

Análise Léxica

O chamado *Lexer*

Hervé Yviquel

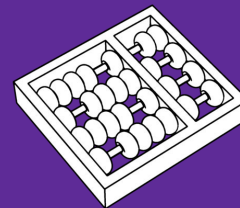
herve@ic.unicamp.br

Universidade Estadual de Campinas (Unicamp)

Instituto de Computação (IC)

Laboratório de Sistemas de Computação (LSC)

MC921 • Projeto e Construção de Compiladores • 2023 S2



UNICAMP

Aula Anterior

Resumo

- O que faz um compilador?
- História dos compiladores
- O que é um compilador?
- Estrutura dos compiladores

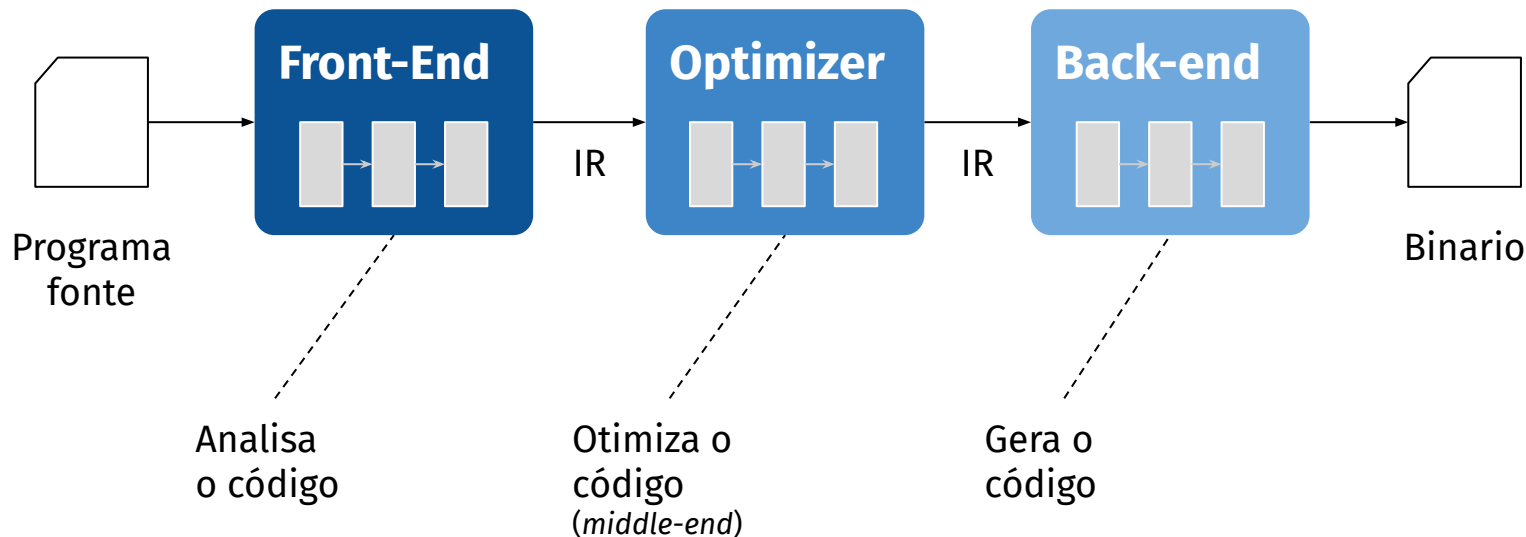
Aula de Hoje

Plano

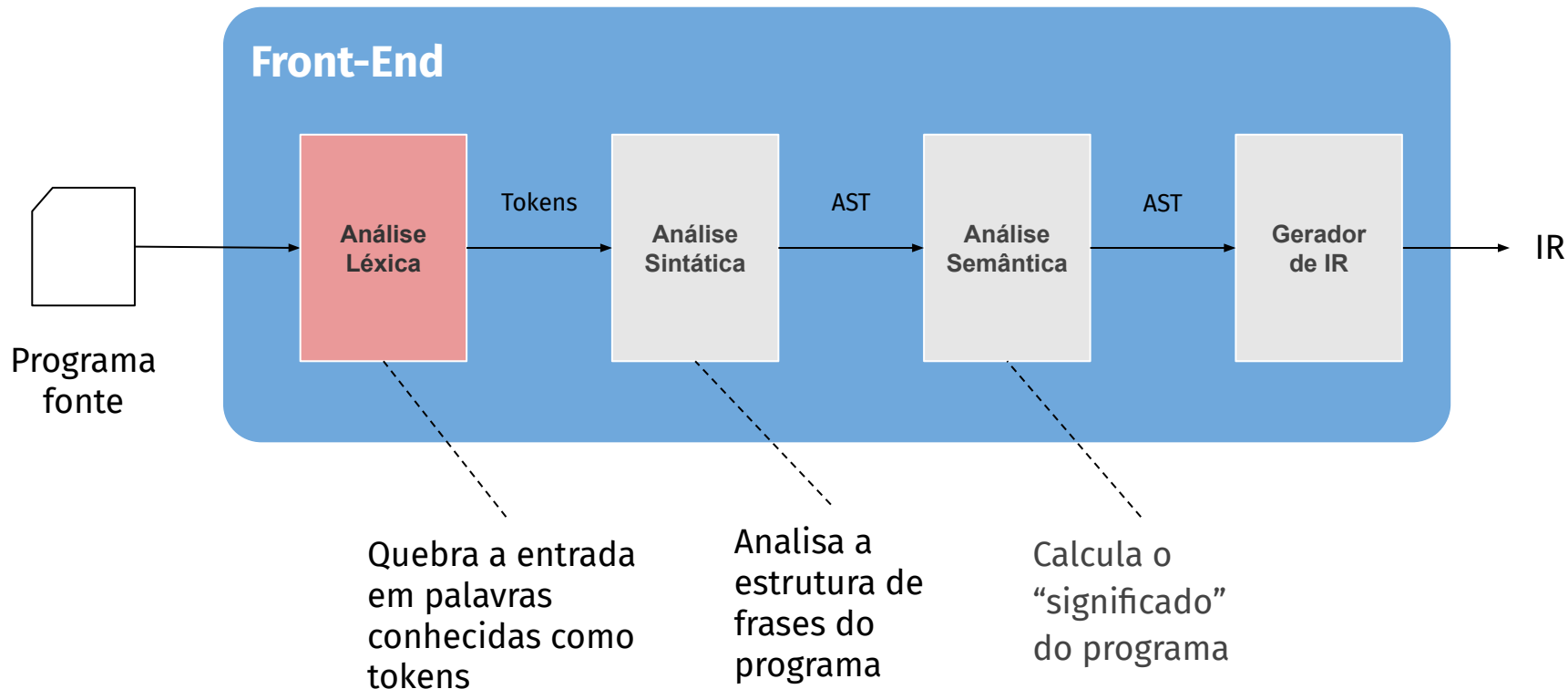
- Visão Geral do Front-End
 - Analizador Léxico
 - Especificação de Tokens
 - Gerador de Lexer
 - Autômato Finito Determinístico
 - Autômato Finito Não-determinístico
 - Simulação de NFA
 - Conversão de NFA em DFA
-

Visão Geral do Front-end



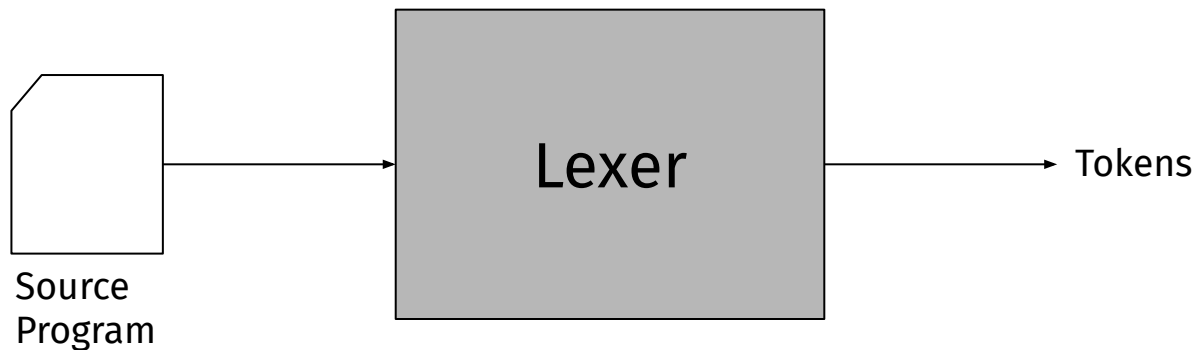


- O compilador traduz o programa de uma linguagem (fonte) para outra (máquina)
- Esse processo demanda sua quebra em várias partes, o entendimento de sua estrutura e significado
- O front-end do compilador é responsável por esse tipo de análise



Analizador Léxico





- Recebe uma sequência de caracteres e produz uma sequência de tokens
 - palavras chaves, pontuação e nomes
- Descarta comentários e espaços em branco


Tipo	Exemplos
ID	foo n14 last
NUM	73 0 00 515 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(

<i>comment</i>	<code>/* try again */</code>
<i>preprocessor directive</i>	<code>#include<stdio.h></code>
<i>preprocessor directive</i>	<code>#define NUMS 5, 6</code>
<i>macro</i>	<code>NUMS</code>
<i>blanks, tabs, and newlines</i>	

Obs.: O pré-processador passa pelo programa antes do léxico

```
float match0(char *s) /* find a zero */  
{  
    if (!strncmp(s, "0.0", 3))  
        return 0.;  
}
```

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN NOT ID(strncmp) LPAREN ID(s)
COMMA STRING(0.0) COMMA NUM(3) RPAREN
RPAREN RETURN REAL(0.0) SEMI RBRACE EOF



Retorno do lexer

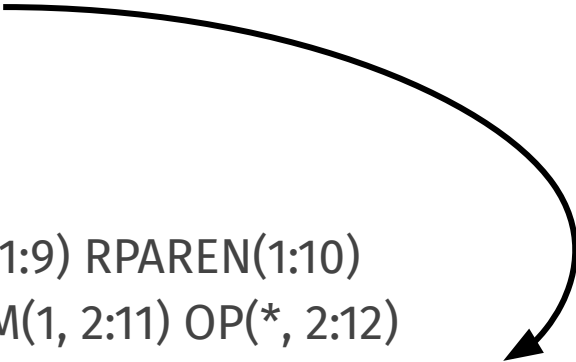
- Cada token tem um nome
- Alguns tokens têm um valor semântico associados a eles
 - IDs, NUMs, REALs, STRINGs, etc
- Também, cada token deve ser associado ao seu número de linha e coluna
 - Serão usado nas mensagens de erros e warning do compilador
 - Muito útil para o programador depurar

FLOAT(1:1) ID(match0, 1:7) ... RBACE(5:1) EOF

Tokeniza esse trecho de código:

```
int main() {  
    return 1*1;  
}
```

INT(1:0) ID(main, 1:4) LPAREN(1:9) RPAREN(1:10)
LBRACE(1:11) RETURN(2:4) NUM(1, 2:11) OP(*, 2:12)
NUM(1, 2:13) SEMI(2:14) RBRACE(3:1) EOF



- Um identificador é uma sequência de letras e dígitos
- O primeiro caractere deve ser uma letra
- O sublinhado _ conta como uma letra
- Letras maiúsculas e minúsculas são diferentes
- Se o fluxo de entrada foi analisado em tokens até um determinado caractere, o próximo token inclui a cadeia de caracteres mais longa que poderia constituir um token
- Espaços em branco, tabs, novas linhas e comentários são ignorados, exceto quando servem para separar tokens
- Algum espaço em branco é necessário para separar identificadores, palavras-chave e constantes adjacentes

**Como os tokens são
especificados?**

Especificação de Tokens



- Alfabeto = $\{ 'a', 'b', 'c', \dots, '0', '1', '2', \dots, '/', '%', '>', \dots \}$
- S é o conjunto de strings válidas em uma linguagem
 - S da linguagem C = $\{ "if", "case", "{", "break", ":", ")", \dots \}$
- Dado um programa fonte deseja-se saber se todas as strings do programa pertencem a S
- String válida em uma linguagem
 - Uma string é válida se definida por um expressão regular correta

- Álgebra matemática que define regras de formação de strings
 - é também chamado de RE, REGEX ou RegExp
- A especificação de expressões regulares é muito parecida entre as diferentes linguagens de programação que possuem o conceito de expressões regulares
 - Mas pode ter leve diferença sintática

- **Símbolo:** Para cada símbolo a no alfabeto da linguagem
 - $a = \{a\}$
- **Alternância:** Dadas duas expressões regulares M e N , o operador de alternância ($|$) gera uma nova expressão $M | N$
 - $a | b = \{a, b\}$
- **Concatenação:** Dadas duas expressões regulares M e N , o operador de concatenação (\cdot) gera uma nova expressão $M \cdot N$
 - $(a | b) \cdot a = \{aa, ba\}$

- **Epsilon:** A expressão regular ϵ representa a linguagem cuja única string é a vazia
 - $(a \cdot b) \mid \epsilon = \{ "", "ab" \}$
- **Repetição:** Dada uma expressão regular M , seu Kleene closure é M^* . Uma string está em M^* se ela é a concatenação de zero ou mais strings, todas em M .
 - $((a \mid b) \cdot a)^* = \{ "", "aa", "ba", "aaaa", "baba", "aaaaaa", \dots \}$

- $(0 \mid 1)^* \cdot 0$
 - Números binários múltiplos de 2.
- $b^*(abb^*)^*(a|\epsilon)$
 - Strings de a's e b's sem a's consecutivos.
- $(a|b)^*aa(a|b)^*$
 - Strings de a's e b's com a's consecutivos.

