Tema 3

Análise Lexical

Gramáticas regulares, autómatos finitos e expressões regulares

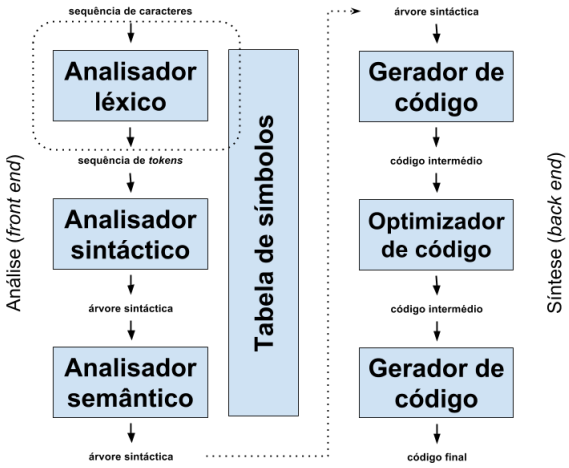
Linguagens Formais e Autómatos, 2º semestre 2018-2019

Miguel Oliveira e Silva, Artur Pereira, DETI, Universidade de Aveiro

Conteúdo

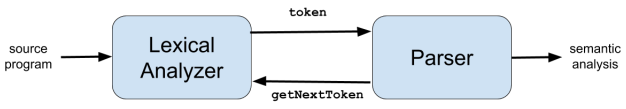
1 Análise Lexical: Estrutura de um Compilador

- 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 carácteres da entrada numa sequência de *tokens* a serem processados pelo *parser*.

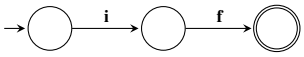


- No entanto, o analisador léxico efectua outras operações como sejam: a exclusão de espaços em branco e de comentários do *parser*, e a correlação entre erros (léxicos e sintácticos) com o código fonte (e.g. número da linha).

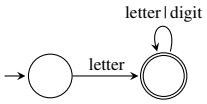
- A análise léxica pode ser feita recorrendo a gramáticas do tipo-3, ou seja, por *gramáticas regulares*, e a sua implementação computacional pode ser feita eficientemente recorrendo a *autómatos finitos*.

Análise Lexical: gramáticas regulares

- Uma *gramática é regular*, sse existir um *autómato finito* que a reconhece.
- Um autómato finito é uma *máquina de estados* com um estado inicial e um ou mais estados de aceitação (reconhecimento).
- As transições são feitas apenas tendo em conta o estado actual e a entrada.
- Um autómato finito para reconhecer a palavra reservada `if` será:



- Outro autómato finito que reconheça um identificador pode ser:



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 $(L_1)^*$ (fecho de Kleene) 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 $A = \{a_1, a_2, \ldots, a_n\}$ o alfabeto da linguagem L
  2. Então as linguagens $L_1 = \{a_1\}$, $L_2 = \{a_2\}$, $\ldots$, $L_n = \{a_n\}$ são LR (regra 2)
  3. Igualmente a linguagem $L_{\text{any}} = L_1 \cup L_2 \cup \cdots \cup L_n$ 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 1

- Uma vez que uma linguagem regular é uma linguagem reconhecida por um autómato finito é fácil, também por aí, chegarmos à mesma conclusão.
- Basta para tal considerar uma máquina de estados que implemente todas as transições das palavras da linguagem.
- Como o número de transições é finito (porque o número de palavras da linguagem também o é), então é sempre possível criar esse autómato finito.

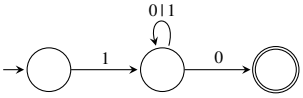
- Mantendo esta perspectiva, podemos ir mais longe e afirmar que numa linguagem finita, os autómatos que a reconhecem não têm ciclos.
- Da mesma forma, a existência de pelo menos um ciclo no autómato finito, implica necessariamente uma linguagem infinita[1]

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)

- Um autómato finito que reconhece esta linguagem pode ser:



- (Nota: A simples existência deste autómato também mostra a regularidade desta linguagem).

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 $\alpha \to \beta$ 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 $(\alpha \to \beta \in P)$ as duas condições seguintes são satisfeitas:

  $\alpha \in N$

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

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

  $\alpha \in N$

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

[1] Esta constatação está de trás de um teorema muito importante para demostrar a não-regularidade de linguagens (*regular grammar pumping lemma*).

- Para linguagem: $\{\mathbf{1}\}\cdot T^*\cdot\{\mathbf{0}\}$ $(T=\{\mathbf{0},\mathbf{1}\})$, temos uma gramática regular à direita: $G=(T,\{S,X,Y\},S,P)$

$$\begin{aligned} S &\to \mathbf{1}X \\ X &\to Y \mid \mathbf{1}X \mid \mathbf{0}X \\ Y &\to \mathbf{0} \end{aligned}$$

- E no caso de uma gramática regular à esquerda:

$$\begin{aligned} S &\to X\,\mathbf{0} \\ X &\to Y \mid X\,\mathbf{1} \mid X\,\mathbf{0} \\ Y &\to \mathbf{1} \end{aligned}$$

- 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:

$$\begin{aligned} T &= T_1\cup T_2 \\ N &= N_1\cup N_2\cup\{S\} \quad \text{com} \quad S\notin(N_1\cup N_2) \\ S &= S \\ P &= \{S\to S_1, S\to S_2\}\cup P_1\cup P_2 \end{aligned}$$

  é regular e gera a linguagem $L=L(G_1)\cup L(G_2)$
- A nova produção $S\to S_i$, com $i=1,2$, permite que G gere a linguagem $L(G_i)$

Operações sobre gramáticas regulares: reunião (exemplo)

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

$$L=L_1\cup L_2$$

  sabendo que:

$$\begin{aligned} L_1 &= \{\mathrm{a}\,w : w\in T^*\} \\ L_2 &= \{w\,\mathrm{a} : w\in T^*\} \end{aligned}$$

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

$$\begin{array}{lcl} S_1 \to \mathrm{a}X_1 & \qquad & S_2 \to \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \\ X_1 \to \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid \varepsilon & & \end{array}$$

- Teremos assim como resultado a gramática:

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to \mathrm{a}X_1 \\ X_1 &\to \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid \varepsilon \\ S_2 &\to \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \end{aligned}$$

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:

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

  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 $T = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cdot L_2$$

  sabendo que:

$$\begin{aligned} L_1 &= \{\mathrm{a}\, w : w \in T^*\} \\ L_2 &= \{w\, \mathrm{a} : w \in T^*\} \end{aligned}$$

- Recuperando as GR que representam L_1 e L_2:

$$\begin{array}{lcl} S_1 & \to & \mathrm{a}X_1 \\ X_1 & \to & \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid \varepsilon \end{array} \qquad \begin{array}{lcl} S_2 & \to & \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \end{array}$$

- Teremos assim como resultado a gramática:

$$\begin{array}{lcl} S_1 & \to & \mathrm{a}X_1 \\ X_1 & \to & \mathrm{a}X_1 \mid \mathrm{b}X_1 \mid \mathrm{c}X_1 \mid S_2 \\ S_2 & \to & \mathrm{a}S_2 \mid \mathrm{b}S_2 \mid \mathrm{c}S_2 \mid \mathrm{a} \end{array}$$

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:

$$\begin{aligned} T &= T_1 \\ N &= N_1 \cup \{S\} \,\mathrm{com}\, S \notin N_1 \\ S &= S_1 \mid \varepsilon \\ P &= \{S \to S_1 \mid \varepsilon\} \cup \\ &\quad \{A \to wS : (A \to w) \in P_1 \wedge w \in T_1^*\} \cup \\ &\quad \{A \to w : (A \to w) \in P_1 \wedge w \in T_1^* N_1\} \end{aligned}$$

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 $(S \to S_1 \mid \varepsilon)$ 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 $T = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$, determine uma gramática regular que represente a linguagem

$$L = L_1^*$$

sabendo que:

$$L_1 \quad = \quad \{\mathrm{a}\, w : w \in T^*\}$$

- Recuperando a GR que representa L_1:

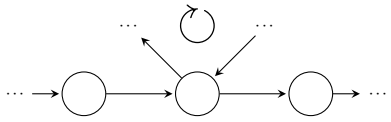
$$\begin{array}{lcl} S_1 & \to & \mathrm{a} X_1 \\ X_1 & \to & \mathrm{a} X_1 \mid \mathrm{b} X_1 \mid \mathrm{c} X_1 \mid \varepsilon \end{array}$$

