

# Compiladores - Autômatos

---

Fabio Mascarenhas - 2013.1

<http://www.dcc.ufrj.br/~fabiom/comp>

# Especificação x Implementação

---

- Usamos expressões regulares para dar a *especificação léxica* da linguagem
- Mas como podemos fazer a *implementação* do analisador léxico a partir dessa especificação?

# Especificação x Implementação

---

- Usamos expressões regulares para dar a *especificação léxica* da linguagem
- Mas como podemos fazer a *implementação* do analisador léxico a partir dessa especificação?
  - Autômatos finitos!
  - Algoritmos para converter expressões regulares são conhecidos e podem ser reaproveitados, e autômatos levam a um analisador léxico bastante eficiente

# Autômatos Finitos

---

- Um autômato finito é formado por:
  - Um *alfabeto* de entrada
  - Um conjunto de *estados*
  - Um *estado inicial*
  - Um conjunto de *estados finais rotulados*
  - Um conjunto de transições entre estados

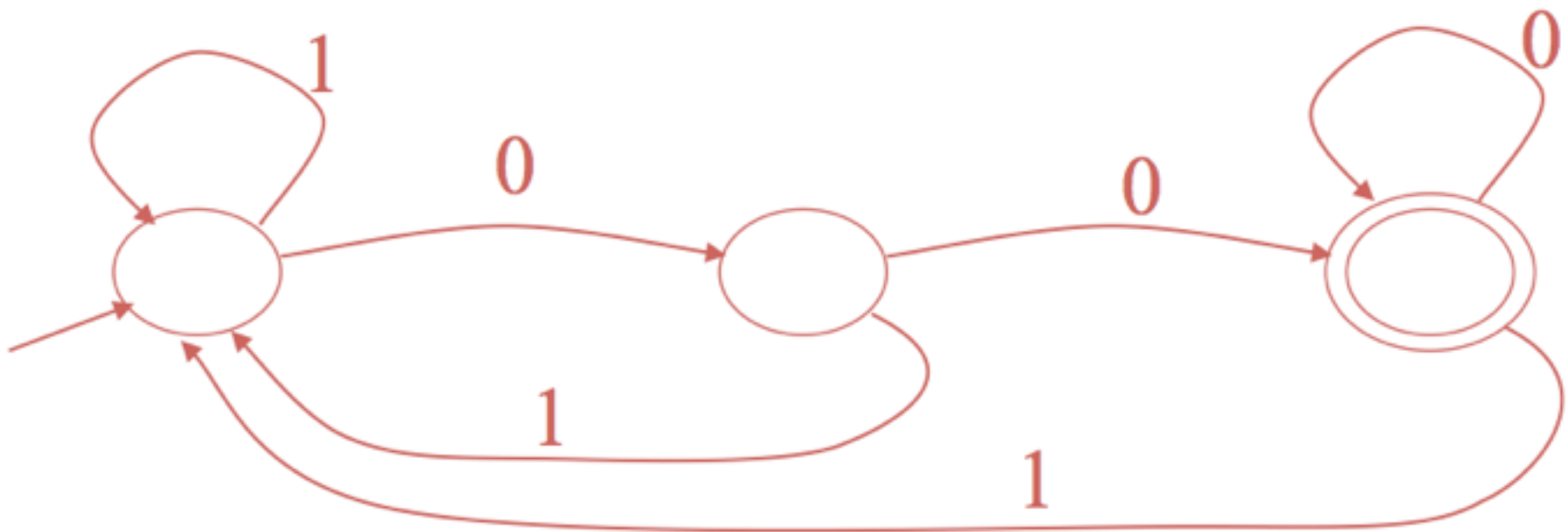
# Transições

---

- Uma transição  $s_1 \xrightarrow{a} s_2$  quer dizer que se autômato está no estado  $s_1$  e o próximo símbolo da entrada é  $a$  então ele vai para o estado  $s_2$
- Se não há mais caracteres na entrada e estamos em um estado final então o autômato *aceitou* a entrada
- Se em algum ponto não foi possível tomar nenhuma transição, ou a entrada acabou e não estamos em um estado final, o autômato *rejeitou* a entrada

