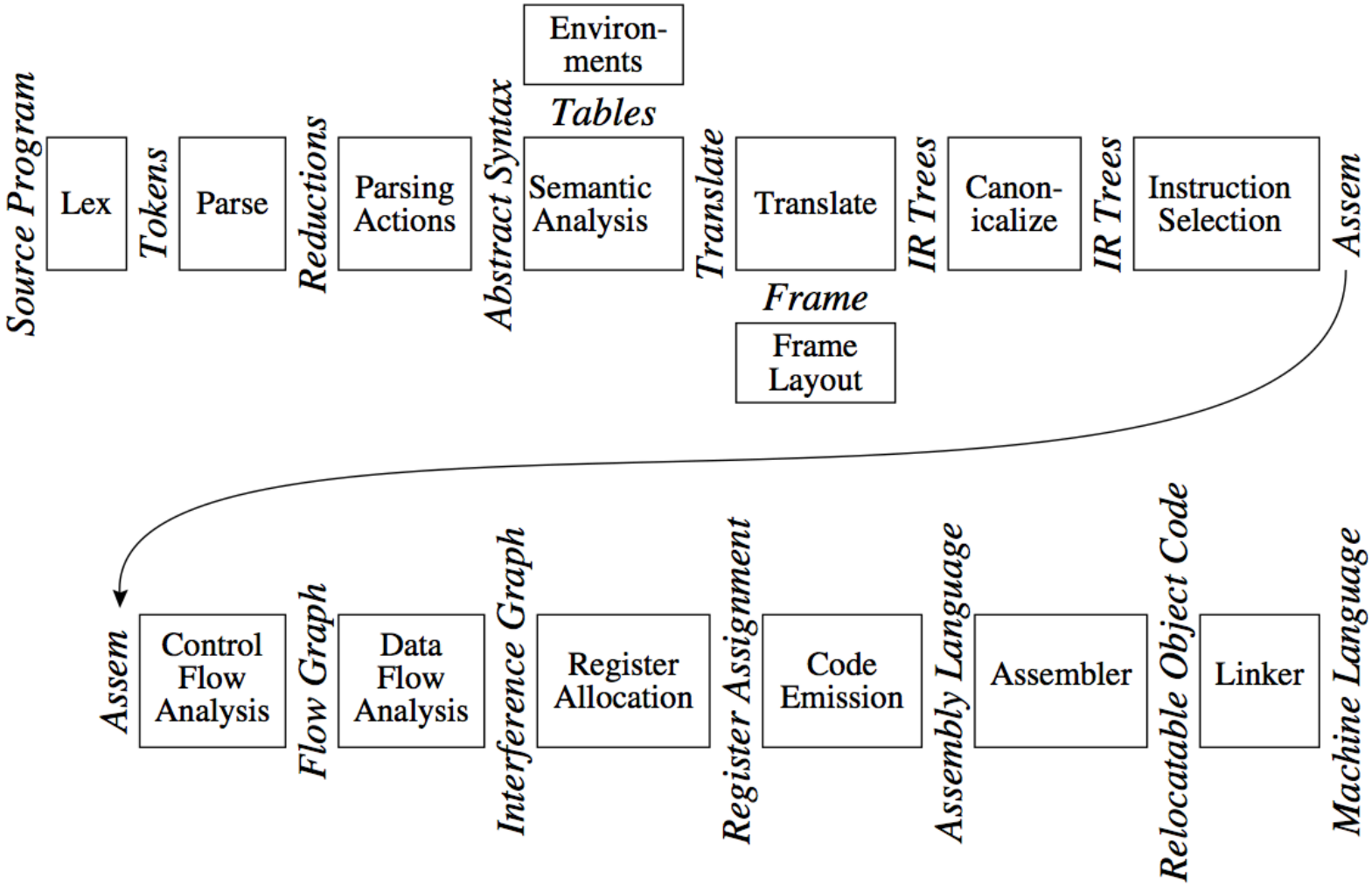
Análise Léxica

Guido Araújo
guido@ic.unicamp.br

Introdução


Source Program
Lex
Tokens
Parse
Reductions
Parsing Actions
Abstract Syntax
Environ-ments
Tables
Semantic Analysis
Translate
Translate
Frame
Frame Layout
IR Trees
Canon-icalize
IR Trees
Instruction Selection
Assem
Assem
Control Flow Analysis
Flow Graph
Data Flow Analysis
Interference Graph
Register Allocation
Register Assignment
Code Emission
Assembly Language
Assembler
Relocatable Object Code
Linker
Machine Language

Introdução

- 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

Introdução

- Análise Léxica:
  - Quebra a entrada em palavras conhecidas como *tokens*
- Análise Sintática:
  - Analisa a estrutura de frases do programa
- Análise Semântica:
  - Calcula o “significado” do programa

Analisador Léxico

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

Tokens

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	(

Não-Tokens

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

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

Exemplo

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

Retorno do analisador léxico:

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s)
COMMA STRING(0.0) COMMA NUM(3) RPAREN RPAREN
RETURN REAL(0.0) SEMI RBRACE EOF

Analisador Léxico

- Alguns tokens têm um valor semântico associados a eles:
  - IDs e NUMs
- Como são descritas as regras léxicográficas?
  - An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.
- Como os tokens são especificados?

Expressões Regulares

- Um linguagem é um conjunto de strings;
- Uma string é uma seqüência de símbolos
- Estes símbolos estão definidos em um alfabeto finito
- Ex: Linguagem C ou Pascal, linguagem dos primos, etc
- Queremos poder dizer se uma string está ou não em um linguagem

Expressões Regulares

- **Símbolo:** Para cada símbolo **a** no alfabeto da linguagem
  - **a** = {a}

- **Alternação:** Dadas duas expressões regulares *M* e *N*, o operador de alternação (|) gera uma nova expressão *M* | *N*
  - **a** | **b** = {*a*, *b*}

- **Concatenação:** Dadas duas expressões regulares *M* e *N*, o operador de concatenação (·) gera uma nova expressão *M* · *N*
  - (**a** | **b**) · **a** = {aa, ba}

Expressões Regulares

- **Epsilon:** A expressão regular $\in$ representa a linguagem cuja única string é a vazia
  - $(a \cdot b) \mid \in$ = {"", "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** | **b**) · **a**)* = {"", "aa", "ba", "aaaa", "baaa", "aaba", "baba", "aaaaaa", …}.

Exemplos

- **(0 | 1)* · 0**
  - Números binários múltiplos de 2.
- **b*(abb*)*(a|∈)**
  - 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.

Notação

- **a** An ordinary character stands for itself.
- ∈ The empty string.
- Another way to write the empty string.
- *M* | *N* Alternation, choosing from *M* or *N*.
- *M* · *N* Concatenation, an *M* followed by an *N*.
- *MN* Another way to write concatenation.
- *M** Repetition (zero or more times).
- *M*+ Repetition, one or more times.
- *M*? Optional, zero or one occurrence of *M*.
- **[a − zA − Z]** Character set alternation.
- . A period stands for any single character except newline.
- "a.+*" Quotation, a string in quotes stands for itself literally.

Exemplos

- Como seriam as expressões regulares para os seguintes tokens?
- IF
  - `if`
- ID
  - `[a-z][a-z0-9]*`
- NUM
  - `[0-9]+`

Exemplos

- 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*
- .
  - *error*

Analisador Léxico

- Ambiguidades:
  - `if8` é um ID ou dois tokens IF e NUM(8) ?
  - `if 89` começa com um ID ou uma palavra-reservada?
- Duas regras:
  - Maior casamento: o próximo token sempre é a substring mais longa possível a ser casada.
  - Prioridade: Para uma dada substring mais longa, a primeira regra a ser casada produzirá o token

Analisador Léxico

- 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?)

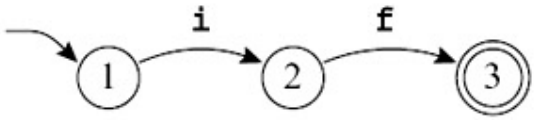
Autômatos Finitos

- Expressões Regulares são convenientes para especificar os tokens
- Precisamos de um formalismo que possa ser convertido em um programa de computador
- Este formalismo são os autômatos finitos

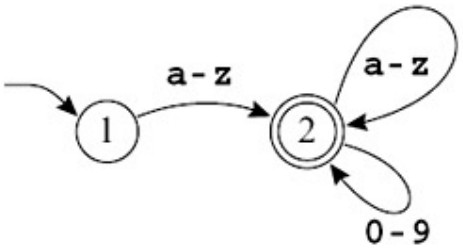
Autômatos Finitos

- Um autômato finito possui:
  - Um conjunto finito de estados
  - Arestas levando de um estado a outro, anotada com um símbolo
  - Um estado inicial
  - Um ou mais estados finais
  - Normalmente os estados são numerados ou nomeados para facilitar a manipulação e discussão