- Teremos assim como resultado a gramática:

$$\begin{array}{lcl} S & \to & S_1 \mid \varepsilon \\ S_1 & \to & \mathrm{a} X_1 \\ X_1 & \to & \mathrm{a} X_1 \mid \mathrm{b} X_1 \mid \mathrm{c} X_1 \mid S \end{array}$$

3.2 Teorema da repetição para linguagens regulares

- Das várias definições equivalentes para linguagens e gramáticas regulares, talvez a mais fácil de compreender seja a de que uma linguagem será regular sse existir um autómato finito que a reconheça.
- Com esta definição é podemos com mais facilidade compreender que uma linguagem finita não só é sempre regular, como também o respectivo autómato não pode ter ciclos.
- Em sentido inverso, podemos também concluir que uma linguagem regular infinita tem de ter pelo menos um ciclo.
- Desta constatação podemos inferir que nas linguagens regulares infinitas, tem de ser sempre possível injectar (repetir) uma determinada sub-palavra as vezes que se quiser, mantendo a palavra resultante dentro da linguagem.
- Visualmente, considerando um autómato finito para este tipo de linguagens, essa palavra será a que resulta do ciclo que necessariamente tem de existir (em linguagens infinitas).



- Uma vez que o ciclo pode requerer palavras com uma dimensão mínima (suficiente para fechar o ciclo), esta condição (necessária mas não suficiente) requer um tamanho mínimo para palavras da linguagem.
- Por outro lado, nos autómatos finitos (para linguagens infinitas) podemos sempre identificar um primeiro ciclo, i.e. um ciclo que aparece a uma distância finita do inicio da palavra.

Teorema da repetição (*pumping lemma*) para linguagens regulares:: Numa linguagem regular infinita L, existe um tamanho p tal que para todas as palavras $w \in L$ para as quais: $|w| \geq p$, podemos particionar w em três palavras: $w = xyz$, tal que:
- $|y| > 0$ (padrão de repetição não vazio)
- $|xy| \leq p$ (primeiro ciclo a uma distância finita do início da palavra)
- $\forall k \geq 0 : xy^k z \in L$ (*pumping*)

- Note que esta é uma condição necessária para uma linguagem regular infinita, mas não suficiente.
- Isto é, podem existir linguagens não regulares onde esta condição se verifique.
- No entanto, a inexistência desta condição numa linguagem infinita é suficiente para determinar a sua não-regularidade.

- Temos assim que este teorema pode ser aplicado para *demostrar a não regularidade de linguagens*.
- Isto é, assumimos a regularidade da linguagem e vamos tentar demonstrar o oposto por contradição.
- Para demostrar a regularidade de uma linguagem podemos aplicar as condições que definem linguagens e gramáticas regulares (ver definição de linguagem regular na página 3).

Teorema da repetição para linguagens regulares: exemplo 1

- Determine a regularidade da linguagem $L_1 = \{a^i b^i : i \geq 0\}$.
- Em primeiro lugar note que não é possível aplicar a condição da concatenação, porque há uma dependência no número de repetições dos símbolos a e b. Isto é, muito embora quer a^i, quer b^i, $(i \geq 0)$ sejam regulares, o número de repetições i tem de ser o mesmo (condição não garantida pela concatenação).
- Dito de outra forma: $a^i b^i \neq a^* b^*$.
- Uma vez que a linguagem é infinita então, para ser regular, o teorema da repetição tem ser aplicável.
- Seja $w = a^n b^n$. Então $|w| = 2n$.
- Para particionar $w = xyz$ temos três possibilidades: y contém só sequências de a's, só sequências de b's, ou uma sequência com prefixos de a's e sufixos de b's.

- Os dois últimos casos quebram desde logo a condição $|xy| \leq p$ já que o x terá de conter a's e existem tantas palavras em L_1 quantas se queira com um número não limitado de a's em x.
- Resta o primeiro caso.
- Seja $x = a^p, y = a^q, z = a^r b^n$, então $p+q+r = n$ para que $w \in L_1$ e $q \neq 0$.
- Como $xy^2z \notin L_1$ (uma vez que $q+2*q+r \neq n$), então L_1 não é regular.

Teorema da repetição para linguagens regulares: exemplo 2

- Determine a regularidade da linguagem $L_2 = \{c^k a^i b^i : k \geq 0 \wedge i \geq 0\}$.
- Repare que neste caso o teorema da repetição aplica-se ($y = c$). No entanto seria estranho que esta linguagem fosse regular uma vez que contém a linguagem anterior (que não era regular).
- Não há aqui nenhuma contradição porque o teorema da repetido é uma condição necessária, mas não suficiente.
- Para determinar a regularidade neste caso temos de aplicar as operações aplicáveis a linguagens regulares.
- Assim: $L_2 = \{c^k : k \geq 0\} \cdot L_1$.
- A linguagem $\{c^k : k \geq 0\}$ é regular, mas L_1 não o é, pelo que L_2 também não vai ser.

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 $\{\text{a}\}$.
  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_1e_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 $A = \{\mathrm{a},\mathrm{b},\mathrm{c}\}$ que satisfazem o requisito de qualquer símbolo b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.
R: $(\mathrm{a}\,|\,\mathrm{abc}\,|\,\mathrm{c})^*$

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: $r^* = \varepsilon \mid r \mid rr \mid \ldots$

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

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

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

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

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 $\cdot$)
  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 $A = \{\mathrm{a},\mathrm{b},\mathrm{c}\}$ que satisfazem o requisito de qualquer símbolo b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.
R: $(\mathrm{a}\,|\,\mathrm{abc}\,|\,\mathrm{c})^* \Leftrightarrow ((\mathrm{a}\,|\,((\mathrm{ab})\mathrm{c}))\,|\,\mathrm{c})^*$

P: Determine uma ER que represente as sequências binárias com um numero 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 $A = \{\mathrm{a},\mathrm{b},\cdots,\mathrm{z}\}$ construa uma ER para a linguagem:

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

R: $(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*\mathrm{a}\,(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*\mathrm{a}\,(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*\mathrm{a}\,(\mathrm{b}\,|\,\mathrm{c}\,|\,\cdots\,|\,\mathrm{z})^*$

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:
  $e? = (e\,|\,\varepsilon)$
- Um símbolo dum sub-alfabeto:
  $[\mathrm{a}_1\mathrm{a}_2\ldots\mathrm{a}_n] = \mathrm{a}_1\,|\,\mathrm{a}_2\,|\ldots|\,\mathrm{a}_n$
  $[\mathrm{a}_1-\mathrm{a}_n] = \mathrm{a}_1\,|\,\mathrm{a}_2\,|\ldots|\,\mathrm{a}_n$
- Um símbolo fora dum sub-alfabeto:
  $[\hat{}\,\mathrm{a}_1\mathrm{a}_2\ldots\mathrm{a}_n]$ ou (`ANTLR`): $\sim[\mathrm{a}_1\mathrm{a}_2\ldots\mathrm{a}_n]$
  $[\hat{}\,\mathrm{a}_1-\mathrm{a}_n]$ ou (`ANTLR`): $\sim[\mathrm{a}_1-\mathrm{a}_n]$

– n ocorrências:
$e\{n\} = \underbrace{e{\cdot}e{\cdot}\ldots{\cdot}e}_{n}$
– de n_1 a n_2 ocorrências:
$e\{n1,n2\} = \underbrace{e{\cdot}e{\cdot}\ldots{\cdot}e}_{n_1,n_2}$

– n ou mais ocorrências:
$e\{n,\} = \underbrace{e{\cdot}e{\cdot}\ldots{\cdot}e}_{n,}$

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 $A = \{\mathrm{a},\mathrm{b},\cdots,\mathrm{z}\}$ construa uma ER para a linguagem:

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

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

Outras extensões notacionais

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

Símbolo:	Significado:
`.`	um símbolo qualquer diferente de `\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. independente 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:
  $S \to \varepsilon$
- A GR para a ER a, qualquer que seja o a, é dada por:
  $S \to \mathrm{a}$
- Vamos exemplificar com a ER $e = (\mathrm{a}\,|\,\mathrm{b})^*\mathrm{a}$