# Graficamente

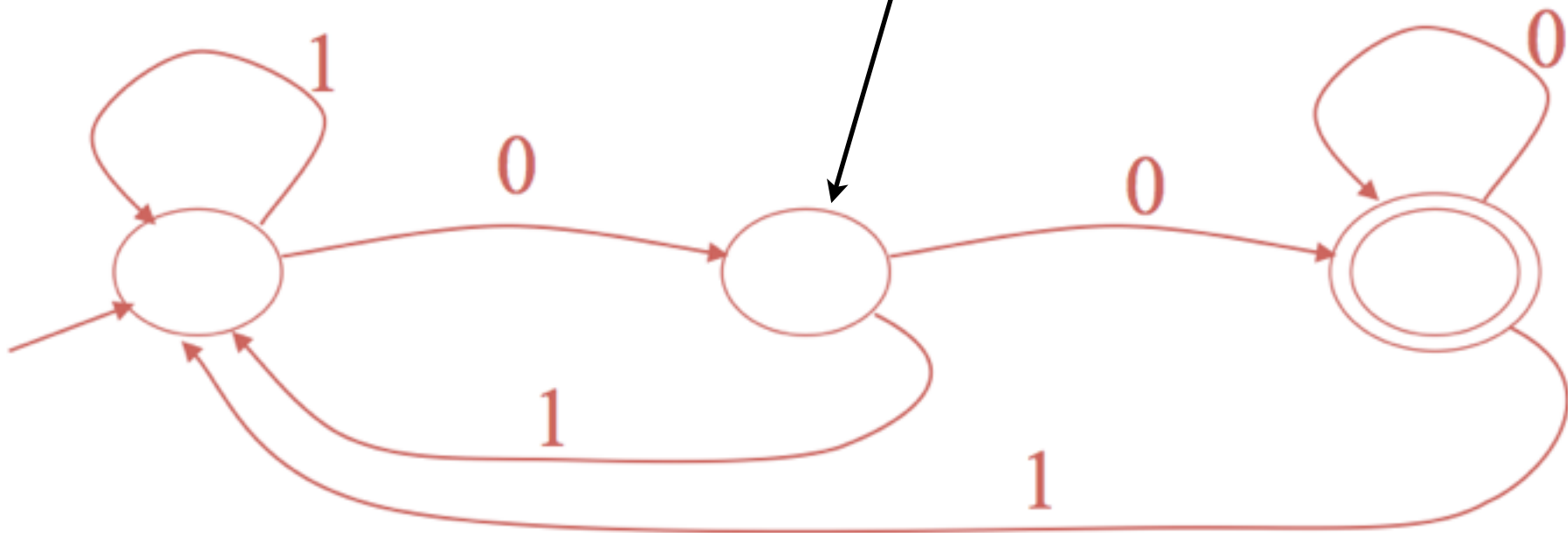
---



# Graficamente

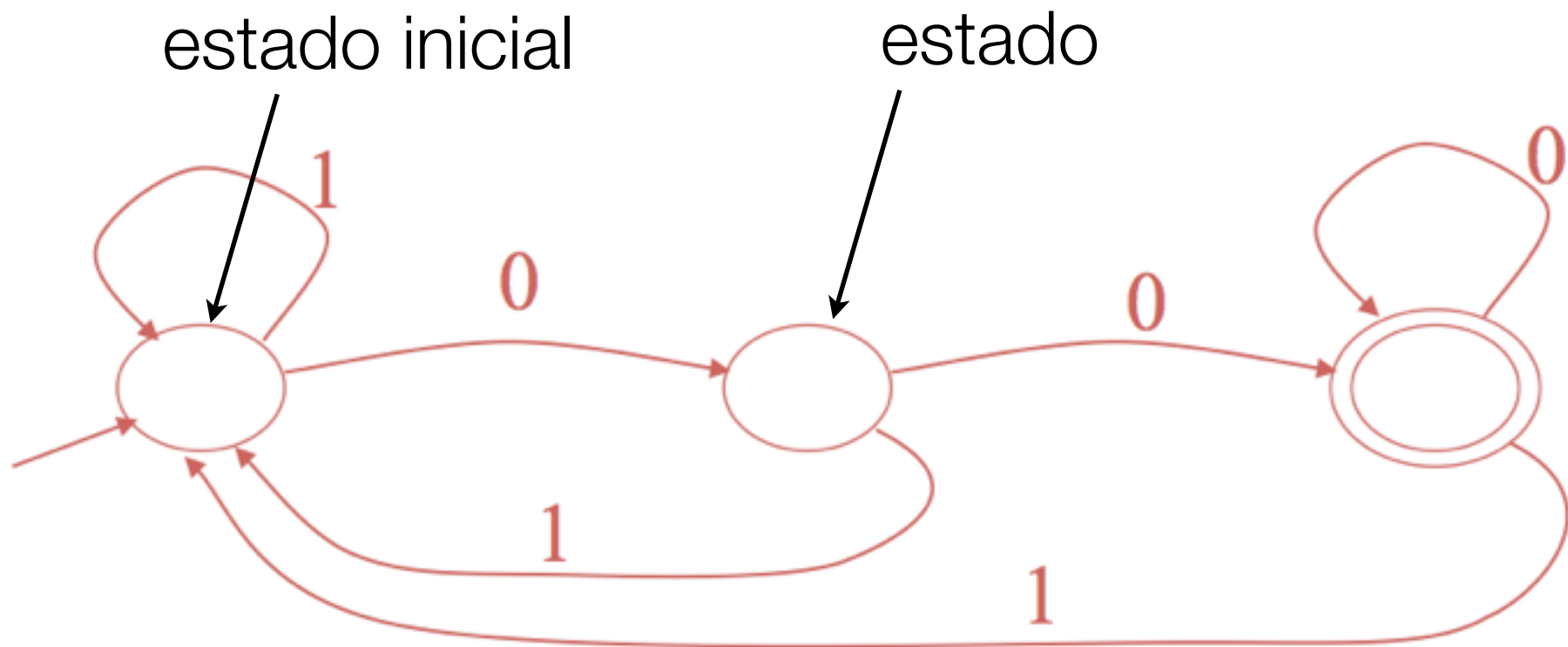
---

estado



# Graficamente

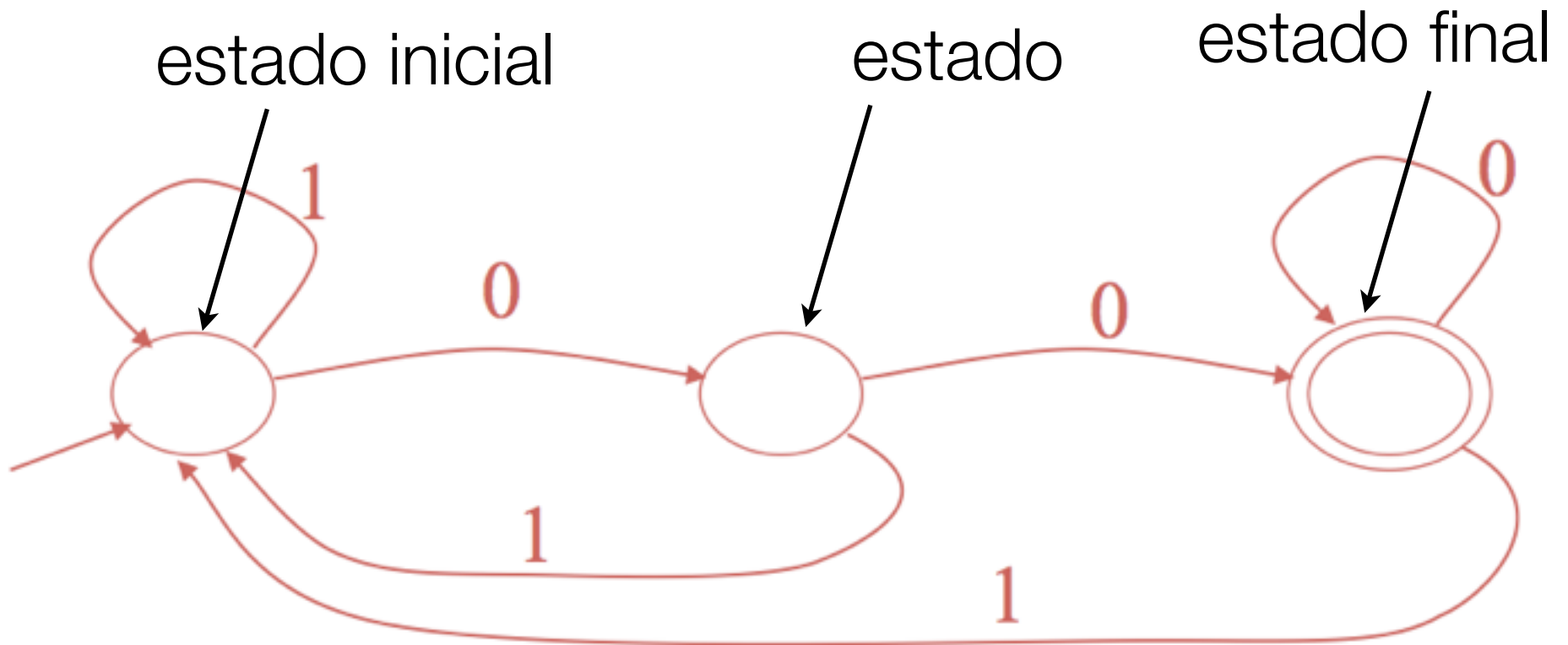
---



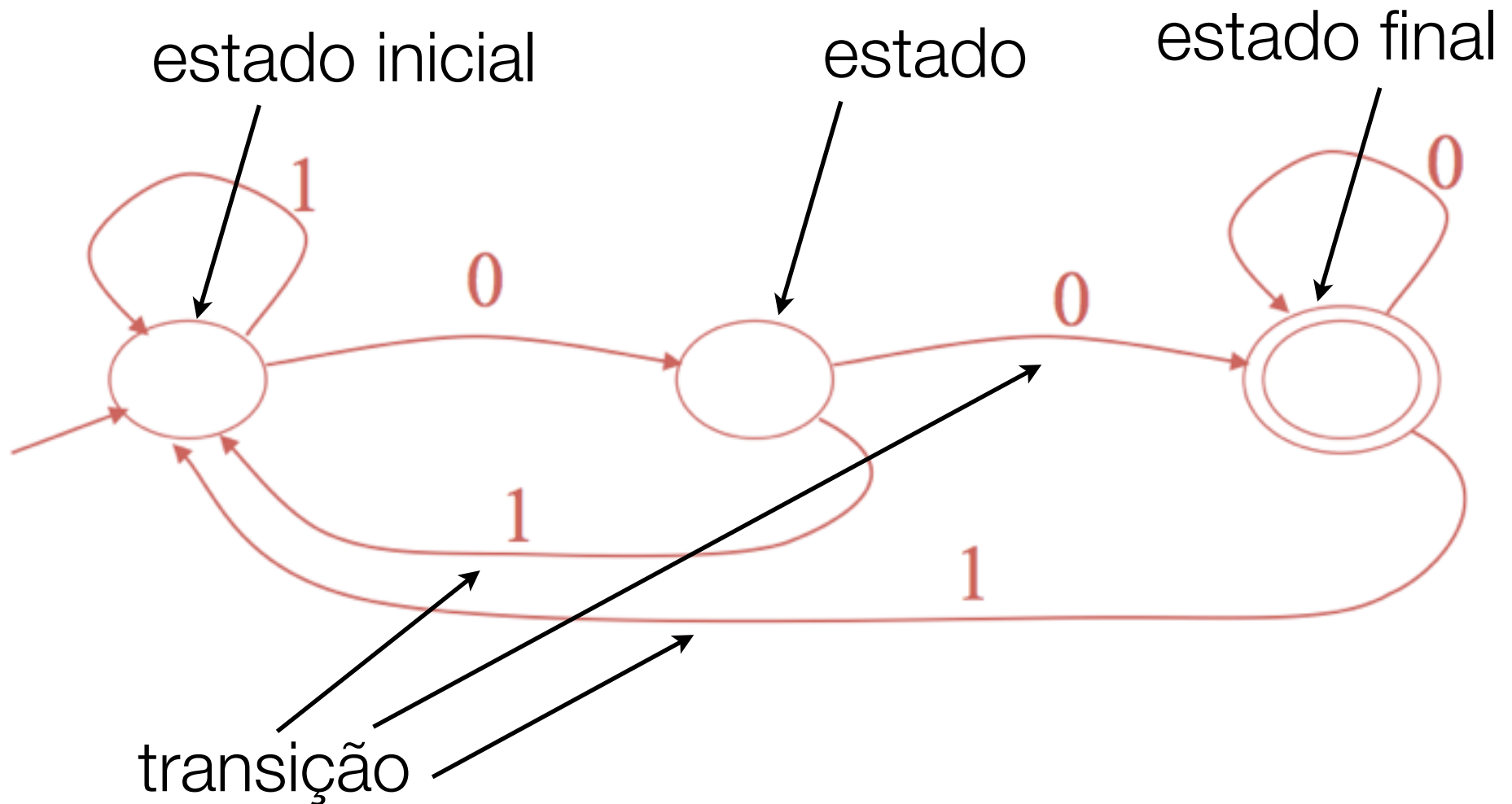