- a Um caractere comum representa a si mesmo
- ϵ A string vazia
- Outra maneira de escrever a string vazia
- $M \mid N$ Alternância, escolhendo entre M ou N
- $M \cdot N$ Concatenação, um M seguido de um N
- MN Outra maneira de escrever concatenação
- M^* Repetição (zero ou mais vezes)
- M^+ Repetição, uma ou mais vezes
- $M?$ Opcional, zero ou uma ocorrência de M
- $[a - zA - Z]$ Alternância do conjunto de caracteres
- $.$ Um ponto final representa qualquer caractere único, exceto nova linha
- `"a.+*"` Citação, uma string entre aspas significa literalmente

Como seriam as expressões regulares para os seguintes tokens?

- IF
 - if
- ID
 - `[a-z][a-z0-9]*`
- NUM
 - `[0-9]+`

Quais tokens representam as seguintes expressões regulares?

- `([0-9]+\".\"[0-9]*)|([0-9]*\".\"[0-9]+)`
 - REAL
- `(\"//\"[a-z]*\"\\n\")|(\" \"|\"\\n\"|\"\\t\")+`
 - nenhum token, somente comentário, brancos, nova linha e tab

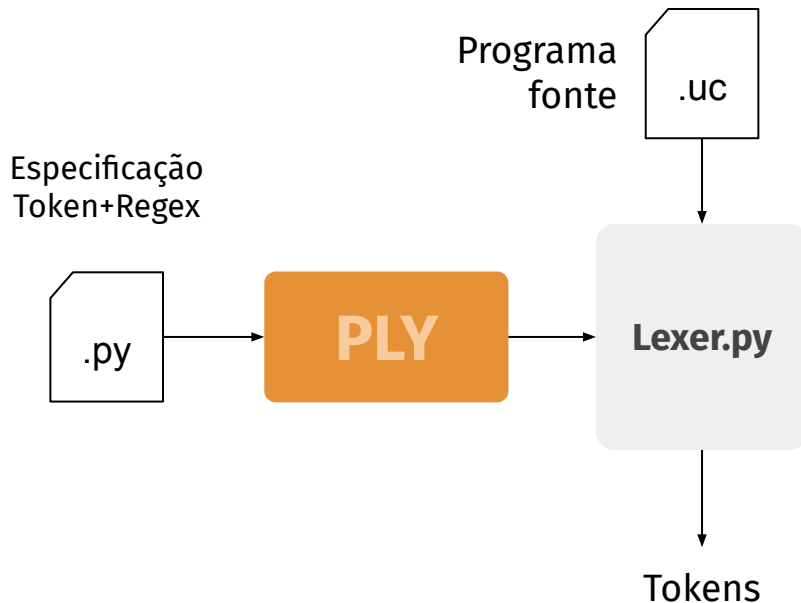
Gerador de Lexer



- Os tokens e as expressões regulares correspondentes são listados
 - em ordem
- Esse arquivo serve como entrada do gerador de lexer
 - por exemplo, a ferramenta *lex*

```
if      :  if
id      :  [a-z][a-z0-9]*
num     :  [0-9]+
...
```

- O gerador de lexer gera o lexer automaticamente a partir da especificação
- Tem vários geradores de lexer disponíveis
 - Lex, Flex, ANTLR, PLY, ...
- Usarão PLY no projeto 1
 - Gerar um lexer para a linguagem uC



- A especificação deve ser completa
 - sempre reconhecendo uma substring da entrada
- Mas quando estiver errada?
 - Use uma regra com o “.”
- Em que lugar da sua especificação deve estar esta regra?
 - Esta regra deve ser a última! (Por que?)

```
if      :  if
id      :  [a-z][a-z0-9]*
num     :  [0-9]+
...
error  :  .
```

if8 é um ID ou dois tokens IF e NUM(8) ?

if 89 começa com um ID ou uma palavra-reservada?

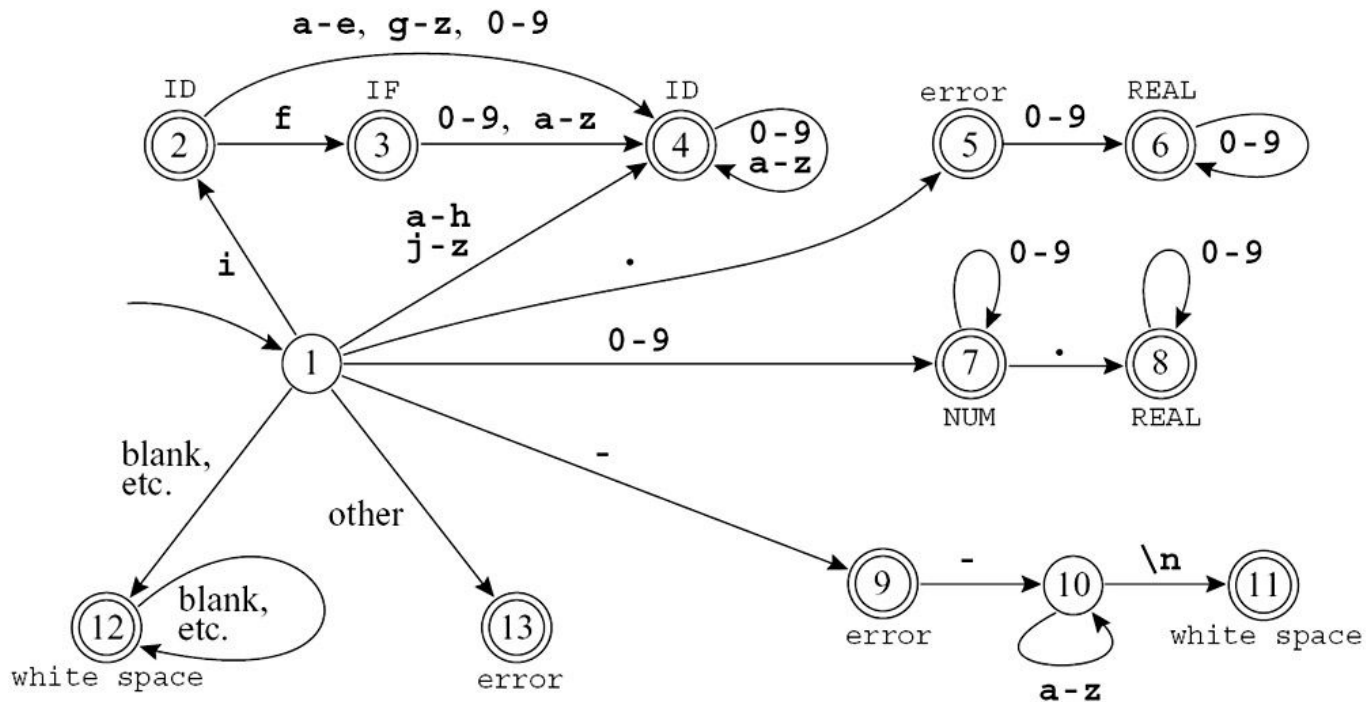
1. Maior casamento

- O próximo token sempre é a substring mais longa possível a ser casada

2. Prioridade

- Para uma dada substring mais longa, a primeira regra a ser casada produzirá o token
- A ordem da regras é importante

**Como funciona
o gerador de lexer?**



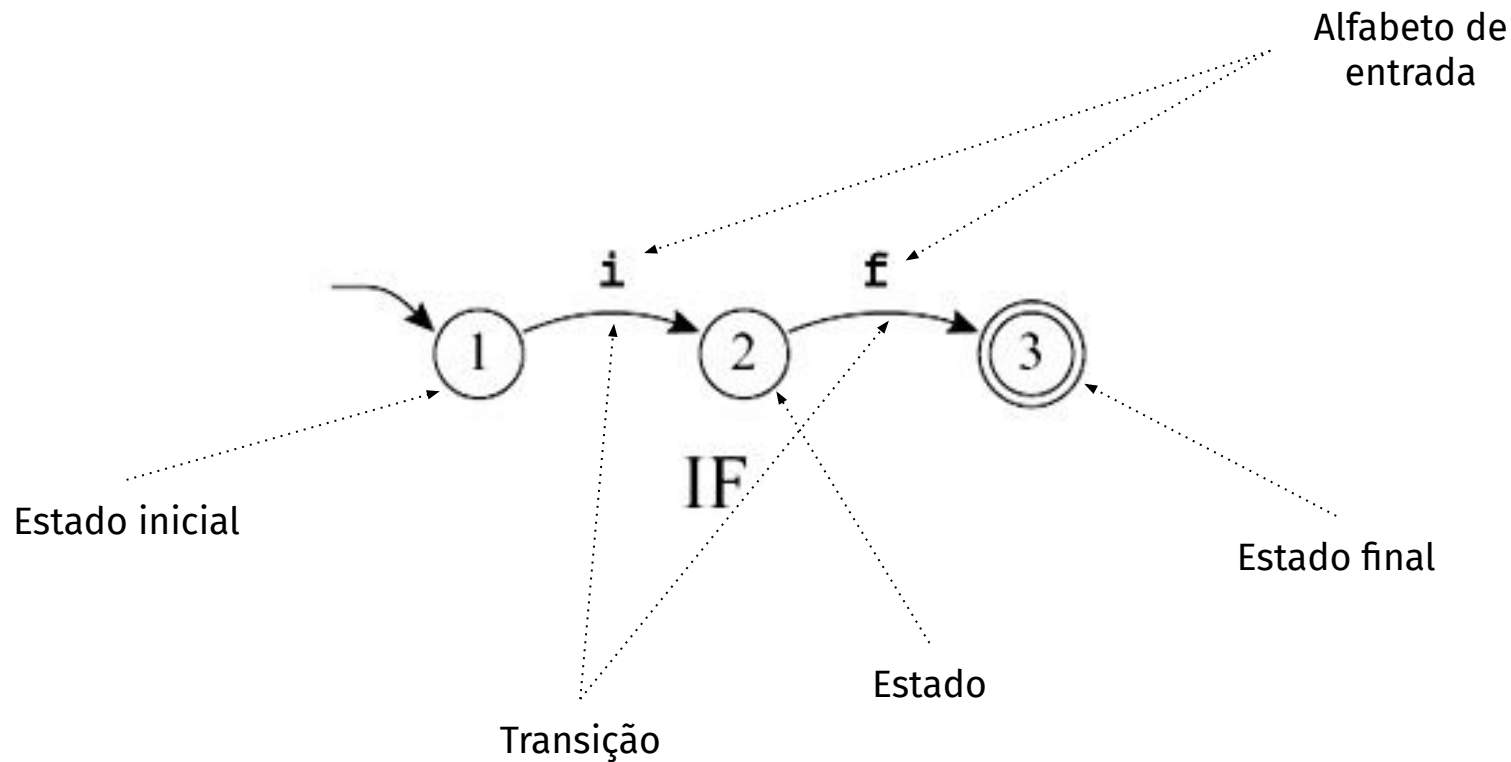
Autômatos Finitos (ou máquina de estados finita)

