

# AULAS TEÓRICO-PRÁTICAS DE COMPILADORES

2º semestre de 2002/2003

---

## AULA Nº 4 (3 horas)

Exercício sobre expressões regulares e autómatos finitos.

### 1 Exercícios com Expressões Regulares

1.1 Escreva expressões regulares para os seguintes exemplos:

- (a) números binários;

$[01]^+$

- (b) URLs da forma: `http://www.ualg.pt` (em que o campo intermédio “ualg” é o único campo variável).

`"http://www." [a-zA-Z]+ ".pt"`

- (c) IPs da forma: 140.192.33.37 (considere que todos os IPs têm o mesmo número de dígitos por cada campo).

$[0-9] [0-9] [0-9] "." [0-9] [0-9] [0-9] "." [0-9] [0-9] "." [0-9] [0-9]$

- (d) números binários que representam inteiros sem zeros supérfluos;

$0 \mid 1[01]^*$

- (e) Strings sobre o alfabeto {a, b, c} com número ímpar de a's;

$[bc]^*a[bc]^*([bc]^*a[bc]^*a[bc]^*)^*$

- (f) Strings sobre o alfabeto {a, b, c} em que o primeiro “a” precede o primeiro “b”;

$c^*[ab(a \mid b \mid c)^*]^?$

- (g) Números binários múltiplos de 4;

$(0 \mid 1)^*00$

- (h) Números binários maiores do que 101001;

Um número é maior do que 101001 se tiver um bit a “1” em casas a seguir à 5ª (contando a partir da casa 0) ou se algum dos “0’s” em 101001 for modificado para “1”.

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

$(1|0)^* 1 (1|0) 1 (1|0) 11 |$

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

$(1|0)^* 11 1 (1|0) (1|0) 1$

- (i) [TPC] A linguagem das constantes em vírgula flutuante (notação utilizada em Java).

Expoente ? `[“e”“E”] ([“+” “-”])? ([0-9])+`  
será usada a notação <Expoente>

(para simplificar: para ser utilizada

$([0-9]^+ \text{ "." } ([0-9])^* (<\text{Expoente}>)? ([\text{"f" "F" "d" "D"}])^?$

$| \text{ "." } ([0-9]^+ (<\text{Expoente}>)? ([\text{"f" "F" "d" "D"}])^?$

$| ([0-9]^+ <\text{Expoente}> ([\text{"f" "F" "d" "D"}])^?$

$| ([0-9]^+ (<\text{Expoente}>)? [\text{"f" "F" "d" "D"}])$

1.2 Acha que é possível escrever expressões regulares para os seguintes exemplos? No caso de ser possível apresente uma solução:

(j) números binários que começam e acabam com o mesmo dígito;

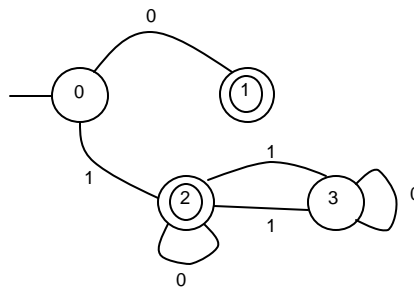
$0 \mid 1 \mid (1[10]^*1) \mid (0[10]^*0)$

(k) sequências de algarismos que formam capicuas;

Não é possível com expressões regulares. Teria de haver um operador que permitisse identificar uma cópia invertida de uma expressão regular.

## 2 Exercícios com Autómatos Finitos

2.1 Considere o autómato finito seguinte:



(a) O autómato é determinista ou não determinista?

Determinista. Qualquer que seja o estado não existem diferentes transições activadas pelo mesmo símbolo.

(b) Qual é o seu estado de início? Quais são os estados de aceitação (fina is)?

Estado de início: 0

Estados de aceitação: 1, 2

(c) Este autómato aceita a sequência 110100? Qual a sequência de estados visitados no reconhecimento desta String?

1 1 0 1 0 0

0 ? 2 ? 3 ? 3 ? 2 ? 2 ? 2

(d) Qual é a String mais pequena que o autómato aceita?

0 ou 1

(e) Pode indicar a String maior que o autómato aceita?

Não. O autómato aceita Strings de comprimento ilimitado.

(f) Por palavras, qual é a linguagem que o autômato aceita?

O autômato aceita um zero de cada vez ou binários começados por 1 (neste último caso as palavras aceites têm um número ímpar de 1's).