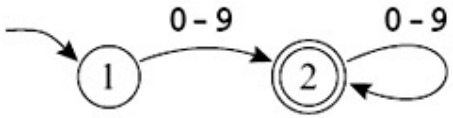
Autômatos Finitos



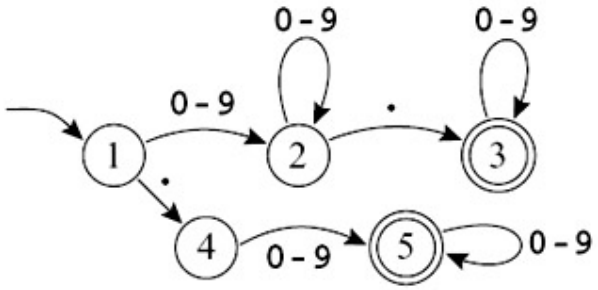
IF



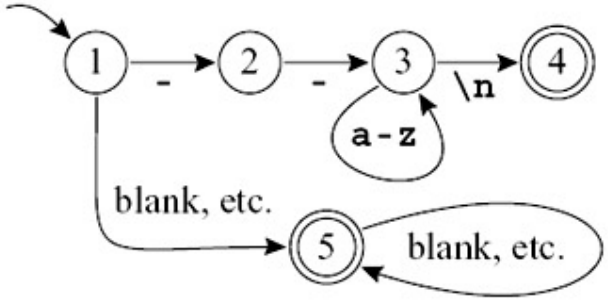
ID



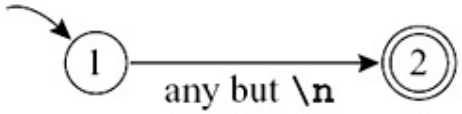
NUM



REAL



white space



error

Deterministic Finite Automata

- 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

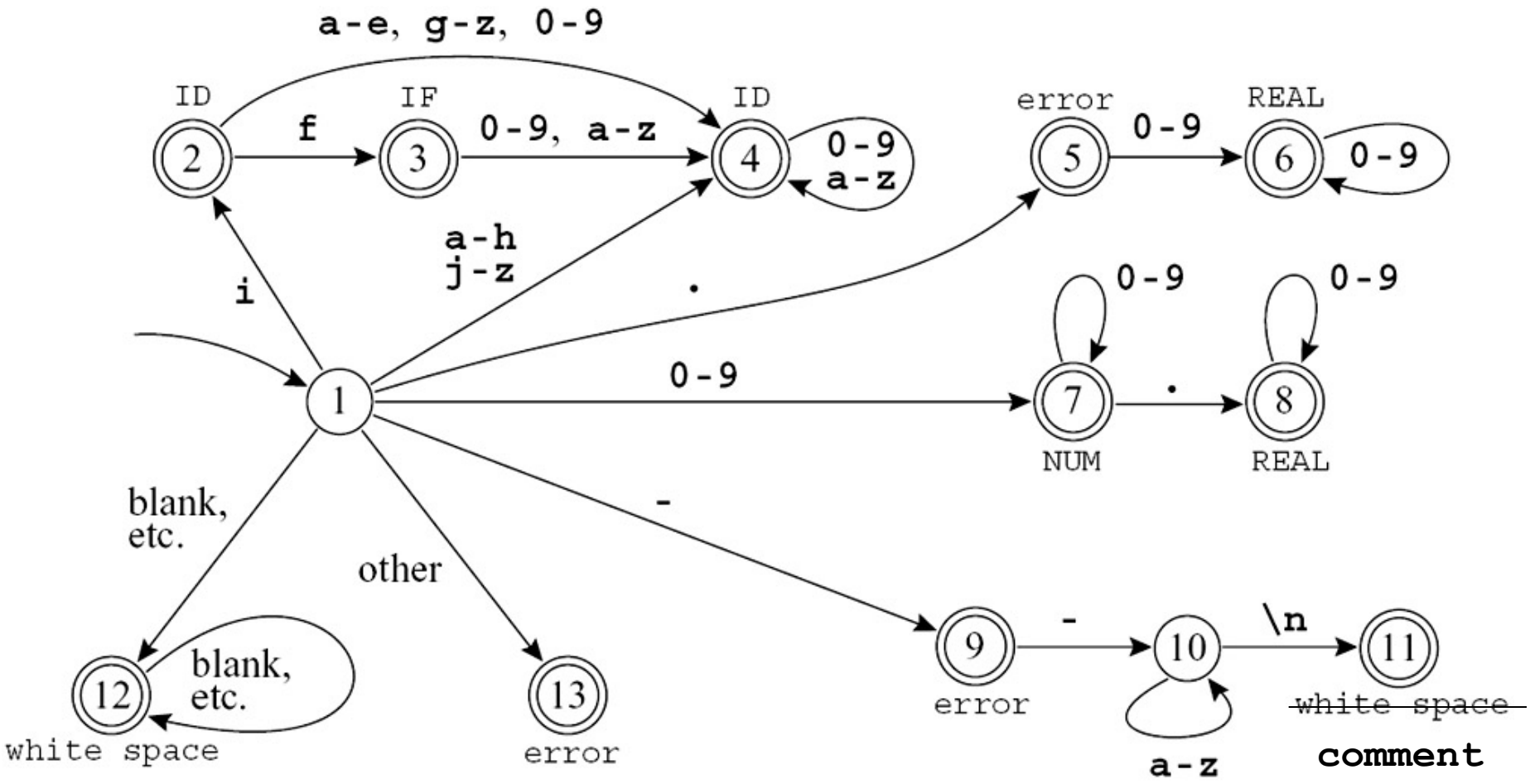
Deterministic Finite Automata

- 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

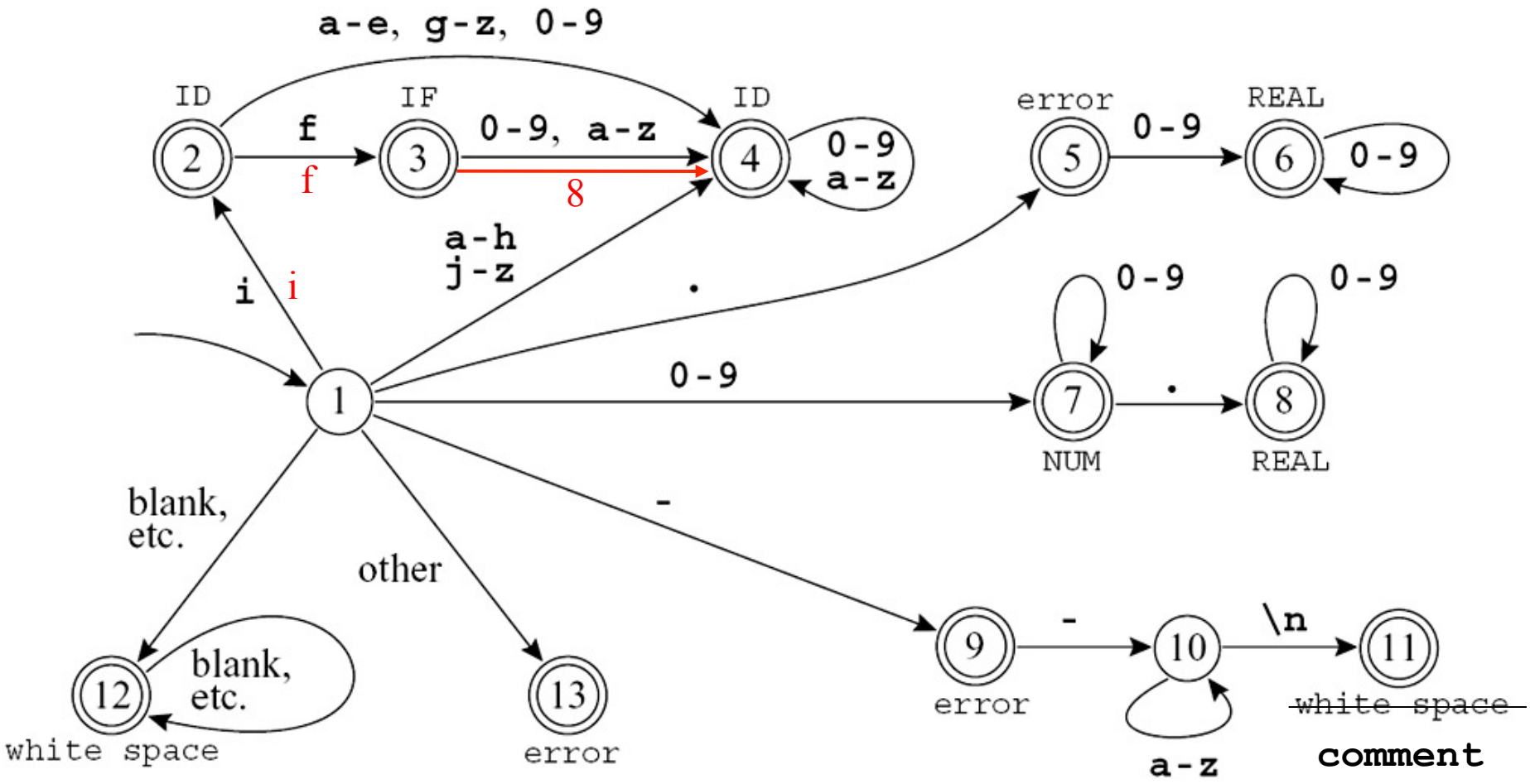
Autômatos Finitos

- Consigo combinar os autômatos definidos para cada token de maneira a ter um único autômato que possa ser usado como analisador léxico?
  - Sim.
  - Veremos um exemplo *ad-hoc* e mais adiante mecanismos formais para esta tarefa