Autômatos Finitos



- Usamos expressões regulares para dar a especificação léxica da linguagem
 - Expressões Regulares são convenientes para especificar os tokens
- Mas como podemos fazer a implementação do analisador léxico a partir dessa especificação?
 - Precisamos de um formalismo que possa ser convertido em um programa de computador
 - Este formalismo são os autômatos finitos
- Algoritmos para converter expressões regulares são conhecidos e podem ser reaproveitados
 - Usar autômatos levam a um analisador léxico bastante eficiente

- Um autômato finito possui:
 - Um alfabeto de entrada
 - Um conjunto finito de estados
 - Um estado inicial
 - Um ou mais estados finais rotulados
 - Um conjunto de transições entre estados
 - Arestas levando de um estado a outro, anotada com um símbolo



- Uma transição $s1 \rightarrow s2$ quer dizer que se autômato está no estado $s1$ e o próximo símbolo da entrada é a então ele vai para o estado $s2$
- 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
- Quando aceitou uma entrada, o autômato volta no estado inicial

```
if
[a-z] [a-z0-9] *
[0-9] +
( [0-9] + "." [0-9] *) | ( [0-9] * "." [0-9] + )
( "-" [a-z] * "\n" ) | ( " " | "\n" | "\t" ) +
.
```

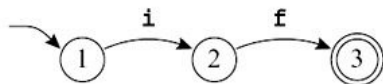
IF

ID

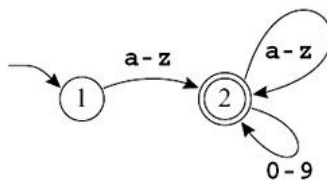
NUM

REAL

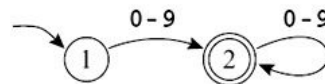
no token, just white space
error



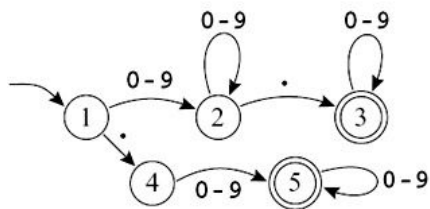
IF



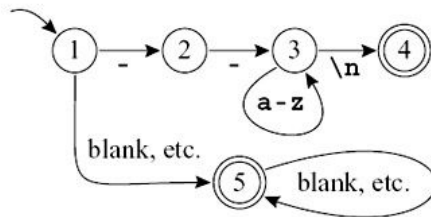
ID



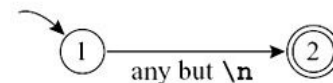
NUM



REAL

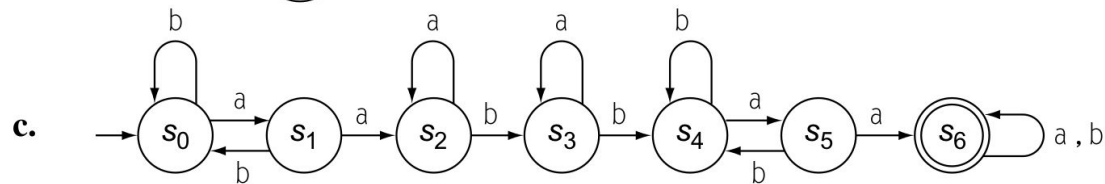
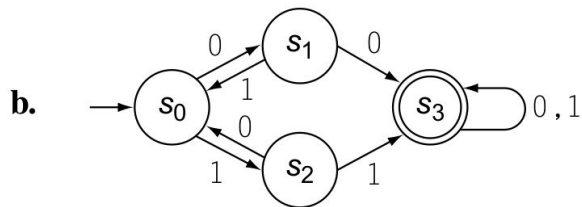
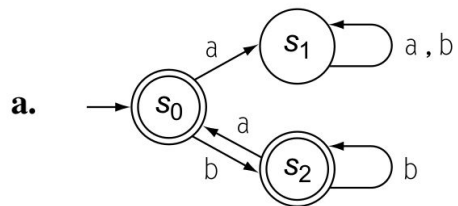


white space



error

Descreva informalmente os idiomas aceitos por:



Construa um autômato aceitando cada uma das seguintes linguagens:

- $\{w \in \{a, b\}^* \mid w \text{ começa com 'a' e contém 'baba' como uma substring}\}$
- $\{w \in \{0, 1\}^* \mid w \text{ contém '111' como substring e não contém '00' como substring}\}$