1. Primeiro definimos regras para os símbolos terminais:

   $S_1 \to \mathrm{a}$

   $S_2 \to \mathrm{b}$
2. Para reconhecer $(\mathrm{a}\,|\,\mathrm{b})$ temos $(S = S_3)$:

   $S_3 \to S_1 \,|\, S_2$

   $S_1 \to \mathrm{a}$

   $S_2 \to \mathrm{b}$
3. Para reconhecer $(\mathrm{a}\,|\,\mathrm{b})^*$ temos $(S = S_3)$:

   $S_3 \to S_1 \,|\, S_2 \,|\, \varepsilon$

   $S_1 \to S_3\,\mathrm{a}$

   $S_2 \to S_3\,\mathrm{b}$
4. Por fim para reconhecer a ER $(\mathrm{a}\,|\,\mathrm{b})^*\mathrm{a}$ temos $(S = S_4)$:

   $S_4 \to S_3\,\mathrm{a}$

   $S_3 \to S_1 \,|\, S_2 \,|\, \varepsilon$

   $S_1 \to S_3\,\mathrm{a}$

   $S_2 \to S_3\,\mathrm{b}$

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:

     $$\begin{aligned}\mathscr{E} \;=\; & \{E, \varepsilon, S\} \cup \\ & \{(A, w, B) : (A \to wB) \in P\} \cup \\ & \{(A, w, \varepsilon) : (A \to w) \in P\}\end{aligned}$$

     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 $\mathrm{w}_1 = \beta_1 \,|\, \beta_2 \,|\, \cdots \,|\, \beta_n$
  (B) Substituir todos os triplos da forma (B, α_i, B) por um único (B, w_2, B), onde $\mathrm{w}_2 = \alpha_1 \,|\, \alpha_2 \,|\, \cdots \,|\, \alpha_m$
  (C) Substituir todos os triplos da forma (B, γ_i, C) por um único (B, w_3, C), onde $\mathrm{w}_3 = \gamma_1 \,|\, \gamma_2 \,|\, \cdots \,|\, \gamma_k$

(D) Substituir o triplo de triplos $((A, \mathrm{w}_1, B), (B, \mathrm{w}_2, B), (B, \mathrm{w}_3, C))$ pelo triplo $(A, \mathrm{w}_1\mathrm{w}_2^*\mathrm{w}_3, C)$

- Vamos exemplificar com a seguinte GR:

$$
\begin{aligned}
S &\rightarrow aS \mid bS \mid cS \mid abaX \\
X &\rightarrow aX \mid bX \mid cX \mid \varepsilon \\
\mathscr{E} &= \{(E, \varepsilon, S), (S, \mathrm{a}, S), (S, \mathrm{b}, S), (S, \mathrm{c}, S), (S, \mathrm{aba}, X), \\
&\quad (X, \mathrm{a}, X), (X, \mathrm{b}, X), (X, \mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(B)} &= \{(E, \varepsilon, S), (S, \mathrm{a}\mid\mathrm{b}\mid\mathrm{c}, S), (S, \mathrm{aba}, X), \\
&\quad (X, \mathrm{a}, X), (X, \mathrm{b}, X), (X, \mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(D)} &= \{(E, (\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*\mathrm{aba}, X), \\
&\quad (X, \mathrm{a}, X), (X, \mathrm{b}, X), (X, \mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(B)} &= \{(E, (\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*\mathrm{aba}, X), (X, \mathrm{a}\mid\mathrm{b}\mid\mathrm{c}, X), (X, \varepsilon, \varepsilon)\} \\
\text{(D)} &= \{(E, (\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*\mathrm{aba}(\mathrm{a}\mid\mathrm{b}\mid\mathrm{c})^*, \varepsilon)\}
\end{aligned}
$$

6 Reconhecimento de *tokens*

- Anteriormente vimos como se podem expressar padrões utilizando *expressões regulares*.
- Assim sendo, vamos definir as expressões regulares para os *tokens* do seguinte excerto duma linguagem: [2]

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

- 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:

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

[2] Exemplo 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:

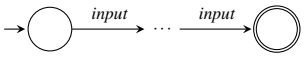
$$\textbf{ws} \quad \rightarrow \quad (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

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

tokens	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 colecção 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.

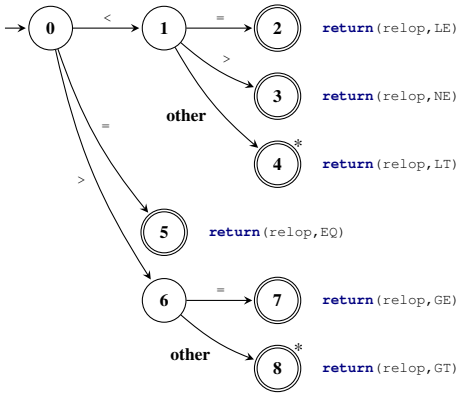


- As convenções a aplicar a estes diagramas são as seguintes:

  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.

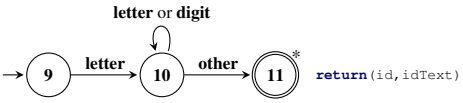
- Na construção destes diagramas vamos simplesmente enumerar novos estados por cada transição resultante de um carácter que, de alguma forma, avance no reconhecimento do *token*.

- O diagrama de transição para reconhecer os operadores relacionais pode ser o seguinte:



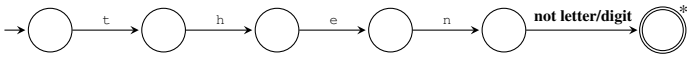
- Note que, para este tipo de *tokens*, este diagrama é uma estrutura de dados tipo árvore (deitada). Isso acontece em *tokens* fixos como o exemplificado ou as palavras reservadas.

- O diagrama de transição para identificadores:

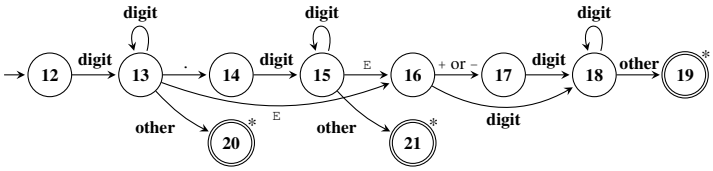


- O reconhecimento de identificadores pode levantar um problema de ambiguidade.
- De facto, as palavras reservadas da linguagem (ex: **then**) também podem ser reconhecidas como identificadores.
- Para resolver este problema, os analisadores léxicos dão prioridade a *tokens* que consomem mais caracteres 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.

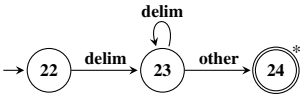
- O diagrama de transição para palavras reservadas é aqui exemplificado com o *token* **then**:



- 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 $\{\mathbf{0},\mathbf{1},\mathbf{6}\}$, é invocada a função `nextChar`; assim como os estados com asterisco, o carácter a mais é reposto 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 chamas *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;
```

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

7 Autómatos finitos

- Um autómato é uma "máquina" que executa sobre uma determinada sequência de entradas passo a passo (de forma discreta no tempo).
- Internamente, o autómato contém uma máquina de estados, que vai evoluindo até uma de duas possibilidades: aceitar ou rejeitar a entrada.
- A palavra de entrada será reconhecida se o estado final for de aceitação.
- Um autómato 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 (estados de aceitação terão um círculo duplo).

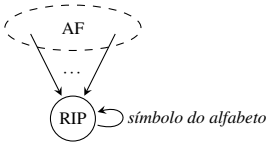
- Os autómatos finitos são classificados em dois tipos:

  a) *Autómato 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) *Autómato finito determinista* (AFD): cada estado indica no máximo uma transição com cada símbolo do alfabeto.

- Qualquer um destes tipos de autómatos reconhece as mesmas linguagens, e demonstra-se que essas linguagens correspondem às linguagens regulares.[3]
- Note que um AFD é um caso particular de um AFND.
- Os autómatos finitos podem ser *completos* ou *incompletos*.

- Será *incompleto* caso não existam transições para todos os símbolos do alfabeto, e *completo* no caso contrário.
- Neste caso, o aparecimento desse símbolo com o autómato nesse estado, leva imediatamente à rejeição da palavra.
- Podemos sempre converter um autómato incompleto num completo destinando todas as transições não expressas para um novo estado que funcione como uma especie de "buraco negro". Isto é, um estado do qual não se pode sair. Obviamente, esse estado não pode ser de aceitação.



8 Autómato finito não determinista

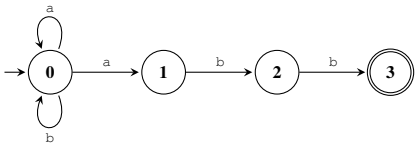
- Um autómato finito não determinista é um autómato finito onde:
  - as transições estão associadas a símbolos individuais do alfabeto ou à palavra vazia (ε);
  - de cada estado *saem zero ou mais transições* por cada símbolo do alfabeto ou ε;

[3] Com uma excepção menor: as linguagens regulares não expressam a linguagem vazia ($\emptyset$), o que pode ser trivialmente feito com autómatos finitos.

  - 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

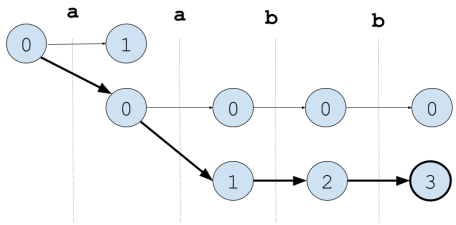
- Um possível diagrama de transição para um AFND que reconhece a expressão regular $(\mathbf{a}\,|\,\mathbf{b})^*\mathbf{abb}$ é o seguinte:



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

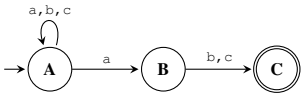
AFND: caminhos alternativos

- Será que a palavra **aabb** pertence esta linguagem?
- Existem 3 caminhos alternativos no diagrama:
  1. $0 \xrightarrow{\mathbf{a}} 1 \xrightarrow{\mathbf{a}} ?$
  2. $0 \xrightarrow{\mathbf{a}} 0 \xrightarrow{\mathbf{a}} 0 \xrightarrow{\mathbf{b}} 0 \xrightarrow{\mathbf{b}} 0$
  3. $0 \xrightarrow{\mathbf{a}} 0 \xrightarrow{\mathbf{a}} 1 \xrightarrow{\mathbf{b}} 2 \xrightarrow{\mathbf{b}} 3$
- Apenas o último termina no estado final.
- Podemos representar estes caminhos com uma estrutura tipo árvore (deitada):



AFND: exemplo

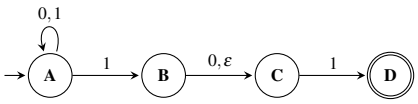
- Considerando o alfabeto $A = \{a, b, c\}$, que palavras são reconhecidas pelo autómato seguinte:



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

AFND: exemplo com transições ε

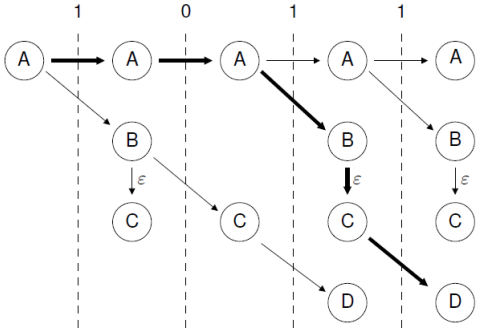
- Considere o seguinte AFND sobre o alfabeto $A = \{0, 1\}$:



- Será que a palavra **1011** é reconhecida?
- Há 6 caminhos possíveis:
  1. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{1}} A$
  2. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{1}} B$
  3. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{1}} B \xrightarrow{\varepsilon} C$
  4. $A \xrightarrow{\mathbf{1}} A \xrightarrow{\mathbf{0}} A \xrightarrow{\mathbf{1}} B \xrightarrow{\varepsilon} C \xrightarrow{\mathbf{1}} D$
  5. $A \xrightarrow{\mathbf{1}} B \xrightarrow{\mathbf{0}} C \xrightarrow{\mathbf{1}} D$
