



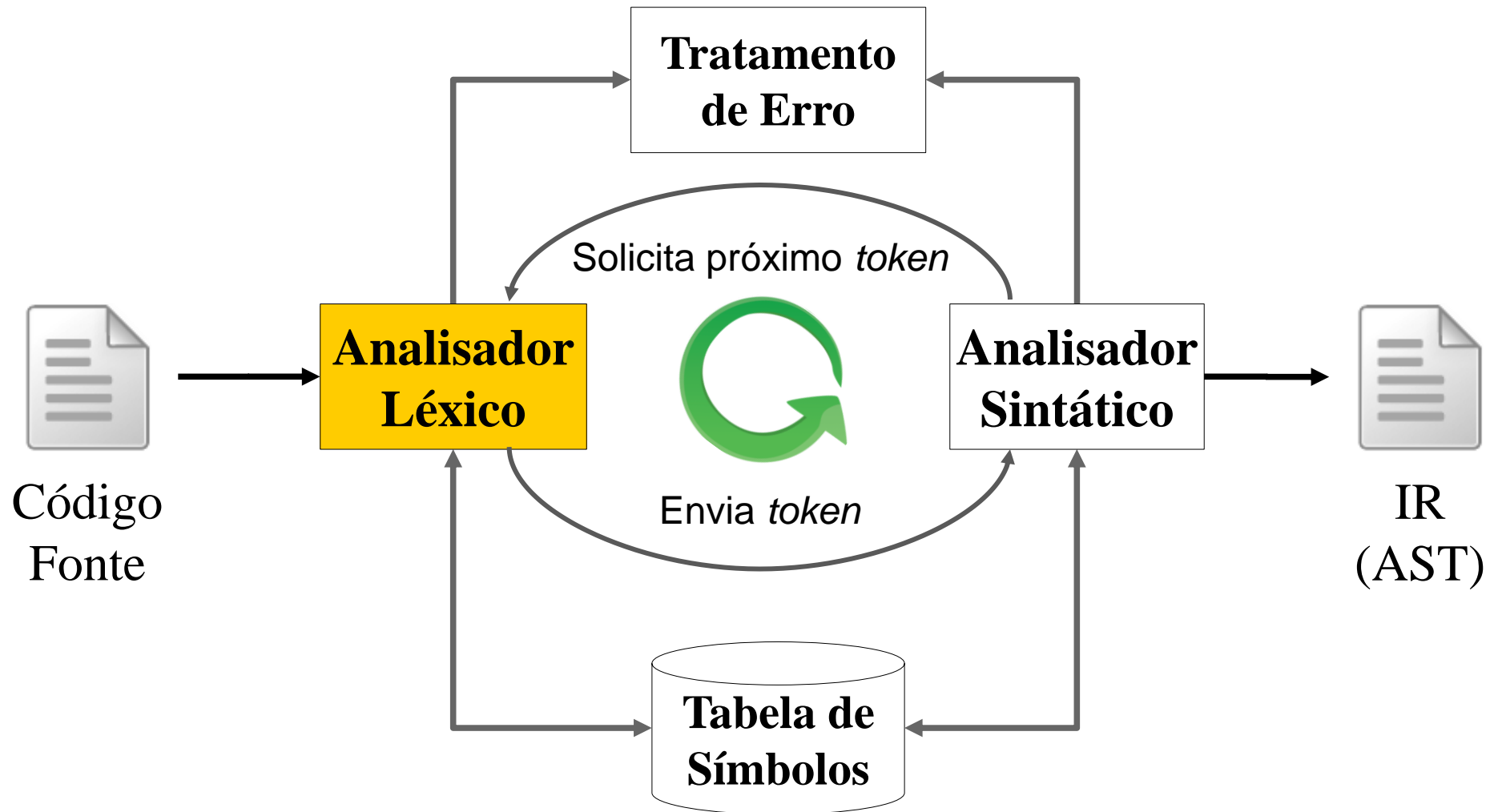
Universidade Federal de Uberlândia
Faculdade de Computação



Análise Léxica

Curso de Bacharelado em Ciência da Computação
GBC071 - Construção de Compiladores
Prof. Luiz Gustavo Almeida Martins

Etapa de Análise do Compilador



Processo **iterativo e reativo**

Projeto da Etapa de Análise

- Por que dividir análise léxica e sintática?
 - Permite simplificar uma das fases (**projeto mais simples**)
 - **Ex:** análise sintática com e sem tratamento para comentários
 - Permite a adoção de técnicas especializadas/otimizadas para certas tarefas (**melhor eficiência do compilador**)
 - **Ex:** técnicas de buferização especializadas para a leitura de caracteres e o processamento de *tokens*
 - Facilita a portabilidade do compilador (**manutenção**)
 - Peculiaridades da linguagem ou dos dispositivos podem ser restringidas ao analisador léxico **facilitando modificações**
 - **Ex:** Tratamento de símbolos especiais (^ em Pascal) ou fora do padrão (ex: letras do alfabeto grego)

Análise Léxica

- Análise léxica é a 1ª fase da etapa de análise
 - Envolve **varredura** (*scanner*) e **análise** propriamente dita (*lexer*)
 - Implementada como uma **sub-rotina do analisador sintático**
- Tarefa principal: **agrupar os caracteres** do código fonte em ***tokens***
 - Ler os caracteres de um ***buffer*** de entrada do programa fonte
 - Agrupá-los em lexemas
 - Produzir uma sequência de ***tokens*** como saída
 - Usada pelo analisador sintático para validar as regras da gramática
- Pode realizar outras **tarefas secundárias**:
 - Remover símbolos desnecessários
 - Comentários e separadores (espaço em branco, tabulação e quebra de linha)
 - Correlacionar as mensagens de erro com o código fonte
 - Processar diretivas de controle (**ex:** expansão de macros)

Token

- **Classe de elementos aceitos** em uma linguagem de programação
 - **Unidade básica da sintaxe**
 - **Ex:** identificadores, operadores, palavras-chave, etc.
- **Elo de ligação** entre as análises léxica e sintática
 - Representa os **lexemas** do código fonte
 - **Palavras aceitas** pela linguagem
 - Corresponde a um **nó folha** da árvore sintática
 - **Símbolo terminal** de uma gramática livre de contexto

Token

- Representado por um par **<Nome, Atributo>**:
 - **Nome**: símbolo abstrato que indica o tipo do *token*
 - Símbolos de entrada do analisador sintático
 - **Atributo**: guarda informações adicionais necessárias (**opcional**)
 - Atributo pode ser um **tipo estruturado** que guarda várias informações
 - **Ex**: lexema encontrado, valor de dado, localização na entrada, etc.
- **Classes de *tokens*** presentes em uma linguagem:
 - **Palavras-chave**
 - **Operadores** (organizados individualmente ou em classes)
 - **Identificadores (ID)**
 - **Constantes** (**ex**: número ou cadeia de literais)
 - **Símbolos de pontuação** (**ex**: parênteses, ponto-e-vírgula, etc.)

Exercício

- Identifique os ***tokens*** dos códigos abaixo, associando a cada um seu par <Nome,Atributo>

Pascal

```
function max (i, j: integer): integer;  
{ return maximum of integers I and j}  
begin  
  if i > j then max := i  
  else max := j  
end;
```

C

```
int max (i, j) int i, j;  
{ /* maximum of integers i and j */  
return i > j ? i : j;  
}
```

- fonte: [Aluisio, 2011]

Padrão de um *Token*

- **Regra** que define o **conjunto de palavras** associado a um *token*
 - **Descreve a forma (cadeia de caracteres)** que os lexemas de um *token* podem assumir
 - **Ex:** Qualquer ID é formado por uma letra seguida por letras, números e “_”
- **Expressões regulares** são uma importante notação para **especificar os padrões de lexemas**
 - **Ex:** *letra (letra | digito | _)**
- Os padrões são usados na **construção dos reconhecedores** das cadeias do conjunto

Lexema

- **Sequência de caracteres** no programa fonte **que casa com o padrão** de um *token*
- **Palavra reconhecida** pelo analisador léxico como uma instância do *token*
- **Exemplos:** (fonte: [Aho, 2008])

<i>Token</i>	Descrição Informal	Exemplo Lexemas
ID	Letra seguida por letras e dígitos	Nome, D2, vlr_max
Literal	Qualquer caractere (≠ de “) entre “s	“exemplo de token”
Número	Qualquer constante numérica	3.14159, -3, 0.32e6
Operador relacional	< ou > ou <= ou >= ou == ou !=	<, ==
while	while	while

Atributos dos *Tokens*

- Usado quando **mais de um lexema** **casa com o padrão** do *token*
 - **Ex:** identificador, operador relacional, etc.
- Fornece **informações adicionais** para as fases seguintes do compilador
 - **Descreve o lexema** representado pelo *token*
 - **Nome do *token*** influencia nas decisões durante a **análise sintática**
 - **Valor do atributo** influencia na tradução do *token* **após o reconhecimento sintático**

Erros Léxicos

- Identificar um erro no código fonte durante a análise léxica é difícil sem o auxílio de outros componentes
 - **Ex:** *fi* é um identificador ou o *if* escrito errado?
- **Erros léxicos** ocorrem quando **nenhum dos padrões de *token* casa** com a entrada restante
 - **Ex:** símbolo desconhecido, lexemas mal formados, identificadores muito grandes e fim de arquivo inesperado
- Erros associados ao **tratamento de constantes:**
 - Exceder o limite de casas decimais e do expoente de números reais (**tanto no tamanho quanto no valor**)
 - Exceder o limite máximo da cadeia de caracteres
 - Exceder o limite do número de dígitos e do valor de um inteiro

Erros Léxicos

- **Estratégias de recuperação** podem ser usadas
 - Envolve transformações na entrada restante:
 - **Remover** um caractere
 - **Inserir** um caractere que falta
 - **Substituir** um caractere por outro
 - **Transpor** dois caracteres adjacentes
 - “**Modo pânico**”: remove caracteres até reconhecer o lexema
 - **Estratégia mais simples** aplica uma **única transformação**
 - **Estratégia mais geral** busca encontrar o **menor número de transformações** necessárias para obter um lexema válido
 - Na prática, é uma **estratégia muito dispendiosa**
 - Não garante efetividade dos resultados

Varredura (*Scanner*)

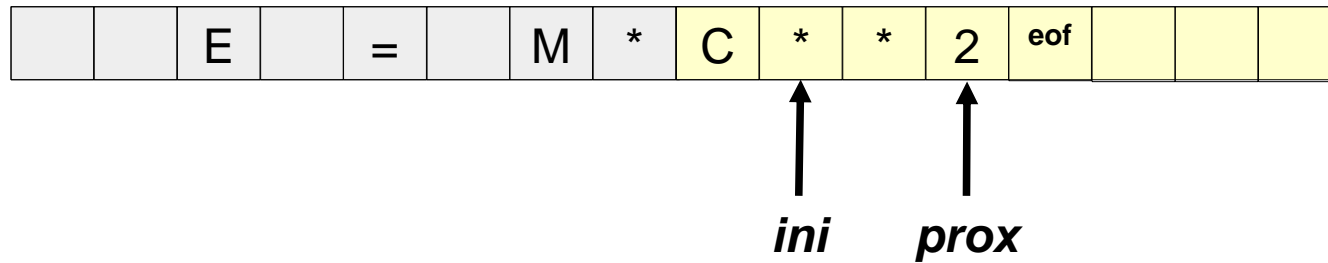
- Consiste na leitura da entrada (código fonte) para a identificação dos lexemas (casamento de um padrão)
 - Consome caractere a caractere da esquerda para direita
- **Identificação correta de um lexema** pode exigir a leitura de **1 ou + caracteres a frente** (***lookahead***)
 - **Ex:** na linguagem Fortran, as palavras-chave não são reservadas e os espaços são desconsiderados
 - DOI=1.25 \Rightarrow lexema: **DOI** *Token: ID*
 - DOI=1,25 \Rightarrow lexema: **DO** *Token: DO*
- **Técnicas especializadas de *buffering*** são empregadas para reduzir o custo da operação
 - **2 buffers** para tratar *lookaheads* grandes com segurança
 - Uso de “**sentinelas**” para evitar verificação de fim do *buffer*

Pares de *Buffer*

- Adota 2 *buffers* de entrada de mesmo tamanho
 - Relacionado com o **tamanho do bloco do disco** (ex: 4096 bytes)
- Comando de leitura do sistema **carrega todo o *buffer*** ao invés de um único caractere
 - *Buffers* são carregados alternadamente
 - Caractere especial **EOF** define fim do arquivo
- Abordagem adota **2 ponteiros**:
 - ***ini***: marca o início do lexema atual
 - ***prox***: indica o próximo caractere a ser lido
 - Implementa o *lookahead* até que haja um casamento de padrão
 - Provoca a operação de recarga sempre que extrapola o tamanho do *buffer*

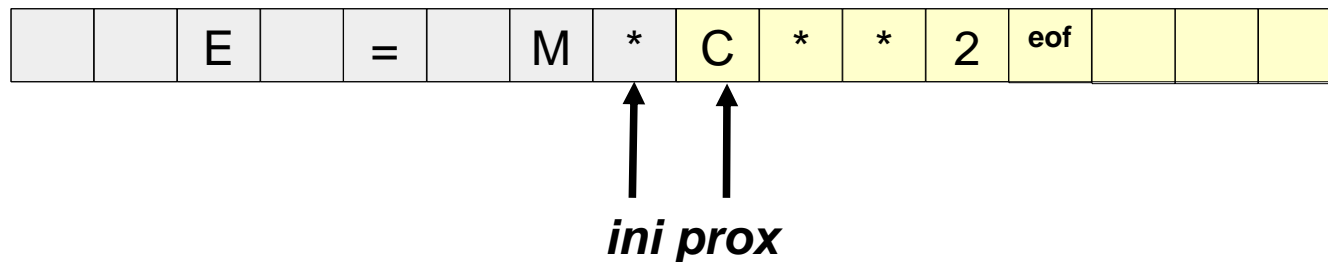
Pares de *Buffer*

- **Exemplo:** (retirado de [Aho, 2008])



Pares de *Buffer*

- **Exemplo:** (retirado de [Aho, 2008])



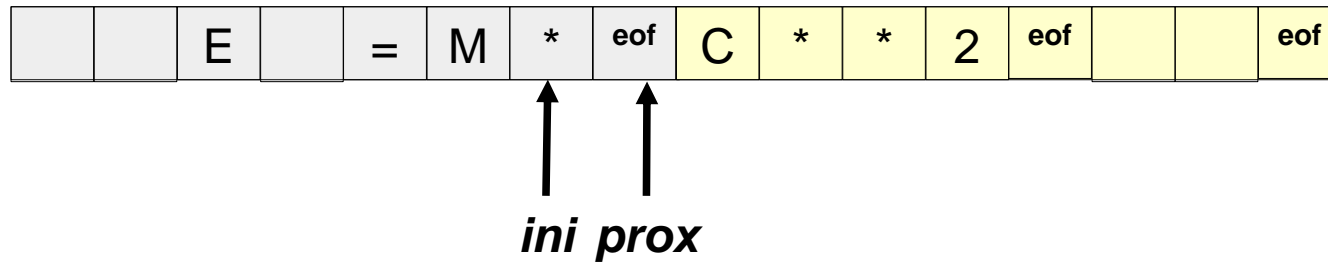
- Podemos esgotar o espaço em *buffer*?
 - **Lexemas são geralmente pequenos**
 - 1 ou 2 caracteres de *lookahead* são suficientes
 - Problema na **leitura de cadeias longas** ($> \text{buffer}$)
 - Linguagens que não tratam palavras-chave como reservadas
 - Existência de lexemas maiores que o *buffer* (ex: literais)

Sentinelas

- Envolve incluir um **caractere extra** (**sentinela**) no fim de cada *buffer*
 - Geralmente é usado o caractere especial **EOF**
 - Exige **uma posição a mais na estrutura** de armazenamento
- Visa reduzir a quantidade de testes a cada caractere lido
 - **Original:** fim de *buffer* e qual caractere lido
 - **Sentinela:** qual caractere lido
- Posição do **EOF** indica o cenário a ser tratado:
 - **Final do *buffer*:** **EOF** na última posição
 - **Final do arquivo de entrada:** **EOF** nas demais posições

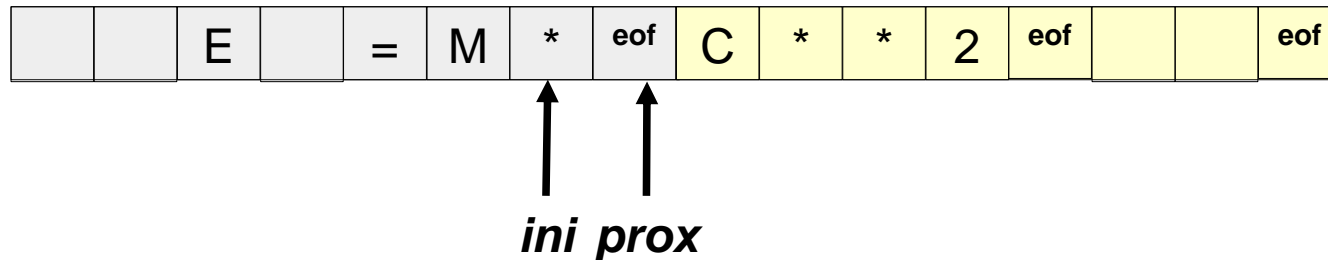
Sentinelas

- **Exemplo:** (retirado de [Aho, 2008])



Sentinelas

- **Exemplo:** (retirado de [Aho, 2008])



- **Tratamento do EOF:**
 - SE *prox* está no fim do 1º *buffer* ENTÃO
 - carrega o 2º *buffer*
 - *prox* = início do 2º *buffer*
 - SENÃO SE *prox* está no fim do 2º *buffer* ENTÃO
 - carrega o 1º *buffer*
 - *prox* = início do 1º *buffer*
 - SENÃO // Fim do arquivo
 - fim da análise léxica

Expressões Regulares (ER)

- **Notação formal** usada para **especificar** a estrutura (**padrões**) dos *tokens*
 - Possibilita um analisador léxico sem erros (**estrutura precisa**)
 - **Ex:** *string* definida como uma cadeia de **caracteres** entre aspas
 - Não são todos os caracteres que são permitidos (ex: '\n')
 - **Ex:** números reais em notação de ponto fixo (ex: 3.0 e 0.12)
 - 3. e .12 são aceitos em Fortran, mas não em Pascal
- Descreve as linguagens a partir de **3 operações** sobre os símbolos de algum alfabeto:
 - **União:** $L \cup M = \{s \mid s \in L \text{ ou } s \in M\}$
 - **Concatenação:** $LM = \{st \mid s \in L \text{ ou } t \in M\}$
 - **Fecho Kleene:** $L^* = \bigcup_{i=0..∞} L^i$

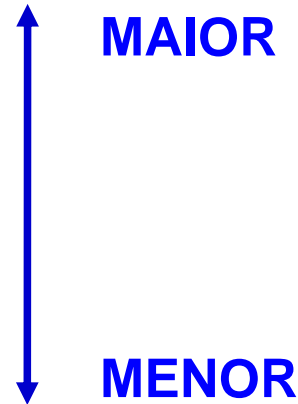
Processo Indutivo

- ER podem ser definidas a partir de **expressões menores**
 - Cada expressão r denota uma linguagem $L(r)$
- Regras que formam a **base** das expressões regulares:
 - ε é uma expressão regular e $L(\varepsilon)$ é $\{\varepsilon\}$
 - Se a é um símbolo de Σ , então **a** é uma expressão regular e $L(\mathbf{a}) = \{a\}$
- Regras que formam a parte indutiva das expressões regulares:
 - $(r) \mid (s)$ é uma expressão regular denotando $L(r) \cup L(s)$
 - $(r)(s)$ é uma expressão regular denotando $L(r)L(s)$
 - $(r)^*$ é uma expressão regular denotando $L(r)^*$
 - Se r é uma expressão regular, **(r)** também é e denota a mesma linguagem

Precedência dos Operadores

- **Precedência:**

- Fecho (*)
- Concatenação
- União (|)



- Todos com **associatividade à esquerda**

Definições Regulares

- Especifica expressões regulares a partir de outras expressões previamente definidas
 - **Expressões mais simples** são nomeadas e seus nomes usados em **expressões mais complexas**
- **Não** são usadas definições recursivas
 - Nova definição baseia-se apenas nas anteriores
- Permite gerar expressões regulares **apenas com os símbolos do Σ**
 - Aplicação de **substituições consecutivas** das definições mais simples nas mais complexas

Definições Regulares

- Seja Σ o alfabeto. Uma definição regular é uma sequência de definições da forma:
 - $d_1 \rightarrow r_1$
 - $d_2 \rightarrow r_2$
 - ...
 - $d_n \rightarrow r_n$
- Cada d_i é um **novo símbolo** $\notin \Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$
- Cada r_i é uma **expressão regular** formada por símbolos $\in \Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

Definições Regulares

- **Exemplos:** (retirado de [Aho, 2008])
 - **Identificadores em C:**
 - $letra_ \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _$
 - $digito \rightarrow 0 \mid 1 \mid \dots \mid 9$
 - $id \rightarrow letra_ (letra_ \mid digito)^*$
 - **Números sem sinal (inteiro ou ponto flutuante):**
 - $digito \rightarrow 0 \mid 1 \mid \dots \mid 9$
 - $digitos \rightarrow digito\ digito^*$
 - $fração \rightarrow .digitos \mid \epsilon$
 - $exponente \rightarrow (E(+ \mid - \mid \epsilon)\ digitos) \mid \epsilon$
 - $num \rightarrow digitos\ fração\ expoente$

Definições Regulares

- **Exemplos:** (retirado de [Aho, 2008])

- **Identificadores em C:**

- $letra_ \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _$
- $digito \rightarrow 0 \mid 1 \mid \dots \mid 9$
- $id \rightarrow letra_ (letra_ \mid digito)^*$

- **Números sem sinal (inteiro ou ponto flutuante):**

- $digito \rightarrow 0 \mid 1 \mid \dots \mid 9$
- $digitos \rightarrow digito\ digito^*$
- $fração \rightarrow .digitos \mid \epsilon$
- $exponente \rightarrow (E(+ \mid - \mid \epsilon)\ digitos) \mid \epsilon$
- $num \rightarrow digitos\ fração\ expoente$

ϵ implementa a
opcionalidade

Como seria a derivação de 512, 0.365 e 6.32E-3?