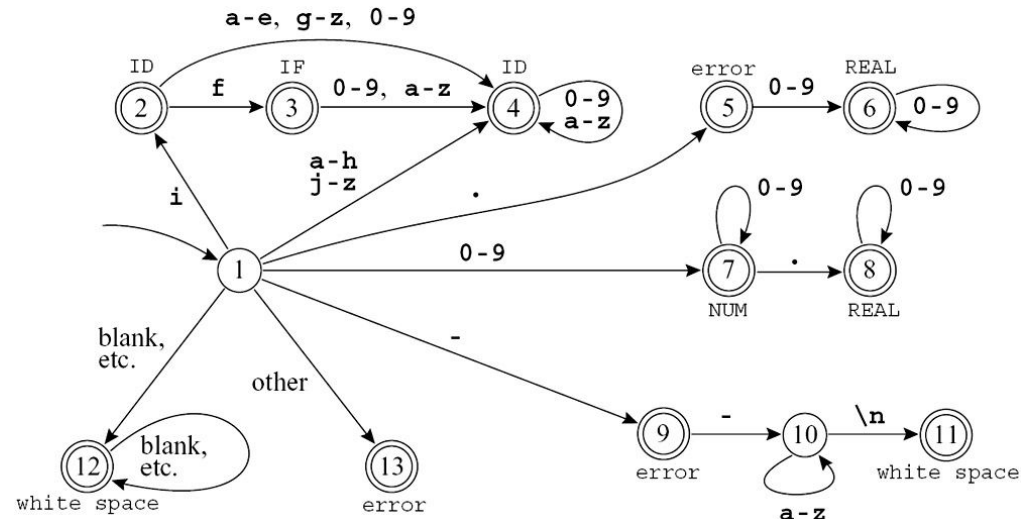
Autômato Finito Determinístico

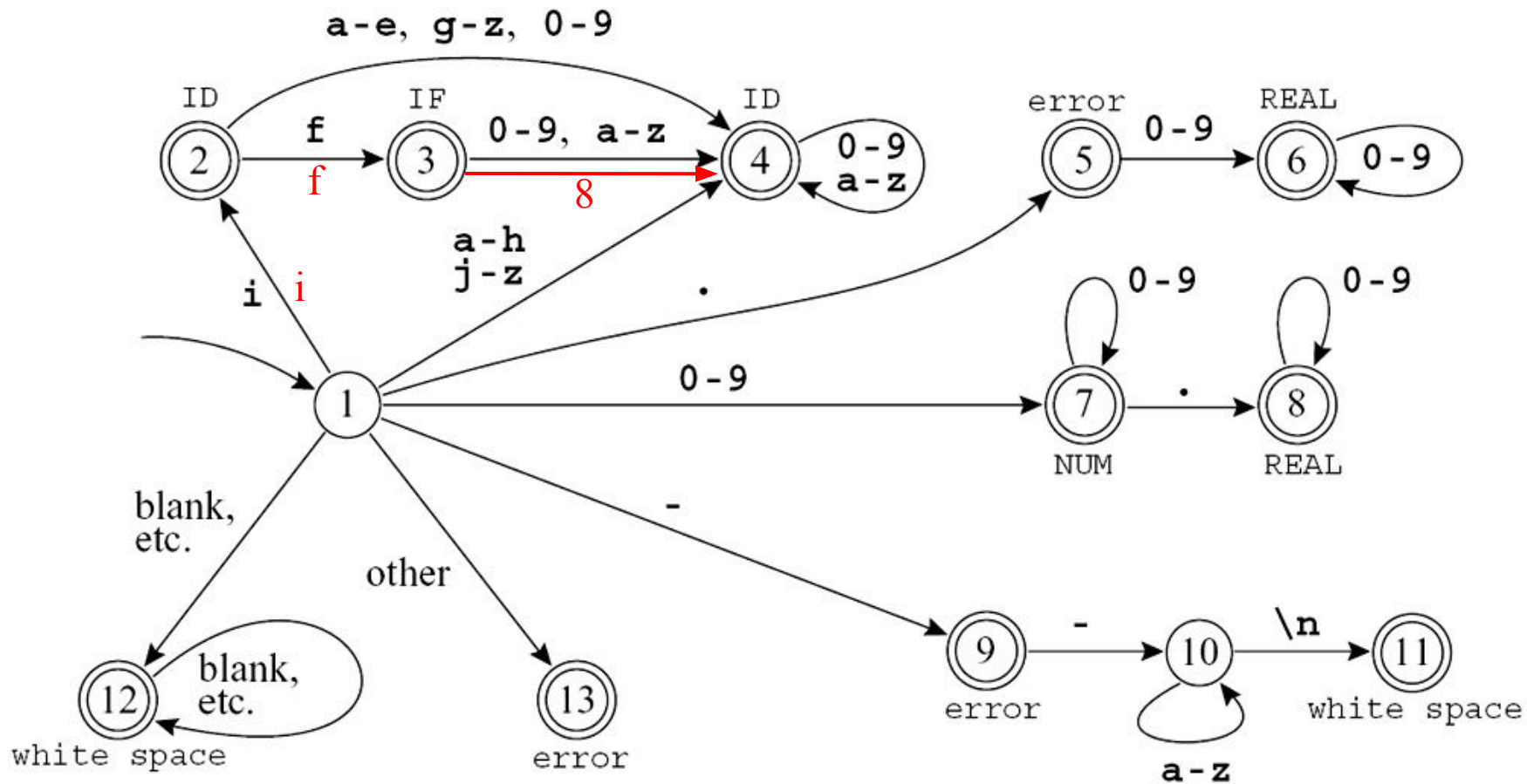


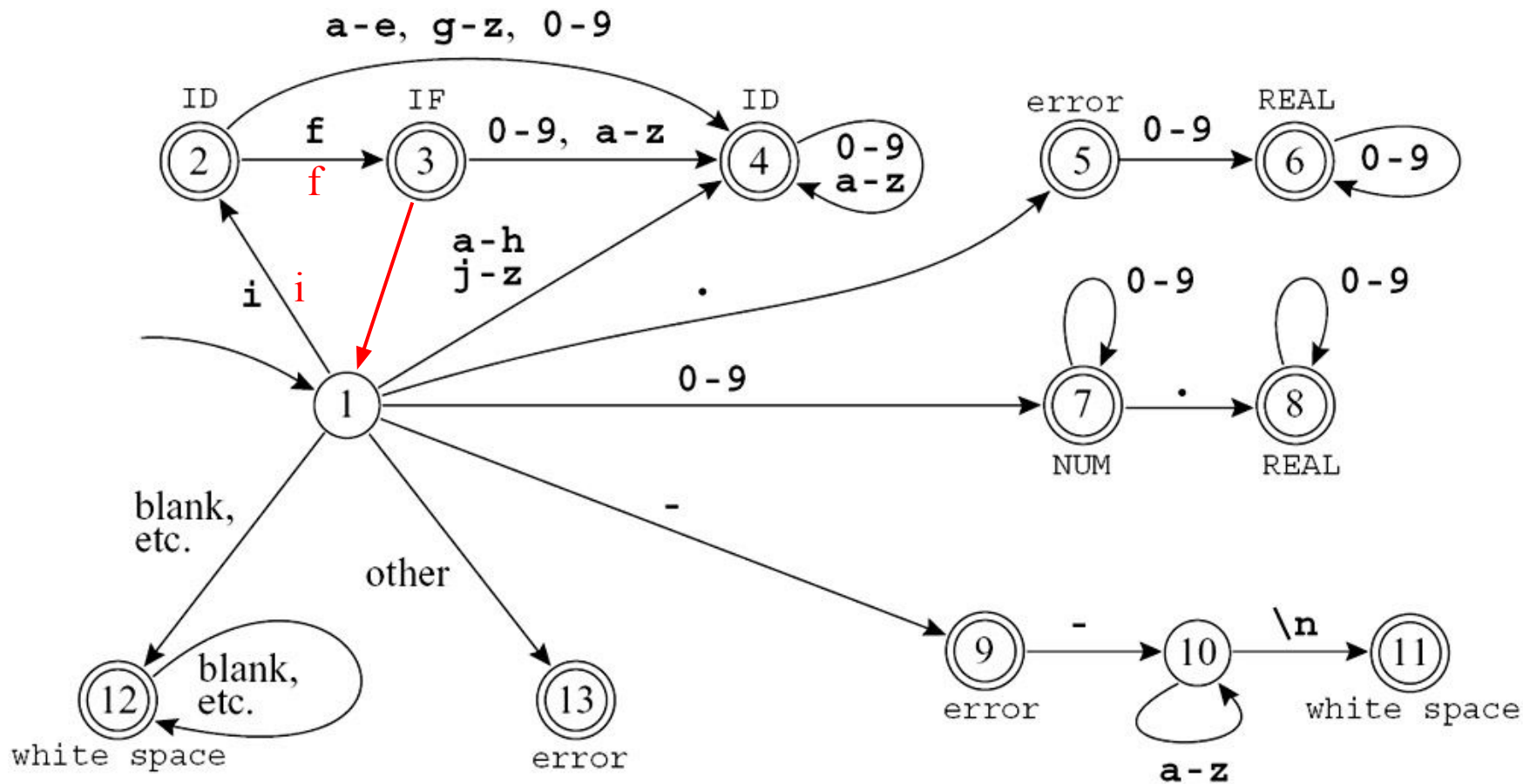
- DFAs não podem apresentar duas arestas que deixam o mesmo estado, anotadas com o mesmo símbolo
- Saindo do estado inicial, o autômato segue exatamente uma aresta para cada caractere da entrada
- O DFA aceita a string se, após percorrer todos os caracteres, ele estiver em um estado final

- Se em algum momento não houver uma aresta a ser percorrida para um determinado caractere ou ele terminar em um estado não-final, a string é rejeitada
- A linguagem reconhecida pelo autômato é o conjunto de todas as strings que ele aceita
- Podemos combinar os autómatos definidos para cada token de maneira a ter um único autômato que possa ser usado como analisador léxico?
 - Veremos um exemplo ad-hoc e mais adiante mecanismos formais para esta tarefa

- Estados finais nomeados com o respectivo token
- Alguns estados apresentam características de mais de um autômato anterior
 - exemplo: 2
- Como ocorre a quebra de ambigüidade entre ID e IF?

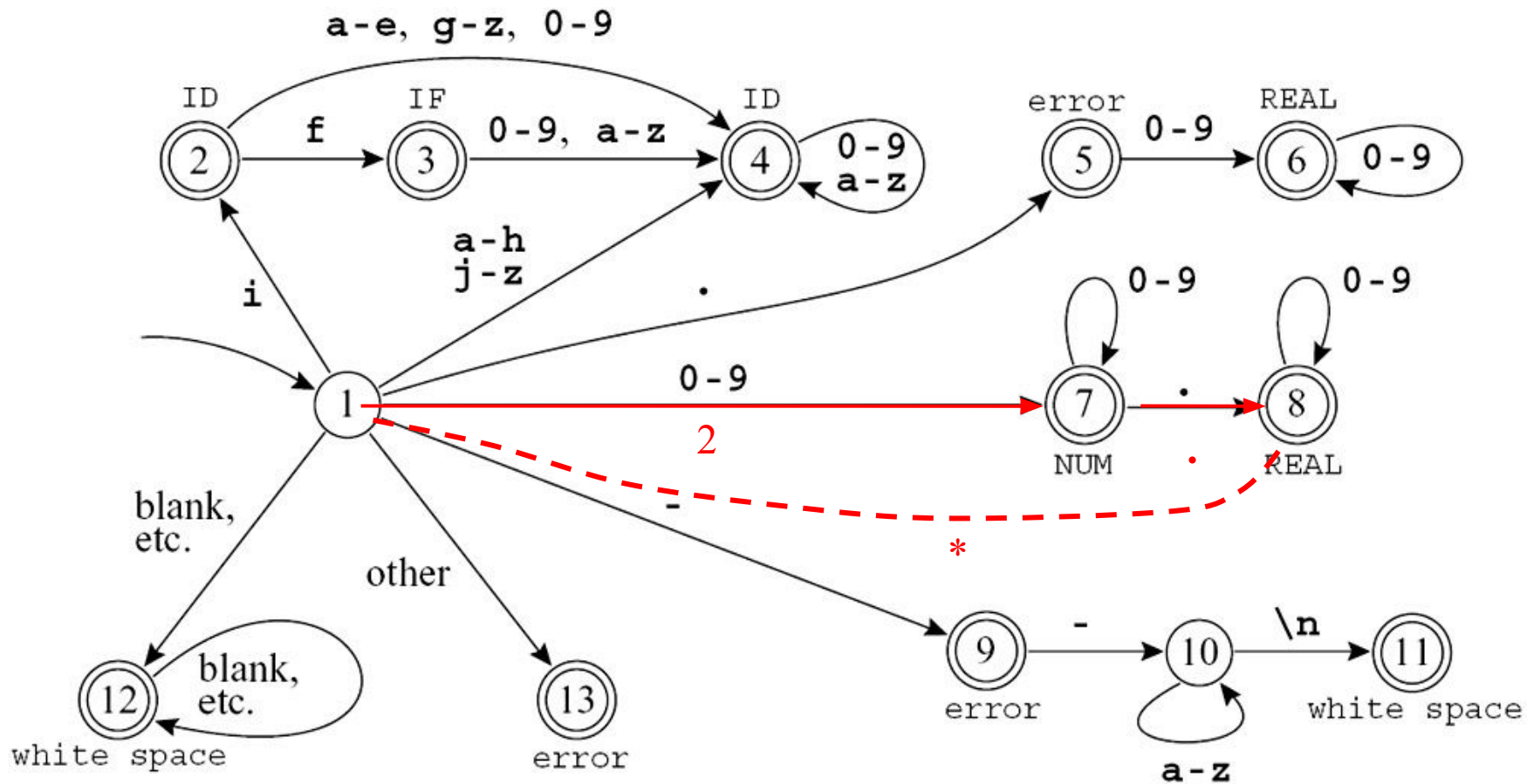






Exemplo – 2.2*

48



Podemos usar um matriz como *transition table*

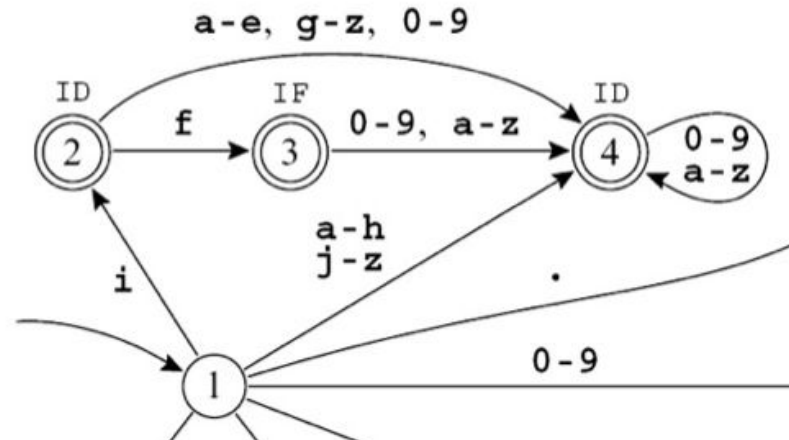
```
int edges[ ][ ] = { /* ... 012 ... - ... e f g h i j ... */
/* state 1 */ {0,0, ... 7,7,7 ... 9 ... 4,4,4,4,2,4 ... },
/* state 2 */ {0,0, ... 4,4,4 ... 0 ... 4,3,4,4,4,4 ... },
/* state 3 */ {0,0, ... 4,4,4 ... 0 ... 4,4,4,4,4,4 ... },
/* state 4 */ {0,0, ... 4,4,4 ... 0 ... 4,4,4,4,4,4 ... },
/* state 5 */ {0,0, ... 6,6,6 ... 0 ... 0,0,0,0,0,0 ... },
/* state 6 */ {0,0, ... 6,6,6 ... 0 ... 0,0,0,0,0,0 ... },
/* state 7 */ {0,0, ... 7,7,7 ... 0 ... 0,0,0,0,0,0 ... },
/* state 8 */ {0,0, ... 8,8,8 ... 0 ... 0,0,0,0,0,0 ... },
  etc }
```

- A tabela anterior é usada para aceitar ou recusar uma string
- Porém, precisamos garantir que a maior string seja reconhecida
- Necessitamos de duas informações
 - Último estado final
 - Posição da entrada no último estado final

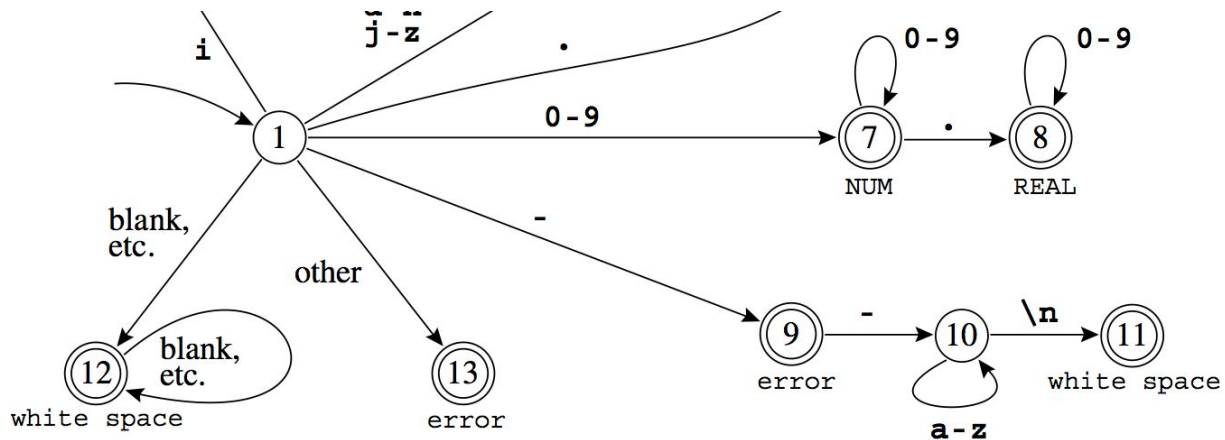
Last Final	Current State	Current Input	Accept Action
0	1	<u>I</u> f --not-a-com	<i>return IF</i>
2	2	I <u>I</u> f --not-a-com	
3	3	I f <u>I</u> --not-a-com	
3	0	I f I <u>-</u> not-a-com	
0	1	i f <u>I</u> --not-a-com	<i>found white space; resume</i>
12	12	i f I <u>I</u> --not-a-com	
12	0	i f I <u>-</u> not-a-com	
0	1	i f <u>I</u> --not-a-com	<i>error, illegal token '-' ; resume</i>
9	9	i f <u>I</u> not-a-com	
9	10	i f <u>I</u> not-a-com	
9	10	i f <u>I</u> not-a-com	
9	10	i f <u>I</u> not-a-com	
9	10	i f <u>I</u> not-a-com	
9	0	i f <u>I</u> not-a-com	
0	1	i f <u>I</u> not-a-com	<i>error, illegal token '-' ; resume</i>
9	9	i f - <u>I</u> not-a-com	
9	0	i f - <u>I</u> not-a-com	