  6. $A \xrightarrow{\mathbf{1}} B \xrightarrow{\varepsilon} C$

AFND: exemplo com transições ε

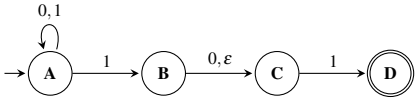
- Com a estrutura tipo árvore:



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

AFND: exemplo

- Que palavras são reconhecidas por este autómato?



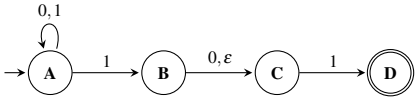
- Todas as palavras terminadas em **11** ou **101**.
- Como expressão regular: $(\mathbf{0}\,|\,\mathbf{1})^*\mathbf{10}?\mathbf{1}$

AFND: definição formal

- Um automato finito não determinista é um quíntuplo $M = (A, Q, q_0, \delta, F)$, em que[4]:
  - 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;
  - $\delta \subseteq (Q \times A_\varepsilon \times Q)$ é a relação de transição entre estados, com $A_\varepsilon = A \cup \{\varepsilon\}$;
  - $F \subseteq Q$ é o conjunto dos estados de aceitação.

AFND: outro exemplo

- Represente analiticamente o AFND:



[4] δ é a letra grega minúscula delta.

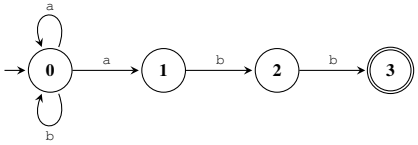
- O quíntuplo $M = (A, Q, q_0, \delta, F)$ é:
  - $A = \{\mathbf{0}, \mathbf{1}\}$
  - $Q = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$
  - $q_0 = \mathbf{A}$
  - $F = \{\mathbf{D}\}$
  - $\delta = \{(\mathbf{A}, \mathbf{0}, \mathbf{A}), (\mathbf{A}, \mathbf{1}, \mathbf{A}), (\mathbf{A}, \mathbf{1},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

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 $u = u_1 u_2 \cdots u_n$, com $u_i \in A_\varepsilon$, e existir uma sequência de estados $s_0, s_1, \cdots, s_n$, que satisfaça as seguintes condições:
  1. $s_0 = q_0$
  2. qualquer que seja o $i = 1, \cdots, n$, $(s_{i-1}, u_i, s_i) \in \delta$
  3. $s_n \in F$
- 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 $q_i \xrightarrow{\mathbf{u}} q_j$ para representar a existência de uma palavra u que conduza do estado q_i ao estado q_j
- Usando esta notação tem-se $L(M) = \{u : q_0 \xrightarrow{\mathbf{u}} q_f \ \wedge \ q_f \in F\}$

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:

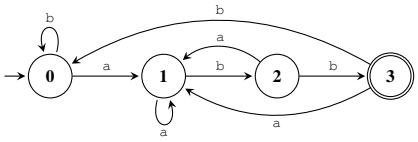


é a seguinte:

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

9 Autómato finito determinista

- Um autómato 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 $(\mathbf{a} \,|\, \mathbf{b})^* \mathbf{abb}$ é o seguinte:



Autómato finito determinista: tabela de transição

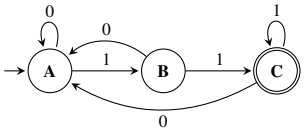
- Se estivermos a representar um AFD com uma tabela de transição, então cada entrada será um único estado (pelo que deixa de ser necessário representá-lo como conjunto):

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

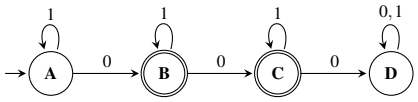
- Enquanto um AFND é uma representação abstracta dum algoritmo para reconhecer expressões regulares, o AFD é um algoritmo simples, concreto e muito eficiente para o mesmo fim.
- Felizmente é possível converter um AFND num AFD que reconhece a mesmo linguagem regular.

AFD: exemplo

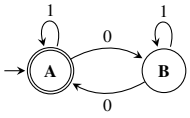
- Que palavras são reconhecidas pelo autómato seguinte:



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



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



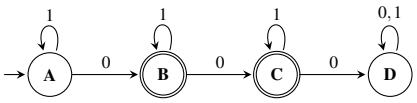
- Todas as palavras com um número par de zeros.
- Como expressão regular: $\mathbf{1}^*(\mathbf{01}^*\mathbf{0})^*\mathbf{1}^*$

AFD: definição formal

- Um automato finito determinista é um quíntuplo $M = (A, Q, q_0, \delta, F)$, 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 (agora é uma função).
- 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 quíntuplo $M = (A, Q, q_0, \delta, F)$ é:
  - $A = \{\mathbf{0}, \mathbf{1}\}$
  - $Q = \{\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}\}$
  - $q_0 = \mathbf{A}$
  - $F = \{\mathbf{B}, \mathbf{C}\}$

$\delta = \{(\mathbf{A},\mathbf{0},\mathbf{B}), (\mathbf{A},\mathbf{1},\mathbf{A}), (\mathbf{B},\mathbf{0},\mathbf{C}), (\mathbf{B},\mathbf{1},\mathbf{B}), (\mathbf{C},\mathbf{0},\mathbf{D}), (\mathbf{C},\mathbf{1},\mathbf{C}), (\mathbf{D},\mathbf{0},\mathbf{D}), (\mathbf{D},\mathbf{1},\mathbf{D})\}$

STATE	0	1
$\rightarrow A$	B	A
B_f	C	B
C_f	D	C
D	D	D

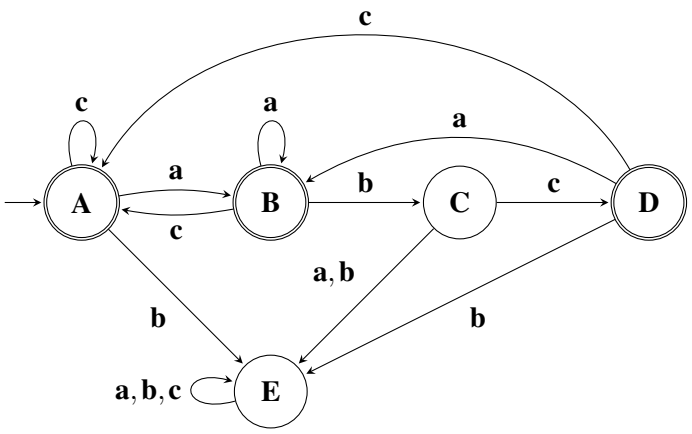
AFD: linguagem reconhecida