# Graficamente

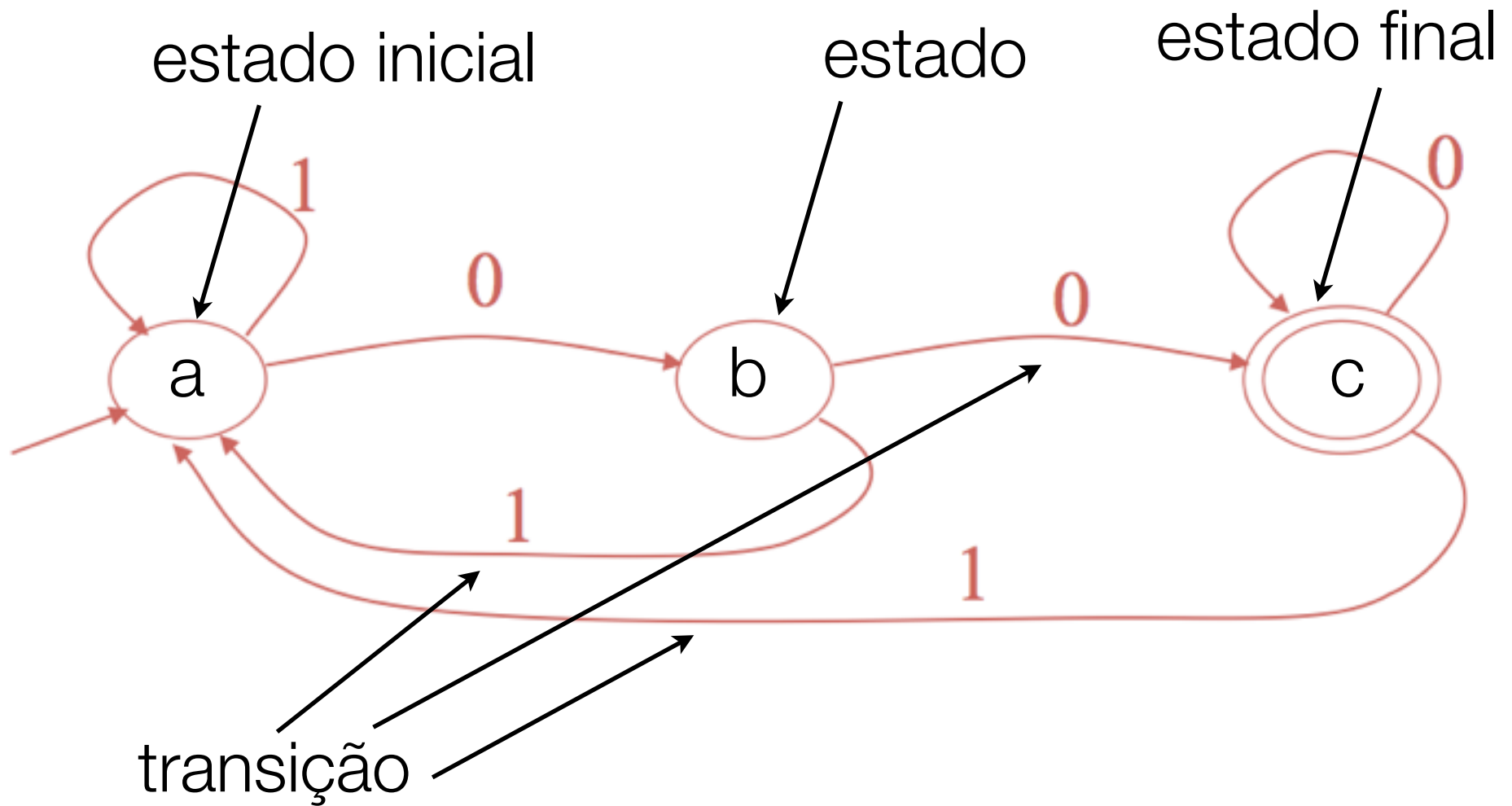
---



# Graficamente



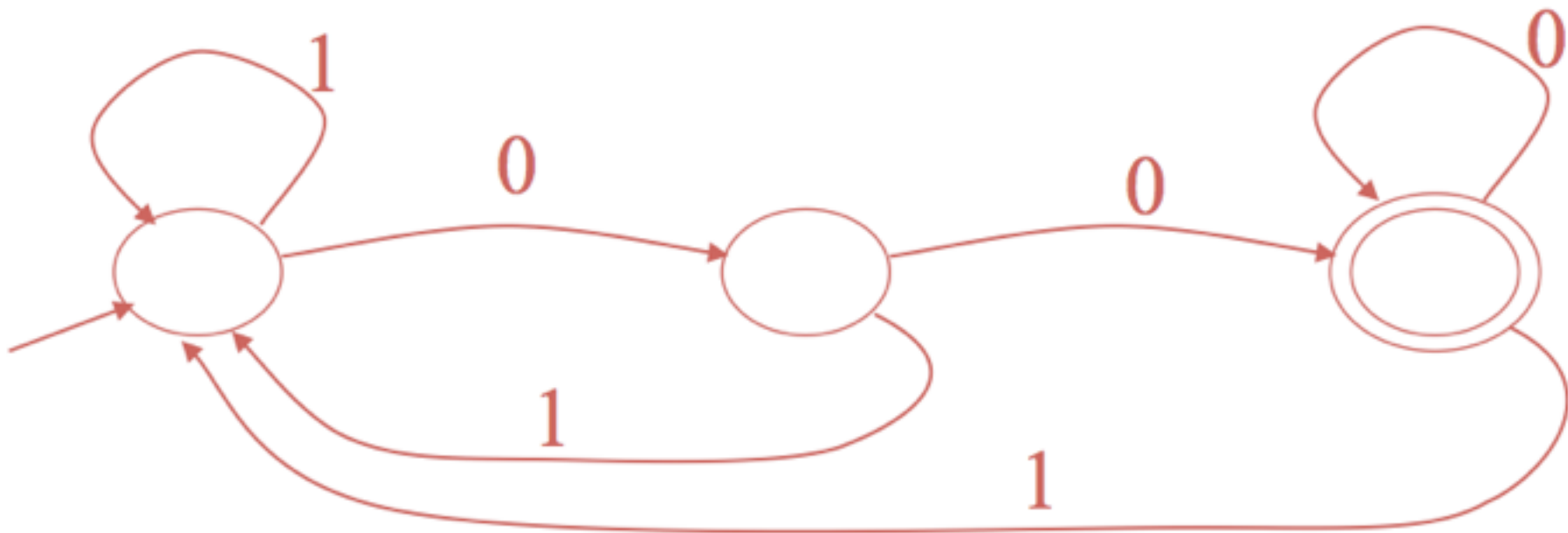
# Graficamente



# Graficamente

---

$(0|1)^*00$



# Transições $\epsilon$

---

- Uma transição  $\epsilon$  é uma transição que pode ser