Notação Estendida

- Extensões são adicionadas para **melhorar a capacidade de especificar** padrões de cadeia
 - Não fazem parte da notação convencional de expressões regulares
 - Usadas na **especificação de analisadores léxicos** (ex: Lex)
- **Extensões importantes:**
 - **Operador +:** representa o **fecho positivo** (uma ou mais instâncias)
 - (r) denota a linguagem $L(r)$
 - $r^* = r^+ \mid \varepsilon$ e $r^+ = rr^*$
 - **Operador ?:** representa **zero ou uma instância**
 - $(r)?$ é equivalente a $r \mid \varepsilon$
 - **Operador []:** representa classes de caracteres
 - $[abc]$ é equivalente a $a \mid b \mid c$
 - Útil para representar **sequências de símbolos**
 - $[a-z]$ é equivalente a $a \mid b \mid \dots \mid z$

Notação Estendida

- **Exemplo: identificadores**

(retirado de [Aho, 2008])

- **Notação convencional:**

- $letra_ \rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z \mid _$
- $digito \rightarrow 0 \mid 1 \mid \dots \mid 9$
- $id \rightarrow letra_ (letra_ \mid digito)^*$

- **Notação estendida:**

- $letra_ \rightarrow [A - Z a - z _]$
- $digito \rightarrow [0 - 9]$
- $id \rightarrow letra_ [letra_ digito]^*$

Notação Estendida

- **Exemplo: números sem sinal**

(retirado de [Aho, 2008])

- **Notação convencional:**

- $digito \rightarrow 0 \mid 1 \mid \dots \mid 9$
- $digitos \rightarrow digito\ digito^*$
- $fração \rightarrow .digitos \mid \varepsilon$
- $exponente \rightarrow (E(+ \mid - \mid \varepsilon)\ digitos) \mid \varepsilon$
- $num \rightarrow digitos\ fração\ expoente$

- **Notação estendida:**

- $digito \rightarrow [0 - 9]$
- $digitos \rightarrow digito^+$
- $fração \rightarrow (.digitos) ?$
- $exponente \rightarrow (E[+ -] ? digitos) ?$
- $num \rightarrow digitos\ fração\ expoente$

Outras Extensões do Lex

EXPRESSÃO	CASA COM	EXEMPLO
c	o único caractere não operador c	a
$\backslash c$	o caractere C literalmente	$\backslash *$
$"s"$	a cadeia s literalmente	$"**"$
$.$	qualquer caractere menos quebra de linha	$a.*b$
\wedge	o início de uma linha	$\wedge abc$
$\$$	o fim de uma linha	$abc\$$
$[s]$	qualquer um dos caracteres na cadeia s	$[abc]$
$[^s]$	qualquer caractere não presente na cadeia s	$[^abc]$
r^*	zero ou mais cadeias casando com r	a^*
r^+	uma ou mais cadeias casando com r	a^+
$r?$	zero ou um r	$a?$
$r\{m,n\}$	entre m e n ocorrências de r	$a[1,5]$
r_1r_2	um r_1 seguido por um r_2	ab
$r_1 \mid r_2$	um r_1 ou um r_2	$a \mid b$
(r)	o mesmo que r	$(a \mid b)$
r_1/r_2	R_1 quando seguido por r_2	$abc/123$

(retirado de [Aho, 2008])

Reconhecimento de *Tokens*

- Considere o fragmento de gramática:

– <i>stmt</i>	→	if <i>expr</i> then <i>stmt</i> else <i>stmt</i>		
–		if <i>expr</i> then <i>stmt</i>		ϵ
– <i>expr</i>	→	<i>term</i> relop <i>term</i>		<i>term</i>
– <i>term</i>	→	id		num

- ***Tokens*** são os **símbolos terminais** da gramática livre de contexto
- Devem ser **reconhecidos e retornados** pelo analisador léxico
 - **Palavras-chave** associadas a **if**, **then** e **else**
 - **Lexemas** que casam com os padrões de **relop**, **id** e **num**
- **Separadores (ws) e comentários** devem ser removidos (**caso especial**)
 - **Não retorna** *token* ao analisador sintático
 - Provoca a **reinicialização da análise léxica** a partir do caractere seguinte

Reconhecimento de *Tokens*

- Definições regulares dos padrões dos *tokens*:
 - *digito* → [0-9]
 - *digitos* → *digito*⁺
 - *letra* → [A-Za-z]
 - *num* → *digitos* (*.digitos*) ? (E[+ -] ? *digitos*) ?
 - *id* → [*letra* _] [*digito letra* _] *
 - *relop* → < | > | <= | >= | = | <>
 - *if* → if
 - *then* → then
 - *else* → else
 - *ws* → (' ' | \t | \n) *

Reconhecimento de *Tokens*

Lexemas	Nome do <i>Token</i>	Valor do Atributo
Qualquer <i>ws</i>	–	–
<i>if</i>	if	–
<i>then</i>	then	–
<i>else</i>	else	–
Qualquer <i>id</i>	id	Posição na tabela de símbolos
Qualquer <i>numero</i>	num	Posição na tabela de símbolos
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

- fonte: [Aho, 2008]

Reconhecimento de *Tokens*

Lexemas	Nome do <i>Token</i>	Valor do Atributo
Qualquer <i>ws</i>	–	–
<i>if</i>	if	–
<i>then</i>	then	–
<i>else</i>	else	–
Qualquer <i>id</i>	id	Posição na tabela de símbolos
Qualquer <i>numero</i>	num	Posição na tabela de símbolos
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

O operador encontrado
influenciará o código
que será gerado pelo
compilador

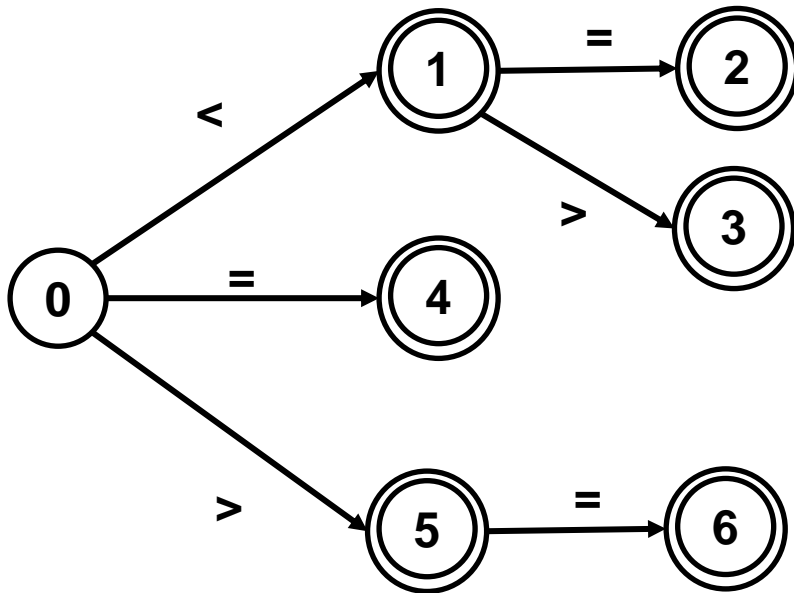
- fonte: [Aho, 2008]

Diagramas de Transição

- Fluxogramas usados no **reconhecimento** dos *tokens*
 - Muito parecido com os **autômatos finitos**
 - Gerados a partir dos **padrões** dos *tokens*
 - **Estados:** nós que representam as condições que podem ocorrer durante a procura de lexemas que casem com um padrão
 - Possui um **estado inicial** e um ou mais **estados finais ou de aceitação**
 - **Transições:** arestas direcionadas associadas à leitura de um ou + símbolos do alfabeto
 - Provoca **mudança de estado** e **avanço do prox**
 - **Estados de aceitação** indicam um **lexema aceito**
 - **NÃO** possuem transição de saída
 - Estão **associados às ações** que devem ser realizadas pelo analisador léxico
 - **Símbolo *** indica recuo do *prox* (**tratamento do lookahead**)

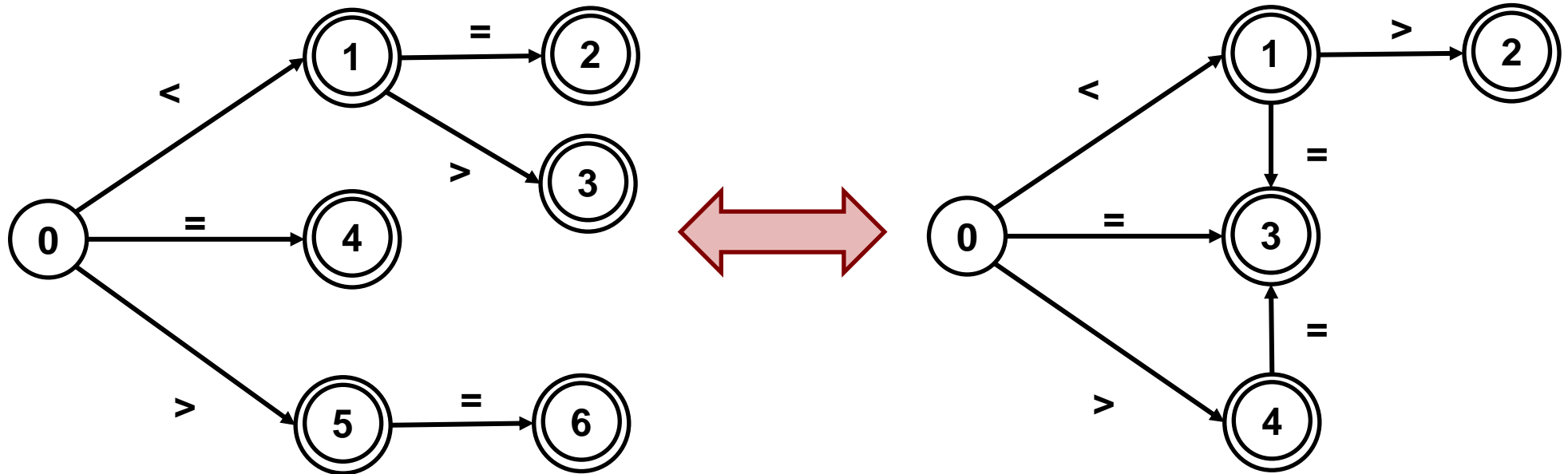
Autômato Finito Determinístico

- **Exemplo:** AFD para o *token relop*



Autômato Finito Determinístico

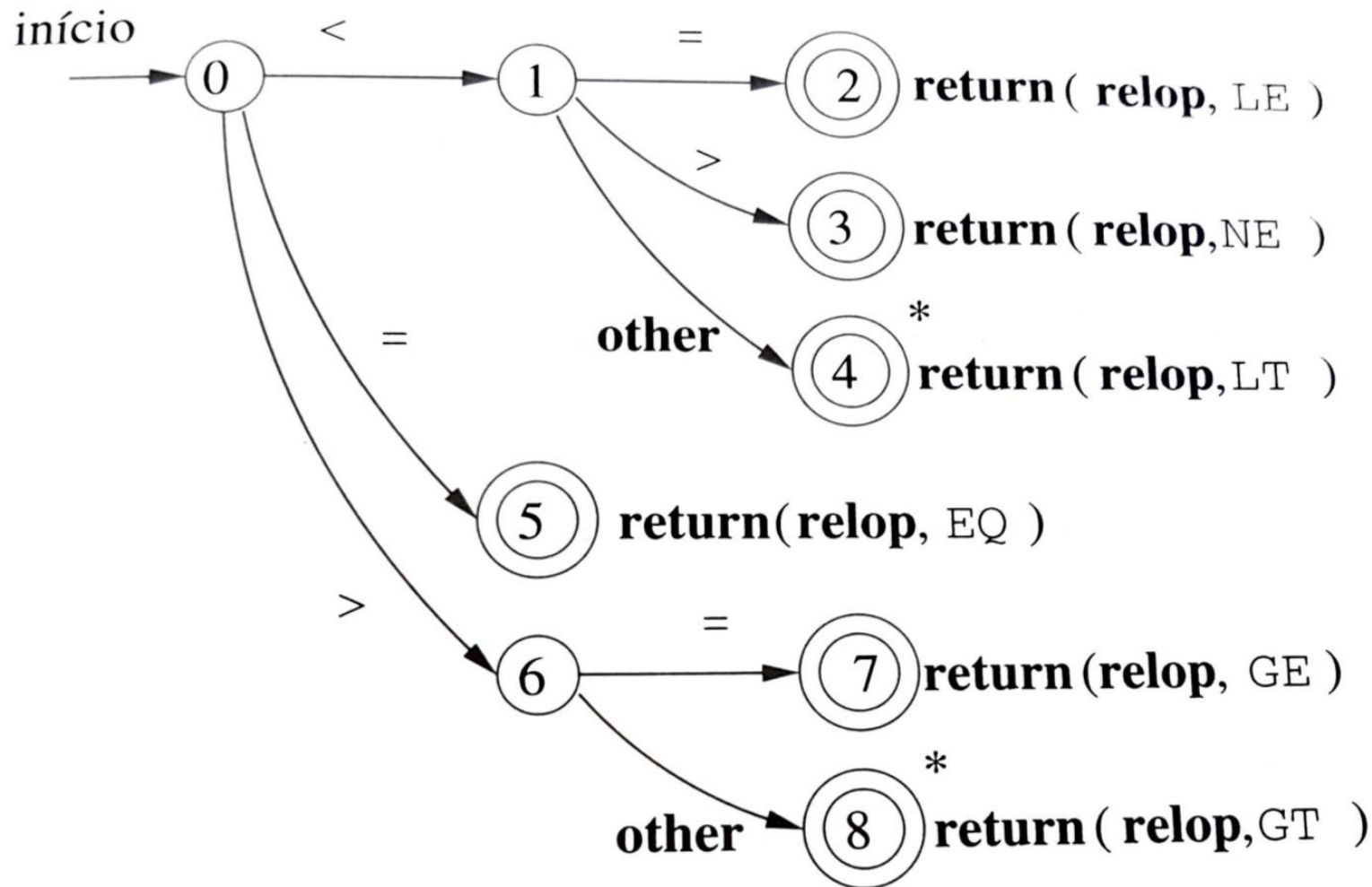
- **Exemplo:** AFD para o *token relop*



Versão simplificada

Diagramas de Transição

- **Exemplo:** diagrama para o *token relop*



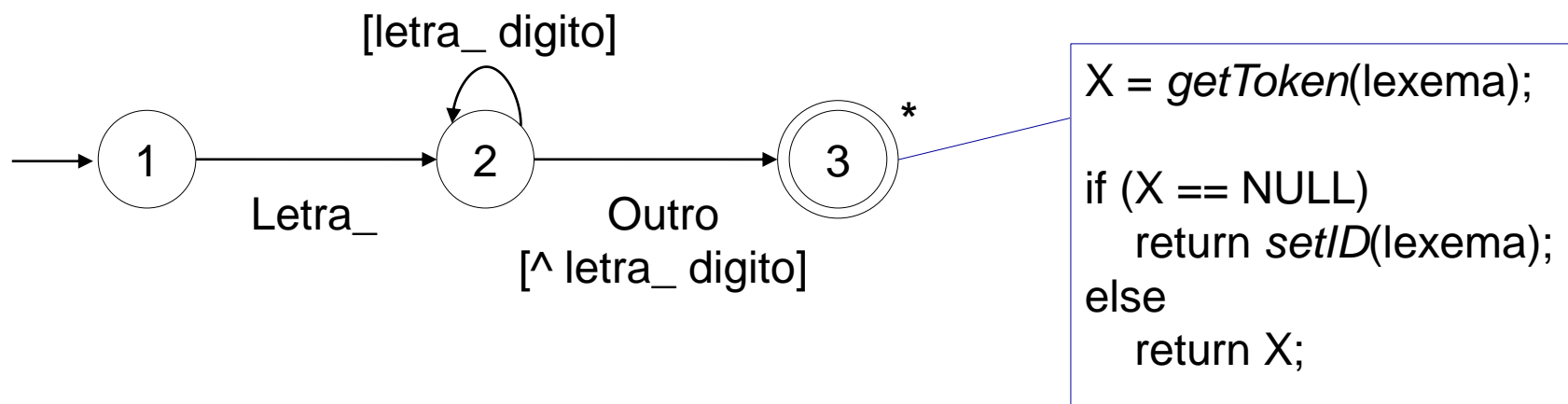
- fonte: [Aho, 2008]

Palavras-Chave vs. Identificadores

- Diferenciar **palavras-chave** e **identificadores** pode ser um problema
 - Palavras-chave casam com o padrão dos identificadores
 - Considerar as **palavras-chave** como **reservadas** ajuda no reconhecimento
 - **Palavras reservadas não podem ser identificadores**
- Existem 2 formas de lidar com palavras reservadas:
 - Tratar **palavras reservadas** como **identificador**
 - **Criar diagramas separados** para cada palavra-chave

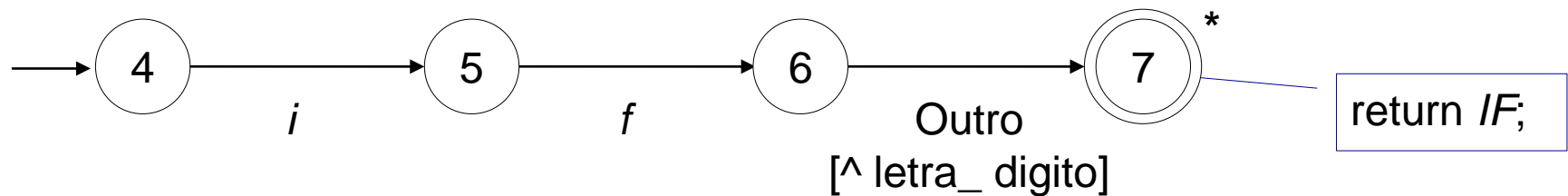
Palavras-Chave vs. Identificadores

- **Tratar palavras reservadas como identificador:**
 - Inicia tabela de símbolos com as palavras reservadas
 - Consulta a tabela antes de incluir um novo lexema
 - Se encontrar registro, retorna o **nome do token**
 - Senão inclui o novo lexema e retorna **ID**



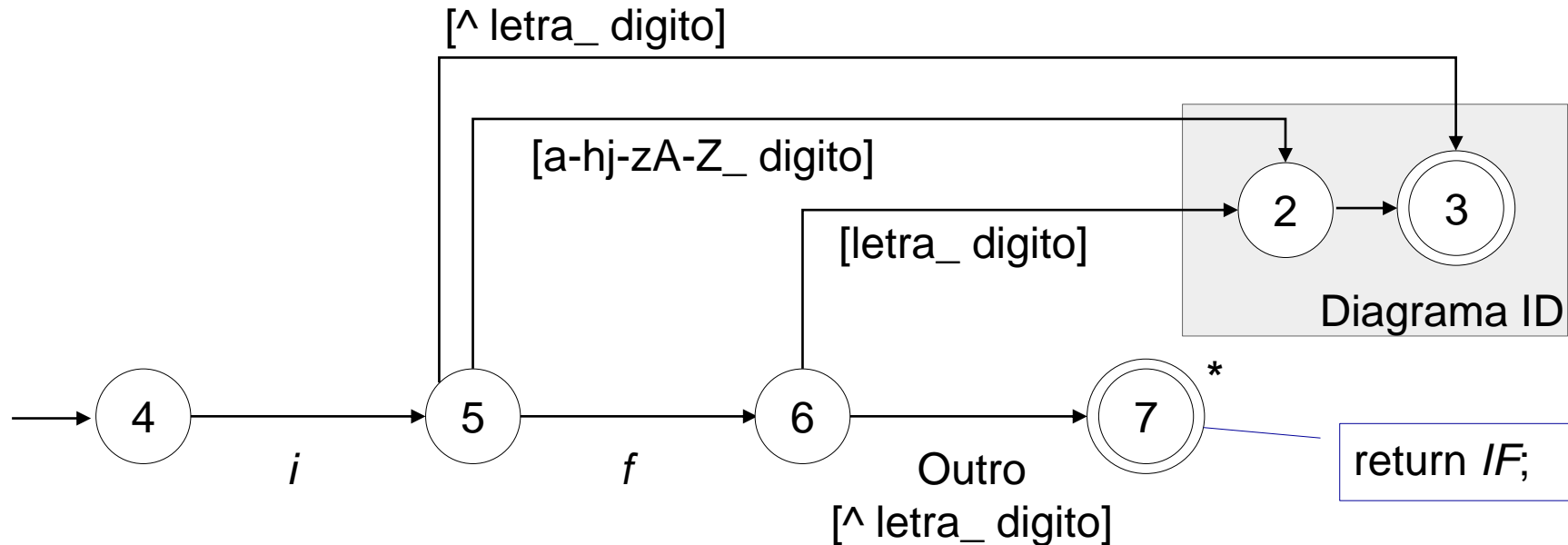
Palavras-Chave vs. Identificadores

- **Criar diagramas separados** para cada palavra chave:
 - **Verifica se a cadeia terminou** antes de associar o lexema ao *token*
 - Para evitar erros **prioriza o maior prefixo aceito**
 - **Ex:** *ifan* é um identificador e não a palavra-chave *if*
 - Se lexema casar com 2 padrões, deve priorizar a **palavra chave**



Palavras-Chave vs. Identificadores

- Variação que **UNIFICA** os diagramas do IF e do ID:
 - Unificação é uma das estratégias para lidar com vários diagramas durante o processo de reconhecimento



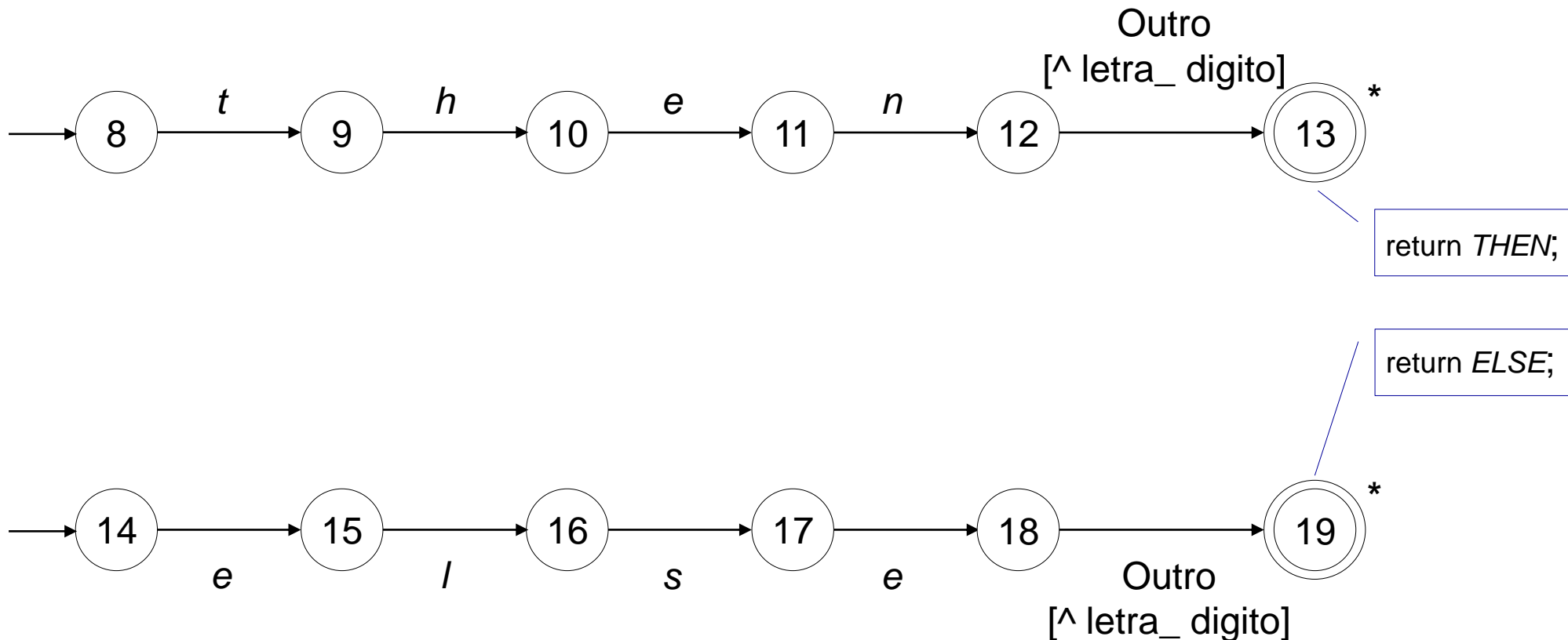
Diagramas de Transição

- Como seriam os diagramas de transição para os demais *tokens* do nosso estudo de caso?
 - **then, else, ws e num**

Diagramas de Transição

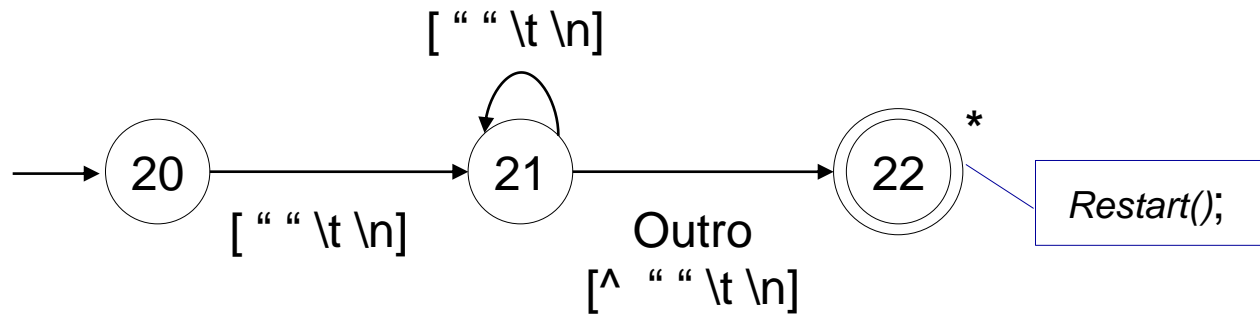
- **then e else**

- Já está sendo coberto pelo autômato da 1ª estratégia (**palavras reservadas como identificadores**)
- **2ª estratégia: similar ao diagrama do *if***



Diagramas de Transição

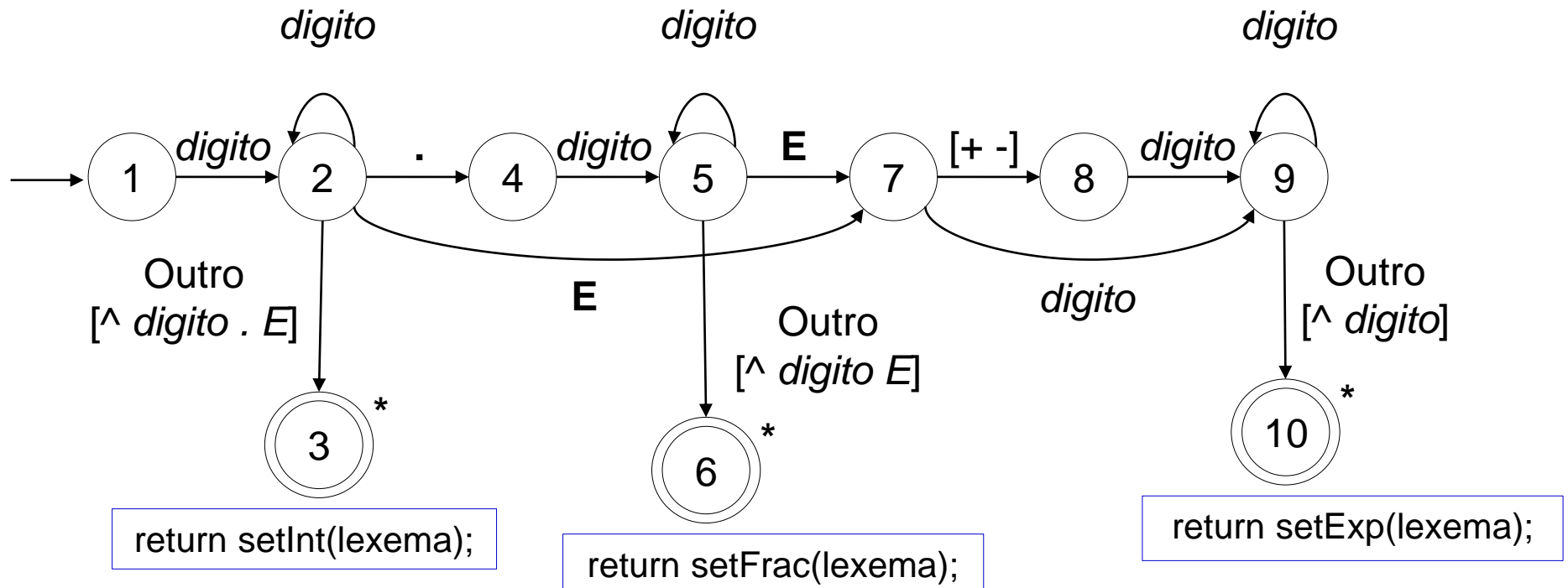
- **ws (separadores)**
 - Espaços em branco, tabulações e quebras de linha



Diagramas de Transição

- **Constante numérica**

- *num* → *digitos* (*.digitos*) ? (E[+ -] ? *digitos*) ?



Exercícios

- Faça o diagrama de transição para reconhecer as expressões regulares:
 - $(a|b)^* a (a|b)(a|b)$
 - $a^*ba^*ba^*ba^*$
 - $(aa|bb)^* ((ab|ba)(aa|bb)^* (ab|ba)(aa|bb)^*)^*$
- Defina a expressão regular e o diagrama de transição para lidar com comentários em C:
 - **Comentários de linha** (ex: *// comentário \n*)
 - **Comentários de blocos** (ex: */* comentário */*)

FORMAS DE IMPLEMENTAÇÃO

Tabela de Símbolos

- Forma de implementação da **estrutura de representação dos *tokens*** afeta a memória usada pela tabela de símbolos
 - Operadores podem ser representados por 2 caracteres
 - Palavras reservadas geralmente não são grandes
 - Também podem ser representadas por **códigos**
 - **Como lidar com identificadores, números e *strings*?**
 - **Alocação estática:**
 - **Vantagem:** implementação mais simples
 - **Desvantagem:** desperdício de memória
 - **Alocação dinâmica:**
 - **Vantagem:** uso otimizado de memória
 - **Desvantagem:** exige um controle mais elaborado

Tabela de Símbolos

- Desempenho do analisador é influenciado pela **eficiência da consulta** a tabela de símbolos
 - **Busca linear:**
 - Mais simples de implementar
 - Pior desempenho ($O(n)$)
 - **Busca binária:**
 - Boa eficiência ($O(\log n)$)
 - **Hashing:**
 - Ideal para consulta a palavras reservadas ($O(1)$)
 - Garante acesso **sem colisões** (palavras já são conhecidas)

Diagramas de Transição

- Formas de implementação **manual** de um código para **simular o comportamento de um diagrama**:
 - Solução *ad hoc*
 - Codificação direta do autômato finito
 - Uso de tabela de transição
- Implementação **automática** do código:
 - Muito usado em **projetos reais**
 - Geralmente adotam **métodos dirigidos por tabela**
 - **Ex:** Lex, JavaCC

Solução *Ad Hoc*

- Implementação focada no **fluxo de entrada**
 - **Simple e fácil**
- Estados do autômato são implementados **implicitamente**
 - **Aninhamento de IF's** define as **mudanças de estado**
 - Aninhamentos implementam a **derivação de um lexema aceito**
 - Mesma **transição é repetida** em diferentes partes do código
- Complexidade do código cresce com o **Nro. de estados**
 - Aplicável a autômatos com **poucos estados**
- **Implementação personalizada**
 - Código sensível a mudanças no autômato

Exemplo (Ad Hoc)

Início **{estado 0}**

$\text{char } c \leftarrow \text{prox_char}()$

se ($c = \text{"b"}$) então

$c \leftarrow \text{prox_char}()$

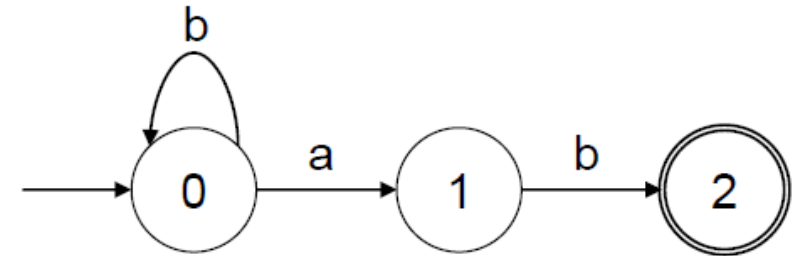
enquanto ($c = \text{"b"}$) faça

$c \leftarrow \text{prox_char}()$

fim enquanto

senão

Fim



{* bloco repetido ***}**

se ($c = \text{"a"}$) então **{estado 1}**

$c \leftarrow \text{prox_char}()$

se ($c = \text{"b"}$) então **{estado 2}**

Faz as ações previstas no estado 2

senão

retorna erro("b era esperado")

senão

retorna erro("a era esperado")

Exemplo (*Ad Hoc*)

Início **{s0}**

$\text{char } c \leftarrow \text{prox_char}()$

enquanto (**$c = \text{"b"}$**) faça

$c \leftarrow \text{prox_char}()$

fim enquanto

se (**$c = \text{"a"}$**) então **{s1}**

$c \leftarrow \text{prox_char}()$

se (**$c = \text{"b"}$**) então **{s2}**

Faz as ações previstas no estado 2

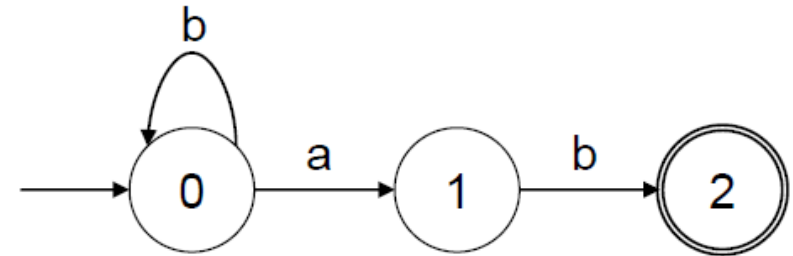
senão

retorna erro("b era esperado")

senão

retorna erro("a era esperado")

Fim



***Variação do código
SEM repetição***

Codificação Direta

- Reflete diretamente o **autômato finito**
- Estados do autômato são implementados **explicitamente**
 - **Estado atual** é armazenado em uma **variável**
- Incorpora as **transições** por meio de **estruturas de seleção**
 - **Verificação externa** seleciona o **estado corrente**
 - Trata cada um dos possíveis estados do autômato
 - **Verificação interna** trata o **símbolo lido**
 - Implementa as ações de um estado
 - Envolve mudanças de estado e leitura do próximo caractere da entrada
- Ainda é uma **implementação personalizada**
 - Código permanece sensível a mudanças no autômato

Exemplo (Codificação Direta)

Início

$s \leftarrow 0$ {estado 0}

enquanto (TRUE) faça

caso (s) seja

0: $c \leftarrow \text{prox_char}()$

 se ($c = a$) então

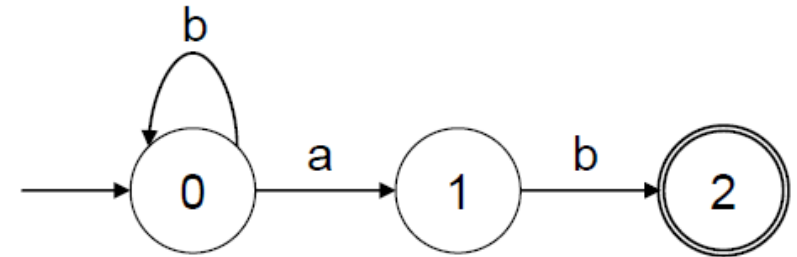
$s \leftarrow 1$ {estado 1}

 senão

 se ($c \neq b$) então

 retorna erro($c + \text{"inesperado"}$)

...



...

1: $c \leftarrow \text{prox_char}()$

 se ($c = b$) então

$s \leftarrow 2$ {estado 2}

 senão

 retorna erro("b era esperado")

2:

 Faz as ações previstas no estado 2

 fim_caso

fim_enquanto

Fim

Exemplo (Codificação Direta)

// Código para o diagrama do RELOP

*Token *getRelop() {*

// Processa entrada até encontrar token ou falhar

while (1) {

switch (state) {

case 0 : { c = nextChar();

if (c == '<') state = 1;

else if (c == '=') state = 5;

else if (c == '>') state = 6;

else fail(); // Lexema não é um relop

break; }

case 1 : {...}

...

RELOP e GT são códigos
definidos por **#define ou enum**

Retirado de [Aho, 2008]

...

case 8 : {

trata_Lookahead();

*Token * tk;*

*tk = (Token *) malloc(sizeof(Token));*

tk.nome = RELOP;

tk.atributo = GT;

return tk; }

} // Fim do switch

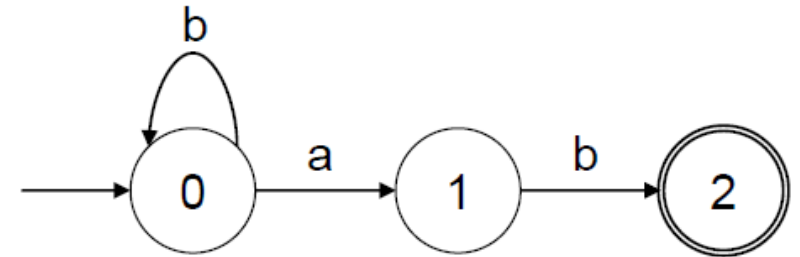
} // Fim do while

} // Fim da função

Tabela de Transição

- Autômato representado por uma **estrutura de dados**
 - **Tipo Abstrato de Dados (TAD)**
 - **Atributos:** descrevem as características do autômato finito
 - **Conjunto de transições** representado por uma **tabela**
 - Deve indicar os **estados finais**
 - Representa as **entradas não previstas (erros)** em um estado
 - **Operações:** possibilitam acesso aos atributos do TAD
- Permite a implementação de um **código genérico**
 - Especificidades do autômato estão **restritas ao TAD**

Exemplo (Tabela de Transição)



Modelo Conceitual

	Estados	Símbolos Entrada	
		<i>a</i>	<i>b</i>
→	0	{1}	{0}
	1	—	{2}
*	2	—	—

Função completa: usa estado extra (**estado de erro**) para tratar entradas não previstas

Possível Implementação

Estado	Final	Símbolos Entrada		
		<i>a</i>	<i>b</i>	<i>outros</i>
0	N	{1}	{0}	{-1}
1	N	{-1}	{2}	{-1}
2	S	{-1}	{-1}	{-1}
-1	S	{-1}	{-1}	{-1}

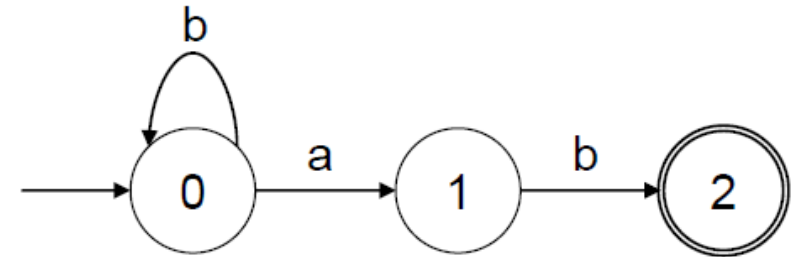
$S_0 = 0$

Retirado de [Aluisio, 2011]

Exemplo (Tabela de Transição)

Início

$s \leftarrow \text{estado_inicial}()$ {retorna o estado inicial}



enquanto (**final(s)** = 'N') faça

$c \leftarrow \text{prox_char}()$

{ Define o próximo estado (transição) }

$s \leftarrow \text{move}(s, c)$

fim enquanto

{ Executa as ações dos estados finais
e trata erro nos demais estados }

acoes(s)

Fim

Possível Implementação

Estado	Final	Símbolos Entrada		
		<i>a</i>	<i>b</i>	<i>outros</i>
0	N	{1}	{0}	{-1}
1	N	{-1}	{2}	{-1}
2	S	{-1}	{-1}	{-1}
-1	S	{-1}	{-1}	{-1}

$S_0 = 0$

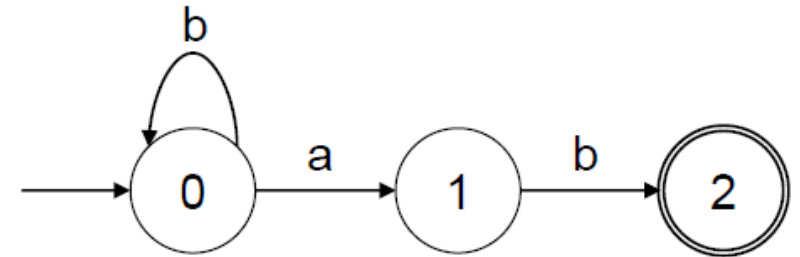
**Métodos do TAD que proporcionam um
algoritmo de reconhecimento GENÉRICO**

Retirado de [Aluisio, 2011]

Exemplo (Tabela de Transição)

Início

$s \leftarrow \text{estado_inicial}()$ {retorna o estado inicial}



enquanto (**final**(s) = 'N') faça

$c \leftarrow \text{prox_char}()$

{ Define o próximo estado (transição) }

$s \leftarrow \text{move}(s, c)$

fim enquanto

{ Executa as ações dos estados finais
e trata erro nos demais estados }

acoes(s)

Fim

Possível Implementação

Estado	Final	Símbolos Entrada		
		<i>a</i>	<i>b</i>	<i>outros</i>
0	N	{1}	{0}	{-1}
1	N	{-1}	{2}	{-1}
2	S	{-1}	{-1}	{-1}
-1	S	{-1}	{-1}	{-1}

$S_0 = 0$

**Código funciona bem se o autômato
finito for determinista (AFD)**

Retirado de [Aluisio, 2011]

Tabela de Transição

- **Vantagens:**

- Código reduzido (**elegância**)
- Mesmo código atende a várias linguagens (**generalidade**)

- **Desvantagem:**

- **Tabelas estáticas:** pode demandar/desperdiçar **muito espaço de memória**
 - Alfabeto grande ou autômato com muitos estados no autômato
- **Tabelas dinâmicas:** torna o **processamento mais lento**

Análise Baseada em Diagramas

- Analisador léxico precisa **executar o código de todos os diagramas** para determinar o *token* a ser retornado
- **Possíveis estratégias:**
 - Executar os diagramas **em sequência**
 - Executar os diagramas **em paralelo**
 - **Combinar todos os diagramas** em um único

Execução Sequencial

- *Tokens* são verificados em sequência
 - Permite **diagramas específicos** para palavras-chave
 - Ordem de execução define a **prioridade de reconhecimento** entre os *tokens*
 - **Ex:** palavras-chave vs. identificadores
- **Falha** no percorrimento de um diagrama provoca **troca de diagrama**
 - Releitura de caracteres (reinicialização do campo ***prox***)
 - Inicialização do próximo diagrama

Execução Paralela

- *Tokens* são analisados ao mesmo tempo
 - Caractere é tratado por **todos diagramas ativos**
 - **Diagrama ativo:** reconhece o prefixo lido até o momento
 - Cada diagrama controla suas transições
 - **Se** símbolo está previsto **então** realiza a transição
 - **Senão** desativa o diagrama
- Precisa de regras para **tratar ambiguidades**
 - **Define o que fazer** quando um padrão é atendido mas ainda existem outros diagramas ativos
 - **Ex:** *thenext* ou *then* ? “->” ou “-” ? “<=” ou “<” ?
 - Estratégia mais usual é **priorizar lexema mais longo**
 - Uso de **palavra reservada** para tratar palavras-chave

Diagrama Combinado

- Todos os diagramas representados em um só
- Diagrama gerado deve:
 - Ler entrada até não existir transição possível
 - Retornar **lexema mais longo** que casou com um padrão
- Tarefa é geralmente complexa:
 - **Problema:** tratar diagramas com a mesma transição
 - **Solução:** construir um autômato finito não determinístico com transições ϵ (AFND ϵ) e convertê-lo no AFD equivalente

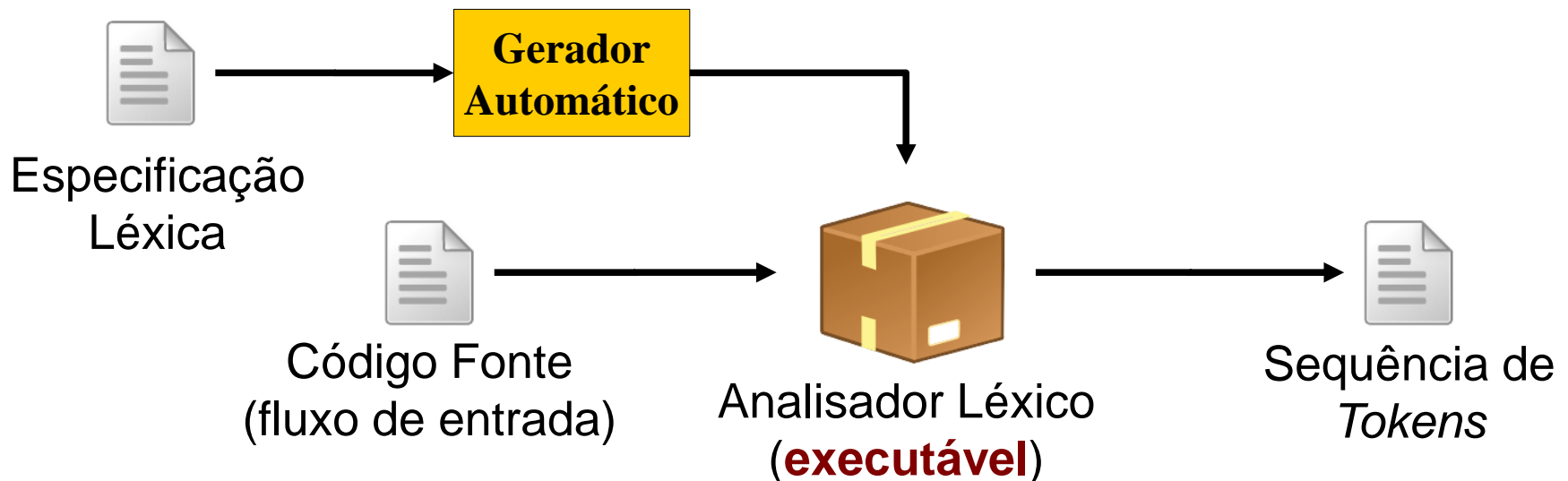
GERADORES AUTOMÁTICOS DE ANALISADORES LÉXICOS (LEX / FLEX)

Geradores Automático

- **Gerador de analisadores léxicos** [Alexandre, 2014]:

*“Programa que **recebe como entrada** a **especificação léxica** para uma linguagem e **produz como saída** um programa que faz a **análise léxica** dessa linguagem”*

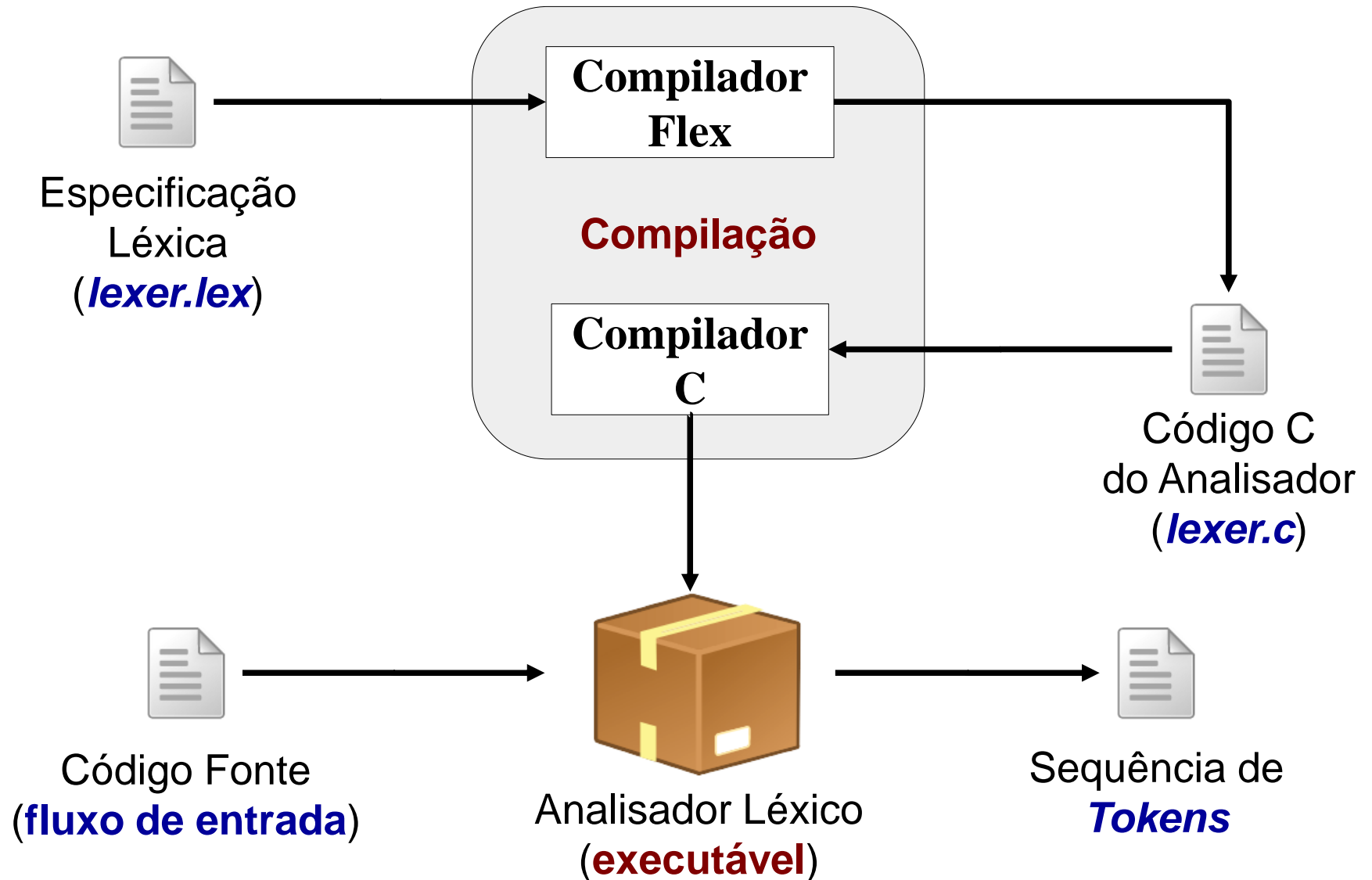
- Por que usar um gerador de analisadores?
 - **Menos trabalho**
 - Garante a geração de um analisador léxico **correto**



Gerador Automático Flex

- **Flex** (*Fast Lexical Analyzer Generator*)
 - Versão GNU do gerador Lex (**software livre**)
- **Especificação léxica** envolve a descrição dos padrões para os *tokens* (**expressões regulares**)
 - Usa notação BNF da **linguagem Lex** (arquivo *.l* ou *.lex*)
- **Compilador Lex** gera um **programa que simula o diagrama de transição** criado a partir dos padrões especificados
 - Programa gerado em **linguagem C**
 - Nome padrão do arquivo de saída: **lex.yy.c**
 - Sintaxe: ***flex -o lexer.c lexer.lex***
- **Compilador C** usado para gerar o **código executável**
 - Ex: ***gcc -o lexer lexer.c***

Gerador Automático Flex



Estrutura de Programas Lex

- Especificação é dividida em 3 partes:

Declarações

%%

Regras de tradução

%%

Código

Estrutura de Programas Lex

- **Declarações:** especifica variáveis globais, constantes manifestas e definições regulares
 - **Constantes manifestas:**
 - Identificadores para constantes (**ex:** nome dos *tokens*)
 - Conjunto de diretivas **#define** do C
 - **Definições regulares:**
 - Expressões regulares usadas como símbolos pelos padrões
 - Variáveis e constantes manifestas são declaradas dentro de **delimitadores especiais** **%{ e %}**
 - Copia o conteúdo diretamente para o arquivo **lex.yy.c**

Estrutura de Programas Lex

- **Regras de tradução:** especifica os padrões e suas ações
 - **Parte principal** da especificação léxica (**única obrigatória**)
- **Sintaxe:** *padrão { ação }*
 - **Padrão:** expressão regular que descreve um *token* da linguagem
 - **Ação:** fragmento de código C que determina o que fazer quando um lexema casa com o padrão especificado
 - **Mais comum:** instanciar um *token* para o lexema encontrado
 - Pode utilizar funções auxiliares descritas na seção código ou arq. externo
- **Código:** especifica funções auxiliares usadas nas ações
 - Também podem ser especificadas em **arquivos separados**
 - Geralmente usadas para **manipular a tabela de símbolos**

Exemplos de Padrões do Flex

Padrão

Padrão	Descrição Informal
[0-9]	Qualquer dígito entre 0 e 9
[0+9]	0, + ou 9
[0,9]	0, ‘, ‘ ou 9
[0 9]	0, ‘ ‘ ou 9
[-0-9]	- ou qualquer dígito entre 0 e 9
[0-9]+	1 ou mais dígitos entre 0 e 9
a?	Indica opcionalidade: (a ϵ)
[^a]	Qualquer caractere, exceto ‘a’
[^A-Z]	Qualquer outro caractere, exceto as letras maiúsculas
^aba	Ocorrência de 'aba' no início de uma linha
aba\$	Ocorrência de 'aba' no final de uma linha
a{2, 4}	‘aa’, ‘aaa’ ou ‘aaaa’
a{2, }	2 ou mais ocorrências de ‘a’
a{4}	Exatamente 4 a’s, ou seja, ‘aaaa’
.	Qualquer caractere, exceto <i>newline</i>
a*	0 ou mais ocorrências de ‘a’
a+	1 ou mais ocorrências de ‘a’
[a-z]	Qualquer letra minúscula
[a-zA-Z]	Qualquer letra do alfabeto (minúscula ou maiúscula)
w(x y)z ou w[xy]z	‘wxz’ ou ‘wyz’
\□, com □ ∈ {., ^, +, ?, etc.}	Ocorrência do caractere □ no padrão
\□, com □ ∈ {0, n, t, etc.}	Ocorrência de caractere especial do C (\0, \n, \t, etc.)
<<EOF>>	fim de arquivo

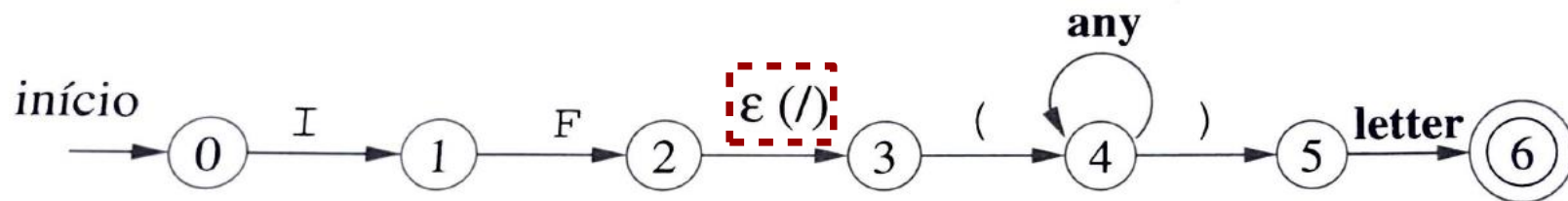
Características do Lex

- Quando **nenhum padrão for casado**, uma regra padrão do Flex **imprime os caracteres não reconhecidos**
- **Precedência na solução de conflitos:**
 1. Preferência pelo **prefixo mais longo**
 2. **Ordem de descrição dos padrões** no programa Lex
- **Operador *lookahead*:**
 - **Padrão:** sempre **lê o próximo caractere** após o casamento do padrão **e depois recua a entrada** para consumir apenas o lexema
 - O que fazer quando o casamento exige que o padrão seja seguido por um certo conjunto de caracteres?
 - **Solução:** uso da barra condicional (/)
 - Inclui um **padrão adicional** após a barra que **não faz parte do lexema**

Características do Lex

- **Exemplo:** **IF / \ (.* \) {letter}**

- Padrão permite diferenciar o comando *IF* de uma variável indexada *IF(x,y)* no Fortran
 - $IF(I,J) = 3$
 - *IF (condição) THEN*
- Necessita que a entrada seja pré-processada para a remoção dos espaços em branco



AFND do IF

Integração com o Programa Usuário

- O analisador léxico gerado a partir de um programa *lex* possui as seguintes características:
 - Usa a rotina ***yylex()*** para chamada do analisador pelo usuário
 - Não possui argumento de entrada e retorna um **inteiro** (padrão)
 - Retorno geralmente é associado pelo usuário ao **tipo do token**
 - Tipo do retorno pode ser alterado por um **tipo definido pelo usuário**
 - Interface realizada por **2 variáveis globais do tipo FILE**:
 - Entrada é lida do arquivo endereçado por ***yyin***
 - Padrão é o ponteiro ***stdin*** (teclado)
 - Resultado é enviado para o arquivo endereçado por ***yyout***
 - Padrão é o ponteiro ***stdout*** (tela)
 - Ambas podem ser modificadas na **seção de código do prog. *lex***
 - Variável ***yytext*** aponta para o último lexema reconhecido
 - **Variável global** do tipo ponteiro para caracteres (***string***)

Exemplo de um Programa Flex

Fonte: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

/ **Declare Section** has one variable which can be accessed inside yylex() and main() */*

```
%{  
    int count = 0;  
}%
```

%%

/ **Rule Section** has 3 rules:*

- 1- matches with capital letters
- 2- matches with any character except newline
- 3- does not take input after the enter */

```
[A-Z] {printf("%s capital letter\n", yytext); count++;}  
.  
\n    {printf("%s not a capital letter\n", yytext);}  
    {return 0;}
```

%%

/ **Code Section** prints the number of capital letter present in the given input */*

```
int yywrap(){}  
  
int main(){
```

```
    // Code to take input from file  
    // FILE *fp;  
    // char filename[50];  
    // printf("Enter the filename: \n");  
    // scanf("%s",filename);  
    // fp = fopen(filename,"r");  
    // yyin = fp;
```

```
    yylex(); // lexer call (runs Rule Section)
```

```
    printf("\nNumber of Capital letters "  
        "in the given input - %d\n", count);
```

```
    return 0;
```

```
}
```

Exemplo de um Programa Flex

Fonte: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

/ **Declare Section** has one variable which can be accessed inside yylex() and main() */*

```
%{  
    int count = 0;  
}%
```

%%

/ **Rule Section** has 3 rules:*

- 1- matches with capital letters
- 2- matches with any character except newline
- 3- does not take input after the enter */

```
[A-Z] {printf("%s capital letter\n", yytext); count++;}  
.  
\n    {printf("%s not a capital letter\n", yytext);  
    {return 0;}}
```

%%

- **yytext**: o lexema encontrado
- **yyin**: ponteiro de arquivo para a entrada
- **yylex()**: função principal do flex
- **yywrap()**: função que define o que fazer ao encontrar EOF

/ **Code Section** prints the number of capital letter present in the given input */*

```
int yywrap(){}  
  
int main(){
```

```
  
    // Code to take input from file  
    // FILE *fp;  
    // char filename[50];  
    // printf("Enter the filename: \n");  
    // scanf("%s",filename);  
    // fp = fopen(filename,"r");  
    // yyin = fp;
```

```
yylex(); // lexer call (runs Rule Section)
```

```
printf("\nNumber of Capital letters in the  
given input - %d\n", count);
```

```
return 0;
```

```
}
```

Exemplo de um Programa Flex

```
shivani@workspace:~/Desktop/Compiler$ lex rough.l
shivani@workspace:~/Desktop/Compiler$ gcc lex.yy.c
shivani@workspace:~/Desktop/Compiler$ ./a.out
GFG123gfg
G capital letter
F capital letter
G capital letter
1 not capital letter
2 not capital letter
3 not capital letter
g not capital letter
f not capital letter
g not capital letter

Number of Captial letters in the given input - 3
```

Fonte: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

Exemplo de um Programa Flex

Fonte: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

```
/** Declare Section**/
```

```
%option noyywrap
```

```
/* Define 2 counters */
```

```
{
```

```
int nLines = 0; /* Nr. of lines */
```

```
int nChars = 0; /* Nr. of lines */
```

```
}
```

```
%% /** Rule Section **/
```

```
\n ++nLines; /* counts lines */
```

```
. ++nChars; /* counts chars */
```

```
end return 0; /* stop */
```

```
%% /** User Section**/
```

```
int main(int argc, char **argv) {
```

```
yylex(); // lexer call
```

```
printf("Lines = %d, Chars = %d\n",  
nLines, nChars );
```

```
return 0;
```

```
}
```

Exemplo de um Programa Flex

Fonte: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

```
/** Declare Section**/
```

```
%option noyywrap
```

```
/* Define 2 counters */
```

```
{
```

```
int nLines = 0; /* Nr. of lines */
```

```
int nChars = 0; /* Nr. of lines */
```

```
}
```

```
%% /** Rule Section **/
```

```
\n ++nLines; /* counts lines */
```

```
. ++nChars; /* counts chars */
```

```
end return 0; /* stop */
```

```
%% /** User Section**/
```

```
int main(int argc, char **argv) {
```

```
yylex(); // lexer call
```

```
printf("Lines = %d, Chars = %d\n",  
nLines, nChars );
```

```
return 0;
```

```
}
```

- Indica que **yywrap()** não será fornecida
 - Irá parar quando encontrar EOF
- Similar ao **uso do flag -lfl** na compilação
 - Ligação com versão padrão da função **yywrap()** que sempre retorna 1

Exemplo de um Programa Flex

```
shubham@gfg-desktop:~$ lex lines.l
shubham@gfg-desktop:~$ gcc lex.yy.c
shubham@gfg-desktop:~$ ./a.out
Geeks
for
Geeks
end
number of lines = 3, number of chars = 13
shubham@gfg-desktop:~$
```

Fonte: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>

Analizador Léxico no Flex

- **Ex:** analisador para expressões [Alexandre, 2014]
 - **Codificação em 3 arquivos (projeto de TAD)**
 - **Arquivo cabeçalho (*exp.h*):**
 - Contém as definições de tipos, das constantes e dos protótipos das funções
 - **Arquivo do Flex (*exp.lex*):**
 - Contém as **especificações dos padrões**
 - **Arquivo de usuário (*exp.c*):**
 - Simula o analisador sintático
 - Responsável pela **chamada do analisador léxico**

Analizador Léxico no Flex

Fonte: [Alexandre, 2014]

// Arquivo exp.h

// constantes booleanas

#define TRUE 1

#define FALSE 0

// constantes para nome de token

#define TOK_NUM 0

#define TOK_OP 1

#define TOK_PONT 2

#define TOK_ERRO 3

// Atributos do TOK_OP

#define SOMA 0

#define SUB 1

#define MULT 2

#define DIV 3

// Atributos do TOK_PONT

#define PARESQ 0

#define PARDIR 1

// estrutura de um token

typedef struct {

int tipo;

int atributo;

} Token;

// funcao para criar um token

*extern Token *token();*

// funcao do analisador lexico

*extern Token *yylex();*

Analizador Léxico no Flex

Fonte: [Alexandre, 2014]

// Arquivo *exp.h*

// constantes booleanas

#define TRUE 1

#define FALSE 0

// constantes para nome de token

#define TOK_NUM 0

#define TOK_OP 1

#define TOK_PONT 2

#define TOK_ERRO 3

// Atributos do TOK_OP

#define SOMA 0

#define SUB 1

#define MULT 2

#define DIV 3

// Atributos do TOK_PONT

#define PARESQ 0

#define PARDIR 1

// estrutura de um token

typedef struct {

int tipo;

int atributo;

} Token;

// funcao para criar um token

extern Token *token();

// funcao do analisador lexico

extern **Token** *yylex();

Mudança do tipo
de retorno padrão



Analizador Léxico no Flex

Fonte: [Alexandre, 2014]

/ Arquivo exp.lex */*

%option noyywrap

%option nodefault

%option outfile="lexer.c" header-file="lexer.h"

%{ #include "exp.h" %}

NUM [0-9]

%%

[[::space:]] { } /* ignora espaços */

{NUM}+ { return token(TOK_NUM, atoi(yytext)); }

\+ { return token(TOK_OP, SOMA); }

- { return token(TOK_OP, SUB); }

* { return token(TOK_OP, MULT); }

\/ { return token(TOK_OP, DIV); }

\({ return token(TOK_PONT, PARESQ); }

\) { return token(TOK_PONT, PARDIR); }

/ Tratamento para token desconhecido */*

. { return token(TOK_ERRO, 0); }

%%

// variavel global para um token

Token tok;

Token * token (int tipo, int valor) {

 tok.tipo = tipo;

 tok.atributo = valor;

 return &tok;

}

Analizador Léxico no Flex

Fonte: [Alexandre, 2014]

/ Arquivo exp.lex */*

%option noyywrap

%option nodefault

%option outfile="lexer.c" header-file="lexer.h"

%{ #include "exp.h" %}

NUM [0-9]

%%

[[::space:]] { } /* ignora espaços */

{NUM}+ { return token(TOK_NUM, atoi(yytext)); }

\+ { return token(TOK_OP, SOMA); }

- { return token(TOK_OP, SUB); }

* { return token(TOK_OP, MULT); }

\ / { return token(TOK_OP, DIV); }

\({ return token(TOK_PONT, PARESQ); }

\) { return token(TOK_PONT, PARDIR); }

/ Tratamento para token desconhecido */*

. { return token(TOK_ERRO, 0); }

%%

// variavel global para um token

Token tok;

Token * token (int tipo, int valor) {

 tok.tipo = tipo;

 tok.atributo = valor;

 return &tok;

}

Analizador Léxico no Flex

Fonte: [Alexandre, 2014]

// Arquivo *exp.c*

```
#include "lexer.h"
```

```
#include "exp.h"
```

```
/* Le uma string como entrada */
```

```
YY_BUFFER_STATE buffer;
```

```
void inicializa(char *str) {
```

```
    buffer = yy_scan_string(str);
```

```
}
```

```
Token *proximo_token() {
```

```
    return yylex();
```

```
}
```

```
void imprime_token( Token *tok) {
```

```
...
```

```
}
```

```
int main(int argc, char **argv) {
```

```
    char entrada[200];
```

```
    Token *tok;
```

```
    printf("\nAnalise Lexica da expressao: ");
```

```
    fgets(entrada, 200, stdin);
```

```
    inicializa(entrada);
```

```
    tok = proximo_token();
```

```
    while (tok != NULL) {
```

```
        imprime_token(tok);
```

```
        tok = proximo_token();
```

```
    }
```

```
    return 0; }
```

Analizador Léxico no Flex

Fonte: [Alexandre, 2014]

// Arquivo *exp.c*

```
#include "lexer.h" // biblioteca do lex
```

```
#include "Expressao.h"
```

```
/* Le uma string como entrada */
```

```
YY_BUFFER_STATE buffer;
```

```
void inicializa(char *str) {
```

```
    buffer = yy_scan_string(str);
```

```
}
```

```
Token *proximo_token() {
```

```
    return yylex();
```

```
}
```

```
void imprime_token( Token *tok) {
```

```
...
```

```
}
```

```
int main(int argc, char **argv) {
```

```
    char entrada[200];
```

```
    Token *tok;
```

```
    printf("\nAnalise Lexica da expressao: ");
```

```
    fgets(entrada, 200, stdin);
```

```
    inicializa(entrada);
```

```
    tok = proximo_token();
```

```
    while (tok != NULL) {
```

```
        imprime_token(tok);
```

```
        tok = proximo_token();
```

```
    }
```

```
    return 0; }
```

Analizador Léxico no Flex

- **Compilação no Flex:**

- Gera os arquivos do analisador léxico (*lexer.c* e *lexer.h*)
- Precisa indicar que está sendo usada uma declaração diferente para a função principal do analisador (*yylex*)
- **Comando:**

*flex -DYY_DECL="Token * yylex()" exp.lex*

- **Compilação no C:**

- Compila os arquivos dos analisadores léxico e “sintático”
- Carrega os códigos objeto para gerar o executável
- **Exemplo do comando:**

gcc -o exp lexer.c exp.c

Analizador Léxico no Flex

- Entrada do Flex é um ponteiro para arquivo (***yyin***)
 - Valor é associado ao *stdin* (entrada padrão)
- Leitura de um **arquivo externo** é feita a partir da atribuição do seu endereço à variável ***yyin***

```
int inicializa (char *nome) {  
  
    FILE *f = fopen(nome, "r");  
  
    if (f == NULL)  
        return FALSE;  
    yyin = f;  
    return TRUE;  
}
```

Exercícios

- 1) Implemente manualmente um analisador léxico para cada um dos *tokens* utilizados para explicar os diagramas de transição, como segue:
 - **Palavras reservadas + Identificador:** *ad hoc*, autômato, tabela
 - **Palavras-chave (específico):** autômato, tabela
 - **Número:** autômato, tabela
 - **Relop:** *ad hoc*, tabela
 - **Separadores (ws):** *ad hoc*, tabela
 - **Comentários:** *ad hoc*, autômato, tabela
- 2) Junte as implementações (tabela) para criar um **único analisador léxico**. Faça um programa de teste que faz o papel do analisador sintático, ou seja, solicita um novo *token* e o apresenta na tela. Verifique se a sequência de *tokens* está de acordo com o código fonte dado como entrada
- 3) Escreva um programa em Flex que reconhece os padrões de CPF e e-mail. Como ação, ele deve indicar se foi digitado um CPF, um e-mail ou algo desconhecido
- 4) Gere um analisador léxico para os *tokens* do item 1 utilizando a ferramenta Flex

CONSTRUÇÃO DO ANALISADOR LÉXICO

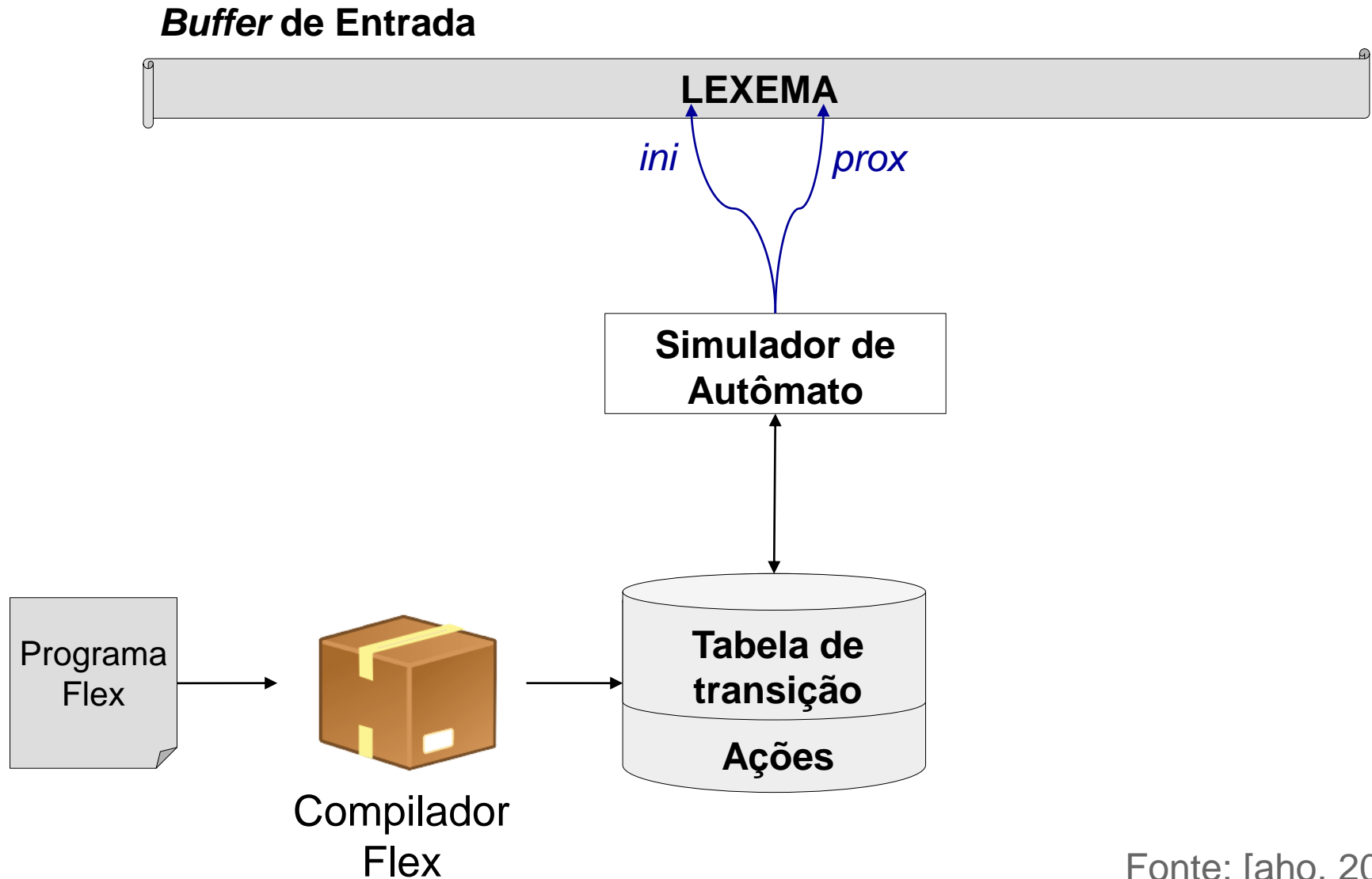
Geração Automática

- Geradores (Flex) **convertem as especificações léxicas em um diagrama de transição**
 - **Especificação:** usa notação para definir os padrões dos *tokens* (**expressões regulares**)
 - **Implementação:** simulação de **AFD** ou **AFND- ϵ**
- **AFND** é uma **representação abstrata** do algoritmo reconhecedor, enquanto que o **AFD** é um **algoritmo simples e concreto**
 - **AFND** facilita a geração do diagrama (**conversão**)
 - **AFD** facilita a simulação (**reconhecimento**)

Geração Automática

- Analisador léxico do Flex consiste:
 - Um **programa fixo** que **simula um autômato (genérico)**
 - Componentes criados a partir do programa Lex (**específico**):
 - **Tabela de transição** para o autômato
 - **Ações** a serem invocadas pelo simulador do autômato
 - **Funções auxiliares** passadas diretamente para a saída
- **Processo mecânico e sistêmico:**
 - Conversão das expressões regulares em AFNDs
 - Unificação dos AFNDs em um único autômato
 - Conversão do AFND em um AFD
 - Minimização do número de estados
 - Simulação do AFD
- Alternativa é **simular diretamente o AFND** gerado

Geração Automática



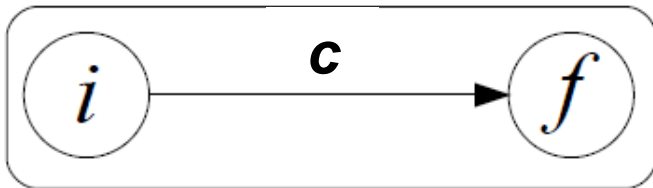
Fonte: [aho, 2008]

Conversão Expressões Regulares

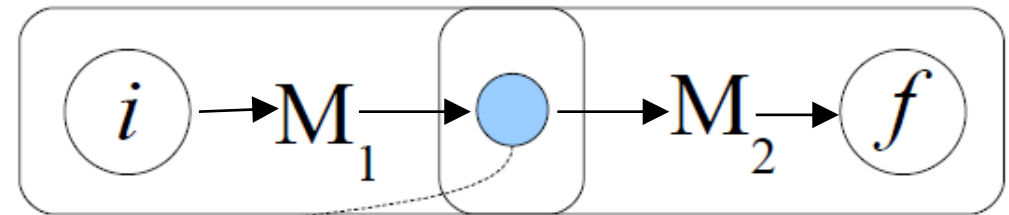
- **Algoritmo de McNaughton-Yamada-Thompson:**
 - Gera AFND para reconhecer cadeias definidas na expressão regular
 - Algoritmo dirigido por sintaxe (produz a derivação para as expressões)
 - Baseado no **tratamento recursivo das relações elementares**:
 - **Base:** símbolos do alfabeto
 - **Passo recursivo:** operações básicas (concatenação, união e fecho)
 - Constrói um AFND para cada relação com um **único estado de aceitação**
- **Etapas:**
 - Decompor a expressão regular em suas relações elementares (**do todo para a parte**)
 - Construir um autômato finito que reconheça cada uma das subexpressões obtidas (**da parte para o todo**)

Conversão Expressões Regulares

Autômato M

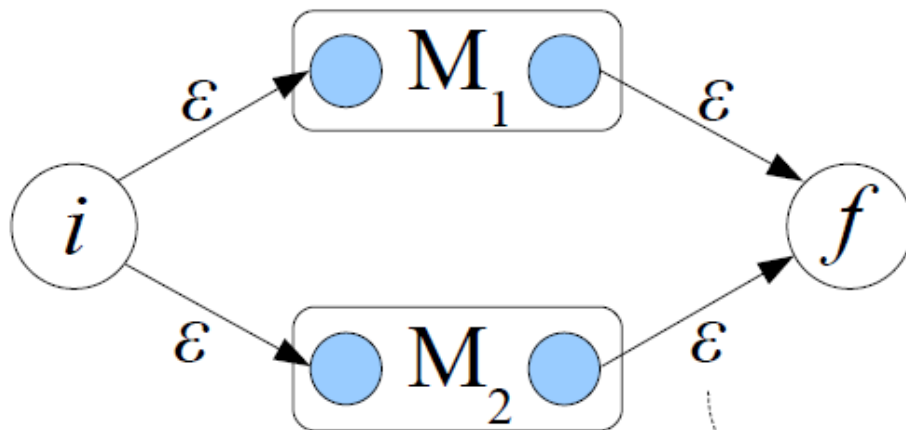


Reconhecimento do símbolo c



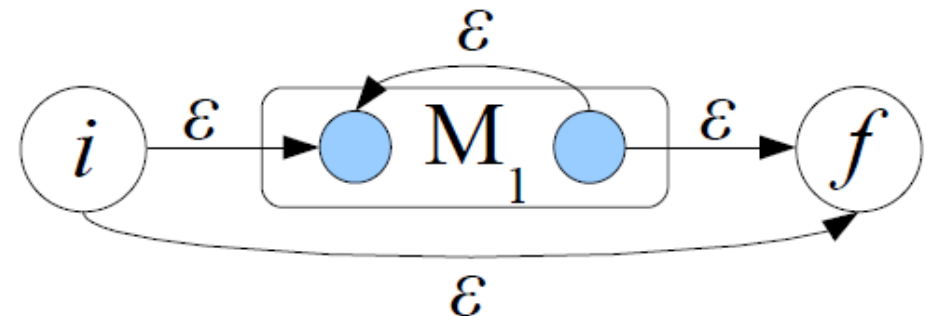
Concatenação

(estado final de M_1 e inicial de M_2)



União

**Movimento
vazio**



Fecho

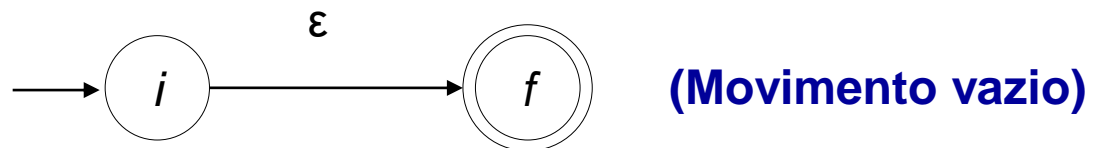
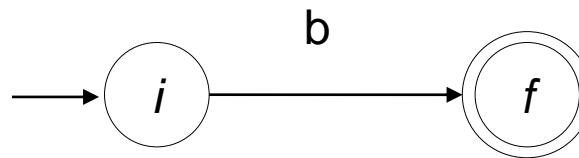
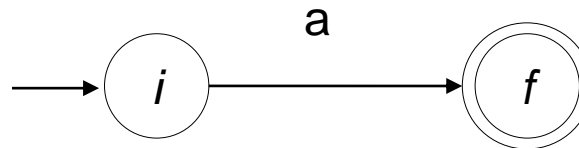
Conversão Expressões Regulares

- **Exemplo:** considere $R = (a/b)^*abb$

Como seria um AFND desta expressão regular?

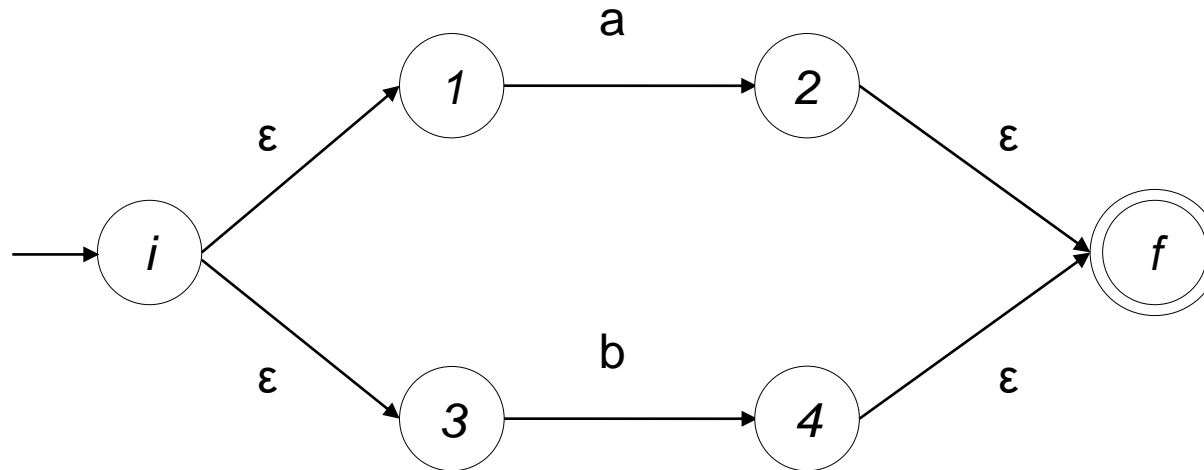
Conversão Expressões Regulares

- **Exemplo:** considere $R = (a/b)^*abb$
 - **1 Passo:** autômato para reconhecimento dos símbolos do alfabeto



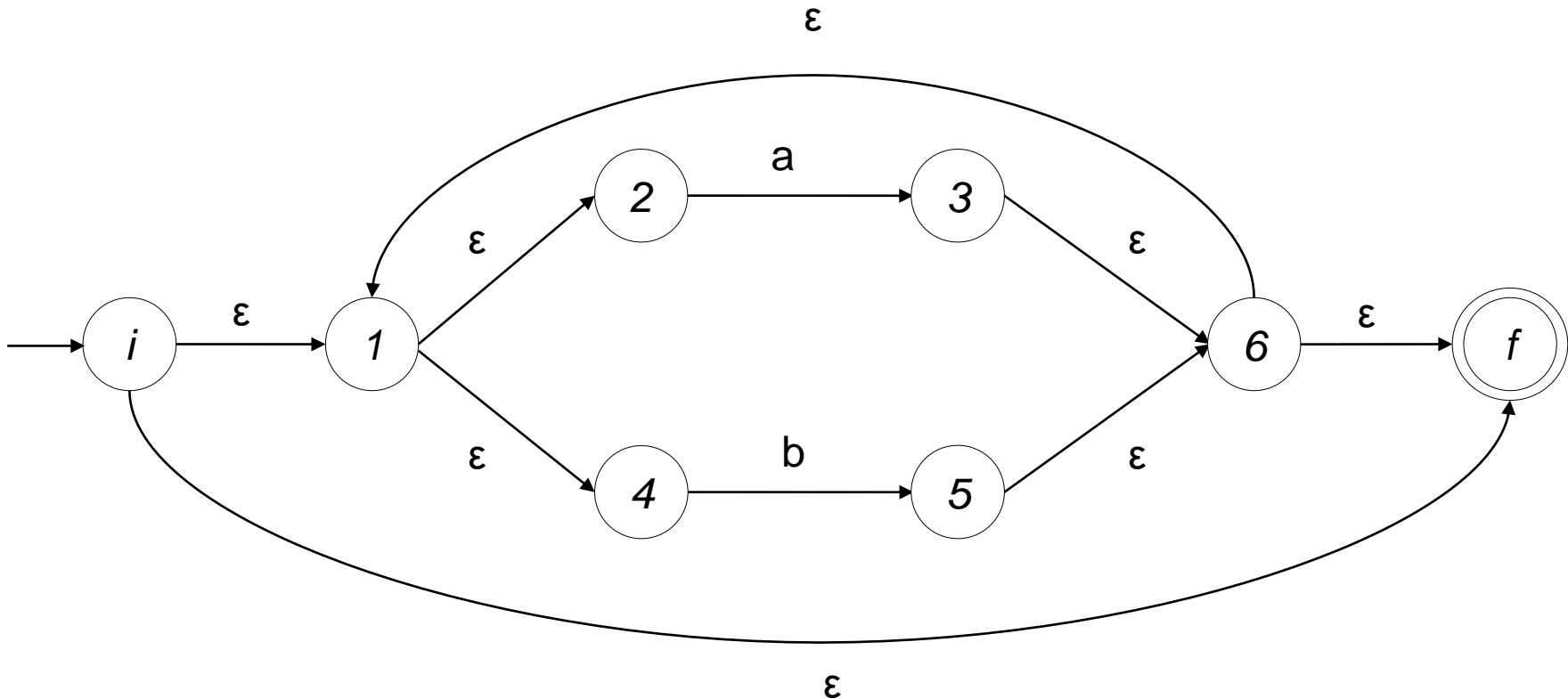
Conversão Expressões Regulares

- **Exemplo:** considere $R = (a/b)^*abb$
 - **2 Passo:** operação de união



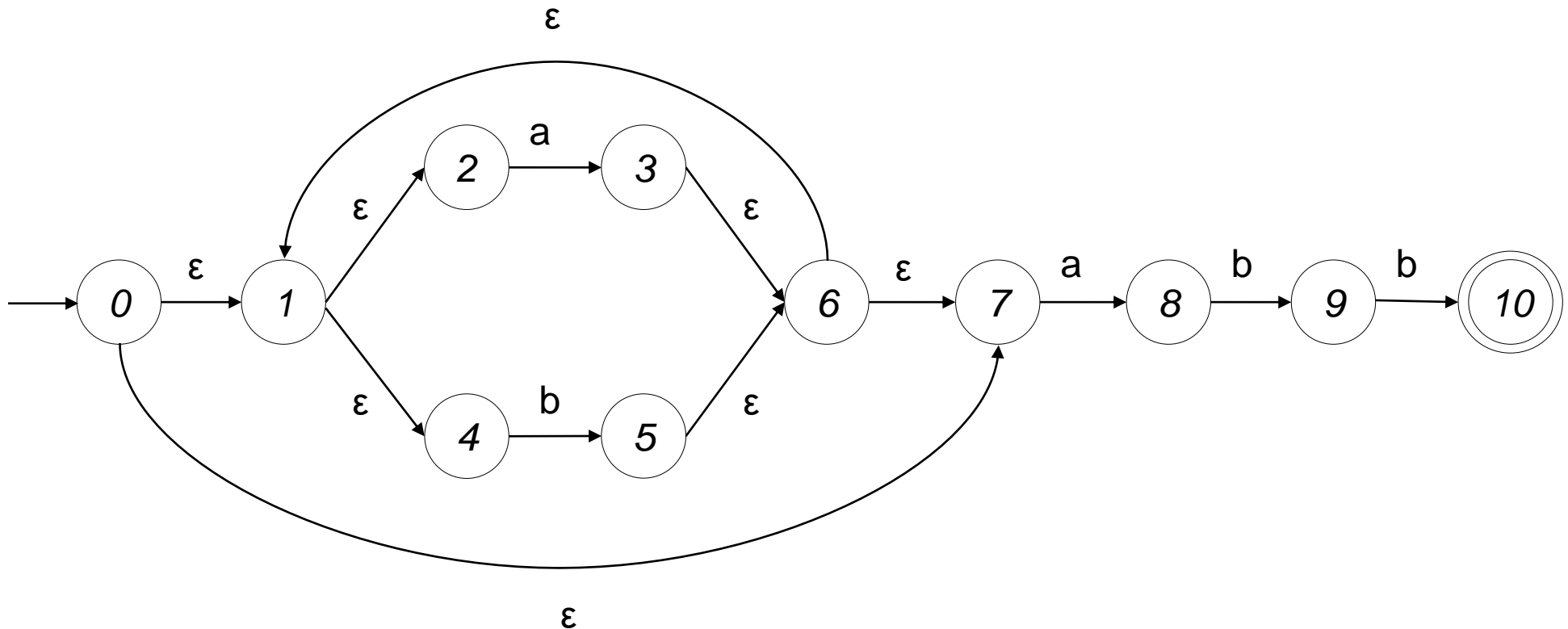
Conversão Expressões Regulares

- **Exemplo:** considere $R = (a/b)^*abb$
 - **3 Passo:** operação de fecho



Conversão Expressões Regulares

- **Exemplo:** considere $R = (a/b)^*abb$
 - **4 Passo:** operações de concatenação



Conversão Expressões Regulares

- **Algumas propriedades do algoritmo:**

- AFND gerado tem um estado inicial e um final
 - Estado inicial não possui transições de entrada
 - Estado final não possui transições de saída
- AFND gerado tem no máximo o dobro de estados em relação ao número de operadores e operandos da expressão regular
 - Cada passo do algoritmo cria no máximo 2 novos estados
- Cada estado do AFND (exceto o final) possui:
 - Uma transição de saída para um símbolo do alfabeto
 - Até duas transições de saída com movimento vazio (ϵ)

Conversão AFND para AFD

- Baseado na **construção de subconjuntos**:
 - Cada estado do AFD corresponde a um conjunto de estados do AFND
 - Tabela de transição do AFD simula todos os movimentos do AFND
- Possíveis dificuldades na conversão:
 - Crescimento exponencial do número de estados
 - Não costuma ocorrer na análise das linguagens reais
 - Tratar corretamente os movimentos vazios
- Operações básicas usadas no processo:
 - ϵ -closure(T): retorna o conjunto de estados do AFND que podem ser alcançados a partir de algum estado $t \in T$, usando movimentos vazios
 - ϵ -closure(s) retorna o fecho- ϵ de s
 - move(T, c): retorna o conjunto de estados do AFND que podem ser alcançados a partir de algum estado $t \in T$, ao ler o símbolo c

Cálculo do ε -closure(T)

Início

create(P) // cria uma pilha vazia P

Fecho $\leftarrow \{ \}$ // ε -closure

para (cada estado $t \in T$) **faça**

push(P,t) // empilha o estado t na pilha

Fecho \leftarrow Fecho \cup t

fim para

enquanto (Pilha não vazia) **faça**

pop(P,&t) // desempilha o topo

para (cada estado $s \in S$) **faça**

se (*move(t, ε) = s e $s \notin$ Fecho*) **então**

Fecho \leftarrow Fecho \cup s

push(P,s)

fim se

fim para

fim enquanto

retorna Fecho

Fim

Cálculo do ε -closure(T)

Início

create(P) // cria uma pilha vazia P

Fecho $\leftarrow \{ \}$ // ε -closure

para (cada estado $t \in T$) **faça**

push(P,t) // empilha o estado t na pilha

Fecho \leftarrow Fecho \cup t

fim para

enquanto (Pilha não vazia) **faça**

pop(P,&t) // desempilha o topo

para (cada estado $s \in S$) **faça**

se (*move(t, ε) = s e $s \notin$ Fecho*) **então**

Fecho \leftarrow Fecho \cup s

push(P,s)

fim se

fim para

fim enquanto

retorna Fecho

Fim

Implementa uma **busca direta em um grafo** a partir de um **conjunto de estados T** e considerando apenas os **movimentos vazios** (arestas com ε)

Conversão AFND para AFD

- **Entrada:** um **AFND** N com estado inicial s_0 e estados finais F
- **Saída:** um **AFD** D equivalente a N com estado inicial S' e estados finais F'

Início

```
 $S' \leftarrow \varepsilon\text{-closure}(s_0)$  // estado inicial de  $D$   
 $S \leftarrow S'$  // conj. estados de  $D$   
 $M \leftarrow \{\}$  // conj. de estados marcados  
enquanto ( $\exists T \in S$  não marcado) faça  
   $M \leftarrow M \cup T$  // marca  $T$   
  para (cada símbolo  $c \in \Sigma$ ) faça  
     $T' \leftarrow \varepsilon\text{-closure}(\text{move}(T, c))$   
    Se ( $T' \notin S$ ) então  $S \leftarrow S \cup T'$   
     $\text{tabTrans}[T, c] \leftarrow T'$  // cria transição em  $D$   
  fim para  
fim enquanto
```

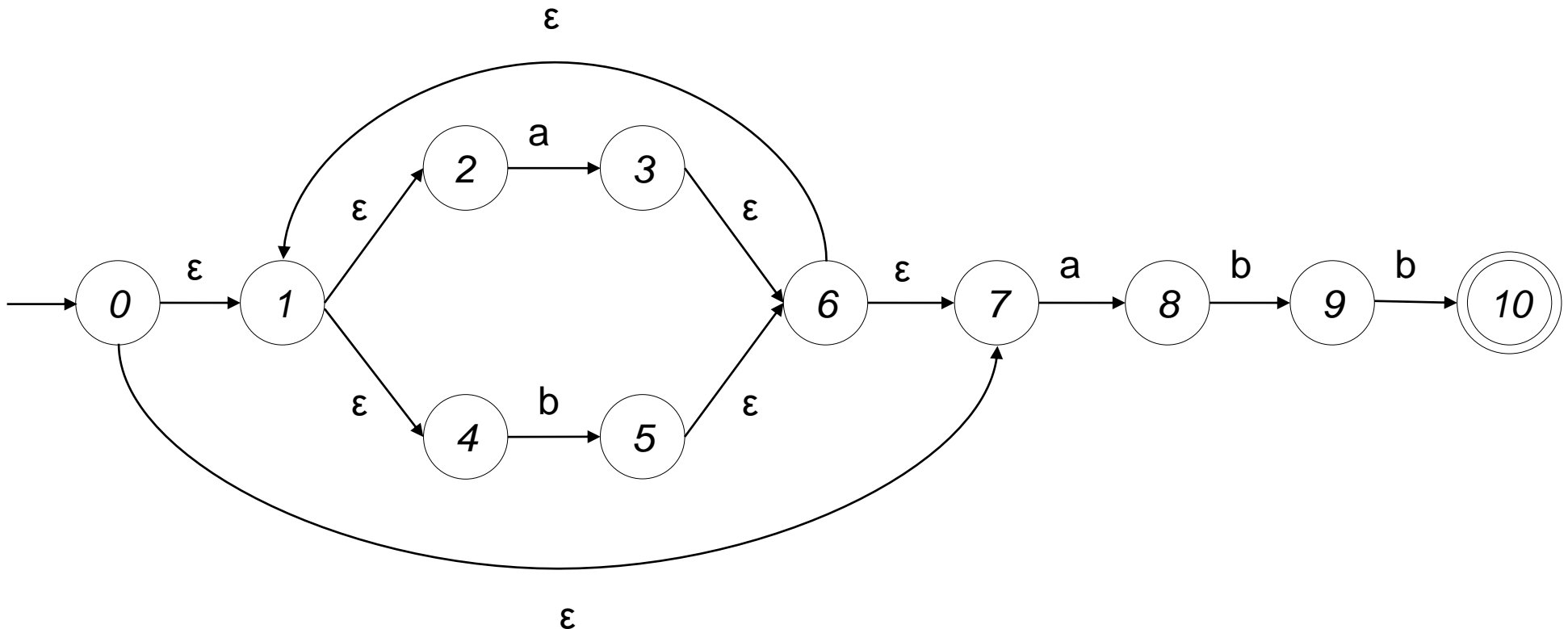
...

...

```
para (cada conj. estado  $T \in S$ ) faça  
  para (cada estado  $s \in T$ ) faça  
    se ( $s \in F$ ) então  
       $F' \leftarrow T$  // estado final de  $D$   
    break  
  fim se  
fim para  
fim para  
Fim
```

Conversão AFND para AFD

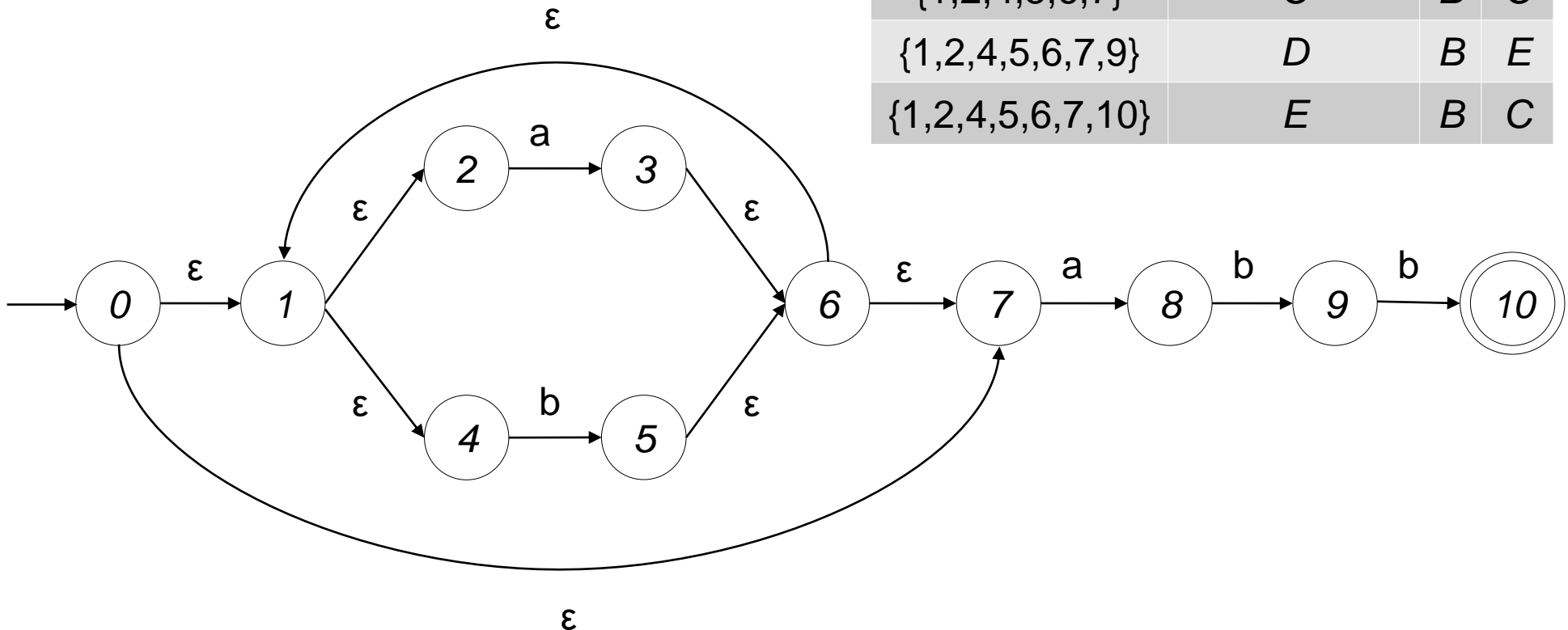
- **Exemplo:** considere o AFND de $R = (a/b)^*abb$



Conversão AFND para AFD

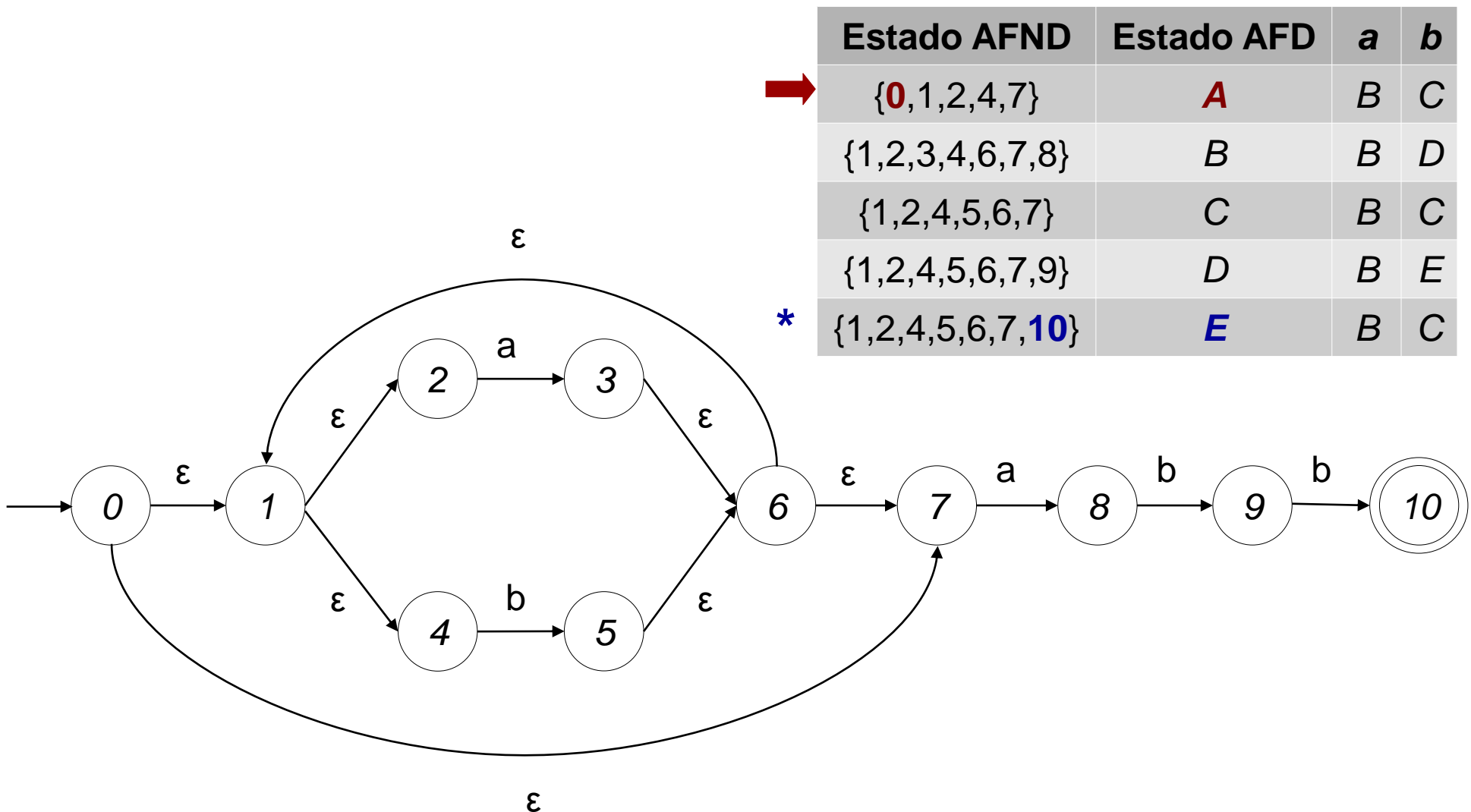
- Exemplo: considere o AFND de $R = (a/b)^*abb$

Estado AFND	Estado AFD	<i>a</i>	<i>b</i>
{0,1,2,4,7}	<i>A</i>	<i>B</i>	<i>C</i>
{1,2,3,4,6,7,8}	<i>B</i>	<i>B</i>	<i>D</i>
{1,2,4,5,6,7}	<i>C</i>	<i>B</i>	<i>C</i>
{1,2,4,5,6,7,9}	<i>D</i>	<i>B</i>	<i>E</i>
{1,2,4,5,6,7,10}	<i>E</i>	<i>B</i>	<i>C</i>



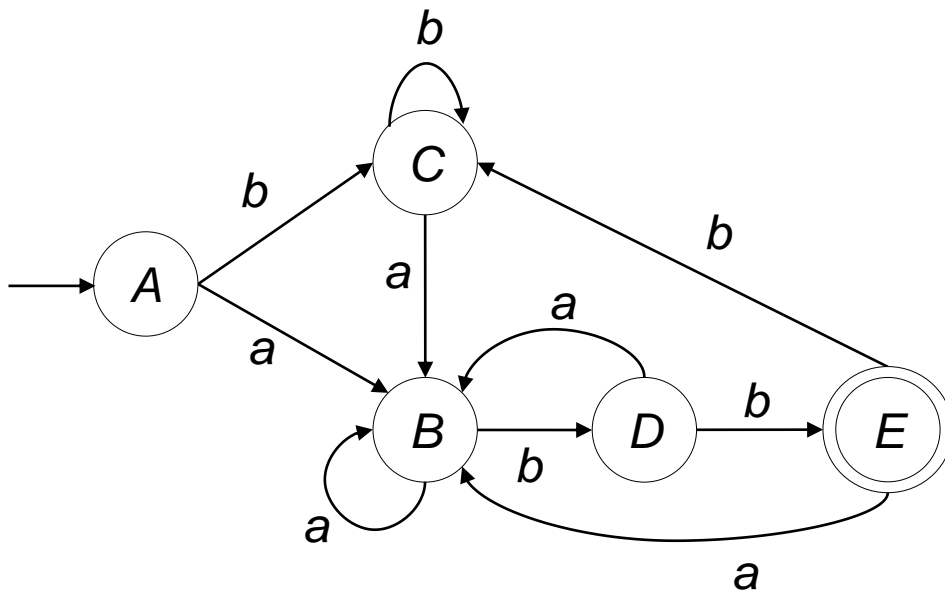
Conversão AFND para AFD

- Exemplo: considere o AFND de $R = (a/b)^*abb$



Conversão AFND para AFD

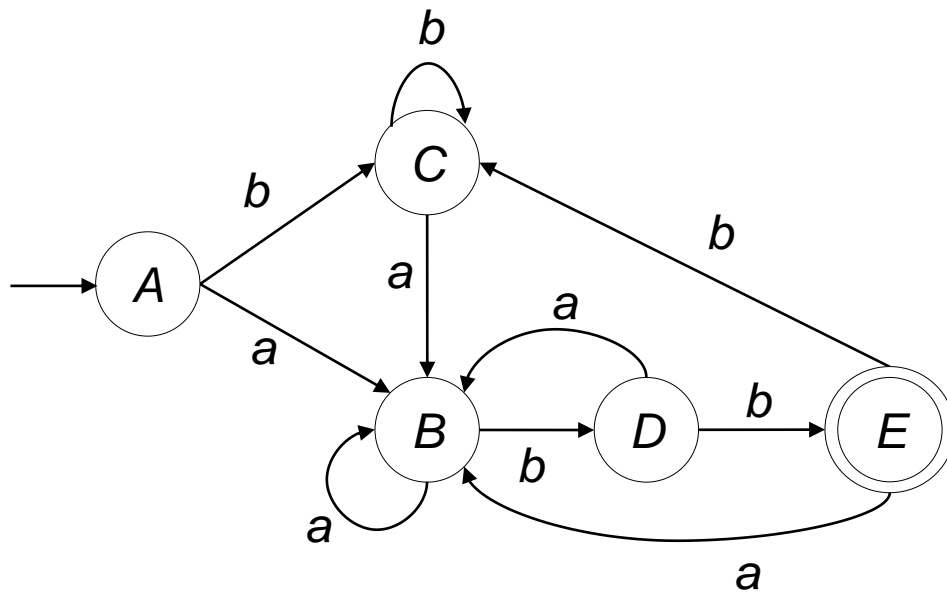
- Exemplo: considere o AFND de $R = (a/b)^*abb$



Estado AFND	Estado AFD	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,4,5,6,7,10\}$	E	B	C

Conversão AFND para AFD

- Exemplo:** considere o AFND de $R = (a/b)^*abb$

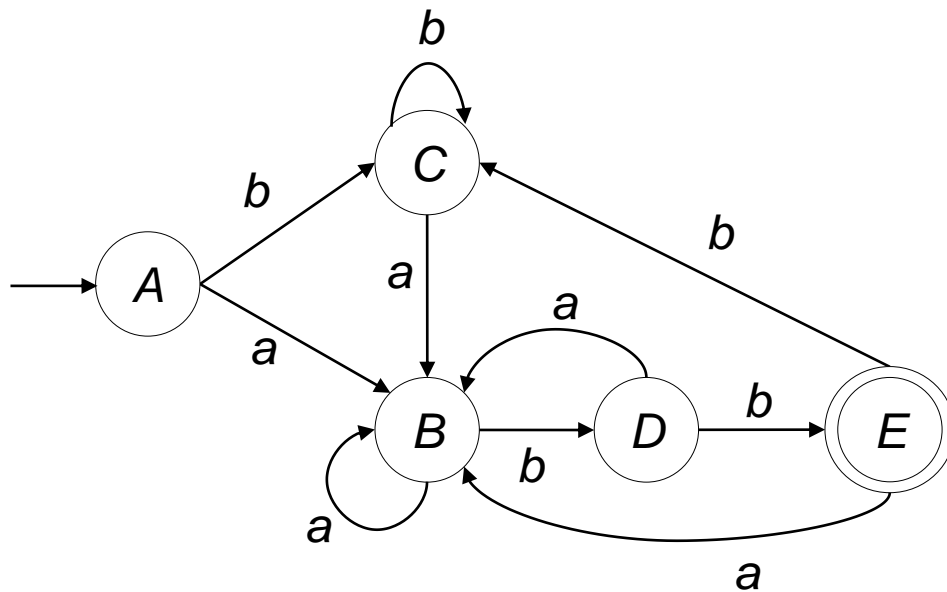


Estado AFND	Estado AFD	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
* $\{1, 2, 3, 5, 6, 7, 10\}$	E	B	C

É o AFD ótimo (menor possível)?

Conversão AFND para AFD

- Exemplo:** considere o AFND de $R = (a/b)^*abb$



Estado AFND	Estado AFD	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,3,5,6,7,10\}$	E	B	C

É o AFD ótimo (menor possível)?

R: Não, pois A e C possuem as mesmas transições e, portanto, podem ser unificados

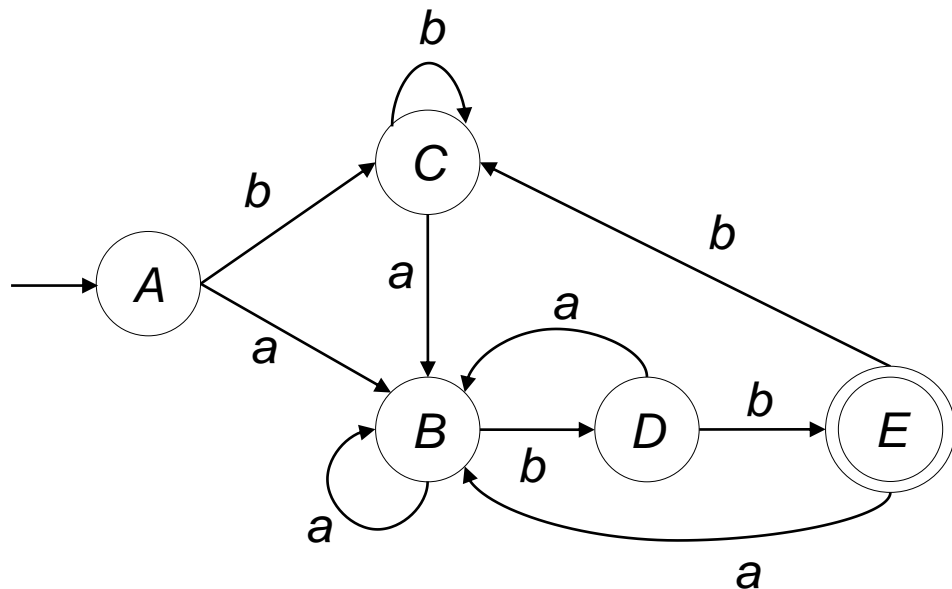
Minimização de Estados do AFD

- **Elimina estados** do AFD que possuem **as mesmas transições**
- **Algoritmo:**
 - **Separe o conjunto de estados** do AFD (S) em:
 - S_1 : subconjunto com todos os **estados finais** (F)
 - S_2 : subgrupo contendo os **estados não-finais** ($S - F$)
 - **Particione cada subconjunto S_N** de modo que 2 estados de S permanecerão juntos se **eles possuírem as mesmas transições**
 - **Repita o passo anterior** para os novos subgrupos até que **não seja possível novas partições**

Minimização de Estados do AFD

- Exemplo:** considere o AFND de $R = (a/b)^*abb$

Estado AFD	a	b
$\rightarrow A$	B	C
B	B	D
C	B	C
D	B	E
$*$ E	B	C

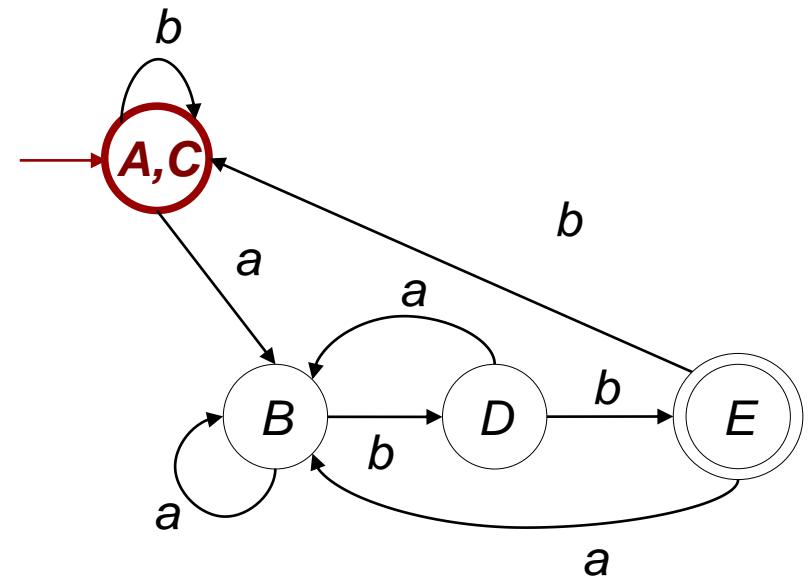


Estados de não aceitação:

$\{A,B,C,D\} \rightarrow \{A,C\}, \{B\}, \{D\}$

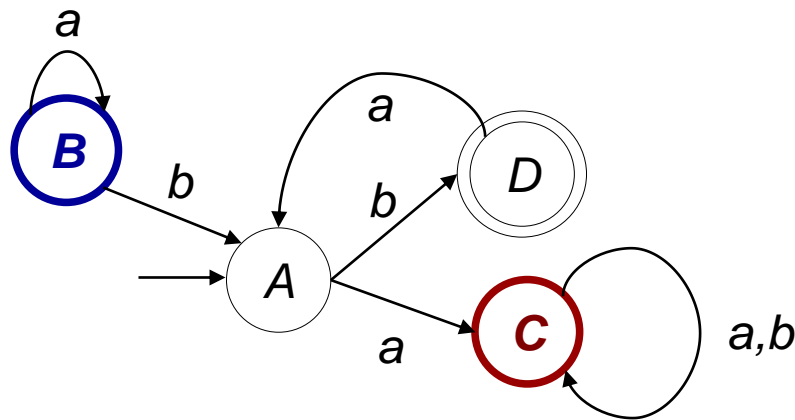
Estados de aceitação:

$\{E\}$



Estados Desnecessários

- AFD minimizado ainda pode conter estados cujo a remoção não altera a linguagem reconhecida
 - **Inalcançáveis** a partir do estado inicial
 - Estados “**mortos**”
 - **Ex:**



- Remoção desses estados pode melhorar a eficiência do autômato
 - **Ex:** antecipa a identificação de erros (rejeição)

Simulação de um AFD

- **Entrada:** Um **AFD** D e uma **cadeia de entrada** x
- **Saída:** **1** – se D aceitar x **ou 0** – caso contrário

- **Algoritmo:** *Início*

$s \leftarrow s_0$ // estado inicial

$c \leftarrow \text{prox_char}()$

enquanto ($c \neq \text{EOF}$) **faça**

$s \leftarrow \text{move}(s, c)$

$c \leftarrow \text{prox_char}()$

fim enquanto

se ($\text{final}(s)$) **então** retorna 1

senão retorna 0

Fim

**Código baseado em
tabela de transição**

Simulação de um AFND

- Simulação baseada na **construção de subconjuntos** durante a execução
 - Ao invés de um estado guarda um **conjunto de possíveis estados**
 - Deve simular o **paralelismo** da busca
 - Busca é encerrada quando **não existe transição** para a entrada
 - Conjunto de estados é vazio
 - Garante a obtenção do maior prefixo de entrada
- Passos do algoritmo:
 - Guardar o **conjunto de estados correntes** S_c
 - Obter os **próximos estados** S_p a partir do caractere de entrada c
 - **Realizar a transição** sobre o conjunto S_c (***move***(S_c , c))
 - **Aplicar o fecho- ϵ** sobre o resultado da transição

Simulação de um AFND

- **Entrada:** Um AFND N e uma cadeia de entrada x
- **Saída:** 1 – se N aceitar x ou 0 – caso contrário

- **Algoritmo:**
 - Início*
 - $S_c \leftarrow \varepsilon\text{-closure}(s_0)$
 - $c \leftarrow \text{prox_char}()$
 - enquanto** ($c \neq \text{EOF}$) **faça**
 - $S_p \leftarrow \varepsilon\text{-closure}(\text{move}(S_c, c))$
 - $c \leftarrow \text{prox_char}()$
 - $S_c \leftarrow S_p$
 - fim enquanto**
 - se** ($\text{final}(S_c)$) **então** retorna 1
 - senão** retorna 0
 - Fim*

Simulação de um AFND

- Forma de implementação pode melhorar eficiência
 - **Ex:** intercalar conjuntos é mais eficiente que copiá-los em cada iteração
- **Estruturas de dados:**
 - **2 pilhas:** armazena um conjunto de estados
 - Conjunto de estados **atual** e o **próximo**
 - **Vetor de visitados:** indica os estados que já estão como próximo
 - Vetor pode ser binário ou booliano
 - Replica dados, mas **facilita a consulta**
 - **Matriz de listas:** contém a tabela de transição do AFND
 - Cada elemento (**lista encadeada**) representa o conjunto de estados resultante da transição
- **Operações auxiliares:**
 - Inicializar o vetor binário (**todos elementos igual a ZERO**)
 - Computar o **fecho- ϵ** sobre um estado (**ϵ -closure(T,s)**)
 - Computar a **transição para um conjunto** de estados

Simulação de um AFND

- Intercalação dos subconjuntos:**

Início

$S_1 \leftarrow \varepsilon\text{-closure}(s_0)$

$Flag \leftarrow 0$

$c \leftarrow \text{prox_char}()$

enquanto ($c \neq EOF$) **faça**

fim enquanto

se (($Flag = 0$ e $\text{final}(S_1)$)) **ou**
 ($Flag = 1$ e $\text{final}(S_2)$)) **então**

 retorna 1

senão retorna 0

Fim

se ($Flag = 0$) **então**

$S_2 \leftarrow \varepsilon\text{-closure}(\text{move}(S_1, c))$

$Flag \leftarrow 1$

senão

$S_1 \leftarrow \varepsilon\text{-closure}(\text{move}(S_2, c))$

$Flag \leftarrow 0$

fim se

$c \leftarrow \text{prox_char}()$

Simulação de um AFND

Cálculo do Fecho- ϵ

Função ϵ -closure (*End. Pilha P , estado s*)

push(P , s)

visitado[s] $\leftarrow 1$

para (t em *move*(s , ϵ)) **faça**

se (*visitado*[t] = 0) **então**

ϵ -closure (P , t)

fim se

fim para

Fim função

Cálculo da Transição (ϵ -closure(S_p , *move*(S_c , c)))

enquanto (S_c não vazia) **faça**

pop(S_c , & s) // *Desempilha atual*

para (t em *move*(s , c)) **faça**

se (*visitado*[t] = 0) **então**

ϵ -closure (S_p , t)

fim se

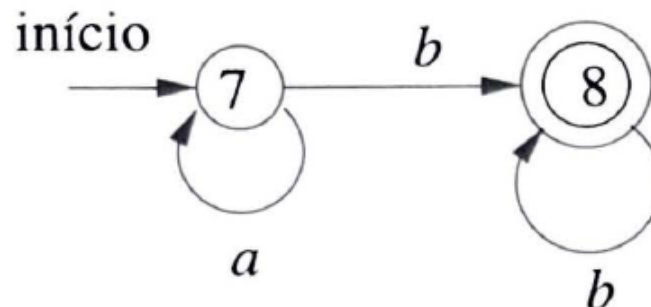
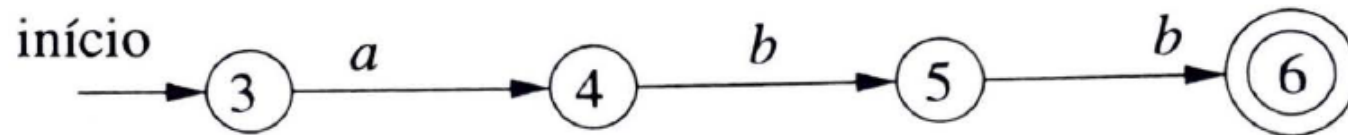
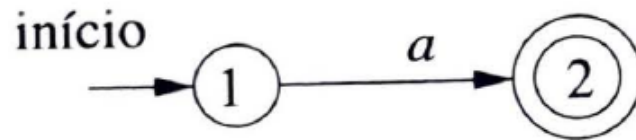
fim para

fim enquanto

Função ϵ -closure modificada para passar **endereço da pilha como parâmetro**

Simulação de um AFND

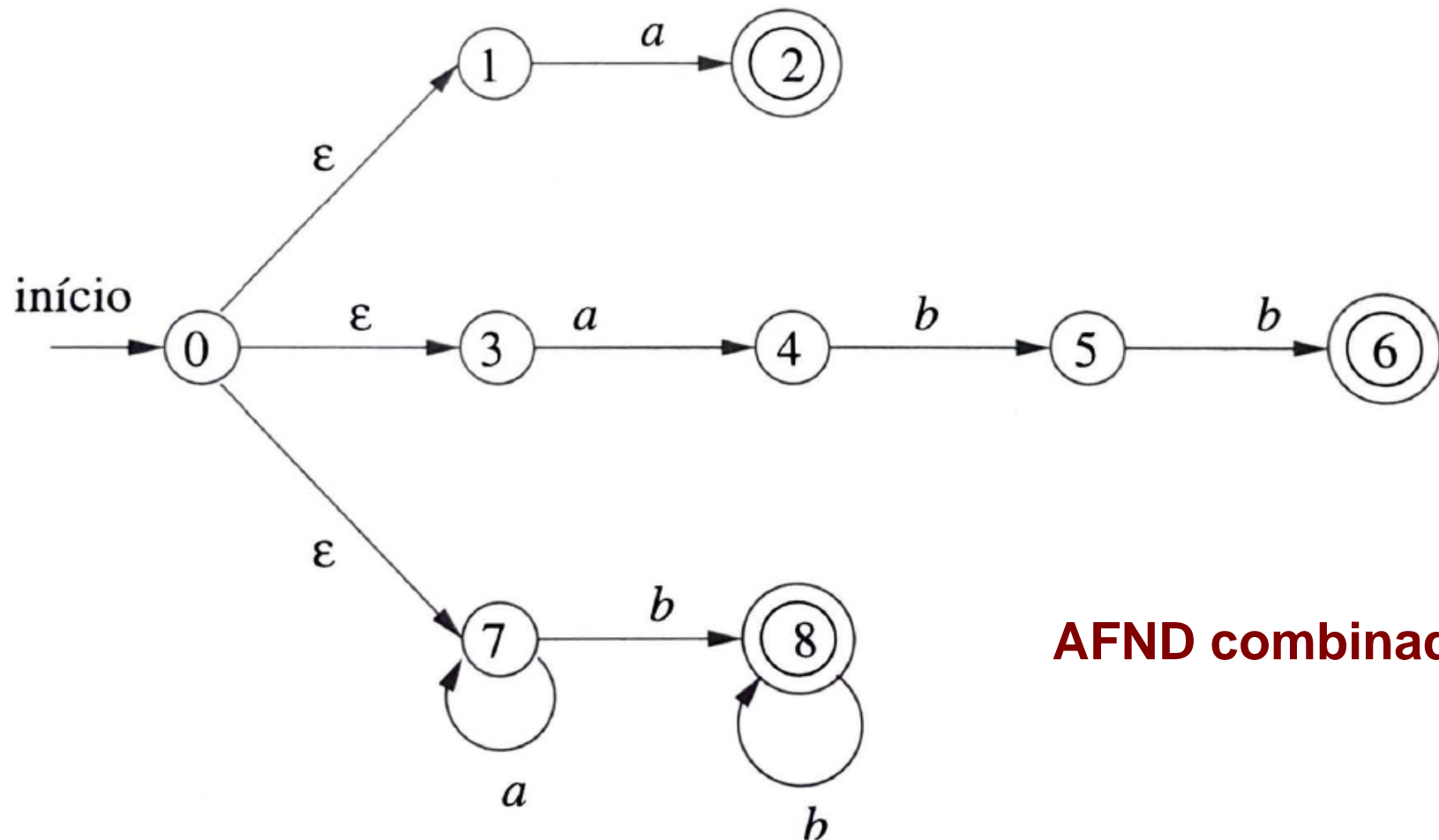
- **Exemplo:** padrões a , abb , a^+b^+



**Autômatos dos
padrões**

Simulação de um AFND

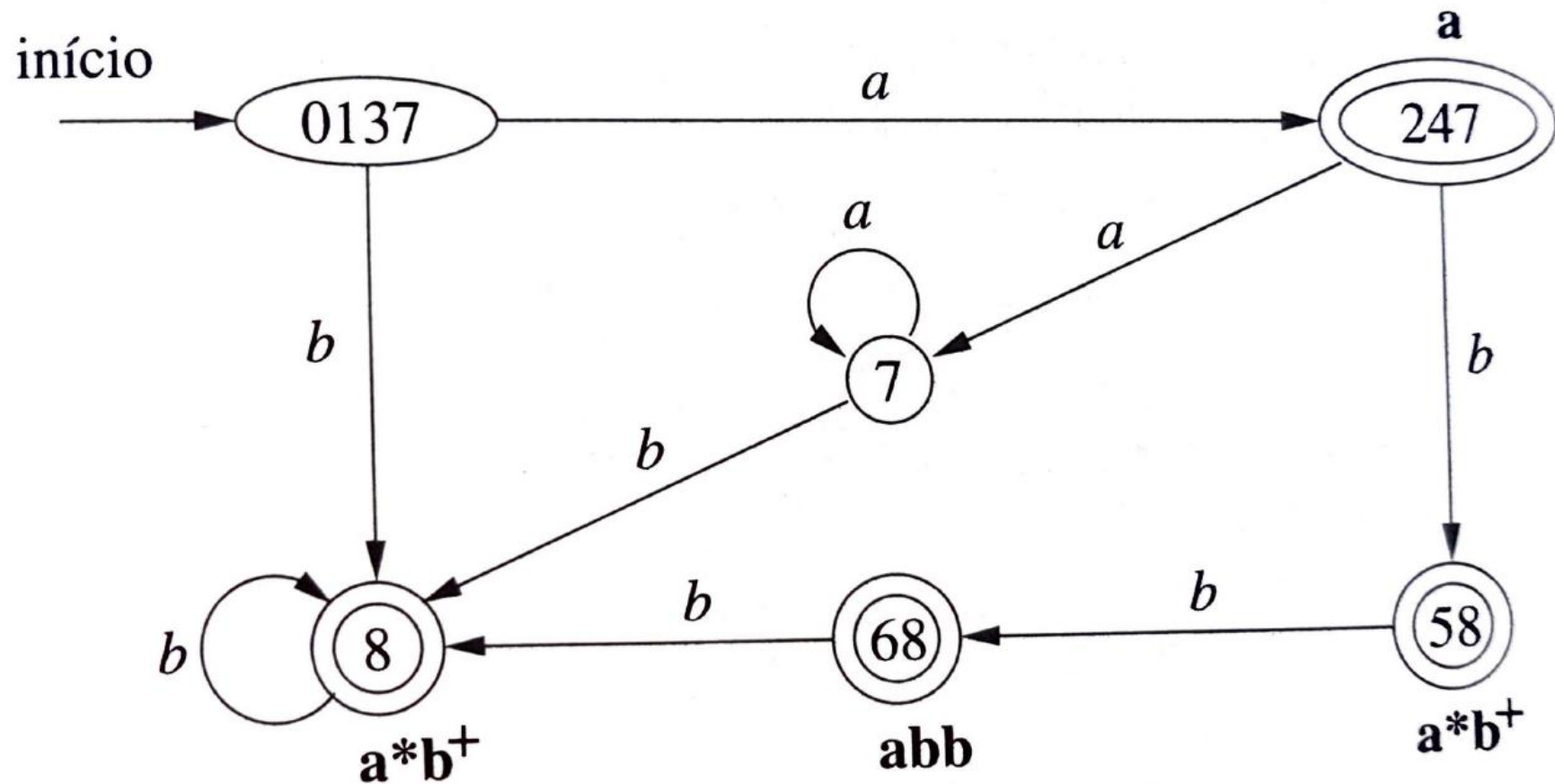
- **Exemplo:** padrões a , abb , a^+b^+



AFND combinado

Simulação de um AFND

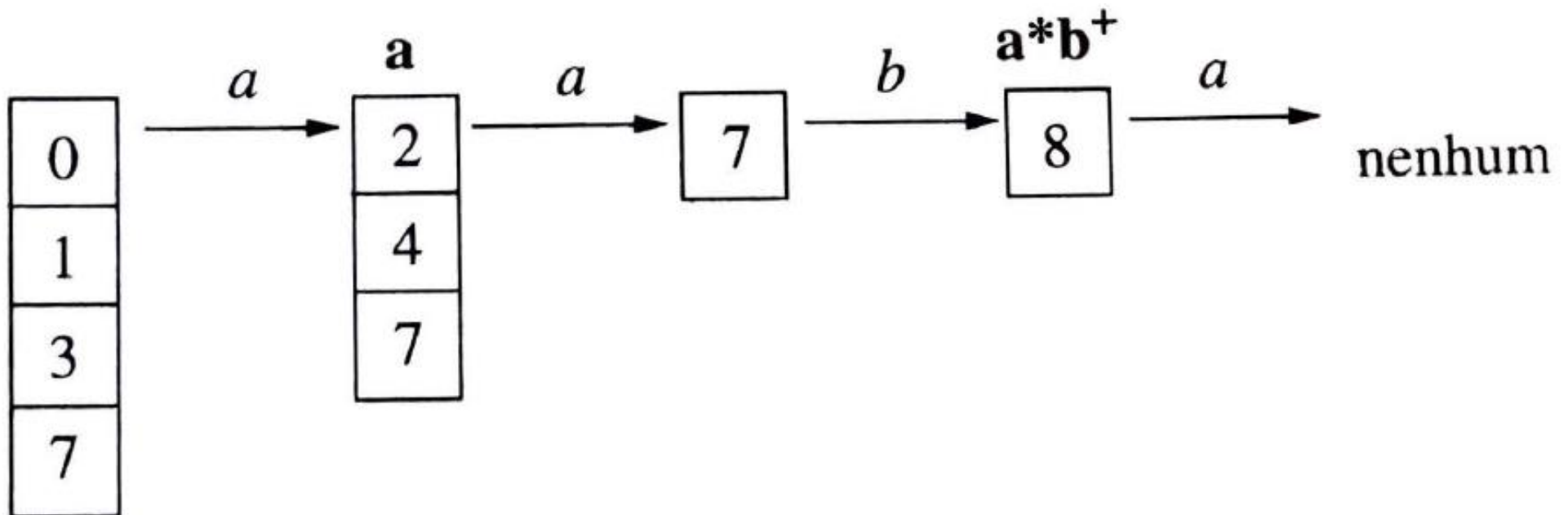
- Exemplo:** padrões a , abb , a^*b^+



AFD resultante

Simulação de um AFND

- **Exemplo:** padrões a , abb , a^*b^+



Processamento da entrada ***aaba***

Referências Bibliográficas

- Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. Compiladores: Princípios, técnicas e ferramentas, 2 edição, Pearson, 2008
- Alexandre, E.S.M. Livro de Introdução a Compiladores, UFPB, 2014
- Aluisio, S. slides da disciplina “Teoria da Computação e Compiladores”, ICMC/USP, 2011
- Dubach, C. slides da disciplina “*Compiling Techniques*”, University of Edinburgh, 2018
- Freitas, R. L. notas de aula - Compiladores, PUC Campinas, 2000
- GeeksforGeeks, Flex (*Fast Lexical Analyser Generator*),
<https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>