2.2 Desenhe os NFAs para as expressões regulares seguintes. Depois converta cada um deles para DFA.

(a)  $[01]^+$

NFA

DFA

(b)  $1^+$

NFA

DFA

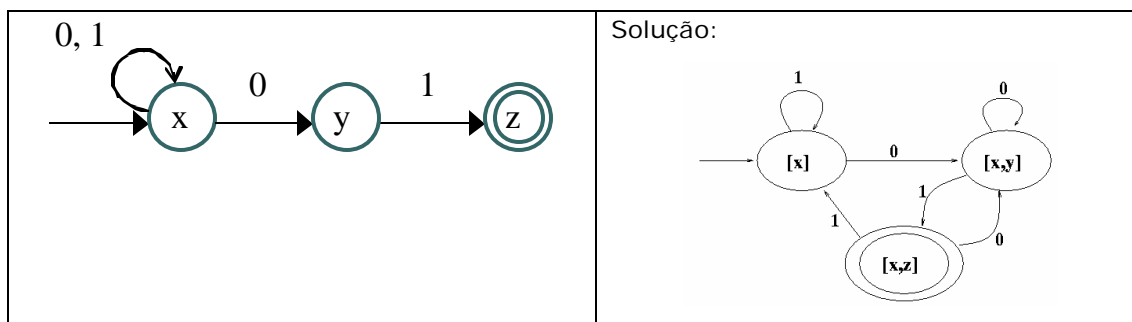
(c)  $0 \mid 1[0 \mid 1]^+$

NFA

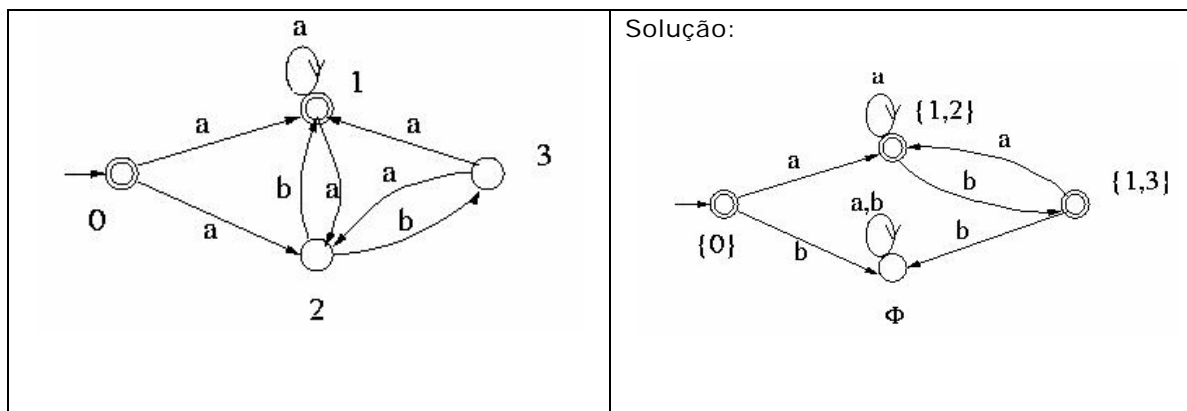
DFA

2.3 Converta os NFAs seguintes para DFAs:

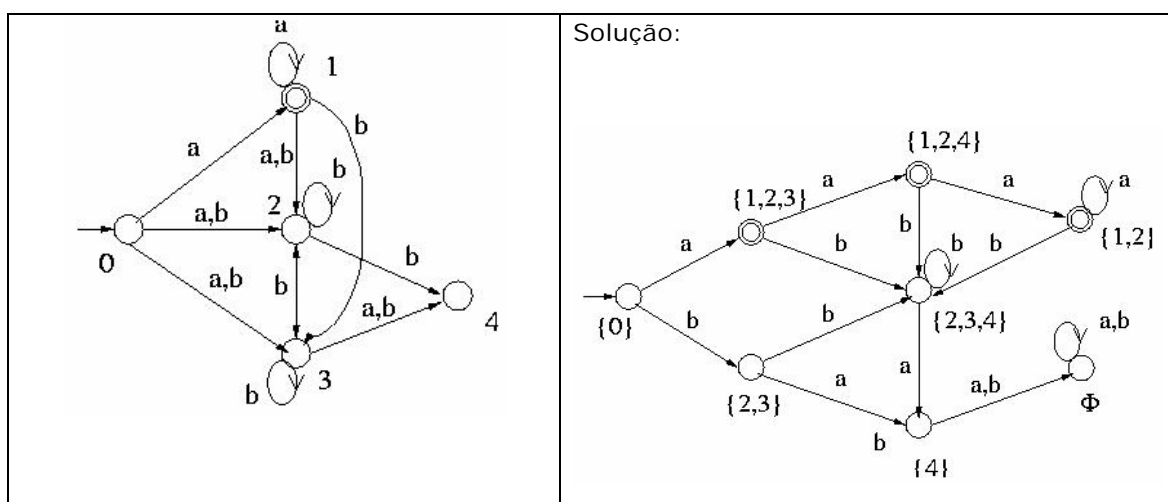
(a)



(b)



(c)



2.4 Um analisador lexical baseado num interpretador de um DFA utiliza duas tabelas:

**edges**: indexada pelo número do estado e símbolo de entrada, retorna o número do estado, e

**final**: indexada pelo número do estado, retorna 0 ou um número representativo da acção a realizar.

Considerando a seguinte especificação:

$(aba)^+$  ? acção número 1

$a(b^*)a$  ? acção número 2

$a \mid b$  ? acção número 3

Apresente as tabelas **edge** e **final** para o analisador lexical.

Solução: Baseado no DFA obtido (em anexo)

Int NumStates = 10

Int Edge[NumStates][256] = { /\* ... 0 1 2 ... 9 ... a b c ... \*/

/\* estado 0 \*/ { ..., 0, 0, 0, ..., 0, ..., 0, 0, 0, ... },

```
/* estado 1 */          { ..., 0, 0, 0, ..., 0, ..., 3, 2, 0, ... } ,
/* estado 2 */          { ..., 0, 0, 0, ..., 0, ..., 0, 0, 0, ... } ,
/* estado 3 */          { ..., 0, 0, 0, ..., 0, ..., 4, 5, 0, ... } ,
/* estado 4 */          { ..., 0, 0, 0, ..., 0, ..., 0, 0, 0, ... } ,
/* estado 5 */          { ..., 0, 0, 0, ..., 0, ..., 7, 6, 0, ... } ,
/* estado 6 */          { ..., 0, 0, 0, ..., 0, ..., 4, 6, 0, ... } ,
/* estado 7 */          { ..., 0, 0, 0, ..., 0, ..., 8, 0, 0, ... } ,
/* estado 8 */          { ..., 0, 0, 0, ..., 0, ..., 0, 9, 0, ... } ,
/* estado 9 */          { ..., 0, 0, 0, ..., 0, ..., 7, 0, 0, ... }
}

// estado          0 1 2 3 4 5 6 7 8 9
int final[NumStates] = {0, 0, 3, 3, 2, 0, 0, 1, 0, 0}
```