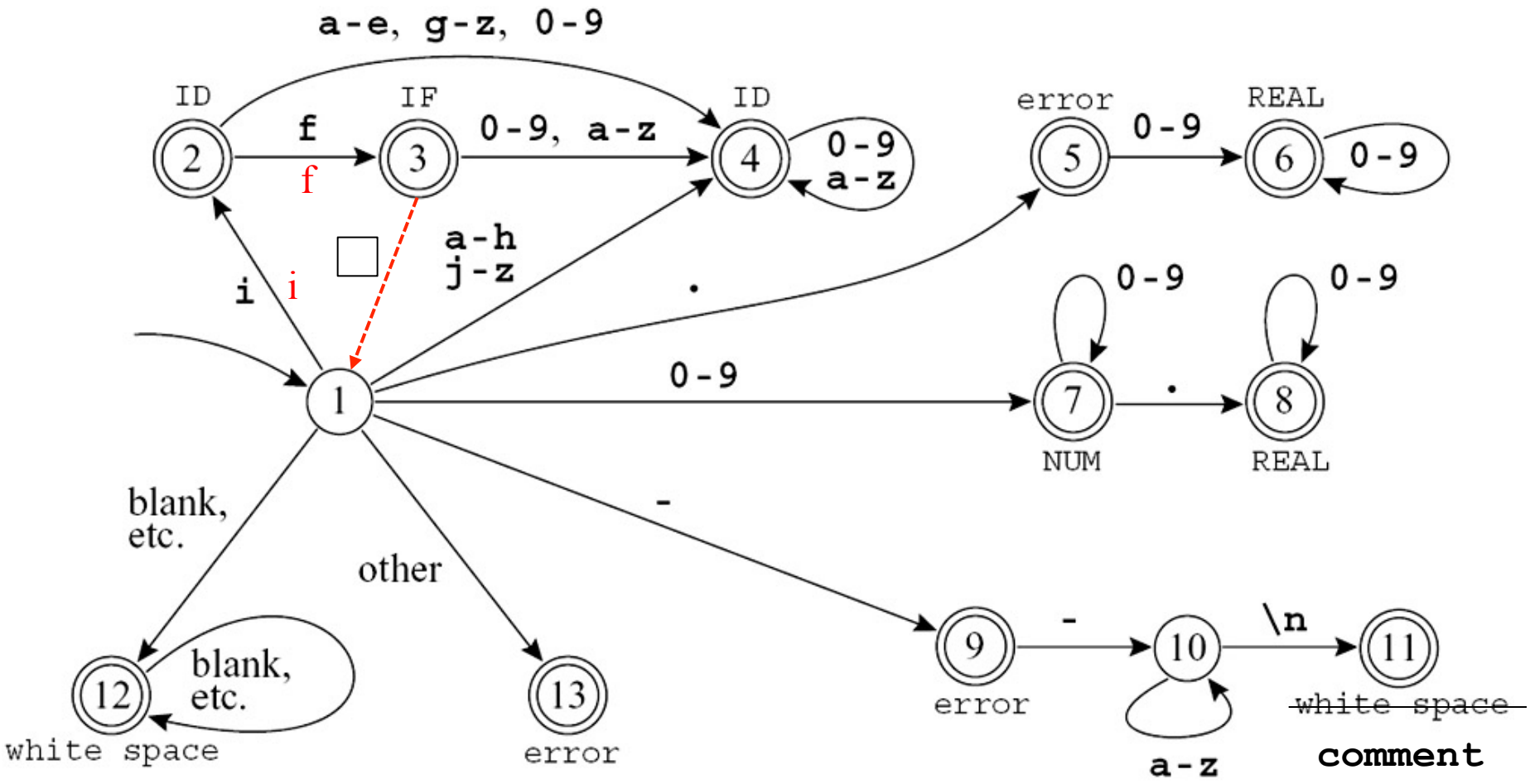
Autômato Combinado



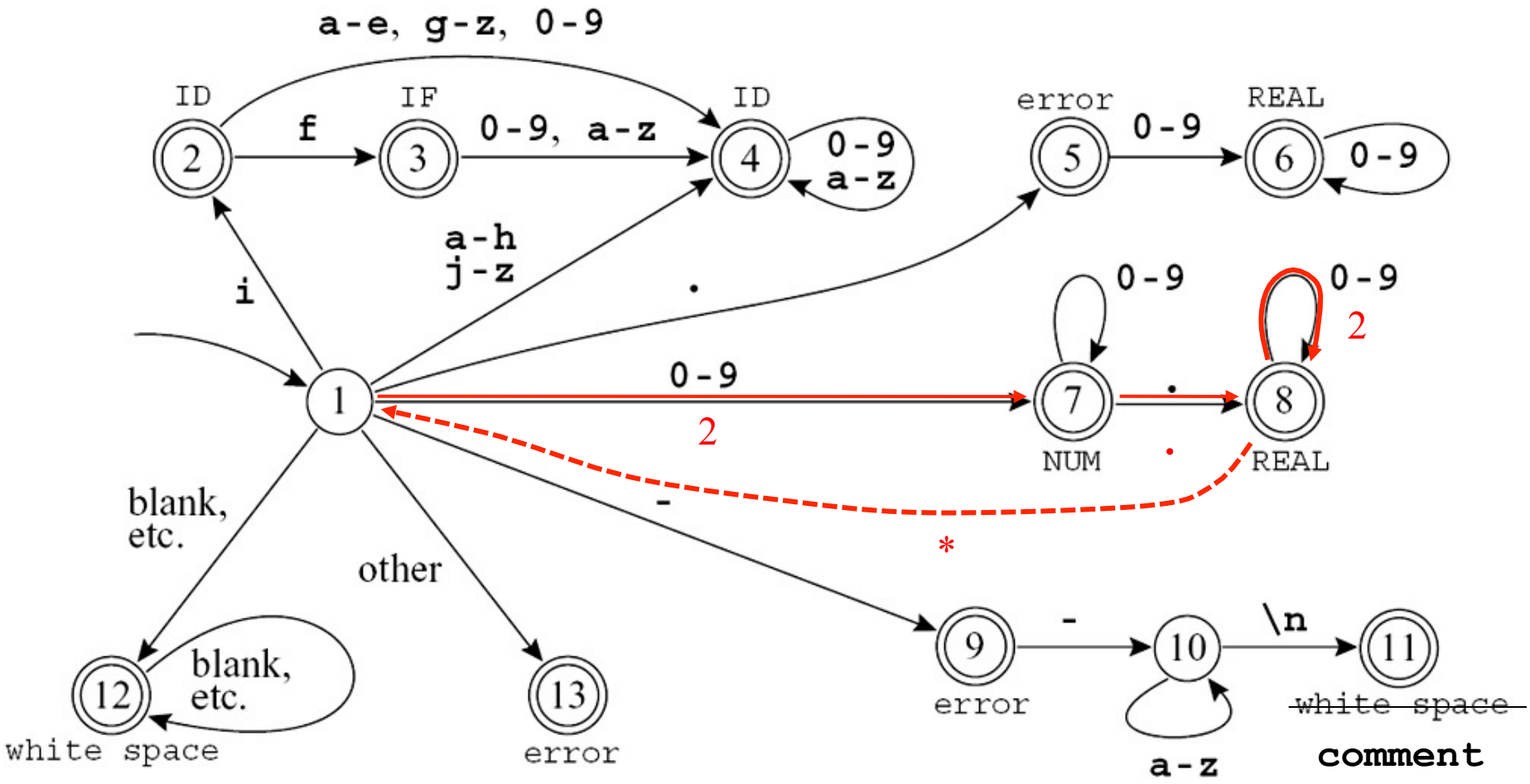
Exemplo – if8


a-e, g-z, 0-9
ID
IF
ID
f
0-9, a-z
0-9
a-z
2
3
4
f
8
error
REAL
0-9
5
6
0-9
a-h
j-z
i
i
.
0-9
0-9
0-9
1
7
8
NUM
REAL
blank,
etc.
other
-
12
blank,
etc.
13
9
-
10
\n
11
white space
error
error
a-z
white space
comment

Exemplo - if □


a-e, g-z, 0-9
ID
IF
ID
f
f
0-9, a-z
2
3
4
0-9
a-z
error
REAL
5
0-9
6
0-9
a-h
j-z
i
i
.
0-9
0-9
0-9
1
7
8
NUM
REAL
blank,
etc.
other
-
12
blank,
etc.
13
9
-
10
\n
11
white space
error
error
a-z
white space
comment

Exemplo – 2.2*


a-e, g-z, 0-9
ID
IF
ID
f
0-9, a-z
0-9
a-z
2
3
4
error
REAL
5
0-9
6
0-9
a-h
j-z
i
.
0-9
0-9
2
0-9
1
7
8
2
NUM
REAL
blank,
etc.
-
*
other
9
-
10
\n
11
12
blank,
etc.
13
error
white space
a-z
white space
comment

Autômato Combinado

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

Autômato Combinado

```
int edges[ ][ ] = { /* ...012...-...e f g h i j... */
/* state 0 */ {0,0,...0,0,0...0...0,0,0,0,0,0...},
/* 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...},
   et cetera }
```

entrada

Ausência de aresta

Reconhecimento da Maior Substring

- 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

Reconhecimento da Maior Substring

Last Final	Current State	Current Input	Accept Action
0	1	⊤|if --not-a-com	
2	2	|i⊤|f --not-a-com	
3	3	|if⊤| --not-a-com	
3	0	|if⊤⊥ --not-a-com	*return* IF
0	1	if⊤| --not-a-com	
12	12	if|⊤|--not-a-com	
12	0	if|⊤⊥-not-a-com	*found white space; resume*
0	1	if ⊤|--not-a-com	
9	9	if |-⊤|-not-a-com	
9	10	if |-⊤-⊥not-a-com	
9	10	if |-⊤-n⊥ot-a-com	
9	10	if |-⊤-no⊥t-a-com	
9	10	if |-⊤-not⊥-a-com	
9	0	if |-⊤-not-⊥a-com	*error, illegal token '-'; resume*
0	1	if -⊤|-not-a-com	
9	9	if -|-⊤|not-a-com	
9	0	if -|-⊤⊥not-a-com	*error, illegal token '-'; resume*

