação $\Rightarrow$ (deriva em um passo)***. Podemos definir a aplicação de uma regra através de uma relação: dizemos que $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$ se e somente se $\alpha \rightarrow \beta$. Caso seja necessário distinguir entre várias gramáticas, podemos usar o símbolo $\Rightarrow_G$ para indicar explicitamente qual a gramática G utilizada.

Quando consideramos vários passos de uma derivação, podemos usar as definições de operações com relações vistas no capítulo 0, e indicar $\Rightarrow^i$ (*deriva em i passos*), $\Rightarrow^*$ (*deriva em zero ou mais passos*, ou, simplesmente, *deriva*), $\Rightarrow^+$ (*deriva em um ou mais passos*).

Uma sequência $\alpha$ derivável de S, (tal que $S \Rightarrow^* \alpha$) é chamada uma *forma sentencial* da gramática considerada.

**Exemplo:** Em referência à gramática do exemplo anterior, podemos escrever

$$S \Rightarrow 0S1,\ 0S1 \Rightarrow 00S11,\ 00S11 \Rightarrow 000S111,\ 000S111 \Rightarrow 000111$$

ou, de forma mais compacta,

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$$

Podemos também escrever $S \Rightarrow^4 000111$, e $S \Rightarrow^* 000111$. As cadeias S, 0S1, 00S11, 000S111, e 000111 são formas sentenciais da gramática. Dessas, a mais importante é 000111, composta *exclusivamente por terminais*, que fará parte da *linguagem da gramática*, de acordo com a próxima definição.

***Linguagem de uma gramática.*** Definimos a linguagem da gramática G = <N, Σ, P, S> por

$$L(G) = \{ x \in \Sigma^* \mid S \Rightarrow^* x \}.$$

Ou seja, a linguagem da gramática G é constituída pelas sequências x, que são compostas apenas de símbolos terminais e que podem ser obtidas em um número arbitrário de passos de derivação a partir do símbolo inicial S, usando as regras de P.

**Exemplo:** Seja G a gramática do exemplo anterior. Vamos mostrar que

$$L(G) = \{ 0^i1^i \mid i \in \mathbf{Nat} \}.$$

Para mostrar a igualdade dos dois conjuntos é necessário mostrar a *"dupla inclusão"* , que no caso se reduz a mostrar *(a)* e *(b)* a seguir:

*(a)* cada sequência da forma $0^i1^i$ pode ser derivada a partir de S, ou seja, $L(G) \supseteq \{ 0^i1^i \mid i \in \mathbf{Nat} \}$.

Basta observar que uma derivação a partir de S, em que se utiliza i vezes a regra $S \rightarrow 0S1$ e uma vez a regra $S \rightarrow \varepsilon$, gera exatamente $0^i\ 1^i$ .

*(b)* uma sequência de terminais derivada de S tem a forma $0^i1^i$, ou seja, $L(G) \subseteq \{ 0^i1^i \mid i \in \mathbf{Nat} \}$.

Basta notar que a partir de S, usando as duas regras de todas as maneiras possíveis, só é possível gerar sequências (formas sentenciais) da forma $0^i1^i$ , e da forma $0^iS1^i$ , e que cadeias da segunda forma não podem fazer parte da linguagem, por conter um símbolo não terminal.

□

***Gramáticas equivalentes.*** Duas gramáticas $G_1$ e $G_2$ são chamadas de equivalentes, se ambas definem a mesma linguagem: $L(G_1) = L(G_2)$.

**Exemplo:** As duas gramáticas abaixo são equivalentes:

$$G_1 = < \{S, T\}, \{a, b\}, \{S \rightarrow aTa \mid bTb, T \rightarrow aT \mid bT \mid \varepsilon\}, S>$$

$$G_2 = < \{S, A, B\}, \{a, b\}, \{S \rightarrow aA \mid bB, A \rightarrow aA \mid bA \mid a, B \rightarrow aB \mid bB \mid b\}, S>$$

Para mostrar a equivalência destas duas gramáticas, basta apenas mostrar que, de forma diferente, ambas geram *exatamente* as cadeias cujo primeiro símbolo é igual ao último. Isso pode ser feito observando que:

- em $G_1$, o primeiro e o último símbolos são introduzidos na mesma regra;
- em $G_2$, os nãoterminais A e B *"lembram"* qual *"foi"* o primeiro símbolo, e determinam qual deve ser o último símbolo.

A demonstração completa deve ser feita por indução, e fica como exercício para o leitor interessado mostrar que

$$L(G_1) = L(G_2) = \{ cxc \mid c \in \{a,b\} \text{ e } x \in \{a,b\}^* \}.$$

Por exemplo, a cadeia aabba é da linguagem, e pode ser derivada como a seguir:

em $G_1$ : $S \Rightarrow aTa \Rightarrow aaTa \Rightarrow aabTa \Rightarrow aabbTa \Rightarrow aabba$

em $G_2$ : $S \Rightarrow aA \Rightarrow aaA \Rightarrow aabA \Rightarrow aabbA \Rightarrow aabba$

□

***A hierarquia das gramáticas.*** Chomsky definiu quatro *"tipos"* de gramáticas, 0, 1, 2 e 3, que formam uma hierarquia: cada gramática de um tipo é também dos tipos menores. Nossa definição não é exatamente a mesma usada por Chomsky, mas é praticamente equivalente.

*Gramáticas tipo 0* (*gramáticas sem restrição*). Exatamente as gramáticas vistas na definição anterior. As regras de uma gramática tipo 0 são regras da forma $\alpha \rightarrow \beta$, com $\alpha$ e $\beta$ quaisquer.

*Gramáticas tipo 1* (*gramáticas sensíveis ao contexto*, ou *gsc*). As gramáticas tipo 1 são as gramáticas com regras da forma $\alpha \rightarrow \beta$, em que se exige $| \alpha | \leq | \beta |$; é entretanto permitida uma regra que viola esta restrição: uma gramática tipo 1 pode ter a regra $S \rightarrow \varepsilon$, se S não aparece do lado direito de nenhuma regra.

*Gramáticas tipo 2* (*gramáticas livres de contexto*, ou *glc*). As gramáticas tipo 2 são as gramáticas com regras da forma $A \rightarrow \beta$, onde A é um símbolo não terminal, e $\beta$ é uma sequência qualquer de V*, possivelmente vazia.

*Gramáticas tipo 3 (gramáticas regulares).* As gramáticas tipo 3 só podem ter regras dos três tipos descritos a seguir:

| | |
|---|---|
| $A \rightarrow a\ B$, | onde A e B são nãoterminais, e *a* é um terminal; |
| $A \rightarrow a$, | onde A é um nãoterminal, e *a* é um terminal; |
| $A \rightarrow \varepsilon$, | onde A é um nãoterminal. |

Se uma linguagem tem uma gramática tipo 0, ela é uma *linguagem tipo 0*; se tem uma gramática tipo 1, ela é uma *linguagem tipo 1*, ou uma *linguagem sensível ao contexto* (*lsc*); se tem uma gramática tipo 2, ela é uma *linguagem tipo 2*, ou uma *linguagem livre de contexto* (*llc*); se tem uma gramática tipo 3, ela é uma *linguagem tipo 3*, ou uma *linguagem regular*.

**Observações:**

- As gramáticas tipo 3 são chamadas regulares pela simplicidade da estrutura de suas linguagens, garantida pelos rígidos formatos de suas regras.
- As gramáticas tipo 2 são chamadas de livres de contexto porque uma regra $A \rightarrow \beta$ indica que o nãoterminal A, *independentemente do contexto* em que estiver inserido, pode ser substituído por $\beta$.
- Finalmente, as gramáticas tipo 1 são chamadas de sensíveis ao contexto por permitirem regras da forma $\alpha\ A\ \gamma \rightarrow \alpha\ \beta\ \gamma$: A pode ser reescrito como $\beta$, *dependendo do contexto* em que A aparece ($\alpha$ à esquerda, $\gamma$ à direita).
- O caso especial da regra $S \rightarrow \varepsilon$, nas gramáticas sensíveis ao contexto tem uma única finalidade: permitir que a cadeia vazia $\varepsilon$ pertença a algumas linguagens sensíveis ao contexto. Com efeito, a aplicação de uma regra $\alpha \rightarrow \beta$ em que $| \alpha | \leq | \beta |$ não pode diminuir o comprimento da sequência à qual é aplicada, porque temos sempre $| \gamma\ \alpha\ \delta | \leq | \gamma\ \beta\ \delta |$. Como $| S | = 1$, e $| \varepsilon | = 0$, há necessidade de alguma regra que permita a diminuição do comprimento, para que uma derivação $S \Rightarrow^* \varepsilon$ seja possível, em algumas gramáticas sensíveis ao contexto.
- *As definições de gramáticas acima <u>não</u> formam exatamente uma hierarquia*. Certamente, uma gramática tipo 3 é sempre tipo 2, e uma gramática tipo 1 é

sempre tipo 0, mas nem todas as gramáticas tipo 2 são tipo 1. Isso acontece porque as regras da forma $A \rightarrow \varepsilon$ não satisfazem a restrição de comprimento, pois $|A| > |\varepsilon|$, já que $1 > 0$. Na última seção deste capítulo, entretanto, mostraremos um resultado que é suficiente para nossos propósitos: qualquer glc pode ser transformada, através de um algoritmo relativamente simples, em uma gramática que satisfaz simultaneamente as definições de glc e de gsc, e que, além disso, é equivalente à gramática original. Com esse resultado, vemos que, ao contrário das classes de gramáticas, as classes de linguagens definidas acima formam uma hierarquia. Ou seja, toda linguagem regular é uma llc, toda llc é uma lsc, e toda lsc é uma linguagem tipo 0.

Seguem-se alguns exemplos de gramáticas e linguagens de diversas classes. Naturalmente, é mais fácil mostrar que uma gramática pertence a uma certa classe, do que mostrar o resultado oposto: num caso, basta exibir uma gramática apropriada; no outro, é necessário mostrar que nenhuma gramática é apropriada. Algumas técnicas para obter esses resultados negativos serão apresentadas posteriormente, durante o decorrer do curso.

**Exemplo:** A linguagem $L = \{ cxc \mid c \in \{a, b\} \text{ e } x \in \{a, b\}^* \}$ do exemplo anterior é livre de contexto, uma vez que $G_1$ é uma glc. Por outro lado, como $G_2$ é uma gramática regular, L também é regular. Como toda linguagem regular é livre de contexto, em geral não se faz referência ao fato de que a linguagem é livre de contexto.

□

**Exemplo:** A linguagem $L = \{ xx^R \mid x \in \{0,1\}^* \}$ é livre de contexto. (A notação $x^R$ indica a sequência *reversa* ou *refletida* de x: se $x = a_1\, a_2 \ldots a_n$ , $x^R = a_n \ldots a_2\, a_1$.) Basta considerar a gramática com regras

$$S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$$

onde S é o único nãoterminal, e portanto é também o nãoterminal inicial. Para derivar 00100100, por exemplo, teríamos:

$$S \Rightarrow 0S0 \Rightarrow 00S00 \Rightarrow 001S100 \Rightarrow 0010S0100 \Rightarrow 00100100$$

A linguagem L *não* é regular, mas ainda não dispomos dos mecanismos apropriados para provar este fato.

□

**Exemplo:** A linguagem $L = \{ x\, x \mid x \in \{0,1\} \}$ é sensível ao contexto. Uma gramática para esta linguagem tem as regras indicadas a seguir:

| | | | |
|---|---|---|---|
| 1. $S \rightarrow 0ZS$ | 5. $Z0 \rightarrow 0Z$ | 9. $ZZ' \rightarrow Z'0$ | 13. $Z' \rightarrow 0$ |
| 2. $S \rightarrow 1US$ | 6. $Z1 \rightarrow 1Z$ | 10. $ZU' \rightarrow Z'1$ | 14. $U' \rightarrow 1$ |
| 3. $S \rightarrow 0Z'$ | 7. $U0 \rightarrow 0U$ | 11. $UZ' \rightarrow U'0$ | |
| 4. $S \rightarrow 1U'$ | 8. $U1 \rightarrow 1U$ | 12. $UU' \rightarrow U1$ | |

A utilização das regras é explicada a seguir, usando-se como exemplo a derivação de 00100010, na qual as regras estão indicadas como índices inferiores.

- *1 a 4:* regras usadas para gerar os símbolos das duas metades de xx. Os símbolos da segunda metade são indicados por Z (zero: 0) e U (um: 1). U' ou Z' indica o último símbolo da segunda metade.

$S \Rightarrow_1 0ZS \Rightarrow_1 0Z0ZS \Rightarrow_2 0Z0Z1US \Rightarrow_3 0Z0Z1U0Z'$

- *5 a 8:* regras usadas para levar todos os símbolos da segunda metade para depois dos símbolos da primeira metade:

  $0Z0Z1U0Z' \Rightarrow_7 0Z0Z10UZ' \Rightarrow_6 0Z01Z0UZ' \Rightarrow_5 0Z010ZUZ' \Rightarrow_5 00Z10ZUZ' \Rightarrow_6 001Z0ZUZ' \Rightarrow_5 0010ZZUZ'$

- *9 a 12:* regras usadas para converter os símbolos da segunda metade. O símbolo Z' (ou U') indica o último símbolo ainda não convertido.

  $0010ZZUZ' \Rightarrow_{11} 0010ZZU'0 \Rightarrow_{10} 0010ZZ'10 \Rightarrow_9 0010Z'010$

- *13 a 14:* regras usadas para converter o primeiro símbolo da segunda metade

  $0010Z'010 \Rightarrow_{13} 00100010$

Pela definição de linguagem de uma gramática, só nos interessam as sequências de terminais que são obtidas a partir do símbolo inicial S, usando as regras de produção da gramática. Por essa razão, a ordem em que as regras são aplicadas não é importante. Uma outra derivação possível para 00100010, em que as mesmas regras são usadas em uma ordem diferente é

$S \Rightarrow_1 0ZS \Rightarrow_1 0Z0ZS_5 \Rightarrow 00ZZS \Rightarrow_2 00ZZ1US \Rightarrow_6 00Z1ZUS \Rightarrow_6 001ZZUS \Rightarrow_3 001ZZU0Z' \Rightarrow_7 001ZZ0UZ' \Rightarrow_5 001Z0ZUZ' \Rightarrow_5 0010ZZUZ' \Rightarrow_{11} 0010ZZU'0 \Rightarrow_{10} 0010ZZ'10 \Rightarrow_9 0010Z'010 \Rightarrow_{13} 00100010$

Outro fato sem importância é a existência de "becos sem saída", isto é, sequências deriváveis de S que não levam a nenhuma sequência de terminais, não contribuindo, portanto, para a linguagem. Por exemplo, considere a derivação:

$S \Rightarrow_1 0ZS \Rightarrow_1 0Z0ZS \Rightarrow_2 0Z0Z1US \Rightarrow_3 0Z0Z1U0Z' \Rightarrow_{13} 0Z0Z1U00$

Note que a regra 13 foi aplicada "precocemente", de forma que as ocorrências restantes dos símbolos Z e U *não podem mais ser convertidas* em ocorrências de 0 e 1. Entretanto, como nenhuma sequência de terminais pode ser obtida a partir de 0Z0Z1U00, o fato de que 0Z0Z1U00 pode ser derivada a partir de S não introduz na linguagem nenhum elemento indevido. (0Z0Z1U00 é apenas uma forma sentencial — inútil — da gramática.)

A linguagem L *não* é livre de contexto, mas ainda não dispomos dos mecanismos para provar este fato.

□

Uma das principais motivações da definição de gsc vem do teorema abaixo: a restrição nos comprimentos dos lados esquerdo e direito das regras tem exatamente a finalidade de garantir a recursividade das linguagens sensíveis ao contexto. Entretanto, observamos que a recíproca do teorema não é verdadeira, ou seja existem linguagens recursivas que não são sensíveis ao contexto. Este último fato será demonstrado posteriormente.

**Teorema:** Toda lsc é um conjunto recursivo.

Dem.: Sejam G = <N, Σ, P, S> uma gsc, e L = L(G). Devemos, pela definição de conjunto recursivo, mostrar que existe um algoritmo que, quando recebe como entrada

uma cadeia arbitrária $x \in \Sigma^*$, indica se é ou não verdade que $x \in L$, parando e emitindo respectivamente *"sim"* ou *"não"*.

Sabemos que uma gsc pode ter uma regra $S \rightarrow \varepsilon$, se S não aparecer à direita em nenhuma regra. Para cada uma das outras regras, o comprimento do lado esquerdo não pode ser maior que o do lado direito, e cada aplicação da regra pode manter ou aumentar o comprimento, mas nunca diminuí-lo. Portanto, para considerar todas as derivações que podem levar a uma sequência x, basta considerar as formas sentenciais de comprimento menor ou igual ao de x: se $x \neq \varepsilon$, $S \Rightarrow^* \gamma \Rightarrow^* x$ implica que $1 \leq |\gamma| \leq |x|$. Esta idéia está aplicada no algoritmo a seguir:

1. Se $x \neq \varepsilon$, vá para 4.
2. Se P contém uma regra $S \rightarrow \varepsilon$, pare e diga *"sim"*.
3. Pare e diga *"não"*.
4. Faça $X = \{ S \}$, e $Y = \emptyset$.
5. Se $x \in X$, pare e diga *"sim"*
6. Se $X = \emptyset$, pare e diga *"não"*
7. Escolha uma cadeia qualquer $\alpha$ em X, retire $\alpha$ de X, e acrescente $\alpha$ a Y
8. Para cada $\beta$ tal que $\alpha \Rightarrow \beta$, se $\beta \notin Y$, e se $|\beta| \leq |x|$, acrescente $\beta$ a X.
9. vá para 5

O tratamento da entrada $x = \varepsilon$ se resume a verificar se existe a regra correspondente $S \rightarrow \varepsilon$; para as demais entradas, todas as possibilidades são consideradas, e uma forma sentencial só é eliminada se já foi considerada antes (pertence a Y), ou se é longa demais (comprimento maior que o de x). Durante a execução do algoritmo, X guarda as formas sentenciais que ainda devem ser consideradas. Note que se trata de um algoritmo: em cada execução do passo 7, é escolhida uma cadeia $\alpha$ de comprimento menor ou igual ao de x, e não são permitidas repetições.

□

**Exemplo:** Considere a gsc G com nãoterminais S e T, terminais a, b e c, símbolo inicial S e regras

$S \rightarrow aSa \mid bSb \mid T$

$T \rightarrow cT \mid c$

(Note que G é *também* uma glc.) Seja x = aabbb. Para determinar se $x \in L(G)$, são consideradas as formas sentenciais abaixo, que compõem o conjunto Y, ao final da execução:

{ S, aSa, bSb, T, aaSaa, abSba, aTa, baSab, bbSbb, bTb, cT, c, aaTaa, abTba,
acTa, aca, baTab, bbTbb, bcTb, bcb, ccT, cc, aacaa, abcba, accTa, acca,
bacab, bbcbb, bccTb, bccb, cccT, ccc, accca, bcccb, ccccT, cccc, ccccc }

Fica como exercício a execução passo a passo do algoritmo, mostrando que $x \notin L(G)$.

□

**Teorema:** Toda linguagem tipo 0 é um conjunto recursivamente enumerável.

Dem.: Seja $G = \langle N, \Sigma, P, S \rangle$ uma gramática (tipo 0). O procedimento abaixo enumera os elementos de L(G).

1. Faça X = { S }
2. Emita todos os elementos $\alpha \in X \cap \Sigma^*$.
3. Faça Y = { $\beta \mid \alpha \in X$ e $\alpha \Rightarrow \beta$ }
4. Faça X = Y
5. vá para 2.

A cada iteração do procedimento acima, X contém todas as sequências deriváveis de S em um passo adicional. Assim, se $S \Rightarrow^* x$ em n passos, em n passos o procedimento emitirá x.

□

Outros resultados sobre as classes de linguagens ainda dependem de resultados que serão apresentados posteriormente. Em particular, mostraremos posteriormente que todo conjunto r. e. é uma linguagem tipo 0, e apresentaremos os exemplos que mostram que as demais inclusões entre as classes de linguagens são próprias.

***Gramáticas livres de contexto, gramáticas sensíveis ao contexto, e regras com lado direito vazio.*** Retomamos aqui o problema das regras com lado direito vazio em glc's e gsc's. Como já vimos anteriormente, a definição de glc permite indiscriminadamente a presença de regras $A \rightarrow \varepsilon$, onde A é um nãoterminal; essas regras podem violar a restrição da definição de gsc, uma vez que a única regra permitida com lado direito vazio é uma regra $S \rightarrow \varepsilon$, desde que o símbolo inicial S não apareça à direita em nenhuma regra.

Dessa maneira, não é imediato que toda llc é uma lsc, uma vez que uma glc pode não satisfazer a definição de gsc.

Esta seção tem como principal finalidade mostrar que *cada glc pode ser transformada em outra glc equivalente, que satisfaz a definição de gsc*. Como corolário desse fato, temos que *toda llc é também uma lsc*.

***Nãoterminais anuláveis.*** Dizemos que um nãoterminal A de uma glc G é anulável se é possível derivar a cadeia vazia $\varepsilon$ a partir de A, ou seja, se $A \Rightarrow^* \varepsilon$.

**Lema:** Dada uma gramática livre de contexto G = <N, Σ, P, S>, o algoritmo a seguir determina quais são os nãoterminais anuláveis de G, isto é, quais são os nãoterminais A a partir dos quais pode ser derivada a cadeia vazia $\varepsilon$.

Dem.: Considere o algoritmo:

1. Faça X = { $A \in N \mid A \rightarrow \varepsilon \in P$ }
2. Repita o passo 3 até que nenhum elemento novo possa ser acrescentado a X:
3. Para cada regra $B \rightarrow X_1 X_2 \ldots X_n$, se todos os símbolos $X_1, X_2, \ldots X_n$ pertencem a X, acrescente B a X.

Claramente, ao final da execução do algoritmo, X contém exatamente os nãoterminais anuláveis da gramática considerada.

□

**Teorema:** Para qualquer gramática livre de contexto G existe uma gramática livre de contexto G', equivalente a G, que satisfaz a definição de gramática sensível ao contexto.

Dem.: Seja G = <N, Σ, P, S>. Execute o algoritmo anterior para determinar quais os nãoterminais anuláveis de G. Seja S' um símbolo novo, não pertencente a N ou a Σ.

Defina a gramática G' = <N ∪ {S'}, Σ, P', S'>, sendo P' como obtido como descrito a seguir:

1. Inicialmente, faça P' = P.
2. Repita o passo 3 enquanto for possível acrescentar regras novas a P'.
3. Para cada regra $A \rightarrow \alpha B \beta$ de P', se B é anulável, acrescente $A \rightarrow \alpha \beta$ a P'.
4. Retire de P' todas as regras com lado direito vazio.
5. Acrescente a P' a regra $S' \rightarrow S$.
6. Se S é anulável, acrescente a P', além da regra acima, a regra $S' \rightarrow \varepsilon$.
   (Se acrescentada, esta será a única regra com lado direito vazio.)

Claramente, a gramática G' satisfaz as definições vistas de glc e de gsc. Note que, como S' é um símbolo novo, não ocorre à direita em nenhuma das regras de G'.

Resta assim provar que G e G' são equivalentes. Para isto, observamos que o uso das regras-ε em G corresponde ao uso de regras mais curtas em G': em vez de usar uma regra de G, que introduz (ocorrências de) nãoterminais anuláveis, e depois usar as regras necessárias para a transformação desses nãoterminais anuláveis em ε, usamos de saída uma regra mais curta de G', em que as correspondentes ocorrências desses nãoterminais anuláveis não aparecem (ver passo 3 do algoritmo acima). Na outra direção, similarmente, o uso de uma regra mais curta pode ser substituído pelo uso da regra mais longa de que se originou, seguida pelo uso das regras necessárias para transformar os nãoterminais que se deseja anular na sequência vazia.

□

**Exemplo:** Considere a gramática G, com o conjunto de regras P a seguir:

$S \rightarrow ABC \mid ABD$
$A \rightarrow aA \mid \varepsilon$
$B \rightarrow Bb \mid AC$
$C \rightarrow CC \mid c \mid \varepsilon$
$D \rightarrow d$

Para a aplicação do algoritmo acima, temos sucessivamente os seguintes valores para o conjunto X:

| | |
|---|---|
| { A, C } | regras $A \rightarrow \varepsilon$ e $C \rightarrow \varepsilon$ |
| { A, B, C } | regra $B \rightarrow AC$ |
| { A, B, C, S } | regra $S \rightarrow ABC$ |

até que X atinge seu valor final: X = { S, A, B, C }.

Na construção do conjunto P' de regras da gramática G', temos inicialmente P' = P:

$S \rightarrow ABC \mid ABD$
$A \rightarrow aA \mid \varepsilon$
$B \rightarrow Bb \mid AC$
$C \rightarrow CC \mid c \mid \varepsilon$
$D \rightarrow d$

Retirando as ocorrências de nãoterminais anuláveis, as seguintes regras são acrescentadas:

$S \rightarrow BC \mid AC \mid AB \mid BD \mid AD \mid C \mid B \mid A \mid D \mid \varepsilon$
$A \rightarrow a$
$B \rightarrow b$
$B \rightarrow A \mid C \mid \varepsilon$
$C \rightarrow C$

As regras com lado direito vazio são então retiradas, as regras $S' \rightarrow S$, e $S' \rightarrow \varepsilon$ são acrescentadas, já que $S \in X$. Ao final, temos as seguintes regras em P':

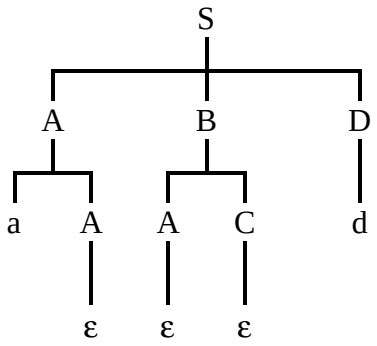
$S' \rightarrow S \mid \varepsilon$
$S \rightarrow ABC \mid ABD \mid BC \mid AC \mid AB \mid BD \mid AD \mid C \mid B \mid A \mid D$
$A \rightarrow aA \mid a$
$B \rightarrow Bb \mid AC \mid b \mid A \mid C$
$C \rightarrow CC \mid c \mid C$
$D \rightarrow d$

(A regra $C \rightarrow C$ é obviamente inútil.)

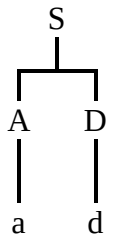
Podemos agora comparar uma derivação em G com a correspondente derivação em G'. Considere a seguinte derivação em G:

$S \Rightarrow ABD \Rightarrow aABD \Rightarrow aBD \Rightarrow aACD \Rightarrow aCD \Rightarrow aD \Rightarrow ad$

A aplicação de $S \rightarrow ABD$ introduz uma ocorrência de B que é depois anulada; o mesmo acontece com a aplicação da regra $A \rightarrow aA$. As regras correspondentes em G' são, portanto, $S \rightarrow AD$ e $A \rightarrow a$. As duas derivações correspondentes, em G e G', podem ser representadas pelas "árvores de derivação" a seguir.




árvore de derivação em G




árvore de derivação em G'

□

**Corolário:** Toda glc cuja linguagem não contém a sequência vazia pode ser transformada em uma glc equivalente que não tem regras com lado direito vazio.

Dem.: Se o símbolo inicial da gramática não é anulável, a construção vista na demonstração do teorema acima nos leva a uma gramática sem nenhuma regra-ε.

□

**Exemplo:** Seja a gramática G com regras

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ ) \mid a$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T' \rightarrow *\ F\ T' \mid \varepsilon$

De acordo com o algoritmo visto na demonstração do teorema anterior, o conjunto dos nãoterminais anuláveis é X = {E', T'}. Assim, a nova gramática G' tem regras

$S' \rightarrow E$
$E \rightarrow T\ E' \mid T$
$T \rightarrow F\ T' \mid F$
$F \rightarrow (\ E\ ) \mid a$
$E' \rightarrow +\ T\ E' \mid +\ T$
$T' \rightarrow *\ F\ T' \mid *\ F$

A conversão de gramáticas livres de contexto em gramáticas sem regras vazias perdeu parte de sua importância, porque os algoritmos de análise sintática usados em compiladores modernos (ao contrário de algoritmos mais antigos) não fazem restrições ao uso de regras com lado direito vazio. Portanto, regras com lado direito vazio podem ser usadas sem problemas nas gramáticas de linguagens de programação, usadas para projeto e implementação de compiladores.

(*revisão de 27fev97)*

# Capítulo 4: Autômatos finitos e expressões regulares

– com Luiz Carlos Castro Guedes

## 4.1 - Introdução

Neste capítulo estudaremos uma máquina (um procedimento aceitador, ou reconhecedor), chamada *autômato finito (af)*. A palavra *finito* é incluída no nome para ressaltar que um af só pode conter uma quantidade finita e limitada de informação, a qualquer momento. Essa informação é representada por um estado da máquina, e só existe um número finito de estados.

Essa restrição faz com que o af seja severamente limitado na classe de linguagens que pode reconhecer, composta apenas pelas linguagens regulares, como mostraremos neste capítulo.

Duas versões do af são estudadas aqui: o *af determinístico (afd)* e o *af não determinístico (afnd).* Este capítulo mostra que uma linguagem regular pode ser definida de quatro formas:

- através de uma gramática regular (definição);
- através de um afd que reconhece a linguagem;
- idem, através de um afnd;
- através de uma *expressão regular*, um mecanismo a ser introduzido com essa expressa finalidade.

## 4.2 - Autômato finito determinístico

Como observado acima, a informação que um af guarda sobre a entrada (mais precisamente sobre a parte da entrada já lida) é representada por um *estado*, escolhido em um conjunto finito de estados. A definição formal de automato finito, na sua versão determinística é dada a seguir.

*Definição.* Um *Autômato Finito Determinístico (afd)* M, sobre um alfabeto $\Sigma$ é um sistema $(K, \Sigma, \delta, i, F)$, onde

$K$ é um *conjunto de estados* finito, não vazio;
$\Sigma$ é um *alfabeto de entrada* (finito)
$\delta: K\times\Sigma \rightarrow K$ é a *função de transição*
$i \in K$ é o *estado inicial*
$F \subseteq K$ é o conjunto de *estados finais*.

O nome *determinístico* faz referência ao fato de que $\delta$ é uma função (também chamada *função próximo-estado*), que determina precisamente o próximo estado a ser assumido quando a máquina M se encontra no estado q e lê da entrada o símbolo a: o estado $\delta(q, a)$.

De forma simplificada, podemos dizer que um afd aceita uma cadeia se, partindo do *estado inicial*, e mudando de estado de acordo com a *função de transição*, o afd atinge um *estado final* ao terminar de ler a cadeia. Uma das maneiras de visualizar o funcionamento de um afd é através de um *controle finito* que lê símbolos de uma *fita* de entrada (onde se encontra a cadeia de entrada), sequencialmente, da esquerda para a direita. Os elementos do conjunto de estados K representam os estados possíveis do controle finito. A operação se inicia no estado inicial i, lendo o primeiro símbolo da fita de entrada. Por conveniência, considera-se que a cabeça de leitura se move sobre a fita, ao contrário do que seria de se esperar.

A Figura 4.1 representa um afd cujo controle está no estado q, e que está lendo o quarto símbolo da cadeia de entrada, um b.


Fig. 4.1 - Autômato Finito

**Exemplo 1:** Considere o afd M = (K, Σ, δ, i, F), onde temos

K = { q0, q1, q2, q3 }
Σ = { a, b }
i = q0
F = { q3 }

e onde a função de transição δ: { q0, q1, q2, q3 } × { a, b } → { q0, q1, q2, q3 } é dada pela tabela abaixo

| δ | a | b |
|---|---|---|
| q0 | q1 | q2 |
| q1 | q0 | q3 |
| q2 | q3 | q0 |
| q3 | q2 | q1 |

Alternativamente, podemos representar o afd M por um *diagrama de transições*, ou *diagrama de estados*, como o da Fig. 4.2. Note que o diagrama de transições determina completamente o automato M, através de algumas convenções:

- os estados são os nós do grafo, ou seja, K = { q0, q1, q2, q3 };
- o estado inicial é indicado pela seta, ou seja, i = q0;
- os estado finais são indicados pelo círculo duplo: q3 é o único estado final, ou seja, F = { q3 };

- as transições são as indicadas pelas arestas: δ(q0, a) = q1, δ(q0, b) = q2, δ(q1, a) = q0, etc, ou seja, δ é a mesma função representada pela tabela acima.

Cada estado de um af corresponde a uma determinada informação sobre a parte da cadeia de entrada já lida. No caso do exemplo, a informação pode ser descrita em frases curtas, mas isso nem sempre acontece. Para o estado q2, por exemplo, podemos dizer

"se o estado atingido é q2,
o número de símbolos a já lidos é par, e
o número de símbolos b já lidos é ímpar".

(Note que 0 é par.)

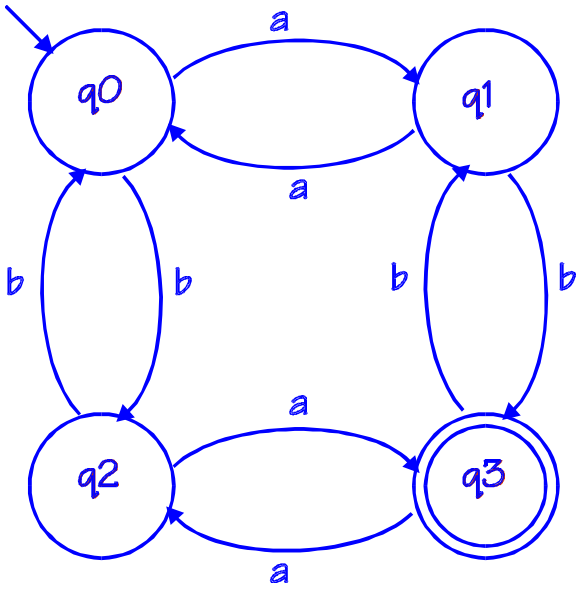



**Fig. 4.2 - afd para o Exemplo 1**

Em resumo, temos:

| | número de a's | número de b's |
|---|---|---|
| q0 | par | par |
| q1 | ímpar | par |
| q2 | par | ímpar |
| q3 | ímpar | ímpar |

A linguagem aceita ou reconhecida por M (ver definição abaixo) é a linguagem formada pelas cadeias em que os números de a's e de b's são ambos ímpares. Isso se deve ao fato de que o único estado final é q3.

Por exemplo, a cadeia abaa é da linguagem de M, porque, com essa cadeia, os seguintes estados são atingidos: q0, q1, q3, q2, q3. Como o último estado é final, a cadeia é aceita.

ð

*A linguagem de um afd.* Para definir a linguagem L(M), a linguagem das cadeias aceitas ou reconhecidas pelo afd M, podemos definir inicialmente uma *configuração* de M como sendo um par $(q, x) \in K \times \Sigma^*$, composto pelo estado corrente (o estado atual da máquina) e pela cadeia x, a parte da entrada que *ainda não foi lida*. Como observado, o

estado representa a parte já lida da cadeia de entrada. A configuração (i, x) é a *configuração inicial* de M para a cadeia x; qualquer configuração (q, ε) é uma *configuração final* se q ∈ F. A mudança de configuração é caracterizada pela relação |—, definida abaixo:

(q, ax) |— (p, x) se e somente se $\delta(q, a) = p$

ou seja, se tivermos $\delta(q, a) = p$, e se M estiver no estado q, lendo um símbolo a da cadeia de entrada, M moverá a cabeça de leitura uma posição para a direita e irá para o estado p. O símbolo a, depois de lido, não precisa mais aparecer na configuração. Podemos agora definir a linguagem L(M) por

$L(M) = \{ x \in \Sigma^* \mid (i, x) \vdash^* (f, \varepsilon), \text{ com } f \in F \}$

Como em casos anteriores, |—* indica o fechamento reflexivo-transitivo da relação, no caso a relação |— de mudança de configuração, indicando que a configuração final pode ser atingida em zero ou mais passos.

**Exemplo 1:** (continuação) Para mostrar que abaa ∈ L(M), basta observar que

(q0, abaa) |— (q1, baa) |— (q3, aa) |— (q2, a) |— (q3, ε),

porque q3 é final. Por outro lado, como

(q0, abab) |— (q1,bab) |— (q3, ab) |— (q2, b) |— (q0, ε),

abab não pertence a L(M).

ð

Uma caracterização alternativa de L(M) pode ser baseada em uma extensão $\hat{d}$ da função δ, feita de forma a aceitar cadeias no segundo argumento, isto é com domínio $K \times \Sigma^*$ em vez de $K \times \Sigma$. Pode-se definir a nova função $\hat{d} : K \times \Sigma^* \to K$ por

$$\hat{d}(q, e) = q, \quad \forall q \in K$$

$$\hat{d}(q, ax) = \hat{d}(d(q, a), x), \quad \forall q \in K, \forall x \in \Sigma^*, \forall a \in \Sigma.$$

**Fato:** A função $\hat{d}$ é realmente uma *extensão* de δ, isto é, sempre que δ é definida, $\hat{d}$ também é, e tem o mesmo valor. Ou seja, se q ∈ K e a ∈ Σ, $\hat{d}(q, a) = \delta(q, a)$.

Dem. Exercício.

**Fato:** A função $\hat{d}$ e a relação |— se relacionam por

$\hat{d}(q, x) = p$ se e somente se $(q, x) \vdash^* (p, \varepsilon)$

Portanto, temos

$L(M) = \{x \in \Sigma^* \mid \hat{d}(q, x) \in F \}$

Demonstração: Exercício.

**Exemplo 1:** (continuação) Para mostrar que abaa ∈ L(M), basta ver que

$\hat{d}$ (q0, abaa) = $\hat{d}$ (q1, baa) = $\hat{d}$ (q3, aa) = $\hat{d}$ (q2, a) = $\hat{d}$ (q3, ε) = q3 ∈ F

Como $\hat{d}$ (q0, abab) = q0 ∉ F, abab ∉ L(M).

**Exercício 1:** Mostre que o afd M do Exemplo 1 aceita a linguagem

L(M) = { x ∈ {a, b}* | os números de a's e de b's em x são ímpares }

Sugestão: indução no comprimento de x.

**Exercício 2:** Mostre que a definição anterior de $\hat{d}$ pode ser substituída pela equivalente

$$\hat{d}(q, e) = q, \quad \forall q \in K$$

$$\hat{d}(q, xa) = d(\hat{d}(q, x), a), \quad \forall q \in K, \forall x \in \Sigma^*, \forall a \in \Sigma.$$

**Exercício 3:** Modifique a definição do af M do Exemplo 1, fazendo F= { q1, q2 } Descreva a linguagem aceita por M assim modificado.

**Exercício 4:** Descreva a linguagem do afd M dado pelo diagrama de estados da Fig. 4.3.

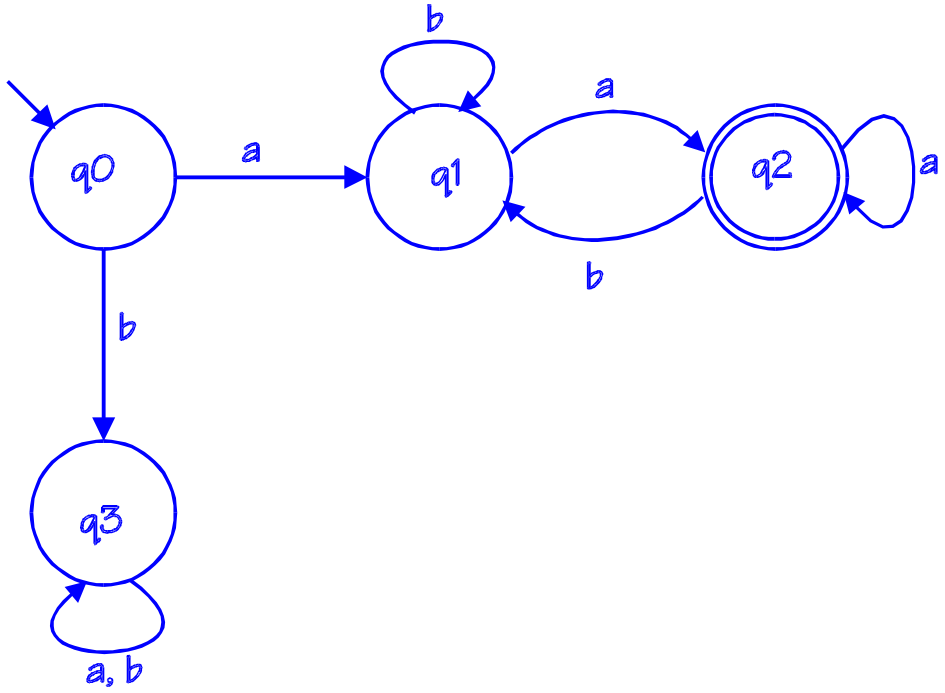



**Fig. 4.3 - afd para o Exercício 4**

**Exercício 5:** Descreva a linguagem do afd M = (K, Σ, δ, i, F), onde K={q0, q1, q2, q3}, Σ = { a, b }, i = q0, F = { q2 }, e δ dada pela tabela abaixo:

| δ | a | b |
|---|---|---|
| q0 | q1 | q3 |
| q1 | q2 | q0 |
| q2 | q3 | q1 |
| q3 | q4 | q2 |

ð

## 4.3 - Autômato finito não determinístico

Passaremos agora ao estudo do af não determinístico. Em oposição ao que acontece com o afd, a função de transição de um afnd não precisa determinar exatamente qual deve ser o próximo estado. Em vez disso, a função de transição fornece uma lista (um conjunto) de estados para os quais a transição poderia ser feita. Essa lista pode ser vazia, ou ter um número qualquer positivo de elementos.

Essa possibilidade de escolha entre vários caminhos a serem seguidos nos leva a modificar a definição de aceitação. Um afd aceita se "o último estado atingido é final"; mas um afnd aceita se "*existe uma sequência de escolhas tal que* o último estado atingido é final". Podemos alternativamente imaginar que o afnd "escolhe", "adivinha", o caminho certo para a aceitação, uma vez que a existência de escolhas erradas, que não levam a um estado final, é irrelevante.

**Exemplo 2:** Considere o afnd dado pelo diagrama da Fig. 4.4 e a cadeia de entrada ababa.

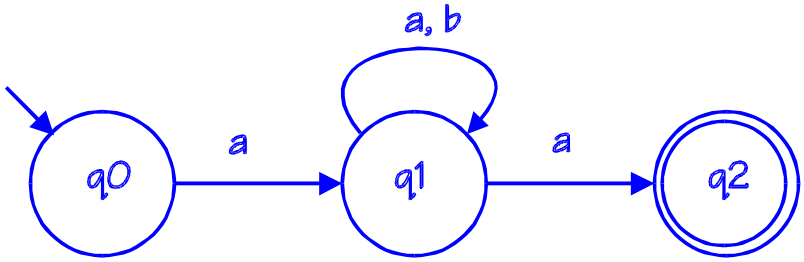



Fig. 4.4 - afnd para o Exemplo 2

A cadeia ababa é aceita, porque uma das possibilidades é a sequência de estados q0, q1, q1, q1, q1, q2. Naturalmente, com a mesma cadeia, poderíamos escolher a sequência q0, q1, q1, q1, q1, q1, que não leva a um estado final. Ou a sequência q0, q1, q1, q2, interrompida, porque q2 não prevê uma transição com o segundo b. Mas estes casos em que "o automato adivinhou errado" não criam problemas para a aceitação, porque "existe um caminho certo".

Este afnd aceita a linguagem das cadeias (de comprimento maior ou igual a 2), cujo primeiro e último símbolos são a, sendo os restantes quaisquer. (Compare este afnd com o afd de um dos exemplos anteriores, que aceita a mesma linguagem.)

ð

*Definição.* Formalmente, um *Autômato Finito não Determinístico (afnd)* M, sobre um alfabeto $\Sigma$ é um sistema $(K, \Sigma, \delta, i, F)$, onde

$K$ é um conjunto (finito, não vazio) de estados,
$\Sigma$ é um *alfabeto de entrada* (finito),
$\delta: K\times(\Sigma \cup \{ \varepsilon \}) \rightarrow P(K)$ é a função de transição,
$i \in K$ é o *estado inicial*,
$F \subseteq K$ é o *conjunto de estados finais*.

A notação P(K) indica o conjunto "partes" de K (conjunto potência de K, ou, ainda, "powerset" de K), o conjunto de todos os subconjuntos de K.

Pela definição, portanto, $\delta$ é uma função que aceita como argumentos q e a, onde q é um estado e a pode ser um símbolo de $\Sigma$ ou a cadeia vazia $\varepsilon$. Em qualquer caso, $\delta(q, a)$ é sempre um conjunto de estados, ou seja, um subconjunto de K.

Se tivermos $\delta(q, a) = \{p_1, p_2, ..., p_k\}$, entendemos que o autômato M, a partir do estado q, pode escolher um dos estados $p_1, p_2, ..., p_k$ para ser o próximo estado. Se $a = \varepsilon$, nenhum símbolo da entrada é lido; se $a \neq \varepsilon$, o símbolo a da entrada é lido. Podemos considerar o caso $a=\varepsilon$ como correspondendo a transições espontâneas: M muda de estado sem estímulo da entrada. Se tivermos $\delta(q, a) = \emptyset$, não há transições possíveis a partir do estado q com o símbolo a.

Definimos configurações para o caso do afnd da mesma forma que anteriormente. A mudança de configuração é dada pela relação |—, definida abaixo:

$(q, ax) \vdash (p, x)$ se e somente se $p \in \delta(q, a)$

Note que a pode ser a cadeia vazia, caso em que temos, particularizando,

$(q, x) \vdash (p, x)$ se e somente se $p \in \delta(q, \varepsilon)$

Podemos agora definir a linguagem L(M) por

$$L(M) = \{ x \in \Sigma^* \mid (i, x) \vdash^* (f, \varepsilon), \text{ com } f \in F \}$$

**Exemplo 2:** (continuação) Temos, para a mesma cadeia ababa de entrada,

$(q0, ababa) \vdash (q1, baba) \vdash (q1, aba) \vdash (q1, ba) \vdash (q1, a) \vdash (q2, \varepsilon)$

e, portanto, $ababa \in L(M)$. Temos também o "caminho errado"

$(q0, ababa) \vdash (q1, baba) \vdash (q1, aba) \vdash (q1, ba) \vdash (q1, a) \vdash (q1, \varepsilon)$

que leva à configuração não final $(q1, \varepsilon)$, e não permite nenhuma conclusão.

Cadeias como bab e abab não levam a configurações finais e não são aceitas. Da configuração (q0, bab) nenhuma configuração é atingível; para abab temos:

$(q0, abab) \vdash (q1, bab) \vdash (q1, ab) \vdash (q1, b) \vdash (q1, \varepsilon)$

Adicionalmente, temos um outro caminho

$(q0, abab) \vdash (q1, bab) \vdash (q1, ab) \vdash (q2, b)$

que também não atinge nenhuma configuração final. Assim, as cadeias bab e abab não são aceitas e não fazem parte de L(M).

ð

**Exemplo 3:** Considere o afnd M da Fig. 4.5. M aceita cadeias da forma c y c, onde c pode ser a ou b e y pode ser qualquer cadeia de a's e b's.

A cadeia ababa = c·y·c = a·bab·a é aceita por M, através da sequência de configurações abaixo, em que a primeira e a última transições são realizadas através de transições-$\varepsilon$.

(A, ababa)  M lê ε e adivinha que c=a
|— (B, ababa)  M lê a e confere que c=a
|— (C, baba)  M lê b
|— (C, aba)  M lê a e adivinha que este a faz parte de y
|— (C, ba)  M lê b
|— (C, a)  M lê a e  adivinha que este a é o último c
|— (D, ε)  M lê ε e adivinha que a cadeia acabou
|— (I, ε)  M aceita

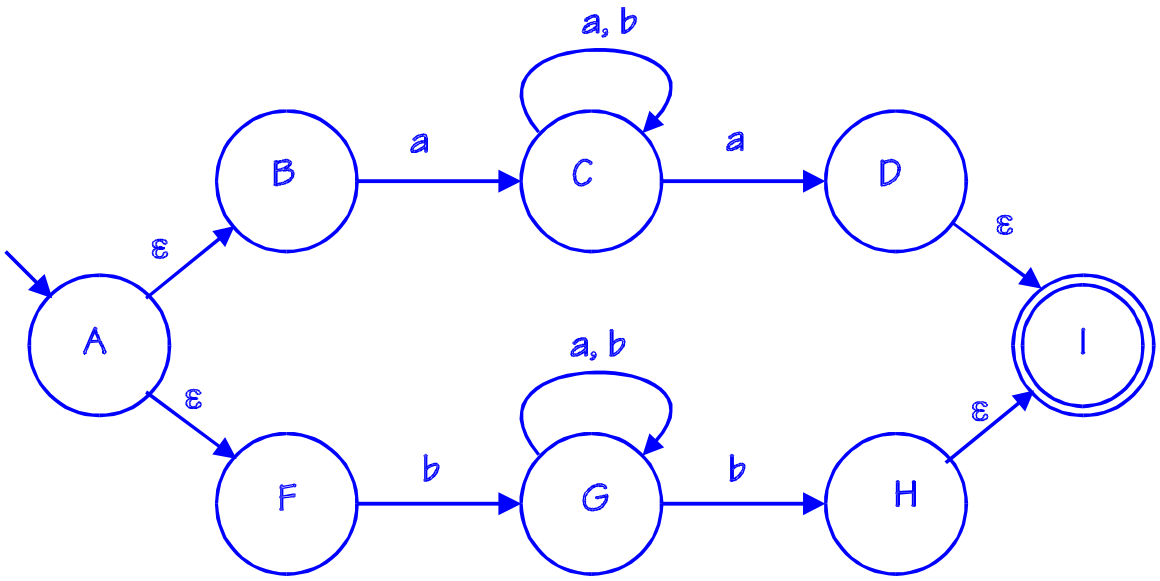



**Fig. 4.5 - afnd para o Exemplo 3**

Todas as configurações atingíveis (caminhos certos e errados) estão indicadas abaixo:

(A, ababa)
|— (B, ababa)
. |— (C, baba)
. |— (C, aba)
. |— (C, ba)
. . |— (C, a)
. . |— (C, ε)  -- não aceita
. . |— (D, ε)
. . |— (I, ε)  -- ok! aceita!
. |— (D, ba)
. |— (I, ba)  -- bloqueado
|— (F, ababa)  -- bloqueado

ð

**Exercício 6:** Considere a linguagem composta pelas cadeias no alfabeto {a, b} que contém a cadeia aaa ou a cadeia bb. Ou seja, a linguagem

L = { x y z | x, z ∈ {a, b}*  e  ( y=aaa  ou  y=bb ) }

Construa um afnd M que aceite L.

Sugestão: M adivinha se a cadeia de entrada contém aaa ou bb, e apenas verifica esse fato.

ð

## 4.4 - Equivalência dos afd's e dos afnd's

Mostraremos nesta seção que uma linguagem é aceita por um af determinístico se e somente se ela é aceita por um af não determinístico. A classe de linguagens reconhecidas por afd's e afnd's é a classe das linguagens regulares (ver seção 4.6).

**Teorema:** Toda linguagem reconhecida por um afd é reconhecida por um afnd.

Demonstração: Exercício.
Sugestão: Basta definir um afnd em que a única transição possível em cada caso é aquela especificada no afd.

ð

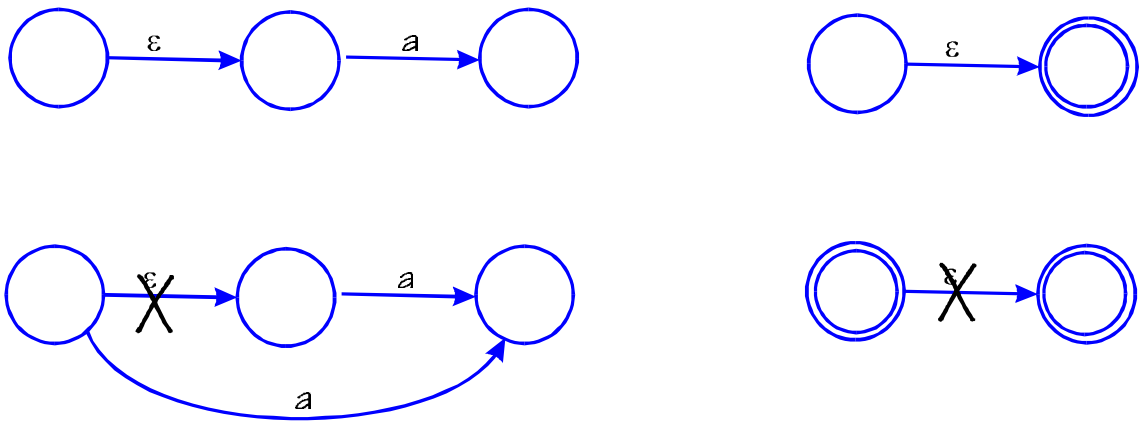
**Teorema:** Toda linguagem reconhecida por um afnd é reconhecida por um afd.
Demonstração: ver Lemas 1 e 2 abaixo.

ð

**Lema 1:** Toda linguagem reconhecida por um afnd é reconhecida por um afnd que não tem transições com ε.
Demonstração: Seja M = (K, Σ, δ, i, F) um afnd qualquer. Vamos construir um afnd M' = (K, Σ, δ', i, F') equivalente a M, isto é L(M') = L(M). Para isso vamos "simular" o efeito das transições com ε de duas maneiras:

- se tivermos a ∈ Σ, $\delta(p_1, \varepsilon) = p_2$, e $\delta(p_2, a) = q$, acrescentaremos a δ' uma transição de $p_1$ para q com a, ou seja, acrescentaremos q ao conjunto $\delta'(p_1, a)$;
- se tivermos $\delta(p_1, \varepsilon) = p_2$, e $p_2 \in F$, acrescentaremos $p_1$ a F.

(ver figura abaixo)




Isso deve ser feito repetidamente enquanto novas transições forem acrescentadas a δ', e enquanto novos estados forem acrescentados a F. Após isso, retiramos de δ as transições com ε, e chamamos os resultados de δ' e F'.

ð

**Exemplo 4:** Considere o afnd M do Exemplo 3 (Fig. 4.5). A construção descrita na prova do Lema 1 permite construir o afnd equivalente M' (Fig. 4.6), que não tem

transições com ε. Note que M' tem estados inúteis: B, F e I passaram a ser inacessíveis a partir do estado inicial.

Para aceitar a cadeia ababa, as configurações de M estão na tabela a seguir:

| Configurações de M | Configurações de M' |
|---|---|
| (A, ababa) | (A, ababa) |
| (B, ababa) | --- |
| (C, baba) | (C, baba) |
| (C, aba) | (C, aba) |
| (C, ba) | (C, ba) |
| (C, a) | (C, a) |
| (D, ε) | (D, ε) --- final |
| (I, ε) --- final | --- |

ð

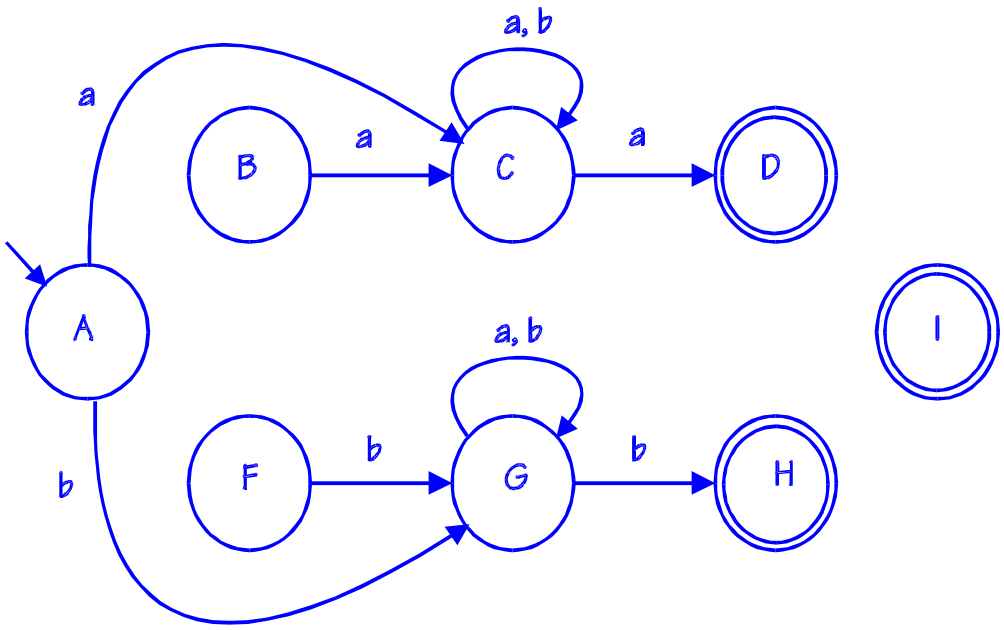



Fig. 4.6 - afnd sem transições-ε

**Lema 2:** Toda linguagem aceita por um afnd sem transições com ε é aceita por um afd.

Demonstração. Seja M = (K, Σ, δ, i, F) um afnd sem transições com ε. Vamos construir um afd M' = (K', Σ, δ', i', F'), equivalente a M, isto é, M' também aceita L. A idéia da demonstração é que os estados de M' são conjuntos de estados de M: a cada momento o estado de M' contém todos os estados em que M poderia se encontrar. Desta maneira, M' pode seguir ao mesmo tempo todos os caminhos percorridos por M. Temos:

K'= P(K); estados de M' são conjuntos de estados de M
i' = { i } o estado inicial de M' contém apenas o estado inicial de M
F' = { $Q \subseteq K \mid Q \cap F \neq \varnothing$ }
os estados finais de M' são os conjuntos de estados de M que contém pelo menos um estado final de M
$\delta': K' \times \Sigma \rightarrow K'$

A função δ' deve cobrir todas as possibilidades: δ'(Q, a) deve incluir todos os estados em todos os conjuntos δ(q, a), para cada q em Q, ou seja,

$$\delta'(Q,a) = \bigcup_{q \in Q} \delta(q,a)$$

para cada a ∈ Σ. A aceitação nas duas máquinas se dá de forma paralela:

M aceita x, e temos em M (i, x) |—* (f, ε), com f ∈ F
M' aceita x, e temos em M' (i', x) |—* (Q, ε), com $Q \cap F \neq \varnothing$

A ligação entre as duas sequências de configurações é feita pelo estado f ∈ Q.

O restante da demonstração consiste na prova de que dada uma das sequências de configurações, é possível construir a outra, e vice-versa.

Observamos que em geral não é necessário levar em consideração todos os estados de M', bastando apenas considerar aqueles que são acessíveis a partir de i'. Se M tem n estados, M' tem um máximo teórico de $2^n$ estados, mas em geral apenas uma fração desses estados é acessível a partir do estado inicial i'.

ð

**Exemplo 4** (continuação): Podemos construir um afd M" a partir de M', como descrito na demonstração do Lema 2. M" será equivalente a M (Exemplo 3) e a M'.

Temos: i" = { A }. A tabela abaixo mostra a função δ". Note que os 251 estados não acessíveis a partir de { A } foram ignorados. O afd pode ser visto também na Fig. 4.7.

| δ" | a | b |
|---|---|---|
| { A } | { C } | { G } |
| { C } | { C, D } | { C } |
| { G } | { G } | { G, H } |
| { C, D } | { C, D } | { C } |
| { G, H } | { G } | { G, H } |

Os estados finais de M" que precisam ser considerados são, portanto, {C, D} e {G, H}, que contém os estados finais D e H de M'. Para comparação, a tabela abaixo apresenta as configurações assumidas por M, M' e M" na aceitação de ababa.

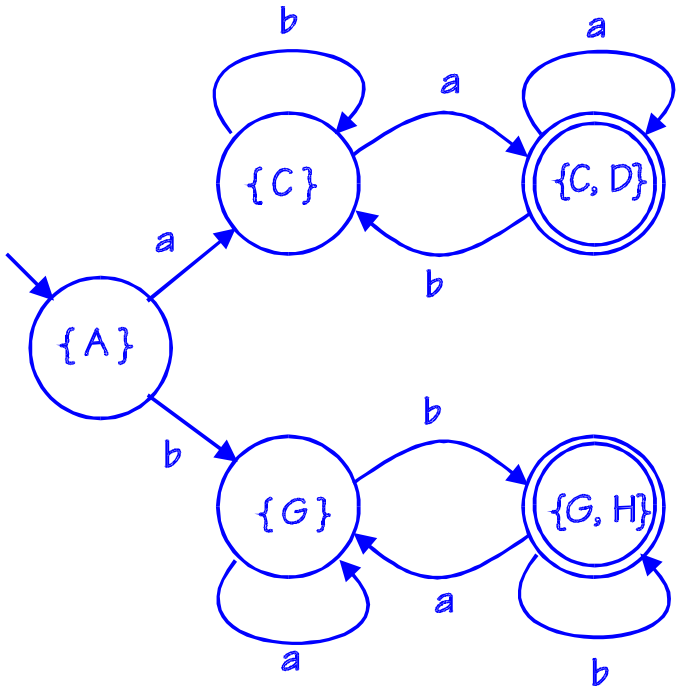



**Fig. 4.7 - afd para o Exemplo 4**

| M | M' | M'' |
|---|---|---|
| (A, ababa) | (A, ababa) | ({A}, ababa) |
| (B, ababa) | --- | --- |
| (C, baba) | (C, baba) | ({C}, baba) |
| (C, aba) | (C, aba) | ({C}, aba) |
| (C, ba) | (C, ba) | ({C, D}, ba) |
| (C, a) | (C, a) | ({C}, a) |
| (D, ε) | (D, ε) | ({C, D}, ε) |
| (I, ε) | --- | --- |

ð

**Exercício 7:** Construa um afd equivalente ao afnd do Exercício 6.

ð

## 4.5 - Minimização de autômatos finitos

Para af determinísticos, é possível fazer uma minimização: dado um afd M, é possível construir um afd M', que é equivalente a M, e que tem o menor número de estados possível para todos os afd's que aceitam L(M). (Esta construção *não se aplica* a af não determinísticos.)

Uma propriedade adicional, que não demonstraremos aqui, é que o afd mínimo é único, exceto pelos nomes dos estados, que podem ser escolhidos arbitrariamente. Ou seja, o mesmo afd mínimo é obtido, a partir de qualquer afd que aceite a linguagem considerada.

A maneira de construir o afd mínimo é baseada na idéia de estados *equivalentes*, que podem ser *reunidos* em um só. Dois estados p e q são *equivalentes* quando as mesmas cadeias levam dos dois estados até a aceitação (até um estado final). Temos:

p ≡ q se e somente se
para toda cadeia $x \in \Sigma^*$,
$\hat{\delta}(p, x) \in F$ se e somente se $\hat{\delta}(q, x) \in F$

A última linha pode ser substituída por

ou $\hat{\delta}(p, x)$ e $\hat{\delta}(q, x)$ são ambos finais, ou são ambos não finais.

A relação ≡ é uma relação de equivalência. Portanto, trivialmente, para estados p, q e r quaisquer, temos

- p ≡ p (reflexividade)
- se p ≡ q, então q ≡ p (simetria)
- se p ≡ q e q ≡ r, então p ≡ r (transitividade)

(Demonstração: exercício.)

O algoritmo que vamos descrever aqui se baseia no fato de que é mais fácil provar que dois estados p e q não são equivalentes do que provar que são. Para mostrar que p e q não são equivalentes, basta achar *uma* cadeia x tal que $\hat{\delta}(p, x)$ é final e $\hat{\delta}(q, x)$ não é final, ou vice-versa. Dizemos que essa cadeia x *distingue* o estado p do estado q, e que p e q são distinguíveis.

As propriedades que vamos usar no algoritmo são:

**Propriedade 1.** *(Equivalência se propaga para a frente.)* Se p ≡ q, então para qualquer $a \in \Sigma$, os estados p' = δ(p, a) e q' = δ(q, a) são equivalentes.
Dem. Seja $x \in \Sigma^*$ uma cadeia qualquer. Devemos mostrar que p" = $\hat{\delta}(p', x)$ e q" = $\hat{\delta}(q', x)$ são ambos finais ou ambos não finais.

Seja y = ax. Temos

$$p'' = \hat{\delta}(p', x) = \hat{\delta}(\delta(p, a), x) = \hat{\delta}(p, ax) = \hat{\delta}(p, y)$$

e

$$q'' = \hat{\delta}(q', x) = \hat{\delta}(\delta(q, a), x) = \hat{\delta}(q, ax) = \hat{\delta}(q, y)$$

Como p ≡ q, p" e q" são ambos finais ou ambos não finais.

**Propriedade 2.** *(Distinguibilidade se propaga para trás.)* Para qualquer $a \in \Sigma$, se p' = δ(p, a) e q' = δ(q, a) não são equivalentes, então p e q também não são equivalentes.

Dem. Se p'e q' não são equivalentes, existe uma cadeia x que distingue p' e q'. Ou seja, chamando p" = $\hat{\delta}(p', x)$ e q" = $\hat{\delta}(q', x)$, temos que um deles é final e o outro não. Fazendo y = ax, temos

$$p'' = \hat{\delta}(p', x) = \hat{\delta}(\delta(p, a), x) = \hat{\delta}(p, ax) = \hat{\delta}(p, y)$$

e

$$q'' = \hat{\delta}(q', x) = \hat{\delta}(\delta(q, a), x) = \hat{\delta}(q, ax) = \hat{\delta}(q, y)$$

e vemos que y = ax distingue p e q.

ð

**Propriedade 3.** *(Iniciação)* Um estado final e um estado não final não podem ser equivalentes.

Demonstração: Sejam $p \in F$, e $q \notin F$. A cadeia vazia ε distingue p de q: $p = \hat{d}(p, \varepsilon) \in F$, e $q = \hat{d}(q, \varepsilon) \notin F$.

ð

Para minimizar um afd M, começamos por determinar quais são os pares de estados de M que são equivalentes, isto é, que podem ser reunidos em um único estado. Como é mais fácil descobrir quais são os pares de estados não equivalentes, consideramos que estados p e q são equivalentes se não conseguirmos mostrar que são distinguíveis (não equivalentes). As estruturas de dados usadas pelo algoritmo são:

para cada par (p, q) de estados distintos,

- um valor booleano marca(p, q), inicialmente com o valor false.
  Se marca(p,q) = true, p e q são distinguíveis.
- uma lista de pares de estados, inicialmente vazia.
  Se (r, s) está na lista de (p, q), isto significa que r e s serão distinguíveis, se p e q forem distinguíveis.

Se marca(p, q) = true, dizemos que o par (p, q) está marcado. Note que o par identificado como (p, q) é o mesmo par identificado como (q, p), e, portanto, tanto faz marcar (p, q), como marcar (q, p).

Note que o algoritmo que determina os pares de estados equivalentes é baseado nas propriedades vistas acima. As listas são usadas para evitar a necessidade de passar mais de uma vez por cada par de estados. Ao final da execução do algoritmo, os pares de estados equivalentes são os que *não* estão marcados. A prova de correção do algoritmo, pode ser encontrada, por exemplo, em [HopUll79][1].

**Algoritmo.** Determinação dos estados equivalentes em um afd M.

```
procedimento mark(p, q);
        se p ≠ q então
                marca(p, q) := true;
                para cada par (r,s) na lista de (p, q)
                        execute mark( r, s);
```

[1] **John E. Hopcroft, Jeffrey D. Ullman**, ***Introduction to Automata Theory, Languages and Computation***, Addison-Wesley, 1979 - **Sec. 3.4, p.68**

```
{parte principal do algoritmo}
        para cada p final,
                para cada q não final,
                        marca(p, q):= true;
        para cada par (p, q) não marcado,
                se existe um símbolo a tal que o par (δ(p,a), δ(q,a)) está marcado,
                        execute mark(p, q)
                senão, para cada símbolo a faça
                        p' := δ(p, a);
                        q' := δ(q, a);
                        se p' ≠ q' e (p, q) ≠ (p', q'),
                                acrescente (p, q) à lista de (p', q').
```

ð

O teste da penúltima linha não é realmente necessário, e pode ser considerado como uma otimização.

Dado M = (K, Σ, δ, i, F), usamos como estados as classes de equivalência de ≡, obtidas para M para a construção do afd mínimo M' = (K', Σ, δ', i', F'). Temos:

$K' = K/\equiv = \{ [q] \mid q \in K \}$
$\delta'$: $K' \times \Sigma \rightarrow K'$, dada por $\delta'([q], a) = [\delta(q, a)]$
$i' = [ i ]$
$F' = \{ [ f ] \mid f \in F \}$

Deixamos como exercício demonstrar a consistência da definição de δ', isto é, a demonstração de que o resultado da aplicação de δ' independe da escolha do representante q da classe de equivalência [q].

**Exemplo 5:** Seja M um afd com estados A, B, C, D, E e F, sendo A o estado inicial; C e F são os estados finais. Os símbolos de entrada são a e b, e δ como na tabela abaixo. M aceita as cadeias que tem um número de a's da forma 6n+2 ou 6n+5. Na realidade, bastaria exigir que o número de a's fosse da forma 3n+2, o que corresponde a um afd com apenas 3 estados, e, por essa razão, M não é mínimo, e deve ter estados equivalentes.

A tabela de transição de M é

| δ | a | b |
|---|---|---|
| A | B | A |
| B | C | B |
| C | D | C |
| D | E | D |
| E | F | E |
| F | A | F |

Os pares de estados (representados em ordem alfabética sem os parenteses) a serem considerados são AB, AC, AD, AE, AF, BC, BD, BE, BF, CD, CE, CF, DE, DF,

e EF. Não há necessidade de incluir pares como AA por causa da reflexividade, nem pares como BA por causa da simetria: basta incluir AB.

Vamos aplicar o algoritmo acima para determinar os pares de estados equivalentes.

(marcação dos pares final / não final)

marcamos AC, AF, BC, BF, CD, CE, DF e EF.

(exame de cada par não marcado)

AB: Temos $\delta(A, a)=B$, $\delta(B, a)=C$, e BC está marcado. Logo, marcamos AB.
AD: Temos $\delta(A, a)=B$, $\delta(D, a)=E$, e $\delta(A, b)=A$, $\delta(D, b)=D$. Como BE não está marcado, incluímos AD na lista de BE. (Note que não há necessidade de incluir AD na lista de AD.)
AE: Temos $\delta(A, a)=B$, $\delta(E, a)=F$, e BF está marcado. Logo, marcamos AE.
BD: Temos $\delta(B, a)=C$, $\delta(D, a)=E$ e CE está marcado. Logo, marcamos BD.
BE: Temos $\delta(B, a)=C$, $\delta(E, a)=F$, e $\delta(B, b)=B$, $\delta(E, b)=E$. Como CF não está marcado, incluímos BE na lista de CF.
CF: Temos $\delta(C, a)=D$, $\delta(F, a)=A$, e $\delta(C, b)=C$, $\delta(F, b)=F$. Como AD não está marcado, incluímos CF na lista de AD.
DE: Temos $\delta(D, a)=E$, $\delta(E, a)=F$ e EF está marcado. Logo, marcamos DE.

(os pares restantes são equivalentes)

Os pares marcados aparecem na tabela abaixo:

| | | | | | | |
|---|---|---|---|---|---|---|
| A | | | | | | |
| B | X | | | | | |
| C | X | X | | | | |
| D | | X | X | | | |
| E | X | | X | X | | |
| F | X | X | | X | X | |
| | A | B | C | D | E | F |

Os pares restantes (não marcados) são AD, BE, CF. Logo, $A \equiv D$, $B \equiv E$ e $C \equiv F$. Naturalmente, além disso, $A \equiv A$, $D \equiv A$, etc. Note que as cadeias que distinguem os pares de estados não equivalentes podem ser deduzidas da ordem de marcação: para os pares final/não final, a cadeia é $\varepsilon$. Para os demais pares, neste exemplo, a cadeia é a. Por exemplo, marcamos AB porque BC estava marcado, e porque de A e B passamos com o símbolo a para B e C. A cadeia correspondente a AB é portanto $a \cdot \varepsilon = a$.

Podemos agora construir o afd mínimo: o conjunto de estados é o das classes de equivalência. Como previsto, tem apenas 3 estados. Temos:

K' = { [A], [B], [C], [D], [E], [F] } = { {A, D}, {B, E}, {C, F} }
i' = [A] = {A, D}
F' = { [C], [F] }= {C, F}

Para calcular as transições, escolhemos representantes das classes. Por exemplo, como [A] = [D] = {A, D}, $\delta'(\{A, D\}, a)$ pode ser calculada como $\delta'([A], a) = [\delta(A, a)] = [B] = \{B, E\}$ ou como $\delta'([D], a) = [\delta(D, a)] = [E] = \{B, E\}$. Qualquer que seja o

representante escolhido, o resultado será o mesmo, porque, como vimos na propriedade 1, "a equivalência se propaga para a frente".

A função de transição pode ser vista na tabela abaixo:

| δ' | a | b |
|---|---|---|
| {A, D} | {B, E} | {A, D} |
| {B, E} | {C, F} | {B, E} |
| {C, F} | {A, D} | {C, F} |

ð

Um resultado interessante, cuja demonstração pode ser encontrada na referência citada, é o de que o afd mínimo que aceita uma dada linguagem é único, exceto por isomorfismo. Neste contexto, isomorfismo quer dizer simplesmente re-nomeação de estados: dados dois afd's $M_1$ e $M_2$ que aceitam a mesma linguagem L, se construirmos os afd's mínimos associados ao dois afd's, encontraremos dois afd's $M_1$' e $M_2$' que são, na prática, idênticos: $M_1$' e $M_2$' só se distinguem pelos nomes de seus estados. Ou seja, é a linguagem L que define o afd mínimo que a aceita, e o resultado é sempre o mesmo, independente do afd aceitador de L do qual partimos.

Isso pode ser usado para resolver dois problemas interessantes. Primeiro, se quisermos determinar se dois afd's $M_1$ e $M_2$ são equivalentes, basta construir os afd's $M_1$' e $M_2$' mínimos correspondentes. Se $M_1$' e $M_2$' forem isomorfos, $M_1$ e $M_2$ são equivalentes. Segundo, se quisermos mostrar que um afd M dado é mínimo, basta aplicar a M o processo de minimização, e verificar que o resultado M' é isomorfo de M. Isto é feito no Exemplo 6, onde, adicionalmente, para cada par de estados (p, q) distintos de M, deduzimos exemplos de cadeias que os distinguem.

**Exemplo 6:** Vamos verificar que um afd é mínimo, aplicando a ele o processo de minimização, e mostrando que o resultado final é isomorfo do afd inicial. Seja M o afd com estados A, B, C, D, E e F, sendo A o estado inicial; e F o único estado final. Os símbolos de entrada são a e b. A tabela de transição de M é

| δ | a | b |
|---|---|---|
| A | B | A |
| B | C | B |
| C | D | C |
| D | E | D |
| E | F | E |
| F | A | F |

Aplicando o processo de minimização, temos:

(marcação dos pares final / não final)

marcamos AF, BF, CF, DF, EF.

(exame de cada par não marcado)

AB: incluímos AB na lista de BC;
AC: incluímos AC na lista de BD;

AD: incluímos AD na lista de BE;
AE: como BF está marcado, marcamos AE;
BC: incluímos BC na lista de CD;
BD: incluímos BD na lista de CE
BE: como CF está marcado, marcamos BE; portanto, marcamos AD (da lista de BE);
CD: incluímos CD na lista de DE;
CE: como DF está marcado, marcamos CE; portanto, marcamos BD (da lista de CE) e AC (da lista de BD);
DE: como EF está marcado, marcamos DE; portanto, marcamos CD (da lista de DE), BC (da lista de CD) e AB (da lista de BC).

Os pares marcados aparecem na tabela abaixo:

| | | | | | | |
|---|---|---|---|---|---|---|
| A | | | | | | |
| B | X | | | | | |
| C | X | X | | | | |
| D | X | X | X | | | |
| E | X | X | X | X | | |
| F | X | X | X | X | X | |
| | A | B | C | D | E | F |

(os pares restantes são equivalentes)

Não há pares de estados distintos restantes. Ou seja, cada estado é equivalente apenas a ele mesmo. O afd mínimo é idêntico a M, apenas tem estados {A}, {B}, {C}, {D}, {E}, {F}. As cadeias d(XY) que distinguem os pares de estados XY são:

d(AF) = d(BF) = d(CF) = d(DF) = d(EF) = ε.
d(AE) = a · d(BF) = a · ε = a
d(BE) = a · d(CF) = a · ε = a
d(AD) = a · d(BE) = a · a = aa
d(CE) = a · d(DF) = a · ε = a
d(BD) = a · d(CEF) = a · a = aa
d(AC) = a · d(BD) = a · aa = aaa
d(DE) = a · d(EF) = a · ε = a
d(CD) = a · d(DE) = a · a = aa
d(BC) = a · d(CD) = a · aa = aaa
d(AB) = a · d(BC) = a · aaa = aaaa

Naturalmente, as cadeias d(XY) podem também ser obtidas por inspeção, sem executar o algoritmo.

ð

**Exercício 8:** Construa um afd mínimo que aceite a linguagem L no alfabeto Σ = {a, b}, com L ={ cdxcd | c, d ∈ Σ, x ∈ Σ* }

ð

**Exercício 9:** Construa um afd mínimo que aceite o complemento da linguagem L do Exercício 8.

ð

## 4.6 - Equivalência entre autômatos finitos e gramáticas regulares

Um dos resultados que devemos estabelecer neste capítulo é que a classe de linguagens reconhecidas por automatos finitos é a classe das linguagens regulares. Já sabemos, da seção anterior, que a classe das linguagens aceitas por af determinísticos é exatamente a mesma classe das linguagens aceitas por af não determinísticos. Trata-se, portanto de estabelecer dois resultados simples, expressos através dos Teoremas 4.6 e 4.7, a seguir.

**Teorema 4.6:** Toda linguagem regular é aceita por um afnd.

Demonstração: Seja L uma linguagem regular. Portanto, L = L(G), para alguma gramática regular G = ( N, Σ, P, S ). Vamos construir um afnd M = ( K, Σ, δ, i, F ) que aceita a linguagem L(G). Temos: K = N U { f }, i = S, F = { f }. Ou seja, os estados de M serão os não terminais de M, mais um estado f criado para ser o único estado final. O estado inicial é o símbolo inicial de G. (Note que f deve ser um símbolo novo, para não causar confusão.)

As transições de M são dadas pelas regras de G:

Inicialmente, faça δ(A, a)= ∅, para todos os nãoterminais A e para todos os símbolos a, e para a = ε. A seguir, para todas as regras de G,

- se G tem uma regra A → a B, acrescente uma transição de A para B com o símbolo a, ou seja, acrescente B a δ(A, a).
- se G tem uma regra A → a, acrescente uma transição de A para f com o símbolo a, ou seja, acrescente f a δ(A, a).
- se G tem uma regra A → ε, acrescente uma transição de A para f com ε, ou seja, acrescente f a δ(A, ε).

Devemos mostrar que, para qualquer x ∈ Σ*, M aceita x sse x ∈ L(G). A demonstração se completa pela verificação de que a sequência de configurações (S, x) |—* (f, ε) em M corresponde exatamente à sequência de passos da derivação S ⇒* x em G.

ð

**Exemplo 7:** Seja a gramática regular G, dada por suas regras:

S → a A | b B
A → a A | b A | a
B → a B | b B | b

que gera a linguagem { c x c | c ∈ {a, b} e x ∈ {a, b}*}. O afnd descrito na prova do teorema anterior é M = ({ S, A, B, f }, { a, b }, δ, S, { f }), com δ dada pela tabela abaixo.

| δ | ε | a | b |
|---|---|---|---|
| S | ∅ | { A } | { B } |
| A | ∅ | { A, f } | { A } |
| B | ∅ | { B } | { B, f } |
| f | ∅ | ∅ | ∅ |

Seja a cadeia x=ababa. A cadeia x pertence à linguagem, como se pode ver pela derivação

$S \Rightarrow aA \Rightarrow abA \Rightarrow abaA \Rightarrow ababA \Rightarrow ababa.$

A cadeia x também é aceita por M, como se pode ver pela sequência de configurações

$(S, ababa) \vdash (A, baba) \vdash (A, aba) \vdash (A, ba) \vdash (A, a) \vdash (f, \varepsilon)$

Note que os estados e os símbolos não terminais aparecem na mesma ordem, exceto por f, que não aparece na derivação. Os símbolos terminais, entretanto, tem tratamento diverso: são gerados na derivação, e aparecem desde sua introdução até a cadeia final, e são consumidos nas transições do afnd, aparecendo desde a configuração inicial até o momento de sua leitura.

ð

**Exemplo 8:** Seja a gramática regular G, dada por suas regras:

$S \rightarrow a\ A \mid b\ A \mid \varepsilon$
$A \rightarrow a\ S \mid b\ S$

Temos $L(G) = \{ x \in \{a, b\}^* \mid |x| \text{ é par } \}$. O afnd descrito na prova do teorema anterior é $M = (\{ S, A, f \}, \{ a, b \}, \delta, S, \{ f \})$, com $\delta$ dada pela tabela abaixo.

| $\delta$ | $\varepsilon$ | a | b |
|---|---|---|---|
| S | { f } | { A } | { A } |
| A | $\varnothing$ | { S } | { S } |
| f | $\varnothing$ | $\varnothing$ | $\varnothing$ |

Seja a cadeia x = abba, de comprimento par. Temos:

$S \Rightarrow aA \Rightarrow abS \Rightarrow abbA \Rightarrow abbaS \Rightarrow abba.$

em G, e

$(S, abba) \vdash (A, bba) \vdash (S, ba) \vdash (A, a) \vdash (S, \varepsilon) \vdash (f, \varepsilon)$

em M.

ð

**Teorema 4.7:** Se L é aceita por um automato finito, então L é regular.

demonstração: Podemos supor que L é aceita por um afd $M = (K, \Sigma, \delta, i, F)$. Vamos construir uma gramática regular G para L. A gramática $G = (K, \Sigma, P, i)$ tem como símbolos não terminais os estados de M, e como símbolo inicial o estado inicial i de M. As regras de G são dadas pelas transições e pelos estados finais de M:

se $p = \delta(q, a)$ em M, G tem uma regra $q \rightarrow ap$ em P.
se q é final ($q \in F$), G tem uma regra $q \rightarrow \varepsilon$ em P

A demonstração é semelhante a anterior: devemos mostrar que, para qualquer $x \in \Sigma^*$, M aceita x sse $x \in L(G)$.

ð

**Exemplo 9:** Seja o afd $M = (\{q_0, q_1\}, \{a, b\}, \delta, q_0, \{q_1\})$, com $\delta$ dada pela tabela abaixo.

| δ | a | b |
|---|---|---|
| $q_0$ | $q_1$ | $q_1$ |
| $q_1$ | $q_0$ | $q_0$ |

M aceita as cadeias de {a, b}* que tem comprimento ímpar.

A gramática G correspondente, de acordo com o teroema acima, é

$q_0 \rightarrow a\ q_1 \mid b\ q_1$
$q_1 \rightarrow a\ q_0 \mid b\ q_0 \mid \varepsilon$

Para a cadeia x= ababa, temos

$(q_0, ababa) \vdash (q_1, baba) \vdash (q_0, aba) \vdash (q_1, ba) \vdash (q_0, a) \vdash (q_1, \varepsilon)$

e

$q_0 \Rightarrow aq_1 \Rightarrow abq_0 \Rightarrow abaq_1 \Rightarrow ababq_0 \Rightarrow ababaq_1 \Rightarrow ababa$

ð

## 4.7 - Expressões regulares

Vamos agora definir expressão regular. A expressão regular é a maneira mais compacta e mais simples de descrever conjuntos regulares, e é usada com essa finalidade em construção de compiladores, editores, sistemas operacionais, protocolos, etc. A definição abaixo é uma definição recursiva, e será usada como base para outras definições, e para as demonstrações.

*Definição.* Definimos uma *expressão regular* (*er*) em um alfabeto Σ através de *ER1 ¼ ER6* abaixo:

*ER1*. ∅ é uma er.
*ER2*. ε é uma er.
*ER3*. para cada a ∈ Σ, a é uma er.
*ER4*. Se α e β são er's, então (α ∨ β) é uma er.
*ER5*. Se α e β são er's, então (α ∘ β) é uma er.
*ER6*. Se α é uma er, então (α*) é uma er.

Naturalmente, α é uma er se e somente se isso pode ser provado a partir de *ER1 ¼ ER6*.

Usualmente, são omitidos os parenteses de er's, de acordo com a ordem de precedência

* → ∨ → ∘

e considerando os operadores como associativos à esquerda. Além disso, o símbolo ∘ é frequentemente omitido.

**Exemplo 10:** Seja Σ = {a, b} e seja a expressão regular α = (a∨b)* a b b, ou seja, com todos os parênteses, α = (((((a∨b)*)∘a)∘b)∘b). Mostramos que α é uma er, mostrando sucessivamente que são er's as expressões a seguir:

| | | |
|---|---|---|
| 1. | a | de *ER3* |
| 2. | b | de *ER3* |
| 3. | (a∨b) | de 1, 2 e *ER4* |
| 4. | (a∨b)* | de 3 e *ER6* |

5. $(a \vee b)^* \circ a$ de 4, 1 e *ER5*
6. $(a \vee b)^* \circ a \circ b$ de 5, 2 e *ER5*
7. $(a \vee b)^* \circ a \circ b \circ b$ de 6, 2 e *ER5*.

ð

*Definição.* A linguagem $L[\alpha]$ associada a uma er (ou denotada pela er) é definida de forma recursiva, seguindo a definição de er:

*ER1.* $L[\varnothing] = \varnothing$;
*ER2.* $L[\varepsilon] = \{\varepsilon\}$;
*ER3.* para cada $a \in \Sigma$, $L[a] = \{a\}$;
*ER4.* $L[(\alpha \vee \beta)] = L[\alpha] \cup L[\beta]$;
*ER5.* $L[(\alpha \circ \beta)] = L[\alpha] \circ L[\beta]$;
*ER6.* $L[(\alpha^*)] = (L[\alpha])^*$.

**Exemplo 11:** Seja $\alpha = (a \vee b)^* \circ a \circ b \circ b$, como acima. Podemos determinar a linguagem $L[\alpha]$ seguindo o mesmo caminho usado para provar que $\alpha$ é uma er.

1. $L[a] = \{a\}$ de *ER3*
2. $L[b] = \{b\}$ de *ER3*
3. $L[a \vee b] = L[a] \cup L[b] = \{a\} \cup \{b\} = \{a, b\}$ de 1, 2 e *ER4*
4. $L[(a \vee b)^*] = (L[a \vee b])^* = \{a, b\}^*$ de 3 e *ER6*
5. $L[(a \vee b)^* \circ a] = L[(a \vee b)^*] \circ L[a] = \{a, b\}^* \circ \{a\}$ de 4, 1 e *ER5*
6. $L[(a \vee b)^* \circ a \circ b] = L[(a \vee b)^* \circ a] \circ L[b] =$
   $= \{a, b\}^* \circ \{a\} \circ \{b\} = \{a, b\}^* \circ \{ab\}$ de 5, 2 e *ER5*
7. $L[(a \vee b)^* \circ a \circ b \circ b] = L[(a \vee b)^* \circ a \circ b] \circ L[b] =$
   $= \{a, b\}^* \circ \{ab\} \circ \{b\} = \{a, b\}^* \circ \{abb\}$ de 6, 2 e *ER5*

Assim, $L[\alpha]$ é a linguagem das cadeias que terminam em abb.

ð

Uma outra forma de indicar as mesmas propriedades de pertinência vistas acima, mais adequada para provar a pertinência em casos isolados é:

*ER1.* Não existe x tal que $x \in L[\varnothing]$.
*ER2.* Se $x \in L[\varepsilon]$, então $x = \varepsilon$
*ER3.* Se $x \in L[a]$ (para $a \in \Sigma$), então $x = a$.
*ER4.* Se $x \in L[\alpha \vee \beta]$, então ou $x \in L[\alpha]$, ou $x \in L[\beta]$.
*ER5.* Se $x \in L[\alpha \circ \beta]$, então $x = yz$, com $y \in L[\alpha]$ e $z \in L[\beta]$.
*ER6.* Se $x \in L[\alpha^*]$, então ou $x = \varepsilon$, ou $x = yz$, com $y \in L[\alpha]$ e $z \in L[\alpha^*]$.

Os casos 1..5 são autoexplicativos; para o caso 6, basta observar a propriedade apresentada no Exercício 10.

**Exercício 10:** Mostrar que, para qualquer linguagem L , $L^* = \{\varepsilon\} \cup (L \circ L^*)$.

**Exemplo 12:** Suponhamos que desejamos provar que $x = abaabb \in L[\alpha]$, para a er $\alpha = (a \vee b)^* \circ a \circ b \circ b$, usando as propriedades acima. Os passos intermediários da prova estão indicados abaixo:

1. $a \in L[a]$
2. $a \in L[a \vee b]$ de 1
3. $b \in L[b]$
4. $b \in L[a \vee b]$ de 3

| | | |
|---|---|---|
| 5. | $\varepsilon \in L[(a \vee b)^*]$ | |
| 6. | $a \in L[(a \vee b)^*]$ | de 2 e 5 |
| 7. | $ba \in L[(a \vee b)^*]$ | de 4 e 6 |
| 8. | $aba \in L[(a \vee b)^*]$ | de 2 e 7 |
| 9. | $abaa \in L[(a \vee b)^* \circ a]$ | de 8 e 1 |
| 10. | $abaab \in L[(a \vee b)^* \circ a \circ b]$ | de 9 e 3 |
| 11. | $abaabb \in L[(a \vee b)^* \circ a \circ b \circ b]$ | de 10 e 3 |

Note que cada ocorrência de um símbolo (a ou b) em x fica associada a uma ocorrência do mesmo símbolo em $\alpha$. No fundo a construção da prova de que $x \in L[\alpha]$ consiste exatamente na descoberta de uma associação adequada. No exemplo acima, a única associação possível está indicada abaixo, pela numeração das ocorrências de símbolos, na er e na cadeia considerada:

$$\alpha = (a_1 \vee b_2)^* \, a_3 \circ b_4 \circ b_5 \, , \; x = a_1 \, b_2 \, a_1 \, a_3 \, b_4 \, b_5$$

Em outros casos, podem ser possíveis várias associações. Por exemplo, considere o alfabeto $\Sigma = \{a, b, c\}$, a er $\beta = (a \vee b)^* \circ (bvc)^*$ e a cadeia y = bb. Neste caso, temos

$$\beta = (a_1 \vee b_2)^* \; \circ \; (b_3 \vee c_4)^*$$

e podemos ter $y = b_2 \, b_2$, ou $y = b_2 \, b_3$, ou ainda, $y = b_3 \, b_3$ .

ð

*Definição.* Dizemos que duas er's $\alpha$ e $\beta$ são *equivalentes* se as linguagens a elas associadas são iguais: $L[\alpha] = L[\beta]$. A equivalência é indicada por $\alpha \equiv \beta$.

**Exercício 11:** Mostre que, dada uma er $\alpha$, é sempre possível construir uma er $\beta$ equivalente a $\alpha$, de forma que ou $\beta = \varnothing$, ou $\beta$ não contém o símbolo $\varnothing$.

Sugestão: considere as equivalências

$\varnothing \vee \alpha \equiv \alpha \vee \varnothing \equiv \alpha$
$\varnothing \circ \alpha \equiv \alpha \circ \varnothing \equiv \varnothing$
$\varnothing^* \equiv \varepsilon$

ð

**Exercício 12:** Mostre que, dada uma er $\alpha$, é sempre possível construir uma er $\beta$ equivalente a $\alpha$, de forma que ou $\beta = \varepsilon \vee \gamma$, ou $\beta = \gamma$, e $\gamma$ não contém o símbolo $\varepsilon$.

Sugestão: considere as equivalências

$\varepsilon \circ \alpha \equiv \alpha$
$(\varepsilon \vee \alpha)^* \equiv \alpha^*$
$(\varepsilon \vee \alpha) \circ \beta \equiv \beta \vee (\alpha \circ \beta)$
$(\varepsilon \vee \alpha) \vee \beta \equiv \varepsilon \vee (\alpha \vee \beta)$

ð

**Exercício 13:** Suponha a seguinte definição: uma er $\alpha$ é *ambígua* se para algum $x \in L[\alpha]$, existe mais de uma associação possível entre as ocorrências de símbolos em x e em $\alpha$. Sejam as expressões

$\beta = (a \vee b)^* \circ (b \vee c)^*$
$\gamma = (a \vee b)^* \circ (a \circ a \vee b \circ b) \circ (a \vee b)^*$

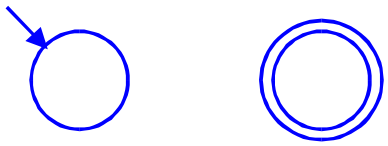
- Mostre que $\beta$ e $\gamma$ são ambíguas, de acordo com a definição dada.
- Construa er's equivalentes a $\beta$ e $\gamma$ que não sejam ambíguas.

**Teorema 4.8:** Toda linguagem denotada por uma expressão regular é regular.

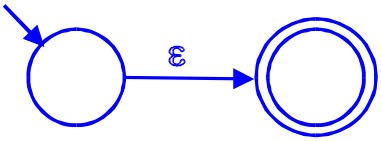
Demonstração. Seja $\alpha$ uma er qualquer. Vamos mostrar que $L(\alpha)$ é regular construindo um afnd $M(\alpha)$ que aceita $L(\alpha)$, preparando para uma demonstração por indução na estrutura de $\alpha$.

Por simplicidade, vamos construir todos os afnd $M(\alpha)$ considerados nesta demonstração com exatamente um estado final, distinto do estado inicial. Para uma er $\alpha$ não elementar, o afnd $M(\alpha)$ será construído a partir dos afnd's das er's componentes. Para evitar a necessidade de nomear cada estado de cada uma dessas máquinas, vamos indicar a forma de composição graficamente. Por convenção, sempre representaremos o estado inicial à esquerda, e o estado final à direita.

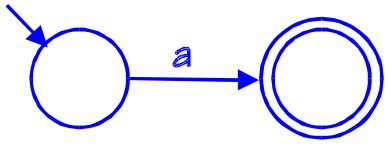
*ER1*. O afnd $M(\varnothing)$ que aceita $\varnothing$ é



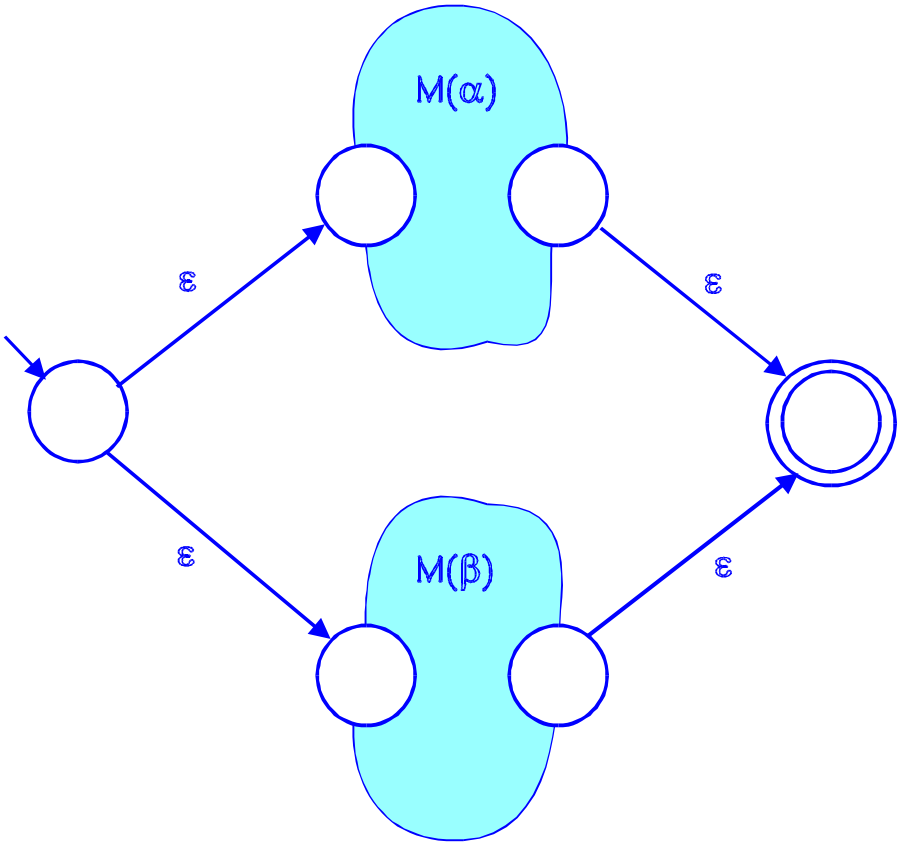
*ER2*. O afnd $M(\varepsilon)$ que aceita $L[\varepsilon]$ é



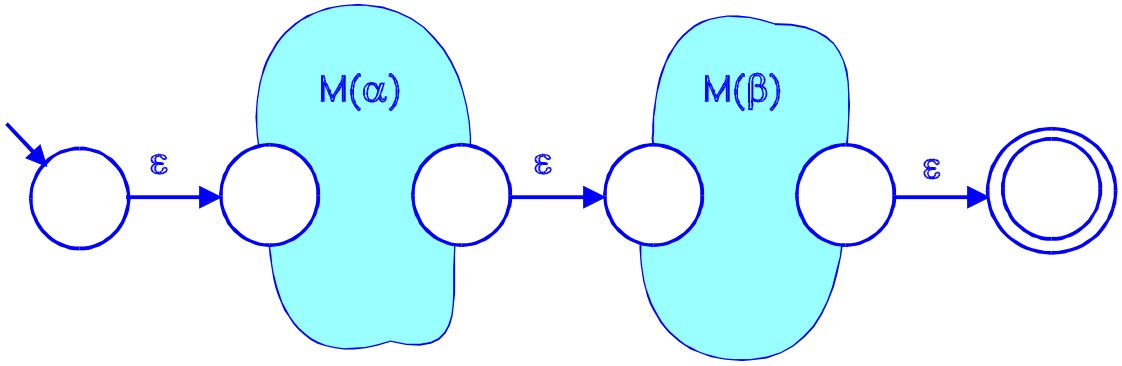

*ER3*. O afnd M(a) que aceita L[a], $a \in \Sigma$ é



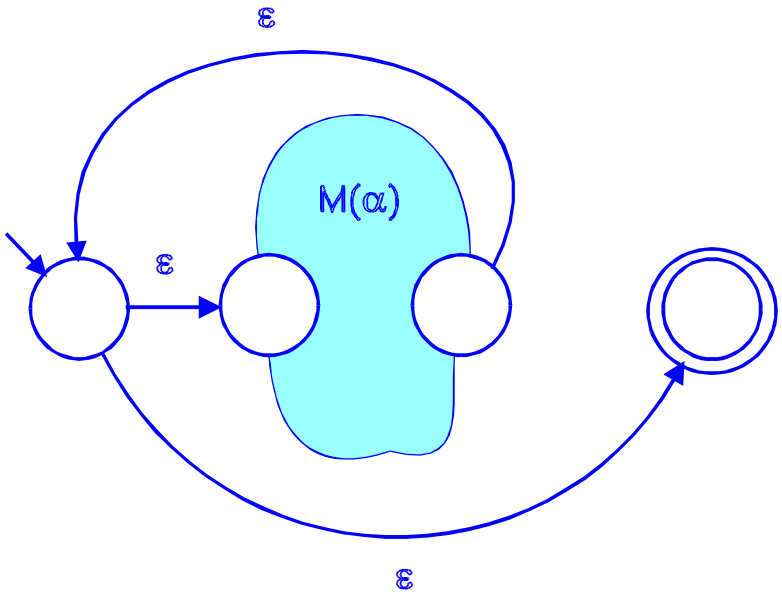

*ER4*. Se $\alpha$ e $\beta$ são er's, podemos supor (pela Hipótese de Indução) que já estão construídos $M(\alpha)$ e $M(\beta)$. O afnd $M(\alpha \vee \beta)$ que aceita $L[\alpha \vee \beta]$ é obtido acrescentando um estado inicial e um final novos a $M(\alpha)$ e $M(\beta)$. As novas transições são transições com entrada $\varepsilon$ do estado inicial novo para os antigos estados iniciais de $M(\alpha)$ e de $M(\beta)$, e dos antigos estados finais de $M(\alpha)$ e de $M(\beta)$ para o novo estado final. O resultado é




*ER5*. Se α e β são er's, podemos supor (pela Hipótese de Indução) que já estão construídos M(α) e M(β). O afnd M(α ∘ β) que aceita L[α ∘ β] é obtido acrescentando um estado inicial e um final novos a M(α) e M(β). As novas transições são transições com entrada ε do estado inicial novo para o antigo estado inicial de M(α), do antigo estado final de M(α) para o antigo estado inicial de M(β), e do antigo estado final de M(β) para o novo estado final. O resultado é




*ER6*. Se α é uma er, podemos supor (pela Hipótese de Indução) que já está construído M(α). O afnd M(α*) que aceita α* é obtido acrescentando um estado inicial e um final novos a M(α). As novas transições são transições com entrada ε do estado inicial novo para o antigo estado inicial de M(α) e para o novo estado final e do antigo estado final de M(α) para o novo estado inicial.




O restante da demonstração é deixado como exercício.

ð

**Exemplo 13:** Seja a er $\alpha = (a \vee b)^* a$. Vamos construir um afnd que aceita $L(\alpha)$, seguindo a construção indicada na demonstração acima. Os passos intermediários e os resultados estão indicados nas tabelas a seguir

| M(a) | | ε | a | b |
|---|---|---|---|---|
| inicial: | A | ∅ | {B} | ∅ |
| final: | B | ∅ | ∅ | ∅ |

| M(b) | | ε | a | b |
|---|---|---|---|---|
| inicial: | C | ∅ | ∅ | {D} |
| final: | D | ∅ | ∅ | ∅ |

| M(a ∨ b) | | ε | a | b |
|---|---|---|---|---|
| inicial: | E | {A, C} | ∅ | ∅ |
| | A | ∅ | {B} | ∅ |
| | B | {F} | ∅ | ∅ |
| | C | ∅ | ∅ | {D} |
| | D | {F} | ∅ | ∅ |
| final: | F | ∅ | ∅ | ∅ |

M((a ∨ b)*)

| | | ε | a | b |
|---|---|---|---|---|
| inicial: | G | {E, H} | ∅ | ∅ |
| | E | {A, C} | ∅ | ∅ |
| | A | ∅ | {B} | ∅ |
| | B | {F} | ∅ | ∅ |
| | C | ∅ | ∅ | {D} |
| | D | {F} | ∅ | ∅ |
| | F | {G} | ∅ | ∅ |
| final: | H | ∅ | ∅ | ∅ |

M((a ∨ b)*∘a)

| | | ε | a | b |
|---|---|---|---|---|
| inicial: | I | {G} | ∅ | ∅ |
| | G | {E, H} | ∅ | ∅ |
| | E | {A, C} | ∅ | ∅ |
| | A | ∅ | {B} | ∅ |
| | B | {F} | ∅ | ∅ |
| | C | ∅ | ∅ | {D} |
| | D | {F} | ∅ | ∅ |
| | F | {G} | ∅ | ∅ |
| | H | {A'} | ∅ | ∅ |
| | A' | ∅ | {B'} | ∅ |
| | B' | {J} | ∅ | ∅ |
| final: | J | ∅ | ∅ | ∅ |

Note que a sub-expressão a ocorre duas vezes em M(α), e porisso foi necessário incluir duas vezes M(a); para a segunda vez renomeamos os estados, que passaram a ser A' e B'.

ð

**Exercício 14:** Construa um afd mínimo para a linguagem denotada pela er α do Exemplo 13, a partir do afnd M(α) construído no exemplo.

ð

**Exercício 15:** Construa afd's mínimos que aceitem as linguagens denotadas pelas expressões regulares do Exercício 13,

β = (a ∨ b)* ∘ (b ∨ c)*

γ = (a ∨ b)* ∘ (a∘a ∨ b∘b) ∘ (a ∨ b)*

ð

**Teorema 4.9:** Toda linguagem regular é denotada por uma expressão regular.
Demonstração: ver referência citada.

ð

A demonstração do Teorema 4.9 constrói a expressão regular que representa a linguagem aceita por um afd examinando os caminhos entre o estado inicial e os estados finais do afd. O operador ∨ é usado para tratar caminhos alternativos, o operador ∘ para tratar caminhos de comprimento maior que 1, e o operador * para tratar laços. Embora a construção seja interessante, na prática o uso normalmente feito de er's é para

especificação de linguagens regulares, e é muito mais frequente a construção de afd's a partir de er's, do que a construção inversa.

## 4.8 - O Lema do Bombeamento para linguagens regulares

Como mencionado anteriormente, precisamos de um resultado que nos permita demonstrar que algumas linguagens não são regulares. Este resultado é o "Lema do Bombeamento", ou "Pumping Lemma", que nos diz que qualquer cadeia suficientemente longa z de uma linguagem regular pode ser decomposta em três partes: z = uvw, de maneira que podemos construir outras cadeias da linguagem pela repetição da parte central v: todas as cadeias da forma $u\ v^i\ w$ são também da linguagem. Ou seja, podemos acionar a *bomba* quantas vezes quisermos, para criar quantas sentenças novas da linguagem desejarmos: uw, uvvw, uvvvw, ... .

Para mostrar que uma linguagem não é regular, mostramos que não há como decompor uma cadeia (qualquer, arbitrariamente longa) da linguagem de forma que seja possível *bombear* e continuar na linguagem.

**Teorema 4.10:** (Lema do Bombeamento) Seja L uma linguagem regular. Então, existe um natural n tal que qualquer cadeia z de L com comprimento maior ou igual a n pode ser decomposta em três cadeias u, v e w (z = uvw) de forma que

$|uv| \leq n$
$v \neq \varepsilon$
para qualquer $i \geq 0$, $u\ v^i\ w \in L$

Demonstração: A demonstração se baseia no fato de que para as cadeias *longas* z é necessário usar pelo menos um *loop* de estados num afd que aceite a linguagem. Assim, os símbolos de u são usados para chegarmos a um estado q do loop; os símbolos de v são usados para dar a volta no loop, de volta ao estado q; os símbolos de w são usados para ir de q até um estado final. Portanto, podemos dar quantas voltas no loop quisermos, e repetir v um número qualquer i de vezes: $u\ v^i\ w$.

As cadeias *curtas* (comprimento < n) devem ser excluídas porque podem ser aceitas sem passar por nenhum loop.

A demonstração completa pode ser encontrada em [HopUll79].

ð

**Exemplo 14:** Seja a linguagem regular $L = L[\alpha] = L[\beta]$, com $\alpha = 1(01)^*$ e $\beta = (10)^*1$. Considere a cadeia z=10101, pertencente a L. Podemos decompor a cadeia, da forma descrita no teorema acima, de diversas formas. Por exemplo:

| | | |
|---|---|---|
| u = 1 | v = 01 | w = 01 |
| u = 10 | v = 10 | w = 1 |
| u = ε | v = 1010 | w = 1 |

Note que todas as cadeias das formas $1(01)^i01$, $10(10)^i1$, $(1010)^i1$ pertencem a L.

ð

**Exercício 16:** (baseado no Exemplo 14)

- Mostre que $\alpha$ e $\beta$ são equivalentes.
- Estime o valor de n (Teorema 4.10) para L.
- Mostre todas as formas de decomposição de z que satisfazem o Teorema.

ð

**Exemplo 15:** A linguagem $L = \{ a^i b^i \mid i \geq 0\}$ não é regular. (Já vimos que L é uma llc.) Vamos mostrar aqui que L não é regular. A demonstração é por contradição. Suponha que L é regular. Se L é regular, o Teorema acima se aplica, e existe n tal que a decomposição descrita pode ser realizada para qualquer cadeia de comprimento igual ou maior que n.

Seja k=n+1. Considere a cadeia $z=a^k b^k$. Qualquer decomposição z=uvw deve ter em v o mesmo número de a's e de b's, para que a propriedade de que o número de a's é igual ao de b's se mantenha nas cadeias $u v^i w$. Se isso não acontecer, quando acrescentarmos mais um v (aumentando i de 1), obteremos uma cadeia fora da linguagem. Portanto, v deve ser da forma $a^j b^j$, com j>0, já que v não pode ser vazia. Mas nesse caso, $u v^2 w$ conterá a cadeia $a^j b^j a^j b^j$, com pelo menos um a depois de um b, o que não pode acontecer na linguagem.

Ou seja, nenhuma decomposição é possível, contrariando o Teorema, e podemos concluir que L não é regular.

ð

**Exercício 17:** Mostre que a linguagem $L = \{ x x^R \mid x \in \{0, 1\}^* \}$ não é regular.

ð

## 4.9 - Propriedades de fechamento das linguagens regulares

Vamos ver agora algumas propriedades de fechamento da classe das linguagens regulares. A maioria das provas pode ser feita usando mais de um dos formalismos usados para definir linguagens regulares: gramáticas regulares, afd's, afnd's e expressões regulares — veja o Exercício 18.

**Teorema 4.11:** A classe das linguagens regulares no alfabeto $\Sigma$ é fechada para as operações de *(a)* união, *(b)* interseção, *(c)* complemento em relação a $\Sigma^*$, *(d)* concatenação e *(e)* fechamento transitivo (estrela).

Demonstração:
*(a)* Vamos mostrar que se $L_1$ e $L_2$ são linguagens regulares, então $L_1 \cup L_2$ é uma linguagem regular. Sejam $\alpha$ e $\beta$ er's tais que $L(\alpha)=L_1$ e $L(\beta) = L_2$.
Portanto, $L_1 \cup L_2 = L(\alpha \vee \beta)$ também é regular.

*(b)* Vamos mostrar que se $L_1$ e $L_2$ são linguagens regulares, então $L_1 \cap L_2$ é uma linguagem regular. Sejam $M_1 = (K_1, \Sigma, \delta_1, i_1, F_1)$ e $M_2 = (K_2, \Sigma, \delta_2, i_2, F_2)$, afd's tais que $L(M_1)=L_1$ e $L(M_2) = L_2$.
Seja $M = (K_1 \times K_2, \Sigma, \delta, (i_1, i_2), F_1 \times F_2)$, com $\delta$ definida por

$$\delta( (q_1, q_2), a) = ( \delta_1(q_1, a), \delta_2(q_2, a) )$$

M aceita $L_1 \cap L_2$ , porque

$( (i_1, i_2), x) \vdash\!\!\!-^* ( (f_1, f_2), \varepsilon)$ em M sse
$( i_1, x) \vdash\!\!\!-^* ( f_1, \varepsilon)$ em $M_1$
e $( i_2, x) \vdash\!\!\!-^* (f_2, \varepsilon)$ em $M_2$,

e

$(f_1, f_2)$ é final em M sse
$f_1$ é final em $M_1$
e $f_2$ é final em $M_2$.

L é aceita por M, e portanto, é regular.

*(c)* Vamos mostrar que se L é regular, o complemento $\overline{L} = \Sigma^* - L$ também é regular. Seja $M = (K, \Sigma, \delta, i, F)$ um afd que aceita L. Então $M' = (K, \Sigma, \delta, i, K-F)$ aceita $\overline{L}$. Temos:

M' aceita $x \Leftrightarrow \hat{\delta}(i, x) \in K - F \Leftrightarrow \hat{\delta}(i, x) \notin F \Leftrightarrow$ M não aceita x
$\Leftrightarrow x \notin L \Leftrightarrow x \in \overline{L}$.

*(d)* Vamos mostrar que, se $L_1$ e $L_2$ são regulares, $L_1 \circ L_2$ é regular. Sejam $\alpha$ e $\beta$ er's tais que $L(\alpha)=L_1$ e $L(\beta) = L_2$. Portanto, $L_1 \circ L_2 = L(\alpha \circ \beta)$ também é regular.

*(e)* Vamos mostrar que se L é regular, o fechamento L* também é regular. Seja a uma er tal que $L(\alpha) = L$. Então $L(\alpha^*)^* = (L(\alpha))^* = L^*$ também é regular.

ð

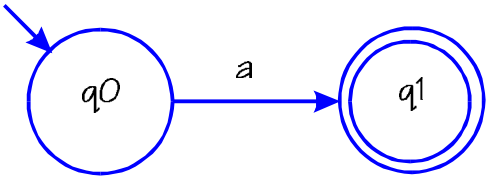
**Exercício 18**: Considere o Teorema 4.11. Construa demonstrações alternativas para os casos indicados:

*(a)* construindo uma gramática regular para a união de $L_1$ e $L_2$, a partir de gramáticas regulares de $L_1$ e $L_2$.
*(b)* usando (a), (c), e a propriedade de conjuntos (de Morgan) que diz que $X \cap Y = \overline{\overline{X} \cup \overline{Y}}$
*(d)* construindo uma gramática regular para a concatenação de $L_1$ e $L_2$, a partir de gramáticas regulares de $L_1$ e $L_2$.
*(e)* construindo uma gramática regular para o fechamento de L, a partir de uma gramática regular de L.

ð

**Exercício 19**: Mostre que, na construção usada na demonstração do Teor. 4.11 parte (c), não pode ser usado um afnd.

Sugestão: Considere o afnd




*(revisão de 27fev97)*

# Linguagens Formais
# Capítulo 5: Linguagens e gramáticas livres de contexto

José Lucas Rangel, maio 1999

## *5.1 - Introdução*

Vimos no capítulo 3 a definição de gramática livre de contexto (glc) e de linguagem livre de contexto (llc). As regras de uma glc são da forma $A \rightarrow \alpha$, onde $\alpha$ é uma cadeia qualquer de terminais e nãoterminais, possivelmente vazia. Como vimos, o que caracteriza a gramática livre de contexto é a propriedade de que o símbolo não terminal A pode ser substituído pela cadeia $\alpha$ do lado direito da regra, onde quer que A ocorra, independentemente do contexto, isto é, do resto da cadeia que está sendo derivada. Por essa razão, é possível representar derivações em glc's através de árvores de derivação: para usar a regra $A \rightarrow \alpha$, acrescentamos à árvore, como filhos de A, nós correspondentes aos símbolos de $\alpha$.

## *5.2 - Árvores de derivação*

Uma árvore de derivação é uma árvore composta da seguinte maneira:

- a raiz tem como rótulo o símbolo inicial S da gramática.
- a cada nó rotulado por um nãoterminal A corresponde uma regra de A. Se a regra for $A \rightarrow X_1X_2 \ldots X_m$, os filhos do nó são rotulados, da esquerda para a direita, por $X_1$, $X_2$, ..., $X_m$. (cada um dos $X_i$ pode ser um terminal ou um nãoterminal.)
- um nó rotulado por um terminal é sempre uma folha da árvore, e não tem filhos.

Os nós interiores da árvore são sempre (rotulados por) nãoterminais. Se a um nó rotulado pelo nãoterminal A for aplicada a regra $A \rightarrow \varepsilon$, considera-se o nó como interior, embora ele tenha zero filhos. Na representação gráfica da árvore, é costume indicar, neste caso, os zero filhos através de um nó $\varepsilon$, que não contribui para o resultado da árvore.

Uma sub-árvore de uma árvore de derivação é um nó da árvore com todos seus descendentes, as arestas que os conectam e seus rótulos. Uma sub-árvore sempre corresponde a uma derivação parcial, a partir do símbolo que rotula a raiz da sub-árvore. O resultado de uma árvore de derivação é a cadeia formada pelos terminais (que aparecem como folhas da árvore), lidos da esquerda para a direita. Note que a ordenação dos filhos de cada nó é fundamental para que se possa definir a ordenação das folhas da árvore.

Exemplo 5.1: Seja a gramática G, dada por suas regras:

$S \rightarrow AB$
$A \rightarrow aaA \mid \varepsilon$
$B \rightarrow Bbb \mid \varepsilon$

A árvore de derivação para a sequência `aaaabb` está representada na Fig. 1. O resultado da árvore é `aaaabb`, sendo as regras escolhidas de acordo com a derivação

$S \Rightarrow B \Rightarrow aaAB \Rightarrow aaaaAB \Rightarrow aaaaB \Rightarrow aaaaBbb \Rightarrow aaaabb.$

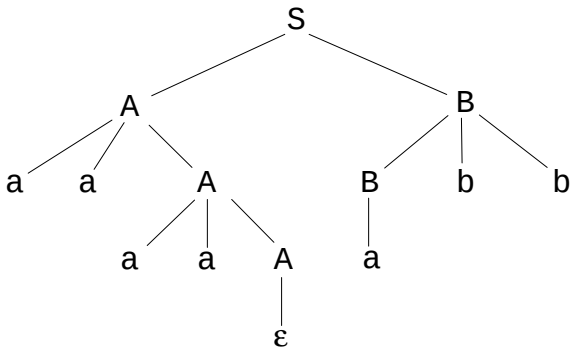



Fig. 1 - Árvore de derivação de aaaabb

Teorema 5.1: Seja G = (N, Σ, P, S) uma glc. Então, para qualquer cadeia x ∈ Σ*,

x ∈ L(G) se e somente se existe uma árvore de derivação A na gramática G, cujo resultado é x.

Demonstração: Basta observar a correspondência entre a substituição de nãoterminais pelos lados direitos de suas regras na derivação e a criação dos filhos correspondentes na árvore. Usando essa correspondência é possível construir uma árvore de derivação cujo resultado é x a partir de uma derivação S ⇒* x; é possível também construir uma derivação S ⇒* x a partir de uma árvore de derivação cujo resultado é x. Os detalhes da demonstração são deixados como exercício.

Observação. A partir de uma derivação. só é possível construir uma árvore; entretanto, na direção oposta, é possível construir várias derivações, dependendo da ordem em que os nós são considerados. O Exemplo 5.2 mostra algumas das várias derivações que correspondem à mesma árvore.

## Derivações esquerdas e direitas.

Diz-se que uma derivação é uma derivação esquerda (*leftmost derivation*) se a cada passo da derivação o nãoterminal A escolhido para aplicação de uma regra A → α for sempre aquele que fica mais à esquerda. Simetricamente, fala-se em derivação direita (*rightmost derivation*) se o nãoterminal escolhido é sempre o que estiver mais à direita.

Exemplo 5.2: Seja a gramática $G_0$ abaixo, dada por suas regras. (Esta gramática será usada em vários exemplos, no que se segue.

```
E → E + T | T
T → T * F | F
F → ( E ) | a
```

Considere a cadeia x = a*(a+a)+a. Temos abaixo três derivações de:

```
E ⇒ E+T ⇒ T+T ⇒ T*F+T ⇒ F*F+T ⇒ a*F+T ⇒ a*(E)+T
      ⇒ a*(E+T)+T ⇒ a*(T+T)+T ⇒ a*(F+T)+T ⇒ a*(a+T)+T
      ⇒ a*(a+F)+T ⇒ a*(a+a)+T ⇒ a*(a+a)+F ⇒ a*(a+a)+a
E ⇒ E+T ⇒ E+F ⇒ E+a ⇒ T+a ⇒ T*F+a ⇒ T*(E)+a
      ⇒ T*(E+T)+a ⇒ T*(E+F)+a ⇒ T*(E+a)+a ⇒ T*(T+a)+a
      ⇒ T*(F+a)+a ⇒ T*(a+a)+a ⇒ F*(a+a)+a ⇒ a*(a+a)+a
E ⇒ E+T ⇒ T+T ⇒ T+F ⇒ T*F+F ⇒ T*F+a ⇒ F*F+a
      ⇒ F*(E)+a ⇒ a*(E)+a ⇒ a*(E+T)+a ⇒ a*(T+T)+a
      ⇒ a*(T+F)+a ⇒ a*(F+F)+a ⇒ a*(a+F)+a ⇒ a*(a+a)+a
```

Note que a primeira derivação é uma derivação esquerda, e a segunda é uma derivação direita. Todas as três derivações correspondem à mesma árvore de derivação, apresentada na Figura 2. Como se pode observar, em todas elas aparecem as mesmas regras, aplicadas nos mesmos lugares, variando apenas a ordem em que as regras são aplicadas.

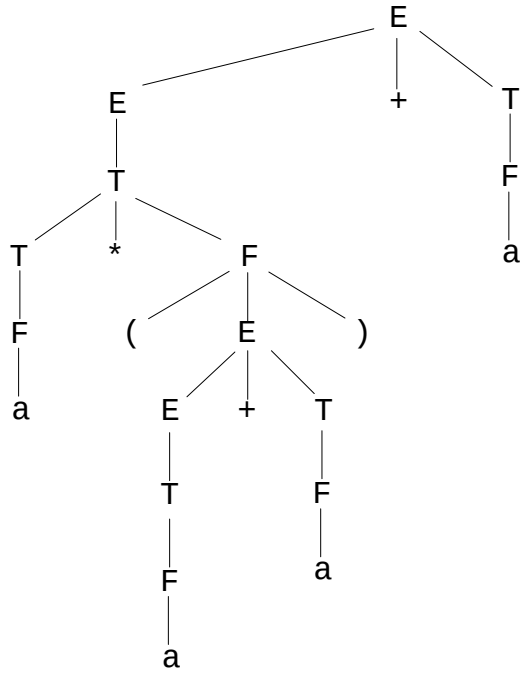



Fig. 2- Árvore de derivação de a*(a+a)+a

Como todas as derivações correspondentes à mesma árvore de derivação descrevem a mesma forma de construção da cadeia derivada - *as mesmas regras aplicadas nos mesmos lugares* - consideramos que a forma de construção da cadeia pode ser representada pela árvore ou por uma derivação esquerda, ou por uma derivação direita. Entretanto, se existem duas ou mais árvores de derivação (duas ou mais derivações esquerdas, duas ou mais derivações direitas), para a mesma cadeia, consideramos que a gramática não define de forma única a maneira pela qual a cadeia é derivada, e dizemos que a gramática é *ambígua*.

Exemplo 5.3: Seja a gramática $G_1$, dada por suas regras:

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

Pode-se verificar que $G_1$ é equivalente a $G_0$, vista no exemplo 5.2 acima. Entretanto, diferentemente de $G_0$, $G_1$ é uma gramática ambígua. Considere, por exemplo a cadeia a+a*a. As duas derivações (esquerdas) abaixo correspondem a duas árvores de derivação distintas.

$E \Rightarrow E+E \Rightarrow a+E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$

$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow a+E*E \Rightarrow a+a*E \Rightarrow a+a*a$

A construção das duas árvores fica como exercício.

Uma linguagem livre de contexto cujas gramáticas são todas ambíguas é chamada uma gramática inerentemente ambígua.

Exemplo 5.4: Sejam $L_1 = \{a^i b^j c^k \mid i=j\}$ e $L_2 = \{a^i b^j c^k \mid j=k\}$. A linguagem L definida por $L = L_1 \cup L_2$ é inerentemente ambígua. As linguagens $L_1$ e $L_2$ não são inerentemente ambíguas, como se pode ver pelas suas respectivas gramáticas $G_1$ e $G_2$.

$G_1$: $S_1 \rightarrow T\ C$ $\quad$ $G_2$: $S_2 \rightarrow A\ V$

$T \rightarrow a\ T\ b \mid \varepsilon$ $\quad$ $A \rightarrow a\ A \mid \varepsilon$

$C \rightarrow c\ C \mid \varepsilon$ $\quad$ $V \rightarrow b\ V\ c \mid \varepsilon$

É fácil construir um exemplo G de gramática ambígua para L, a partir de $G_1$ e $G_2$:

G: $S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow T\ C$ $\quad$ $S_2 \rightarrow A\ V$

$T \rightarrow a\ T\ b \mid \varepsilon$ $\quad$ $A \rightarrow a\ A \mid \varepsilon$

$C \rightarrow c\ C \mid \varepsilon$ $\quad$ $V \rightarrow b\ V\ c \mid \varepsilon$

Para verificar que G é ambígua, basta observar que todas as cadeias pertencentes à interseção $M = L_1 \cap L_2 = \{a^i\ b^j\ c^k \mid i=j=k\ \}$ podem ser derivadas de duas formas distintas, dependendo da regra inicial escolhida para a derivação. Por exemplo, aabbcc pode ser obtida por uma das duas derivações esquerdas abaixo:

$S \Rightarrow S_1 \Rightarrow TC \Rightarrow aTbC \Rightarrow aaTbbC \Rightarrow aabbC \Rightarrow aabbcC$
$\Rightarrow aabbccC \Rightarrow aabbcc$

$S \Rightarrow S_2 \Rightarrow AV \Rightarrow aAV \Rightarrow aaAV \Rightarrow aaV \Rightarrow aabVc \Rightarrow aabbVcc$
$\Rightarrow aabbcc$

Naturalmente, isto prova apenas que G é ambígua, e não que todas as gramáticas de L são ambíguas. Esta demonstração não será incluída aqui.

Observamos também que M não é uma llc. Isto será visto no Exemplo 5.7.

### *5.3 - Simplificação de gramáticas livres de contexto*

Não existe a possibilidade de simplificação de gramáticas livres de contexto, no mesmo sentido da minimização de automatos finitos vista anteriormente. É, entretanto possível fazer uma simplificação que elimina todos os símbolos e regras inúteis da gramática. Podemos dizer que um símbolo terminal é *inútil* quando não aparece em alguma cadeia da linguagem; podemos dizer que um símbolo não terminal é *inútil* quando não aparece em alguma derivação de alguma cadeia da linguagem. Uma regra é *inútil* contém algum símbolo inútil.

Em alguns casos, a gramática é ambígua, e algumas regras podem ser removidas sem que a linguagem se altere, mas pode não ficar claro qual regra deve ser considerada inútil. Por exemplo, se tivermos

1, 2. $S \rightarrow A\ X \mid Y\ C$
3. $X \rightarrow B\ C$
4. $Y \rightarrow A\ B$

há duas maneiras de gerar ABC a partir de S. Quais as regras que devem ser retiradas? 1 e 3 ou 2 e 4? Tanto faz.

Todas as simplificações que podem ser feitas, entretanto, não alteram a essência de uma gramática: apenas a tornam mais limpa. Algumas transformações de gramáticas visam obter uma gramática equivalente à inicial que tem alguma forma particular, por exemplo, que simplifique a demonstração de algum teorema. Para ver alguns algoritmos para simplificação ou transformação de gramáticas sugerimos a mesma referência citada anteriormente.

O exemplo a seguir procura esclarecer alguns dos conceitos mencionados.

Exemplo 5.5: Uma gramática com regras e símbolos inúteis. Seja a gramática

1, 2, 3: S → A B | A C | B D
4, 5: A → a A | a
6, 7: B → b B | b
8, 9: C → c D | d C
10, 11: Y → y | z Z
12, 13: Z → z | y Y

*Símbolos não terminais acessíveis:*

Em derivações a partir de S podem aparecer S A B C D (regras 1, 2, 3).

*Símbolos não terminais produtivos:*

Derivações que levam a cadeias de terminais: A (regra 5), B (regra 7), Y (regra 10), Z (regra 11), S (regra 1, já que A e B são produtivos.)

Logo, todos os nãoterminais, exceto S A B são inúteis. Retirando todas as regras que fazem referência a nãoterminais inúteis, temos:

1: S → A B
4, 5: A → a A | a
6, 7: B → b B | b

Os símbolos restantes podem ser considerados inúteis: C D Y Z c d y z.

### *5.4 - O lema do bombeamento para linguagens livres de contexto.*

Vamos examinar agora um resultado que nos permitirá provar que algumas linguagens não são livres de contexto. O resultado é conhecido como Lema do Bombeamento, ou *Pumping Lemma*, e é semelhante ao resultado correspondente visto para linguagens regulares.

Teorema 5.2: Lema do Bombeamento. Seja L uma llc. Então, existe um número n, que só depende de L, tal que qualquer cadeia z de L com comprimento maior ou igual a n pode ser decomposta de maneira que z = uvwxy e

$|vx| \geq 1$

$|vwx| \leq n$

para todo $i \geq 0$, $uv^iwx^iy$ pertence a L.

Demonstração (simplificada). Se L é uma llc, existe uma glc G tal que L(G)=L. Se z tem um comprimento suficientemente longo, não será possível gerar z sem que na derivação de z ocorra um não terminal A repetido, de forma que a partir de A é derivada uma cadeia que contém outra ocorrência de A. Para que isto ocorra, as duas ocorrências de A devem estar num mesmo caminho da raiz da árvore de derivação até as folhas. (Cadeias mais curtas podem ser geradas sem que essa repetição aconteça.) Através (possivelmente) de uma rearrumação dos passos da derivação, temos

$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy$

ou seja,

$S \Rightarrow^* uAy$,

$A \Rightarrow^* vAx$,

$A \Rightarrow^* w$

Portanto, podemos derivar

| | | |
|---|---|---|
| i=0: | uwy | $S \Rightarrow^* uAy \Rightarrow^* uwy$ |
| i=1: | uvwxy | $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvAxy \Rightarrow^* uvwxy$ |
| i=2: | uvvwxxy | $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* uvvwxxy$ |
| i=3: | uvvvwxxxy | $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy$ $\Rightarrow^* uvvvAxxxy \Rightarrow^* uvvvwxxxy$ |

…

Uma demonstração completa do Lema do Bombeamento pode ser encontrada na referência citada.

Observação. A recíproca do Lema do Bombeamento não é verdadeira, isto é, existem linguagens que não são livres de contexto, mas que tem a propriedade da decomposição.

Exemplo 5.6: Considere a gramática $G_0$, a cadeia z = `a*(a+a)+a`, e a árvore de derivação correspondente a z (Exemplo 5.2), reproduzida na Figura 3. Podemos ver que existem vários casos de repetição de nãoterminais da forma indicada no teorema acima. Por exemplo, vamos considerar as duas ocorrências de T indicadas pelas setas:

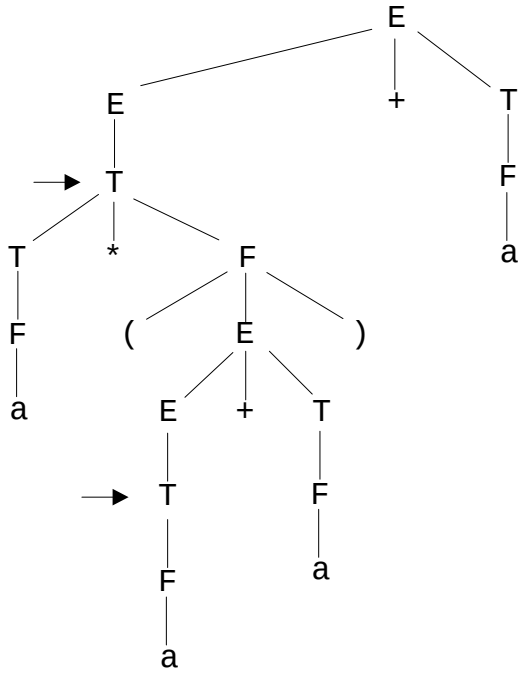



Fig. 3- Árvore de derivação de `a*(a+a)+a`

Temos: `E` $\Rightarrow^*$ `T+a`, `T` $\Rightarrow^*$ `a*(T+a)`, `T` $\Rightarrow^*$ `a`. Ou seja, u = ε, v = `a*(`, w = `a`, x = `+a)`, y=`+a`. Ou seja, as seguintes cadeias devem ser da linguagem:

| | | |
|---|---|---|
| i=0: | uwy | a +a |
| i=1: | $uv^2wx^2y$ | a*( a +a) +a |
| i=2: | $uv^3wx^3y$ | a*( a*( a +a) +a) +a |
| i=3: | | a*( a*( a*( a +a) +a) +a)+a |

…

Exercício 5.2: (Ver Exemplo 5.3)

1. Construa árvores de derivação para as cadeias $uv^iwx^iy$, para i=0, 1, 2, 3, observando que essas árvores são construídas de pedaços da árvore de derivação de x. (Nem todos esses pedaços são sub-árvores.)
2. Verifique todas as combinações de nãoterminais repetidos que satisfazem as condições do teorema, e quais as decomposições possíveis para a cadeia z.
3. Estime o valor de n para a linguagem considerada.

Exercício 5.3: Considere a llc

$$L=\{ x\, x^R \mid x \in \{a, b\}^* \} \cup \{ aaaab \}$$

e a cadeia z = aabaabaa. Estime n para essa linguagem. Determine todas as decomposições possíveis de z, de acordo com o teorema.

Exemplo 5.7: Vamos agora mostrar que $L = \{ a^m b^m c^m \mid n \geq 0 \}$ não é livre de contexto, usando o teorema acima. A demonstração é por contradição: suporemos que L é livre de contexto, e deduziremos um absurdo.

Se L é llc, L satisfaz o teorema acima para algum n. Suponha que a cadeia $z=a^k b^k c^k$ é suficientemente longa: $| z | \geq n$. Então z pode ser decomposta, z = uvwxy, de forma que para qualquer i, $z_i = uv^iwx^iy$ pertence a L. A contradição está em que qualquer decomposição, existem cadeias $z_i$ que não pertencem a L: ou tem o número errado de a's, b's, e c's, ou aparecem símbolos fora da ordem a - b - c: as combinações ba, cb e ca não podem ocorrer em L.

Para eliminar todas as decomposições:

- se v e x não tem o mesmo número de a's, b's e c's, algum $z_i$ terá números diferentes dos três símbolos;
- se v e x tem o mesmo número de a's, b's e c's, devemos ter $v = a^j$ e $x = b^j c^j$, ou $v = a^j b^j$ e $x=c^j$ . No primeiro caso, $z_2$ contém $x^2 = b^j c^j b^j c^j$, que contém a combinação cb, que não ocorre em L; no segundo caso, de forma semelhante, ocorre a combinação ba.

Logo, L não é uma llc.

Exercício 5.4: Mostre que a linguagem $\{ x\, x \mid x \in \{a, b\}^* \}$ não é uma llc.

*Nota: a primeira versão deste capítulo contou com a colaboração de Luiz Carlos Castro Guedes*

*(maio 1999)*

# Capítulo 6:

# Linguagens livres de contexto e autômatos de pilha

José Lucas Rangel, maio 1999

## *6.1 - Introdução.*

Os aceitadores, ou reconhecedores, das linguagens livres de contexto são os chamados *autômatos de pilha* ou *ap*'s. Usaremos aqui o modelo mais geral de ap, o *ap não determinístico*, ou *apnd*, que consiste basicamente de um autômato finito não determinístico, com uma memória adicional, em forma de pilha. Numa pilha, símbolos novos só podem ser acrescentados no topo da pilha; apenas o último símbolo armazenado, o símbolo que se encontra no topo da pilha pode ser consultado; esse símbolo deve ser retirado para que os demais possam ser alcançados.

Neste capítulo vamos provar que a classe de linguagens reconhecidas pelos apnd's é exatamente a classe das llc.

## *6.2 - Autômatos de pilha não determinísticos*

Definição. Definimos um autômato de pilha não determinístico (apnd) como uma construção (tupla) $A = < K, \Sigma, \Gamma, \delta, i, I, F >$, formada pelos seguintes elementos:

K - conjunto (finito) de estados

$\Sigma$ - alfabeto de entrada

$\Gamma$ - alfabeto da pilha

$\delta$ - função de transição

$\delta: K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(K \times \Gamma^*)$

i - estado inicial

$i \in K$

I - símbolo inicial da pilha

$I \in \Gamma$

F - conjunto de estados finais

$F \subseteq K$

O alfabeto $\Gamma$ contém os símbolos que podem ser armazenados na pilha, ou seja, *empilhados*. Para simplificar a especificação da função de transição $\delta$, consideramos que a cada passo, um símbolo é lido (e retirado) do topo da pilha, e, no mesmo passo, uma seqüência de comprimento qualquer pode ser empilhada. Assim, se queremos retirar um símbolo do topo da pilha, basta que a sequência a ser empilhada seja a seqüência vazia $\varepsilon$. Por outro lado, se quisermos manter o símbolo do topo da pilha, ele deve ser re-empilhado.

Escolher de forma não determinística, uma opção $(p, \alpha) \in \delta(q, a, Z)$ quer dizer que a partir do estado q, lendo a da entrada, e lendo Z do topo da pilha, uma transição

possível para A terá como próximo estado p, e a seqüência $\alpha$ será empilhada, depois da remoção de Z da pilha.

Da mesma forma que num autômato finito não determinístico (afnd), temos a possibilidade de executar uma transição sem avançar na leitura dos símbolos da entrada, e essa ação é representada pela leitura da cadeia vazia $\varepsilon$. Por essa razão, o segundo argumento de $\delta$ pode ser $\varepsilon$, além de poder ser um símbolo de $\Sigma$.

Usamos $P(K \times \Gamma^*)$ para representar o conjunto potência de $K \times \Gamma^*$, ou seja o conjunto de todos os subconjuntos de $K \times \Gamma^*$. Como se trata aqui de um conjunto infinito, é necessário especificar que, em qualquer caso, $\delta(q, a, Z)$ será sempre um conjunto finito, de forma a manter finita a descrição do apnd A.

Para descrever os passos de uma computação realizada por um apnd, utilizamos *configurações* $[q, y, \gamma] \in K \times \Sigma^* \times \Gamma^*$. Os três elementos de uma configuração são o estado corrente $q \in K$, a parte da entrada ainda a ser lida, $y \in \Sigma^*$, e o conteúdo $\gamma \in \Gamma^*$ da pilha. Por convenção, *o topo da pilha fica à esquerda*, isto é, o primeiro símbolo de $\gamma$ é considerado como sendo o símbolo do topo da pilha. A configuração inicial correspondente à entrada x é [i, x, I].

Vamos definir a relação *mudança de configuração*, representada por $\vdash$ ou por $\vdash_A$, se quisermos explicitar o apnd A considerado. Para isso utilizamos a função de transição $\delta$. Suponha uma configuração $[q, ax, Z\gamma]$, com $(p, \alpha) \in \delta(q, a, Z)$. Escolhida esta opção, o próximo estado será p, e a cadeia $\alpha$ deve ser empilhada, após a leitura de Z. Assim, podemos passar da configuração $[q, ax, Z\gamma]$para a configuração $[p, x, \alpha\gamma]$, em que o estado passou a ser p, o "símbolo" a da entrada e o símbolo Z da pilha foram lidos, e a seqüência $\alpha$ foi empilhada. Ou seja,

se $(p, \alpha) \in \delta(q, a, Z)$

então $[q, ax, Z\gamma] \vdash [p, x, \alpha\gamma]$

Temos duas situações em que se pode dizer que um apnd aceita sua entrada, que correspondem a duas definições independentes do que se entende por linguagem reconhecida por um apnd. A primeira é semelhante à usada nos afnd: o apnd aceita se, após ler sua entrada, *o estado é um estado final*; a segunda é mais característica dos apnd: o apnd aceita se, após ler sua entrada, *a pilha está vazia*. Assim, podemos definir a linguagem de um apnd A (a linguagem aceita, ou reconhecida por A) de duas maneiras diferentes:

- aceitação por estado final:

  $L_{ef}(A) = \{ x \in \Sigma^* \mid [i, x, I] \vdash^* [f, \varepsilon, \gamma], \text{ com } f \in F \text{ e } \gamma \in \Gamma^* \text{ qualquer } \}$

- aceitação por pilha vazia:

  $L_{pv}(A) = \{ x \in \Sigma^* \mid [i, x, I] \vdash^* [f, \varepsilon, \varepsilon], \text{ com } q \in K \text{ qualquer } \}$

  São, portanto, duas as definições de configuração final:

  $[q, \varepsilon, \gamma]$ é uma configuração final para aceitação por estado final se $q \in F$;

  $[q, \varepsilon, \gamma]$ é uma configuração final para aceitação por pilha vazia, se $\gamma = \varepsilon$.

No primeiro caso, o conteúdo $\gamma$ da pilha é irrelevante; no segundo caso, o estado q é irrelevante. Note-se que o conjunto de estados finais F de um apnd A não é

utilizado na definição da linguagem que A aceita por pilha vazia. Por essa razão costuma-se, neste caso, fazer F = ∅, sem perda de generalidade.

Um ponto importante a observar é que, no caso geral, as linguagens $L_{ef}(A)$ e $L_{pv}(A)$ podem ser distintas.

Exemplo 6.1: Seja o apnd A = < K, Σ, Γ, δ, i, I, F >, com

K = {0, 1, 2}  i = 0

Σ = {a, b}  I = X

Γ = {X, A}  F = {2}

A função $\delta:\{0,1,2\}\times\{a,b,\varepsilon\}\times\{X,A\} \rightarrow P(\{0,1,2\}\times\{X,A\}^*)$ é dada por

$\delta(0, a, X) = \{(0, AX)\}$  $\delta(1, b, A) = \{(1, \varepsilon)\}$

$\delta(0, a, A) = \{(0, AA)\}$  $\delta(1, \varepsilon, X) = \{(2, X)\}$

$\delta(0, b, A) = \{(1, \varepsilon)\}$

Convencionamos, neste exemplo, e nos seguintes, que o valor da função δ, para as combinações de valores não especificadas, é sempre o conjunto vazio ∅. No caso presente, portanto,

$\delta(0, b, X) = \delta(0, \varepsilon, X) = \delta(0, \varepsilon, A) = \delta(1, a, X) = \delta(1, a, A) = \delta(1, b, X)$
$= \delta(1, \varepsilon, A) = \delta(2, a, X) = \delta(2, a, A) = \delta(2, b, X) = \delta(2, b, A)$
$= \delta(2, \varepsilon, X) = \delta(2, \varepsilon, A) = \emptyset.$

Suponhamos agora que a entrada do apnd A é x=aaabbb. A configuração inicial correspondente é [0, aaabbb, X]. A partir dessa configuração, os seguintes passos são possíveis:

$[0, aaabbb, X] \vdash [0, aabbb, AX] \vdash [0, abbb, AAX] \vdash [0, bbb, AAAX]$
$\vdash [1, bb, AAX] \vdash [1, b, AX] \vdash [1, \varepsilon, X] \vdash [2, \varepsilon, X]$

A última configuração indica que o estado final 2 foi atingido e que toda a entrada foi lida. Podemos assim dizer que $aaabbb \in L_{ef}(A)$.

Se examinarmos o funcionamento de A para a entrada acima, e, através da função δ o funcionamento para as demais sequências em {a, b}*, podemos verificar:

- o primeiro símbolo deve ser um a
- o número de a's deve ser igual ao de b's.
- após o primeiro b, nenhum outro a pode ser aceito.
- após o último b, uma transição-ε leva A para uma configuração em que o estado é 2, e nenhuma transição adicional é possível.

Portanto, $L_{ef}(A) = \{ a^j b^j \mid j \geq 1 \}$. Por outro lado, nenhuma das transições prevê a retirada de X do fundo da pilha, de forma que $L_{pv}(A)=\emptyset$.

Exemplo 6.2: Seja o apnd $A = < K, \Sigma, \Gamma, \delta, i, I, F >$, com

$K = \{0, 1\}$ $\quad i = 0$

$\Sigma = \{a, b\}$ $\quad I = X$

$\Gamma = \{X, A\}$ $\quad F = \varnothing$

sendo a função $\delta:\{0,1\}\times\{a,b,\varepsilon\}\times\{X,A\} \rightarrow P(\{0,1\}\times\{X,A\}^*)$ dada por:

$\delta(0, a, X) = \{(0, A)\}$ $\quad \delta(0, b, A) = \{(1, \varepsilon)\}$

$\delta(0, a, A) = \{(0, AA)\}$ $\quad \delta(1, b, A) = \{(1, \varepsilon)\}$

Com entrada aaabbb, os seguintes são passos possíveis:

$[0, aaabbb, X] \vdash [0, aabbb, A] \vdash [0, abbb, AA] \vdash [0, bbb, AAA]$

$\vdash [1, bb, AA] \vdash [1, b, A] \vdash [1, \varepsilon, \varepsilon]$

Como toda a entrada foi lida e a pilha está vazia, temos $aaabbb \in L_{pv}(A)$.

Examinando o funcionamento de A, podemos verificar que $L_{pv}(A) = \{ a^j b^j \mid j \geq 1 \}$, ou seja, que a linguagem aceita por pilha vazia por este autômato de pilha é idêntica à linguagem aceita por estado final pelo autômato do exemplo anterior. Por outro lado, $F=\varnothing$ faz com que $L_{ef}(A)=\varnothing$.

Exercício 6.1: Construa um autômato de pilha A que aceita a linguagem dos dois exemplos anteriores, simultaneamente, por pilha vazia e por estado final, ou seja, construa um apnd A tal que $L_{ef}(A) = L_{pv}(A) = \{ a^i b^i \mid i \geq 1 \}$

Exemplo 6.3: Seja a glc $G_0 = <N, \Sigma, P, S>$, com $N = <\{E, T, F\}$,

$\Sigma = \{+, *, (, ), a\}, P, E>, S=E$ e P composto pelas regras:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow ( E ) \mid a$

O apnd A, a seguir, aceita $L(G_0)$, por pilha vazia:

$A = < \{q\}, \Sigma, N \cup \Sigma, \delta, q, E, \varnothing >$,

sendo $\delta$ dada por:

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

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

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

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

De acordo com a especificação da função $\delta$ acima, vemos que há dois tipos de transições possíveis neste apnd, conforme o símbolo de $N \cup \Sigma$ que aparece no topo da pilha:

- se o símbolo do topo da pilha é um nãoterminal de $G_0$, nenhum símbolo da entrada é lido, e há uma possibilidade para cada regra do nãoterminal.

  O apnd A depende do não-determinismo para escolher entre as diversas possibilidades aquela que corresponde à regra de $G_0$ usada no passo correspondente numa derivação esquerda da entrada x.

- se o símbolo do topo da pilha é um terminal de $G_0$, e o mesmo símbolo é lido da entrada.

  Esta transição só pode ser feita se a escolha (não-determinística) das transições do primeiro tipo foi feita de forma correta para a entrada.

Por exemplo, para a entrada (a+a)*a, temos a derivação esquerda

$$E \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow (E)*F \Rightarrow (E+T)*F \Rightarrow (T+T)*F \Rightarrow (F+T)*F$$
$$\Rightarrow (a+T)*F \Rightarrow (a+F)*F \Rightarrow (a+a)*F \Rightarrow (a+a)*a$$

A mesma seqüência de regras é usada, e passos dos dois tipos são intercalados, de acordo com a natureza do símbolo que aparece no topo da pilha:

$$[q, (a+a)*a, E] \vdash [q, (a+a)*a, T] \vdash [q, (a+a)*a, T*F] \vdash [q, (a+a)*a, F*F]$$
$$\vdash [q, (a+a)*a, (E)*F] \vdash [q, a+a)*a, E)*F] \vdash [q, a+a)*a, E+T)*F]$$
$$\vdash [q, a+a)*a, T+T)*F] \vdash [q, a+a)*a, F+T)*F]$$
$$\vdash [q, a+a)*a, a+T)*F] \vdash [q, +a)*a, +T)*F] \vdash [q, a)*a, T)*F]$$
$$\vdash [q, a)*a, F)*F] \vdash [q, a)*a, a)*F] \vdash [q, )*a, )*F] \vdash [q, *a, *F]$$
$$\vdash [q, a, F] \vdash [q, a, a] \vdash [q, \varepsilon, \varepsilon]$$

(Usamos aqui colchetes para representar as configurações, para evitar confusão com os parênteses de $\Sigma$.)

Veremos posteriormente que é possível construir, para qualquer glc G, de forma semelhante à utilizada neste último exemplo, um apnd que aceita L(G) por pilha vazia.

### *6.3 - Equivalência das duas formas de aceitação*

Mostraremos agora que as duas formas de aceitação são equivalentes, no sentido de que definem a mesma classe de linguagens. Naturalmente, isto não quer dizer que para o mesmo apnd A as linguagens $L_{ef}(A)$ e $L_{pv}(A)$ são iguais. A idéia aqui é que se $L = L_{ef}(A)$, então existe um apnd B (possivelmente diferente de A) tal que $L_{pv}(B) = L$, e vice-versa.

Teorema 6.1: Para qualquer apnd A, existe um apnd B tal que B aceita por estado final a mesma linguagem que A aceita por pilha vazia.

Dem.: Seja o apnd $A = <K, \Sigma, \Gamma, \delta, i, I, F >$.

Defina $B = < K \cup \{i', f'\}, \Sigma, \Gamma \cup \{I'\}, \delta', i', I', \{f'\} >$, com $\delta'$ dada por

$\delta'(i', \varepsilon, I') = \{(i, I\ I')\}$

para $q \in K$, $a \in \Sigma \cup \{\varepsilon\}$, $Z \in \Gamma$, $\delta'(q, a, Z) = \delta(q, a, Z)$

para $q \in K$, $\delta'(q, \varepsilon, I') = \{(f', \varepsilon)\}$

A idéia fundamental da construção de B é que, com entrada x,

- B passa de sua configuração inicial [i', x, I'] para a configuração [i, x, I I'], idêntica à configuração inicial de A, exceto pelo símbolo I' acrescentado no fundo da pilha;
- B simula todas as transições de A (exceto pela presença de I' no fundo da pilha), e quando A atinge uma configuração [q, ε, ε], indicando a aceitação de x, B atinge a configuração [q, ε, I'], com apenas o símbolo I' na pilha;
- B passa então dessa configuração para a configuração [f', ε, ε], que indica a aceitação da entrada x.

Esquematicamente, temos:

| configurações de A | configurações de B |
|---|---|
| | [i', x, I'] |
| [i, x, I] | [i, x, I I'] |
| .... | .... |
| [q, ?, ?] | [q, ?, I'] |
| | [f', ?, ?] |

Portanto, se A aceita x por pilha vazia, B aceita x por estado final. Para a recíproca, observamos que A pode esvaziar a pilha sem completar a leitura de sua entrada x, atingindo uma configuração [q, y, ε], com y≠ε, que não é uma configuração final. Simulando A, B pode atingir configurações [q, y, I'] e [f', y, I'], mas esta última, apesar do estado final, não é uma configuração de aceitação, pela mesma razão anterior.

Teorema 6.2: Para qualquer apnd A, existe um apnd B tal que B aceita por pilha vazia a mesma linguagem que A aceita por estado final.

Dem.: Seja $A = < K, \Sigma, \Gamma, \delta, i, I, F >$.

Defina $B = < K \cup \{i', d'\}, \Sigma, \Gamma \cup \{I'\}, \delta', i', I', \varnothing >$, com $\delta'$ dada por

$\delta'(i', \varepsilon, I') = \{(i, I\ I')\}$

para $q \in K$, $a \in \Sigma \cup \{\varepsilon\}$, $Z \in \Gamma$, $\delta'(q, a, Z) = \delta(q, a, Z)$

para $f \in F$, $Z \in \Gamma \cup \{I'\}$, $\delta'(f, \varepsilon, Z) = \{(d', Z)\}$

para $Z \in \Gamma \cup \{I'\}$, $\delta'(d', \varepsilon, Z) = \{(d', \varepsilon)\}$

A idéia fundamental da construção de B é que, com entrada x,

- B passa de sua configuração inicial [i', x, I'] para a configuração [i, x, I I'], idêntica à configuração inicial de A, exceto pelo símbolo I' acrescentado no fundo da pilha;
- B simula todas as transições de A (exceto pela presença de I' no fundo da pilha), e quando A atinge uma configuração [f, ε, γ], indicando a aceitação de x, B atinge a configuração [f, ε, γI'], com o símbolo I' acrescentado no fundo da pilha;

- B passa então dessa configuração para a configuração [d', ε, γI'], preparando-se para esvaziar a pilha, para indicar a aceitação da entrada x.
- A única ação possível no estado d' é a remoção do símbolo do topo da pilha, de forma que B atinge finalmente a configuração de aceitação [d', ε, ε].

Esquematicamente, temos

| configurações de A | configurações de B |
|---|---|
| | [i', x, I'] |
| [i, x, I] | [i, x, I I'] |
| .... | .... |
| [f, ε, γ] | [f, ε, γI'] |
| | [d', ε, γI'] |
| | …. |
| | [d', ε, I'] |
| | [d', ε, ε] |

Portanto, se A aceita x por estado final, B aceita x por pilha vazia. Para a recíproca, observamos que A pode esvaziar a pilha sem aceitar sua entrada x, atingindo uma configuração [q, y, ε], com $q \in F$, que não é uma configuração final. Simulando A, B pode atingir a configuração [q, y, I'], mas, como q não é final, o estado d' não pode ser atingido, e I' não pode ser removido da pilha.

### *6.4 - Equivalência das classes de linguagens definidas por apnd's e glc's*

Esta seção mostra que os apnd's reconhecem exatamente as llc's, isto é, as linguagens definidas por glc's. Devido aos resultados da seção anterior, basta considerar aqui uma forma de aceitação, no caso, a aceitação por pilha vazia. O primeiro resultado corresponde ao Teorema abaixo.

Teorema 6.3: Seja L uma llc. Então existe um apnd que aceita L por pilha vazia.

Dem.: Se L é uma llc, então existe uma glc $G = < N, \Sigma, P, S >$ tal que L=L(G). A demonstração é feita através da construção de um apnd A que aceita L por pilha vazia a partir da gramática G, de forma semelhante à utilizada em um dos exemplos vistos anteriormente.

O apnd em questão tem apenas um estado: $A = < \{q\}, \Sigma, N \cup \Sigma, \delta, q, S >$, com δ como se segue:

- para cada nãoterminal A, se as regras de A são $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, … $A \rightarrow \alpha_n$, temos $\delta(q, \varepsilon, A) = \{ (q, \alpha_1), (q, \alpha_2), \ldots (q, \alpha_n) \}$.
- para cada terminal a, temos $\delta(q, a, a) = \{ (q, \varepsilon) \}$.

A prova de que $L_{pv}(A) = L(G)$ se baseia na relação entre uma computação de A (uma seqüência de configurações sucessivas de A) que aceita uma cadeia x, e uma derivação esquerda da mesma cadeia x. Deixamos como exercício completar os

detalhes da demonstração. O Exemplo 6.3 permite ver, para um caso particular, a correspondência entre os passos da derivação de uma cadeia x em G e as configurações assumidas por A até a aceitação de x.

O segundo resultado corresponde ao Teorema 6.4. A demonstração encontrada na referência citada anteriormente difere ligeiramente da apresentada aqui.

Teorema 6.4: Se L é uma linguagem aceita por pilha vazia por um apnd A, então L é uma llc.

Dem.: o teorema decorre dos Lemas 1 e 2 a seguir.

Lema 1: Para cada apnd A, existe um apnd B satisfazendo as duas condições:

- $L_{pv}$ (B) = $L_{pv}$ (A).
- B só tem um estado.

Dem.: Seja A = < K, Σ, δ, i, I, >. Vamos construir o apnd equivalente

B = <{ u }, Σ, Γ', δ', u, X, ∅ >, com Γ' = { X } ∪ (K×Γ×K).

Como o estado do apnd B não contém nenhuma informação (é sempre u), durante a simulação de A por B, a informação sobre o estado de A deve ser anotada na pilha de B. À primeira vista, bastaria anotar o estado $s_1$ de A no topo da pilha de B, no momento do empilhamento de um símbolo Z, juntamente com esse símbolo: [$s_1$, Z]. Entretanto, neste caso, quando Z fosse desempilhado, toda a informação sobre o estado de A se perderia. Por essa razão, vamos anotar também, com cada símbolo Z, o estado $s_2$, o estado assumido por A quando esta particular instância de Z for desempilhada. Note que se Z é substituído por outro símbolo Z', o mesmo estado $s_2$ será descoberto quando Z' for retirado.

Os símbolos do alfabeto Γ' são triplas [$s_1$, Z, $s_2$ ], com um símbolo Z de Γ e dois estados $s_1$ e $s_2$. O símbolo X de B será usado apenas como símbolo inicial, e não precisa ser tratado da mesma forma.

Assim, a cada símbolo Z são acrescidos dois estados $s_1$ e $s_2$: $s_1$ será o estado de A quando esta ocorrência de Z aparecer no topo; $s_2$ será o estado de A quando o símbolo do topo passar a ser aquele imediatamente abaixo de Z. Para garantir a correção da simulação de A, dois símbolos [$s_1$, $Z_1$, $s_2$ ] e [$s_3$, $Z_2$, $s_4$ ], que apareçam em posições consecutivas na pilha de B, devem sempre satisfazer a propriedade $s_2 = s_3$. Assim, quando $Z_2$ se tornar o estado do topo, o estado será $s_2$, aquele estado previsto (de forma não deterministica) por ocasião do empilhamento de $Z_1$.

Adicionalmente, se a pilha só contem um símbolo [$s_1$, Z, $s_2$ ], $s_2$ será o estado de A em que a pilha ficará vazia. A definição da função de transição δ' de B garante estas propriedades:

- transição geral:

  Se ($q_1$, $Z_1Z_2$ … $Z_n$) ∈ δ(q, a, Z),

  então (u, [$q_1$, $Z_1$, $q_2$][$q_2$, $Z_2$, $q_3$] … [$q_n$, $Z_n$, $q_{n+1}$]) ∈ δ'(u, a, [q, Z, $q_{n+1}$]),

  para quaisquer valores de $q_2$, …, $q_n$ escolhidos em K.

- caso particular n = 0:

  Se ($q_1$, ε) ∈ δ(q, a, Z), então (u, ε) ∈ δ'(u, a, [q, Z, $q_1$]).

- início da simulação:

para cada $q \in K$, $(u, [i, I, q]) \in \delta'(u, \varepsilon, X)$.

Note que B prepara a simulação, colocando na pilha de B um símbolo [i, I, q], com a seguinte informação: o estado inicial i de A, o símbolo inicial I de A, e a indicação de que eventualmente, quando a pilha se tornar vazia, o estado será o estado q. O estado q é escolhido de forma não determinística, razão pela qual todas as possibilidades estão previstas no caso do início da simulação.

O restante da demonstração consiste em mostrar que a cada computação de A corresponde uma computação de B, e vice versa, de forma que A e B aceitam a mesma linguagem por pilha vazia. Novamente, os detalhes são deixados como exercício.

Exemplo 6.4: Seja o apnd $A = < K, \Sigma, \Gamma, \delta, i, I, \varnothing >$, com $K=\{0,1\}$, $\Sigma=\{a, b\}$ e $\Gamma=\{I, a, b\}$, sendo $\delta$ dada pela tabela a seguir. Cada linha descreve um elemento $(p, \alpha)$ de $\delta(q, a, Z)$. As linhas estão numeradas para referência posterior.

| | q | a | Z | p | $\alpha$ |
|---|---|---|---|---|---|
| *1* | 0 | a | I | 0 | a |
| *2* | 0 | b | I | 0 | b |
| *3* | 0 | a | a | 0 | aa |
| *4* | 0 | a | b | 0 | ab |
| *5* | 0 | b | a | 0 | ba |
| *6* | 0 | b | b | 0 | bb |
| *7* | 0 | $\varepsilon$ | I | 1 | $\varepsilon$ |
| *8* | 0 | $\varepsilon$ | a | 1 | a |
| *9* | 0 | $\varepsilon$ | b | 1 | b |
| *10* | 1 | a | a | 1 | $\varepsilon$ |
| *11* | 1 | b | b | 1 | $\varepsilon$ |

O apnd A é essencialmente não determinístico, e aceita por pilha vazia a linguagem $\{ xx^R \mid x \in \{a, b\}^* \}$. Lembramos que $x^R$ indica a cadeia reversa ou refletida de x: se tivermos $x = x_1x_2\ldots x_n$, então $x^R = x_n\ldots x_2x_1$.

Vamos construir o apnd B que deve aceitar por pilha vazia a mesma linguagem. Temos $B = <\{ u \}, \Sigma, \Gamma', \delta', u, X, \varnothing >$, onde

$\Gamma' = \{ X, [0, I, 0], [0, I, 1], [1, I, 0], [1, I, 1], [0, a, 0], [0, a, 1],$

$[1, a, 0], [1, a, 1], [0, b, 0], [0, b, 1], [1, b, 0], [1, b, 1] \},$

e $\delta'$ é descrita pela tabela a seguir:

| q | a | Z | p | $\alpha$ | comentários |
|---|---|---|---|---|---|
| u | $\varepsilon$ | X | u<br>u | [0, I, 0]<br>[0, I, 1] | iniciação |
| u<br>u | a<br>a | [0, I, 0]<br>[0, I, 1] | u<br>u | [0, a, 0]<br>[0, a, 1] | 1 |
| u<br>u | b<br>b | [0, I,0]<br>[0, I, 1] | u<br>u | [0, b, 0]<br>[0, b, 1] | 2 |

| q | a | Z | p | α | comentários |
|---|---|---|---|---|---|
| u | a | [0, a, 0] | u | [0, a, 0] [0, a, 0] | 3 |
| | | | u | [0, a, 1] [1, a, 0] | |
| | | [0, a, 1] | u | [0, a, 0] [0, a, 1] | |
| | | | u | [0, a, 1] [1, a, 1] | |
| u | a | [0, b, 0] | u | [0, a, 0] [0, b, 0] | 4 |
| | | | u | [0, a, 1] [1, b, 0] | |
| u | a | [0, b, 1] | u | [0, a, 0] [0, b, 1] | |
| | | | u | [0, a, 1] [1, b, 1] | |
| u | b | [0, a, 0] | u | [0, b, 0] [0, a, 0] | 5 |
| | | | u | [0, b, 1] [1, a, 0] | |
| u | b | [0, a, 1] | u | [0, b, 0] [0, a, 1] | |
| | | | u | [0, b, 1] [1, a, 1] | |
| u | b | [0, b, 0] | u | [0, b, 0] [0, b, 0] | 6 |
| | | | u | [0, b, 1] [1, b, 0] | |
| u | b | [0, b, 1] | u | [0, b, 0] [0, b, 1] | |
| | | | u | [0, b, 1] [1, b, 1] | |
| u | ε | [0, I, 1] | u | ε | 7 |
| u | ε | [0, a, 0] | u | [1, a, 0] | 8 |
| u | ε | [0, a, 1] | u | [1, a, 1] | |
| u | ε | [0, b, 0] | u | [1, b, 0] | 9 |
| u | ε | [0, b, 1] | u | [1, b, 1] | |
| u | a | [1, a, 1] | u | ε | 10 |
| u | a | [1, b, 1] | u | ε | 11 |

A última coluna indica a associação das linhas da tabela de δ' com as linhas da tabela de δ.

Suponhamos agora que as máquinas A e B recebem como entrada x = aabaabaa. Vamos mostrar as configurações assumidas pelas duas máquinas, durante a análise de x.

| A | B |
|---|---|
| | [u, aabaabaa, X] |
| [0, aabaabaa, I] | [u, aabaabaa, [0, I, 1]] |
| [0, abaabaa, a] | [u, abaabaa, [0, a, 1]] |
| [0, baabaa, aa] | [u, baabaa, [0, a, 1][1, a, 1]] |
| [0, aabaa, baa] | [u, aabaa, [0, b, 1][1, a, 1][1, a, 1]] |
| [0, abaa, abaa] | [u, abaa, [0, a, 1][1, b, 1][1, a, 1][1, a, 1]] |
| [1, abaa, abaa] | [u, abaa, [1, a, 1][1, b, 1][1, a, 1][1, a, 1]] |
| [1, baa, baa] | [u, baa, [1, b, 1][1, a, 1][1, a, 1]] |
| [1, aa, aa] | [u, aa, [1, a, 1][1, a, 1]] |
| [1, a, a] | [u, a, [1, a, 1]] |
| [1, ε, ε] | [u, ε, ε] |

Como se pode ver, cada terceira componente de símbolo da pilha de B corresponde exatamente ao estado de A, no momento em que o símbolo (ou aquele que o substituiu) é retirado da pilha.

Lema 2: Se o apnd A tem apenas um estado, $L_{pv}$ (A) é uma llc.

Dem.: Seja A = < { u }, Σ, Γ, δ, u, I, ∅ > um apnd com apenas um estado u. Construímos uma glc G tal que L(G) = $L_{pv}$(A):

G = < N, Σ, P, I >

sendo as seguintes as regras de P:

se (u, γ) ∈ δ(u, a, Z), então P tem uma regra Z → aγ.

A demonstração consiste na verificação de que a cada computação (sequência de configurações) de A, corresponde uma derivação (esquerda) de G.

Exemplo 6.5: Considere o apnd B do Exemplo 6.4. A gramática G pode ser construída como se segue:

X → [0, I, 0] | [0, I, 1]

[0, I, 0] → a [0, a, 0]

[0, I, 1] → a [0, a, 1]

[0, I, 0] → b [0, b, 0]

[0, I, 1]→ b [0, b, 1]

[0, a, 0] → a [0, a, 0] [0, a, 0] | a [0, a, 1] [1, a, 0]

[0, a, 1] → a [0, a, 0] [0, a, 1] | a [0, a, 1] [1, a, 1]

[0, b, 0] → a [0, a, 0] [0, b, 0] | a [0, a, 1] [1, b, 0]

[0, b, 1] → a [0, a, 0] [0, b, 1] | a [0, a, 1] [1, b, 1]

[0, a, 0] → b [0, b, 0] [0, a, 0] | b [0, b, 1] [1, a, 0]

[0, a, 1] → b [0, b, 0] [0, a, 1] | b [0, b, 1] [1, a, 1]

[0, b, 0] → b [0, b, 0] [0, b, 0] | b [0, b, 1] [1, b, 0]

[0, b, 1] → b [0, b, 0] [0, b, 1] | b [0, b, 1] [1, b, 1]

[0, I, 1] → ε

[0, a, 0] → [1, a, 0]

[0, a, 1] → [1, a, 1]

[0, b, 0] → [1, b, 0]

[0, b, 1] → [1, b, 1]

[1, a, 1] → a

[1, b, 1] → b

Assim, podemos derivar x=aabaabaa, pela derivação correspondente às computações de A e de B vistas no exemplo anterior:

X ⇒ [0, I, 1] ⇒ a [0, a, 1] ⇒ a a [0, a, 1] [1, a, 1]

⇒ a a b [0, b, 1] [1, a, 1] [1, a, 1]

⇒ a a b a [0, a, 1] [1, b, 1] [1, a, 1] [1, a, 1]

$\Rightarrow$ a a b a [1, a, 1] [1, b, 1] [1, a, 1] [1, a, 1]

$\Rightarrow$ a a b a a [1, b, 1] [1, a, 1] [1, a, 1]

$\Rightarrow$ a a b a a b [1, a, 1] [1, a, 1]

$\Rightarrow$ a a b a a b a [1, a, 1]

$\Rightarrow$ a a b a a b a a

Exercício 6.3: Verificar se a gramática acima tem regras inúteis, isto é, regras que nunca são usadas na derivação de alguma sentença da linguagem.

Um assunto que não será discutido neste capítulo é a definição de um *autômato de pilha determinístico* (*apd*), a partir do qual são definidas as *linguagens determinísticas*. Esta classe é um subconjunto próprio da classe das (de todas as) llc's. A importância das linguagens determinísticas, pode ser deduzida do fato de que todas as linguagens de programação são tratadas como linguagens determinísticas pelos compiladores.

*(maio 99)*

# Linguagens Formais

# Capítulo 7: Máquinas de Turing

*José Lucas Rangel*

## *7.1 - Introdução*

Neste capítulo vamos estudar a máquina de Turing (mT), introduzida por Alan Turing como um modelo matemático do processo de computação. Sua estrutura simples é proposital, uma vez que Turing pretendia, com sua definição, chegar a um modelo que fosse universalmente aceito. A máquina que vamos apresentar aqui é apenas uma das diversas variantes encontradas na literatura, escolhida por ser suficientemente próxima de um sistema de computação real, e por não oferecer maiores dificuldades de definição.

A mT é o principal modelo usado para o estudo do que é ou não computável. De acordo com a tese de Church, todos os modelos razoáveis de procedimento são equivalentes, e a mT se revelou simples e flexível o suficiente para permitir todas as demonstrações dos resultados principais. Pode-se usar uma mT como aceitador ou reconhecedor, ou ainda como a implementação de um procedimento mais geral, que transforma uma cadeia de entrada em uma cadeia de saída. Neste capítulo vamos definir a mT e apresentá-la como o reconhecedor das linguagens tipo 0, e procurar mostrar, através de exemplos, sua universalidade. Demonstrar a tese de Church, naturalmente, está fora de cogitação, de maneira que a universalidade da máquina de Turing só pode ser *confirmada* pelo fato de que todos os procedimentos encontrados na prática podem ser implementados através de máquinas de Turing.

## *7.2 - A máquina de Turing*

Vamos definir a máquina de Turing com um controle finito, como todas as máquinas até aqui, e uma fita, que pode servir como dispositivo de entrada (inicialmente a cadeia de entrada está escrita na fita), como área de trabalho (memória, ou rascunho) , ou como dispositivo de saída (possíveis resultados estarão escritos na fita ao final da computação).

O modelo aqui apresentado tem uma fita “semi-infinita”: considera-se a fita arbitrariamente extensível apenas para o lado direito. Por convenção, a fita é finita, mas pode ser aumentada sempre que houver necessidade de mais espaço, com mais uma célula em branco. A justificativa para isso é que o que é computável não pode depender do tamanho do bloco usado como rascunho pelo matemático: sempre deve ser possível arranjar outro bloco e continuar escrevendo. O novo bloco do matemático estará em branco, como sempre estará em branco a parte da fita em que ainda nada foi escrito.

Por convenção, um trecho da fita em branco será representado por um ou mais símbolos “branco”. Além disso, vamos convencionar um branco não pode ser escrito, ou seja um trecho de fita em que ocorre o símbolo branco é um trecho onde nada foi escrito até agora. Assim, a qualquer momento, valem as seguintes propriedades:

- apenas um número finito de posições da fita pode conter um símbolo distinto de branco;

- após o primeiro branco da fita, todos os símbolos são brancos.

Isto acontece porque a máquina avança ou recua na fita uma célula de cada vez e, em um número finito de passos, apenas um número finito de posições da fita pode ter sido alcançado.

Alguns autores preferem falar de uma fita infinita, cujo conteúdo é de símbolos "branco", exceto nas posições que já foram visitadas; outros preferem dizer que a fita é potencialmente infinita e que é estendida a cada vez que se tenta andar para a direita a partir da última posição. O fato importante a ser considerado é que, a qualquer momento, a quantidade de informação contida na fita é finita, embora não haja um limite para a quantidade de informação que nela pode ser armazenada.

Essa informação pode ser representada por uma cadeia de caracteres composta pelos símbolos gravados na fita, da primeira posição até o primeiro branco (exclusive). Note que, ao contrário do que ocorre em gravadores de fita, consideramos que é a cabeça de leitura/gravação que se move, e não a fita.

Outros características poderiam ser acrescentadas ao nosso modelo, tais como:

- várias fitas;
- fitas extensíveis para ambos os lados;
- fitas com finalidade específica, por exemplo fitas para entrada e fitas para saída
- a máquina pode ter mais de uma cabeça de leitura por fita;
- em vez de fita, a máquina pode usar como memória um reticulado bi-dimensional;
- a máquina pode ser não determinística.

De acordo com a tese de Church, entretanto, detalhes como estes não devem influenciar a capacidade de computação da mT. O modelo que vamos utilizar é um modelo determinístico bastante simples, próximo do mínimo necessário para uma máquina razoável, de acordo com a tese de Church. Discutiremos mais tarde a equivalência entre este modelo e suas variantes que possuem características como as mencionadas.

Definição. Uma máquina de Turing M é uma tupla $M = < K, \Sigma, \Gamma, \delta, i, F >$, onde K é um conjunto (finito, não vazio) de estados, $\Gamma$ é o alfabeto (finito) de símbolos da fita, $\Sigma \subseteq \Gamma$ é o alfabeto de símbolos de entrada, $\delta$ é a função de transição, $i \in K$ é o estado inicial, e $F \subseteq K$ é o conjunto de estados finais.

O símbolo branco será representado aqui por $\Diamond$. Este símbolo é um símbolo de $\Gamma$ que não pode ser escrito, nem pode fazer parte da entrada. Assim, temos $\Diamond \in \Gamma - \Sigma$.

A função de transição $\delta$ é um mapeamento $\delta: K \times \Gamma \rightarrow K \times \Gamma \times \{L, R\}$. Quando tivermos $\delta(q, a) = (p, b, R)$, a mT M, quando está no estado q, e lê o símbolo a na fita, escreve o símbolo b na mesma posição em que a estava escrito, move-se para a direita uma célula, e passa para o estado p. (Idem, para a esquerda no caso de $\delta(q, a) = (p, b, L)$ ). Por simplicidade, podemos deixar alguns valores de $\delta$ indefinidos, de maneira que $\delta$ deve ser entendida como uma função parcial.

Atingida uma situação em que o estado é q, o símbolo lido é a, e $\delta(q, a)$ é indefinido, dizemos que a máquina pára. Poderíamos, se desejado, definir máquinas de Turing que sempre param quando atingem um estado final, e que nunca param em um estado não final. Mas sempre que esta propriedade for desejada, podemos alterar

uma mT introduzindo um estado adicional, não final, do qual a máquina nunca mais sai.

*Configuração*. Para representar uma *configuração* (ou às vezes uma *descrição instantânea*) de uma mT usaremos uma cadeia $xqy \in \Gamma^* \bullet K \bullet \Gamma^*$, em que $xy \in \Gamma^*$ é o conteúdo da fita, sem incluir nenhum dos brancos que existem à direita, e $q \in K$ é o estado corrente. Para que esta notação seja usada, é necessário que os símbolos de $\Gamma$ sejam distintos dos símbolos usados para representar os estados de K, tornando impossível a confusão entre as partes constituintes da configuração, que são o estado q e o conteúdo xy da fita.

Esta notação tem a vantagem de dispensar a necessidade de indicar separadamente a posição da cabeça da máquina, uma vez que o próprio estado é usado para indicar essa posição. Assim, em uma configuração xqy, a posição onde a leitura e a escrita se realizam é o primeiro símbolo de y. Se y é a cadeia vazia, ou seja, se a configuração é da forma xq, a máquina lê um branco, à direita da parte escrita x da fita.

*Configurações inicial e final*. Para iniciar a computação, a configuração inicial para a entrada x é ix, composta pelo estado inicial i e pela cadeia de entrada x. O símbolo lido na configuração inicial é o primeiro símbolo de x, exceto no caso $x=\varepsilon$, em que o símbolo lido é um branco. Uma configuração xfy é final se o estado f é final. Como veremos a seguir, a mT pode, se for conveniente, aceitar sua entrada sem lê-la até o final.

*Mudança de configuração*. A definição da relação "mudança de configuração", $\vdash\!\!\!-$, não oferece surpresas:

- movimento para a direita:
  se $\delta(q, a) = (p, b, R)$, então $xqay \vdash\!\!\!- xbpy$
  (no estado q, lendo a, escreve b, anda para a direita e muda para o estado p)
- movimento para a direita, lendo um branco:
  se $\delta(q, \Diamond) = (p, b, R)$, então $xq \vdash\!\!\!- xbp$
  (caso particular do movimento para a direita: no estado q, lendo um branco, escreve b, anda para a direita e muda para o estado p)
- movimento para a esquerda:
  se $\delta(q, a) = (p, b, L)$, então $xcqay \vdash\!\!\!- xpcby$
  (no estado q, lendo a, escreve b, anda para a esquerda e muda para o estado p)
- movimento para a esquerda, lendo um branco:
  se $\delta(q, \Diamond) = (p, b, L)$, então $xcq \vdash\!\!\!- xpcb$
  (caso particular do movimento para a esquerda: no estado q, lendo um branco, escreve b, anda para a esquerda e muda para o estado p)

Naturalmente, se o valor de $\delta(q, a)$ é indefinido, nenhuma configuração pode ser alcançada a partir de uma configuração xqay, e a máquina pára. No caso particular $a=\Diamond$, se a configuração for xq, a máquina pára quando $\delta(q, \Diamond)$ é indefinido.

As assimetrias que podem ser notadas na definição da relação mudança de configuração $\vdash\!\!\!-$ são devidas às assimetrias na definição de configuração e na definição da própria máquina de Turing: *(1)* na configuração xqy o símbolo lido é o

primeiro de y, e *(2)* a fita é semi-infinita para a esquerda. Note que esta definição impede a passagem da cabeça para a esquerda do primeiro símbolo.

Notamos, para uso futuro, que a mudança de configuração pode ser descrita através da substituição de cadeias que envolvem no máximo três símbolos:

$qa \rightarrow bp$, $q \rightarrow bp$, $cqa \rightarrow pcb$ ou $cq \rightarrow pcb$,

O restante da configuração não se altera.

*Linguagem reconhecida por uma mT*. A linguagem aceita, ou reconhecida por uma mT M é definida por

$$L(M) = \{ x \in \Sigma^* \mid ix \vdash\!\!\!\!-^* yfz, \text{ onde } f \in F \}$$

ou seja, L(M) é o conjunto das cadeias x, compostas de símbolos do alfabeto de entrada, que levam M da configuração inicial ix (correspondente à entrada x) para alguma configuração final yfz (em que o estado f é final).

A aceitação se dá quando o estado final é atingido, não interessando em que ponto da fita está a cabeça, ou se há possibilidade de continuar a partir desse ponto. Se é possível atingir uma configuração cujo estado é final, a cadeia de entrada já está aceita. Caso contrário, a cadeia não será aceita. Não é sequer necessário que a entrada seja inteiramente lida.

Para não aceitar uma cadeia x (x não pertence à linguagem), a máquina pode parar em algum estado não final, ou pode simplesmente <u>não parar</u>, e continuar se movendo indefinidamente sem nunca aceitar.

<u>Exemplo 7.1</u>: Considere a mT $M = ( K, \Gamma, \Sigma, \delta, i, F )$, onde

$K = \{ q_0, q_1, q_2, q_3, q_4 \}$

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

$\Gamma = \{ a, b, X, Y, \Diamond \}$

$i = q_0$

$F = \{ q_4 \}$,

e $\delta$ é dada pela tabela abaixo, onde a notação foi simplificada, e cada valor da função $\delta$ em algum ponto é representado por uma quíntupla. Assim, se $\delta(q, a)=(p, b, D)$, representaremos este fato pela quíntupla qapbD. Quando nenhuma quíntupla iniciada por qa aparece, o valor de $\delta(q, a)$ é indefinido. (Como estamos considerando apenas o caso determinístico, no máximo pode ocorrer uma quíntupla para cada par (q, a).

| | | |
|---|---|---|
| $q_0aq_1XR$ | $q_0Yq_3YR$ | $q_0\Diamond q_4XR$ |
| $q_1aq_1aR$ | $q_1bq_2YL$ | $q_1Yq_1YR$ |
| $q_2aq_2aL$ | $q_2Xq_0XR$ | $q_2Yq_2YL$ |
| $q_3Yq_3YR$ | $q_3Yq_4XR$ | $q_3\Diamond q_4XR$ |

Considere a cadeia de entrada x = aaabbb. A seqüência de configurações assumida por M com entrada x é:

$q_0aaabbb \vdash\!\!\!\!- Xq_1aabbb \vdash\!\!\!\!- Xaq_1abbb \vdash\!\!\!\!- Xaaq_1bbb \vdash\!\!\!\!- Xaq_2aYbb$
$\vdash\!\!\!\!- Xq_2aaYbb \vdash\!\!\!\!- q_2XaaYbb \vdash\!\!\!\!- Xq_0aaYbb \vdash\!\!\!\!- XXq_1aYbb \vdash\!\!\!\!- XXaq_1Ybb$
$\vdash\!\!\!\!- XXaYq_1bb \vdash\!\!\!\!- XXaq_2YYb \vdash\!\!\!\!- XXq_2aYYb \vdash\!\!\!\!- Xq_2XaYYb \vdash\!\!\!\!- XXq_0aYYb$
$\vdash\!\!\!\!- XXXq_1YYb \vdash\!\!\!\!- XXXYq_1Yb \vdash\!\!\!\!- XXXYYq_1b \vdash\!\!\!\!- XXXYq_2YY$
$\vdash\!\!\!\!- XXXq_2YYY \vdash\!\!\!\!- XXq_2XYY \vdash\!\!\!\!- XXXq_0YYY \vdash\!\!\!\!- XXXYq_3YY$

$\vdash XXXYYq_3Y \vdash XXXYYYq_3 \vdash XXXYYYXq_4$

A mT M acima aceita a linguagem $\{ a^n b^n \mid n \geq 0 \}$. Podemos observar as ações correspondentes a cada um dos estados:

| | |
|---|---|
| $q_0$ | o primeiro a é marcado com um X, passa para $q_1$;<br>se não houver mais nenhum a, passa para $q_3$; |
| $q_1$ | procura o primeiro b, que é marcado com Y, passa para $q_2$; |
| $q_2$ | volta até o X, e passa para $q_0$ ; |
| $q_3$ | verifica que todos os símbolos já estão marcados:<br>vai para a direita até encontrar um ◊, e passa para $q_4$; |
| $q_4$ | aceita. |

Outros exemplos de computações:

$q_0 \vdash Xq_4$ (aceita)

$q_0ab \vdash Xq_1b \vdash q_2XY \vdash Xq_0Y \vdash XYq_3 \vdash XYXq_4$ (aceita)

$q_0ba$ (para sem aceitar)

$q_0aab \vdash Xq_1ab \vdash Xaq_1b \vdash Xq_2aY \vdash q_2XaY \vdash Xq_0aY \vdash XXq_1Y \vdash XXYq_1$ (para sem aceitar)

mostrando que ε e ab pertencem à linguagem, mas isso não acontece com ba e aab.

Exercício 7.1: Construa uma MT que aceite a linguagem $L = \{ a^n b^n c^n \mid n \geq 0 \}$.

### *7.3 - A máquina de Turing como modelo geral de procedimento.*

A máquina de Turing é, como já foi observado anteriormente, o modelo formal mais usado de procedimento. Pela tese de Church, todos os modelos razoáveis do processo de computação, definidos e por definir, são equivalentes. Por essa razão podemos usar a máquina de Turing como sendo nosso modelo formal de procedimento. Lembramos que, por sua natureza, a tese de Church não admite prova formal.

A definição da linguagem aceita por uma mT caracteriza o uso de uma mT como procedimento aceitador, de forma que (via tese de Church) a classe das linguagens aceitas por máquinas de Turing e a classe dos conjuntos recursivamente enumeráveis se identificam. Semelhantemente, identificamos algoritmo com "mT que sempre pára" - ou seja, para qualquer entrada sempre atinge uma configuração a partir da qual não há nenhuma outra configuração acessível. Um conjunto recursivo é, então, a linguagem aceita por uma mT que sempre pára. Fazemos assim a correspondência entre dizer "Sim" e estar num estado final, e dizer "Não" e estar em um estado não final.

Exemplo 7.2. Seja construir uma máquina de Turing para converter um inteiro dado em notação unária, para a notação binária. (Em notação unária, o natural i é representado pela cadeia $1^i$, obtida pela concatenação de i instâncias do símbolo i.) Suporemos que entrada e saída da máquina estarão delimitadas entre dois $. Por exemplo, se a entrada é $1111$, a saída deve ser $100$, sendo permitida a presença adicional de algum "lixo" na fita.

A idéia usada para a construção da mT é a seguinte: se um número é ímpar, o último bit" de sua representação binária é 1; se é par, 0. Para obter os demais bits, o número é dividido por 2. Na fita da mT, dividir por 2 é cancelar um símbolo sim, um não. Isto combina com a determinação da paridade do número: se não sobrar nenhum, o número é par.

Podemos usar a seguinte máquina, descrita através de quíntuplas:

| | | | | |
|---|---|---|---|---|
| A$B$R | | A1A1L | AxAxL | |
| B$H$R | | B1CxR | BxBxR | |
| C$E$R | | C1D1R | CxCxR | |
| D$F$R | | D1CxR | | |
| | E0E0R | E1E1R | | E◊1GL |
| | F0F0R | F1F1R | | F◊0GL |
| G$A$L | G0G0L | G1G1L | | |
| | H0H0R | H1H1R | | H◊I$L |
| I$M$R | I0JxR | I1KxR | IxIxL | |
| J$J$R | J0J0R | J1J1R | JxJxR | J◊L0L |
| K$K$R | K0K0R | K1K1R | KxKxR | K◊L1R |
| L$I$L | L0L0L | L1L1L | | |
| M$M$R | M0M0R | M1M1R | MxMxR | M◊N$R |

Nesta mT, o estado inicial é A, e o final é N. A computação abaixo corresponde à entrada $1111$. A última parte da computação corresponde à inversão da representação binária, que é necessária porque os bits do resultado são gerados a partir do menos significativo. No nosso caso, $4 = 1111_1 = 100_2$ aparece como 001, e precisa ser invertido.

| | | |
|---|---|---|
| A$1111$ | $B1111$ | $xC111$ |
| $x1D11$ | $x1xC1$ | $x1x1D$ |
| $x1x1$F | $x1x1G$0 | $x1xA1$0 |
| $x1Ax1$0 | $xA1x1$0 | $Ax1x1$0 |
| A$x1x1$0 | $Bx1x1$0 | $xB1x1$0 |
| $xCxx1$0 | $xxCx1$0 | $xxxC1$0 |
| $xxx1D$0 | $xxx1$F0 | $xxx1$0F |
| $xxx1$G00 | $xxx1G$00 | $xxxA1$00 |
| $xxAx1$00 | $xAxx1$00 | $Axxx1$00 |
| A$xxx1$00 | $Bxxx1$00 | $xBxx1$00 |
| $xxBx1$00 | $xxxB1$00 | $xxxxC$00 |
| $xxxx$E00 | $xxxx$0E0 | $xxxx$00E |
| $xxxx$0G01 | $xxxx$G001 | $xxxxG$001 |
| $xxxAx$001 | $xxAxx$001 | $xAxxx$001 |
| $Axxxx$001 | A$xxxx$001 | $Bxxxx$001 |
| $xBxxx$001 | $xxBxx$001 | $xxxBx$001 |
| $xxxxB$001 | $xxxx$H001 | $xxxx$0H01 |
| $xxxx$00H1 | $xxxx$001H | $xxxx$00I1$ |
| $xxxx$00xK$ | $xxxx$00x$K | $xxxx$00xL$1 |
| $xxxx$00Ix$1 | $xxxx$0I0x$1 | $xxxx$0xJx$1 |
| $xxxx$0xxJ$1 | $xxxx$0xx$J1 | $xxxx$0xx$1J |

| | | |
|---|---|---|
| $xxxx$0xx$L10 | $xxxx$0xxL$10 | $xxxx$0xIx$10 |
| $xxxx$0Ixx$10 | $xxxx$I0xx$10 | $xxxx$xJxx$10 |
| $xxxx$xxJx$10 | $xxxx$xxxJ$10 | $xxxx$xxx$J10 |
| $xxxx$xxx$1J0 | $xxxx$xxx$10J | $xxxx$xxx$1L00 |
| $xxxx$xxx$L100 | $xxxx$xxxL$100 | $xxxx$xxIx$100 |
| $xxxx$xIxx$100 | $xxxx$Ixxx$100 | $xxxxI$xxx$100 |
| $xxxx$Mxxx$100 | $xxxx$xMxx$100 | $xxxx$xxMx$100 |
| $xxxx$xxxM$100 | $xxxx$xxx$M100 | $xxxx$xxx$1M00 |
| $xxxx$xxx$10M0 | $xxxx$xxx$100M | $xxxx$xxx$100$N |

Exercício 7.2. Construa uma máquina de Turing que faça a conversão inversa, isto é, recebendo como entrada a representação binária de um número natural, constrói sua representação unária.
Sugestão: A saída desejada pode ser construída usando o seguinte ciclo: se o bit for 1, acrescente um 1 à saída; dobre o número de 1's para todos os bits exceto o mais significativo. Por exemplo, para converter 1101 (13) podemos fazer:
para o bit 1, acrescenta 1 (1);
para o bit 1, dobra (11) e acrescenta 1 (111)
para o bit 0, dobra (111111)
para o bit 1, dobra (111111111111) e acrescenta 1 (1111111111111)

Como a máquina de Turing só trabalha com símbolos e cadeias, para que uma máquina de Turing possa trabalhar com conjuntos que não são linguagens, é necessário usar uma codificação, usando cadeias de símbolos para representar os elementos do conjunto. Na nossa definição informal de procedimento, não foi feita nenhuma restrição sobre os valores aceitáveis como entradas para procedimentos, mas esta restrição existe para a mT: apenas cadeias de símbolos podem ser usadas como entradas ou saídas. Conjuntos que não são linguagens só podem ser tratados por um procedimento através de alguma codificação de seus elementos em cadeias de alguma linguagem. Por exemplo: podemos dizer que o conjunto $Nat^2$ de pares de naturais é r.e., e justificar esta afirmativa apresentando o procedimento enumerador (ou gerador) abaixo.

1. Faça i=0.
2. Para j=0, ..., i, emita (i-j, j).
3. Incremente i de 1.
4. Vá para 2.

Neste procedimento, não foi especificada a forma de representação dos números naturais, ou de seus pares, de forma que qualquer representação pode ser usada.

Entretanto, usando máquinas de Turing como reconhecedores, só podemos mostrar que $Nat^2$ é r.e. descrevendo uma linguagem cujos elementos representam os elementos de $Nat^2$, e uma mT que reconhece esta linguagem. As linguagens $L_1$, $L_2$ e $L_3$ abaixo são exemplos de linguagens que podem ser usadas com essa finalidade.

$L_1 = \{ a^i b^j \mid i, j \in Nat \}$
$L_2 = \{ \$ 1^i \$ 1^j \$ \mid i, j \in Nat \}$
$L_3 = \{ \$ x \$ y \$ \mid x, y \in Z\}$, onde $Z = \{ 0 \} \cup \{1\} \bullet \{ 0, 1 \}^*$

Nas duas primeiras codificações, o par (2, 3) é representado respectivamente por aabbb ou por $11$111$. No terceiro caso, usamos a representação binária dos dois elementos do par, de forma que (2, 3) corresponde a $10$11$.

A única propriedade exigida de uma codificação, é que deve ser definida por uma bijeção. Tomando $L_1$ como exemplo, a cada elemento de $Nat^2$ corresponde exatamente um elemento de $L_1$=L[a*b*], e vice-versa. Já que existe a codificação, para mostrar que $Nat^2$ é r.e., basta mostrar que $L_1$ é r.e., o que pode ser feito neste caso observando que $L_1$ é regular.

Exercício 7.3: Construa máquinas de Turing que aceitem as linguagens $L_1$, $L_2$ e $L_3$.

Sugestão: Como as linguagens são regulares, construa afd's, que podem então ser transformados em mT's que nunca se movem para a esquerda.

Para representar um procedimento qualquer com uma mT precisamos permitir a geração de cadeias de saída, uma vez que a fita da mT funciona como memória e como dispositivo de entrada/saída. As cadeias geradas como saída de um procedimento devem ser escritas na fita. Isto vale para o caso de um procedimento gerador, que enumera uma linguagem infinita, e que portanto não pode parar (veja Exemplo 7.3), ou para um procedimento que calcula uma função que transforma cadeias de entrada em cadeias de saída (veja Exemplo 7.2).

Exemplo 7.3: Vamos descrever a mT M que enumera os elementos de $Nat^2$, codificados usando elementos de $L_1$. Note que M não recebe nenhuma entrada. As diversas cadeias geradas estarão separadas por $'s.

1. Inicialmente, M escreve $$ na fita, enumerando o par (0, 0).
2. Para cada cadeia $x=a^ib^j$ não considerada na fita,
   M acrescenta no fim da fita a cadeia $ax\$xb\$ = a^{i+1}b^j\$a^ib^{j+1}\$$,

correspondente à enumeração de (i+1, j) e de (i, j+1).

Em seguida M marca a cadeia $a^ib^j$ como já considerada.

O conteúdo da fita de M, após alguns passos, seria

$$a$b$aa$ab$ab$bb$aaa$aab$

onde $ marca a separação entre as cadeias já consideradas e as ainda não consideradas. O processo se repete indefinidamente. Observe que é necessário conhecer a codificação, para identificar quais os elementos de $Nat^2$ que foram enumerados.

A construção da máquina completa fica como exercício.

Exercício 7.4: Descreva mT's que enumeram $Nat^2$ de acordo com as outras duas codificações consideradas.

Exemplo 7.4: Vamos construir M, uma mT que soma dois números em unário, ou seja um procedimento que implementa a soma unária. M recebe como entrada uma cadeia A da forma $x$y$, em que x e y são cadeias de 1's, representando dois naturais i e j em unário. Ao final o resultado z (a cadeia de 1's que representa i+j) deve ficar na fita, delimitado por $'s; para "remover" quaisquer caracteres que não façam parte de z, usaremos o símbolo X.

A idéia é a seguinte:

substituímos o segundo $ por 1;
substituímos o último 1 por $;
substituímos o terceiro $ por X.

Assim, com entrada $11$111$, representando as parcelas 2 e 3, teremos

$11$111$ ⊢—* $111111$ ⊢—* $11$11$$ ⊢—* $11111$X

Ignorando o X, esta última cadeia é a representação do resultado 5. Outro exemplo seria 0+0=0, em que a entrada $$$ deveria ser transformada em $$X.

A máquina de Turing pode ter estados { A, … F } sendo A inicial e F final, com as seguintes transições:

A$B$R
B1B1R B$C1R
C1C1R C$D$L
D1E$R
E$FXR

de maneira que para a entrada $11$111$ teríamos a seguinte computação:

A$11$111$ ⊢— $B11$111$ ⊢— $1B1$111$ ⊢— $11B$111$ ⊢— $111C111$
⊢— $1111C11$ ⊢— $11111C1$ ⊢— $111111C$ ⊢— $11111D1$ ⊢— $11111$E$
⊢— $11111$XF

ou seja, 2+3=5. Semelhantemente, para a entrada $$$ teríamos

A$$$ ⊢— $B$$ ⊢— $1C$ ⊢— $D1$ ⊢— $$E$ ⊢— $$XF

ou seja, 0+0=0.

Exemplo 7.5: Seja descrever uma mT M, que soma dois números em binário, ou seja mais uma mT que implementa o procedimento soma. M recebe como entrada uma cadeia A da forma $x$y$, em que x e y são cadeias de 0's e 1's, representando dois naturais i e j em binário. Vamos supor que o resultado z (a cadeia que representa i+j) deve ficar no fim da parte escrita, com delimitadores especiais: $$z$$. (Antes do primeiro $$ fica o "lixo" resultante das computações.)

O plano de execução de M é o seguinte:

M “decora” um bit de x, soma com o bit correspondente de y e escreve no fim da fita o bit correspondente de z. Se houver um “carry” (“vai-um”), este deve ser considerado da próxima vez. Este processo é repetido até que os bits de x ou de y estejam todos marcados. A partir deste ponto, só os bits da outra cadeia são levados em consideração, até serem todos marcados. Note que os bits de z estão sendo gerados na ordem invertida, de forma que a sentença no fim da fita é efetivamente $z^R$. Para terminar, os bits de z são copiados um a um, a partir do último, para o fim da fita, onde a cadeia z é construída.

Note que os bits decorados e o carry devem ser anotados em estados especiais para cada combinação de valores. Os conteúdos da fita em alguns pontos da computação estão mostrados a seguir, para a soma 3+7=10.

$11$111$
$1x$11x$0
$xx$1xx$01
$xx$xxx$010
$xx$xxx$0101
$xx$xxx$010x$$1
$xx$xxx$01xx$$10

$xx$xxx$0xxx$$101
$xx$xxx$xxxx$$1010
$xx$xxx$xxxx$$1010$$

Outra possibilidade é a seguinte: usamos técnicas semelhantes às do Exemplo 7.2 e do Exercício 7.2, para as conversões de binário para unário e vice-versa; a soma pode ser feita em unário. Neste caso, podemos ter conteúdos de fita, em alguns instantes da computação, como os seguintes:

| | |
|---|---|
| $11$111$ | $3 = 11_2$; $7=111_2$ |
| $xx$111$111$ | $3= 111_1$ |
| $xx$xxx$111$1111111$ | $7 = 1111111_1$ |
| $xx$xxx$1111111111$x | $10 = 1111111111_1$ |
| $xx$xxx$xxxxxxxxxx$x$1010$ | $10 = 1010_2$ |

A efetiva construção da máquina, por uma das duas maneiras mencionadas, fica como um exercício.

## *7.4 - Técnicas para a construção de máquinas de Turing*

Vamos apresentar a seguir algumas técnicas de construção de mT's. Em virtude do interesse apenas teórico das mT's, apenas algumas máquinas muito pequenas são efetivamente construídas. Isso se deve ao fato de que a construção de uma mT pode ser extremamente trabalhosa. Assim, normalmente é feita apenas uma descrição da máquina, como fizemos nos exemplos 7.3 e 7.5. Vamos examinar aqui algumas técnicas, que podem ser usadas para efetivamente construir mT's, mas que são mais frequentemente usadas em descrições de mT's. Vamos apresentar também algumas modificações do modelo básico, que não alteram a capacidade de computação da mT.

1. *anotação de informações no controle finito*. A mT pode mudar de estado para guardar uma informação adicional. Por exemplo, a mT pode decorar o símbolo que encontrou em uma posição, para movê-lo para outra, ou para compará-lo com outro símbolo em outra posição.
2. *cópia e verificação de símbolos*, possívelmente em ordem inversa. Uma mT pode decorar e marcar símbolos em uma parte A (devidamente demarcada) da fita, para alguma ação em outra parte B (também devidamente demarcada) da fita. O símbolo decorado em A pode ser copiado em B, ou pode ser comparado com outro símbolo em B para verificar se duas cadeias são iguais. Se em A começamos pelo último símbolo, e em B pelo primeiro, a operação se dá na ordem inversa.
3. *Múltiplas trilhas*. Podemos incluir no alfabeto $\Gamma$ símbolos que sejam tuplas de outros símbolos. Por exemplo, $\Gamma$ pode conter pares de elementos de $\Sigma$. Esta construção permite simular a existência de várias trilhas na fita. Por exemplo, usando triplas como símbolos, $(a_1, b_1, c_1)$ $(a_2, b_2, c_2)$ ... $(a_n, b_n, c_n)$ pode ser vista como uma cadeia de n símbolos (que são triplas) ou pode ser vista como a combinação de três cadeias, uma em cada trilha: $a_1a_2...a_n$, $b_1b_2...b_n$, e $c_1c_2...c_n$.

A situação está representada na figura abaixo, onde podemos ver a cadeia de triplas,

| | | | | | |
|---|---|---|---|---|---|
| … | (a1, b1, c1) | (a2, b2, c2) | … | (an, bn, cn) | … |

ou, alternativamente, três trilhas separadas.

| | | | | | |
|---|---|---|---|---|---|
| | a1 | a2 | … | an | |
| … | b1 | b2 | … | bn | … |
| | c1 | c2 | … | cn | |

Em alguns casos, queremos verificar alguma condição sobre uma cadeia x, o que usualmente destrói a cadeia, mas, para alguma finalidade, a cadeia deve ser preservada. Podemos construir uma cópia de x na "outra" trilha, simplesmente trocando cada símbolo $a_i$ de x por $(a_i, a_i)$. Se $x = a_1a_2...a_n$, então x pode ser substituído por $(a_1, a_1)$ $(a_2, a_2)$ ... $(a_n, a_n)$. Uma das cópias pode ser destruída no processamento, e ao final x é reconstituído a partir da cópia salva na outra trilha.

4. *Subrotinas*. Podemos considerar que qualquer ação para a qual uma mT foi construída (ou descrita) pode ser usada na construção de outra mT, em que a ação é desejada como parte do processamento. Basta para isso incluir uma cópia dos estados e transições da subrotina. Se quisermos fazer várias chamadas, entretanto, será necessário dar novos nomes aos estados, para isolar uma chamada da sub-rotina das demais.

5. *Fita infinita nos dois sentidos*. Se for interessante, podemos considerar que a fita da mT se estende de forma ilimitada também para a esquerda da posição inicial. Para a simulação de uma mT com fita infinita nos dois sentidos, bastaria considerar uma fita semi-infinita com duas trilhas, cada qual representando uma das "metades". Podemos anotar no controle finito em qual das duas metades da fita a mT está no momento, e ignorar a outra metade.

Por exemplo, se o conteúdo da fita "infinita" é

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| … | $a_{-3}$ | $a_{-2}$ | $a_{-1}$ | $a_0$ | $a_1$ | $a_2$ | $a_3$ | … |

podemos usar uma fita semi-infinita para representar o mesmo conteúdo:

| | | | | |
|---|---|---|---|---|
| $a_0$ | $a_1$ | $a_2$ | $a_3$ | … |
| $a_{-1}$ | $a_{-2}$ | $a_{-3}$ | $a_{-4}$ | … |

ou seja, por

| | | | | |
|---|---|---|---|---|
| $(a_0, a_{-1})$ | $(a_1, a_{-2})$ | $(a_2, a_{-3})$ | $(a_3, a_{-4})$ | … |

6. *Várias fitas*. Podemos considerar uma mT com várias fitas, a que podem ser atribuídas funções diferentes, caso desejado. Por exemplo, podemos considerar que a máquina tem uma fita de entrada, uma fita de saída, e uma ou mais fitas de memória. A simulação de uma mT com n fitas pode ser feita em uma mT com um fita só, cujos símbolos são tuplas, que contém todos os símbolos de todas as fitas, e marcas com as posições de todas as cabeças.

7. *Máquinas de Turing não determinísticas*. Para definir uma máquina não determinística, devemos considerar uma função $\delta$ que pode oferecer mais de uma possibilidade de transição, a partir de cada configuração. Uma mT não determinística pode ser simulada por uma mt determinística. Isto será descrito a seguir.

Definição. Uma máquina de Turing não determinística M pode ser definida por uma tupla $M = < K, \Sigma, \Gamma, \delta, i, F >$, onde K é um conjunto (finito, não vazio) de estados, $\Gamma$ é

o alfabeto (finito) de símbolos da fita, $\Sigma \subseteq \Gamma$ é o alfabeto de símbolos de entrada, $\delta$ é a função de transição, $i \in K$ é o estado inicial, e $F \subseteq K$ é o conjunto de estados finais.

A função de transição $\delta$ é um mapeamento $\delta: K \times \Gamma \rightarrow P(K \times \Gamma \times \{L, R\})$. Todos os valores de $\delta$ são conjunto finitos.

*Configuração*. Da mesma maneira que para uma mT determinística, uma *configuração* de uma mT é uma cadeia xqy, em que $xy \in \Gamma^*$ é o conteúdo da fita, sem incluir nenhum dos brancos que existem à direita, e $q \in K$ é o estado corrente. Semelhantemente, a configuração inicial para a entrada x é ix, e uma configuração xfy é final se o estado f é final.

*Mudança de configuração*. A definição da relação "mudança de configuração", $\vdash\!\!\!-$, não oferece surpresas:

- movimento para a direita:
  se $(p, b, R) \in \delta(q, a)$, então $xqay \vdash\!\!\!- xbpy$
  caso particular: se $(p, b, R) \in \delta(q, \Diamond)$, então $xq \vdash\!\!\!- xbp$
- movimento para a esquerda:
  se $(p, b, L) \in \delta(q, a)$, então $xcqay \vdash\!\!\!- xpcby$
  caso particular: se $(p, b, L) \in \delta(q, \Diamond)$, então $xcq \vdash\!\!\!- xpcb$

Naturalmente, se $\delta(q, a)=\varnothing$, nenhuma configuração pode ser alcançada a partir de uma configuração xqay, e a máquina pára. Caso particular: se a configuração for xq, a máquina pára quando $\delta(q, \Diamond)=\varnothing$.

*Linguagem reconhecida por uma mT não determinística*. A linguagem aceita, ou reconhecida por uma mT M é definida por

$$L(M) = \{ x \in \Sigma^* \mid ix \vdash\!\!\!-^* yfz, \text{ onde } f \in F \}$$

A aceitação se dá quando o estado final é atingido, não interessando em que ponto da fita está a cabeça, se há possibilidade de continuar a partir desse ponto, ou se há outras configurações não finais atingíveis com a mesma entrada. Como nos outros casos em que definimos máquinas não determinísticas, só nos interessa saber se existe ou não a possibilidade de atingir um estado final.

Exemplo 7.6. A mT a seguir é não determinística. A linguagem aceita por essa máquina é $\{ xx \mid x \in \{0,1\}^* \}$.

| | | | | |
|---|---|---|---|---|
| A0BxR | A1CxR | | | A◊KxR |
| B0B0R B0DyL | B1B1R | | | |
| C0C0R | C1C1R C1DyL | | | |
| D0D0L | D1D1L | DxExR | DyDyL | |
| E0FxR | E1GxR | | EyJyR | |
| F0F0R | F1F1R | | FyHyR | |
| G0G0R | G1G1R | | GyIyR | |
| H0DyL | | | HyHyR | |
| | I1DyL | | IyIyR | |
| | | | JyJyR | J◊KxR |

A escolha não determinística é feita em dois estados, B e C. Em ambos os casos, trata-se de adivinhar qual o primeiro símbolo da segunda metade da cadeia xx. No caso do

estado B este símbolo é um 0; no caso do estado C, trata-se de um 1. Os demais símbolos são apenas conferidos nos estados H e I. Abaixo, um exemplo de computação:

| | | | | |
|---|---|---|---|---|
| A11011101 | xC1011101 | x1C011101 | x10C11101 | x101C1101 |
| x10D1y101 | x1D01y101 | xD101y101 | Dx101y101 | xE101y101 |
| xxG01y101 | xx0G1y101 | xx01Gy101 | xx01yI101 | xx01Dyy01 |
| xx0D1yy01 | xxD01yy01 | xDx01yy01 | xxE01yy01 | xxxF1yy01 |
| xxx1Fyy01 | xxx1yHy01 | xxx1yyH01 | xxx1yDyy1 | xxx1Dyyy1 |
| xxxD1yyy1 | xxDx1yyy1 | xxxE1yyy1 | xxxxGyyy1 | xxxxyIyy1 |
| xxxxyyIy1 | xxxxyyyI1 | xxxxyyDyy | xxxxyDyyy | xxxxDyyyy |
| xxxDxyyyy | xxxxEyyyy | xxxxyJyyy | xxxxyyJyy | xxxxyyyJy |
| xxxxyyyyJ | xxxxyyyyxK | | | |

Teorema 7.1. Os modelos determinístico e não determinístico da máquina de Turing são equivalentes.

Dem. Basta mostrar como simular uma mT não determinística M em uma mT determinística M'. A fita de M' conterá diversas configurações de M. Se alguma delas for final, M' aceitará sua entrada. O símbolo $ será usado como separador de configurações.

1. Inicialmente, M' encontra na sua fita a entrada x de M. Acrescentando marcas e o estado inicial i de M, M' escreve na fita a configuração inicial de M para x, $C_0$=ix, de forma que a fita de M' contém $\$C_0\$$.

2. Para cada configuração C não considerada na fita, M' constrói no fim da fita, as configurações atingíveis a partir de C, separadas por $. Se M tem m opções de escolha, e as configurações atingíveis são $C_1$, $C_2$, ... $C_m$, neste passo a cadeia $\$C_1\$C_2\$\ldots\$C_m\$$ é acrescentada no final da fita.

3. Se alguma das configurações de M construídas por M' for final, M' pára e aceita a entrada.

Para construir as configurações, observamos que no máximo três símbolos são alterados, quando se passa de uma configuração C para uma configuração C' em um passo. Ou seja, boa parte do trabalho de construção de C' é simplesmente um trabalho de cópia.

Exercício 7.5. Construa uma máquina determinística que aceite a linguagem do Exemplo 7.6, sem usar a construção sugerida no Teorema 7.1.

## *7.5 - Linguagens tipo 0 e conjuntos recursivamente enumeráveis*

Vamos agora caracterizar a classe de linguagens aceitas por mT's, mostrando que se trata da mesma classe de linguagens geradas por gramáticas (tipo 0), ou seja, via tese de Church, que a classe dos conjuntos recursivamente enumeráveis é idêntica à classe das linguagens tipo 0. Este resultado será objeto dos dois teoremas a seguir.

Teorema 7.2: Toda linguagem tipo 0 é recursivamente enumerável.

Demonstração. Seja L uma linguagem tipo 0. Seja G uma gramática tal que L = L(G). Vamos descrever a construção de uma mt M, não determinística, que aceita L(G).

M ignora sua entrada, e escreve $S$ no fim da fita. Em seguida, M simula uma derivação em G escolhendo e aplicando não-deterministicamente uma regra em cada passo. Para isso, escolhida a regra $\alpha \rightarrow \beta$, M encontra (não-deterministicamente) uma ocorrência de $\alpha$ e substitui esta ocorrência por $\beta$. Quando esta derivação tiver como resultado a cadeia x, M terá em sua fita duas cópias de x que devem ser comparadas. Neste ponto M passa para um estado final.

Uma outra possibilidade de construção de M é através de uma mT determinística, que em vez de "adivinhar" as regras a serem usadas, testa todas, sucessivamente.

O próximo resultado é a recíproca do teorema acima.

Teorema 7.3. Toda linguagem recursivamente enumerável é tipo 0.

Demonstração. Seja L uma linguagem r.e. Portanto, L é aceita por alguma mT (determinística) M. Vamos mostrar que L é tipo 0, construindo uma gramática (tipo 0) G que aceita L. Uma derivação em G simula uma computação de M, e tem o seguinte aspecto:

$$S \Rightarrow^* ix\$x \Rightarrow^* yfz\$x \Rightarrow^* x$$

A partir do símbolo inicial S, derivamos uma cadeia da forma ix$x, onde i é o estado inicial de M, de forma que ix é a configuração inicial de M com entrada x. A simulação da computação de M será feita a partir dessa configuração inicial. A segunda cópia de x fica inalterada até o final. A cadeia x contém apenas símbolos do alfabeto de entrada de M, que são os terminais de M; o separador $ e os símbolos de estados de M são nãoterminais de G. Para esta fase as regras de G podem ser obtidas adaptando as regras de uma gramática sensível ao contexto de $\{ xx \mid x \in \Sigma^* \}$, acrescentando os símbolos nãoterminais i e $.

A segunda parte,

$$ix\$x \Rightarrow^* yfz\$x$$

simula a computação de M com entrada x. Para isto, as regras de G são construídas a partir das transições de M. Para cada valor $\delta(q, a)$, G tem uma ou mais regras que simulam a mudança de configuração. Se $\delta(q, a) = (p, b, R)$, basta uma regra é $qa \rightarrow bp$. Se $\delta(q, a) = (p, b, L)$, há necessidade de uma regra $cqa \rightarrow pcb$ para cada símbolo c do alfabeto da fita. O fim da fita de M é identificado pela presença do separador $. Portanto, quando M lê um branco, o separador $ é alcançado, e transições em que um $\Diamond$ é envolvido tem regras correspondentes das formas $q\$ \rightarrow pb\$$ e $cq\$ \rightarrow pcb\$$. Supondo que x pertence a L(M), uma configuração final yfz será eventualmente alcançada, com $ix \vdash^* yfz$. Portanto, em G, será possível a derivação $ix\$x \Rightarrow^* yfz\$x$, onde f é um estado final.

Na última fase, os restos da computação de M são removidos, restando apenas a cópia não usada da cadeia x. Para obter $yfz\$x \Rightarrow^* x$, G tem regras $f \rightarrow X$ para cada estado final f. O nãoterminal X pode ser visto como um “apagador” usando regras $cX \rightarrow X$ e $Xc \rightarrow X$ para cada símbolo c do alfabeto da fita, e $X\$ \rightarrow \varepsilon$, podemos eliminar todos os vestígios da simulação, e ficar apenas com a cadeia x.

Por construção, G gera x se e somente se M aceita x. Para a construção da primeira parte de G, veja o exercício abaixo.

Exercício 7.6: Seja Σ um alfabeto qualquer, e seja Σ' = Σ ∪ { i, $ }. Considere a linguagem L' no alfabeto Σ', dada por L' = { ix$x | x ∈ Σ'* }. Mostre que L' é sensível ao contexto, construindo uma gsc para L'.

Nota: na demonstração do teorema anterior, i e $ são nãoterminais.

*Este capítulo foi escrito a partir de uma versão inicial escrita com a colaboração de Luiz Carlos Castro Guedes*

*2jul99*

# Capítulo 8: O problema da parada.

# Decidibilidade e computabilidade.

*José Lucas Rangel*

## *8.1 - Introdução.*

Como observado no capítulo anterior, podemos substituir a definição informal de procedimento pela definição formal de máquina de Turing (mT). Já vimos que, dependendo da forma como a mT é construída, ela pode ser um procedimento gerador ou um procedimento reconhecedor; por outro lado, uma mT que sempre pára, qualquer que seja a sua entrada, é um algoritmo. Portanto, um conjunto recursivamente enumerável (r.e.) é uma linguagem reconhecida por uma mT; e um conjunto recursivo é uma linguagem aceita por uma mT que sempre pára.

Devemos, neste capítulo, apresentar exemplos de linguagens que mostrem que as definições de conjunto recursivamente enumerável e de conjunto recursivo são não triviais. Uma vez que já foi provada a inclusão da segunda classe na primeira, basta apresentar exemplos de conjuntos r.e. que não são recursivos, e de conjuntos enumeráveis que não são r.e. Efetivamente, o reconhecimento de um conjunto não recursivamente enumerável é, então, uma tarefa que não pode ser realizada por nenhuma máquina de Turing, por nenhum procedimento, ou seja, é uma tarefa incomputável.

Para que esses exemplos sejam apresentados, precisamos de alguns conceitos auxiliares.

## *8.2 - Enumeração de Máquinas de Turing.*

Para definir uma enumeração (e uma numeração) de todas as máquinas de Turing (mT) vamos introduzir uma codificação, que vai representar cada mT $M = <K, \Sigma, \Gamma, \delta, i, F>$ através de uma cadeia em um alfabeto adequado $\Delta$; a enumeração das mT, determinísticas e não determinísticas, será baseada em uma enumeração de $\Delta^*$. Para isso é conveniente fazer algumas suposições:

Os estados de K estão ordenados: $q_0, q_1, q_2, \ldots$, de tal forma que $i = q_0$.

De forma semelhante, os símbolos de $\Gamma$, $s_0, s_1, s_2, \ldots$ estão ordenados, de forma que $s_0$ é o símbolo especial branco $\Diamond$.

Assim, para representar M, usaremos um alfabeto $\Delta = \{q, s, |, \#, L, R\}$ de forma que o estado $q_i$ é representado por $q|^i$, e o símbolo $s_i$ é representado por $s|^i$. A representação então usa as cadeias q, q|, q||, q|||, …, s, s|, s||, s|||, … para os estados e símbolos; note, em particular, que q é o estado inicial, e s é o símbolo branco.

Os símbolos #, L, e R de $\Delta$ são usados respectivamente como separador e como indicadores de movimento para a esquerda, e para a direita.

Para representar a função $\delta$, devemos representar todos os valores de $\delta(q, s)$. Isso pode ser feito através de uma sequência de quíntuplas $(q_i, s_j, q_k, s_l, r)$, com $r \in \{L, R\}$, que indicam que $(qk, sl, r) \in \delta(qi, sj)$.

A cadeia correspondente à tupla (qi, sj, qk, sl, r) em Δ* é q|i s|j q|k s|l r. Se M é uma mT não determinística, podemos ter várias tuplas com os mesmos valores nos dois primeiros elementos.

Por exemplo, se tivermos a tupla ($q_1$, $s_2$, $q_3$, $s_4$, L), indicando que ($q_3$, $s_4$, L) ∈ δ($q_1$, $s_2$), a cadeia correspondente em Δ* será q|s||q|||s||||L. Para representar a função δ, concatenamos as representações das quíntuplas correspondentes a δ, separadas por #.

Para representar F, basta concatenar, sem separação, as cadeias que representam os estados. Por exemplo, F = {q1, q3, q5 } é representado por q|q|||q|||||. A representação de M se reduz apenas à representação de F e de δ, que podem ser separadas por ##.

Exemplo: Seja M = <K, Σ, Γ, δ, i, F>, onde

K = { p, q, r }
Σ = { a, b }
Γ = { ◊, a, b, X }
i = q
F = { r },

sendo δ dada por

δ(p, a) = {(q, a, R)}
δ(q, a) = {(q, X, R)}
δ(q, b) = {(r, a, L)}
δ(q, ◊) = {(r, X, L)}

Fazendo $q_0$ = q, $q_1$ = p, $q_2$ = r, $s_0$ = ◊, $s_1$ = a, $s_2$ = b, $s_3$ = X, temos a seguinte representação de M:

q||##q|s|qs|R#qs|qs|||R#qs||q||s|L#qsq||s|||L

Uma entrada aaab de M será codificada em Δ* como s|s|s|s||.

Com essa codificação (ou alguma outra codificação semelhante), é possível representar as mT's e suas entradas através de cadeias em Δ*, onde Δ é um alfabeto apropriado. Sabemos que Δ* é enumerável, e que podemos atribuir um natural a cada cadeia de Δ*, e fazer referência às cadeias de Δ* como $x_0$, $x_1$, $x_2$, … . Assim, podemos atribuir à mT representada por $x_i$ o número i, e fazer a referência a ela como a i-ésima máquina de Turing $M_i$.

Entretanto, para que a notação $M_i$ sempre represente uma mT, uma convenção adicional é necessária. Observe que nem todas as cadeias $x_i$ de Δ* correspondem a mT's. Por exemplo, a cadeia LLL não descreve nenhuma mT. Para que a cada natural i corresponda uma mT $M_i$, vamos associar a cada i tal que $x_i$ não descreve nenhuma máquina a máquina vazia ##, que não tem nenhum estado final, e nenhuma transição possível. Incidentalmente, ## aceita a linguagem vazia ∅.

Assim, podemos falar em $M_i$, para qualquer i: ou M é a máquina representada por $x_i$, ou $x_i$ não representa nenhuma máquina, e Mi é a máquina representada por ##. Note que é fácil distinguir as cadeias de Δ* que representam mT's das demais, uma vez que elas formam um conjunto regular.

Semelhantemente, podemos enumerar as cadeias que representam entradas:

$x_0$, $x_1$, $x_2$ ...,

de forma que poderemos falar na i-ésima cadeia de entrada $x_i$.

Exercício: Escolha uma enumeração de $\Delta^*$. Para essa enumeração, descreva algoritmos que permitam

dado um natural i, determinar a cadeia $x_i$ de $\Delta^*$ correspondente.
dada uma cadeia x em ?*, determinar o natural i tal que $x = x_i$.
dada uma cadeia $x_i$ em $\Delta^*$, determinar se $x_i$ é ou não a representação de uma mT.

Exercício: Para uma enumeração determinada de $\Delta^*$,

(a) determine o número da sequência ##.
(b) determine qual a máquina M23.

Suporemos daqui para a frente que está fixada uma enumeração de todas as máquinas de Turing, em algum alfabeto $\Delta$ fixo.

$M_0, M_1, M_2, M_3, \ldots$

e de suas entradas,

$x_0, x_1, x_2, x_3, \ldots$

*Codificação e decodificação.* Podemos sempre supor que existem algoritmos com as seguintes finalidades:

dado um natural i, determinar a cadeia xi.
dado um natural i, determinar a representação de Mi.
dada uma cadeia x, determinar i tal que x = xi.
dada uma mT M, determinar i tal que M = Mi.

Faremos referência a esses algoritmos como algoritmos de codificação ou de decodificação. Como se trata de algoritmos, podemos supor que podem ser simulados por uma mT, ou usados por mT's como subrotinas.

## *8.3 - Máquina de Turing Universal.*

Vamos agora mostrar como pode ser construída uma mT universal U, que tem a propriedade de poder simular qualquer outra mT. Por simplicidade, vamos descrever uma mT universal U não determinística. Para simular a computação de uma mT M qualquer, quando recebe como entrada a cadeia x, a idéia é a seguinte:

U recebe uma entrada $\alpha\$\beta$, onde $\alpha$ e $\beta$ são as representações de M e x em algum alfabeto $\Delta$;
U simula M com entrada x;
U aceita $\alpha\$\beta$ se M aceita x.

Para a simulação, U constrói uma representação de uma configuração de M em sua fita, e faz a simulação alterando essa configuração. A simulação parte da configuração inicial $q\alpha$ Para cada mudança de configuração $C_i \vdash C_{i+1}$, U deve identificar o estado e o símbolo lido em $C_i$, escolher uma entre as diversas transições previstas na função de transição $\delta$ que faz parte de $\beta$ e construir a nova configuração $C_{i+1}$.

Como U é não determinística, pode decidir "não-deterministicamente" quando uma configuração final $C_f$ de M foi atingida, e apenas verificar que o estado contido em $C_f$ é um dos estados finais listados em $\beta$.

Todas as operações envolvidas são operações de cópia e de comparação de cadeias, e não oferecem nenhum problema maior. Naturalmente, a simulação de um passo da máquina M corresponde a um número considerável de passos de U, mas isto não é um problema.

Assim, a construção de U é possível. Frequentemente, é conveniente supor que a mT U recebe como entrada i$j, para indicar que deve simular Mi com entrada xj, sendo i e j representados em alguma base adequada. Os passos adicionais necessários são de codificação e decodificação, e podem, como já vimos, ser realizados por uma mT.

A mT U é frequentemente usada como uma subrotina, na construção de outras mT's, em geral através de frases como

"então, M simula $M_i$ com entrada $x_j$ e se M parar, ...".

Uma comparação que se faz frequentemente é a da máquina de Turing universal U com o modelo de von Neumann, em que se baseiam os computadores comuns. O modelo de von Neumann se caracteriza justamente por ter o programa armazenado na memória, juntamente com os dados, durante a execução.

Note que U é uma máquina de Turing, e, portanto, faz parte da enumeração $M_0$, $M_1$, $M_2$, $M_3$, … sendo, para algum natural u, a máquina $M_u$.

### *8.4 - Uma linguagem não recursivamente enumerável.*

Vamos agora apresentar um exemplo de linguagem não recursivamente enumerável. Considere a linguagem L abaixo:

$L = \{ x_i \mid x_i \text{ não é aceita por } M_i \}$

Fato: L não é recursivamente enumerável.

Dem.: Vamos mostrar, por contradição, que L não é aceita por nenhuma máquina de Turing. Para isso, suponha que L é aceita pela mT M.

Como toda mT, M faz parte da enumeração das mT's, ou seja, para algum i, $M = M_i$, de forma que $L = L(M_i)$. Vamos verificar se $x_i \in L$.

- se tivermos $x_i \in L$, como $L = L(M_i)$, temos que $M_i$ aceita x, e portanto $x_i \notin L$.
- se, alternativamente, tivermos $x_i \notin L$, ou seja, $x_i \notin L(Mi)$, naturalmente, M não aceita $x_i$, e portanto $x_i \in L$.

Estabelecida a contradição, concluímos que L não é aceita por nenhuma mT M, e que L não é recursivamente enumerável.

Para a linguagem L, portanto, não existem

- um procedimento reconhecedor de L
- um procedimento enumerador de L
- uma gramática, tipo 0 ou não, que gere L.

Fato: , o complemento de L, é uma linguagem recursivamente enumerável.

Dem.: Temos  = { $x_i$ | $x_i$ é aceita por $M_i$ }. Podemos descrever uma mT M que aceita L, da seguinte forma:

a partir de xi, M obtém a representação de Mi.
como foi descrito para a máquina universal U, M simula Mi com entrada xi.

se Mi parar e aceitar xi, M também pára e aceita xi.

Note que se $M_i$ não pára, com entrada $x_i$, então M também não pára, com a mesma entrada

Fato: A linguagem $\overline{L}$ não é recursiva.

Dem.: Já vimos anteriormente que o complemento de uma linguagem recursiva também é uma linguagem recursiva. Como o complemento de $\overline{L}$ é L, que não é r.e., e portanto, também não é recursiva, $\overline{L}$ não pode ser recursiva.

A linguagem $\overline{L}$, portanto, tem uma gramática tipo 0, mas não sendo recursiva, também não pode ser sensível ao contexto, e não pode ter nenhuma gramática tipo 1.

### *8.5 - Problemas decidíveis e indecidíveis.*

Para qualquer conjunto X não recursivo, a pergunta "$x \in X$?" não admite solução através de um algoritmo que aceite x como entrada e responda SIM ou NÃO corretamente à pergunta. Mas, do ponto de vista prático, a diferença entre um conjunto não recursivamente enumerável, e um conjunto recursivamente enumerável que não é recursivo pode ser considerada pequena. Com efeito, suponha que um conjunto L é recursivamente enumerável mas não recursivo, e que dispomos de uma mT M que reconhece L, mas, não pára quando sua entrada não pertence a L. Suponha que, com entrada x, M foi executada por, digamos, mil passos, e que não parou. Nada podemos responder à pergunta "$x \in L$ ?". Será que M vai parar nos próximos mil passos?

Uma das maneiras de dizer que uma linguagem L não é recursiva é dizer que o problema "$x \in L$ ?" não é decidível. De uma forma geral, um problema indecidível é um conjunto de questões que pode ser reduzido por codificação ao problema da pertinência em uma linguagem não recursiva, e, assim, não pode ser respondido por uma mT que sempre pára; se um problema pode ser reduzido por codificação ao problema de pertinência em uma linguagem recursiva, dizemos que é decidível.

Um ponto importante é que um problema indecidível sempre envolve uma família de questões, ou questões em que aparece um parâmetro. O problema P(i) - "$M_i$ aceita $x_i$?" pode ser indecidível, mas o problema P(23) - "$M_{23}$ aceita $x_{23}$?" é decidível. Como prova disso, oferecemos duas mTs: $M_{sim}$ e $M_{não}$. $M_{sim}$ sempre pára e aceita qualquer entrada (sempre responde SIM); $M_{não}$ sempre pára, mas nunca aceita qualquer de suas entradas (sempre responde NÃO). É irrelevante o fato de ser difícil ou trabalhoso descobrir qual das duas máquinas corresponde à resposta correta da pergunta "$M_{23}$ aceita $x_{23}$?", mas certamente uma das duas resolve corretamente o problema. De fato, o que é indecidível em "$M_i$ aceita $x_i$ ?" é exatamente qual das duas respostas corresponde a um valor de i arbitrário, ou seja, qual das duas máquinas consideradas deve ser usada em cada caso.

*Redução*. Uma das técnicas mais comuns de estabelecer se um problema é decidível ou indecidível é através de redução desse problema a outro problema cuja resposta é conhecida.

> Dizemos que $P_1$ *se reduz a* $P_2$ se a resposta a qualquer consulta a $P_1$ pode ser deduzida da resposta a uma consulta apropriada a $P_2$.
>
> Se $P_1$ se reduz a $P_2$, podemos dizer que, de certa forma, $P_2$ é mais geral que $P_1$.

Supondo que $P_2$ é indecidível, e que $P_2$ se reduz a $P_1$, podemos concluir que $P_1$ também é indecidível; por outro lado, se $P_1$ é decidível, e $P_2$ se reduz a $P_1$, podemos concluir que $P_2$ também é decidível.

*O problema da parada*. Como exemplo da técnica de redução, vamos mostrar que o problema da parada das máquinas de Turing (*halting problem*) é indecidível. Este problema pode ser formulado como "$M_i$ pára com entrada $x_j$?", embora algumas vezes seja confundido o problema da aceitação "$M_i$ aceita $x_j$?".

Como é fácil reduzir qualquer um dos problemas ao outro, os dois problemas podem ser considerados equivalentes, e a confusão de certa forma é justificada. Para verificar a equivalência, basta verificar que é sempre possível alterar uma mT M, construindo outra mT M' que aceita a mesma linguagem, de maneira tal que M' tem as propriedades de parar sempre que atinge um estado final, e de nunca parar em um estado que não seja final. Isso quer dizer que M' pára se e somente se aceita sua entrada. Vamos representar o problema da parada por Q(i,j) = "$M_i$ aceita $x_j$?"

Como sabemos que P(i) = "Mi aceita xi?" é indecidível (veja a linguagem L acima), e sabemos que podemos reduzir P a Q, porque P(i) é equivalente a Q(i, i), concluímos que o problema da parada Q é indecidível.

Usualmente, a demonstração de que um problema é indecidível é, feita através de redução, a partir do problema da parada, diretamente, ou então, de forma indireta, a partir de problemas cuja indecidibilidade já foi provada anteriormente.

*(junho 99)*

# Capítulo 9: Linguagens sensíveis ao contexto e autômatos linearmente limitados.

*José Lucas Rangel*

## *9.1 - Introdução.*

Como já vimos anteriormente, a classe das linguagens sensíveis ao contexto (lsc) é uma classe intermediária entre a classe das linguagens recursivas e a classe das linguagens livres de contexto (llc). Já vimos anteriormente que a classe das llc está propriamente contida na classe das lsc's: como aplicação do Lema do Bombeamento, provamos que alguns exemplos de linguagens geradas por gramáticas sensíveis ao contexto não são llc's.

Neste capítulo, queremos apresentar os autômatos linearmente limitados, uma classe de máquinas de Turing cuja operação é restringida, de forma a aceitar apenas as lsc's. Adicionalmente, queremos mostrar que a inclusão da classe das lsc's na classe das linguagens recursivas é própria, através de um exemplo de linguagem recursiva que não é uma lsc.

## *9.2 - Autômatos linearmente limitados.*

Um autômato linearmente limitado (all) é uma máquina de Turing não determinística $A = <K, \Gamma, \Sigma, \delta, i, F>$, que satisfaz certas restrições:

- dois símbolos especiais, um marcador esquerdo [ e um marcador direito ], são incluídos em $\Gamma$;
- quando [ é lido, a máquina não pode se mover para a esquerda, nem escrever um símbolo distinto de [;
- quando ] é lido, a máquina não pode se mover para a direita, nem escrever um símbolo distinto de ].
- [ e ] não podem ser escritos em nenhuma outra situação.

Supondo que a fita de um all contém inicialmente uma sequência da forma [ x ], estas restrições fazem com que o all não possa abandonar a região entre os dois marcadores [ e ], nem apagá-los ou movê-los. Portanto, todas as computações de um all devem ser feitas no espaço que originalmente contém sua entrada x.

Como já vimos anteriormente, é possível definir uma fita com k trilhas, e, num trecho da fita de comprimento n, podemos armazenar kn símbolos. Esta é a origem do nome "linearmente limitado": o all pode manipular em sua fita informação com tamanho limitado por uma função linear do tamanho de sua entrada.

Definimos configurações e transições para um all da mesma forma que para uma mT. Entretanto, a configuração inicial associada à entrada $x \in \Sigma^*$ é i [ x ]. Definimos a linguagem de um all A, por

$$L(A) = \{ x \in \Sigma^* \mid i [ x ] \vdash\!\!\!\!-^* [ \alpha f \beta ], \text{ com } f \in F \}$$

Exemplo: Vamos descrever informalmente um all A que aceita a lsc $\{ x x \mid x \in \Sigma^* \}$, sendo $\Sigma = \{ a, b \}$.

Inicialmente, a fita de A contém [ y ]. Em seguida, A marca o primeiro símbolo de y, e o primeiro símbolo da segunda metade de y, que verifica ser igual ao

primeiro. (A posição inicial da segunda metade é escolhida de forma não determinística.) A seguir, o processo se repete com os demais símbolos das duas metades, marcando o primeiro símbolo não marcado da primeira metade e o primeiro símbolo não marcado (igual) da segunda metade, até que nenhum símbolo não marcado reste na fita.

Atingido este ponto, A verificou que o ponto escolhido como início da segunda metade era o correto, porque havia um número igual de símbolos nas duas metades, e além disso, que os símbolos correspondentes das duas metades eram iguais em todos os casos. Portanto, a cadeia de entrada y pertence à linguagem, e pode ser aceita por A, que passa para um estado final, onde pára.

Exercício: Construa um all que aceite a linguagem $\{x\ x \mid x \in \Sigma^*\}$, onde $\Sigma = \{a, b\}$, em todos os detalhes.

### *9.3 - All's e lsc's*

Vamos agora provar dois teoremas que mostram que a classe das linguagens aceitas por all's não determinísticos e a classe das linguagens sensíveis ao contexto (lsc's) são idênticas.

Teorema 9.1: Toda lsc é reconhecida por um all não determinístico.

Dem.: Seja L uma lsc, e G uma gsc que gera L. Vamos mostrar como construir um all A, que aceita L(G). Para verificar que $x \in L(G)$, A simula uma derivação $S \Rightarrow^* x$ em sua fita.

Primeiro, observamos que, se G tem uma regra $S \rightarrow \varepsilon$, A deve aceitar a entrada vazia, identificada pelo conteúdo de fita []. Este caso deve ser tratado especialmente, porque a derivação $S \Rightarrow \varepsilon$ não pode ser simulada sem um espaço de comprimento pelo menos $|S| = 1$.

Para considerar as demais entradas $x = x_1\ x_2\ \ldots\ x_n$, A inicialmente divide sua fita em duas trilhas, copiando x na primeira trilha, e preparando a segunda para simular uma derivação de x. Para representar as duas trilhas, vamos usar pares de símbolos, de forma que

$$(a_1, b_1)\ (a_2, b_2)\ \ldots\ (a_n, b_n)$$

representa uma fita com duas trilhas, que contêm, respectivamente, $a_1a_2\ldots a_n$ e $b_1b_2\ldots b_n$.

Inicialmente, a fita de A contém

$$[\ x_1\ x_2\ \ldots\ x_n\ ].$$

Dividindo a fita em duas trilhas, temos

$$[\ (x_1, S)\ (x_2, \Diamond)\ \ldots\ (x_n, \Diamond)\ ]$$

Na segunda trilha, que contém inicialmente $S\Diamond\ldots\Diamond$, A simula uma derivação em G, sem alterar o conteúdo da primeira trilha, escolhendo as regras de forma não determinística. Supondo-se, naturalmente, que $x \in L$, uma derivação de x pode ser simulada, e ao final da simulação, a fita conterá duas cópias de x, uma em cada trilha:

$$[\ (x_1, x_1)\ (x_2, x_2)\ \ldots\ (x_n, x_n)\ ]$$

Neste ponto, A verifica que o conteúdo da primeira trilha é idêntico ao da segunda, e passa para um estado final.

A idéia central da demonstração é a de que, como G é uma gsc, as formas sentenciais intermediárias entre S e x, na derivação $S \Rightarrow^* x$, tem todas comprimento menor ou igual ao de x, e podem portanto ser escritas na segunda trilha.

Nota: a demonstração acima usa o fato de que o all A é não-determinístico de forma essencial, e essa hipótese não pode ser retirada, pelo menos de forma simples. Na demonstração, apenas uma forma sentencial é construída a cada vez, sendo a escolha da regra e da maneira de sua aplicação feita de forma não-determinística. A verificação final de que a cadeia x foi obtida, verifica a correção da escolha. (Escolhas erradas não poderiam derivar x, e portanto não levariam à aceitação.)

No caso de um all determinístico, entretanto, isto não seria possível, e seria necessário, em princípio, examinar todas as formas sentenciais possíveis, até o comprimento n da entrada, como foi feito no algoritmo usado para demonstrar que todas as linguagens sensíveis ao contexto são conjuntos recursivos. Não é possível, entretanto, considerar todas as formas sentenciais simultaneamente por causa da restrição de espaço, porque o número de formas sentenciais de comprimento menor ou igual a n pode ser exponencial em n. Uma outra alternativa seria tratar uma forma sentencial por vez, e anotar apenas o caminho percorrido durante a derivação da forma sentencial corrente (que regras foram aplicadas em que pontos, por exemplo), de forma a recuar (*backtrack*) posteriormente, se a cadeia x não for atingida, para voltar e considerar as alternativas restantes. Isto também não é possível, pois o comprimento do caminho também pode ser exponencial em n.

Como esta demonstração não pode ser adaptada para o caso determinístico, pelo menos de forma simples, e nenhuma outra demonstração alternativa foi descoberta, o problema da equivalência das classes das linguagens aceitas por all's determinísticos e all's não determinísticos é um problema ainda em aberto.

Teorema 9.2: Toda linguagem aceita por um all é uma lsc.

Dem.: Seja A um all (não determinístico). Para aceitar uma entrada x, A realiza uma computação

$$i\ [\ x\ ] \vdash\!\!\!-^*\ \alpha\ f\ \beta$$

onde i é o estado inicial, e f um dos estado finais de A. Vamos construir uma gsc G que simula esta computação em cada derivação. Note, entretanto, que uma derivação de x em G deve partir do símbolo inicial S, e terminar com x, e que a computação acima só pode ser iniciada após a definição de x. Usando símbolos não terminais da forma (a, b, c, d), podemos simular quatro "trilhas":

- a primeira trilha guarda uma cópia de x, que não será alterada;
- a segunda guarda outra cópia, que será usada na simulação;
- a terceira guarda o estado corrente e a posição da cabeça, durante a simulação;
- a quarta tem as marcas que indicam as extremidades da fita.

A derivação se dá em várias fases:

$$\text{fase 1: } S \Rightarrow^* (x_1, x_1, i, [\,])\ (x_2, x_2, \Diamond, \Diamond)\ \ldots\ (x_n, x_n, \Diamond, ])$$

Nesta primeira fase, são geradas as duas cópias de $x=x_1x_2\ldots x_n$, e é preparada a configuração inicial, com o estado i na primeira posição.

fase 2: $(x_1, x_1, i, [])\ (x_2, x_2, \Diamond, \Diamond) \ldots (x_n, x_n, \Diamond, ])$
$\Rightarrow^* (x_1, z_1, \Diamond, [])\ (x_2, z_2, \Diamond, \Diamond) \ldots (x_m, z_m, f, \Diamond) \ldots (x_n, z_n, \Diamond, ])$

Na segunda fase, é feita a simulação de A até que uma configuração final $\alpha f \beta$ seja obtida, com $\alpha = z_1 z_2 \ldots z_{m-1}$ e $\beta = z_m z_{m+1} \ldots z_n$.

fase 3: $(x_1, z_1, \Diamond, [])\ (x_2, z_2, \Diamond, \Diamond) \ldots (x_m, z_m, f, \Diamond) \ldots (x_n, z_n, \Diamond, ]) \Rightarrow^* x_1 x_2 \ldots x_n$

Na última fase, a partir dos símbolos em que o estado (a terceira componente) é final, é feita a substituição, de forma a deixar apenas os símbolos da primeira "trilha".

Os detalhes da construção da gramática ficam como exercício para o leitor.

### *9.4 - Uma linguagem recursiva que não é sensível ao contexto.*

Vamos agora apresentar uma linguagem L, definida a seguir, que é recursiva, mas não é uma lsc. Para definir L, vamos inicialmente supor uma enumeração das gsc's, $G_0$, $G_1$, $G_2$, …, limitadas ao caso em que o alfabeto dos terminais contém apenas 0 e 1. Como feito anteriormente com máquinas de Turing, essa enumeração pode ser baseada na codificação das gramáticas em cadeias em um alfabeto adequado.

Apenas para fixar as idéias, podemos supor que os símbolos nãoterminais são codificados como n, n|, n||, …, sendo o nãoterminal inicial indicado apenas por n. Dessa forma, para codificar uma gramática $G = < N, \Sigma, P, S >$, não há necessidade de incluir na codificação de uma gramática o conjunto $\Sigma$ de seus terminais (0 e 1 podem ser representados simplesmente por 0 e 1), o conjunto de nãoterminais N (sempre representados por n, n|, n||, …, ou o símbolo inicial S (codificado como n). Basta assim apenas codificar as regras da gramática, que podem ser separadas por #.

Com isso, o alfabeto pode ser $\Delta = \{0, 1, n, |, \rightarrow, \#\}$. Como fizemos na enumeração das máquinas de Turing, a enumeração deve eliminar as cadeias de $\Delta^*$ que não correspondem a gsc's, e substituí-las por alguma gsc, por exemplo $\varepsilon$, que representa a gsc "vazia", que não tem nenhuma regra, e que gera a linguagem vazia $\varnothing$.

Para cada gramática $G_i$, é possível construir uma mT que sempre pára e que reconhece a linguagem de $G_i$. Seja $M_i$ a mT construída a partir de $G_i$. Portanto, a enumeração $M_0$, $M_1$, $M_2$, … inclui mT's que aceitam todas as lsc's.

Defina a linguagem L por

$L = \{ x_i \mid M_i \text{ não aceita } x_i \}$

Fato: L é recursiva.

Dem.: Note que, por construção, todas as máquinas $M_i$ aceitam conjuntos recursivos, e param para todas suas entradas. L é aceita por uma mT M que

- a partir de sua entrada x, M determina i tal que $x=x_i$, e constrói uma representação de $M_i$.
- simula $M_i$ com entrada $x_i$.
- M aceita $x=x_i$ se e somente se $M_i$ não aceita $x_i$, ou seja, se $M_i$, com entrada $x_i$, pára em um estado que não é final.

Fato: L não é uma lsc.

Dem.: Por contradição. Suponha que L é uma lsc. Neste caso, L tem uma gsc $G=G_i$, e é aceita pela máquina $M_i$: $L = L(G_i) = L(M_i)$. Mas então

$x_i \in L \Leftrightarrow M_i$ não aceita $x_i \Leftrightarrow x_i \notin L(M_i) \Leftrightarrow x_i \notin L$,

estabelecendo uma contradição. Concluímos, portanto, que L não é uma lsc.

As duas propriedades acima estabelecem que a classe das linguagens sensíveis ao contexto está propriamente contida na classe das linguagens recursivas.

*(julho 99)*

# Capítulo 10: Linguagens determinísticas e seus aceitadores

José Lucas Rangel

## *10.1 - Introdução.*

Nos capítulos relativos às linguagens livres de contexto, observamos que as classes de linguagens aceitas por autômatos de pilha determinísticos (apd) e não determinísticos (apnd) não são as mesmas, ao contrário do que acontece, por exemplo, com os autômatos finitos. Neste capítulo, vamos apresentar exemplos e algumas propriedades das linguagens determinísticas, isto é, das linguagens aceitas por apd's.

## *10.2 - Autômatos de pilha determinísticos.*

Como observado anteriormente, um apd é um caso particular de apnd, em que, a partir de qualquer configuração existe, no máximo, uma configuração acessível. Em termos da definição de um apd $A = <K, \Sigma, \Gamma, \delta, i, I, F>$, isto significa:

- para quaisquer $q \in K$, $a \in \Sigma \cup \{\varepsilon\}$, $Z \in \Gamma$, $\delta(q, a, Z)$ tem, no máximo, um elemento;
- se, para algum $q \in K$ e algum $Z \in \Gamma$, $\delta(q, \varepsilon, Z)$ não é vazio, então para todos os símbolos $a \in \Sigma$, $\delta(q, a, Z)$ deve ser vazio.

A primeira condição impede a possibilidade de escolha entre duas transições com um símbolo; a segunda condição evita a possibilidade de escolha entre a leitura de um símbolo de $\Sigma$, e a leitura de um $\varepsilon$, ou seja, a escolha entre ler e não ler um símbolo da entrada.

Note, entretanto, que, ao contrário do que acontece com os autômatos finitos determinísticos, pode não existir nenhuma configuração alcançável a partir de uma dada configuração, mesmo que a entrada ainda não tenha sido completamente lida. Por exemplo, a pilha pode ficar vazia. Assim, uma entrada x pode ser deixar de ser aceita por um apd sem ter sido completamente lida.

Definimos uma *linguagem determinística* como sendo uma linguagem aceita *por estado final* por um apd. A razão para essa escolha está nas propriedades a seguir, que limitam o interesse das linguagens aceitas por apd's por pilha vazia.

Dizemos que uma cadeia y é um *prefixo* de outra cadeia x se x = y z para algum z; dizemos que y é um *prefixo próprio* de x se y é um prefixo de x, e $y \neq x$.

Dizemos que uma linguagem L tem a *propriedade dos prefixos* se nenhuma cadeia de L tem um prefixo próprio que também pertence a L, ou seja, se dadas duas cadeias x, y pertencentes a L,

$x = yz$ implica $z = \varepsilon$.

Fato: Se $L = L_{pv}(M)$, para algum apd M, então L tem a propriedade dos prefixos.

Dem.: Suponha que $x \in L$. Isto significa que a partir da configuração inicial [i, x, I], correspondente a x, pode ser alcançada uma configuração final (para aceitação por pilha vazia) [q, ε, ε ]. Considere uma cadeia y=xz, com $z \neq \varepsilon$. A partir da configuração inicial correspondente a y, [i, y, I] = [i, xz, I] pode ser alcançada a configuração [q, z, ε], sem que a parte z da entrada seja lida. Como a partir desta última

configuração nenhuma outra pode ser atingida, nenhum símbolo de z será lido, e a cadeia y não será aceita.

Exercício: Seja L uma linguagem qualquer num alfabeto Σ, e seja $ um símbolo novo, não pertencente a Σ.

(a) Mostre que a linguagem L•{$} = { x$ | x(L} tem a propriedade dos prefixos;
(b) Mostre também que se L é uma linguagem determinística, ou seja, se L é aceita por estado final por um apd, L•{$} é aceita por um apd por pilha vazia.

Exercício: Mostre que as linguagens { $a^i b^j$ | i=j }, { $a^i b^j$ | i>j }, { $a^i b^j$ | i<j } e { $a^i b^j$ | i≠j }, são determinísticas.

### *10.3 Propriedades de fechamento da classe das linguagens determinísticas*

Para as linguagens determinísticas, não existe um correspondente do Lema do Bombeamento (*pumping lemma*), de forma que a principal maneira de provar que uma linguagem livre de contexto não é determinística é através do uso de propriedades de fechamento. Por exemplo, veremos posteriormente que a classe das linguagens determinísticas é fechada para a operação de complemento. Assim, uma llc cujo complemento não é uma llc não pode ser determinística.

Vamos começar pela propriedade do fechamento com o complemento.

Teorema: A classe das linguagens determinísticas é fechada para o complemento.

Dem.: Seja M um apd que aceita uma linguagem L, por estado final.

Vamos mostrar como construir um apd M' que aceita o complemento de L. A idéia central é semelhante à da prova do fechamento da classe das linguagens regulares para o complemento: vamos trocar os estados finais pelos não finais. Assim, pelo menos em princípio, se L é a linguagem aceita por M = < K, Σ, Γ, δ, i, I, F>, o apd M' = < K, Σ, Γ, δ, i, I, K-F> deve aceitar o complemento de L.

Entretanto, algumas possibilidades devem ser consideradas:

1. M não aceita uma entrada x porque M pára sem que a entrada x tenha sido toda lida. Se construirmos M' apenas pela troca dos estados finais e não finais de M, x continuaria não sendo aceita, o que seria incorreto.
2. M lê toda a entrada x, mas após isso, M executa uma infinidade de passos com entrada ε, sem retirar símbolos da pilha, e, nessa fase, passa por estados finais e não finais. Note que M aceita x, mas se construirmos M' apenas pela troca dos estados finais e não finais, x continuaria sendo aceita, o que seria incorreto.

Para resolver esses problemas, é necessário fazer várias transformações em M antes de trocar os estados finais e não finais. Os detalhes da efetiva construção de M' a partir de M podem ser vistos no livro de Hopcroft e Ullman.

A partir deste resultado, podemos mostrar que a classe das linguagens determinísticas é contida propriamente na classe das llc.

Fato: A llc L = { $a^i b^j c^k$ | ou i≠j ou j≠k } não é determinística.

Dem. Basta mostrar que o complemento não é uma livre de contexto.

Intuitivamente, L "não pode" ser determinística porque comparar i e j envolve um tratamento da pilha (empilhar i símbolos, desempilhar j símbolos) completamente diferente do necessário para comparar j e k (empilhar j símbolos, desempilhar k símbolos).

Fato: A classe das linguagens determinísticas não é fechada para as operações de união e interseção.

Dem.: Para a união, note que a linguagem L do exemplo anterior pode ser vista como a união de duas llc determinísticas:

$L = \{ a^i b^j c^k \mid \text{ou } i \neq j \text{ ou } j \neq k \} = L_1 \cup L_2$,

sendo

$L_1 = \{ a^i b^j c^k \mid i \neq j \}$
$L_2 = \{ a^i b^j c^k \mid j \neq k \}$

Para mostrar que a classe das linguagens determinísticas não é fechada para a interseção, podemos usar os fatos de que a classe não é fechada para a união, mas é fechada para o complemento. Uma das relações de de Morgan é

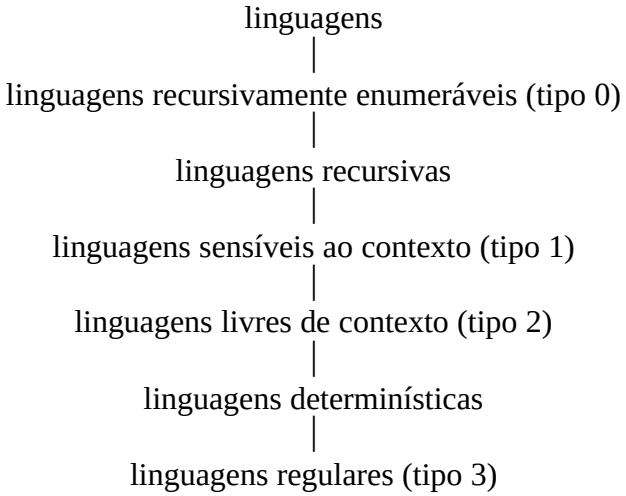
$$L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$$

Assim, se a classe das linguagens determinísticas fosse fechada para a interseção, também seria fechada para a união. Pela contradição, vemos que a classe não pode ser fechada para interseção.

Exercicio: Mostrar que $L = \{ a^i b^j \mid i=j \text{ ou } i=2j \}$ é uma linguagem não determinística, que pode ser escrita como a união de duas linguagens determinísticas.

## *10.4 Conclusão*

Com os exemplos e as propriedades estabelecidas neste capítulo e nos anteriores, a hierarquia das classes de linguagens definidas aqui pode ser representada pela figura abaixo, em que todas as inclusões indicadas são próprias.




*(julho 99)*

### *Exemplo 1:*

mT M: aceita L={ 0w0 | w∈{0,1}* }

```
K = { A, B, C, D }
entrada: 0, 1
fita: ◊, 0, 1, x
inicial: A
final: D
transições:
     A0BxR      B0B0R      B1B1R      B◊CxL      C0DxR
aceitação de 01100
     A01100     xB1100     x1B100     x11B00     x110B0
     x1100B     x110C0x    x110xDx
```

### *Exemplo 2:*

(ver Ex.1) Gramática que gera L, simulando M

```
terminais: 0, 1
nãoterminais: S, T, X, Y, Z, W, $, A, B, C, D
inicial: S
regras:
```

fase 1: (prepara para simular M, salvando uma cópia de x)

$S \Rightarrow^* \$Ax\$x$

```
S → $AT
T → 0XT | 1YT | Z
X0 → 0X   Y0 → 0Y
X1 → 1X   Y1 → 1Y
XZ → Z0   YZ → Z1
Z → $
```

fase 2: (simula M)

$\$Ax\$x \Rightarrow^* \$\alpha D\beta\$x$

(D é o único estado final)

```
A0 → xB
B0 → 0B   B1 → 1B
0B$ → C0x$      1B$ → C1x$      xB$ → Cxx$
     (isto é, aB$ → Cax$, para a∈Γ-{◊})
C0 → 0D
```

fase 3: (limpa)

```
D → W
0W → W    1W   W    xW   W
W0 → W    W1   W    Wx   W
$W$ → ε
```

Para gerar 01100, temos a derivação:
(fase 1)

S ⇒ $AT ⇒ $A0XT ⇒ $A0X1YT ⇒ $A0X1Y1YT ⇒ $A0X1Y1Y0XT
⇒ $A0X1Y1Y0X0XT ⇒ $A0X1Y1Y0X0XT ⇒ $A0X1Y1Y0X0XZ
⇒ $A0X1Y1Y0X0XZ ⇒ $A01XY1Y0X0XZ ⇒ $A01X1YY0X0XZ
⇒ $A011XYY0X0XZ ⇒ $A011XY0YX0XZ ⇒ $A011X0YYX0XZ
⇒ $A0110XYYX0XZ ⇒ $A0110XYY0XXZ ⇒ $A0110XY0YXXZ
⇒ $A0110X0YYXXZ ⇒ $A01100XYYXXZ ⇒ $A01100XYYXZ0
⇒ $A01100XYYZ00 ⇒ $A01100XYZ100 ⇒ $A01100XZ1100
⇒ $A01100Z01100 ⇒ $A01100$01100

(fase 2)

$A01100$01100 ⇒ $xB1100$01100 ⇒ $x1B100$01100
⇒ $x11B00$01100 ⇒ $x110B0$01100 ⇒ $x1100B$01100
⇒ $x110C0x$01100 ⇒ $x110xDx$01100

(fase 2)

$x110xDx$01100⇒ $x110xWx$01100 ⇒ $x110Wx$01100
⇒ $x11Wx$01100 ⇒ $x1Wx$01100 ⇒ $xWx$01100
⇒ $Wx$01100 ⇒ $W$01100 ⇒ 01100

### *Exemplo 3:*

(ver Exemplo 1)
all M: aceita L={ 0w0 | w∈{0,1}* }

```
K = { I, A, B, C, D }
entrada: 0, 1
fita: ◊, 0, 1, x, [, ]
inicial: A
final: D
transições:
     I[A[R
     A0BxR      B0B0R      B1B1R      B]C]L      C0DxR
aceitação de 01100
     I[01100]   [A01100]   [xB1100]   [x1B100]   [x11B00]
     [x110B0]   [x1100B]   [x110C0x]  [x110xDx]
```

### *Exemplo 4:*

(ver Ex.3) Gramática que gera L, simulando M

```
terminais: 0, 1
nãoterminais:
```

Temos:

a∈{0, 1, x}
b∈{0, 1}
q∈{I, A, B, C, D}

os não terminais são:

S
(ab) (duas trilhas, com a e b)
([ab) (no início da fita)
(ab]) (no fim da fita)

| | |
|---|---|
| (qab) | (q lê a) |
| (q[ab | (no início da fita, q lê [) |
| ([qab | (no início da fita, q lê a) |
| (qab]) | (no fim da fita, q lê a) |
| (abq]) | (no fim da fita, q lê ]) |
| (q[ab]) | (caso de um símbolo, q lê [) |
| ([qab]) | (caso de um símbolo, q lê a) |
| ([abq]) | (caso de um símbolo, q lê ]) |

inicial: S
regras:
(pode esquecer caso particular ε: $\varepsilon \notin L$)

fase 1: (prepara para simular M, salvando uma cópia de x)

S → (I[00]) | (I[11]) | (I[00) T | (I[11) T
T → (00) T | (11) T | (00]) | (11])

fase 2: (simula M)
Temos:

a, c ∈ {0, 1, x}
b, d ∈ {0, 1}

| passo de M | regras correspondentes |
|---|---|
| I[A[R | (I[ab]) → ([Aab])<br>(I[ab → ([Aab) |
| A0BxR | (A0b)(cd) → (xb)(Bcd)<br>([A0b)(cd) → ([xb)(Bcd)<br>(A0b)(cd]) → (xb)(Bcd])<br>([A0b)(cd]) → ([xb)(Bcd])<br>([A0b]) → ([xbB] |
| BaBaR | (Bab)(cd) → (ab)(Bcd)<br>([Bab)(cd) → ([ab)(Bcd)<br>(Bab)(cd]) → (ab)(Bcd])<br>([Bab)(cd]) → ([ab)(Bcd])<br>(Bab]) → (abB])<br>([Bab]) → ([abB]) |
| B]C]L | (abB]) → (Cab])<br>([abB]) → ([Cab]) |
| C0DxR | (C0b)(cd) → (xb)(Dcd)<br>([C0b)(cd) → ([xb)(Dcd)<br>(C0b)(cd]) → (xb)(Dcd])<br>([C0b]) → ([xbD]) |

fase 3: (limpa)

(Dab) → b
([Dab) → b  (D[ab) → b

(Dab]) → b  (abD]) → b
(D[ab]) → b  ([Dab]) → b  ([abD]) → b

b(cd) → bc  (cd)b → cb
([cd)b → cb  b(cd]) → bc

Para gerar 01100, temos a derivação:
(fase 1)

S ⇒ (I[00)T ⇒ (I[00)(11)T ⇒ (I[00)(11)(11)T
⇒ (I[00)(11)(11)(00)T ⇒ (I[00)(11)(11)(00)(00])
⇒ (I[00)(11)(11)(00)(00])

(fase 2)

(I[00)(11)(11)(00)(00]) ⇒ ([A00)(11)(11)(00)(00])
⇒ ([x0)(B11)(11)(00)(00]) ⇒ ([x0)(11)(B11)(00)(00])
⇒ ([x0)(11)(11)(B00)(00]) ⇒ ([x0)(11)(11)(00)(B00])
⇒ ([x0)(11)(11)(00)(00B]) ⇒ ([x0)(11)(11)(00)(B00])
⇒ ([x0)(11)(11)(00)(C00]) ⇒ ([x0)(11)(11)(00)(x0D])

(fase 3)

([x0)(11)(11)(00)(x0D]) ⇒ ([x0)(11)(11)(00)0 ⇒
([x0)(11)(11)00 ⇒ ([x0)(11)100 ⇒ ([x0)1100 ⇒ 01100