T Last final
└ File scan
└ Begin scan

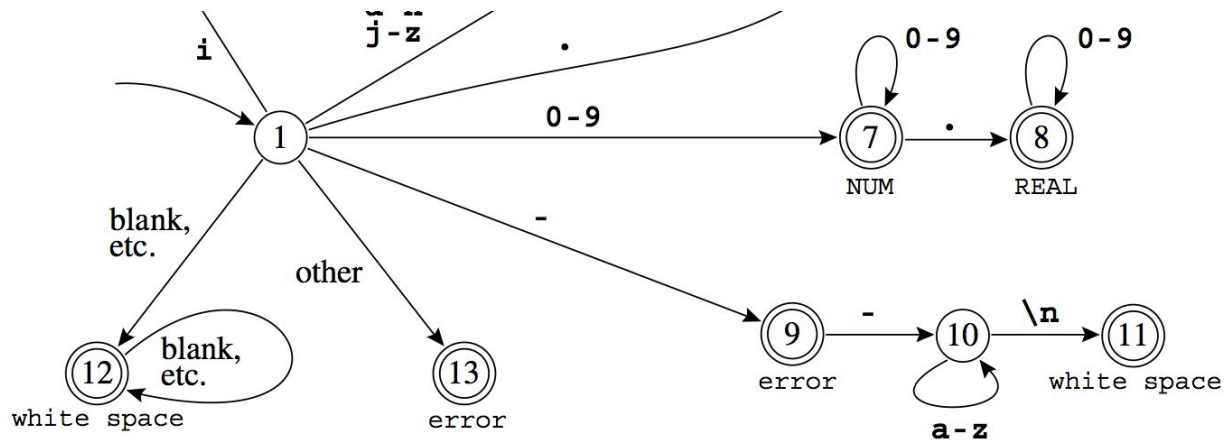
Last Final	Current State	Current Input	Accept Action
0	1	<u>I</u> if --not-a-com	
2	2	<u>i</u> f --not-a-com	
3	3	i <u>f</u> --not-a-com	
3	0	if <u>T</u> --not-a-com	<i>return IF</i>



0	1	if --not-a-com	
12	12	if --not-a-com	
12	0	if --not-a-com	<i>found white space; resume</i>



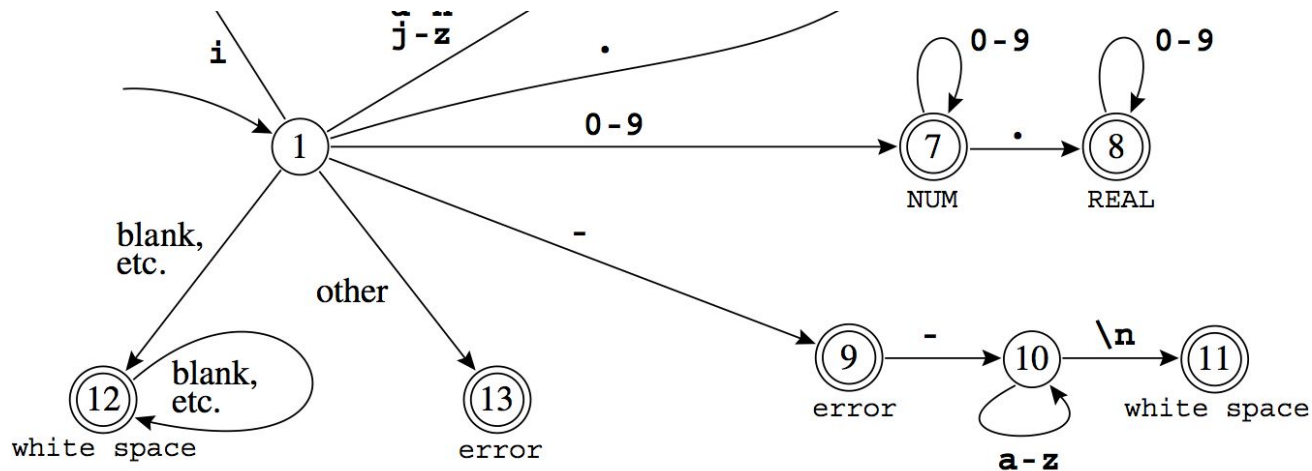
0	1	if <u>I</u> -not-a-com
9	9	if <u>I</u> -not-a-com
9	10	if <u>T</u> not-a-com
9	10	if <u>T</u> not-a-com
9	10	if <u>T</u> not-a-com
9	10	if <u>T</u> not-a-com
9	0	if <u>T</u> not-a-com <i>error, illegal token '-' ; resume</i>



Operações do Lexer – Error (2)

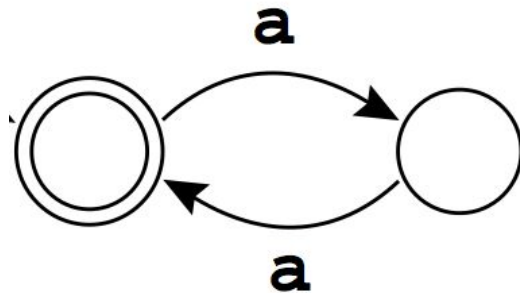
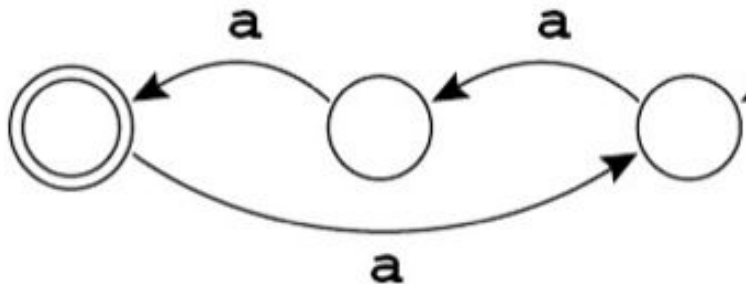
55

0	1	if	-not-a-com	
9	9	if	-not-a-com	
9	0	if	-not-a-com	<i>error, illegal token '-' ; resume</i>

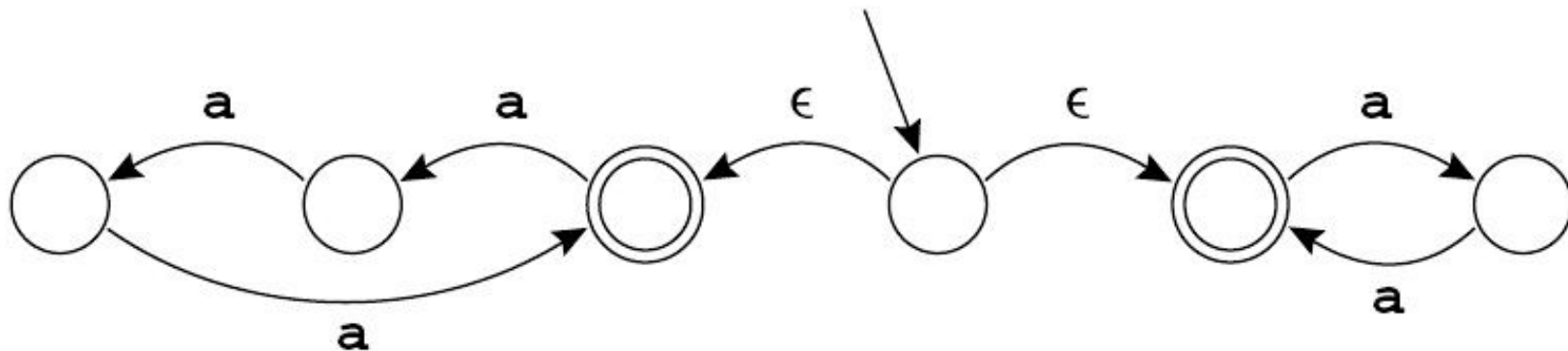


Autômato Finito Não-determinístico





Que linguagem este autômato reconhece?

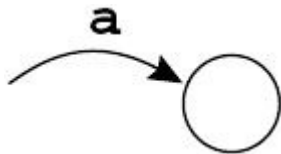


Ele é obrigado a aceitar a string se existe alguma escolha de caminho que leva à aceitação

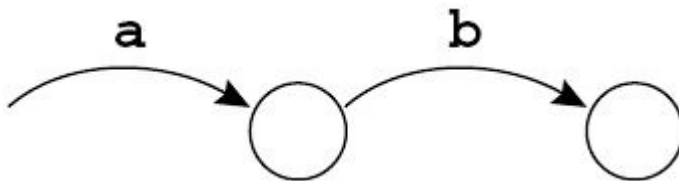
- Pode ter mais de uma aresta saindo de um determinado estado com o mesmo símbolo
- Pode ter arestas anotadas com o símbolo ϵ
 - Essa aresta pode ser percorrida sem consumir nenhum caractere da entrada!
- Não são apropriados para transformar em programas de computador
 - “Adivinhar” qual caminho deve ser seguido não é uma tarefa facilmente executada pelo HW dos computadores
- Solução é transformar em um DFA

- NFAs se tornam úteis porque é fácil converter expressões regulares (ER) para NFA
- Exemplos:

a



a.b

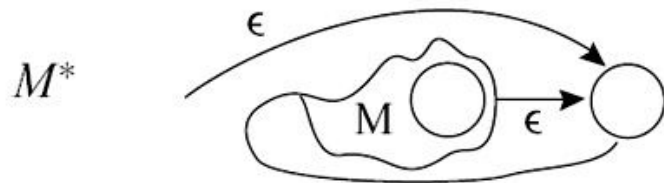
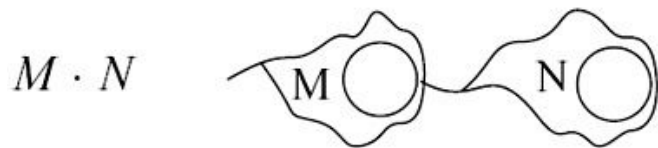
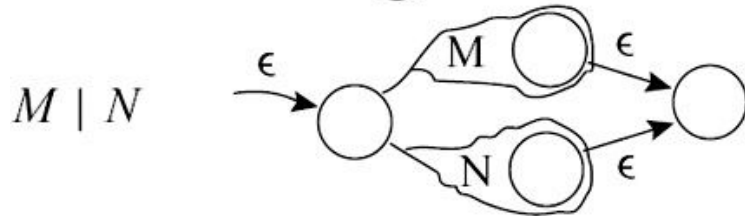
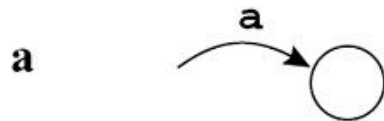




- De maneira geral, toda ER terá um NFA com uma cauda (aresta de entrada) e uma cabeça (estado final)
- Podemos definir essa conversão de maneira indutiva pois:
 - Uma ER é primitiva (único símbolo ou vazio) ou é uma combinação de outras ERs.
 - O mesmo vale para NFAs