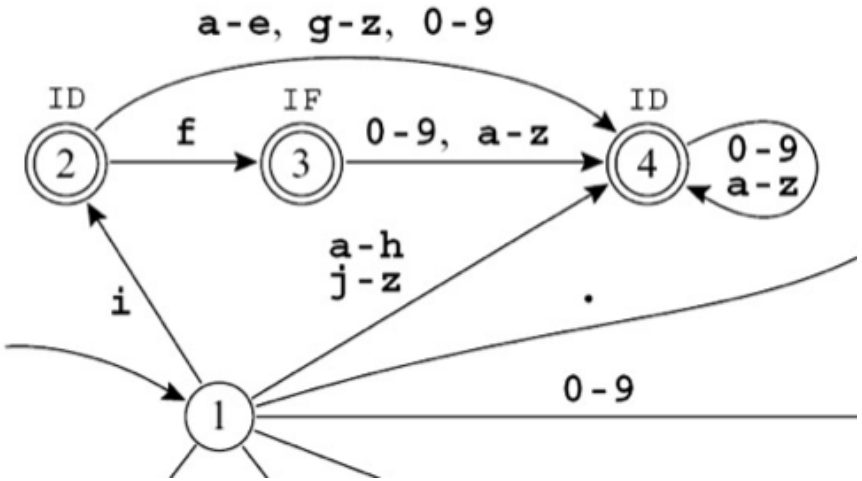
⊤ Last final

⊥ File scan

\| Begin scan

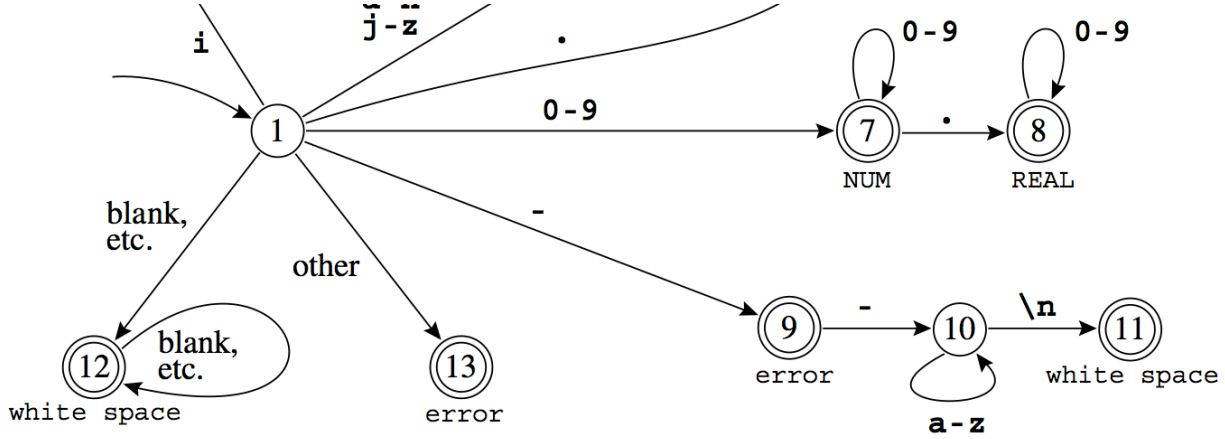
Reconhecimento da Maior Substring

Last Final	Current State	Current Input	Accept Action
0	1	⌶if --not-a-com	
2	2	|i⌶f --not-a-com	
3	3	|if⌶ --not-a-com	
3	0	|if⊤⊥--not-a-com	*return* IF



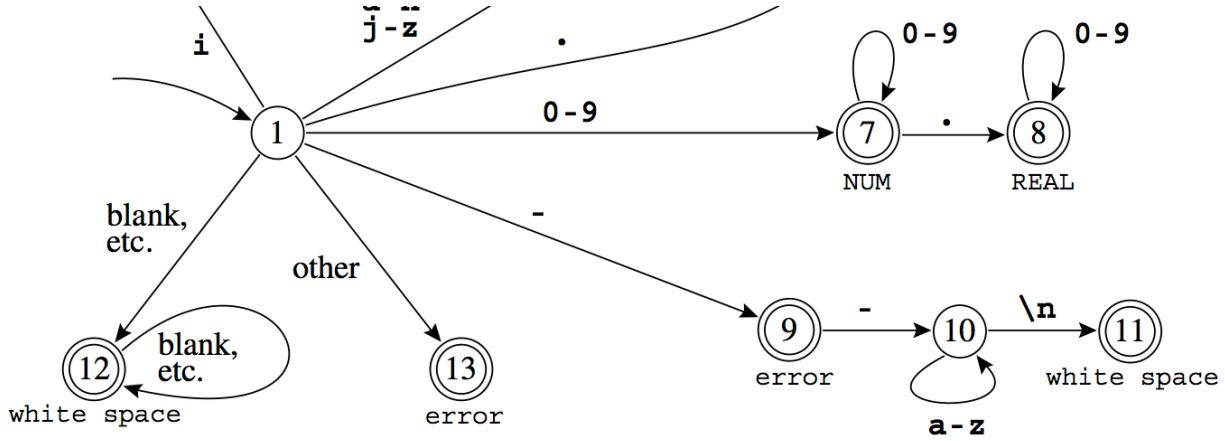
Reconhecimento da Maior Substring

0	1	if --not-a-com	
12	12	if -not-a-com	
12	0	if -not-a-com	*found white space; resume*



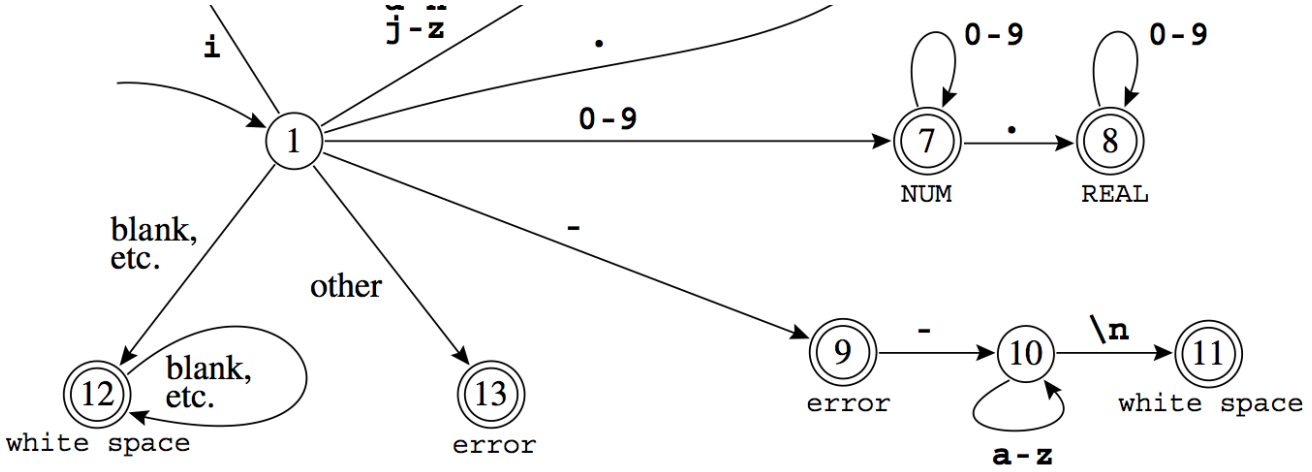
Reconhecimento da Maior Substring

0	1	if -not-a-com	
9	9	if -not-a-com	
9	10	if -not-a-com	
9	10	if -not-a-com	
9	10	if -not-a-com	
9	10	if -not-a-com	
9	0	if -not-a-com	*error, illegal token '-'; resume*

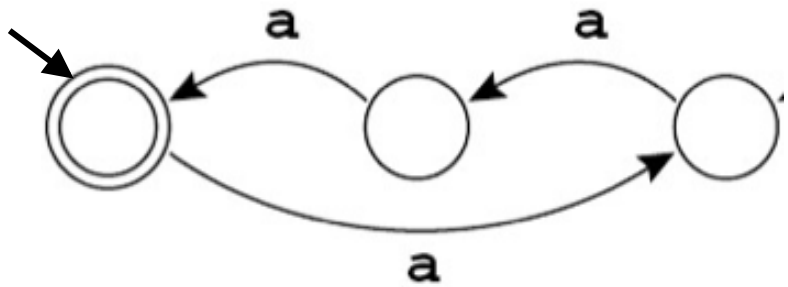


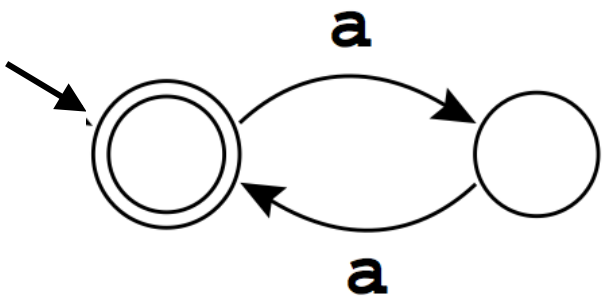
Reconhecimento da Maior Substring

0	1	if -|-not-a-com	
9	9	if -|-|not-a-com	
9	0	if -|-||ot-a-com	*error, illegal token '-'; resume*



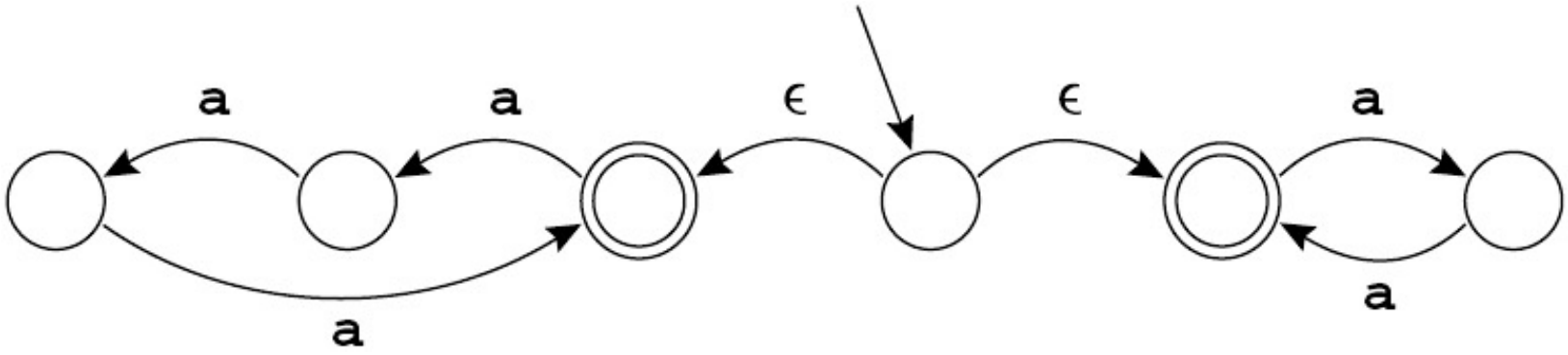
Como combinar autômatos?