tomada espontaneamente pelo autômato, sem ler nenhum símbolo da entrada
- Podemos também construir um autômato que pode tomar mais de uma transição dado um estado e um símbolo
- Autômatos com transições  $\epsilon$  e múltiplas transições saindo de um mesmo estado para um mesmo caractere são *não-determinísticos*

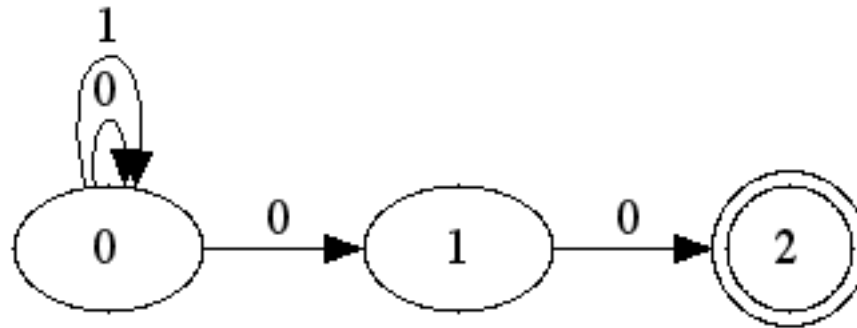
# DFA vs NFA

---

- Um DFA é um autômato determinístico, um NFA é não-determinístico
- Um DFA, dada uma entrada, toma apenas um caminho através dos seus estados
- Um NFA toma **todos** os caminhos possíveis para aquela entrada, e aceita entrada se **pelo menos um** caminho termina em um estado final

# Funcionamento de um NFA

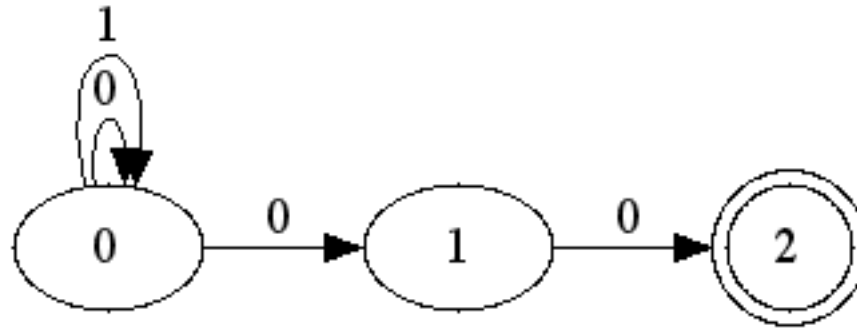
---



- Entrada:
- Estados:

# Funcionamento de um NFA

---

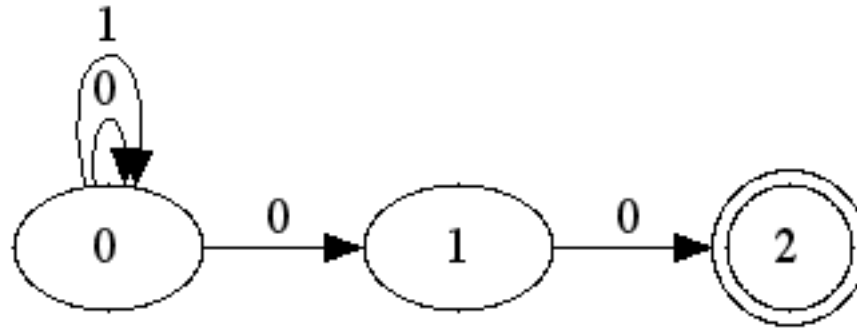


- Entrada: 1
- Estados: { 0 }



# Funcionamento de um NFA

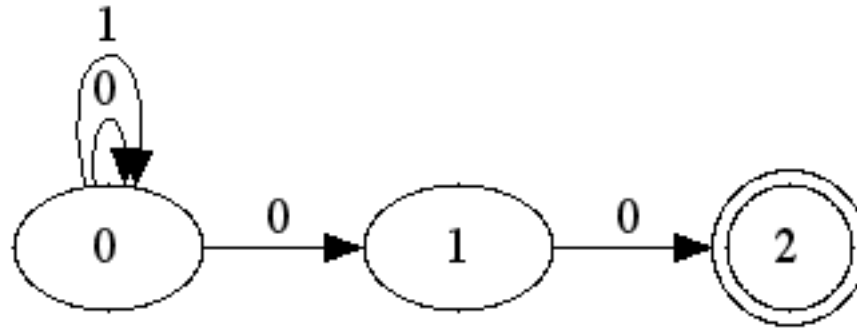
---



- Entrada: 1 0
- Estados: { 0 } { 0, 1 }

# Funcionamento de um NFA

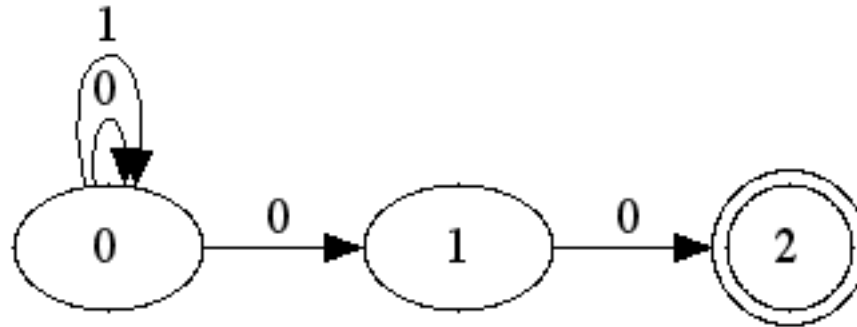
---



- Entrada: 1 0 0
- Estados: { 0 } { 0, 1 } { 0, 1, 2 }

# Funcionamento de um NFA

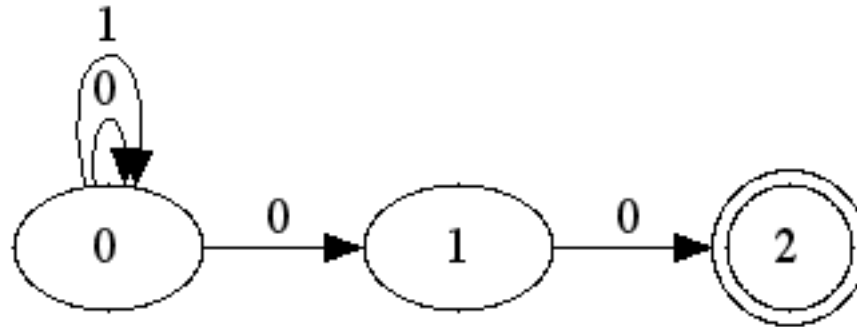
---



- Entrada: 1 0 0
- Estados: { 0 } { 0, 1 } { 0, 1, 2 }
- Aceita!