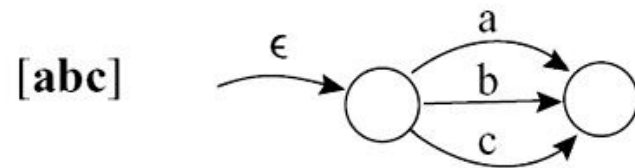
Convertendo ERs para NFAs

62



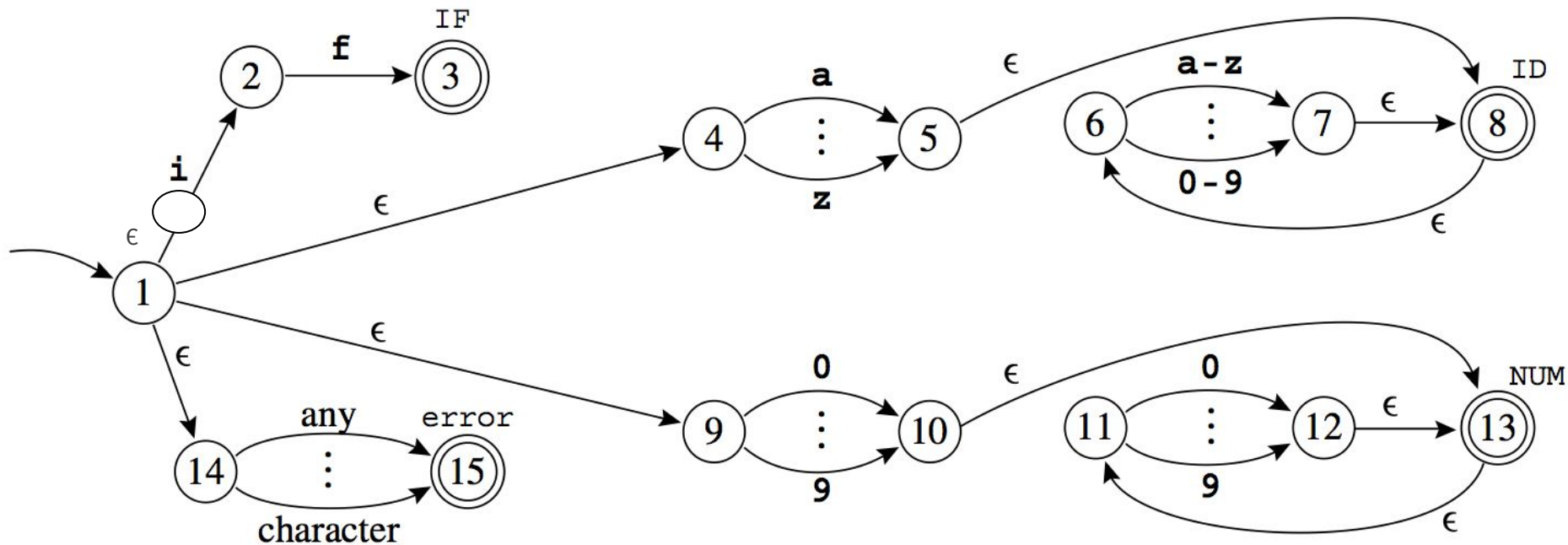
M^+ constructed as $M \cdot M^*$

$M?$ constructed as $M \mid \epsilon$



"abc" constructed as **a · b · c**

ERs para IF, ID, NUM e error

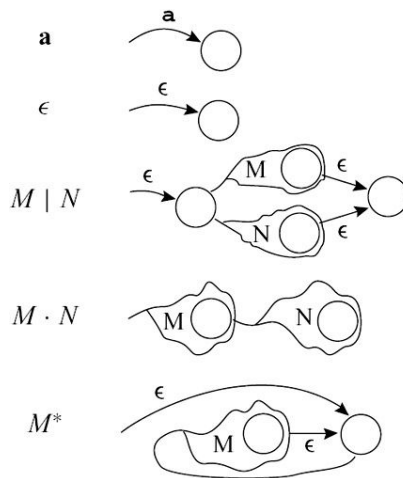


- DFAs são facilmente simuláveis por programas de computador
- NFAs são mais complexos, pois o programa teria que “adivinhar” o melhor caminho em alguns momentos
 - Outra alternativa seria tentar todas as possibilidades

1. Construir um NFA para cada regra, o estado final desse NFA é rotulado com o tipo do token
 - Construção de Thompson
2. Combinar os NFAs em um único NFA
 - Um estado inicial que leva aos estados iniciais do NFA de cada regra via uma transição ϵ
3. Transformar esse NFA em um DFA
 - Os estados finais ficam com o rótulo da regra que aparece primeiro
4. Eventualmente minimizar o DFA

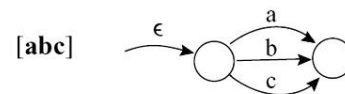
Use a construção de Thompson para construir um NFA para cada RE

- (if | then | else)
- (ab | ac)*
- (0 | 1)* 1100 1*
- (01 | 10 | 00)* 11



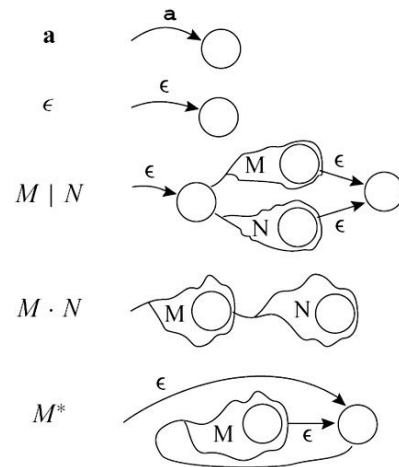
M^+ constructed as $M \cdot M^*$

$M?$ constructed as $M \mid \epsilon$



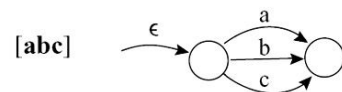
"abc" constructed as $a \cdot b \cdot c$

1. Construir um NFA para cada regra, o estado final desse NFA é rotulado com o tipo do token
 - Construção de Thompson
2. Combinar os NFAs em um único NFA
 - Um estado inicial que leva aos estados iniciais do NFA de cada regra via uma transição ϵ
3. Transformar esse NFA em um DFA
 - Os estados finais ficam com o rótulo da regra que aparece primeiro
4. Eventualmente minimizar o DFA



M^+ constructed as $M \cdot M^*$

$M?$ constructed as $M \mid \epsilon$






"abc" constructed as $\mathbf{a \cdot b \cdot c}$

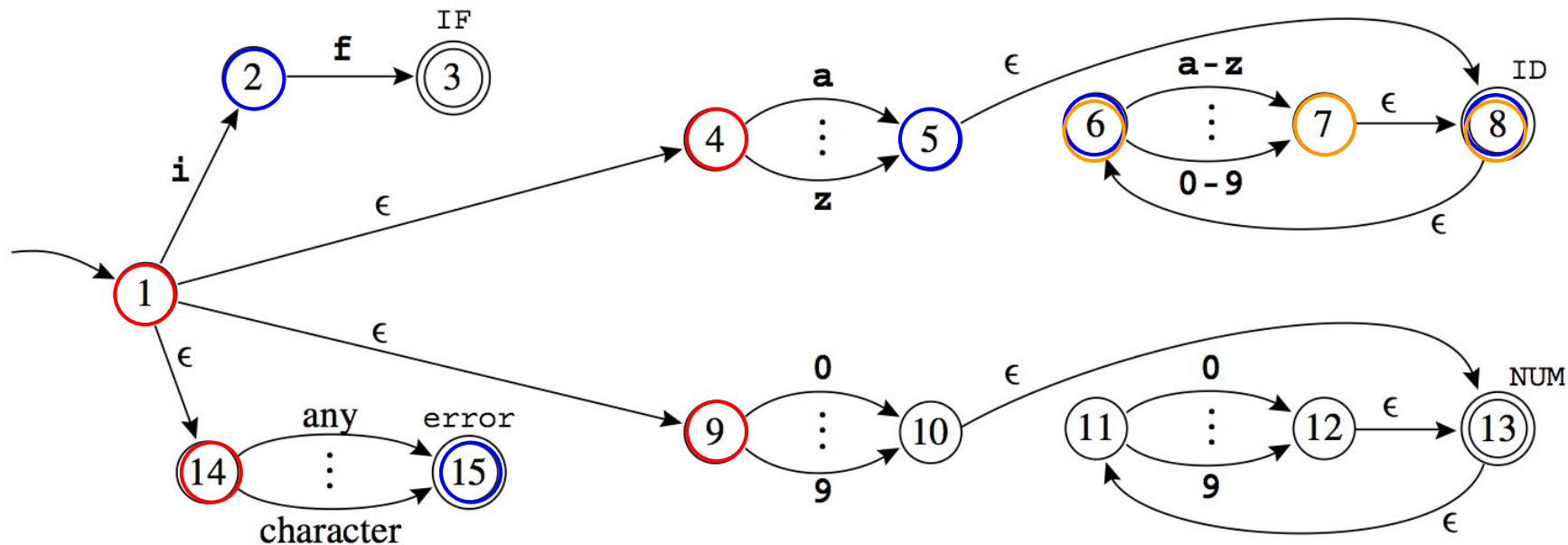
Simulação de NFA



- DFAs são facilmente simuláveis por programas de computador
 - Vimos uma implementação possível com *transition table*
- NFAs são mais complexos em simular
 - O programa teria que “adivinhar” o melhor caminho em alguns momentos
 - Outra alternativa seria tentar todas as possibilidades

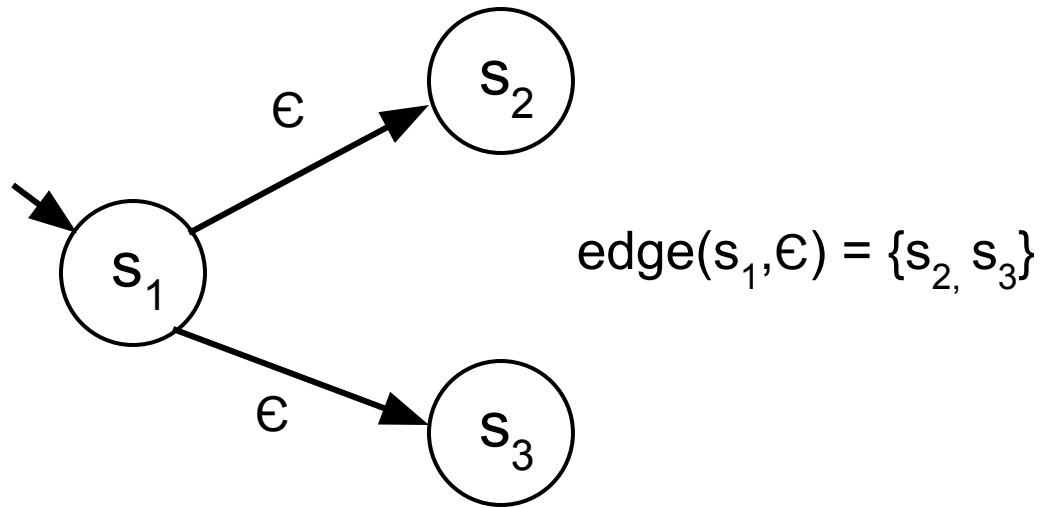
Simulando NFA para “in”

Início (1) -> NFA pode estar em 
Consome i -> NFA pode estar em 
Consome n -> NFA pode estar em 



- $\text{edge}(s,c)$: todos os estados alcançáveis a partir de s , consumindo c
- $\text{Closure}(S)$: todos os estados alcançáveis a partir do conjunto S , sem consumir caractere da entrada
- $\text{Closure}(S)$ é o conjunto T tal que:

$$T = S \cup \left(\bigcup_{s \in T} \text{edge}(s, \epsilon) \right)$$



$$S \cup \left(\bigcup_{s \in T} \mathbf{edge}(s, \epsilon) \right) = \{s_1, s_2, s_3\}$$

Computado por iteração:

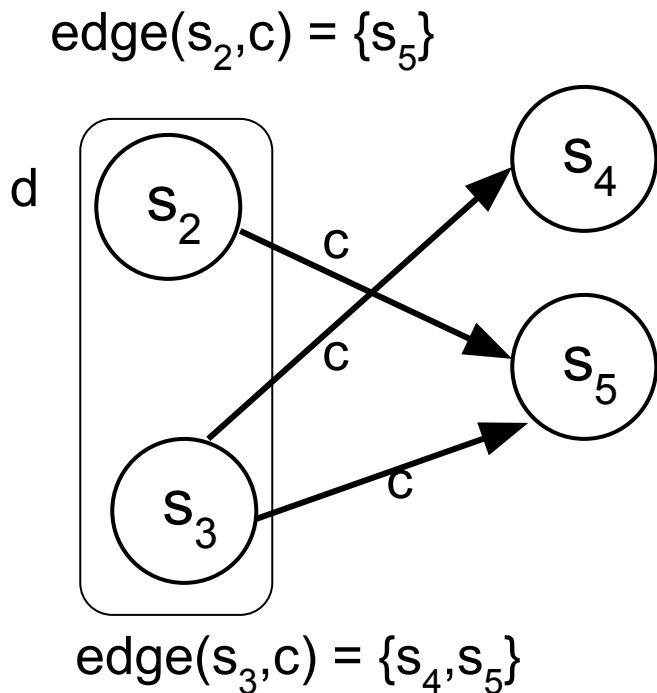
```

$$T \leftarrow S$$
repeat  $T' \leftarrow T$   
           $T \leftarrow T' \cup (\bigcup_{s \in T'} \mathbf{edge}(s, \epsilon))$   
until  $T = T'$ 
```

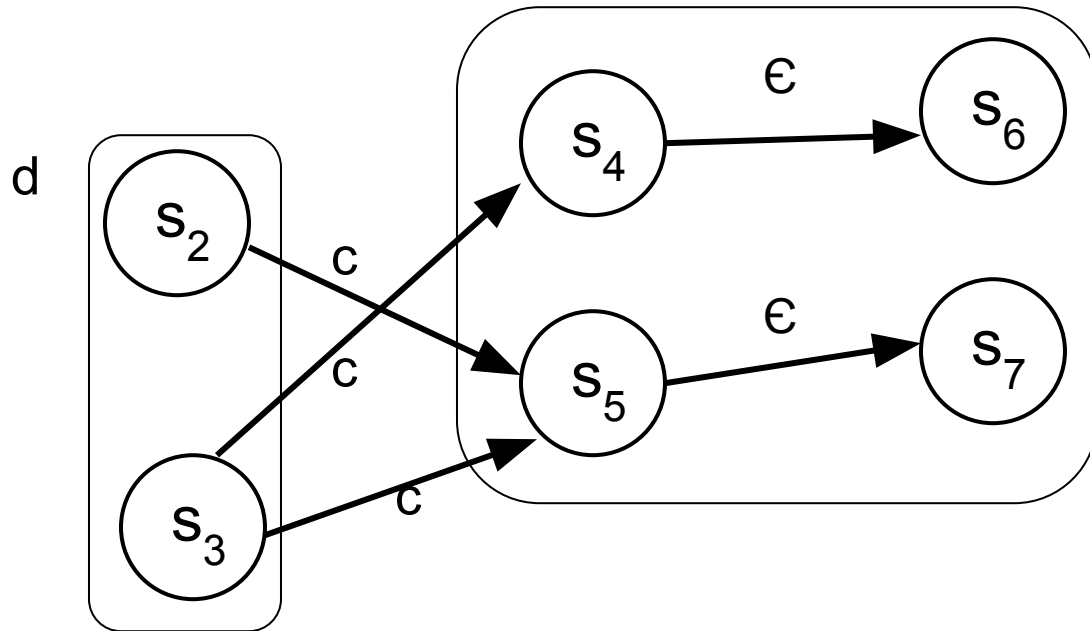
- Da maneira que fizemos a simulação, vamos definir:

$$\mathbf{DFAedge}(d, c) = \mathbf{closure}\left(\bigcup_{s \in d} \mathbf{edge}(s, c)\right)$$

como o conjunto de estados do NFA que podemos atingir a partir do conjunto d , consumindo c



$$\bigcup_{s \in d} \text{edge}(s, c) = \{s_4, s_5\}$$



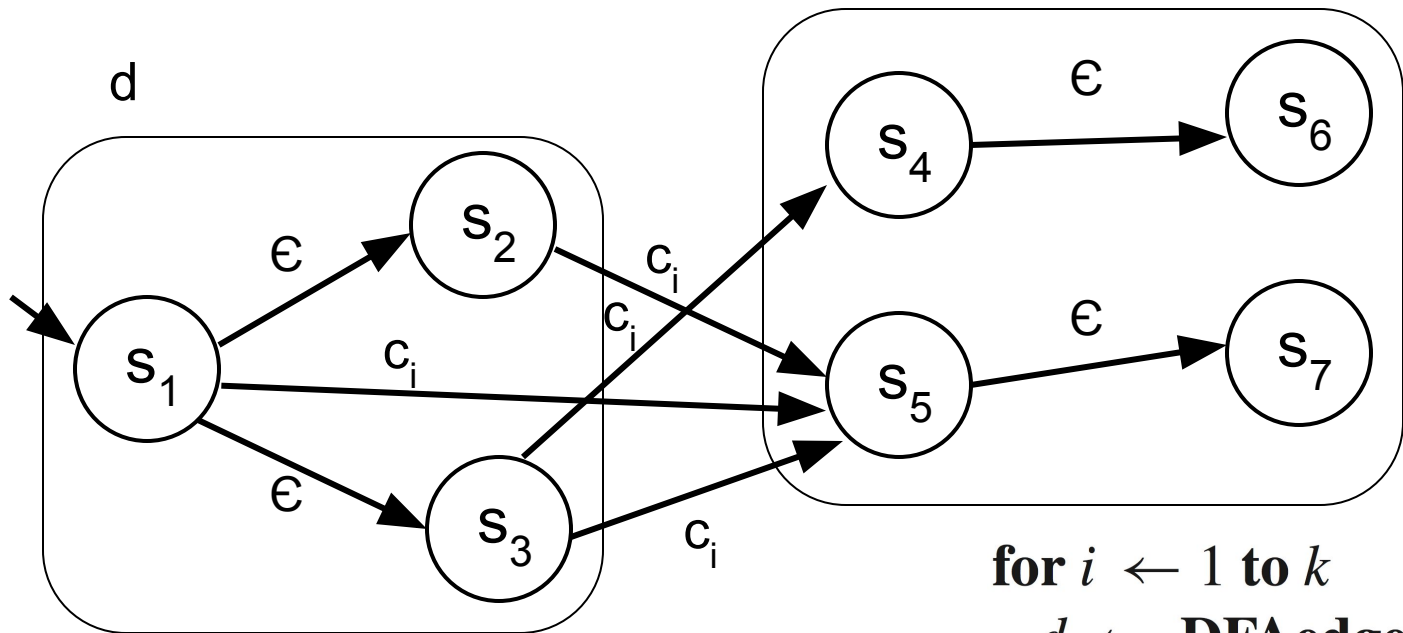
$$\text{closure}\left(\bigcup_{s \in d} \text{edge}(s, c)\right) = \{s_4, s_5, s_6, s_7\}$$

- Estado inicial s_1 e string de entrada c_1, \dots, c_k

$d \leftarrow \mathbf{closure}(\{s_1\})$

for $i \leftarrow 1$ **to** k

$d \leftarrow \mathbf{DFAedge}(d, c_i)$



$d \leftarrow \text{closure}(\{s_1\})$

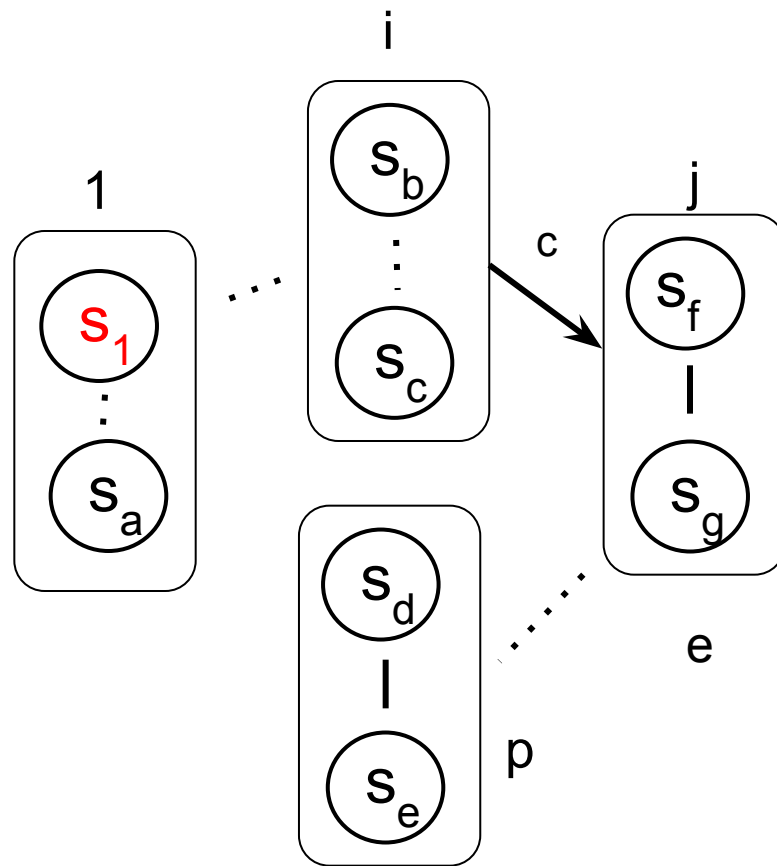
for $i \leftarrow 1$ to k
 $d \leftarrow \text{DFAedge}(d, c_i)$

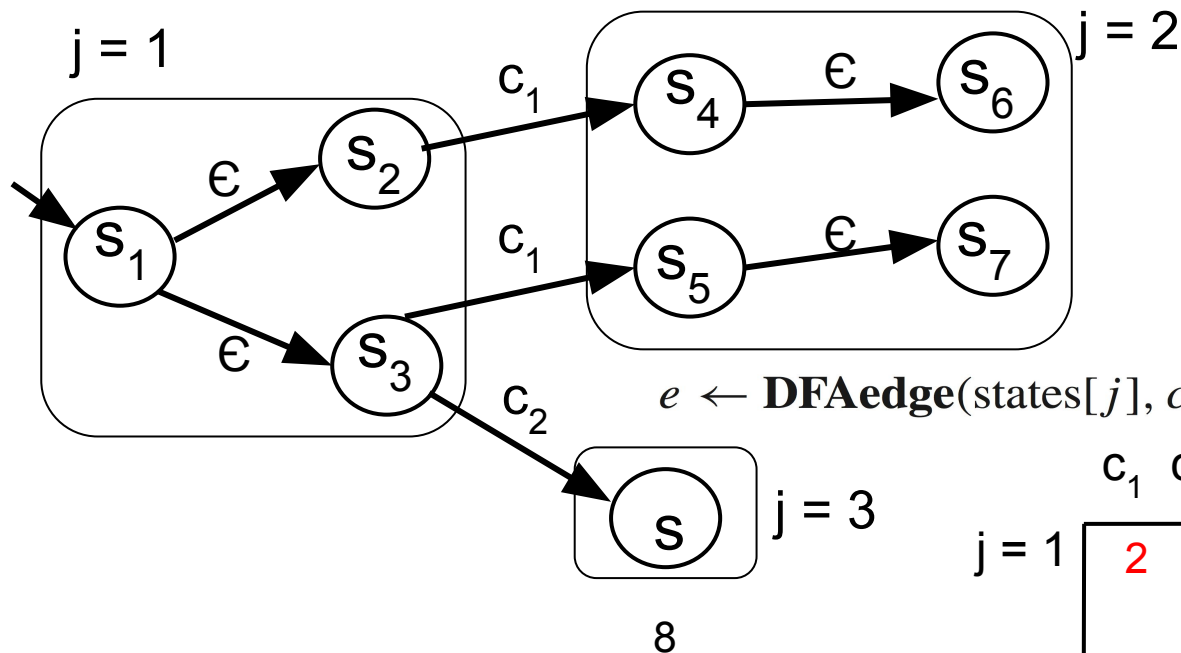
Convertendo NFA em DFA



- Manipular esses conjuntos de estados é muito caro durante a simulação
- Solução:
 - Calcular todos eles antecipadamente
- Isto converte o NFA em um DFA !!
 - Cada conjunto de estados no NFA se torna um estado no DFA


```
states[0]  $\leftarrow$  {};    states[1]  $\leftarrow$  closure( $\{s_1\}$ )  
 $p \leftarrow 1$ ;     $j \leftarrow 0$   
while  $j \leq p$   
  foreach  $c \in \Sigma$   
     $e \leftarrow$  DFAedge(states[ $j$ ],  $c$ )  
    if  $e = \text{states}[i]$  for some  $i \leq p$   
      then trans[ $j, c$ ]  $\leftarrow i$   
    else  $p \leftarrow p + 1$   
        states[ $p$ ]  $\leftarrow e$   
        trans[ $j, c$ ]  $\leftarrow p$   
   $j \leftarrow j + 1$ 
```



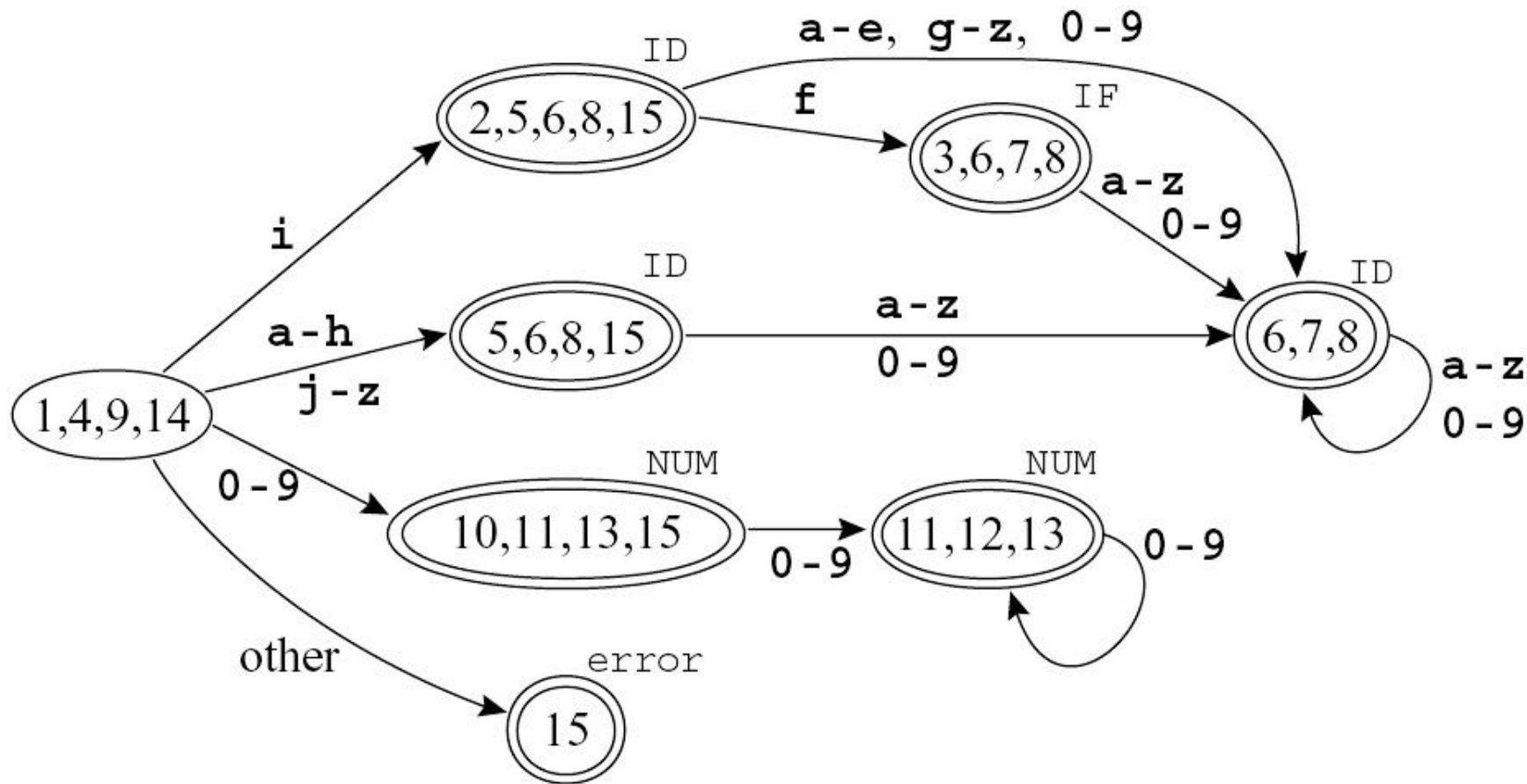


states [] = $\{\{\}, \{s_1, s_2, s_3\}, \{s_4, s_5, s_6, s_7\}, \{s_8\}\}$

	c_1	c_2
$j = 1$	2	3
		

trans

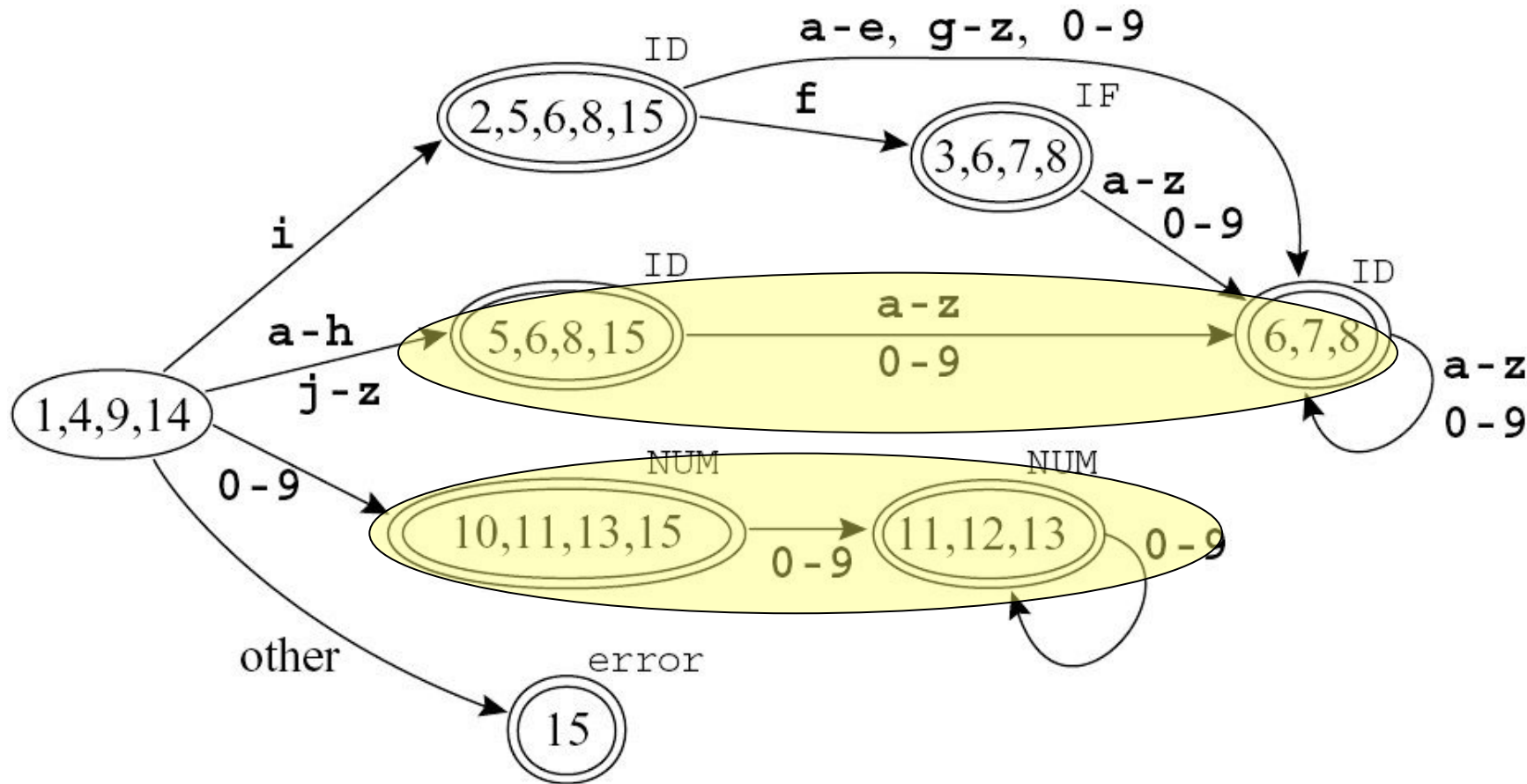
- O estado d é final se qualquer um dos estados de $states[d]$ for final
- Pode haver vários estados finais em $states[d]$
 - d será anotado com o token que ocorrer primeiro na especificação léxica (ERs) -> Regra de prioridade
- Ao final
 - Descarta $states[]$ e usa $trans[]$ para análise léxica



- Esse é o menor autômato possível para essa linguagem?
 - Não, existem estados que são equivalentes!
- s_1 e s_2 são equivalentes quando o autômato aceita cadeia σ começando em $s_1 \Leftrightarrow$ ele também aceita cadeia σ começando em s_2

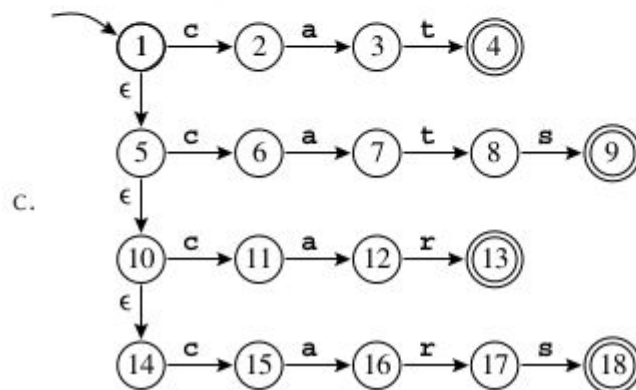
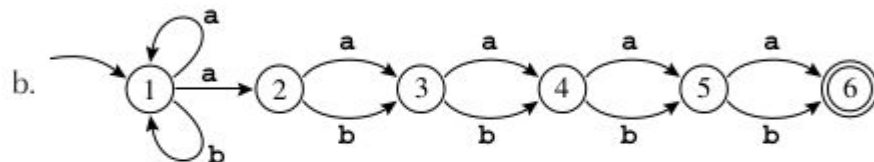
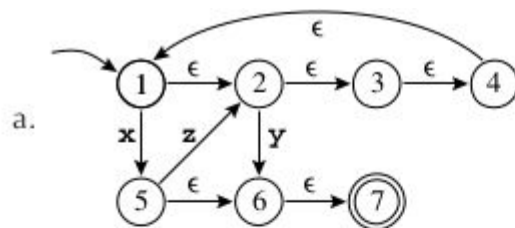
Quais estados são equivalentes no autômato anterior?

86



- Algoritmo de Hopcroft
- Algoritmo de Moore
- Algoritmo de Brzozowski
 1. Inverte o autômato
 2. Converte o NFA em DFA
 3. Inverte o autômato
 4. Converte o NFA em DFA

Converte estes NFAs em DFAs



Resumo

- Estrutura do Frontend
- Trabalho do Lexer
- Especificação
- Gerador de Lexer

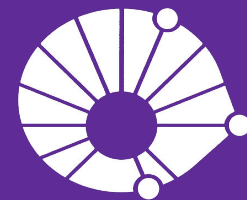
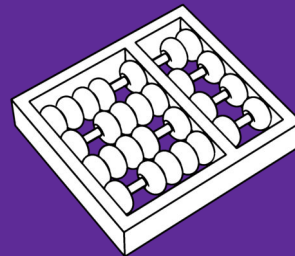
Leitura Recomendada

- Capítulo 2 do livro do Cooper.
- Capítulo 2 do livro do Appel.

Proxima Aula

- Front-end do Compilador
 - Análise Sintática (Parser)
 - Como transformar palavras em frases?
 - Gramática
 - Gerador parser

Obrigado!
Merci!



UNICAMP

Pallette



BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

DRACULA

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