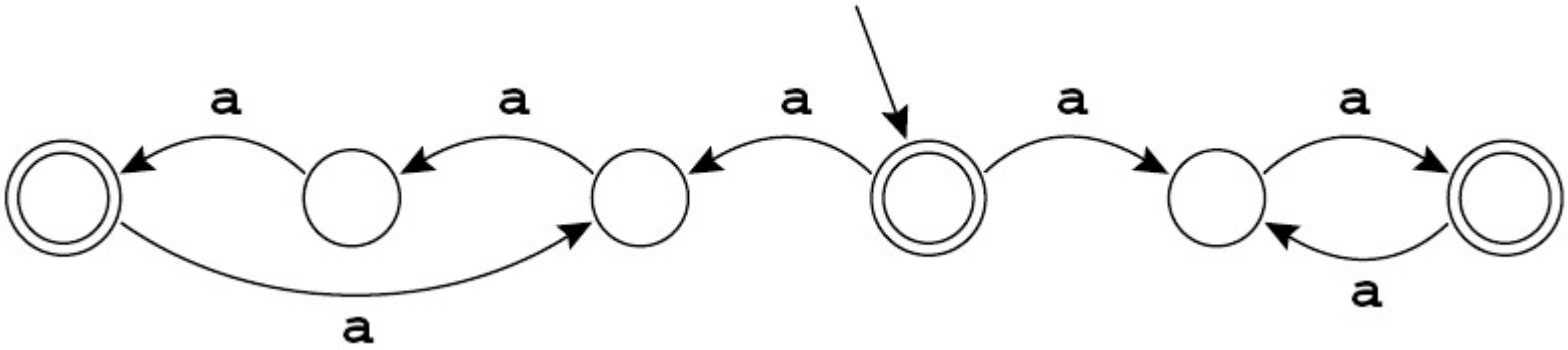
Que linguagem estes autômatos reconhecem?

Como combinar autômatos?



Que linguagem este autômato reconhece?

Nondeterministic Finite Automata (NFA)



Que linguagem este autômato reconhece?

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

Nondeterministic Finite Automata (NFA)

- Como combinar autômatos?
- Pode ter mais de uma aresta saindo de um determinado estado com o mesmo símbolo
- Pode ter arestas anotadas com o símbolo Є
  - Essa aresta pode ser percorrida sem consumir nenhum caractere da entrada!

Nondeterministic Finite Automata (NFA)

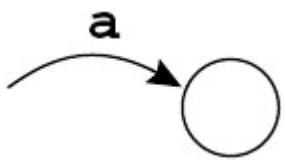
- Não são apropriados para transformar em programas de computador
  - "Adivinhar" qual caminho deve ser seguido não é uma tarefa facilmente executada pelo HW dos computadores

- Solução é transformar em um DFA

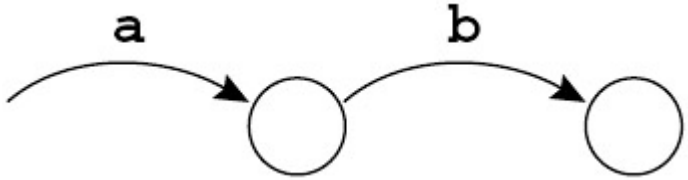
Convertendo ER's para NFA's

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

a



ab



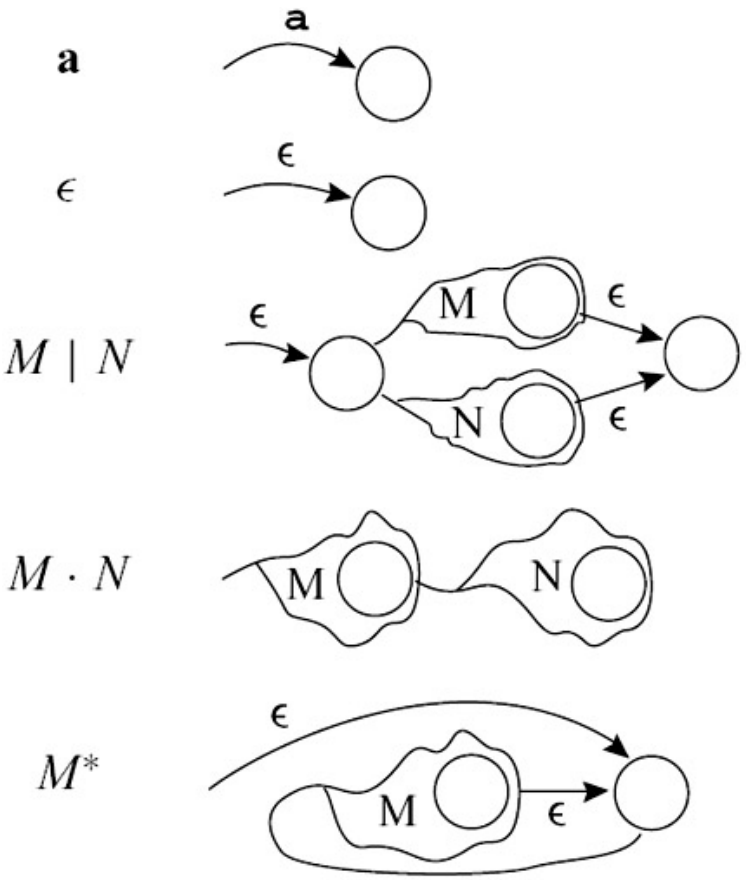
Convertendo ER's para NFA's

- De maneira geral, toda ER terá um NFA com uma cauda (aresta de entrada) e uma cabeça (estado final).

Convertendo ER's para NFA's

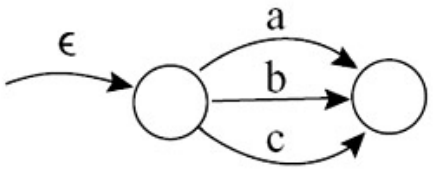
- Podemos definir essa conversão de maneira indutiva pois:
  - Uma ER é primitiva (único símbolo ou vazio) ou é uma combinação de outras ERs.
  - O mesmo vale para NFAs