# Funcionamento de um NFA

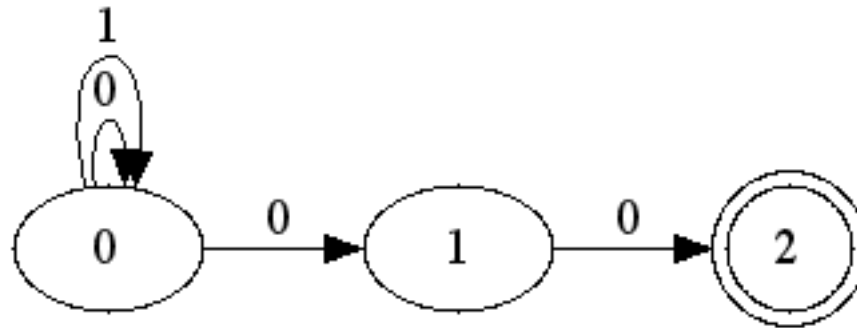
---



- Entrada: 0
- Estados: { 0, 1 }

# Funcionamento de um NFA

---



- Entrada:     0        1
- Estados:    { 0, 1 }   { 0 }
- Não aceita!

# Autômatos e linguagens

---

- DFAs, NFAs e expressões regulares todos expressam a mesma classe de conjunto de símbolos
  - Linguagens regulares
- Isso quer dizer que podemos converter de um para outro
- DFAs são mais rápidos para executar
- NFAs têm representação mais compacta
- Expressões regulares são mais fáceis de entender qual conjunto está sendo expresso

# Autômatos e linguagens

---

- DFAs, NFAs e expressões regulares todos expressam a mesma classe de conjunto de símbolos
  - Linguagens regulares
- Isso quer dizer que podemos converter de um para outro
- DFAs são mais rápidos para executar
- NFAs têm representação mais compacta
- Expressões regulares são mais fáceis de entender qual conjunto está sendo expresso

Por isso usamos expressões regulares para a especificação, e DFAs (ou NFAs) para implementação!

# DFA de análise léxica

---

- Um DFA de análise léxica tem os estados finais rotulados com tipos de token
- A ideia é executar o autômato até chegar no final da entrada, ou dar erro por não conseguir fazer uma transição, mantendo uma pilha de estados visitados e o token que está sendo lido
- Então voltamos atrás, botando símbolos de volta na entrada, até chegar em um estado final, que vai dar o tipo do token



# Analizador léxico de tabela

---

```
// reconhecer palavras
estado = s0
token = ""
pilha.limpa()
while (!eof && estado ≠ erro ) do
    char = leChar()

    token = token + char
    push (estado)
    estado = trans(estado,char)
end;
```

```
// limpar estado final
while (estado ∉ SA and !pilha.vazia()) do

    estado ← pilha.pop()
    token = token.truncaUltimo()
    voltaChar()
end;

if (estado ∈ SA )

    // rótulo do estado é tipo do token
    then return <estado, token>
    else return erro
```

# Uma otimização

---

- Se visitamos um estado final então podemos limpar a pilha, já que vamos parar nele na volta

```
// reconhecer palavras
```

```
estado =  $s_0$ 
```

```
token = ""
```

```
pilha.limpa()
```

```
pilha.push(erro)
```

```
while (estado  $\neq$  erro) do
```

```
  char = leChar()
```

```
  token = token + char
```

```
  if estado  $\in S_A$ 
```

```
    then pilha.limpa()
```

```
    push (estado)
```

```
    estado = trans(estado,char)
```

```
end;
```

```
// limpar estado final
```

```
while (estado  $\notin S_A$  and estado  $\neq$  erro) do
```

```
  estado  $\leftarrow$  pilha.pop()
```

```
  token = token.truncaUltimo()
```

```
  voltaChar()
```

```
end;
```

```
if (estado  $\in S_A$ )
```

```
  // rótulo do estado é tipo do token
```

```
  then return <estado, token>
```

```
  else return erro
```

# Construindo o DFA de análise léxica

---

- Passo 1: construir um NFA para cada regra, o estado final desse NFA é rotulado com o tipo do token
  - Construção de Thompson
- Passo 2: combinar os NFAs em um NFA com um estado inicial que leva aos estados iniciais do NFA de cada regra via uma transição  $\epsilon$
- Passo 3: transformar esse NFA em um DFA, estados finais ficam com o rótulo da regra que aparece primeiro
  - Algoritmo de construção de subconjuntos

# DFA da linguagem de comandos simples

