O Processo de Varredura e Expressões Regulares

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

Introdução

Marcas ou *tokens*

Sequência de caracteres que representa uma unidade de informação do programa fonte.

- Palavras-chave;
- identificadores;
- símbolos especiais.

Reconhecimento de Padrões

Expressões Regulares e Autômatos.

O Processo de Varredura

Processo de Varredura

Ler caracteres do código-fonte e organizá-los em unidades lógicas para outras fases do compilador.

Marcas

```
typedef enum
   {IF, THEN, ELSE, PLUS, MINUS, NUM, ID, ...}
   TokenType;
```

- A marca IF tem como **lexema** ou valor, na maioria das linguagens, a cadeia *if* ;
- palavras reservadas tem um único lexema;
- identificadores podem ter diversos lexemas;
- números tem como lexemas sua representação em cadeias, porém outros **atributos**, como o próprio valor númerico, podem ser armazenados no *token*;
- o *token* ou marca final é a coleção de todos seus atributos.

Registro de Marcas

```
typedef struct {
   TokenType tokenval;
   char *stringval;
   int numval;
 }
```

```
typedef struct {
   TokenType tokenval;
   union {
      char *stringval;
      int numval;
   } attribute;
} TokenRecord;
```

Metodologia

Uma solução comum é o sistema de varredura retornar apenas o valor da marca e colocar os outros atributos em variáveis que possam ser acessadas por outras partes do compilador.

Interface com Analisador Sintático

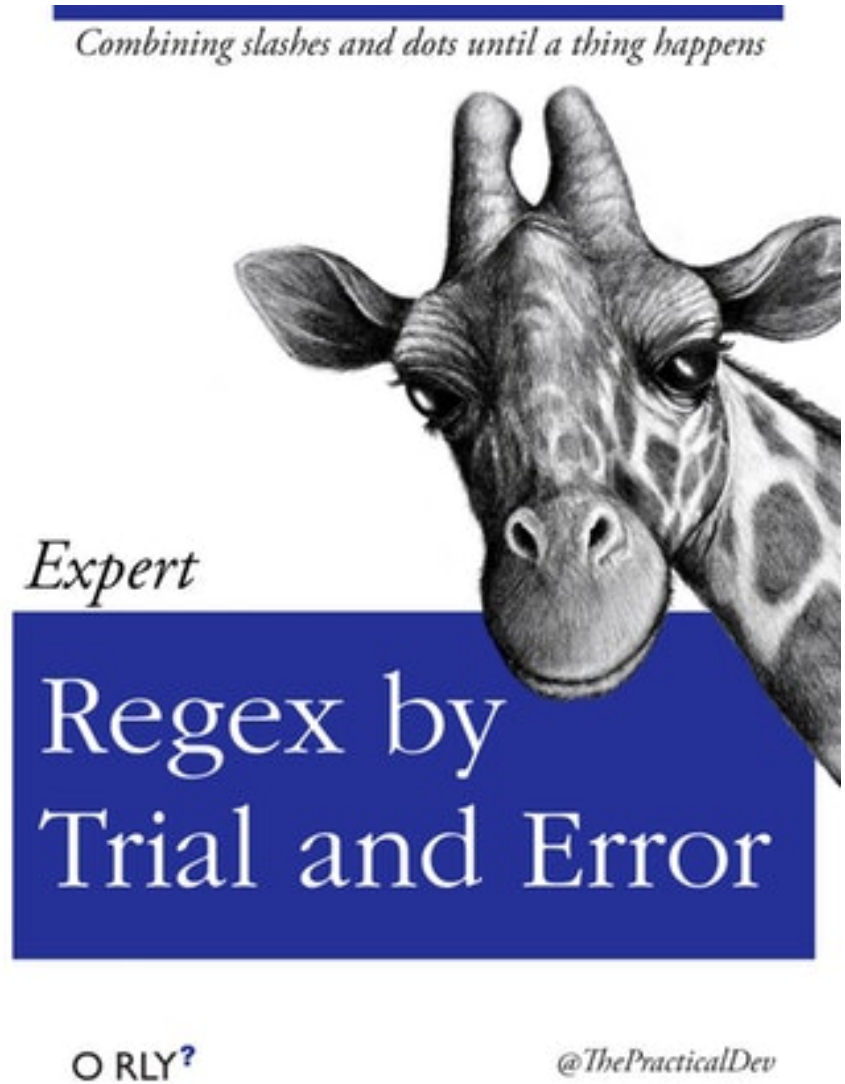
```
TokenType getToken(void);
```

- A varredura está sob controle do analisador sintático;
- em outras palavras, a varredura é invocada, não tem o controle do processo.

Expressões Regulares

Padrões

- Uma expressão regular r é completamente definida pelo conjunto de cadeiras de caracteres com as quais ela casa;
- linguagem gerada;
- alfabeto Σ;
- metacaracateres ou metasímbolos.



Definição de Expressões Regulares

Expressões Regulares Básicas

- Dado um caractere a de Σ, $L(a) = a$;
- cadeia vazia: ε, $L(\varepsilon) = \varepsilon$;
- conjunto vazio: Φ, $L(\Phi) = \{\}$.

Operações de Expressões Regulares

- Escolha entre alternativas: $|$;
- concatenação;
- repetição ou *fecho*: *.

Definição de Expressões Regulares

Escolha Entre Alternativas

- Se r e s são ERs, então $r|s$ é uma ER;
- casa com qualquer cadeia que case com r **ou** s;
- $L(r|s) = L(r) \cup L(s)$.

Concatenação

- Se r e s são ERs, então rs é uma ER;
- casa cadeias que sejam concatenação de duas cadeias, desde que a primeira case com r e a segunda s;
- $L(rs) = L(r)L(s)$.

Definição de Expressões Regulares

Repetição

- Se r é uma ER, então r^* é uma ER;
- r^* casa com qualquer concatenação finita de cadeias de caracteres, desde que cada cadeia case com r;
- $L(r^*) = L(r)^*$.

Precedência de Operações e Parênteses

- $a|b^*$ significa $(a|b)^*$ ou $a|(b^*)$?
- $*$ tem maior precedência, seguido da concatenação e por último $|$;
- os parênteses servem para alterar a ordem da precedência.

Nomes ou Abreviaturas para ERs

Representar sequências de um ou mais dígitos:

`(0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9)*`

ERs que aparecem com frequência podem ser abreviadas:

`digito digito*, onde:`
`digito = 0|1|2|3|4|5|6|7|8|9`

Definição de Expressões Regulares

Uma **expressão regular** é uma das seguintes:

1. Uma expressão regular **básica**, composta por um único caractere **a**, onde a pertence a um alfabeto Σ de caracteres legais; o metacaractere ε; ou o metacaractere Φ.
2. Uma expressão da forma $r|s$, onde r e s são expressões regulares. Nesse caso, $L(r|s) = L(r) \cup L(s)$.
3. Uma expressão da forma rs, onde r e s são expressões regulares. Nesse caso, $L(rs) = L(r)L(s)$.
4. Uma expressão da forma r^*, onde r é uma expressão regular. Nesse caso, $L(r^*) = L(r)^*$.
5. Uma expressão da forma (r), onde r é uma expressão regular. Nesse caso, $L((r)) = L(r)$. Assim parênteses não modificam a linguagem. Eles são utilizados apenas para ajudar a precedência de operadores.

Exemplos de Expressões Regulares

- $(a|c)^*b(a|c)^*$
- $(a|c)^*|(a|c)^*b(a|c)^*$
- Cadeias em $\Sigma = \{a, b\}$ compostas por um único b rodeadas pelo mesmo número de a's
- Cadeias em $\Sigma = \{a, b, c\}$ sem b's consecutivos
- $((b|c)^*a(b|c)^*a)^*(b|c)^*$

Extensões de Expressões Regulares

Uma ou Mais Repetições

- Semelhante ao $*$;
- *uma* ou mais repetições em vez de zero ou mais repetições;
- símbolo $+$.

Qualquer Caractere

- Representar qualquer caractere do alfabeto;
- o símbolo . (ponto) é uma abreviatura para esse caso.

Intervalo de Caracteres

- $a|b|...|z$ para letras e $0|1|...|9$ para números;
- $[a-z]$ e $[0-9]$ desempenham o mesmo papel.

Extensões de Expressões Regulares

Qualquer Caractere Fora de Um Dado Conjunto

- O uso do circunflexo elimina qualquer caractere de um conjunto;
- $[^{\wedge}abc]$.

Subexpressões Opcionais

- Expressões que ocorrem uma vez ou nenhuma;
- basta seguir a expressão com ?.

Expressões Regulares para Marcas

Números

```
nat = [0-9]+
signedNat = (+|-)? nat
number = signedNat("." nat)?(E signedNat)?
```

Palavras Reservadas e Identificadores

```
reservadas = if | while | do | ...

letra = [a-zA-Z]
digito = [0-9]
identificador = letra(letra | digito)*
```

Expressões Regulares para Marcas

Comentários

- Comentários de uma linha são simples;
- comentários de várias linhas, estilo C, resultam em ERs complexas;
- o desenvolvedor de compiladores acaba tratando de maneira *ad hoc*.

Expressões Regulares para Marcas

Ambigüidade

- Quando uma cadeia pode ser um identificador ou palavra-chave, a última prevalece;
- **princípio da subcadeia mais longa**: a cadeia mais longa de caracteres que poderia constituir uma única marca deve representar a próxima marca;
- *whileYouWereSleeping* é um identificador, não o começo de um bloco *while*;
- **delimitadores**: caracteres que fixam um fim para a subcadeia mais longa:
  - operadores;
  - espaços em branco;
  - ponto-e-vírgula.

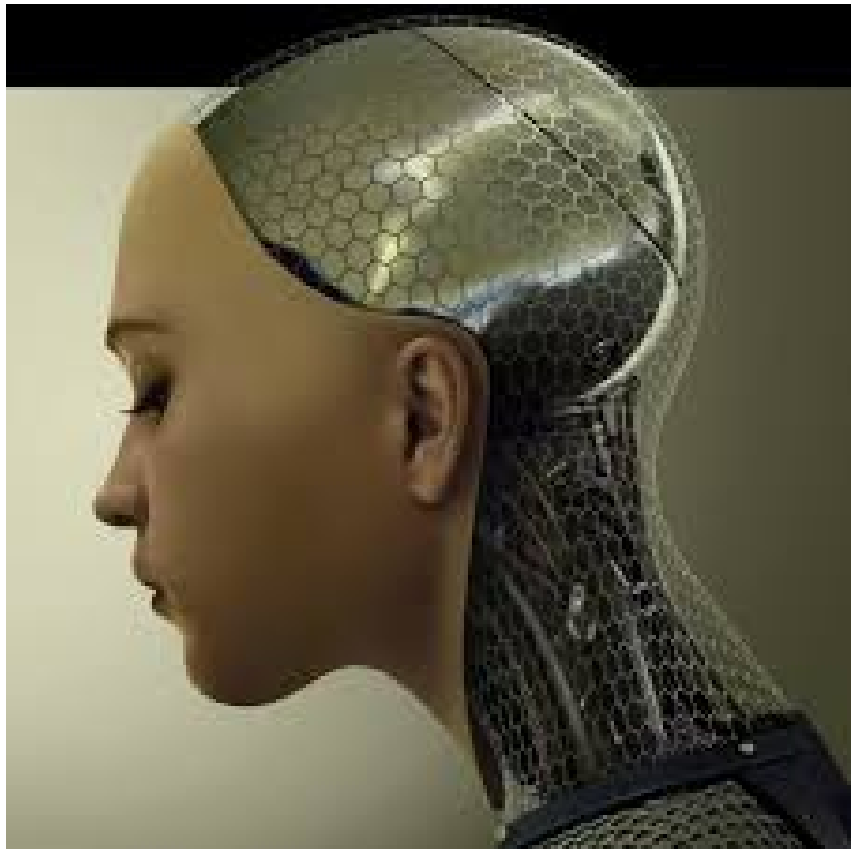
OFF-TOPIC: grep e awk

- **grep** (exercício 2.3): ferramenta que analisa um fluxo de linhas (arquivos) e retorna quais elementos do fluxo casam com determinada ER;
- **awk**: linguagem de programação completa com foco na manipulação de cadeia de caracteres.

Autômatos Finitos

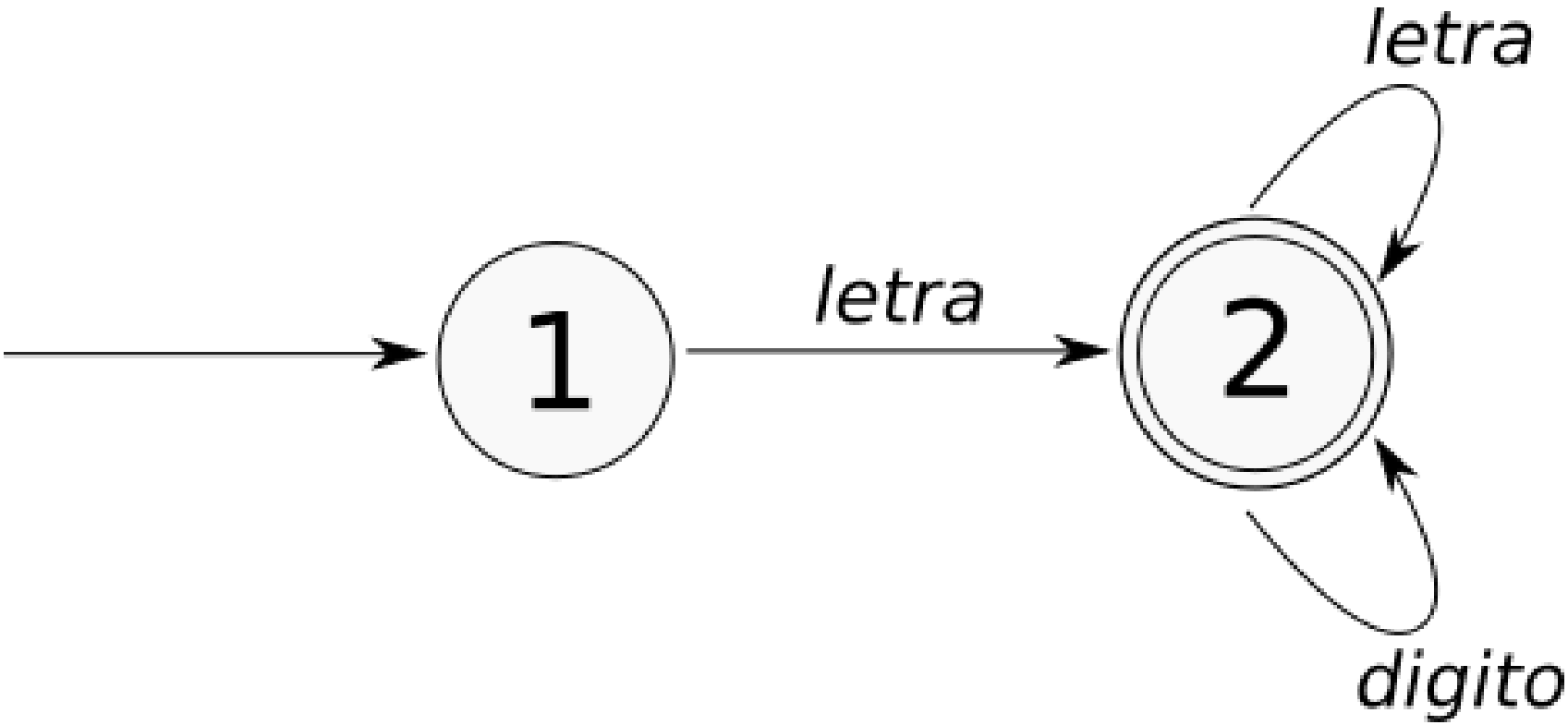
Máquina de Reconhecimento

Autômatos finitos podem ser utilizados para descrever o processo de reconhecimento de padrões em cadeias de entrada.



Autômatos Finitos

```
identificador = letra(letra|digito)*
```



Autômatos Finitos

Definição

Um **DFA** (autômato finito determinístico) M é composto por:

- Um alfabeto Σ;
- um conjunto de estados S;
- uma função de transição $T : SX\Sigma \rightarrow S$;
- um conjunto de estados de aceitação $A \subset S$.

A linguagem aceita por M, denotada como $L(M)$, é definida como o conjunto de cadeias de caracteres $c_1c_2...c_n$ no qual cada $c_i \in \Sigma$ é tal que existem estados $s_1 = T(s_0, c_1), s_2 = T(s_1, c_2), ..., s_n = T(s_{n-1}, c_n)$ em que cada s_n é um elemento de A (ou seja, um estado de aceitação).

Algumas Simplificações

1. Utilizamos números para os estados no diagrama do identificador, mas a definição não restringe o conjunto de estados a números;
2. para evitar inúmeras transições, podemos usar abreviaturas de classes de caracteres se todos os membros tenham o mesmo estado origem e destino;
3. a definição exige que exista uma transição para cada par (s_i, c_i), porém podemos não representar aquelas que não levam a estados de aceitação.

Autômatos Finitos

Exemplos

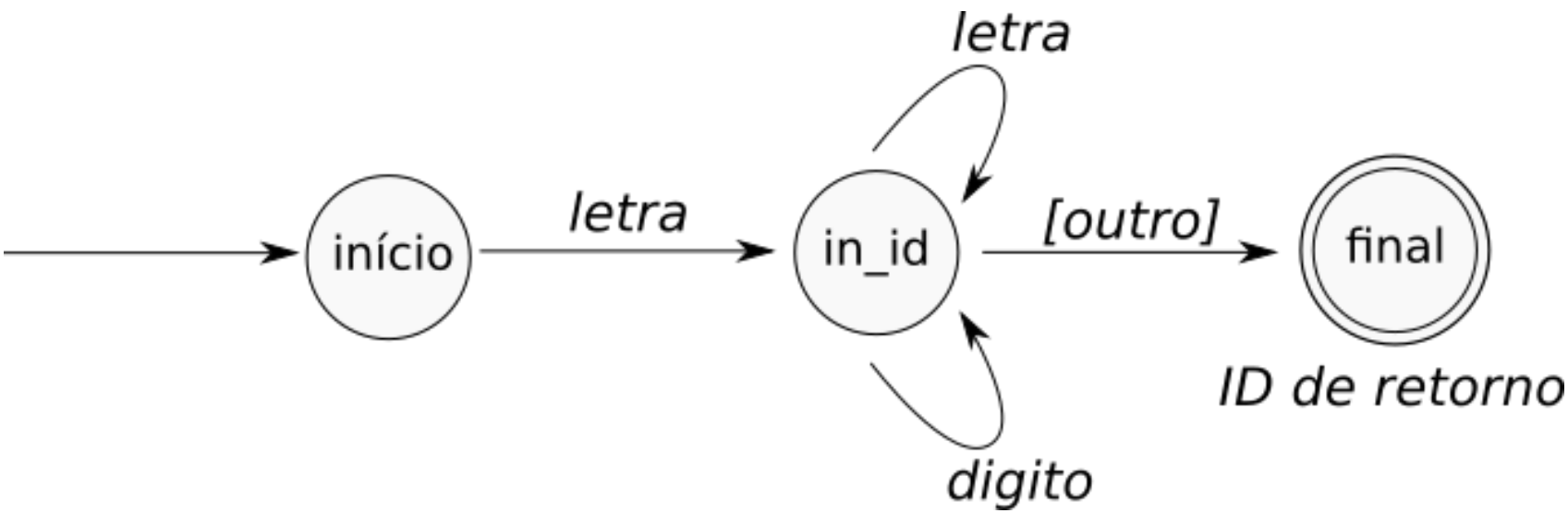
1. O conjunto de cadeias de caracteres que contém exatamente um *b*;
2. o conjunto de cadeias de caracteres que contém no máximo um *b*;
3. comentários da linguagem C;
4. constantes númericas em notação científica:

```
digito = [0-9]
nat = digito+
signedNat = (+|-)? nat
numero = signedNat("." nat)?(E signedNat)?
```

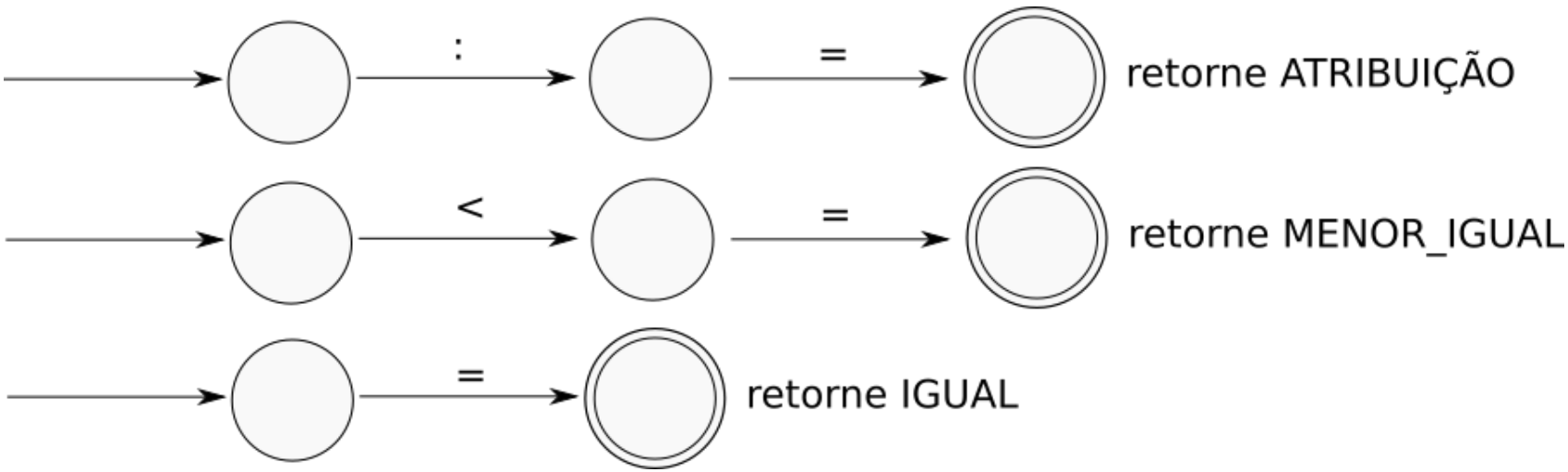
Verificação à Frente

Programa de Varredura que implementa o Autômato

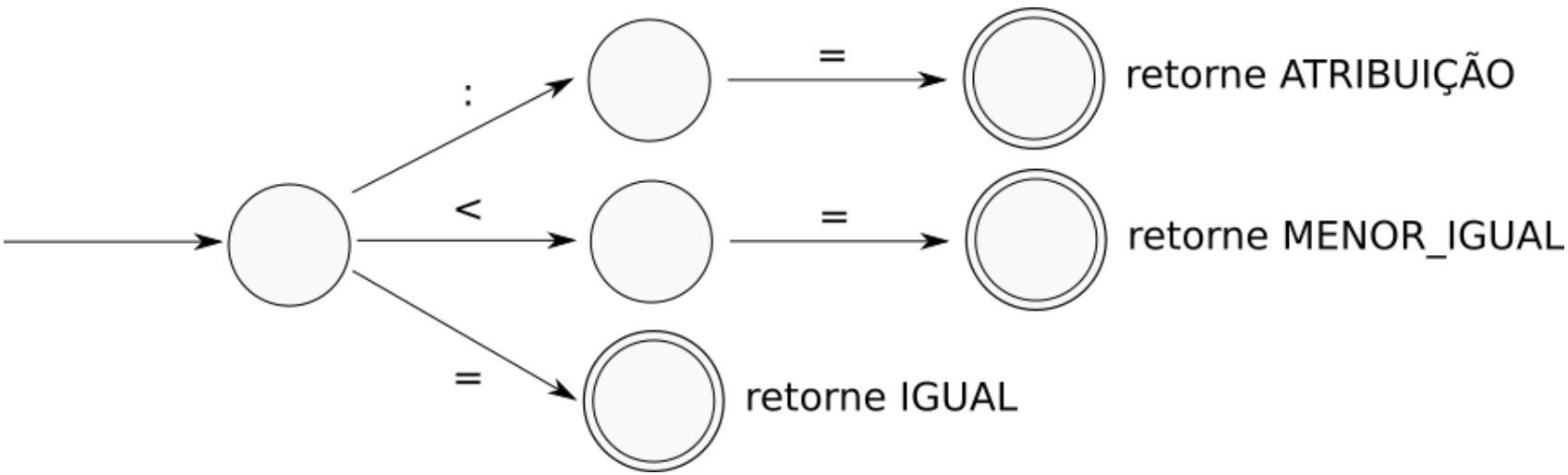
- Ação de transição: mover caractere da entrada para o lexema da marca;
- ação de aceitação: retornar a marca, lexema e outros atributos;
- ação de erro: voltar atrás na entrada ou gerar uma marca de erro.



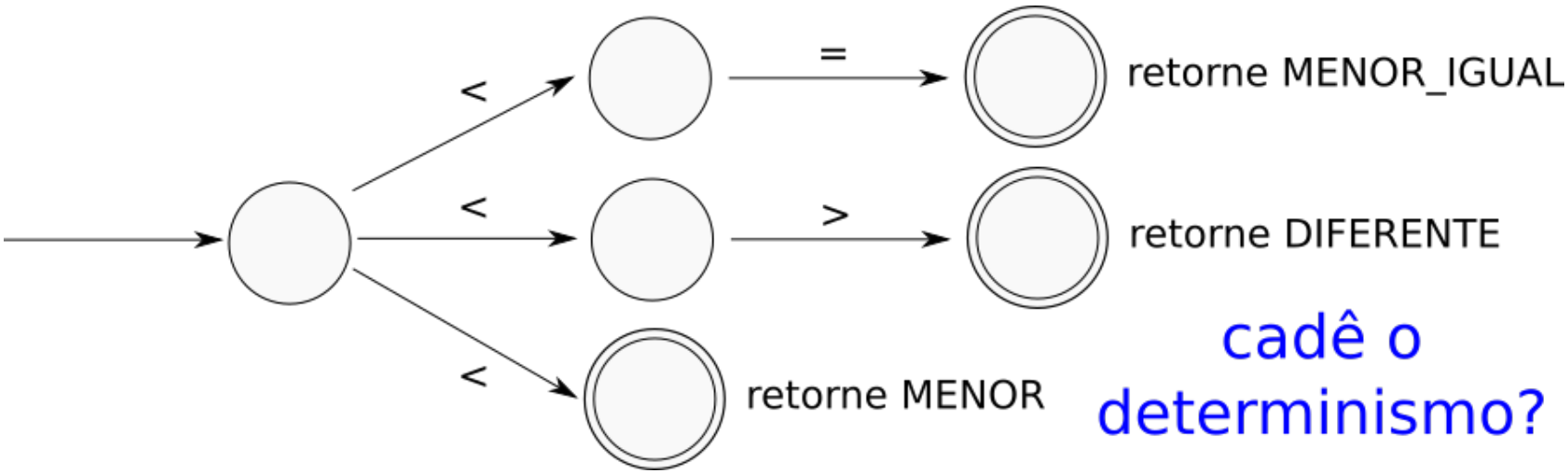
Marcas e Autômatos



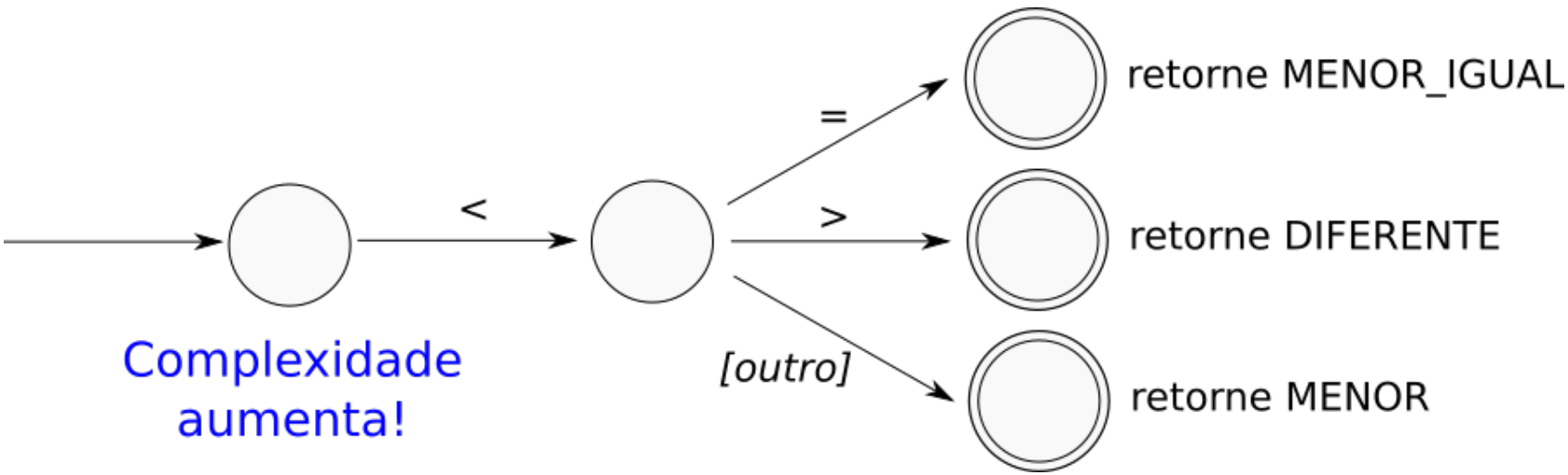
Marcas e Autômatos



Marcas e Autômatos



Marcas e Autômatos

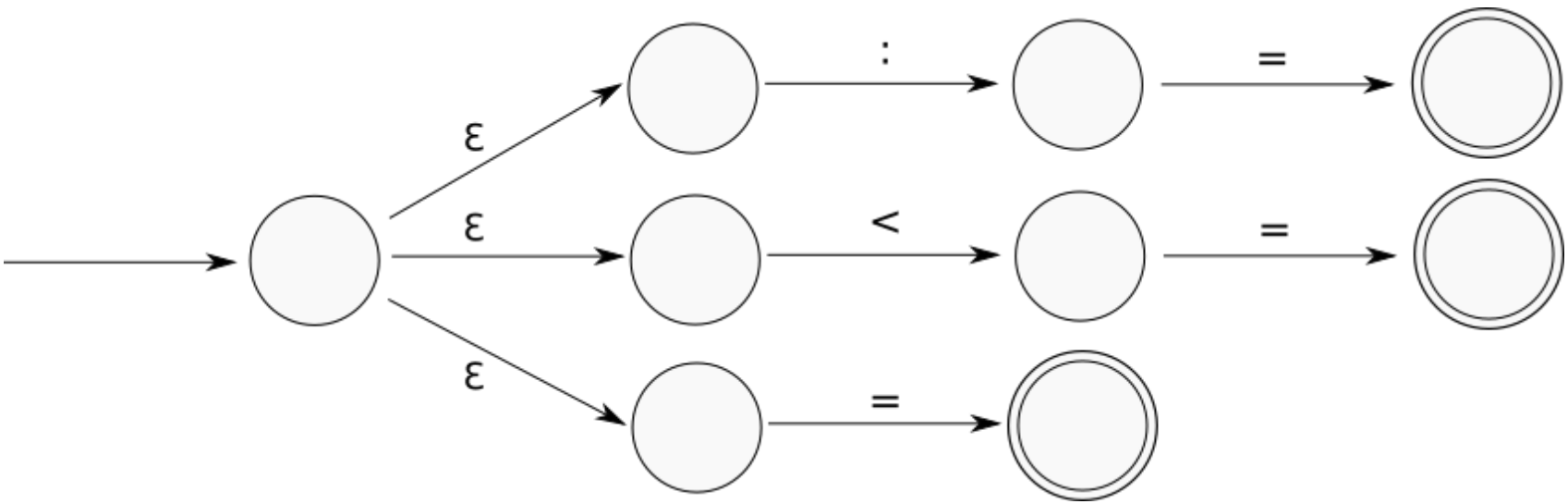


Autômato Finito Não Determinístico

Motivação

- Expandir a definição de DFA para incluir o caso de mais de uma transição de um estado para um caractere em particular;
- desenvolver algoritmo para transformar os novos autômatos em DFAs;
- ε-**transição** é uma transição que pode ocorrer sem consultar a cadeia de entrada.

Marcas e Autômatos



Autômato Finito Não Determinístico

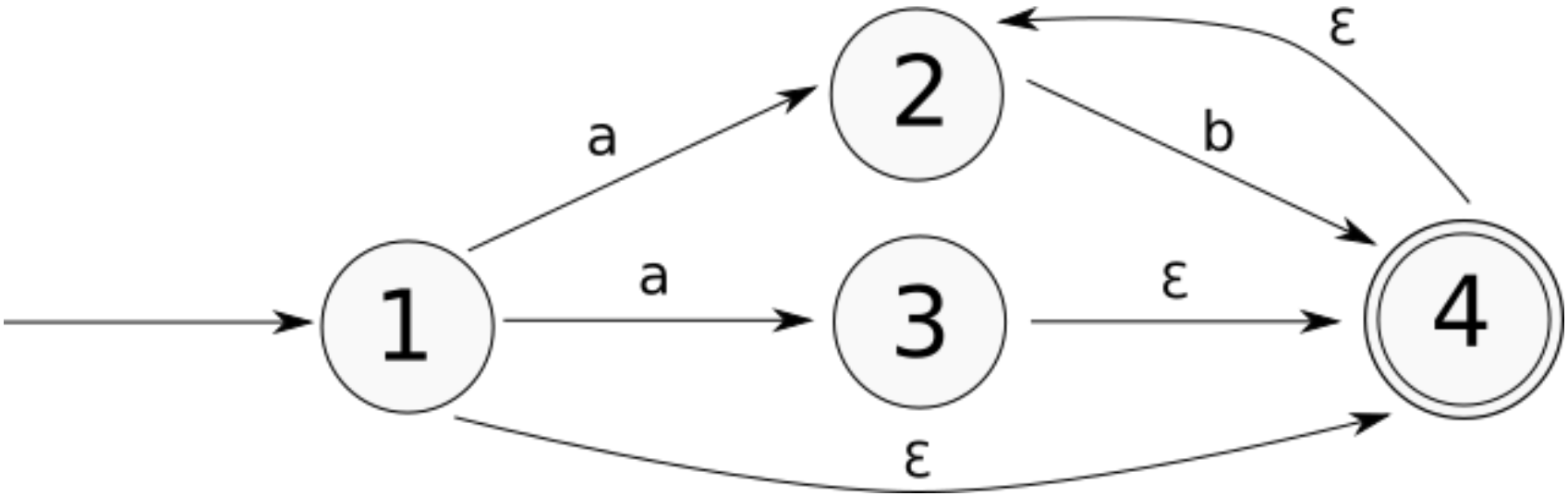
Definição

Um **NFA** (autômato finito não determinístico) é composto por:

- Um alfabeto Σ;
- um conjunto de estados S;
- uma função de transição $T : Sx\Sigma \cup \{\varepsilon\} \rightarrow \varphi(S)$;
- um estado inicial $s_0 \in S$;
- um conjunto de estados de aceitação $A \subset S$.

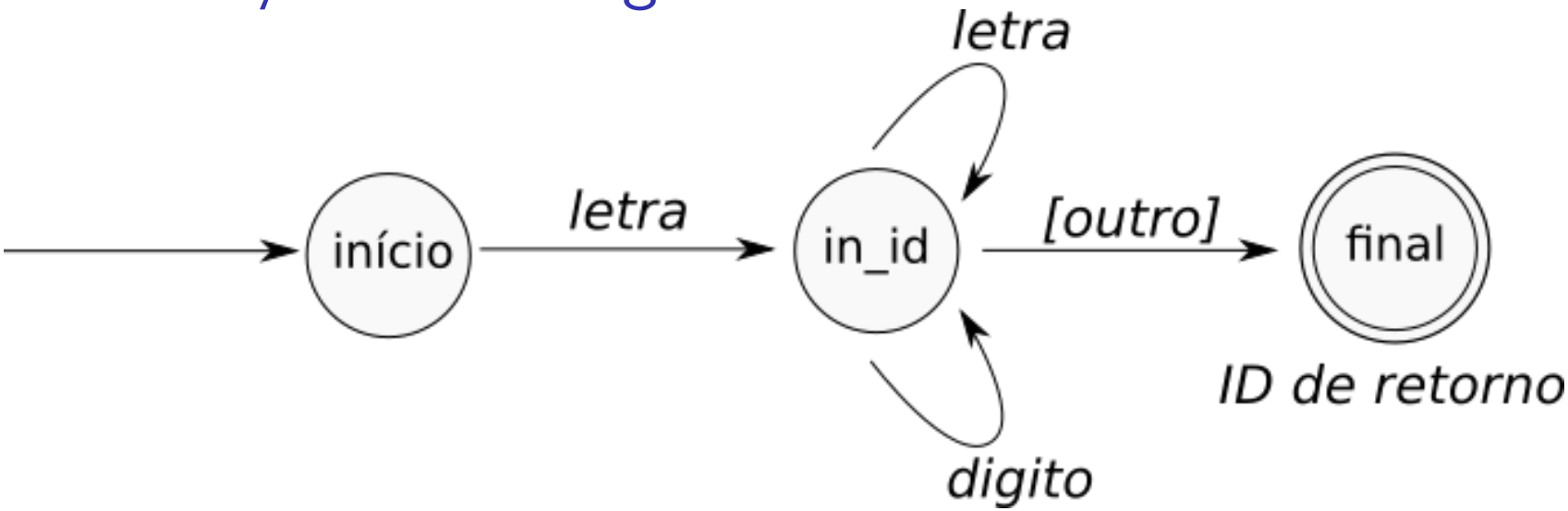
A linguagem aceita por M, denotada como $L(M)$, é definida como o conjunto de cadeias de caracteres $c_1c_2...c_n$ em que cada c_i de $\Sigma \cup \{\varepsilon\}$ para qual existam estados s_1 em $T(s_0, c_1)$, s_2 em $T(s_1, c_2)$, ... , s_n em $T(s_{n-1}, c_n)$, em que s_n seja elemento de A.

Autômato Finito Não Determinístico



Qual é a expressão regular?

Implementação em Código de Autômatos Finitos




```
{início - estado 1}
if proximo caractere letra then
   avance entrada;
   {in_id - estado 2}
   while proximo caractere letra ou digito do
      avance entrada; {permanece no estado 2}
   end while;
   {passa para o estado final - 3 - sem avancar na entrada}
   aceitacao;
else
   {erro ou outros casos}
end if;
```

Implementação em Código de Autômatos Finitos

- O estado é armazenado de forma implícita, de acordo com a posição do código na execução;
- é um código *ad hoc*, ou seja, foi criado a partir da observação das transições do autômato, mas sem obedecer a uma metodologia sistemática;
- para três estados, o código está coeso. Mas para uma quantidade maior de estados, a complexidade aumenta de forma considerável.

Implementação em Código de Autômatos Finitos

```
{estado 1}
if proximo caractere igual a / then
  avance entrada; {estado 2}
  if proximo caractere igual a * then
    avance entrada; {estado 3}
    fim := false;
    while not fim do
      while proximo caractere diferente de * do
        avance entrada;
      end while;
      avance entrada; {estado 4}
      while proximo caractere for * do
        avance entrada;
      end while;
      if proximo caractere igual a / then
        fim := true;
      end if;
      avance entrada;
    end while;
    aceitacao; {estado 5}
  else {outro processamento}
  end if;
else {outro processamento}
end if;
```

Usando *case/switch* para por Ordem no Caos

- Uma variável para manter o estado atual;
- um laço, com um primeiro *case* testando o estado e outro o caractere;

```
estado := 1; {inicio}
while estado = 1 ou estado = 2 do
  case estado of
  1: case caractere de entrada of
     letra: avance entrada;
            estado := 2;
     else estado := ... {erro ou outro};
     end case;
  2: case caractere de entrada of
     letra, digito: avance entrada;
                    estado := 2; {desnecessario}
     else estado := 3;
     end case;
  end case;
end while;
if estado = 3 then aceitacao else erro;
```

Usando *case/switch* para por Ordem no Caos

```
estado := 1; {inicio}
while estado = 1, 2, 3 ou 4 do
  case estado of
  1: case caractere de entrada of
     / : avance entrada;
         estado := 2;
     end case;
  2: case caractere de entrada of
     * : avance entrada;
         estado := 3;
     else estado := ... {erro ou outro};
     end case;
  3: case caractere de entrada of
     * : avance entrada;
         estado := 4;
     else avance entrada {continua no estado 3}
     end case;
  4: case caractere de entrada of
     / : avance entrada;
         estado := 5;
     * : avance entrada; {continha no estado 4}
     else avance entrada;
          estado := 3;
  end case;
end while;
if estado = 5 then aceitacao else erro;
```

Tabela de Transição

	Caracteres no alfabeto c
Estados s	Estados que representam transições $T(s, c)$

caractere de entrada **estado**	*letra*	*digito*	*outro*	*Aceitação*
1	2			não
2	2	2	[3]	não
3				sim

Algoritmo para Autômatos com Tabela de Transição

```
estado := 1;
ch := proximo caractere;
while not Aceita[estado] and not erro(estado) do
  novoestado := T[estado, ch].
  if Avance[estado, ch] then ch := proximo caractere;
  estado := novo estado;
end while
if Aceita[estado] then aceitacao;
```

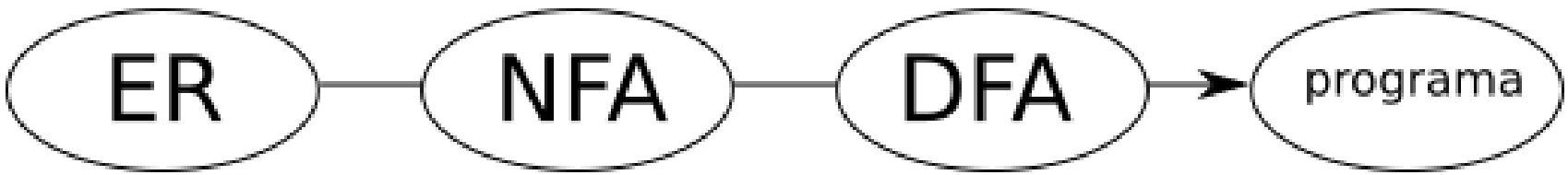
Observações

- Algoritmos dirigidos por tabela são simples, mas consomem muita memória;
- um algoritmo para NFA teria que armazenar vários fluxos de derivações, só retornando sucesso caso uma delas termine em aceitação;
- portanto, algoritmos que trabalham diretamente com NFAs são ineficientes. Melhor transformar o NFA em DFA e aplicar os algoritmos apresentados.

A Estrada de uma ER para um DFA

O caminho mais simples...

O algoritmo mais simples para traduzir uma ER para um DFA usa um NFA intermediário.

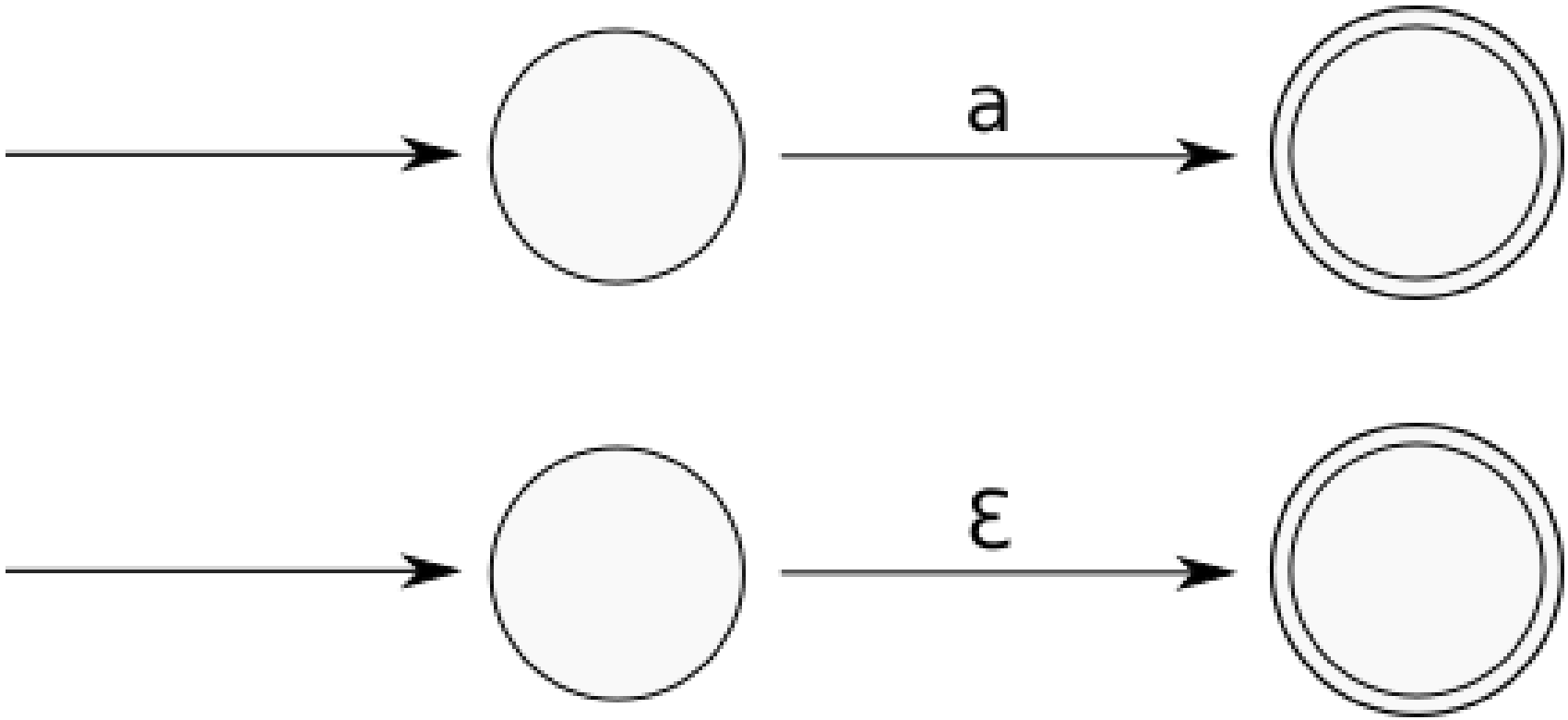


De uma ER para um NFA

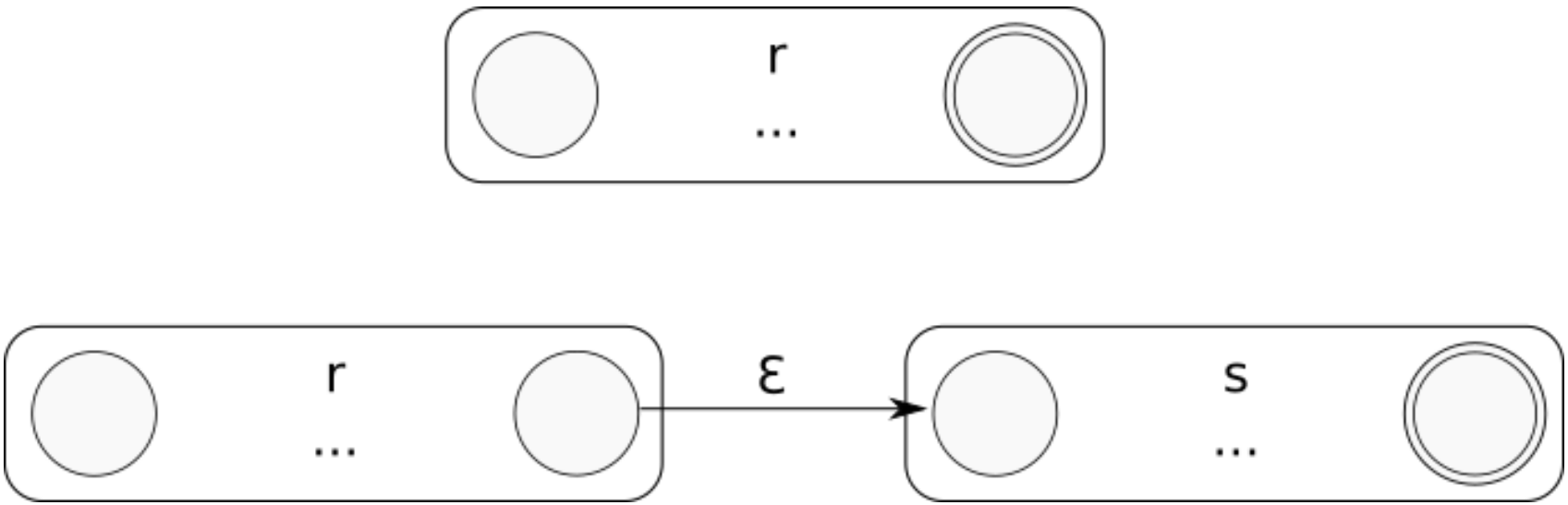
Construção de Thompson

Usamos transições ε para juntar máquinas de cada pedaço de uma expressão regular e formar uma máquina correspondente à expressão toda.

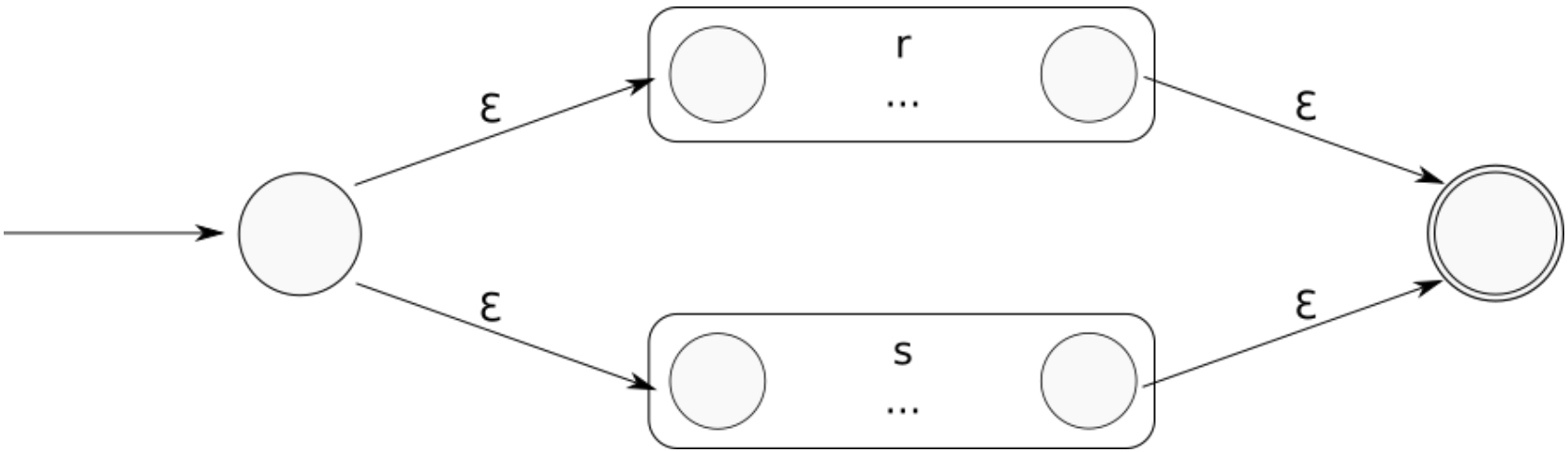
Expressões Regulares Básicas



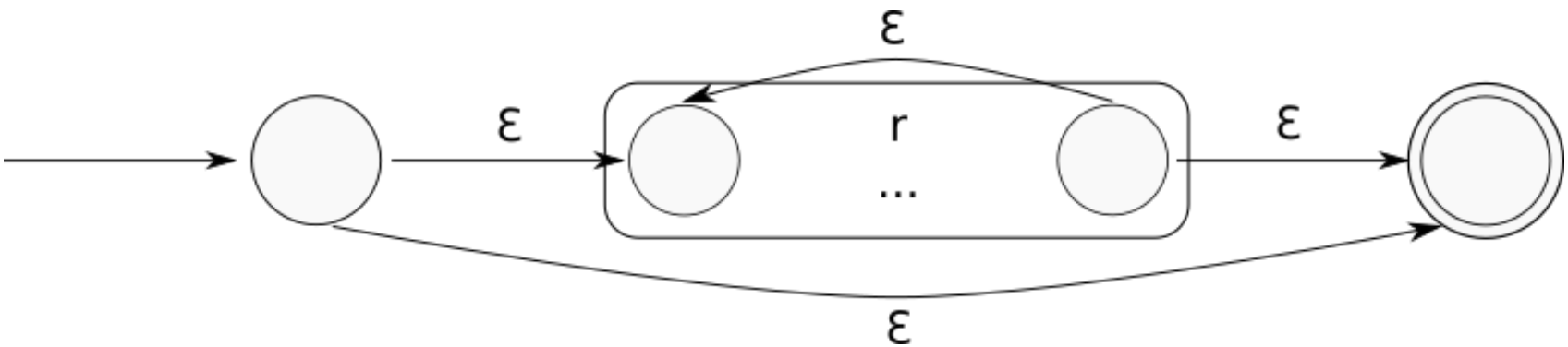
Concatenação



Escolha entre Alternativas



Repetição



Exemplos para Construção de Thompson

Criar um NFA para as ERs:

- $ab|a$;
- $letra(letra|digito)^*$.

De um NFA para um DFA

Construção de Subconjuntos

- Precisamos eliminar transições ε e múltiplas;
- um ε-**fecho** é o conjunto de todos os estados atingíveis por ε-transições a partir de um estado ou estados;
- a eliminação de transições múltiplas a partir de um estado para um caractere requer o acompanhamento do conjunto de estados atingíveis pelo casamento de um único caractere.

De um NFA para um DFA

O ε-fecho de um Conjunto de Estados

O ε-fecho de um único estado s é o conjunto de estados atingíveis por uma série de zero ou mais ε-transições, denominado $\bar{s}$. O ε-fecho de um conjunto de estados S é dado por: $\bar{S} = \bigcup_{s \in S} \bar{s}$.

Construção de Subconjuntos

Para construir um DFA $\bar{M}$ partir de um NFA M:

1. Definimos $S^{'} = \emptyset$ como o conjunto de estados de $\bar{M}$;
2. adicionamos a $S^{'}$ o estado inicial de $\bar{M}$ como o ε-fecho do estado inicial de M;
3. enquanto novos estados ainda forem criados:
   - 3.1 dado um caractere a do alfabeto defina $S^{'}_{a} = \{t |$ para algum $s \in S$ existe uma transição de s para t em $a\}$.
   - 3.2 defina $\bar{S}^{'}_{a}$ e o adicione a $S^{'}$.
   - 3.3 defina a nova transição $S \xrightarrow{a} \bar{S}^{'}_{a}$
4. todos os estados de $S^{'}$ que contenham um estado de aceitação de M são de aceitação em $\bar{M}$.

Exemplos para Construção de Subconjuntos

Exemplos

- a^*
- $ab|a$
- $letra(letra|digito)^*$

Minimização do Número de Estados de um DFA

Algoritmo de Hopcroft

- Dividir os estados do autômato original em dois conjuntos, aceitação e não aceitação;
- analisar quais transições ocorrem para os próprios dois novos estados e entre eles, considerando cada caractere do alfabeto;
- analisar, dentro de cada estado, quais transições distinguem os estados internos entre si;
- repartir os estados caso haja distinção;
- repetir o processo até não existir mais distinções.

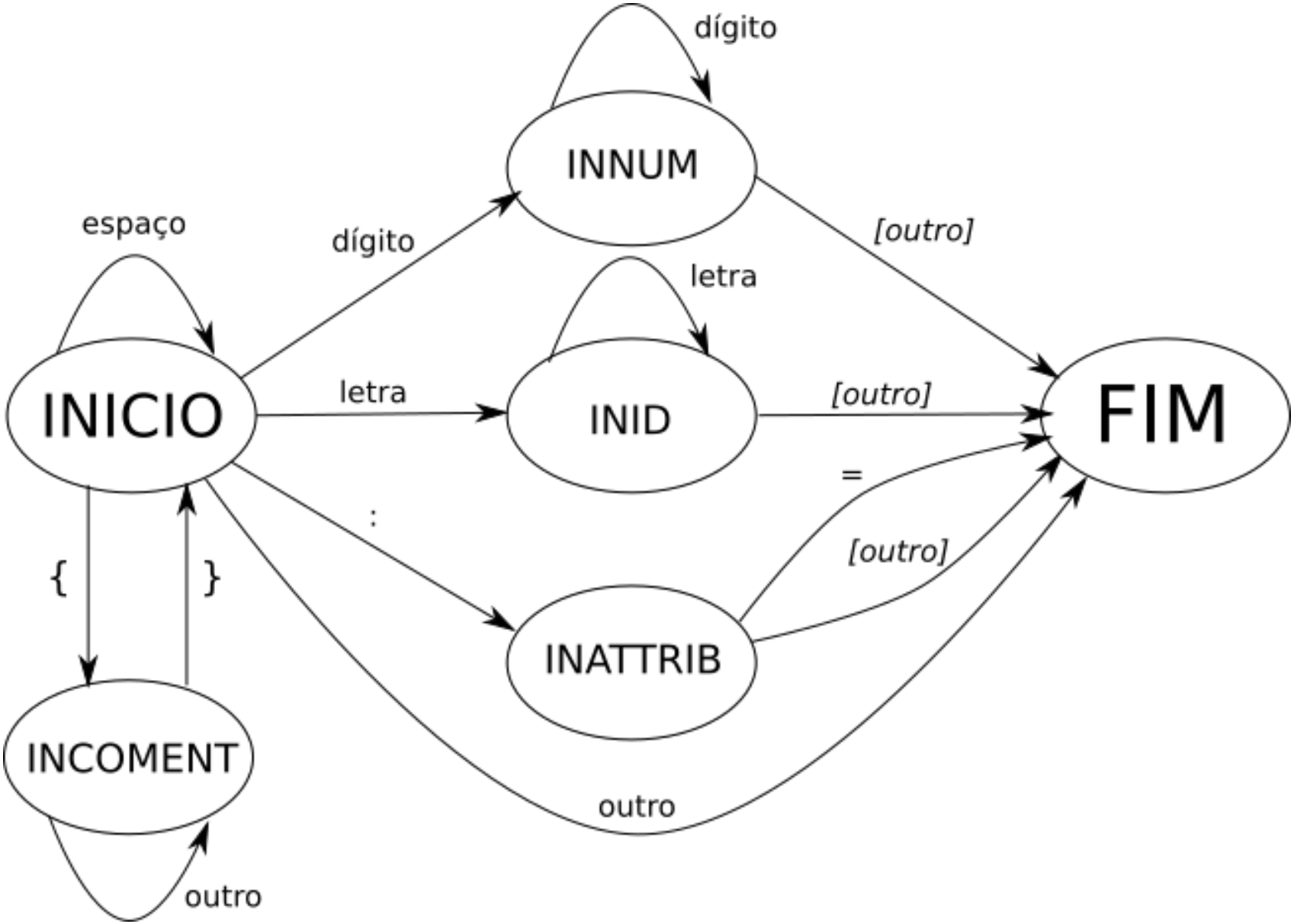
Marcas para a Linguagem TINY

Palavras Reservadas	Símbolos Especiais	Outras
if	+	*número*
then	-	(1 ou mais dígitos)
else	*	
end	/	
repeat	=	
until	<	*identificador*
read	(	(1 ou mais letras)
write	)	
	;	
	:=	

Comentários

Comentários são cercados por chaves $\{...\}$ e não pode ser aninhados.

DFA para a Linguagem TINY



Discussão do Código

- Arquivos **scan.h** e **scan.c** no Apêndice B do livro do Louden;
- o principal procedimento é getToken, consome caracteres de entrada e devolve a marca reconhecida de acordo com o DFA;
- utiliza a metodologia do *case* aninhado, com um *if-then-else* no lugar de um dos *case* para melhorar a legibilidade;
- as marcas estão definidas no arquivo **globals.h**;
- em TINY, o único atributo é o próprio *lexema* (ver *scan.h*);
- variáveis globais: *source, listing* e *lineno*;

Discussão do Código

Funcionamento de getToken

- A tabela **reservedWords** e o procedimento *reservedLookup* verificam palavras reservadas;
- uma variável de controle *save* é utilizada para indicar se um caractere será adicionado a **tokenString**;
- **getNextChar** vai adicionando caracteres a partir de **linebuf**.

Programa Exemplo na Linguagem TINY

```
{
   Programa de exemplo na
   linguagem TINY - computa o fatorial
}
read x; {entrada de um inteiro}
if 0 < x then { não calcula se x <= 0}
   fact := 1;
   repeat
      fact := fact * x;
      x := x - 1;
   until x = 0;
   write fact {saída do fatorial de x}
end
```

Uso do Lex para Gerar um Sistema de Varredura

- Vamos repetir a criação de um analisador léxico, agora usando uma ferramenta que automatiza o processo: *Lex*;
- é um programa que recebe como entrada um arquivo contendo **expressões regulares** e **ações** associadas a cada expressão;
- é gerado um procedimento *yylex*, implementação de um DFA;

Convenções Lex para Expressões Regulares

- Usar aspas para qualquer caractere que se deseje o casamento direto, seja metacaractere ou não;
- $* + ()|?$ tem o significado já discutido;
- $[abxz]$ e $[a - z]$ também são suportados;
- conjunto complementar: $[\hat{}0 - 9abc]$;
- dentro dos colchetes, a maioria dos metacaracteres perde seu *status* especial;
- uso de chaves para nomear expressões regulares;

Convenções Lex para Expressões Regulares

```
("aa"|"bb")("a"|"b")*"c"?

nat [0-9]+
signedNad (+|-)? {nat}
```

Formato de um Arquivo de Entrada Lex

Composto por três partes:

Uma coleção de **definições**, uma coleção de **regras** e uma coleção de **rotina auxiliares**

```
{definições}
%%
{regras}
%%
{rotinas auxiliares}
```

Formato de um Arquivo de Entrada Lex

```
%{
/* programa Lex para adicionar números
   de linhas a linhas de um texto, e
   imprimir o novo texto
*/
#include<stdio.h>
int lineno = 1;
%}
line .*\n
%%
{line} { printf("%5d %s", lineno++, yytext); }
%%
main()
{ yylex(); return 0;}
```

Gerando o Programa

```
$ lex exemplo220.lex
$ gcc lex.yy.c -ll -o exemplo220
exemplo220.lex:14:1: warning: return type defaults
  to ‘int’ [-Wimplicit-int]
 { yylex(); return 0;}
$ ./exemplo220 < exemplo220.lex
    1 %{
    2 /* programa Lex para adicionar números
    3    de linhas a linhas de um texto, e
    4    imprimir o novo texto
    5 */
...
```

Explicando a Entrada Lex

- O que está entre os delimitadores %{ e %} é inserido no código, sem alteração;
- logo em seguida definimos uma expressão regular com o nome *line*;
- após %%, temos a ação a ser tomada toda vez que a entrada casar com a expressão;
- **yytex** é o nome que Lex dá à cadeira que casa com a expressão;
- o código após o último %% é inserido ao final do código C gerado;
- **yylex** é o nome dado à função do DFA implementado.

Convenções do Lex

- Se um caractere ou cadeia de caracteres não casar com nenhuma expressão regular, Lex ecoará a entrada na saída;
- Lex sempre casa com a subcadeia mais longa;
- se a subcadeia mais longa casar com duas regras, Lex seleciona a primeira regra definida;

Nome	Significado
lex.yy.c ou lexyy.c	Arquivo de Saída
yylex	Função do DFA
yytext	Cadeia que casa com a ER
yyin	Entrada Lex
yyout	Saída Lex
input	Rotina de entrada
ECHO	Ação Básica

Finalizamos Varredura

Dúvidas?