Convertendo ER's para NFA's



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

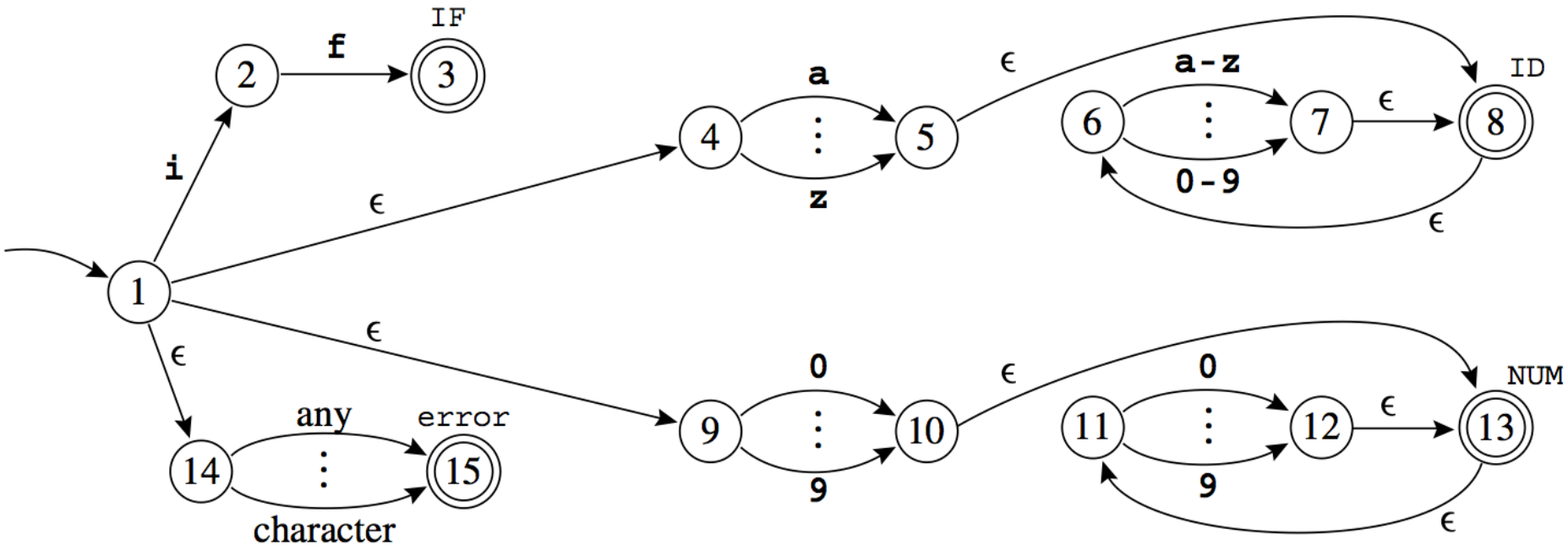
$M?$ constructed as $M \mid \epsilon$



"**abc**" constructed as **a** · **b** · **c**

Exemplo

ERs para IF, ID, NUM e **error**



NFA vs. DFA

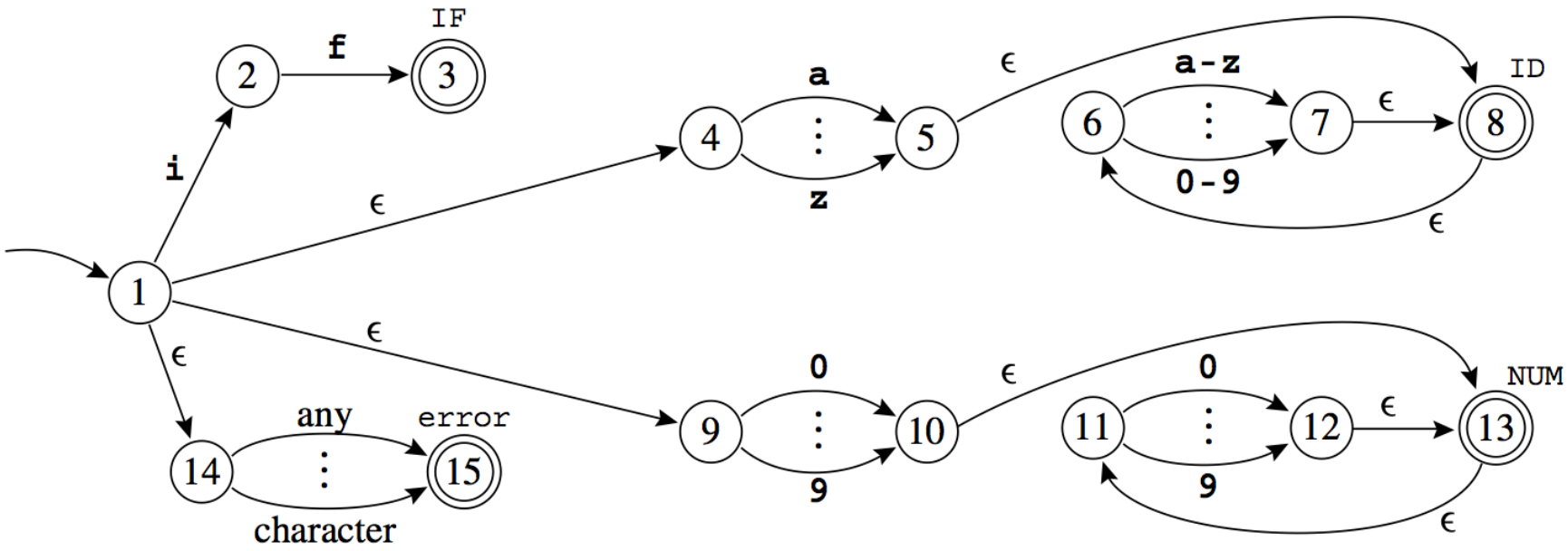
- DFAs são facilmente simuláveis por programas de computador
- NFAs são mais complexos, pois o programa teria que “adivinhar” o melhor caminho em alguns momentos
- Outra alternativa seria tentar todas as possibilidades

Simulando NFA para "`in`"

Início (1) -> NFA pode estar em {1,4,9,14}

Consome `i` -> NFA pode estar em {2,5,6,8,15}

Consome `n` -> NFA pode estar em {6,7,8}

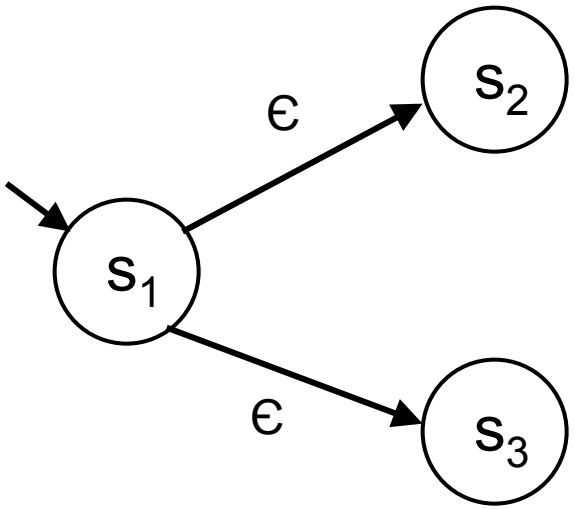


Є-Closure

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

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

Closure(S)



edge(s_1,ϵ) = {s_2}

edge(s_1,ϵ) = {s_3}

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

Algoritmo

Computado por iteração:

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

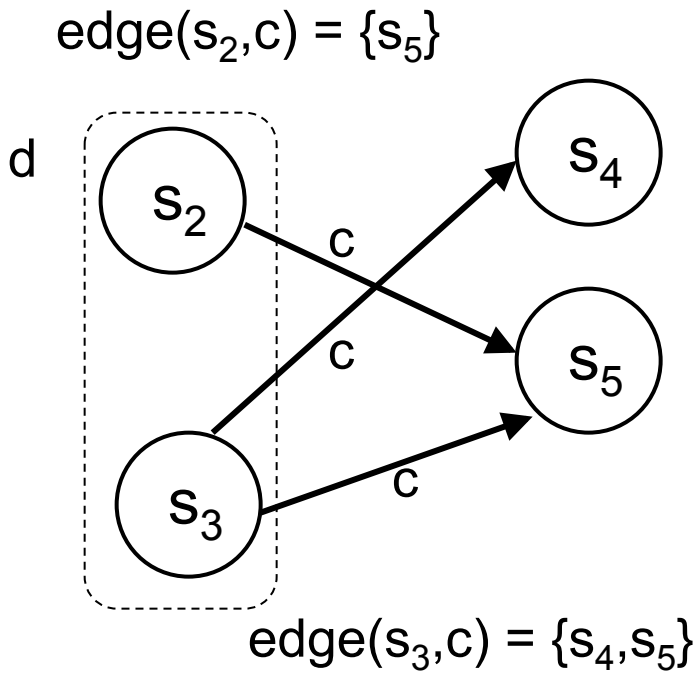
Algoritmo da Simulação

- Da maneira que fizemos a simulação, vamos definir:

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

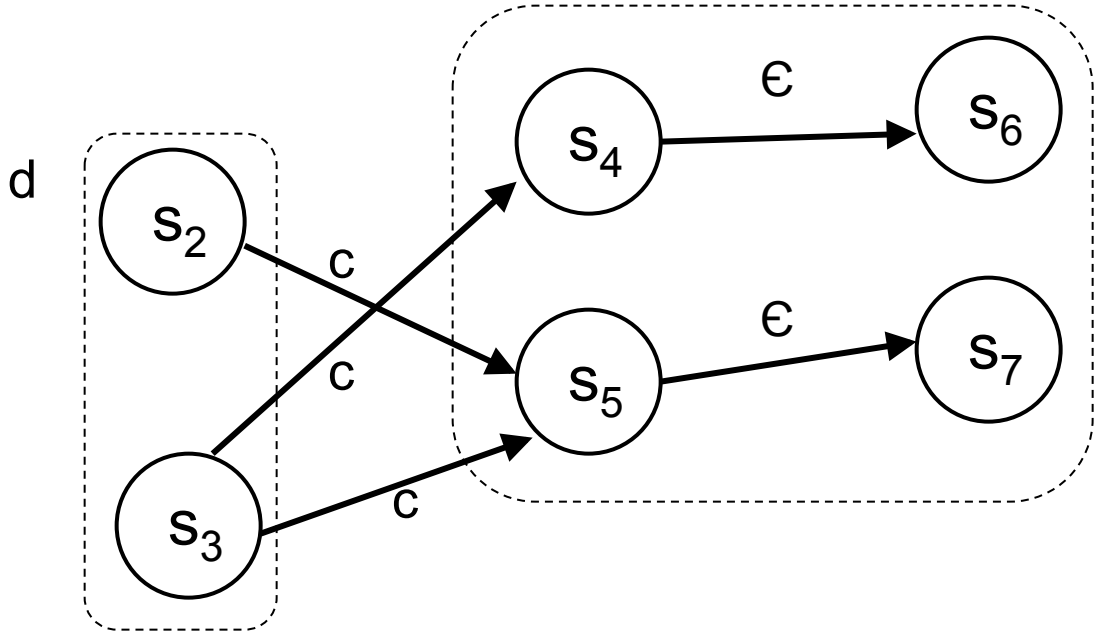
como o conjunto de estados do NFA que podemos atingir a partir do conjunto *d*, consumindo *c*

DFAedge(d,c)



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

DFAedge(d,c)



