

Tema 3

Análise Lexical

Gramáticas regulares, autómatos finitos e expressões regulares

Linguagens Formais e Autómatos, 2º semestre 2017-2018

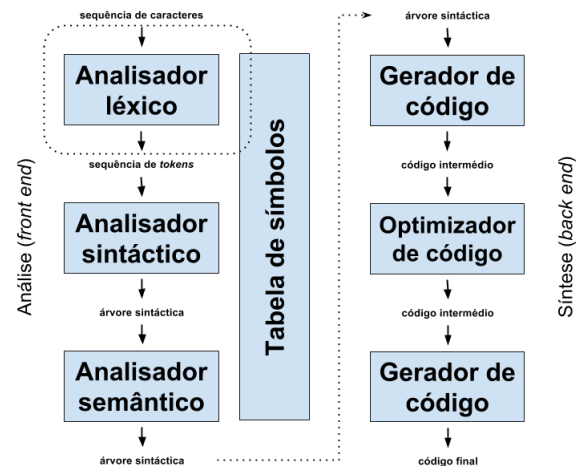
Miguel Oliveira e Silva, DETI, Universidade de Aveiro

Conteúdo

| | | |
|-----------|--|-----------|
| 1 | Análise Lexical: Estrutura de um Compilador | 2 |
| 2 | Linguagens regulares | 2 |
| 3 | Gramáticas regulares | 3 |
| 3.1 | Operações sobre gramáticas regulares | 4 |
| 4 | Expressões regulares | 6 |
| 4.1 | Gramática para expressões regulares | 9 |
| 5 | Conversão entre ER e GR | 10 |
| 5.1 | Conversão de ER para GR | 10 |
| 5.2 | Conversão de GR para ER | 10 |
| 6 | Reconhecimento de <i>tokens</i> | 11 |
| 6.1 | Diagramas de transição | 12 |
| 7 | Autómatos finitos | 15 |
| 8 | Autómato finito não determinista | 15 |
| 8.1 | Tabelas de transição | 18 |
| 9 | Autómato finito determinista | 18 |
| 10 | Autómato finito determinista | 20 |
| 10.1 | Projecto de autómato finito determinista | 20 |
| 10.2 | Redução de autómato finito determinista | 21 |
| 11 | Conversão de AFND em AFD | 26 |
| 12 | Conversão de uma expressão regular num AFND | 28 |
| 13 | Autómato finito generalizado (AFG) | 29 |
| 13.1 | AFG reduzido | 30 |
| 13.2 | Conversão de uma AFG numa ER | 30 |

1 Análise Lexical: Estrutura de um Compilador

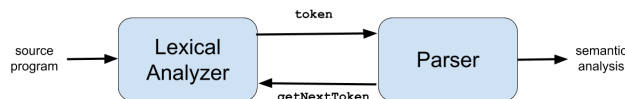
- O processo de compilação envolve diferentes fases:



- A primeira delas é a *análise léxica*, que consiste na conversão da sequência de caracteres de entrada numa sequência de elementos lexicais (*tokens*).



- A principal função da análise léxica é estruturar a sequência de caracteres da entrada numa sequência de *tokens* a serem processados pelo *parser*.



- No entanto, o analisador léxico efectua outras operações como sejam: a exclusão de espaços em branco e de comentários do *parser*, e a correlação entre erros (léxicos e sintácticos) com o código fonte (e.g. número da linha).



- A análise léxica pode ser feita recorrendo a gramáticas do tipo-3, ou seja, por *gramáticas regulares*, e a sua implementação computacional pode ser feita eficientemente recorrendo a *autómatos finitos*.

2 Linguagens regulares

- As gramáticas regulares geram *linguagens regulares*.
- A classe das linguagens regulares sobre um qualquer alfabeto A define-se indutivamente da seguinte forma:

1. O conjunto vazio, \emptyset , é uma linguagem regular (LR).
2. Qualquer que seja o símbolo $a \in A$, o conjunto $\{a\}$ é uma LR.

3. Se L_1 e L_2 são linguagens regulares, então $L_1 \cup L_2$ (união) é uma LR.
 4. Se L_1 e L_2 são linguagens regulares, então $L_1 \cdot L_2$ (concatenação) é uma LR.
 5. Se L_1 é uma linguagem regular, então $(L_1)^*$ (potenciação) é uma LR.
 6. Nada mais é linguagem regular.
- Note que o conjunto $\{\varepsilon\}$, isto é, o conjunto composto pela palavra vazia, é também uma linguagem regular uma vez que: $\{\varepsilon\} = \emptyset^*$
 - Uma vez que operações sobre LR geram uma LR, diz-se que a LR é *fechada* sobre as suas operações.

Linguagens regulares: exemplo 1

- Esta definição tem implicações interessantes.
- Uma delas é que qualquer linguagem finita, isto é que descreva um número finito de sequências de símbolos do seu alfabeto, é uma linguagem regular.
- Porquê?
 1. Seja $A = \{a_1, a_2, \dots, a_n\}$ o alfabeto da linguagem L
 2. Então as linguagens $L_1 = \{a_1\}, L_2 = \{a_2\}, \dots, L_n = \{a_n\}$ são LR (regra 2)
 3. Igualmente a linguagem $L_{\text{any}} = L_1 \cup L_2 \cup \dots \cup L_n$ é também uma LR (regra 3)
 4. Qualquer que seja uma sequência finita de n símbolos do alfabeto A , podemos sempre descrevê-la como:

$$\text{seq}_n = \text{prefix}_{n-1}(\text{seq}_n) \cdot L_{\text{any}}$$
 5. Logo, a sequência será uma LR sse a subsequência $\text{prefix}_{n-1}(\text{seq}_n)$ também o for (regra 4).
 6. Aplicando indutivamente a demonstração, facilmente se chega à conclusão que se há-de de chegar à subsequência vazia, logo qualquer linguagem finita é uma linguagem regular.

Linguagens regulares: exemplo 2

- Mostre que o conjunto dos números binários começados em 1 e terminados em 0 é uma LR sobre o alfabeto $A = \{0, 1\}$
 - O conjunto pretendido pode ser representado por $L = \{1\} \cdot A^* \cdot \{0\}$
1. $\{1\}$ e $\{0\}$ são regulares (regra 2)
 2. $A = \{0, 1\} = \{0\} \cup \{1\}$ é regular (regra 3)
 3. Se A é regular então A^* também é (regra 5)
 4. Finalmente, $\{1\} \cdot A^* \cdot \{0\}$ é também regular (regra 4)

3 Gramáticas regulares

Definição de gramática

- Qualquer que seja a linguagem que se queira reconhecer, podemos sempre defini-las por intermédio de *gramáticas*.
- Uma *gramática* é um quádruplo $G = (T, N, S, P)$, onde:
 1. T é um conjunto finito não vazio designado por alfabeto terminal, onde cada elemento é designado por símbolo *terminal*;
 2. N é um conjunto finito não vazio, disjunto de T ($N \cap T = \emptyset$), cujos elementos são designados por símbolos *não terminais*;
 3. $S \in N$ é um símbolo não terminal específico designado por *símbolo inicial*;
 4. P é um conjunto finito de *regras* (ou produções) da forma $\alpha \rightarrow \beta$ onde $\alpha \in (T \cup N)^* N (T \cup N)^*$ e $\beta \in (T \cup N)^*$, isto é, α é uma cadeia de símbolos terminais e não terminais contendo, pelo menos, um símbolo não terminal; e β é uma cadeia de símbolos terminais e não terminais.

Definição de gramática regular

- Uma gramática diz-se *regular* (à direita) se para qualquer produção ($\alpha \rightarrow \beta \in P$) as duas condições seguintes são satisfeitas:

$$\alpha \in N$$

$$\beta \in T^* \cup T^*N$$

- Alternativamente, podemos ter os não terminais (sempre) à esquerda:

$$\alpha \in N$$

$$\beta \in T^* \cup NT^*$$

- Isto é, uma gramática regular à direita contém apenas regras (P), onde ($B, C \in N$) e ($a \in T$):

$$B \rightarrow a$$

$$B \rightarrow aC$$

$$B \rightarrow \varepsilon$$

...

- E no caso de uma gramática regular à esquerda:

$$B \rightarrow a$$

$$B \rightarrow Ca$$

$$B \rightarrow \varepsilon$$

...

- Uma gramática regular gera uma linguagem regular.
- As gramáticas regulares são também *fechadas* nas suas operações.
- Isto é, aplicar uma qualquer das operações definidas sobre gramáticas regulares resulta também numa gramática regular.

3.1 Operações sobre gramáticas regulares

Operações sobre gramáticas regulares: reunião

- Sejam $G_1 = (T_1, N_1, S_1, P_1)$ e $G_2 = (T_2, N_2, S_2, P_2)$ duas gramáticas regulares quaisquer com $N_1 \cap N_2 = \emptyset$
- A gramática $G = (T, N, S, P)$ onde:

$$T = T_1 \cup T_2$$

$$N = N_1 \cup N_2 \cup \{S\} \quad \text{com} \quad S \notin (N_1 \cup N_2)$$

$$S = S$$

$$P = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2$$

é regular e gera a linguagem $L = L(G_1) \cup L(G_2)$

- A nova produção $S \rightarrow S_i$, com $i = 1, 2$, permite que G gere a linguagem $L(G_i)$

Operações sobre gramáticas regulares: reunião (exemplo)

- Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cup L_2$$

sabendo que:

$$L_1 = \{aw : w \in T^*\}$$

$$L_2 = \{wa : w \in T^*\}$$

- Vamos primeiro obter as GR que representam L_1 e L_2 :

$$\begin{aligned} S_1 &\rightarrow aX_1 & S_2 &\rightarrow aS_2 \mid bS_2 \mid cS_2 \mid a \\ X_1 &\rightarrow aX_1 \mid bX_1 \mid cX_1 \mid \varepsilon \end{aligned}$$

- Teremos assim como resultado a gramática:

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow aX_1 \\ X_1 &\rightarrow aX_1 \mid bX_1 \mid cX_1 \mid \varepsilon \\ S_2 &\rightarrow aS_2 \mid bS_2 \mid cS_2 \mid a \end{aligned}$$

Operações sobre gramáticas regulares: concatenação

- Sejam $G_1 = (T_1, N_1, S_1, P_1)$ e $G_2 = (T_2, N_2, S_2, P_2)$ duas gramáticas regulares quaisquer com $N_1 \cap N_2 = \emptyset$
- A gramática $G = (T, N, S, P)$ onde:

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \\ S &= S_1 \\ P &= \{A \rightarrow wS_2 : (A \rightarrow w) \in P_1 \wedge w \in T_1^*\} \cup \\ &\quad \{A \rightarrow w : (A \rightarrow w) \in P_1 \wedge w \in T_1^*N_1\} \cup \\ &\quad P_2 \end{aligned}$$

é regular e gera a linguagem $L = L(G_1) \cdot L(G_2)$

- As produções da segunda gramática mantêm-se inalteradas.
- As produções da primeira gramática que terminam num não terminal, também se mantêm inalteradas.
- As produções da primeira gramática que só têm terminais ganham o símbolo inicial da segunda gramática no fim.

Operações sobre gramáticas regulares: concatenação (exemplo)

- Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cdot L_2$$

sabendo que:

$$L_1 = \{aw : w \in T^*\}$$

$$L_2 = \{wa : w \in T^*\}$$

- Recuperando as GR que representam L_1 e L_2 :

$$\begin{aligned} S_1 &\rightarrow aX_1 & S_2 &\rightarrow aS_2 \mid bS_2 \mid cS_2 \mid a \\ X_1 &\rightarrow aX_1 \mid bX_1 \mid cX_1 \mid \varepsilon \end{aligned}$$

- Teremos assim como resultado a gramática:

$$\begin{aligned} S_1 &\rightarrow aX_1 \\ X_1 &\rightarrow aX_1 \mid bX_1 \mid cX_1 \mid S_2 \\ S_2 &\rightarrow aS_2 \mid bS_2 \mid cS_2 \mid a \end{aligned}$$

Operações sobre gramáticas regulares: fecho de Kleene

- Seja $G_1 = (T_1, N_1, S_1, P_1)$ uma gramática regular qualquer.
- A gramática $G = (T, N, S, P)$ onde:

$$T = T_1$$

$$N = N_1 \cup \{S\} \text{ com } S \notin N_1$$

$$S = S_1 \mid \varepsilon$$

$$P = \{S \rightarrow S_1 \mid \varepsilon\} \cup \\ \{A \rightarrow wS : (A \rightarrow w) \in P_1 \wedge w \in T_1^*\} \cup \\ \{A \rightarrow w : (A \rightarrow w) \in P_1 \wedge w \in T_1^*N_1\}$$

é regular e gera a linguagem $L = (L(G_1))^*$

- As produções que terminam num não terminal mantêm-se inalteradas
- As produções só têm terminais ganham o símbolo inicial no fim
- As novas produções ($S \rightarrow S_1 \mid \varepsilon$) garantem que $(L(G_1))^n \subseteq L(G)$, para qualquer $n \geq 0$

Operações sobre gramáticas regulares: fecho de Kleene (exemplo)

- Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1^*$$

sabendo que:

$$L_1 = \{aw : w \in T^*\}$$

- Recuperando a GR que representa L_1 :

$$S_1 \rightarrow aX_1$$

$$X_1 \rightarrow aX_1 \mid bX_1 \mid cX_1 \mid \varepsilon$$

- Teremos assim como resultado a gramática:

$$S \rightarrow S_1 \mid \varepsilon$$

$$S_1 \rightarrow aX_1$$

$$X_1 \rightarrow aX_1 \mid bX_1 \mid cX_1 \mid S$$

4 Expressões regulares

- As expressões regulares foram introduzidas em 1956 por Stephen Kleene.
- O conjunto das expressões regulares sobre um alfabeto A define-se indutivamente da seguinte forma:
 1. $()$ é uma expressão regular (ER) que representa a LR $\{\}$ (\emptyset).
 2. Qualquer que seja o $a \in A$, a é uma ER que representa a LR $\{a\}$.
 3. Se e_1 e e_2 são ER representando respectivamente as LR L_1 e L_2 , então $(e_1 \mid e_2)$ é uma ER representando a LR $L_1 \cup L_2$.
 4. Se e_1 e e_2 são ER representando respectivamente as LR L_1 e L_2 , então $(e_1 e_2)$ é uma ER representando a LR $L_1 \cdot L_2$.
 5. Se e_1 é uma ER representando a LR L_1 , então e_1^* é uma ER representando a LR $(L_1)^*$.
 6. Nada mais é expressão regular.

- É habitual representar-se por $\{\varepsilon\}$ a ER $()^*$. Representa a linguagem $\{\varepsilon\}$.
- A evidente semelhança entre LR e ER não é casual. Ambas expressam *gramáticas regulares*.
- Uma expressão regular, tal como uma gramática regular, gera uma linguagem regular.
 - Logo, é possível converter uma gramática regular numa expressão regular que represente a mesma linguagem e *vice-versa*.
- Tal como as gramáticas regulares, as expressões regulares são *fechadas* nas suas operações.

Expressões regulares: exemplos

P: Determine uma ER que represente o conjunto de números binários começados por 1 e terminados por 0.

R: $1(0|1)^*0$

P: Determine uma ER que representa as sequências definidas sobre o alfabeto $A = \{a, b, c\}$ que satisfazem o requisito de qualquer símbolo b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.

R: $(a|abc|c)^*$

P: Determine uma ER que represente as sequências binárias com um numero par de zeros.

R: $1^*(01^*01^*)^*$

Propriedades das expressões regulares

- Operação de escolha ($|$):
 - comutativa: $e_1 | e_2 = e_2 | e_1$
 - associativa: $e_1 | (e_2 | e_3) = (e_1 | e_2) | e_3 = e_1 | e_2 | e_3$
 - existência de elemento neutro: $e_1 | () = () | e_1 = e_1$
 - idempotência: $e_1 | e_1 = e_1$
- Operação de concatenação (implícita ou \cdot):
 - associativa: $e_1(e_2e_3) = (e_1e_2)e_3 = e_1e_2e_3$
 - existência de elemento neutro: $e_1\varepsilon = \varepsilon e_1 = e_1$
 - existência de elemento absorvente: $e_1() = ()e_1 = ()$ (a concatenação com o conjunto vazio resulta no próprio)
 - não goza da propriedade comutativa
- Operações de escolha e de concatenação:
 - concatenação distributiva relativamente à escolha:

$$e_1(e_2 | e_3) = (e_1e_2) | (e_1e_3) = e_1e_2 | e_1e_3$$

$$(e_1 | e_2)e_3 = (e_1e_3) | (e_2e_3) = e_1e_3 | e_2e_3$$
- Operação de fecho: $r^* = \varepsilon | r | rr | \dots$

$$(e^*)^* = e^*$$

$$(e_1^* | e_2^*)^* = (e_1 | e_2)^*$$

$$(e_1 | e_2)^* \neq e_1^* | e_2^*$$

$$(e_1e_2)^* \neq e_1^*e_2^*$$

Simplificação notacional

- Para simplificar a escrita das expressões regulares (de forma análoga às expressões aritméticas) existe uma precedência bem definida na aplicação dos diferentes operadores.
- A ordem decrescente de precedência é a seguinte:
 1. Parêntesis
 2. Fecho de Kleene (*)
 3. Concatenação (implícita ou ·)
 4. Escolha (|)
- A utilização destas precedências permite simplificar as ER
$$e = (((1^*)0)(1^*)0)(1^*) \Leftrightarrow e = 1^*01^*01^*$$
$$e_1 | e_2 \cdot e_3^* = e_1 | (e_2 \cdot (e_3^*))$$

Expressões regulares: exemplos

- Recuperando os exemplos anteriores.

P: Determine uma ER que represente o conjunto de números binários começados por 1 e terminados por 0.

R: $1(0|1)^*0 \Leftrightarrow (1((0|1)^*))0$

P: Determine uma ER que representa as sequências definidas sobre o alfabeto $A = \{a, b, c\}$ que satisfazem o requisito de qualquer símbolo b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.

R: $(a|abc|c)^* \Leftrightarrow ((a|((ab)c))|c)^*$

P: Determine uma ER que represente as sequências binárias com um número par de zeros.

R: $1^*(01^*01^*)^* \Leftrightarrow (1^*)(((((0(1^*))0)(1^*)))^*)$

Expressões regulares: mais exemplos

P: Sobre o alfabeto $A = \{0, 1\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(0, w) = 2\}$$

R: $1^*01^*01^*$

P: Sobre o alfabeto $A = \{a, b, \dots, z\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(a, w) = 3\}$$

R: $(b|c|\dots|z)^*a(b|c|\dots|z)^*a(b|c|\dots|z)^*a(b|c|\dots|z)^*$

Extensões notacionais

Por forma a simplificar ao máximo a construção de expressões regulares é usual definir-se algumas extensões.

- Uma ou mais ocorrências:
$$e^+ = ee^*$$
- Uma ou nenhuma ocorrência:
$$e? = (e|\epsilon)$$
- Um símbolo dum sub-alfabeto:
$$[a_1a_2\dots a_n] = a_1|a_2|\dots|a_n$$
$$[a_1-a_n] = a_1|a_2|\dots|a_n$$
- Um símbolo fora dum sub-alfabeto:
$$[\wedge a_1a_2\dots a_n] \text{ ou (ANTLR): } \sim[a_1a_2\dots a_n]$$
$$[\wedge a_1-a_n] \text{ ou (ANTLR): } \sim[a_1-a_n]$$

– n ocorrências:

$$e\{n\} = \underbrace{e \cdot e \cdot \dots \cdot e}_n$$

– de n_1 a n_2 ocorrências:

$$e\{n_1, n_2\} = \underbrace{e \cdot e \cdot \dots \cdot e}_{n_1, n_2}$$

– n ou mais ocorrências:

$$e\{n, \} = \underbrace{e \cdot e \cdot \dots \cdot e}_{n,}$$

Expressões regulares: mais exemplos

P: Sobre o alfabeto $A = \{0, 1\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(0, w) = 2\}$$

R: $1^*01^*01^* = (1^*0)\{2\}1^*$

P: Sobre o alfabeto $A = \{a, b, \dots, z\}$ construa uma ER para a linguagem:

$$L = \{w : w \in A^* \wedge \#(a, w) = 3\}$$

R: $(b|c|\dots|z)^*a(b|c|\dots|z)^*a(b|c|\dots|z)^*a(b|c|\dots|z)^*$
 $= ([b-z]^*a)\{3\}[b-z]^*$

Outras extensões notacionais

Existem outras extensões a expressões regulares (utilizadas, por exemplo, em muitos comandos UNIX):

| Símbolo: | Significado: |
|----------|--|
| . | um símbolo qualquer diferente de $\backslash n$ (em ANTLR significa diferente de EOF) |
| ^ | palavra vazia no início de linha |
| \$ | palavra vazia no fim de linha |
| \< | palavra vazia no início de palavra |
| \> | palavra vazia no fim de palavra |

4.1 Gramática para expressões regulares

- Podemos definir a linguagem das expressões regulares com uma gramática (A é o conjunto dos caracteres):

ER \rightarrow ER '|' Term $\{alternativa\}$

ER \rightarrow Term

Term \rightarrow Term Primary $\{concatenação\}$

Term \rightarrow Primary

Primary \rightarrow Factor '*' $\{iteração\}$

Primary \rightarrow Factor

Factor \rightarrow '(' ER ')' $\{grupo\}$

Factor \rightarrow A $\{qualquer terminal\}$

- (Note que é uma gramática de tipo-2, i.e. livre de contexto.)