- 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 $u = u_1 u_2 \cdots u_n$ e existir uma sequência de estados $s_0, s_1, \cdots, s_n$, que satisfaça as seguintes condições:
  1. $s_0 = q_0$
  2. qualquer que seja o $i = 1, \cdots, n$, $s_i = \delta(s_{i-1}, u_i)$
  3. $s_n \in F$
- Caso contrario diz-se que M *rejeita* a entrada.

10 Autómato finito determinista

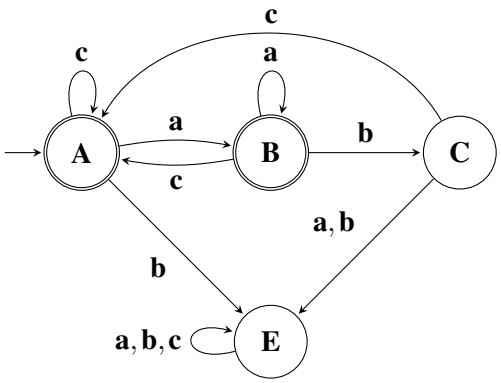
10.1 Projecto de autómato finito determinista

- Projete um AFD que reconheça as sequências definidas sobre o alfabeto $\mathbf{A} = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ 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 requirido 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 autómatos finitos deterministas

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

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

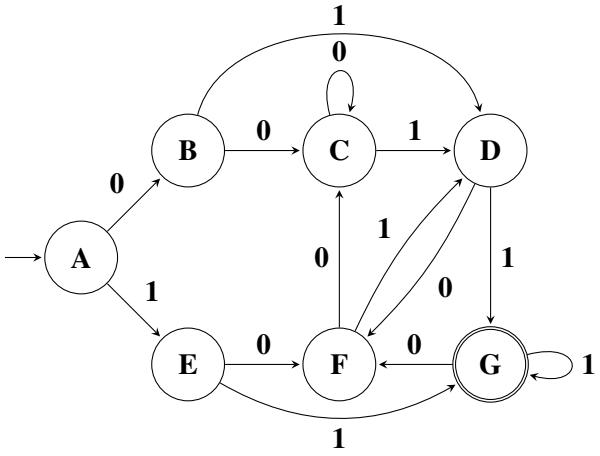
- Em que o fecho da função de transição $\delta^* : Q \times A \to Q$ é definido por:

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

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

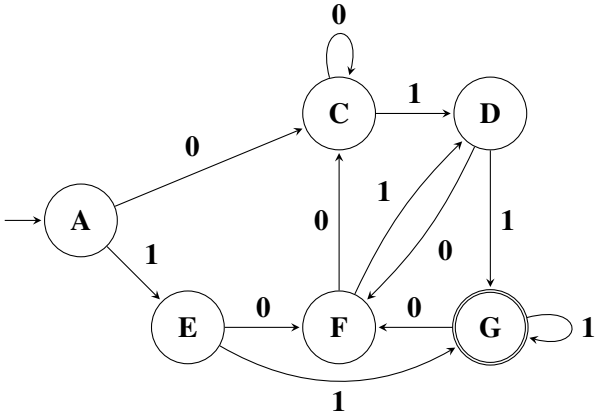
$$L(M) = \{u \in A^* \mid M \text{aceita} u\} = \{u \in A^* \mid \delta^*(q_0,u) \in F\}$$

- Considere o seguinte autómato:



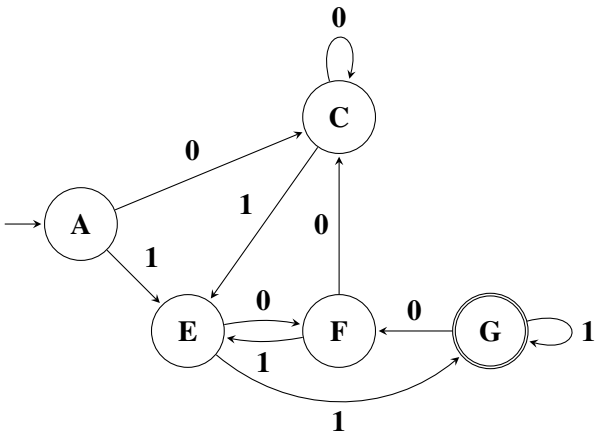
STATE	**0**	**1**
$\to A$	B	E
B	C	D
C	C	D
D	F	G
E	F	G
F	C	D
G_f	F	G

- Vamos primeiro aplicar a regra 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:



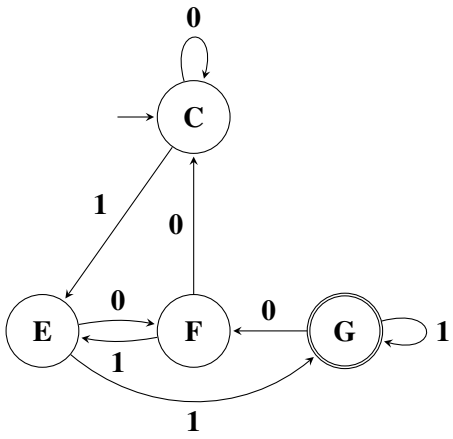
STATE	**0**	**1**
$\to A$	C	E
C	C	D
D	F	G
E	F	G
F	C	D
G_f	F	G

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



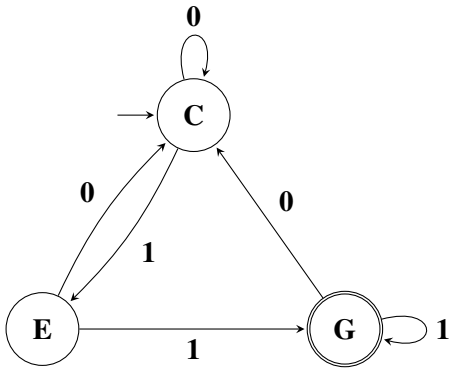
STATE	**0**	**1**
$\to A$	C	E
C	C	E
E	F	G
F	C	E
G_f	F	G

- 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.
- No entanto, este método para simplificar AFDs não garante um 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;[5]

  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.

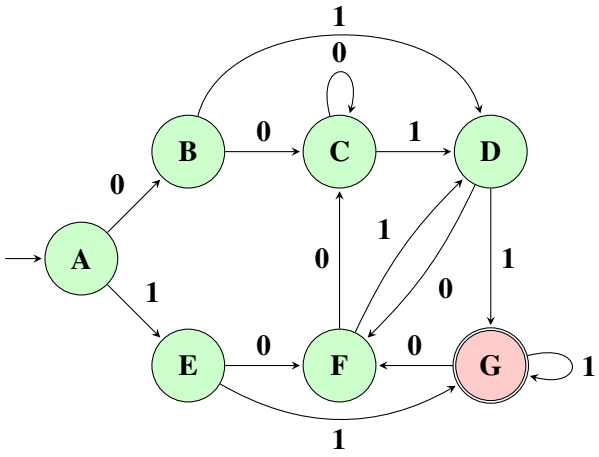
  3. O passo anterior é repetido até que não sejam possíveis mais partições. Nessa situação, o AFD está minimizado.

- Recuperando o exemplo anterior, temos como conjuntos de partida:

$$C_1 = Q - F = \{A,B,C,D,E,F\}$$
$$C_2 = F = \{G\}$$

[5] Se o AFD for incompleto, então primeiro tem de ser transformado num completo.

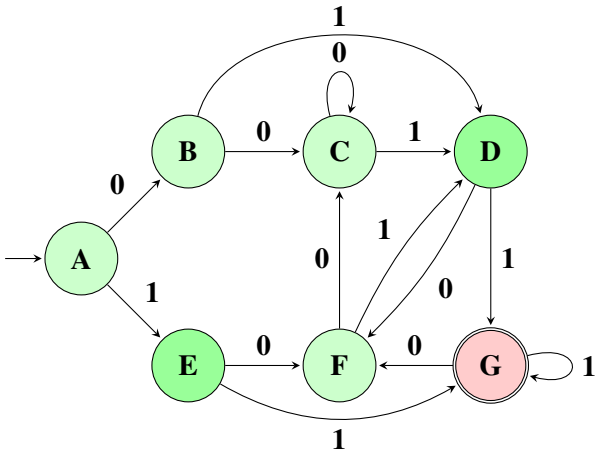
C_1 C_2



SET/STATE	0	1
$\rightarrow C_1/A$	C_1	C_1
C_1/B	C_1	C_1
C_1/C	C_1	C_1
C_1/D	C_1	C_2
C_1/E	C_1	C_2
C_1/F	C_1	C_1
C_2/G_f	C_1	C_2

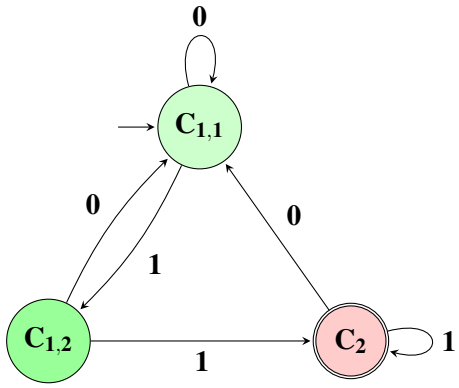
- O conjunto C_1 tem de ser partido em dois, já que para a entrada **1** os estados **D** e **E** têm uma transição para um conjunto diferente (C_2) do que os restantes estados.

$C_{1,1}$ $C_{1,2}$ C_2



SET/STATE	0	1
$\rightarrow C_{1,1}/A$	$C_{1,1}$	$C_{1,2}$
$C_{1,1}/B$	$C_{1,1}$	$C_{1,2}$
$C_{1,1}/C$	$C_{1,1}$	$C_{1,2}$
$C_{1,1}/F$	$C_{1,1}$	$C_{1,2}$
$C_{1,2}/D$	$C_{1,1}$	C_2
$C_{1,2}/E$	$C_{1,1}$	C_2
C_2/G_f	$C_{1,1}$	C_2

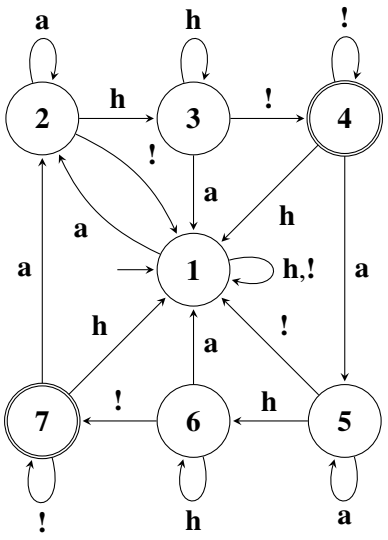
- Chegamos assim a um AFD minimizado equivalente ao que já tínhamos chegado:



STATE	0	1
$\rightarrow C_{1,1}$	$C_{1,1}$	$C_{1,2}$
$C_{1,2}$	$C_{1,1}$	C_2
$C_{2,f}$	$C_{1,1}$	C_2

Redução de AFD: exemplo 2

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



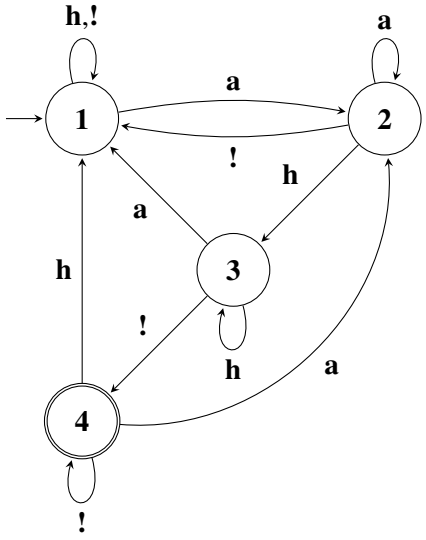
STATE	a	h	!
$\rightarrow 1$	2	1	1
2	2	3	1
3	1	3	4
4_f	5	1	4
5	5	6	1
6	1	6	7
7_f	2	1	7

STATE		a	h	!
$\rightarrow 1$	C_1	C_1	C_1	C_1
2	C_1	C_1	C_1	C_1
3	C_1	C_1	C_1	C_2
5	C_1	C_1	C_1	C_1
6	C_1	C_1	C_1	C_2
4_f	C_2	C_1	C_1	C_2
7_f	C_2	C_1	C_1	C_2

STATE		a	h	!
$\rightarrow 1$	$C_{1,1}$	$C_{1,1}$	$C_{1,1}$	$C_{1,1}$
2	$C_{1,1}$	$C_{1,1}$	$C_{1,2}$	$C_{1,1}$
5	$C_{1,1}$	$C_{1,1}$	$C_{1,2}$	$C_{1,1}$
3	$C_{1,2}$	$C_{1,1}$	$C_{1,2}$	C_2
6	$C_{1,2}$	$C_{1,1}$	$C_{1,2}$	C_2
4_f	C_2	$C_{1,1}$	$C_{1,1}$	C_2
7_f	C_2	$C_{1,1}$	$C_{1,1}$	C_2

STATE		a	h	!
$\rightarrow 1$	$C_{1,1,1}$	$C_{1,1,2}$	$C_{1,1,1}$	$C_{1,1,1}$
2	$C_{1,1,2}$	$C_{1,1,2}$	$C_{1,2}$	$C_{1,1,1}$
5	$C_{1,1,2}$	$C_{1,1,2}$	$C_{1,2}$	$C_{1,1,1}$
3	$C_{1,2}$	$C_{1,1,1}$	$C_{1,2}$	C_2
6	$C_{1,2}$	$C_{1,1,1}$	$C_{1,2}$	C_2
4_f	C_2	$C_{1,1,2}$	$C_{1,1,1}$	C_2
7_f	C_2	$C_{1,1,2}$	$C_{1,1,1}$	C_2

- Donde resulta o seguinte autómato final:



STATE	a	h	!
$\rightarrow 1$	2	1	1
2	2	3	1
3	1	3	4
4_f	2	1	4

11 Conversão de AFND em AFD

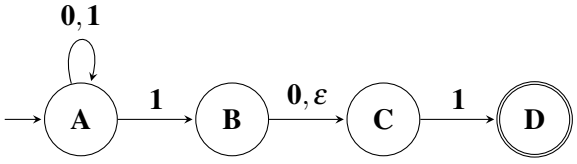
- Como já foi referido um AFD é um AFND, mas o contrário não é necessariamente verdadeiro.
- Nos AFD as transições são funções e as tabelas de transição são mais simples havendo sempre uma transição para no máximo um único estado.
- Assim, em geral, a implementação de AFD é preferível à implementação 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
ε-closure($\mathbf{s}$)	Conjunto de estados do AFND para os quais pode haver uma transição a partir do estado **s** apenas pela palavra vazia (ε).
ε-closure($\mathbf{C}$)	Conjunto de estados do AFND para os quais pode haver uma transição a partir de qualquer estado do conjunto **C** apenas pela palavra vazia.
move($\mathbf{C},\mathbf{a}$)	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 ε-closure ao estado inicial do AFND.
- Os estados finais do AFD, serão todos os estados que contiverem pelo menos um estado final do AFDN.

- 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	$\emptyset$	$\emptyset$	$\emptyset$

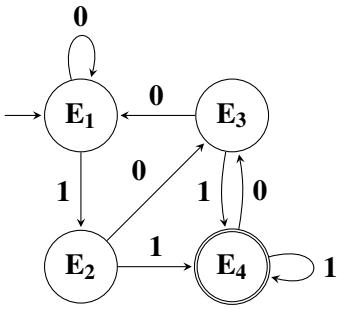
- Vamos identificar os estados do AFD por $E_i,\ i \in \mathbb{N}$
- O estado inicial para o AFD será dado por: $\varepsilon\text{-closure}(\{\mathbf{A}\}) = \{\mathbf{A}\} = E_1$

- Seguidamente vamos determinar os subconjuntos de estados AFND que resultam da transição de E_1 por cada símbolo do alfabeto.
  $\varepsilon\text{-closure}(\text{move}(E_1,\mathbf{0})) = \varepsilon\text{-closure}(\{\mathbf{A}\}) = \{\mathbf{A}\} = E_1$
- Como chegámos a um subconjunto que já existe ($\{\mathbf{A}\}$), não há lugar à criação de um novo estado para o AFD.
  $\varepsilon\text{-closure}(\text{move}(E_1,\mathbf{1})) = \varepsilon\text{-closure}(\{\mathbf{A},\mathbf{B}\}) = \{\mathbf{A},\mathbf{B},\mathbf{C}\} = E_2$
- 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).
  $\varepsilon\text{-closure}(\text{move}(E_2,\mathbf{0})) = \varepsilon\text{-closure}(\{\mathbf{A},\mathbf{C}\}) = \{\mathbf{A},\mathbf{C}\} = E_3$

ε-closure(move(E_2, **1**)) = ε-closure({**A**, **B**, **D**}) = {**A**, **B**, **C**, **D**} = E_4

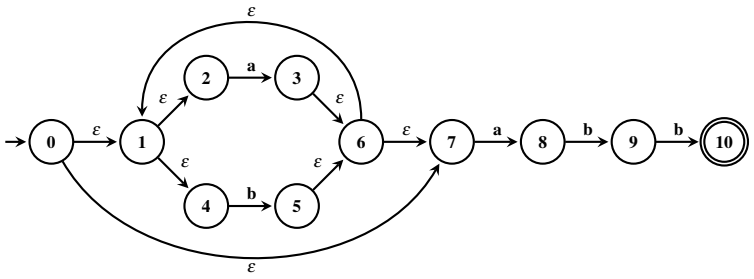
ε-closure(move(E_3, **0**)) = ε-closure({**A**}) = {**A**} = E_1
ε-closure(move(E_3, **1**)) = ε-closure({**A**, **B**, **D**}) = {**A**, **B**, **C**, **D**} = E_4
ε-closure(move(E_4, **0**)) = ε-closure({**A**, **C**}) = {**A**, **C**} = E_3
ε-closure(move(E_4, **1**)) = ε-closure({**A**, **B**, **D**}) = {**A**, **B**, **C**, **D**} = E_4

- Logo, vamos ter o seguinte AFD:



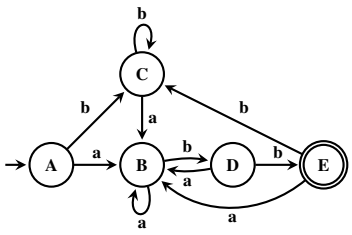
AFND	AFD	**0**	**1**
{**A**}	$\rightarrow E_1$	E_1	E_2
{**A**, **B**, **C**}	E_2	E_3	E_4
{**A**, **C**}	E_3	E_1	E_4
{**A**, **B**, **C**, **D**}	$E_{4,f}$	E_3	E_4

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



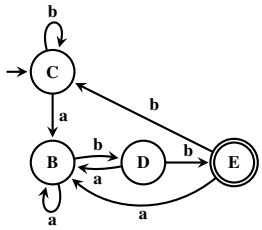
STATE	**a**	**b**	ε
$\rightarrow 0$	$\emptyset$	$\emptyset$	{1, 7}
1	$\emptyset$	$\emptyset$	{2, 4}
2	{3}	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	{6}
4	$\emptyset$	{5}	$\emptyset$
5	$\emptyset$	$\emptyset$	{6}
6	$\emptyset$	$\emptyset$	{1, 7}
7	{8}	$\emptyset$	$\emptyset$
8	$\emptyset$	{9}	$\emptyset$
9	$\emptyset$	{10}	$\emptyset$
10_f	$\emptyset$	$\emptyset$	$\emptyset$

- Estado inicial: ε-closure({**0**}) = {**0**, **1**, **2**, **4**, **7**} = A
ε-closure(move(A, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(A, **b**)) = ε-closure({**5**}) = {**1**, **2**, **4**, **5**, **6**, **7**} = C
ε-closure(move(B, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(B, **b**)) = ε-closure({**5**, **9**}) = {**1**, **2**, **4**, **5**, **6**, **7**, **9**} = D
ε-closure(move(C, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(C, **b**)) = ε-closure({**5**}) = {**1**, **2**, **4**, **5**, **6**, **7**} = C
ε-closure(move(D, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(D, **b**)) = ε-closure({**5**, **10**}) = {**1**, **2**, **4**, **5**, **6**, **7**, **10**} = E
ε-closure(move(E, **a**)) = ε-closure({**3**, **8**}) = {**1**, **2**, **3**, **4**, **6**, **7**, **8**} = B
ε-closure(move(E, **b**)) = ε-closure({**5**}) = {**1**, **2**, **4**, **5**, **6**, **7**} = C



AFND	AFD	**a**	**b**
{**0**, **1**, **2**, **4**, **7**}	$\rightarrow A$	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

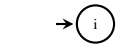
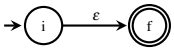
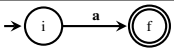
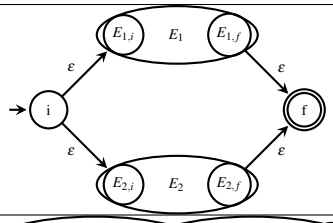
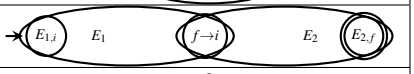
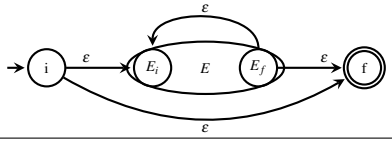
- Este AFD pode ainda ser minimizado (os estados **A** e **C** são equivalentes):



AFD	a	b
B	B	D
$\rightarrow C$	B	C
D	B	E
E_f	B	C

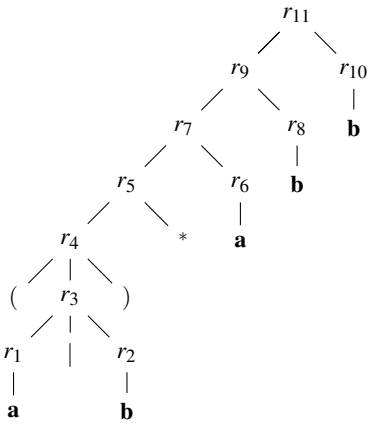
12 Conversão de uma expressão regular num AFND

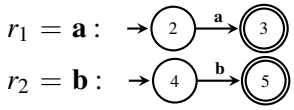
- Para compreendermos minimamente a construção de analisadores léxicos só falta abordarmos o problema da conversão automática de expressões regulares para autómatos.
- Para esse fim vamos apresentar um algoritmo (*McNaughton-Yamada-Thompson*) que converte uma qualquer `ER` num `AFND`.
- A estratégia 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`

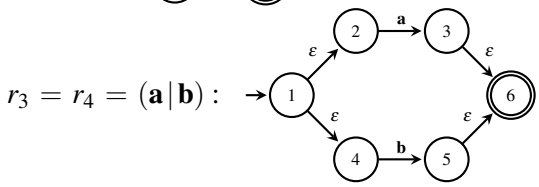
Descrição	ER	AFND
Linguagem vazia	()	
Palavra vazia	ε	
Símbolo do alfabeto	**a**	
União de AFND	$(E_1 \mid E_2)$	
Concatenação de AFND	E_1E_2	
Fecho de Kleene de AFND	E^*	

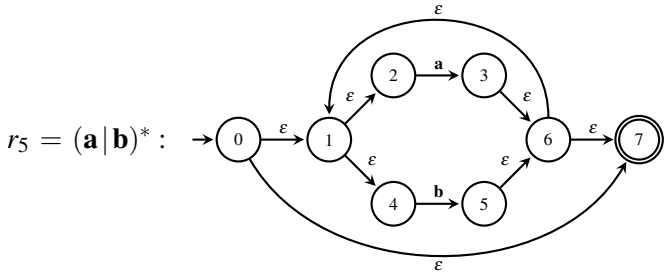
Conversão de uma ER num AFND: Exemplo

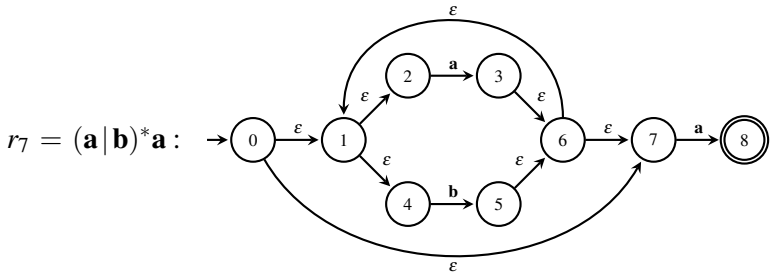
- Vamos então construir um AFND para a `ER`: $(\mathbf{a} \mid \mathbf{b})^*\mathbf{abb}$
- A árvore sintáctica desta `ER` é a seguinte:

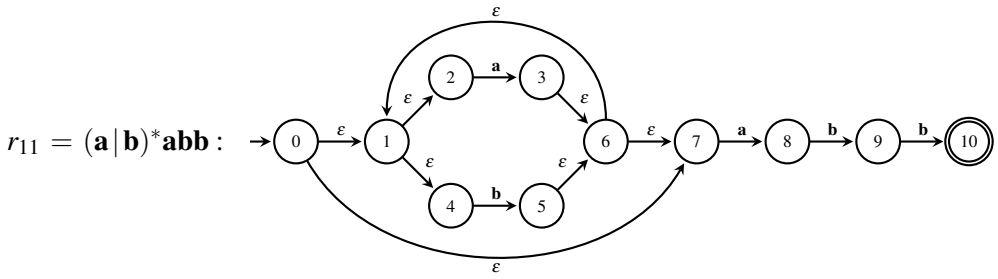








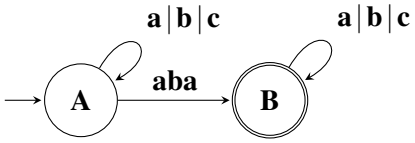




13 Autómato finito generalizado (AFG)

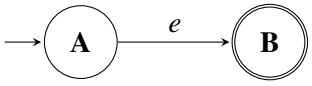
- Nos autómatos finitos apresentados as transições entre estados apenas decorrem de símbolos do alfabeto ou, no caso dos AFND, da palavra vazia (ε).
- No entanto podemos aproximar ainda mais os autómatos finitos das expressões regulares fazendo

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 *Autómato finito generalizado* (AFG).
- Por exemplo, um AFG sobre o alfabeto $A = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ para o conjunto de palavras que contém a palavra **aba** será:



13.1 AFG reduzido

- Um AFG com a forma



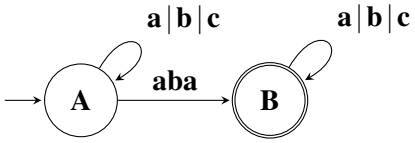
designa-se por *autómato finito generalizado reduzido*.
- Note que:
  - O estado *A* não é de aceitação e não tem arcos a chegar de outros estados.
  - O estado *B* é de aceitação e não tem arcos a sair.
- Se reduzir um AFG à forma anterior a expressão – *e* – é uma expressão regular equivalente ao autómato.

13.2 Conversão de uma AFG numa ER

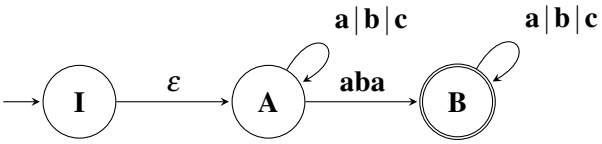
- Assim transformar uma AFG num AFG reduzido corresponde a determinar a ER que lhe é equivalente.
- Algoritmo de conversão:
  1. Transformação de um AFG noutro cujo estado inicial não tenha arcos a chegar.
     - Se necessário, acrescenta-se um novo estado inicial com um arco em ε para o antigo.
  2. Transformação de um AFG noutro 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

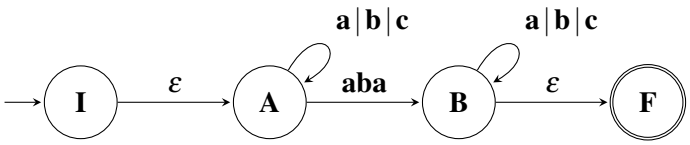
- Recuperando o AFG atrás apresentado vamos aplicar este algoritmo para o transformar numa ER.



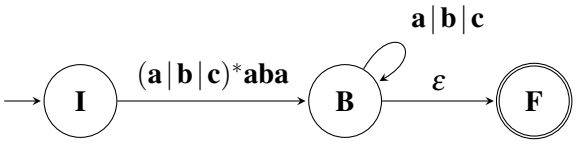
- Aplicando a regra 1 :



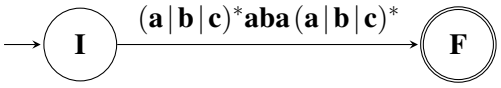
- Aplicando a regra 2 :



- Eliminando o estado *A* aplicando a regra 3 :



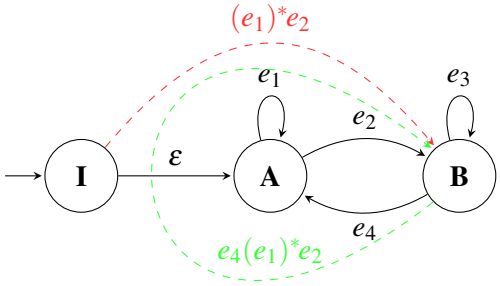
- Por fim, eliminando o estado *B* aplicando novamente a regra 3 :



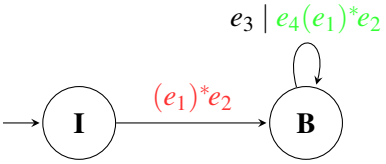
Eliminar estado com arcos a chegar de outros estados

- 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

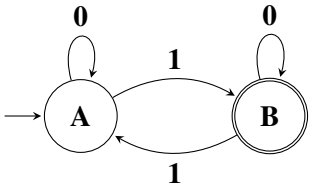


resulta no seguinte AFG:

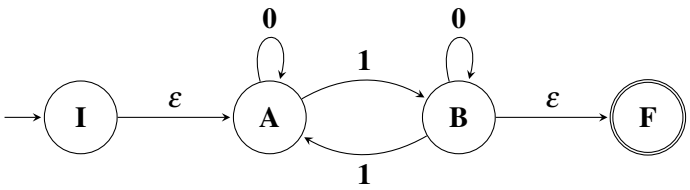


Conversão de uma AFG numa ER: Exemplo 2

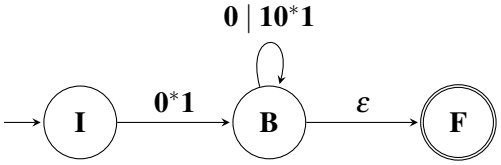
- Obtenha uma ER equivalente ao AF seguinte:



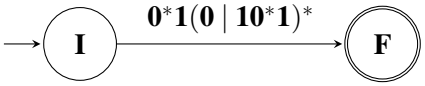
- Regras 1 e 2 :



- Eliminar o estado *A* pela regra 4 :



- Finalmente, eliminar o estado *B* pela regra 3 :



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