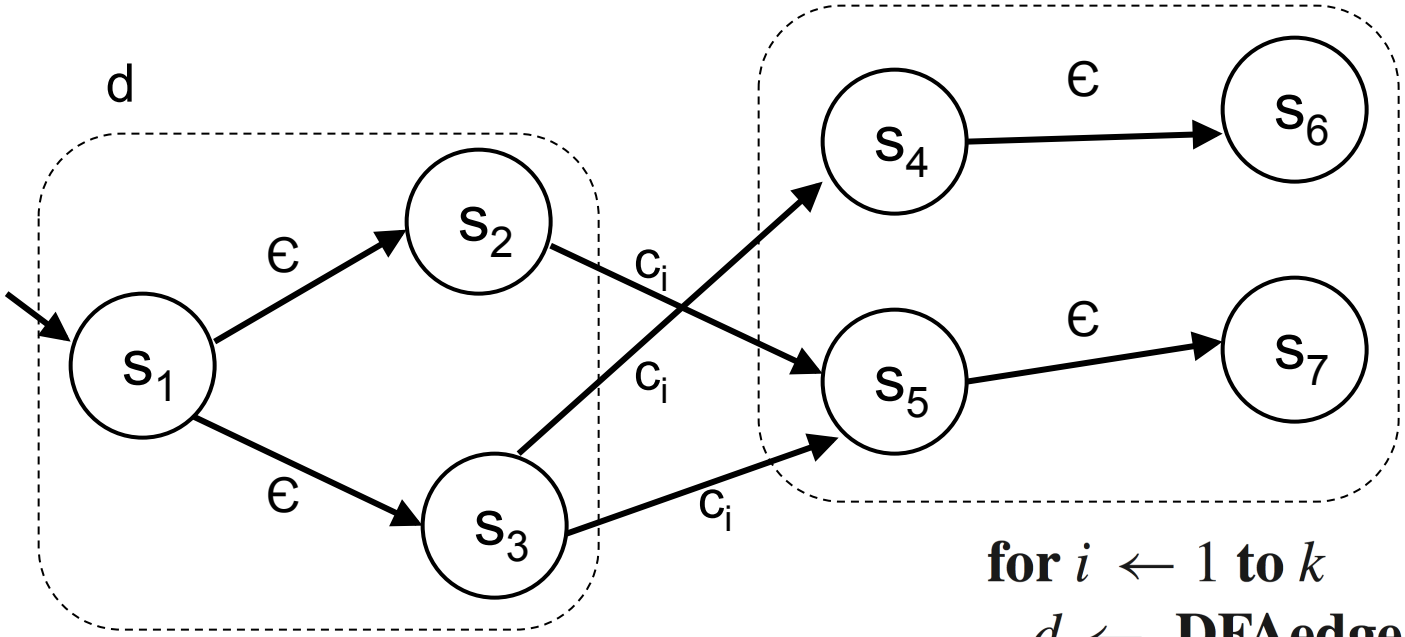
$$\textbf{closure}\left(\bigcup_{s \in d} \textbf{edge}(s, c)\right) = \{s_4, s_5, s_6, s_7\}$$

Algoritmo da Simulação

- Estado inicial s_1 e string c_1, ..., c_k

$$
\begin{aligned}
&d \leftarrow \textbf{closure}(\{s_1\}) \\
&\textbf{for } i \leftarrow 1 \textbf{ to } k \\
&\quad d \leftarrow \textbf{DFAedge}(d, c_i)
\end{aligned}
$$

DFAedge(d,c)



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

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

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

Convertendo NFA em DFA

```
states[0] ← {};      states[1] ← closure({s1})
p ← 1;      j ← 0
while j ≤ p
  foreach c ∈ Σ
    e ← DFAedge(states[j], c)
    if e = states[i] for some i ≤ p
      then trans[j, c] ← i
      else p ← p + 1
           states[p] ← e
           trans[j, c] ← p
  j ← j + 1
```

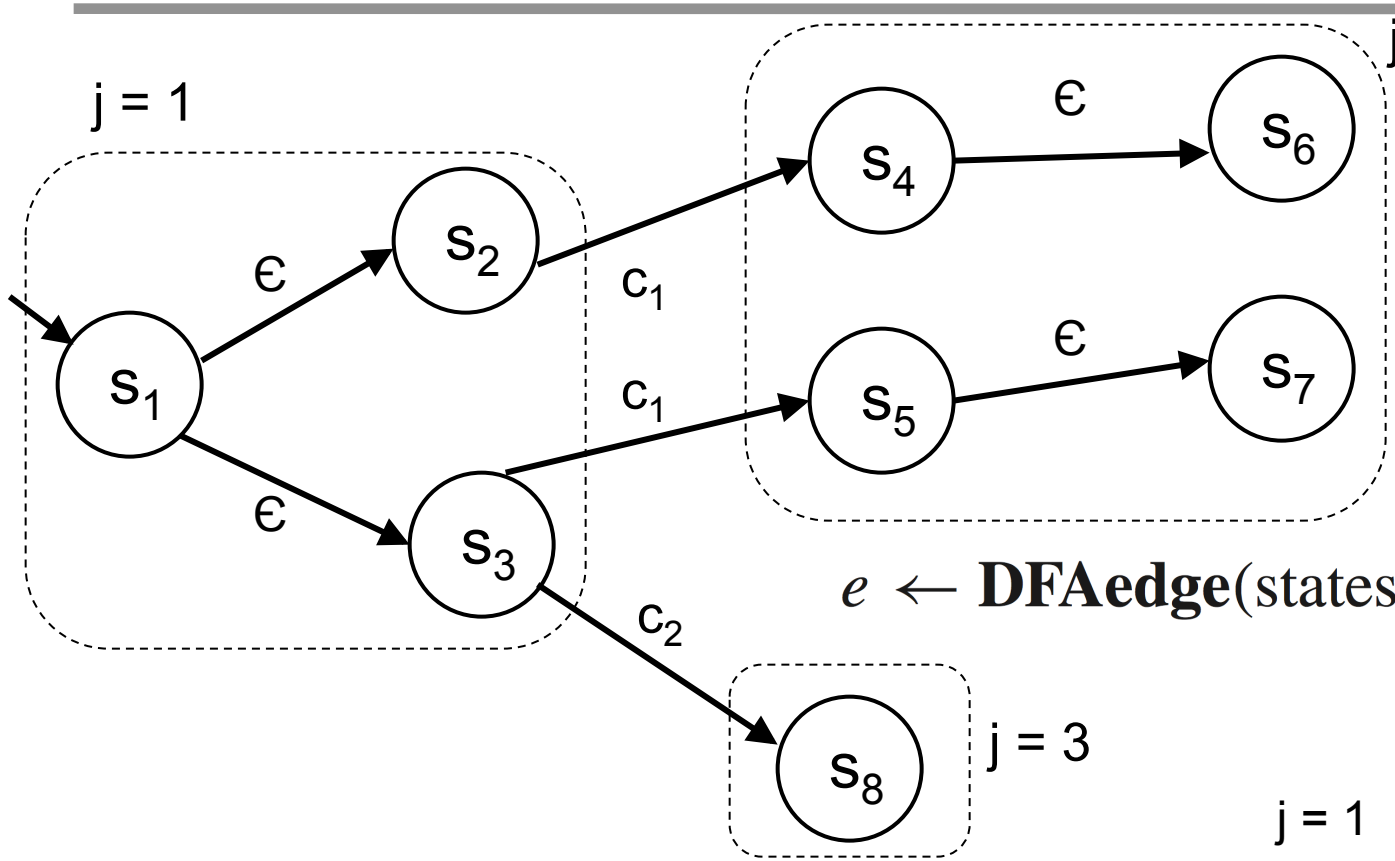
Convertendo NFA em DFA

```
states[0] ← {};      states[1] ← closure({s1})
p ← 1;      j ← 0
while j ≤ p
  foreach c ∈ Σ
    e ← DFAedge(states[j], c)
    if e = states[i] for some i ≤ p
      then trans[j, c] ← i
      else p ← p + 1
           states[p] ← e
           trans[j, c] ← p
  j ← j + 1
```

} Já vi este estado

} Ainda não vi este estado

DFAedge(d,c)