5 Conversão entre ER e GR

5.1 Conversão de ER para GR

- É suficiente obter a GR para as ER primitivas e aplicar as operações regulares sobre a GR
- A GR para a ER ε é dada por:
 $S \rightarrow \varepsilon$
- A GR para a ER a , qualquer que seja o a , é dada por:
 $S \rightarrow a$
- Vamos exemplificar com a ER $e = (a|b)^*a$

1. Primeiro definimos regras para os símbolos terminais:

$$S_1 \rightarrow a$$

$$S_2 \rightarrow b$$

2. Para reconhecer $(a|b)$ temos ($S = S_3$):

$$S_3 \rightarrow S_1 | S_2$$

$$S_1 \rightarrow a$$

$$S_2 \rightarrow b$$

3. Para reconhecer $(a|b)^*$ temos ($S = S_3$):

$$S_3 \rightarrow S_1 | S_2 | \varepsilon$$

$$S_1 \rightarrow S_3 a$$

$$S_2 \rightarrow S_3 b$$

4. Por fim para reconhecer a ER $(a|b)^*a$ temos ($S = S_4$):

$$S_4 \rightarrow S_3 a$$

$$S_3 \rightarrow S_1 | S_2 | \varepsilon$$

$$S_1 \rightarrow S_3 a$$

$$S_2 \rightarrow S_3 b$$

5.2 Conversão de GR para ER

- Seja $G_1 = (T_1, N_1, S_1, P_1)$ uma gramática regular qualquer.
- Uma ER que represente a mesma linguagem que a gramática G pode ser obtida por um processo de transformação de equivalência:

1. Converte-se a gramática G no conjunto de triplos seguinte:

$$\mathcal{E} = \{E, \varepsilon, S\} \cup$$

$$\{(A, w, B) : (A \rightarrow wB) \in P\} \cup$$

$$\{(A, w, \varepsilon) : (A \rightarrow w) \in P\}$$

$$\text{com } E \notin N \wedge w \in T^* \wedge A \in N \wedge B \in N$$

2. Removem-se, por transformações de equivalência, um a um, todos os símbolos de N , até se obter um único triplo da forma: (E, e, ε) . A ER equivalente será a expressão e .

- Remoção dos símbolos de N :

(A) Substituir todos os triplos da forma (A, β_i, B) por um único (A, w_1, B) , onde $w_1 = \beta_1 | \beta_2 | \dots | \beta_n$

(B) Substituir todos os triplos da forma (B, α_i, B) por um único (B, w_2, B) , onde $w_2 = \alpha_1 | \alpha_2 | \dots | \alpha_m$

(C) Substituir todos os triplos da forma (B, γ_i, C) por um único (B, w_3, C) , onde $w_3 = \gamma_1 | \gamma_2 | \dots | \gamma_k$

(D) Substituir o triplo de triplos $((A, w_1, B), (B, w_2, B), (B, w_3, C))$ pelo triplo $(A, w_1 w_2^* w_3, C)$

- Vamos exemplificar com a seguinte GR:

$$S \rightarrow aS \mid bS \mid cS \mid abaX$$

$$X \rightarrow aX \mid bX \mid cX \mid \varepsilon$$

$$\mathcal{E} = \{(E, \varepsilon, S), (S, a, S), (S, b, S), (S, c, S), (S, aba, X), \\ (X, a, X), (X, b, X), (X, c, X), (X, \varepsilon, \varepsilon)\}$$

$$(B) = \{(E, \varepsilon, S), (S, a \mid b \mid c, S), (S, aba, X), \\ (X, a, X), (X, b, X), (X, c, X), (X, \varepsilon, \varepsilon)\}$$

$$(D) = \{(E, (a \mid b \mid c)^* aba, X), \\ (X, a, X), (X, b, X), (X, c, X), (X, \varepsilon, \varepsilon)\}$$

$$(B) = \{(E, (a \mid b \mid c)^* aba, X), (X, a \mid b \mid c, X), (X, \varepsilon, \varepsilon)\}$$

$$(D) = \{(E, (a \mid b \mid c)^* aba(a \mid b \mid c)^*, \varepsilon)\}$$

6 Reconhecimento de *tokens*

- Já vimos como se podem expressar padrões utilizando *expressões regulares*.
- Assim sendo, vamos definir as expressões regulares para os *tokens* do seguinte excerto duma linguagem: ¹

$$\begin{aligned} stmt &\rightarrow \text{if } expr \text{ then } stmt \\ &\quad \mid \text{if } expr \text{ then } stmt \text{ else } stmt \\ &\quad \mid \varepsilon \\ expr &\rightarrow term \text{ relop } term \\ &\quad \mid term \\ term &\rightarrow \text{id} \\ &\quad \mid \text{number} \end{aligned}$$

- Os símbolos terminais desta gramática são: **if**, **then**, **else**, **relop**, **id** e **number**.
- Os padrões para reconhecer estes *tokens* podem ser descritos com expressões regulares:

$$\begin{aligned} \text{if} &\rightarrow \text{if} \\ \text{then} &\rightarrow \text{then} \\ \text{else} &\rightarrow \text{else} \\ \text{relop} &\rightarrow < \mid > \mid <= \mid >= \mid = \mid <> \\ \text{digit} &\rightarrow [0-9] \\ \text{digits} &\rightarrow \text{digit}^+ \\ \text{number} &\rightarrow \text{digits} (\cdot \text{digits})? (E [+-]? \text{digits})? \\ \text{letter} &\rightarrow [A-Za-z] \\ \text{id} &\rightarrow \text{letter} (\text{letter} \mid \text{digit})^* \end{aligned}$$

¹Exemplo retirado do livro: “Compilers: Principles, Techniques, & Tools”, 2ed, Aho, et.al

- Adicionalmente, o analisador léxico deve reconhecer e eliminar os caracteres correspondentes ao espaço em branco:

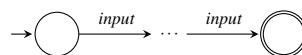
ws \rightarrow (**blank** | **tab** | **newline**)⁺

- Vamos tentar construir um analisador léxico que para além de reconhecer os *tokens* crie a seguinte informação:

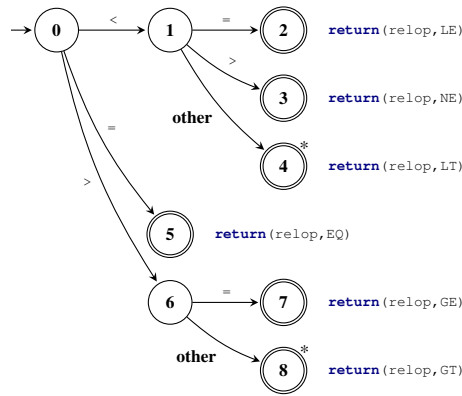
| <i>tokens</i> | Nome | Valor |
|---------------|--------|------------------------|
| ws | – | – |
| if | if | – |
| then | then | – |
| else | else | – |
| id | id | texto do identificador |
| number | number | texto do número |
| < | relop | LT |
| <= | relop | LE |
| = | relop | EQ |
| <> | relop | NE |
| > | relop | GT |
| >= | relop | GE |

6.1 Diagramas de transição

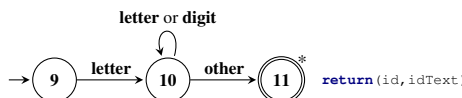
- Como passo intermédio para a construção do analisador léxico, vamos converter “à mão” as expressões regulares em máquinas de estados (representadas por *diagramas de transição*)
- Veremos mais à frente que este processo pode ser sistematizado recorrendo a *autómatos finitos*.
- Os diagramas de transição contêm uma coleção de estados (representados por círculos).
- Cada estado representa uma sequência de condições que ocorreram no processo de reconhecimento léxico.
- Isto é, cada estado representa o que já aconteceu até esse ponto no reconhecimento da sequência de caracteres de entrada no analisador.
- As transições são dirigidas de um estado para outro, e são anotadas com a entrada que lhes corresponde.



- As convenções a aplicar a estes diagramas são as seguintes:
 1. Cada estado é representado por um círculo e tem a si associado um rótulo que o identifica (em geral, um número ou uma letra).
 2. Alguns estados são considerados como *finais* (ou de aceitação). Estes estados indicam que um *token* foi reconhecido. Estes estados são representados com um círculo duplo.
 3. Se, por necessidade, tiver sido consumido um carácter a mais nesse processo de aceitação final, o nó que lhe corresponde será anotado com um asterisco.
 4. As transições entre estados são representadas por setas anotadas com o carácter que a despoleta.
 5. Um estado é designado como estado inicial, sendo indicado por uma transição (seta) sem estado de origem.
- Na construção destes diagramas vamos simplesmente enumerar novos estados por cada transição resultante de um carácter que, de alguma forma, avance no reconhecimento do *token*.
- O diagrama de transição para reconhecer os operadores relacionais pode ser o seguinte:

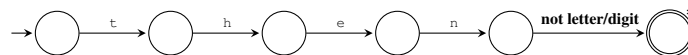


- Note que, para este tipo de *tokens*, este diagrama é uma estrutura de dados tipo árvore (deitada). Isso acontece em *tokens* fixos como o exemplificado ou as palavras reservadas.
- O diagrama de transição para identificadores:

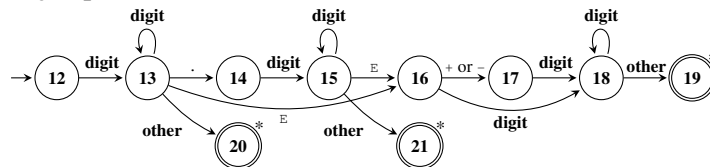


- O reconhecimento de identificadores pode levantar um problema de ambiguidade.
- De facto, as palavras reservadas da linguagem (ex: **then**) também podem ser reconhecidas como identificadores.
- Para resolver este problema, os analisadores léxicos dão prioridade a *tokens* que consomem mais caracteres e, em caso de conflito, estabelecem diferentes prioridades entre estes.
- Assim, o conflito entre identificadores e palavras reservadas é resolvido dando mais prioridade a estas.

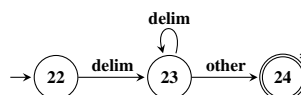
- O diagrama de transição para palavras reservadas é aqui exemplificado com o token **then**:



- O diagrama de transição para números:



- O diagrama de transição para espaço em branco:



- Agora podemos traduzir de uma forma quase automática os diagramas de transição para analisadores léxicos:

```
protected Token getRelop() {
    Token res = null;
    char c = 0; boolean fail = false; int state = 0;
    while(!fail && res == null) {
        switch(state) {
            case 0:
                c = nextChar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else { fail = true; retract(1); }
                break;
            case 1:
                ...
            case 2:
                res = new Token("relop", "LE");
                break;
            ...
        }
    }
}
```

```

        case 4:
            res = new Token("relop", "LT");
            retract(1);
            break;
        ...
    }
}

return res;
}

```

- Note que nos estados que requerem um carácter para fazerem uma transição $\{0, 1, 6\}$, é invocada a função `nextChar`; assim como os estados com asterisco, o carácter a mais é repostado com a função `retract`
- Podemos implementar uma função por tipo de *token* (`getID`, `getReserved`, `getWS`, `getnumber`), e depois invocar sequencialmente cada uma delas (até que uma seja bem sucedida).

```

public Token nextToken() {
    Token res = getWS();
    if (res == null)
        res = getReserved();
    if (res == null)
        res = getID();
    if (res == null)
        res = getNumber();
    if (res == null)
        res = getRelop();
    if (EOF)
        res = new Token("EOF", "");
    else if (res == null)
        res = new Token("ERROR", "");
    return res;
}

```

- No entanto, esta solução não é a mais eficiente já quem na presença de falhas, estamos a rebobinar a fila de caracteres de entrada.
- Alternativamente, podemos tentar executar os vários diagramas em paralelo.
- Se se utilizar uma numeração diferente nos estados de cada *token* (como foi feito neste exemplo), a melhor solução será simplesmente juntar todas as máquinas de estados numa única.
- Esta solução não só pode ser automatizada, como também é bastante eficiente (isso pode ser medido pelo número de vezes em que é necessário voltar atrás no consumo de caracteres, i.e., nas invocações da função `retract`).
- As máquinas que permitem uma aproximação automática a este problema são os chamados *autómatos finitos* (como veremos, os diagramas de transição apresentados descrevem autómatos finitos deterministas incompletos).

```

protected Token get() {
    Token res = null;
    char c=0; String value=""; boolean fail=false; int state=0;
    while(!fail && res == null) {
        switch(state) {
            case 0:
                c = nextChar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else state = 9;
                break;
            ...
            case 2:
                res = new Token("relop", "LE");
                break;
            ...
            case 9:
                if (Character.isLetter(c)) {value+=c; state=10;}
                else state = 22;
                break;
        }
    }
    return res;
}

```

```

...
    case 22:
        if (Character.isWhitespace(c)) state = 23;
        else { fail = true; retract(1); }
        break;
    ...
}
}
return res;
}

```

7 Autômatos finitos

- Um autômato é uma “máquina” que executa sobre uma determinada sequência de entradas passo a passo (de forma discreta no tempo).
- Internamente, o autômato contém uma máquina de estados, que vai evoluindo até uma de duas possibilidades: aceitar ou rejeitar a entrada.
- Um autômato finito é caracterizado por definir uma máquina de estados finita, em que as transições de estados apenas têm em conta o estado actual e a entrada.
- Graficamente, associam-se círculos aos estados e anota-se as transições entre estados com as entradas correspondentes.
- Os autômatos finitos são classificados em dois tipos:
 - a) *Autômato finito não determinista* (AFND): não existem restrições às condições colocadas nas transições. O mesmo símbolo (entrada) pode anotar várias transições a partir do mesmo estado, sendo também permitidas transições com a palavra vazia (ϵ).
 - b) *Autômato finito determinista* (AFD): cada estado indica no máximo uma transição com cada símbolo do alfabeto. Será incompleto caso não existam transições para todos os símbolos do alfabeto.
- Qualquer um destes tipos de autômatos reconhece as mesmas linguagens, e demonstra-se que essas linguagens correspondem às linguagens regulares.²
- Note que um AFD é um caso particular de um AFND.

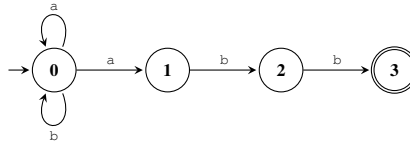
8 Autômato finito não determinista

- Um autômato finito não determinista é um autômato finito onde:
 - as transições estão associadas a símbolos individuais do alfabeto ou à palavra vazia (ϵ);
 - de cada estado *saem zero ou mais transições* por cada símbolo do alfabeto ou ϵ ;
 - há um estado inicial;
 - há zero ou mais estados de aceitação, que determinam as palavras aceites;
 - uma dada palavra sobre o alfabeto faz o sistema avançar do estado inicial a *zero ou mais estados finais*, determinando estes a aceitação ou rejeição da palavra.
- Os arcos múltiplos permitem alternativas de reconhecimento.
- Os arcos ausentes representam quedas num estado de *morte* (estado não representado, logo implicando palavra não reconhecida).

AFND: exemplo

- Um possível diagrama de transição para um AFND que reconhece a expressão regular $(a|b)^*abb$ é o seguinte:

²Com uma excepção menor: as linguagens regulares não expressam a linguagem vazia (\emptyset), o que pode ser trivialmente feito com autômatos finitos.



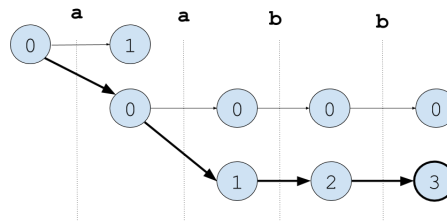
- É bem evidente o efeito do fecho de Kleene no diagrama, assim como a alternativa (**a|b**) e as sequências.

AFND: caminhos alternativos

- Será que a palavra **aabb** pertence esta linguagem?
- Existem 3 caminhos alternativos no diagrama:

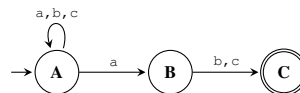
1. $0 \xrightarrow{a} 1 \xrightarrow{a} ?$
2. $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$
3. $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$

- Apenas o último termina no estado final.
- Podemos representar estes caminhos com uma estrutura tipo árvore (deitada):



AFND: exemplo

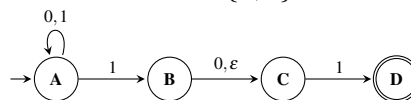
- Considerando o alfabeto $A = \{a, b, c\}$, que palavras são reconhecidas pelo autômato seguinte:



- Como expressão regular: $(a|b|c)^*a(b|c)$
- Como conjunto: $L = \{waX : w \in A^* \wedge X \in \{b, c\}\}$

AFND: exemplo com transições ϵ

- Considere o seguinte AFND sobre o alfabeto $A = \{0, 1\}$:

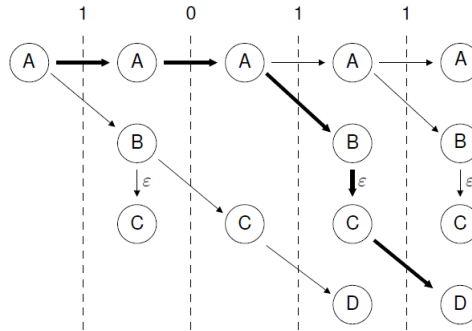


- Será que a palavra **1011** é reconhecida?
- Há 6 caminhos possíveis:

1. $A \xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} A$
2. $A \xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} B$
3. $A \xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} B \xrightarrow{\epsilon} C$
4. $A \xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{\epsilon} C \xrightarrow{1} D$
5. $A \xrightarrow{1} B \xrightarrow{0} C \xrightarrow{1} D$
6. $A \xrightarrow{1} B \xrightarrow{\epsilon} C$

AFND: exemplo com transições ϵ

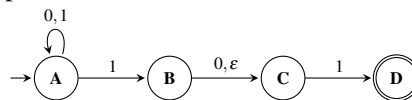
- Com a estrutura tipo árvore:



- A palavra é reconhecida uma vez que existe (pelo menos um) caminho que leva a **D**.

AFND: exemplo

- Que palavras são reconhecidas por este autômato?



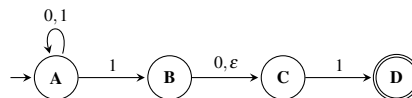
- Todas as palavras terminadas em **11** ou **101**.
- Como expressão regular: **$(0|1)^*10?1$**

AFND: definição formal

- Um automato finito não determinista é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:
 - A é o alfabeto de entrada (sem a palavra vazia ϵ);
 - Q é um conjunto finito não vazio de estados;
 - $q_0 \in Q$ é o estado inicial;
 - $\delta \subseteq (Q \times A_\epsilon \times Q)$ é a relação de transição entre estados, com $A_\epsilon = A \cup \{\epsilon\}$;
 - $F \subseteq Q$ é o conjunto dos estados de aceitação.

AFND: outro exemplo

- Represente analiticamente o AFND:



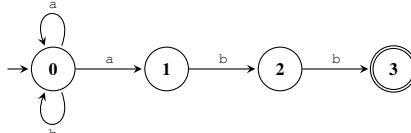
- O quintuplo $M = (A, Q, q_0, \delta, F)$ é:
 - $A = \{0, 1\}$
 - $Q = \{A, B, C, D\}$
 - $q_0 = A$
 - $F = \{D\}$
 - $\delta = \{(A, 0, A), (A, 1, A), (A, 1, B), (B, \epsilon, C), (B, 0, C), (C, 1, D)\}$
- Como expressão regular: **$(0|1)^*1(0|\epsilon)1$**
- Ou alternativamente: **$(0|1)^*1(0)?1$**

AFND: linguagem reconhecida

- Diz-se que um AFND $M = (A, Q, q_0, \delta, F)$, *aceita* uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \dots u_n$, com $u_i \in A_\epsilon$, e existir uma sequência de estados s_0, s_1, \dots, s_n , que satisfaça as seguintes condições:
 - $s_0 = q_0$
 - qualquer que seja o $i = 1, \dots, n$, $(s_{i-1}, u_i, s_i) \in \delta$
 - $s_n \in F$
- Caso contrario diz-se que M *rejeita* a entrada.
- Note que n pode ser maior que $|u|$, porque alguns dos u_i podem ser ϵ .
- Usar-se-á a notação $q_i \xrightarrow{u} q_j$ para representar a existência de uma palavra u que conduza do estado q_i ao estado q_j
- Usando esta notação tem-se $L(M) = \{u : q_0 \xrightarrow{u} q_f \wedge q_f \in F\}$

8.1 Tabelas de transição

- Podemos representar um AFND por uma *tabela de transição*, em que as linhas correspondem aos estados, e as colunas aos símbolos de entrada incluindo a palavra vazia (A_ϵ).
- A tabela de transição para o AFND:

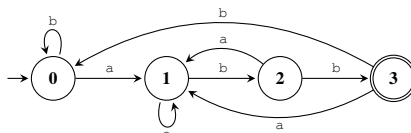


é a seguinte:

| STATE | a | b | ϵ |
|-----------------|-------------|-------------|-------------|
| $\rightarrow 0$ | $\{0, 1\}$ | $\{0\}$ | \emptyset |
| 1 | \emptyset | $\{2\}$ | \emptyset |
| 2 | \emptyset | $\{3\}$ | \emptyset |
| 3_f | \emptyset | \emptyset | \emptyset |

9 Autômato finito determinista

- Um autômato finito determinista (AFD) é um caso especial de um AFND onde:
 - Não há transições com a palavra vazia (ϵ).
 - Para cada estado s e símbolo de entrada a existe no máximo uma transição de s anotada com a .
- Num AFD completo, existe uma transição para todos os símbolos do alfabeto.
- Um diagrama de transição para um AFD que reconhece a expressão regular $(a|b)^*abb$ é o seguinte:



Autômato finito determinista: tabela de transição

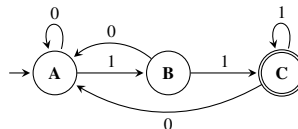
- Se estivermos a representar um AFD com uma tabela de transição, então cada entrada será um único estado (pelo que deixa de ser necessário representá-lo como conjunto):

| STATE | a | b |
|-----------------|---|---|
| $\rightarrow 0$ | 1 | 0 |
| 1 | 1 | 2 |
| 2 | 1 | 3 |
| 3_f | 1 | 0 |

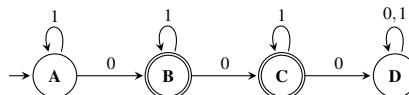
- Enquanto um AFND é uma representação abstracta dum algoritmo para reconhecer expressões regulares, o AFD é um algoritmo simples, concreto e muito eficiente para o mesmo fim.
- Felizmente é possível converter um AFND num AFD que reconhece a mesma linguagem regular.

AFD: exemplo

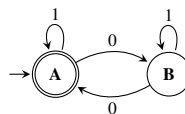
- Que palavras são reconhecidas pelo autómato seguinte:



- Todas as palavras terminadas em 11.
- Como expressão regular: $(0|1)^*11$
- Que palavras são reconhecidas pelo autómato seguinte:



- Todas as palavras com 1 ou 2 zeros.
- Como expressão regular: $1^*01^*0?1^*$
- Que palavras são reconhecidas pelo autómato seguinte:



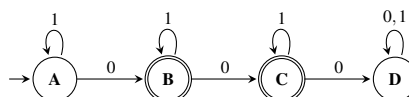
- Todas as palavras com um número par de zeros.
- Como expressão regular: $1^*(01^*0)^*1^*$

AFD: definição formal

- Um automato finito determinista é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:
 - A é o alfabeto de entrada (sem a palavra vazia ϵ);
 - Q é um conjunto finito não vazio de estados;
 - $q_0 \in Q$ é o estado inicial;
 - $\delta : Q \times A \rightarrow Q$ é uma função que determina a transição entre estados;
 - $F \subseteq Q$ é o conjunto dos estados de aceitação.
- Note que apenas muda a definição de δ relativamente aos AFND.
- A função δ pode ser representada pelo conjunto de triplos $\in Q \times A \times Q$, ou pela tabela de transição (matriz de $|Q|$ linhas e $|A|$ colunas).

AFD: exemplo (2)

- Represente analiticamente o AFD:



- O quintuplo $M = (A, Q, q_0, \delta, F)$ é:

- $A = \{0, 1\}$
- $Q = \{A, B, C, D\}$
- $q_0 = A$
- $F = \{B, C\}$

| | | | |
|--|-----------------|---|---|
| $\delta = \{(A, 0, B), (A, 1, A),$ $(B, 0, C), (B, 1, B),$ $(C, 0, D), (C, 1, C),$ $(D, 0, D), (D, 1, D)\}$ | STATE | 0 | 1 |
| | $\rightarrow A$ | B | A |
| | B_f | C | B |
| | C_f | D | C |
| | D | D | D |

AFD: linguagem reconhecida

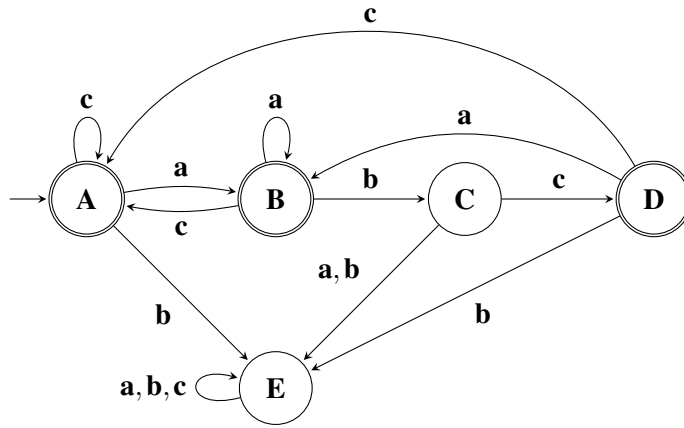
- Diz-se que um AFD $M = (A, Q, q_0, \delta, F)$, *aceita* uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \dots u_n$ e existir uma sequência de estados s_0, s_1, \dots, s_n , que satisfaça as seguintes condições:
 1. $s_0 = q_0$
 2. qualquer que seja o $i = 1, \dots, n$, $s_i = \delta(s_{i-1}, u_i)$
 3. $s_n \in F$
- Caso contrario diz-se que M *rejeita* a entrada.

10 Autômato finito determinista

10.1 Projecto de autômato finito determinista

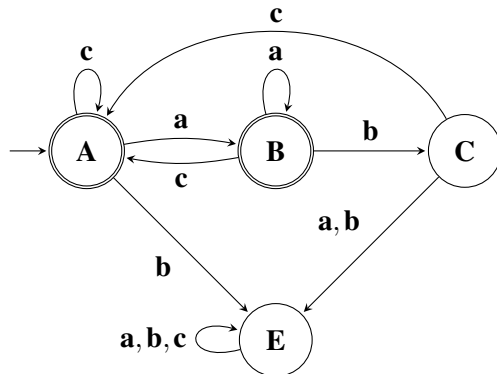
- Projete um AFD que reconheça as sequências definidas sobre o alfabeto $A = \{a, b, c\}$ que satisfazem o requisito de qualquer **b** ter um **a** imediatamente à sua esquerda e um **c** imediatamente à sua direita.
- Aproximação possível:
 - Note que para estar num estado final, caso apareça um símbolo **b** na entrada, é necessário garantir um estado prévio (**a**) e um estado seguinte (**c**).
 - Note também que esse estado seguinte é final (cumpre o requisito), o mesmo acontecendo com o estado inicial.
 - Temos assim a necessidade de pelo menos quatro estados (estado inicial, e os três estados correspondentes à sequência **abc**).
 - Por outro lado, caso apareça um símbolo **b**, sem que exista um **a** imediatamente à sua esquerda, ou um **c** imediatamente à sua direita, podemos desde logo afirmar que a entrada não cumpre o requerido pelo que precisamos de um estado que não seja final mas donde não seja possível sair (tipo “buraco negro”).

- Chegamos assim ao seguinte AFD:



| STATE | a | b | c |
|-------------------|---|---|---|
| $\rightarrow A_f$ | B | E | A |
| B_f | B | C | A |
| C | E | E | D |
| D_f | B | E | A |
| E | E | E | E |

- Será que podemos simplificar esta autómato?
- Se compararmos os estados **A** e **D**, constata-se que são ambos finais e as transições para fora são equivalentes.
- logo podem ser fundidos:



| STATE | a | b | c |
|-------------------|---|---|---|
| $\rightarrow A_f$ | B | E | A |
| B_f | B | C | A |
| C | E | E | A |
| E | E | E | E |

10.2 Redução de autómato finito determinista

Redução de AFD

- O exemplo anterior mostra que por vezes é possível simplificar os AFD reduzindo o número de estados.
- A ideia base é ter um procedimento sistemático para identificar estados equivalentes, fundindo-os num único estado.
- Dois estados são equivalentes se forem do mesmo tipo (final ou não final) e se todas as suas transições para o exterior forem iguais (i.e. para os mesmos estados).
- Formalmente podemos ir mais longe e afirmar que dois estados s_i e s_j de um autómato $M = (A, Q, q_0, \delta, F)$ são equivalentes se e só se

$$\forall u \in A^* \quad \delta^*(s_i, u) \in F \Leftrightarrow \delta^*(s_j, u) \in F$$

- Em que o fecho da função de transição $\delta^* : Q \times A \rightarrow Q$ é definido por:

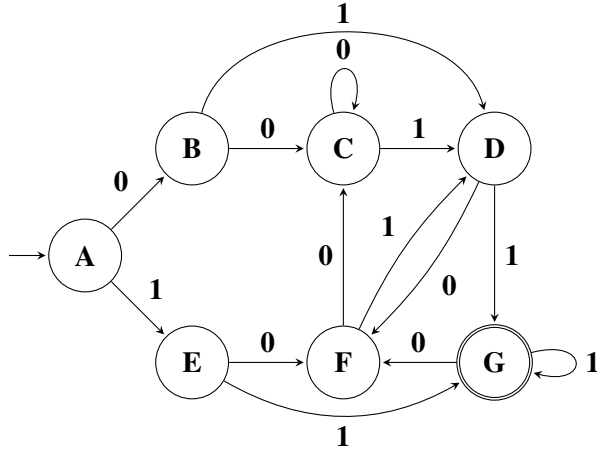
$$\delta^*(q, \varepsilon) = q$$

$$\delta^*(q, av) = \delta^*(\delta(q, a), v), \quad \text{com } a \in A \wedge v \in A^*$$

- Note que esse autômato M aceita uma sequência de símbolos u se $\delta^*(q_0, u) \in F$.
- A linguagem reconhecida por $M - L(M)$ – é definida por

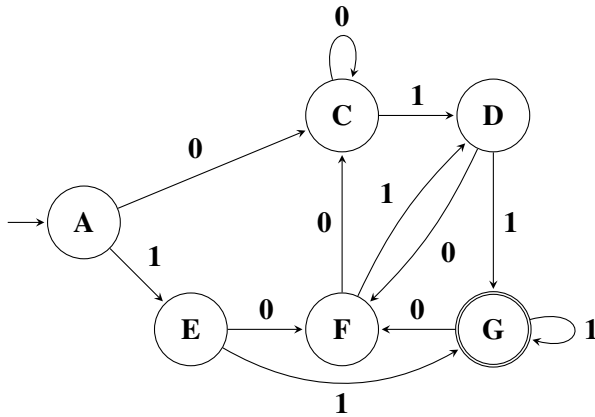
$$L(M) = \{u \in A^* \mid M \text{ aceita } u\} = \{u \in A^* \mid \delta^*(q_0, u) \in F\}$$

- Considere o seguinte autômato:



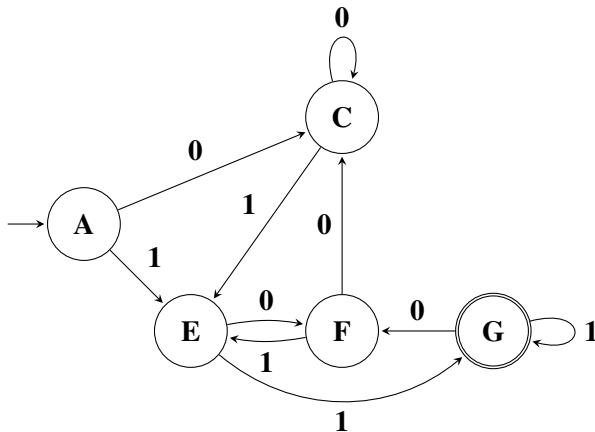
| STATE | 0 | 1 |
|-----------------|---|---|
| $\rightarrow A$ | B | E |
| B | C | D |
| C | C | D |
| D | F | G |
| E | F | G |
| F | C | D |
| G_f | F | G |

- Aplicando primeiro a regra mais intuitiva de fundir estados na mesma situação (mesmo tipo e transições iguais).
- Olhando para a tabela de transições constata-se que os estados **B** e **C** são equivalentes:



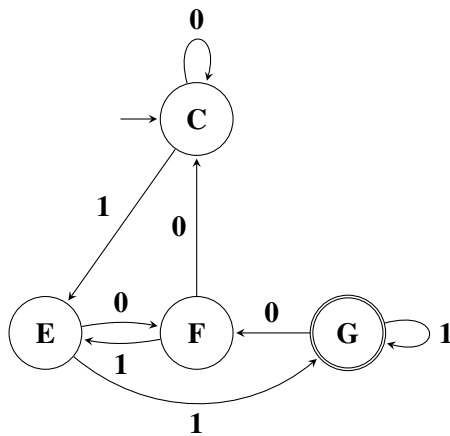
| STATE | 0 | 1 |
|-----------------|---|---|
| $\rightarrow A$ | C | E |
| C | C | D |
| D | F | G |
| E | F | G |
| F | C | D |
| G_f | F | G |

- Temos também equivalência nos estados **D** e **E**:



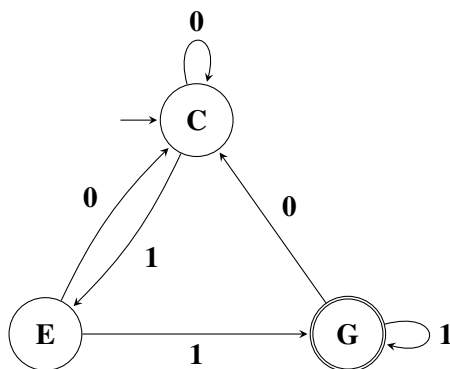
| STATE | 0 | 1 |
|-----------------|---|---|
| $\rightarrow A$ | C | E |
| C | C | E |
| E | F | G |
| F | C | E |
| G_f | F | G |

- Agora há equivalência nos estados **A** e **C**:



| STATE | 0 | 1 |
|-----------------|---|---|
| $\rightarrow C$ | C | E |
| E | F | G |
| F | C | E |
| G_f | F | G |

- Por fim há equivalência nos estados **C** e **F**:



| STATE | 0 | 1 |
|-----------------|---|---|
| $\rightarrow C$ | C | E |
| E | C | G |
| G_f | C | G |

- Note que os estados **E** e **G** embora tendo as mesmas transições, não são equivalentes.
- Este método para simplificar AFDs não garante uma minimização total, já que não lida com o problema de poder haver ciclos entre estados equivalentes.
- Esse problema é resolvido com o algoritmo que se apresenta a seguir.

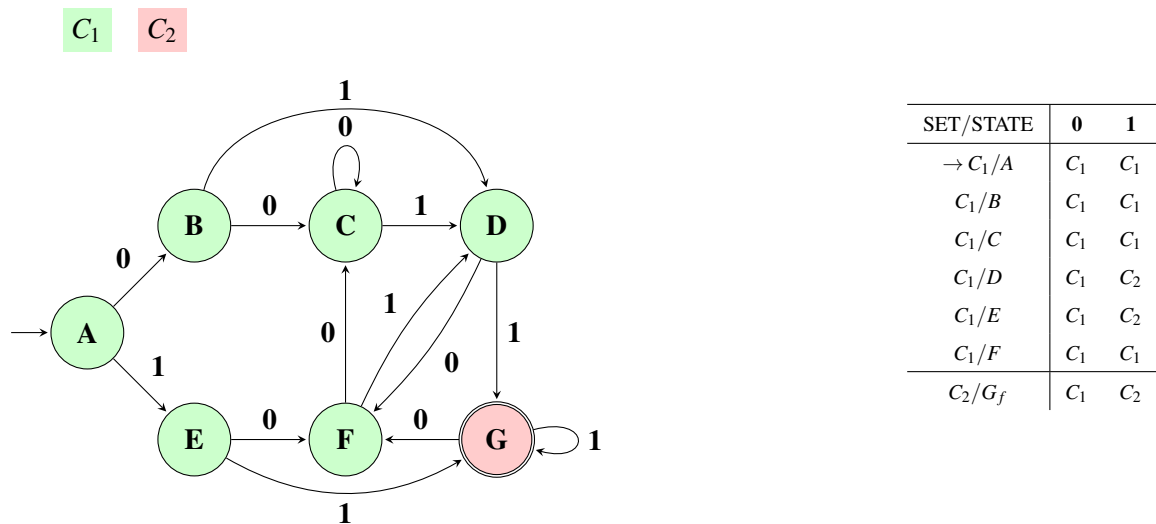
Algoritmo de redução de AFD

- Procedimento:
 1. Primeiro divide-se os estados em dois conjuntos: o conjunto dos estados finais (aceitação) e o conjunto com os restantes estados;³
 2. Depois vai-se particionando sucessivamente os conjuntos existentes, sempre que dentro do conjunto existam estados que tenham transições com o mesmo símbolo para diferentes conjuntos.
 3. O passo anterior é repetido até que não sejam possíveis mais partições. Nessa situação, o AFD está minimizado.
- Recuperando o exemplo anterior, temos como conjuntos de partida:

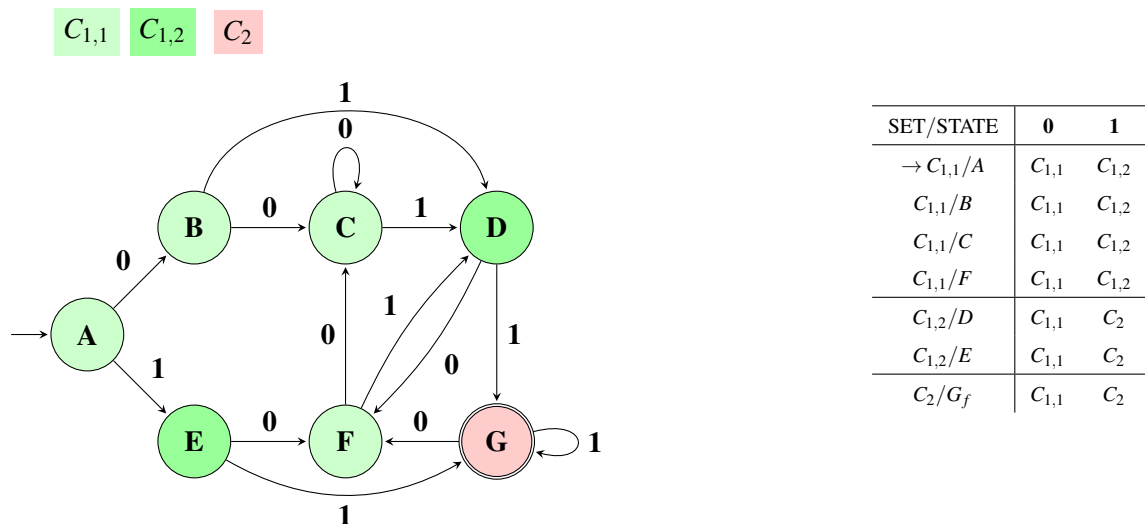
$$C_1 = Q - F = \{A, B, C, D, E, F\}$$

$$C_2 = F = \{G\}$$

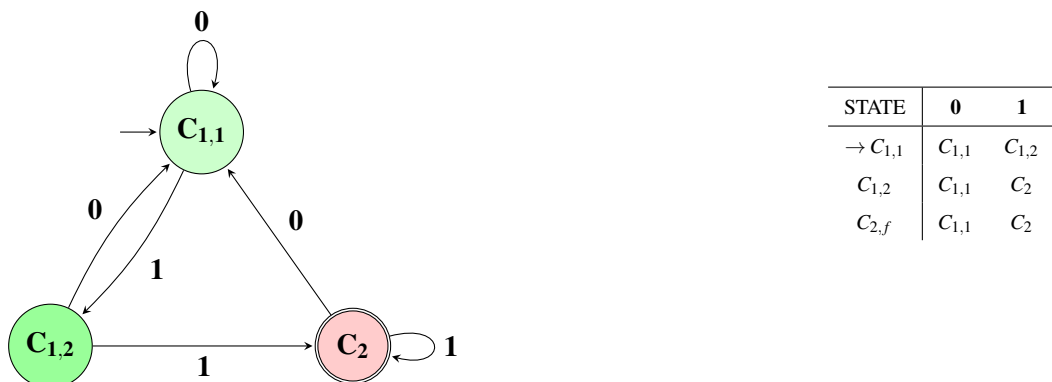
³Se o AFD for incompleto, então existirá um terceiro conjunto: o conjunto de erro que representa todos os estados de erro.



- O conjunto C_1 tem de ser partido em dois, já que para a entrada **1** os estados **D** e **E** têm uma transição para um conjunto diferente do que os restantes estados (C_2).

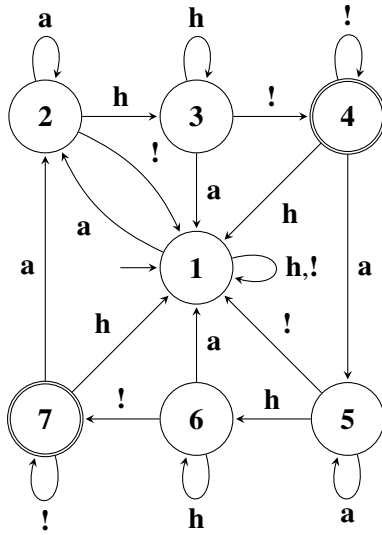


- Chegamos assim a um AFD minimizado equivalente ao que já tínhamos chegado:



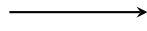
Redução de AFD: exemplo 2

- Considere o seguinte autômato para reconhecer frases tipo **ah! ah!**:



| STATE | a | h | ! |
|----------------|---|---|---|
| → 1 | 2 | 1 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 1 | 3 | 4 |
| 4 _f | 5 | 1 | 4 |
| 5 | 5 | 6 | 1 |
| 6 | 1 | 6 | 7 |
| 7 _f | 2 | 1 | 7 |

| STATE | a | h | ! |
|----------------|---|---|---|
| → 1 | 2 | 1 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 1 | 3 | 4 |
| 4 _f | 5 | 1 | 4 |
| 5 | 5 | 6 | 1 |
| 6 | 1 | 6 | 7 |
| 7 _f | 2 | 1 | 7 |



| STATE | a | h | ! |
|----------------|----------------|----------------|----------------|
| → 1 | C ₁ | C ₁ | C ₁ |
| 2 | C ₁ | C ₁ | C ₁ |
| 3 | C ₁ | C ₁ | C ₂ |
| 5 | C ₁ | C ₁ | C ₁ |
| 6 | C ₁ | C ₁ | C ₂ |
| 4 _f | C ₂ | C ₁ | C ₂ |
| 7 _f | C ₂ | C ₁ | C ₂ |

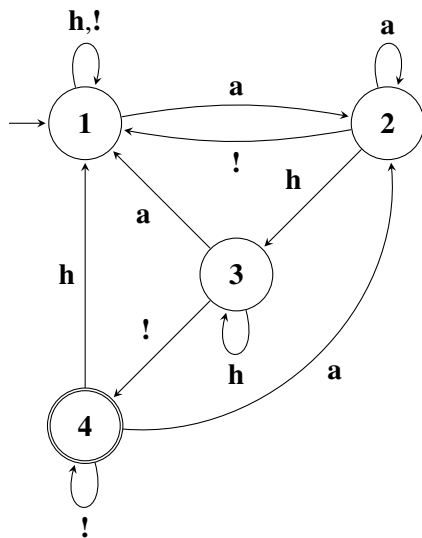


| STATE | a | h | ! |
|----------------|--------------------|--------------------|--------------------|
| → 1 | C _{1,1,1} | C _{1,1,2} | C _{1,1,1} |
| 2 | C _{1,1,2} | C _{1,1,2} | C _{1,1,1} |
| 5 | C _{1,1,2} | C _{1,1,2} | C _{1,1,1} |
| 3 | C _{1,2} | C _{1,1,1} | C ₂ |
| 6 | C _{1,2} | C _{1,1,1} | C ₂ |
| 4 _f | C ₂ | C _{1,1,2} | C ₂ |
| 7 _f | C ₂ | C _{1,1,2} | C ₂ |



| STATE | a | h | ! |
|----------------|------------------|------------------|------------------|
| → 1 | C _{1,1} | C _{1,1} | C _{1,1} |
| 2 | C _{1,1} | C _{1,1} | C _{1,1} |
| 5 | C _{1,1} | C _{1,1} | C _{1,1} |
| 3 | C _{1,2} | C _{1,1} | C ₂ |
| 6 | C _{1,2} | C _{1,1} | C ₂ |
| 4 _f | C ₂ | C _{1,1} | C ₂ |
| 7 _f | C ₂ | C _{1,1} | C ₂ |

- Donde resulta o seguinte autómato final:



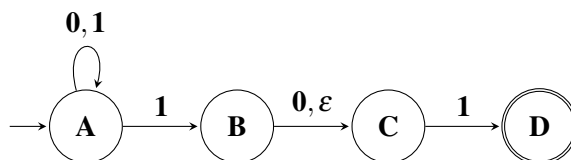
| STATE | a | h | ! |
|----------------|---|---|---|
| → 1 | 2 | 1 | 1 |
| 2 | 2 | 3 | 1 |
| 3 | 1 | 3 | 4 |
| 4 _f | 2 | 1 | 4 |

11 Conversão de AFND em AFD

- Como já foi referido um AFD é um AFND, mas o contrário não é necessariamente verdadeiro.
- Nos AFD as transições são funções e as tabelas de transição são mais simples havendo sempre uma transição para no máximo um único estado.
- Assim, em geral, a implementação de AFD é preferível à implementação de AFND.
- É sempre possível converter um AFND num AFD.
- Vamos ver um algoritmo genérico para esse efeito.
- A ideia geral do algoritmo resulta da constatação de que num AFND as transições fazem-se de um subconjunto dos seus estados para outro subconjunto.
- Assim podemos transformar um AFND num AFD tomando como estados os subconjuntos do AFND.
- Com esta estratégia, para um AFND com n estados no pior caso podemos ter 2^n estados no AFD.
- No entanto, em geral constata-se que o número de estados é da mesma ordem de grandeza.
- O algoritmo assenta na construção, passo a passo, da tabela de transição para o AFD, partindo do AFND.
- Considerando que: s é um qualquer estado do AFND, C é um conjunto de estados do AFND e a é um qualquer símbolo de entrada, então:

| Operação | Descrição |
|---------------------------------|--|
| $\varepsilon\text{-closure}(s)$ | Conjunto de estados do AFND para os quais pode haver uma transição a partir do estado s apenas pela palavra vazia (ε). |
| $\varepsilon\text{-closure}(C)$ | Conjunto de estados do AFND para os quais pode haver uma transição a partir de qualquer estado do conjunto C apenas pela palavra vazia. |
| $\text{move}(C, a)$ | Conjunto de estados do AFND para os quais pode haver uma transição a partir de qualquer estado do conjunto C pelo símbolo de entrada a . |

- O estado inicial do AFD será o resultante da aplicação de $\varepsilon\text{-closure}$ ao estado inicial do AFND.
- Os estados finais do AFD, serão todos os estados que contiverem pelo menos um estado final do AFND.
- Como exemplo, vamos considerar o AFND seguinte:



| STATE | 0 | 1 | ε |
|-----------------|-------------|-------------|---------------|
| $\rightarrow A$ | $\{A\}$ | $\{A, B\}$ | \emptyset |
| B | $\{C\}$ | \emptyset | $\{C\}$ |
| C | \emptyset | $\{D\}$ | \emptyset |
| D_f | \emptyset | \emptyset | \emptyset |

- Vamos identificar os estados do AFD por E_i , $i \in \mathbb{N}$
- O estado inicial para o AFD será dado por: $\varepsilon\text{-closure}(\{A\}) = \{A\} = E_1$
- Seguidamente vamos determinar os subconjuntos de estados AFND que resultam da transição de E_1 por cada símbolo do alfabeto.
 $\varepsilon\text{-closure}(\text{move}(E_1, 0)) = \varepsilon\text{-closure}(\{A\}) = \{A\} = E_1$
- Como chegámos a um subconjunto que já existe ($\{A\}$), não há lugar à criação de um novo estado para o AFD.
 $\varepsilon\text{-closure}(\text{move}(E_1, 1)) = \varepsilon\text{-closure}(\{A, B\}) = \{A, B, C\} = E_2$
- Agora temos um novo subconjunto pelo que é necessário um novo estado (E_2).
- Aplicamos agora a mesma receita a esse novo estado até que não resultem novos estados (situação em que teremos o AFD equivalente ao AFND de que partimos).
 $\varepsilon\text{-closure}(\text{move}(E_2, 0)) = \varepsilon\text{-closure}(\{A, C\}) = \{A, C\} = E_3$

$$\varepsilon\text{-closure}(\text{move}(E_2, 1)) = \varepsilon\text{-closure}(\{A, B, D\}) = \{A, B, C, D\} = E_4$$

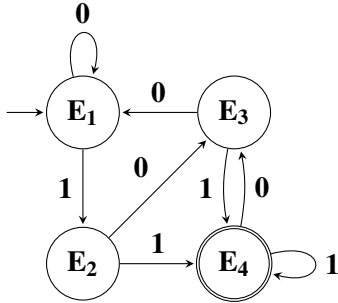
$$\varepsilon\text{-closure}(\text{move}(E_3, 0)) = \varepsilon\text{-closure}(\{A\}) = \{A\} = E_1$$

$$\varepsilon\text{-closure}(\text{move}(E_3, 1)) = \varepsilon\text{-closure}(\{A, B, D\}) = \{A, B, C, D\} = E_4$$

$$\varepsilon\text{-closure}(\text{move}(E_4, 0)) = \varepsilon\text{-closure}(\{A, C\}) = \{A, C\} = E_3$$

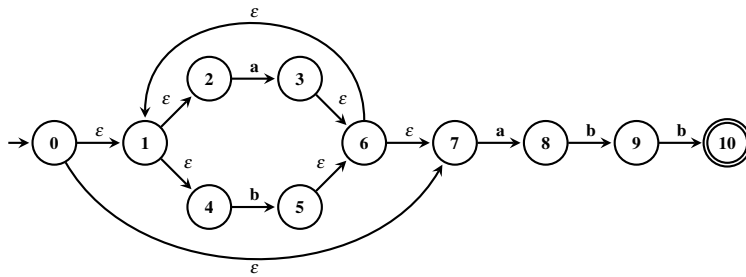
$$\varepsilon\text{-closure}(\text{move}(E_4, 1)) = \varepsilon\text{-closure}(\{A, B, D\}) = \{A, B, C, D\} = E_4$$

- Logo, vamos ter o seguinte AFD:



| AFND | AFD | 0 | 1 |
|--------------|------------------|----------------|----------------|
| {A} | → E ₁ | E ₁ | E ₂ |
| {A, B, C} | E ₂ | E ₃ | E ₄ |
| {A, C} | E ₃ | E ₁ | E ₄ |
| {A, B, C, D} | E _{4,f} | E ₃ | E ₄ |

- A AFND seguinte foi uma implementação (que, como veremos, resulta directamente da aplicação de um algoritmo) da expressão regular: **(a|b)*abb**



| STATE | a | b | ε |
|-----------------|-----|------|--------|
| → 0 | ∅ | ∅ | {1, 7} |
| 1 | ∅ | ∅ | {2, 4} |
| 2 | {3} | ∅ | ∅ |
| 3 | ∅ | ∅ | {6} |
| 4 | ∅ | {5} | ∅ |
| 5 | ∅ | ∅ | {6} |
| 6 | ∅ | ∅ | {1, 7} |
| 7 | {8} | ∅ | ∅ |
| 8 | ∅ | {9} | ∅ |
| 9 | ∅ | {10} | ∅ |
| 10 _f | ∅ | ∅ | ∅ |

- Estado inicial: $\varepsilon\text{-closure}(\{0\}) = \{0, 1, 2, 4, 7\} = A$

$$\varepsilon\text{-closure}(\text{move}(A, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

$$\varepsilon\text{-closure}(\text{move}(A, b)) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$$

$$\varepsilon\text{-closure}(\text{move}(B, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

$$\varepsilon\text{-closure}(\text{move}(B, b)) = \varepsilon\text{-closure}(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\} = D$$

$$\varepsilon\text{-closure}(\text{move}(C, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

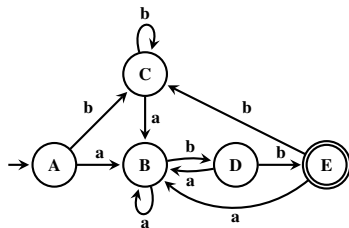
$$\varepsilon\text{-closure}(\text{move}(C, b)) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$$

$$\varepsilon\text{-closure}(\text{move}(D, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

$$\varepsilon\text{-closure}(\text{move}(D, b)) = \varepsilon\text{-closure}(\{5, 10\}) = \{1, 2, 4, 5, 6, 7, 10\} = E$$

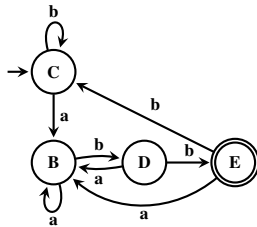
$$\varepsilon\text{-closure}(\text{move}(E, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

$$\varepsilon\text{-closure}(\text{move}(E, b)) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$$



| AFND | AFD | a | b |
|------------------------|----------------|---|---|
| {0, 1, 2, 4, 7} | → A | B | C |
| {1, 2, 3, 4, 6, 7, 8} | B | B | D |
| {1, 2, 4, 5, 6, 7} | C | B | C |
| {1, 2, 4, 5, 6, 7, 9} | D | B | E |
| {1, 2, 4, 5, 6, 7, 10} | E _f | B | C |

- Este AFD pode ainda ser minimizado (os estados A e C são equivalentes):



| AFD | a | b |
|----------------|---|---|
| B | B | D |
| → C | B | C |
| D | B | E |
| E _f | B | C |

12 Conversão de uma expressão regular num AFND

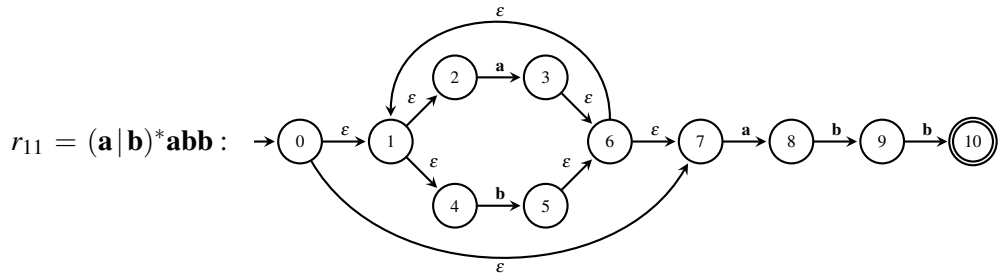
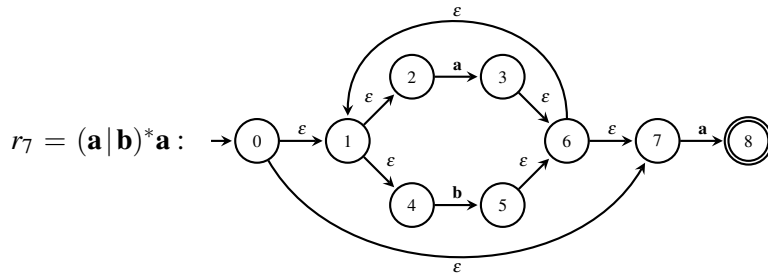
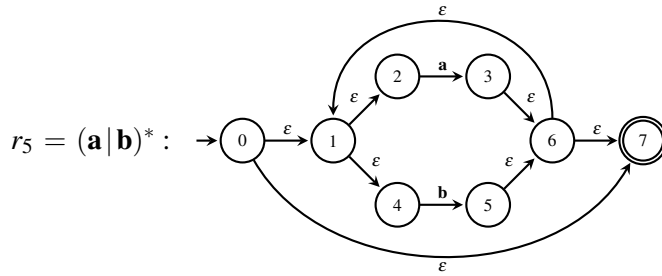
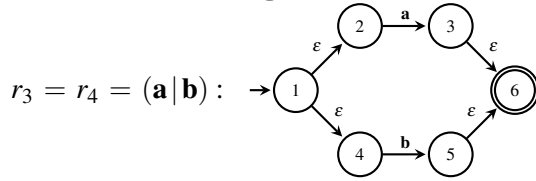
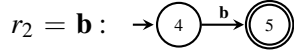
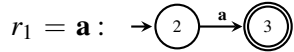
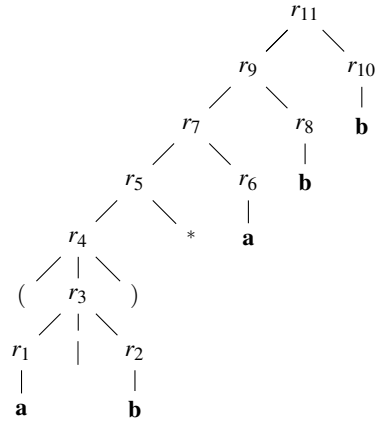
- Para compreendermos minimamente a construção de analisadores léxicos só falta abordarmos o problema da conversão automática de expressões regulares para autómatos.
- Para esse fim vamos apresentar um algoritmo (*McNaughton-Yamada-Thompson*) que converte uma qualquer ER num AFND.
- A estratégia baseia-se no seguinte:
 - Ter AFND definidos para ER elementares;
 - Ter padrões para AFND resultantes das operações sobre ER (reunião, concatenação, fecho de Kleene, ...).
 - Construir o AFND recorrendo à árvore sintáctica da ER.

- A tabela seguinte mostra os padrões para AFND de ER

| Descrição | ER | AFND |
|-------------------------|---------------|--|
| Linguagem vazia | $()$ | $\rightarrow (i)$ |
| Palavra vazia | ϵ | $\rightarrow (i) \xrightarrow{\epsilon} (f)$ |
| Símbolo do alfabeto | a | $\rightarrow (i) \xrightarrow{a} (f)$ |
| União de AFND | $(E_1 E_2)$ | |
| Concatenação de AFND | $E_1 E_2$ | |
| Fecho de Kleene de AFND | E^* | |

Conversão de uma ER num AFND: Exemplo

- Vamos então construir um AFND para a ER: $(a|b)^*abb$
- A árvore sintáctica desta ER é a seguinte:

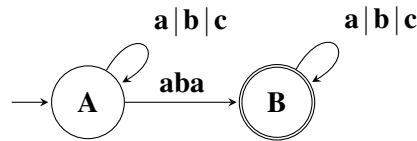


13 Autómatos finitos generalizados (AFG)

- Nos autómatos finitos apresentados as transições entre estados apenas decorrem de símbolos do alfabeto ou, no caso dos AFND, da palavra vazia (ϵ).
- No entanto podemos aproximar ainda mais os autómatos finitos das expressões regulares fazendo

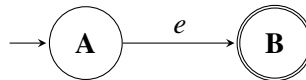
com que as transições possam decorrer de ER (nas quais os símbolos do alfabeto e a palavra vazia são casos elementares).

- Este tipo de autómatos designa-se por *Autômato finito generalizado* (AFG).
- Por exemplo, um AFG sobre o alfabeto $A = \{a, b, c\}$ para o conjunto de palavras que contém a palavra **aba** será:



13.1 AFG reduzido

- Um AFG com a forma



designa-se por *autômato finito generalizado reduzido*.

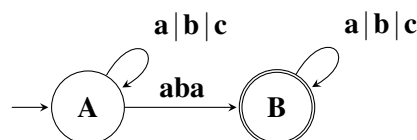
- Note que:
 - O estado A não é de aceitação e não tem arcos a chegar de outros estados.
 - O estado B é de aceitação e não tem arcos a sair.
- Se reduzir um AFG à forma anterior a expressão – e – é uma expressão regular equivalente ao autômato.

13.2 Conversão de uma AFG numa ER

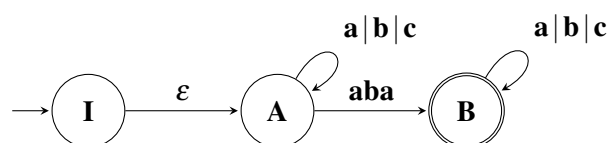
- Assim transformar uma AFG num AFG reduzido corresponde a determinar a ER que lhe é equivalente.
- Algoritmo de conversão:
 1. Transformação de um AFG noutra cujo estado inicial não tenha arcos a chegar.
 - Se necessário, acrescenta-se um novo estado inicial com um arco em ϵ para o antigo.
 2. Transformação de um AFG noutra com um único estado de aceitação, sem arcos de saída.
 - Se necessário, acrescenta-se um novo estado, que passa a ser o único de aceitação, que recebe arcos em ϵ dos anteriores estados de aceitação, que deixam de o ser.
 3. Eliminação dos restantes estados.
 - Os estados são eliminados um a um, em processos de transformação que mantêm a equivalência.

Conversão de uma AFG numa ER: Exemplo

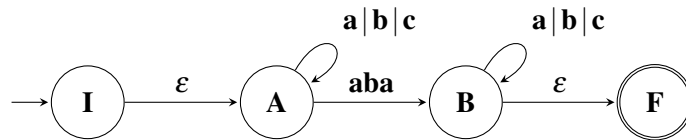
- Recuperando o AFG atrás apresentado vamos aplicar este algoritmo para o transformar numa ER.



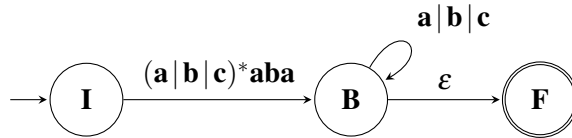
- Aplicando a regra 1 :



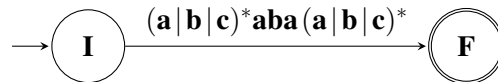
- Aplicando a regra 2 :



- Eliminando o estado A aplicando a regra 3 :



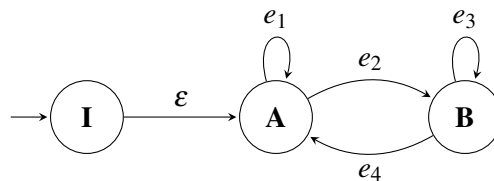
- Por fim, eliminando o estado B aplicando novamente a regra 3 :



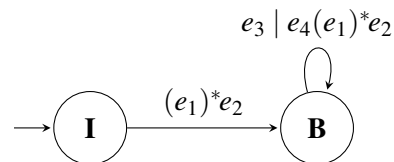
Eliminar estado com arcos a chegar de outros estados

- Se for necessário eliminar um estado que seja destino de arcos de outros estados, é necessário garantir – nesses estados – que o caminho de reconhecimento garantido pelo estado a eliminar não se altera.

4. Considerando que (e_1, e_2, e_3, e_4) são ER, a eliminação de estado A do AFG

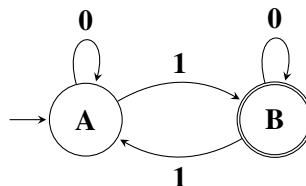


resulta no seguinte AFG:

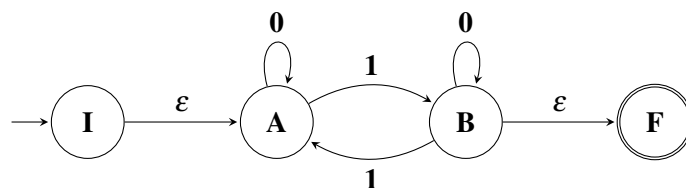


Conversão de uma AFG numa ER: Exemplo 2

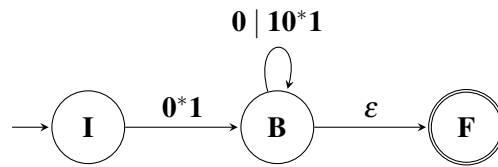
- Obtenha uma ER equivalente ao AF seguinte:



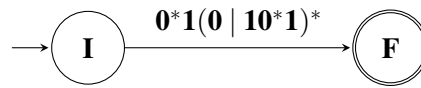
- Regras 1 e 2 :



- Eliminar o estado A pela regra 4 :



- Finalmente, eliminar o estado B pela regra 3 :



- Logo a ER equivalente será: $0^*1(0 \mid 10^*1)^*$