$e \leftarrow \textbf{DFAedge}(\text{states}[j], c)$

	c_1	c_2	.....
j = 1	2	3	.....
	.....		
	trans		

0 1 2 3

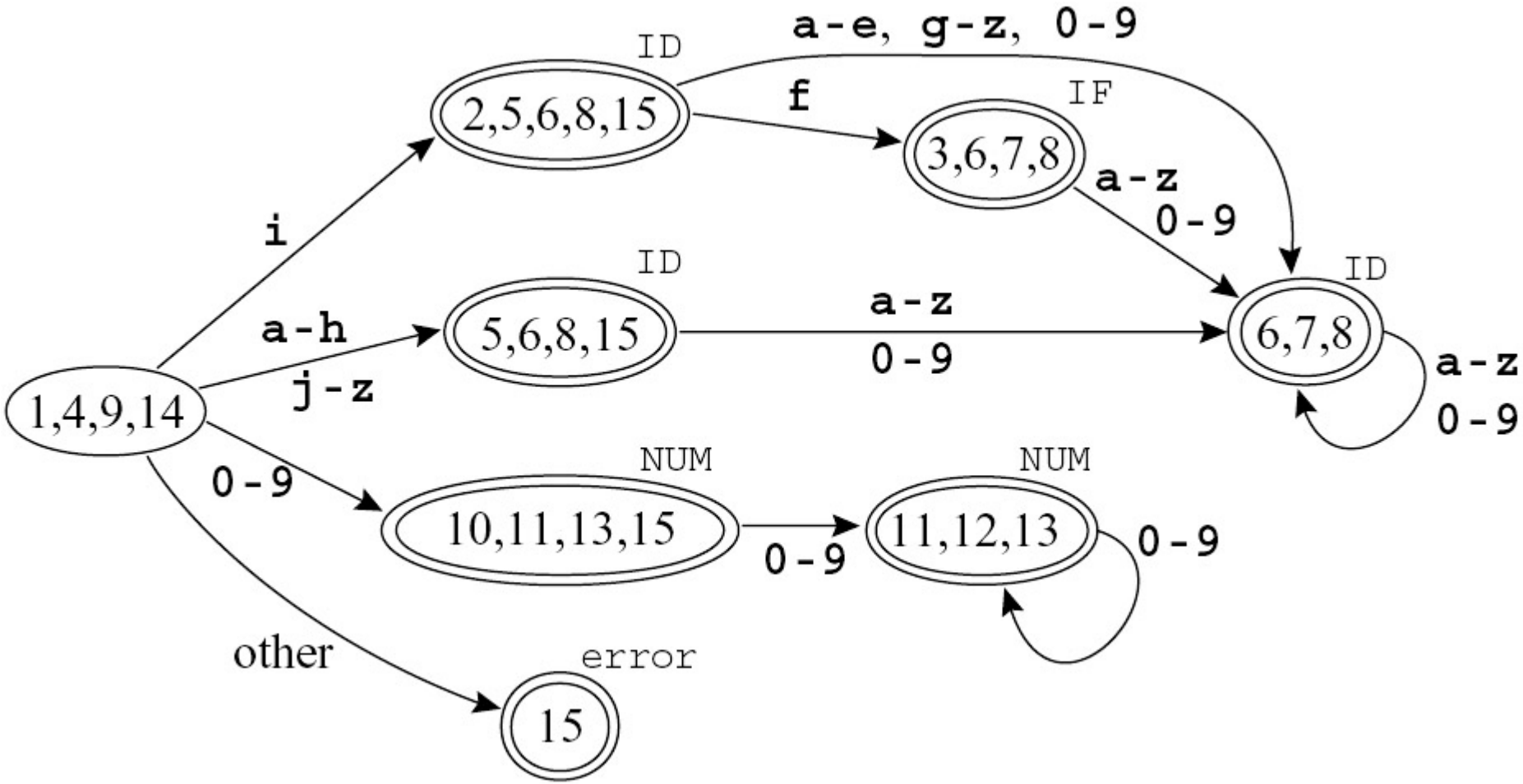
states [ ]= {{}, {s_1,s_2,s_3}, {s_4,s_5,s_6,s_7}, {s_8}}

p

Convertendo NFA em DFA

- 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

Convertendo NFA em DFA

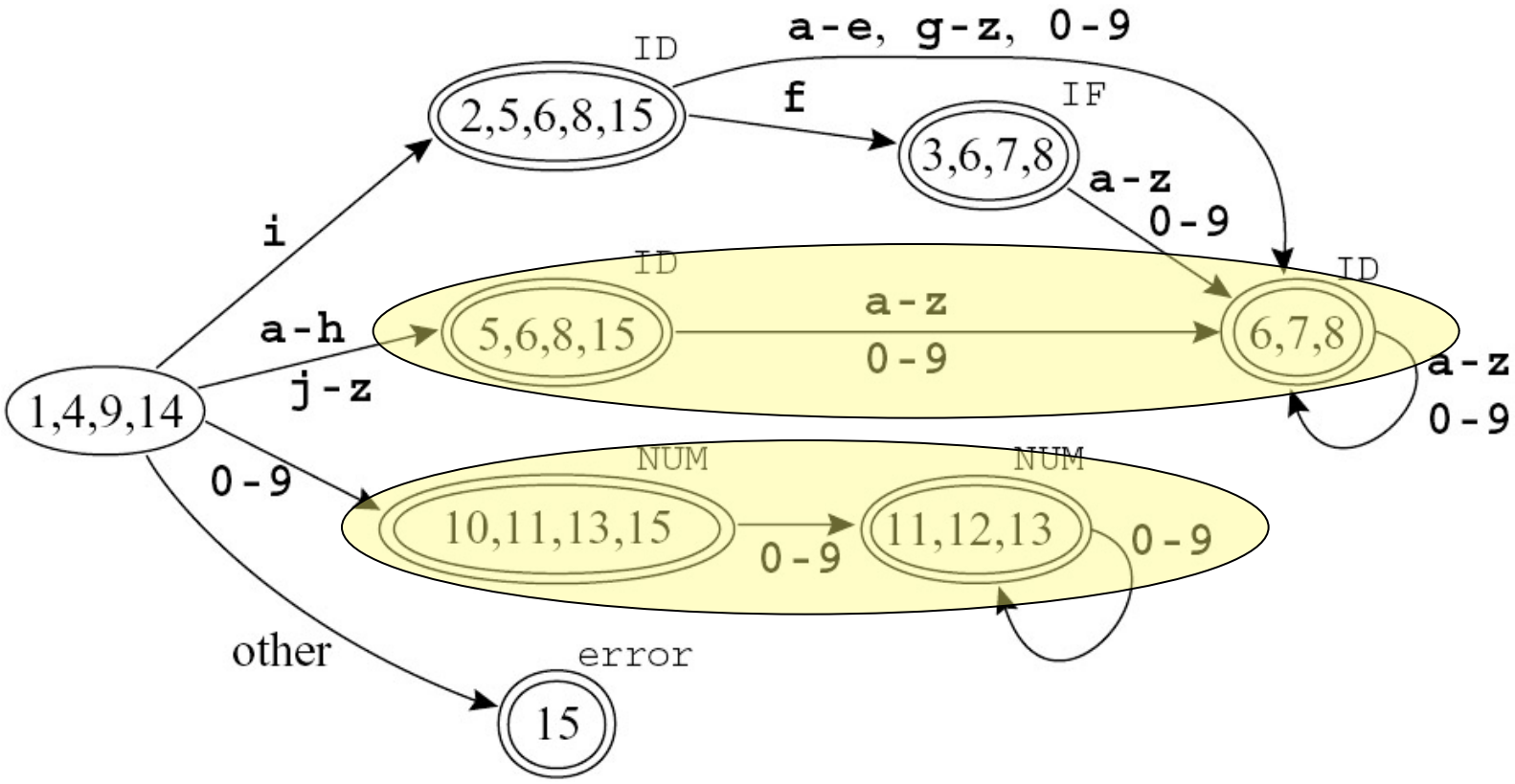


Convertendo NFA em DFA

- 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 σ começando em s_1 ⇔ ele também aceita σ começando em s_2

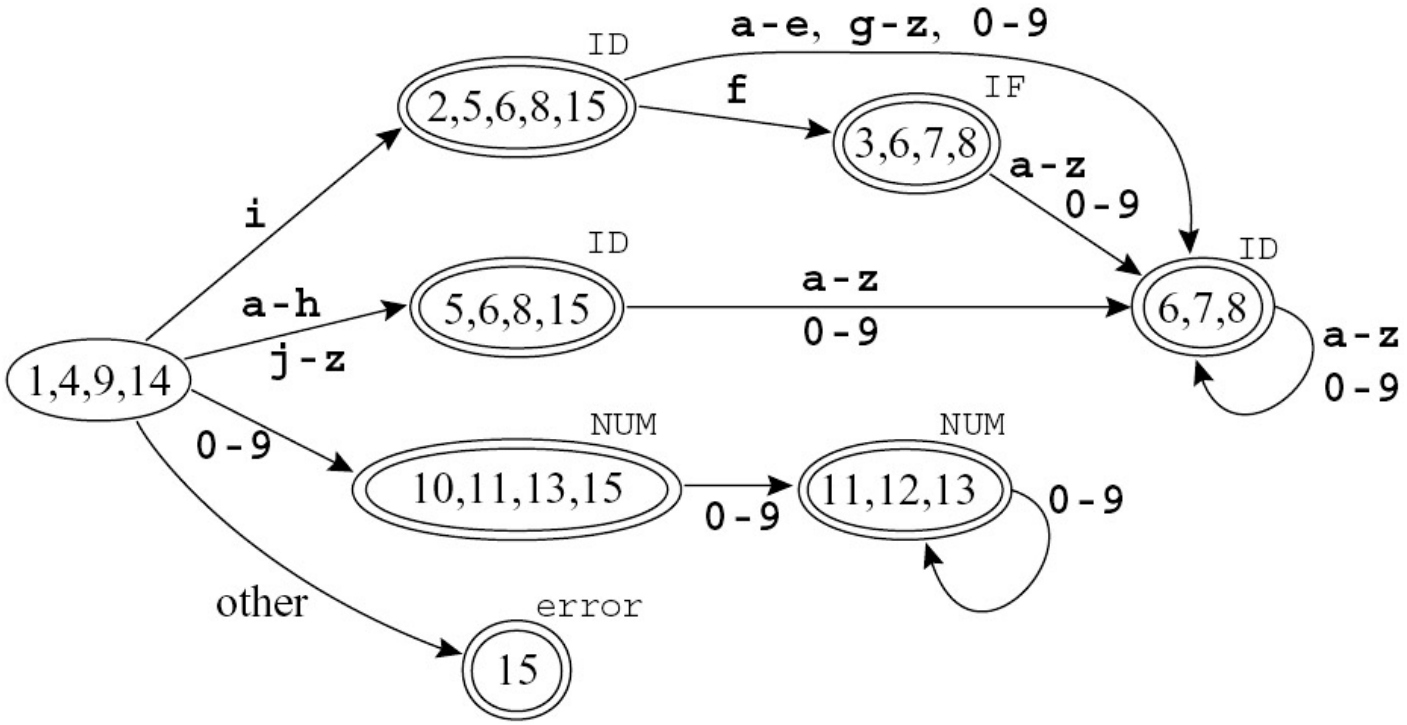
Convertendo NFA em DFA

- Quais estados são equivalentes no autômato anterior?



Convertendo NFA em DFA

- Como encontrar estados equivalentes?
  - trans[s1,c] = trans[s2,c] para ∀ c
  - Não é suficiente!!!



Convertendo NFA em DFA

- Contra-exemplo: