Compiler Principle

Prof. Dongming LU

Mar. 4th, 2024

Content

2 Lexical Analysis

2.3 Finite Automata

A finite automaton

A formalism Implemented as a computer program using finite automata (N.B. the singular of automata is automaton)

Definition

- A finite **set of *states***;
- ***Edges*** lead from one state to another, and each edge is **labeled with a *symbol*** ;
- One state is the ***start* state**, and certain of the states are distinguished as ***final* states**.

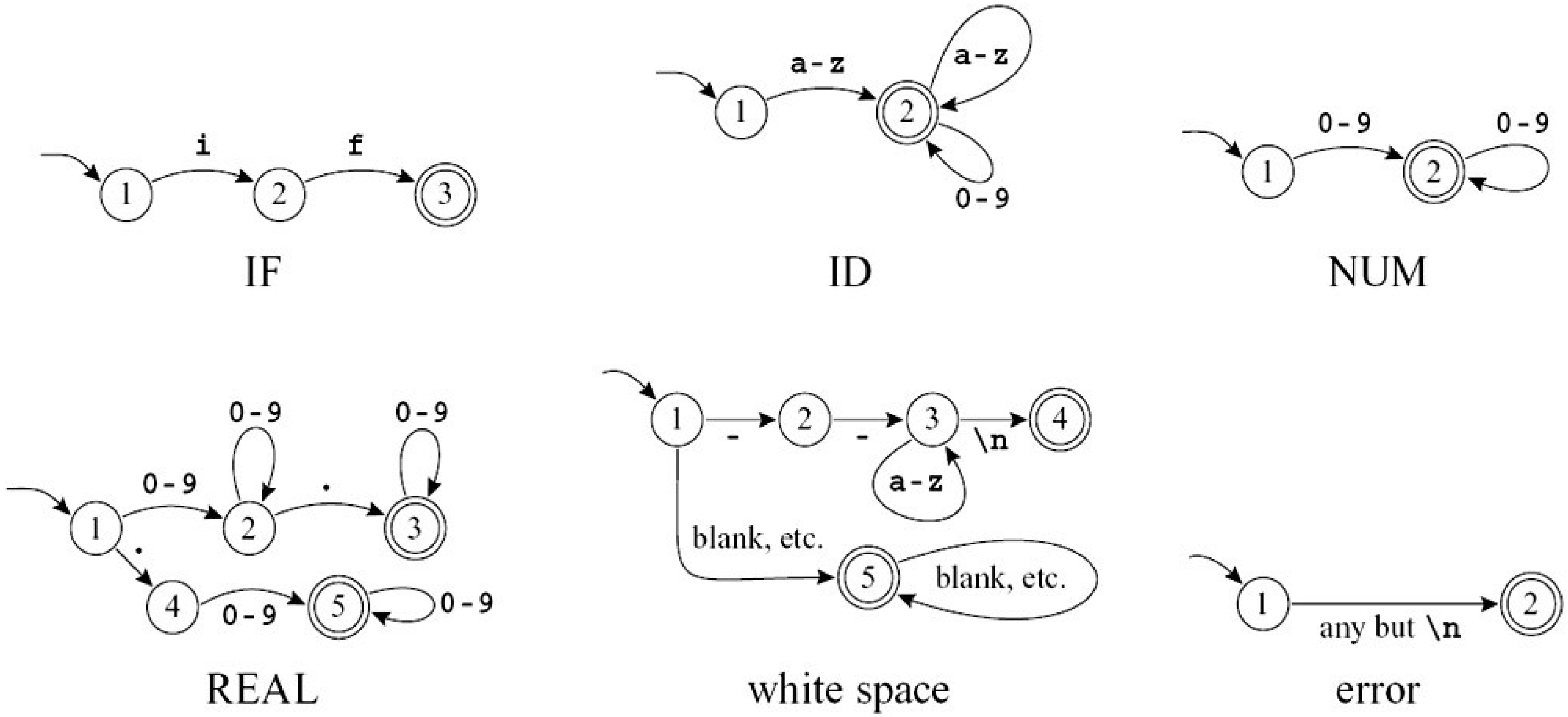
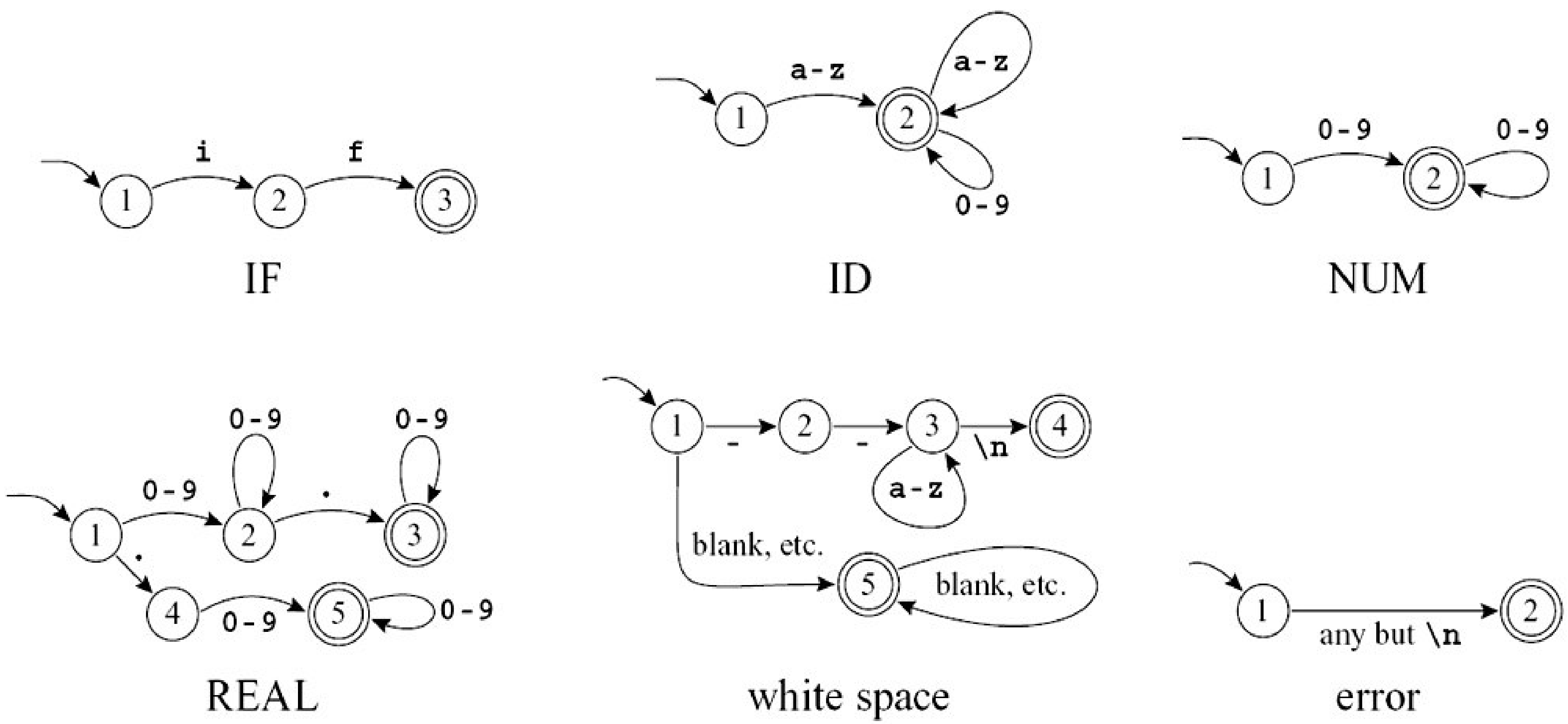


Figure 2.3: Finite automata for lexical tokens.

- The states are indicated by circles;
- Final states are indicated by double circles.
- The start state has an arrow coming in from nowhere.
- An edge labeled with several characters is shorthand for many parallel edges.

Deterministic Finite Automaton (DFA)

- No two edges leaving from the same state are labeled with the same symbol.



Deterministic Finite Automaton (DFA)

A DFA ***accepts*** or ***rejects*** a string as follows.

1. Starting in the start state, for each character in the input string the automaton follows exactly one edge to get to the next state.
2. The edge must be labeled with the input character.

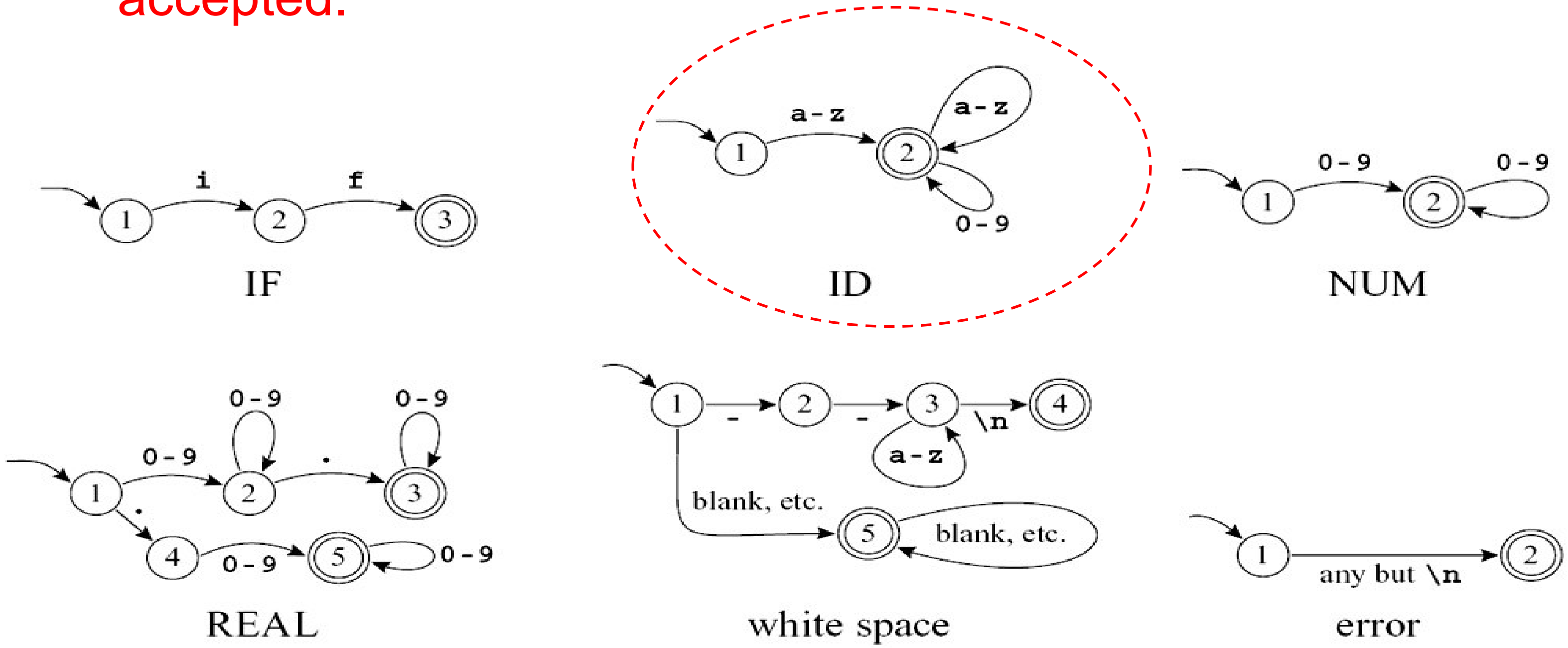
3. After making *n* transitions for an *n*-character string, if the automaton is in a **final** state, then it accepts the string.
4. If it is not in a **fina**l state, or if at some point there was no appropriately labeled edge to follow, it rejects.

The *language* recognized by an automaton is the set of strings that it accepts.

An example

- Any string in the language recognized by automaton ID must begin with a letter.

1. Any single letter leads to state 2, which is final; so a single-letter string is accepted.
2. From state 2, any letter or digit leads back to state 2, so a letter followed by any number of letters and digits is also accepted.



DFA and RE

- In fact, the machines of Figure 2.3 accept the **same** languages as the regular expressions of Figure 2.2

```
if                              {return IF;}
[a-z][a-z0-9]*                  {return ID;}
[0-9]+                          {return NUM;}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)          {return REAL;}
 ("--"[a-z]*"\n")|(" "|"\n"|"\t")+          {/*do nothing*/}
                                { error();}
```

Figure 2.2

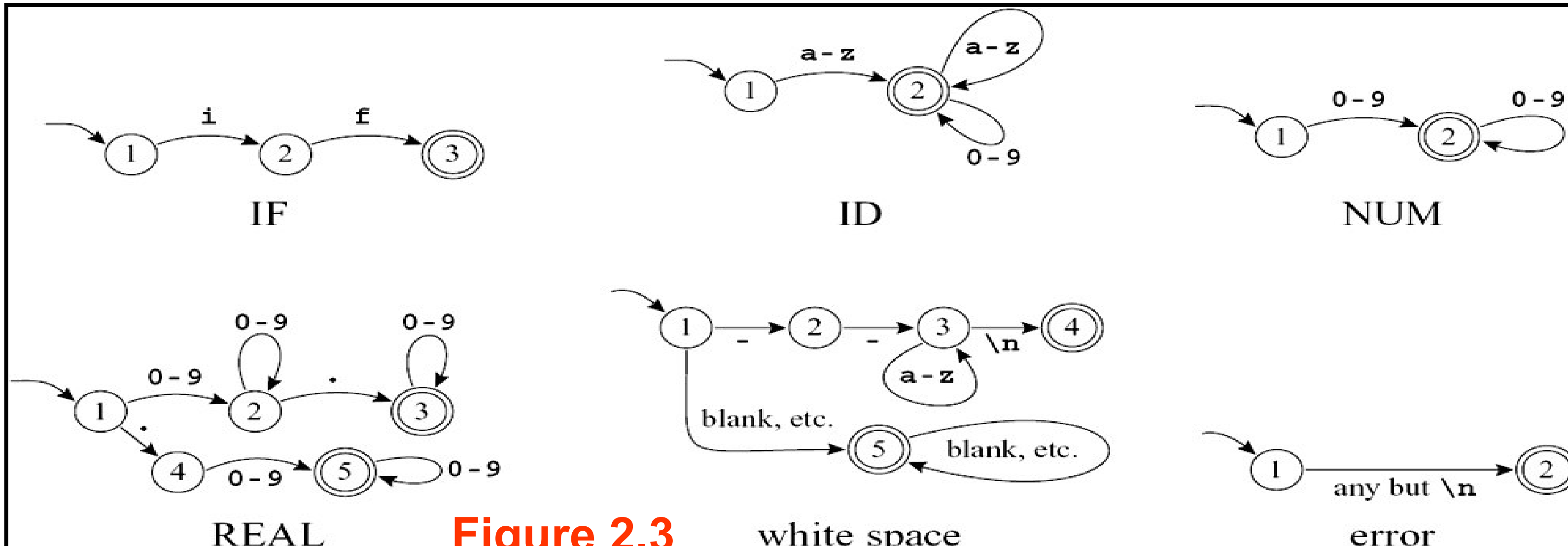
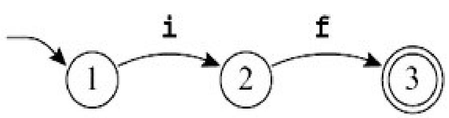


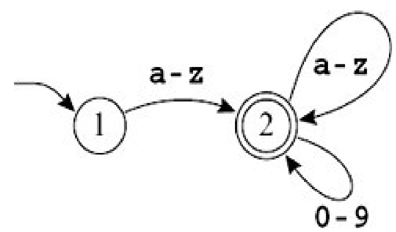
Figure 2.3

Combined finite automaton

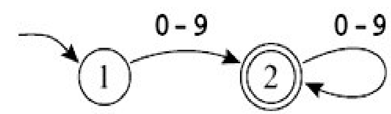
These are six separate automata; how can they be combined into a single machine that can serve as a lexical analyzer? ---- Ad hoc method!



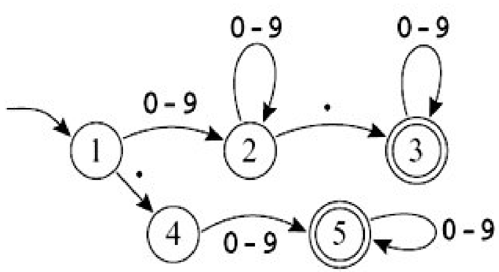
IF



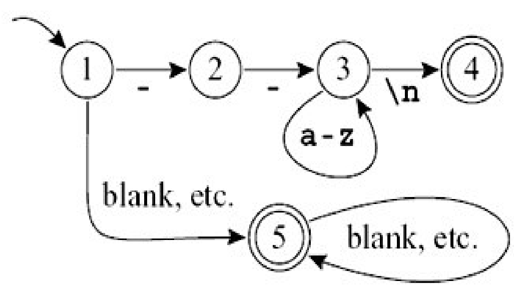
ID



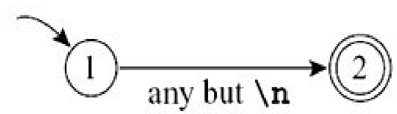
NUM



REAL



white space



error

Combined finite automaton

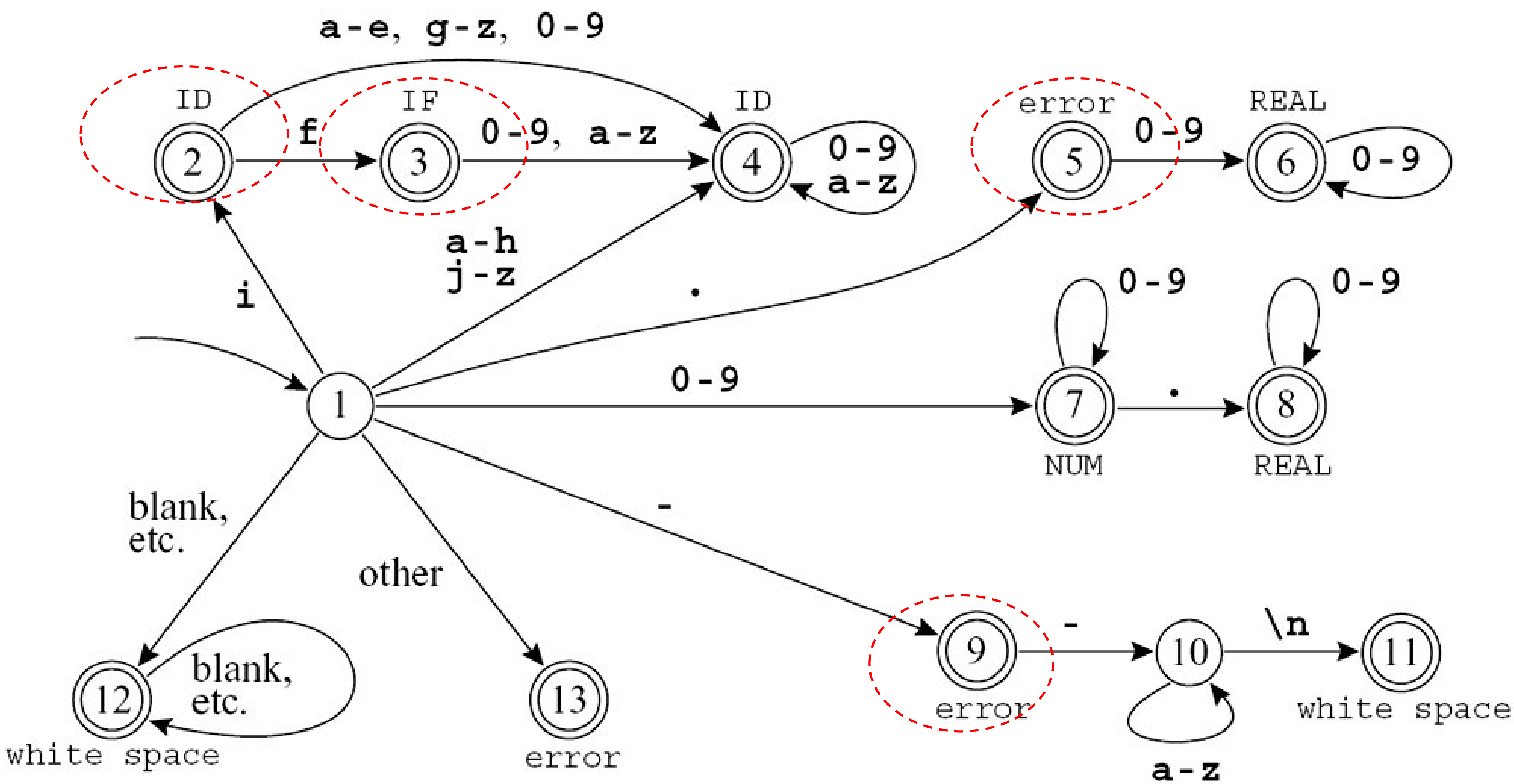


Figure 2.4: Combined finite automaton

Combined finite automaton

- Labeling each final state with the accepted token-type
- State 2 has aspects of *state 2* of the IF machine and *state 2* of the ID machine; since the latter is final, then the combined state must be final.
- State 3 is like *state 3* of the IF machine and *state 2* of the ID machine;
- To disambiguate both final using ***rule priority* to** - labeling state 3 with IF because we want this token to be recognized as a reserved word, not an identifier

A transition matrix

Encoding the above machine

- A two-dimensional array (a vector of vectors), subscripted by state number and input character.

```
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
}
```

A transition matrix

Encoding the above machine

- A "dead" state (state 0) that loops to itself on all characters; to encode the absence of an edge.

```
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
}
```

There is a "finality" array

- *Mapping state numbers to actions - final state 2 maps to action ID, and so on.*

RECOGNIZING THE LONGEST MATCH

The job of a lexical analyzer : to find the longest match

- The lexer must keep track of the longest match with two variables
  - ✓ **Last-Final** (the state number of the most recent final state encountered)
  - ✓ **Input-Position-at-Last-Final**

- Every time a final state is entered, the lexer updates these variables

- A *dead* state (a nonfinal state with no output transitions) reached : the variables tell what token was matched and where it ended.

Last Final	Current State	Current Input	Accept Action
0	1	`if --not-a-com`	
2	2	`|if --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 |--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*
0	1	`if --not-a-com`	
9	9	`if -|-not-a-com`	
9	0	`if -|-not-a-com`	*error, illegal token '-'; resume*

Figure 2.5: The automaton of Figure 2.4 recognizes several tokens.

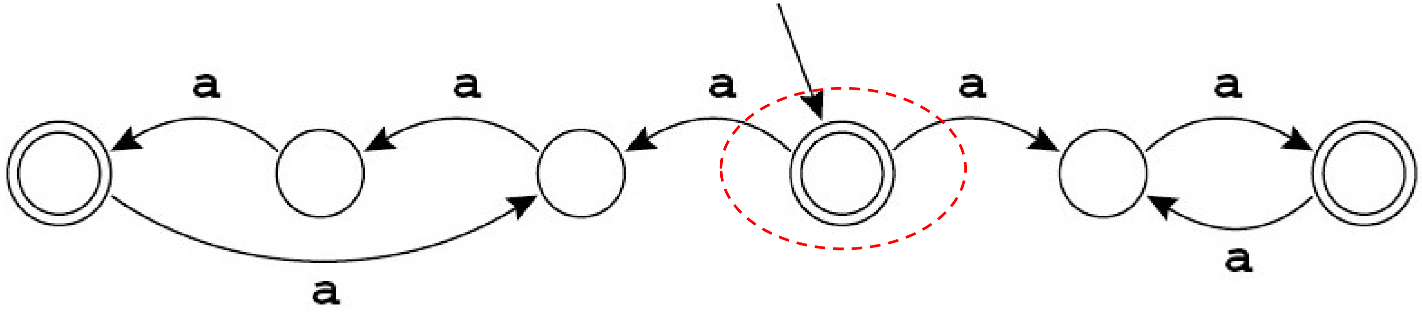
2.4 Nondeterministic Finite Automata

Nondeterministic Finite Automaton (NFA)

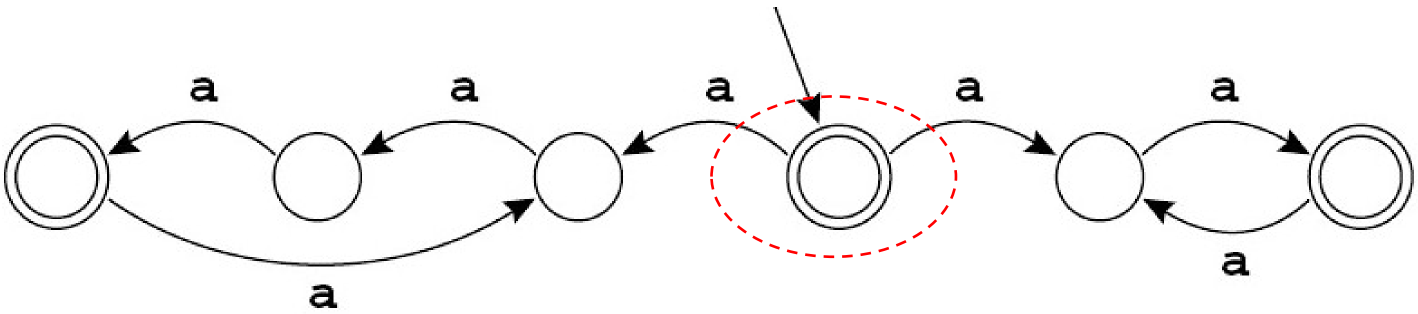
A NFA:

- **Have to choose** one from the edges (labeled with the same symbol -) to follow out of a state
- Have **special edges** labeled with ∈ (epsilon)

An example of an NFA:



Nondeterministic Finite Automaton (NFA)



The language recognized by this NFA

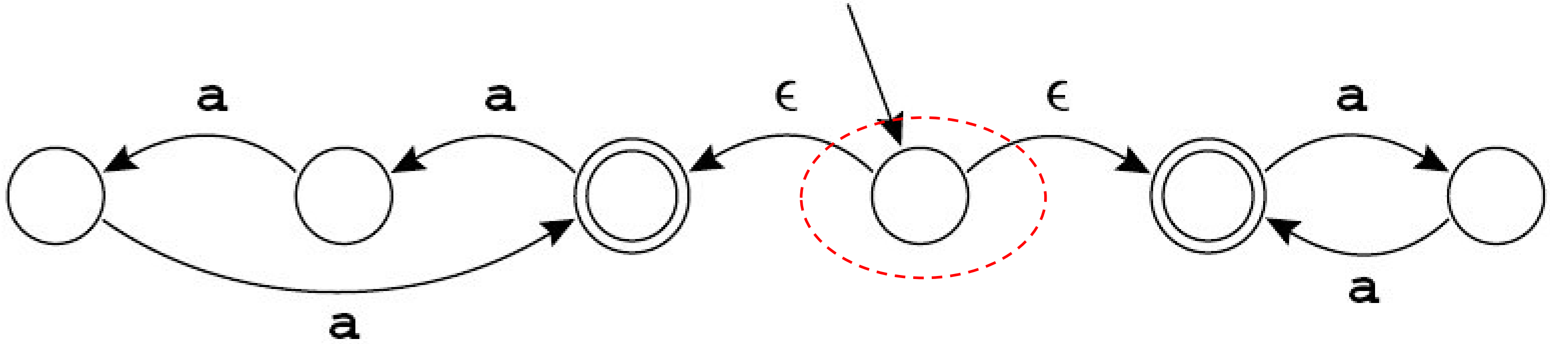
- All strings containing a multiple of two or three a's

On the first transition

- This machine must choose which way to go?
- Must "guess“ !! And Always guess correctly!!

Nondeterministic Finite Automaton (NFA)

Another NFA that accepts the same language



Edges labeled with ∈ may be taken
without using up a symbol from the input

The machine must choose which ∈-edge to take !!

A state with some ∈-edges and edges labeled by symbols

- Follow the corresponding symbol-labeled edge
- Or to follow an ∈-edge instead

Why NFA?

A (static, declarative) regular expression can be easy to be converted to a (simulatable, quasi-executable) NFA.

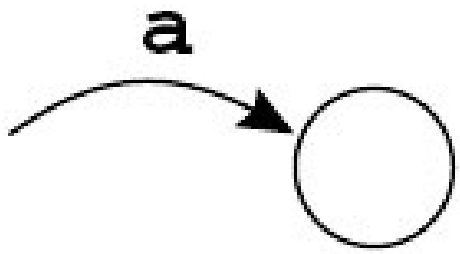
From a RE to an NFA

The conversion algorithm:

Turning each regular expression into an NFA with a *tail* (start edge) and a *head* (ending state).

For example:

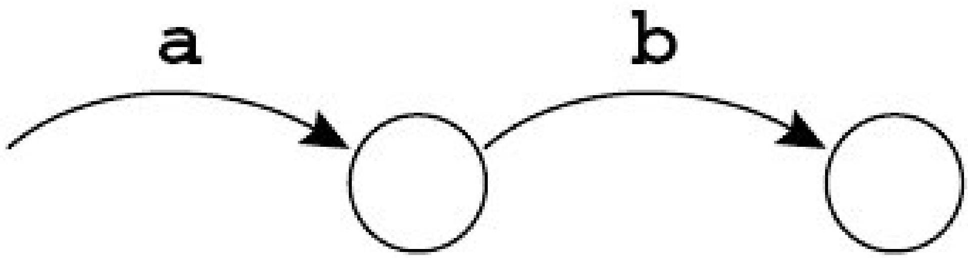
The single-symbol regular expression **a**



From a RE to an NFA

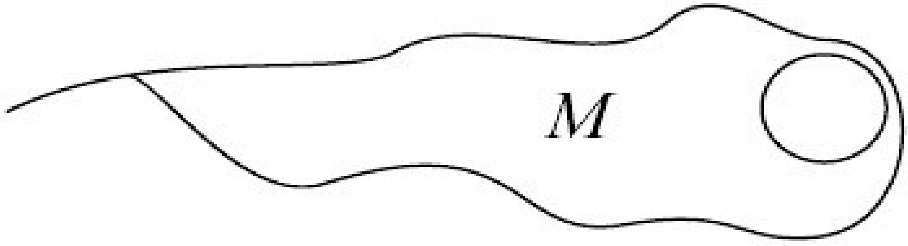
The regular expression **ab:**

- Combining the two NFAs
- Hooking the head of **a** to the tail of **b**



In general, any regular expression *M*

- Some NFA with a tail and head:



From a RE to an NFA

The rules for translating regular expressions to nondeterministic automata

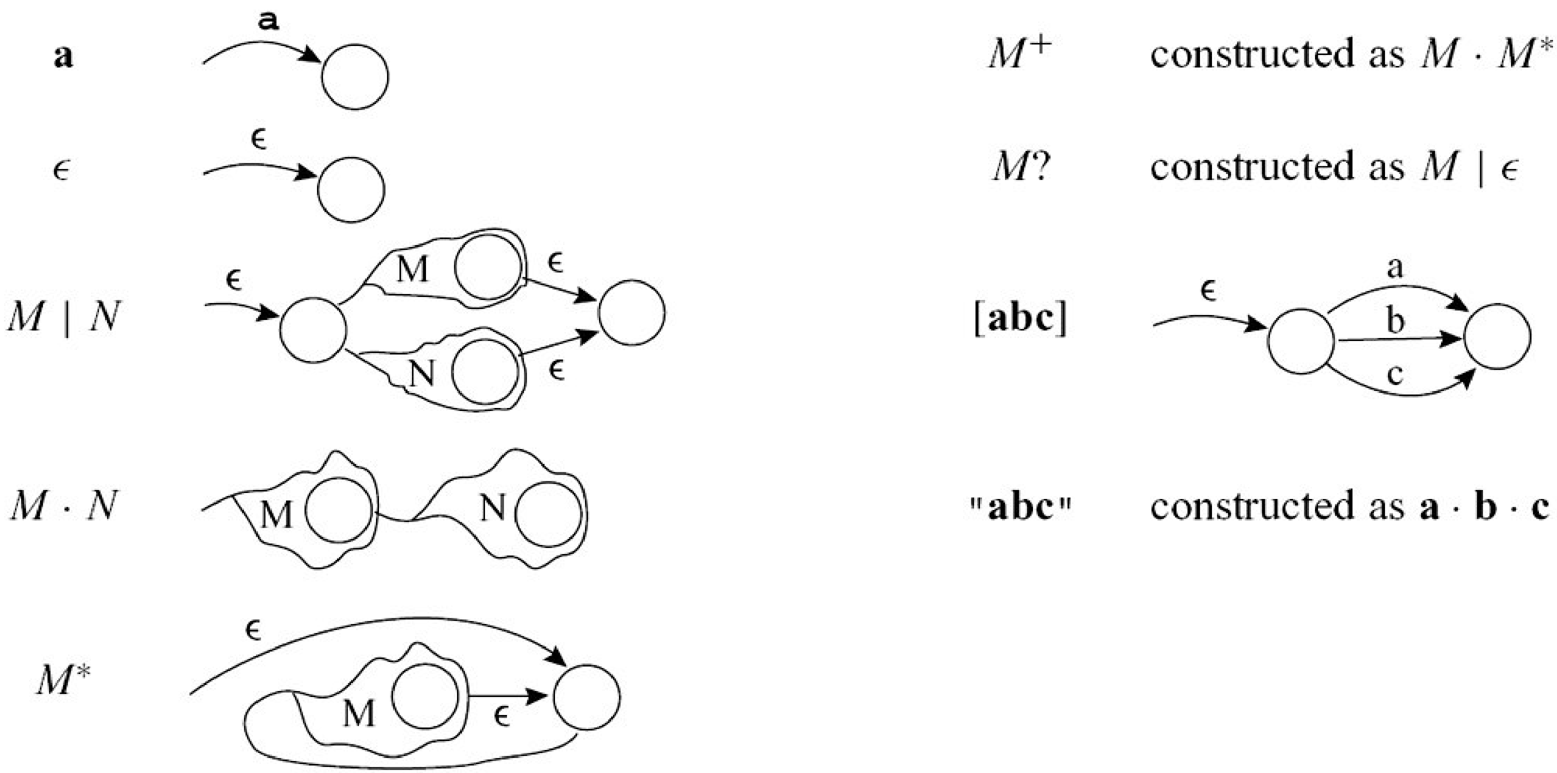


Figure 2.6: Translation of regular expressions to NFAs.

From a RE to an NFA

The merged NFA for **IF**, **ID**, **NUM**, and **error**

- Each expression is translated to an NFA,
- The "head" state marked final with a different token type
- The tails of all the expressions joined to a new start node

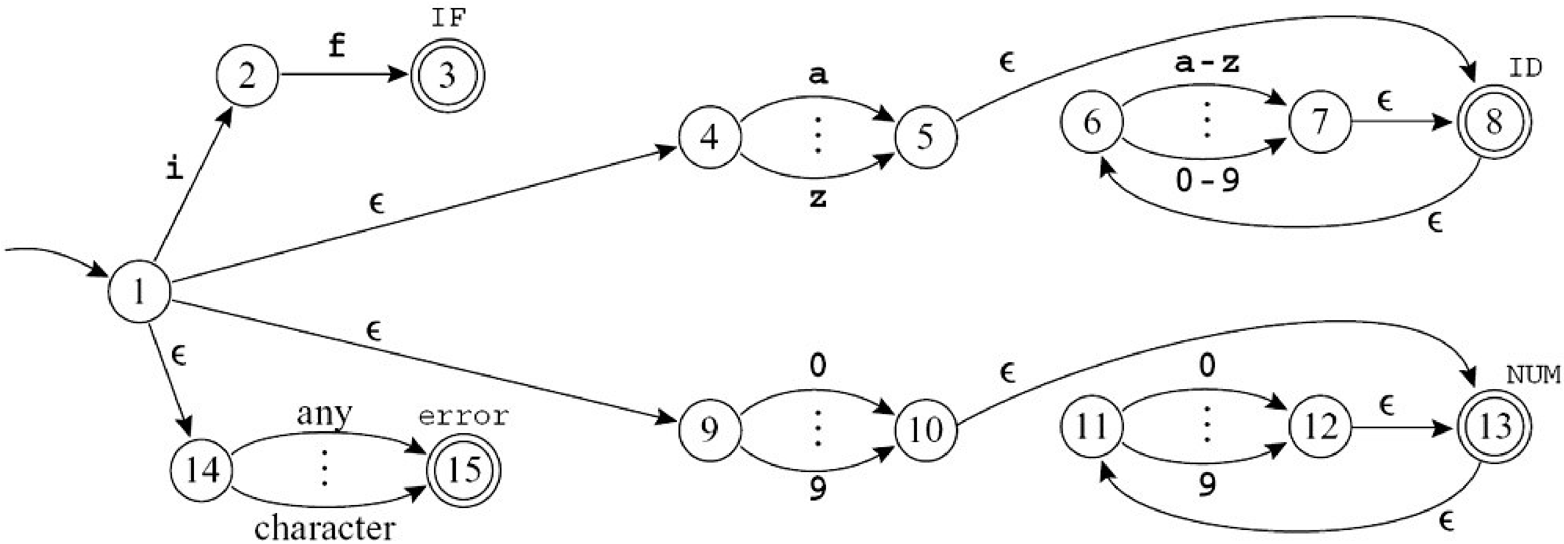


Figure 2.7: Four regular expressions translated to an NFA

WHY CONVERTING AN NFA TO A DFA

Implementing deterministic finite automata (DFAs) as computer programs is easy

From an NFA to a DFA

To avoid guesses by trying every possibility at once

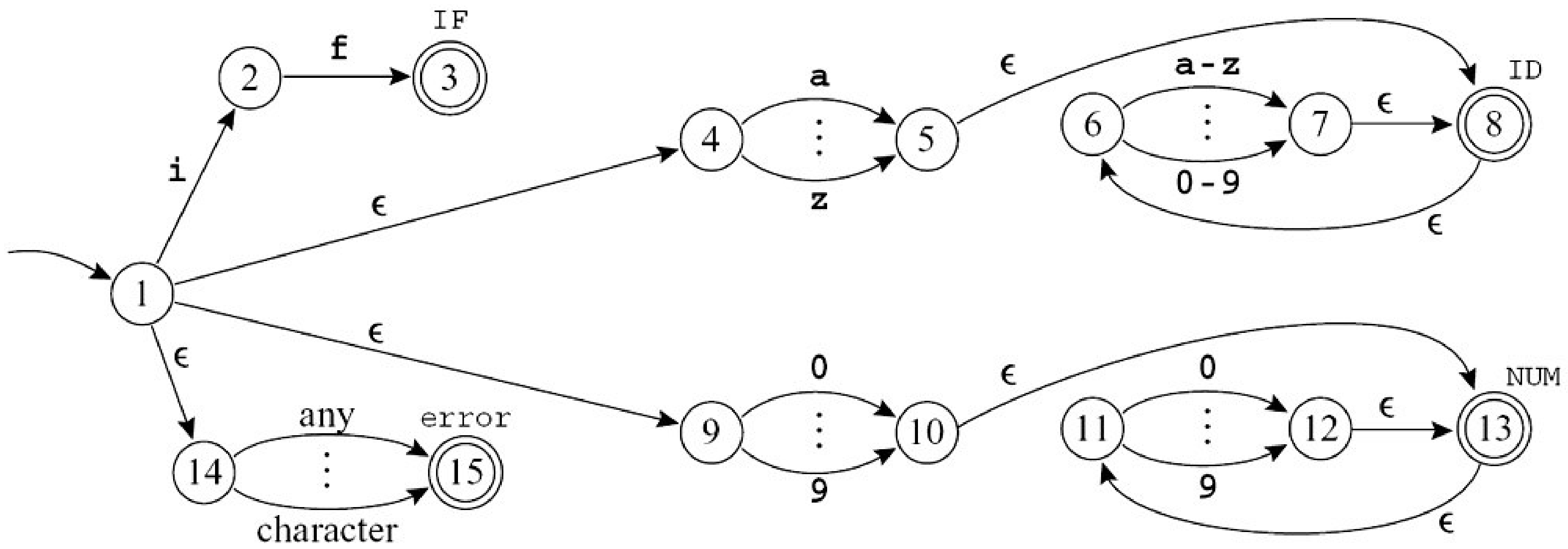


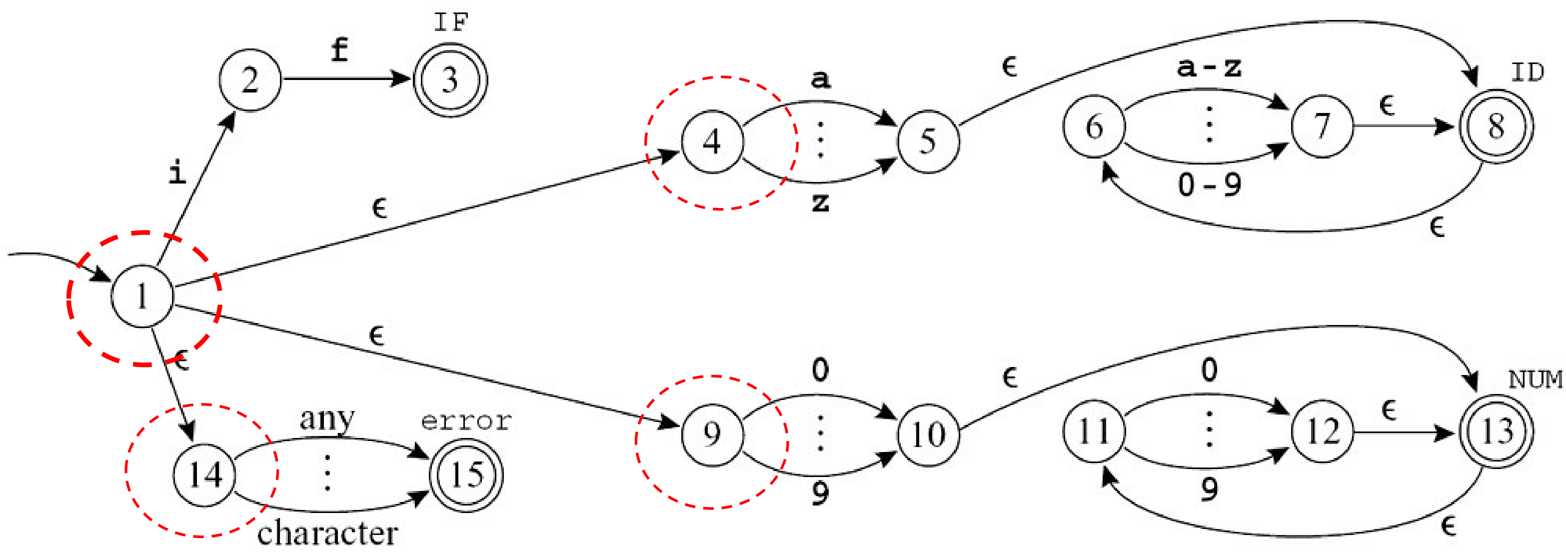
Figure 2.7: Four regular expressions translated to an NFA.

Simulating the NFA of Figure 2.7 on the string *in*

From an NFA to a DFA

Starting in state 1

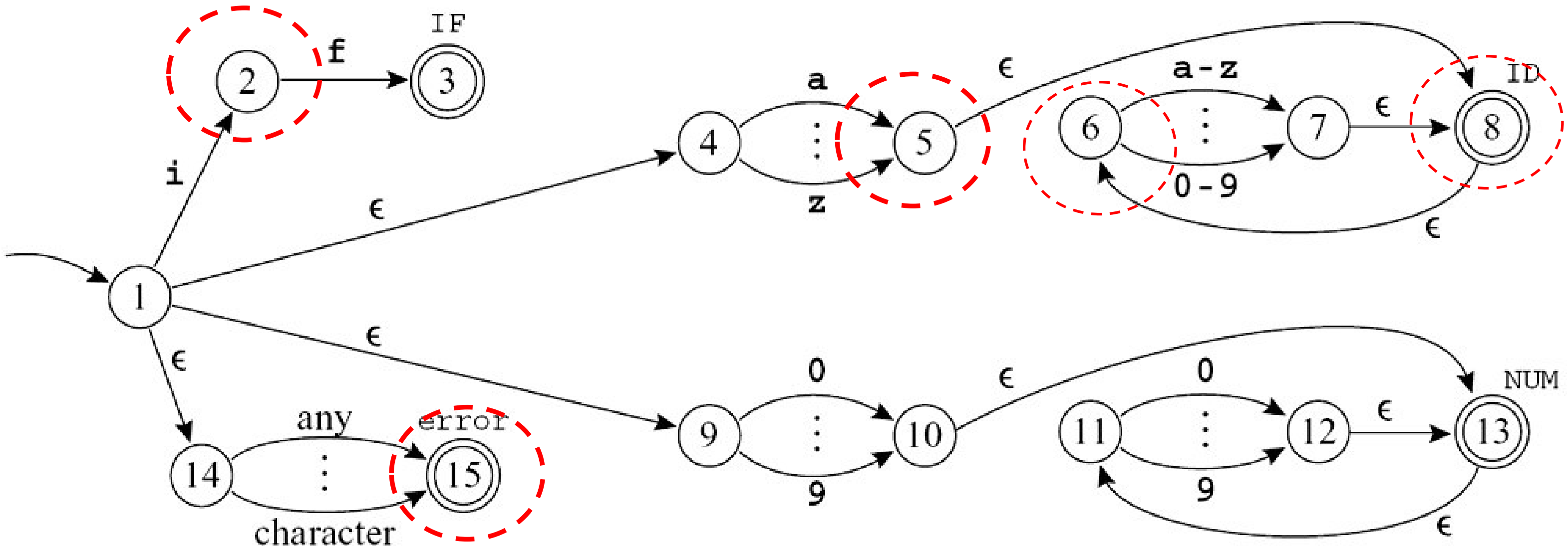
- Instead of guessing which ∈-transition to take, the NFA might **take any of them**
  - ✓ It is in **one of the states {1, 4, 9, 14};**
  - ✓ That is, **the ∈-*closure* of {1}**
- No other states reachable without eating the first character



From an NFA to a DFA

Making the transition on the character i

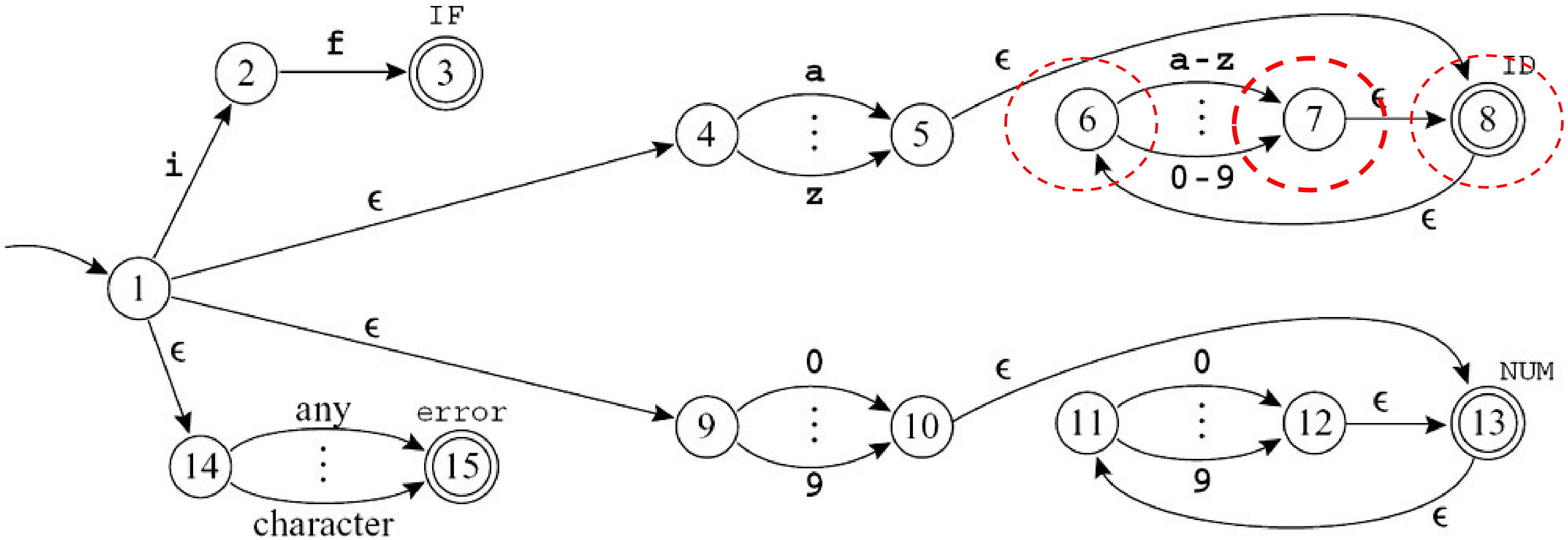
- From state 1 to reach 2; from 4 to 5, from 9 to nowhere, and from 14 to 15
  - ✓ So the set {2, 5, 15}.
- Again compute the ∊-closure
  - ✓ From 5 there is an ∊-transition to 8 and From 8 to 6
  - ✓ So the NFA in one of the states {2, 5, 6, 8, 15}.



From an NFA to a DFA

On the character n {2, 5, 6, 8, 15}.

- Get from state 6 to 7, from 2 to nowhere, from 5 to nowhere, from 8 to nowhere, and from 15 to nowhere.
- So the set {7}; its ∊-closure is {6, 7, 8}.



From an NFA to a DFA

Formally define ∈-closure as follows

1. Let **edge**(s, c) be the set of all NFA states reachable by following a single edge with label c from state s.

2. For a set of states S, **closure**(S) is the set of states that can be reached from a state in S without consuming any of the input, that is, **by going only through ∈-edges.**

Mathematically, express the idea of going through ∈-edges by saying that **closure**(S) is the smallest set T such that

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

From an NFA to a DFA

Calculating T by iteration

$$
\begin{aligned}
&T \leftarrow S \\
&\textbf{repeat } T' \leftarrow T \\
&\qquad T \leftarrow T' \cup \left(\textstyle\bigcup_{s \in T'} \textbf{edge}(s, \epsilon)\right) \\
&\textbf{until } T = T'
\end{aligned}
$$

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

T can only grow in each iteration

- The final T must include S.
- T must also include $\bigcup_{s \in T'} \textbf{edge}(s, \epsilon)$

The algorithm must terminate, because there are only a finite number of distinct states in the NFA.

From an NFA to a DFA

Simulating an NFA as described above

- Suppose a set $d = \{s_i; s_k; s_l\}$ of NFA states s_i ; s_k; s_l.
- Starting in *d* and eating the input symbol *c*, reaching a new set of NFA states called set **DFAedge**(*d; c*)

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

The NFA simulation algorithm more formally using **DFAedge**

- If the start state of the NFA is *s*1, the input string is *c*1,…, *ck,* The algorithm is

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

From an NFA to a DFA

Manipulating sets of states is expensive

- Costly to do on every character in the source program
- Do all the sets-of-states calculations **in advance**.

Making a DFA from the NFA

- Each set of NFA states corresponds to one DFA state
- The NFA has a finite number *n* of states
- The DFA will have a finite number (at most 2^n) of states.

From an NFA to a DFA

DFA construction with **closure** and **DFAedge** algorithms.

- The DFA start state *d*1 is just **closure**(*s*1)
- There is an edge from *di* to *dj* labeled with *c* if *dj* = **DFAedge**(*di*, *c*).

Let Σ be the alphabet, DFA construction is as follows:

```
states[0] ← {};     states[1] ← closure({s_1})
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
```

From an NFA to a DFA

A state *d* is *final* in the DFA

- if any NFA state in states[*d*] is final in the NFA

Labeling a state *final* is not enough

- Also say what token is recognized

Several members of states[*d*] are final in the NFA

- Label *d* with the token-type that occurred first in the list of regular expressions
- How *rule priority* is implemented.

After the DFA is constructed

- The "states" array may be discarded
- The "trans" array is used for lexical analysis.

Not visit unreachable states of the DFA

- In principle the DFA has 2^n states
- But only about *n* of them are reachable from the start state.

To avoid an exponential blowup in the size of the DFA interpreter's transition tables

Applying the DFA construction algorithm to the NFA of Figure 2.7 gives the automaton in Figure 2.8

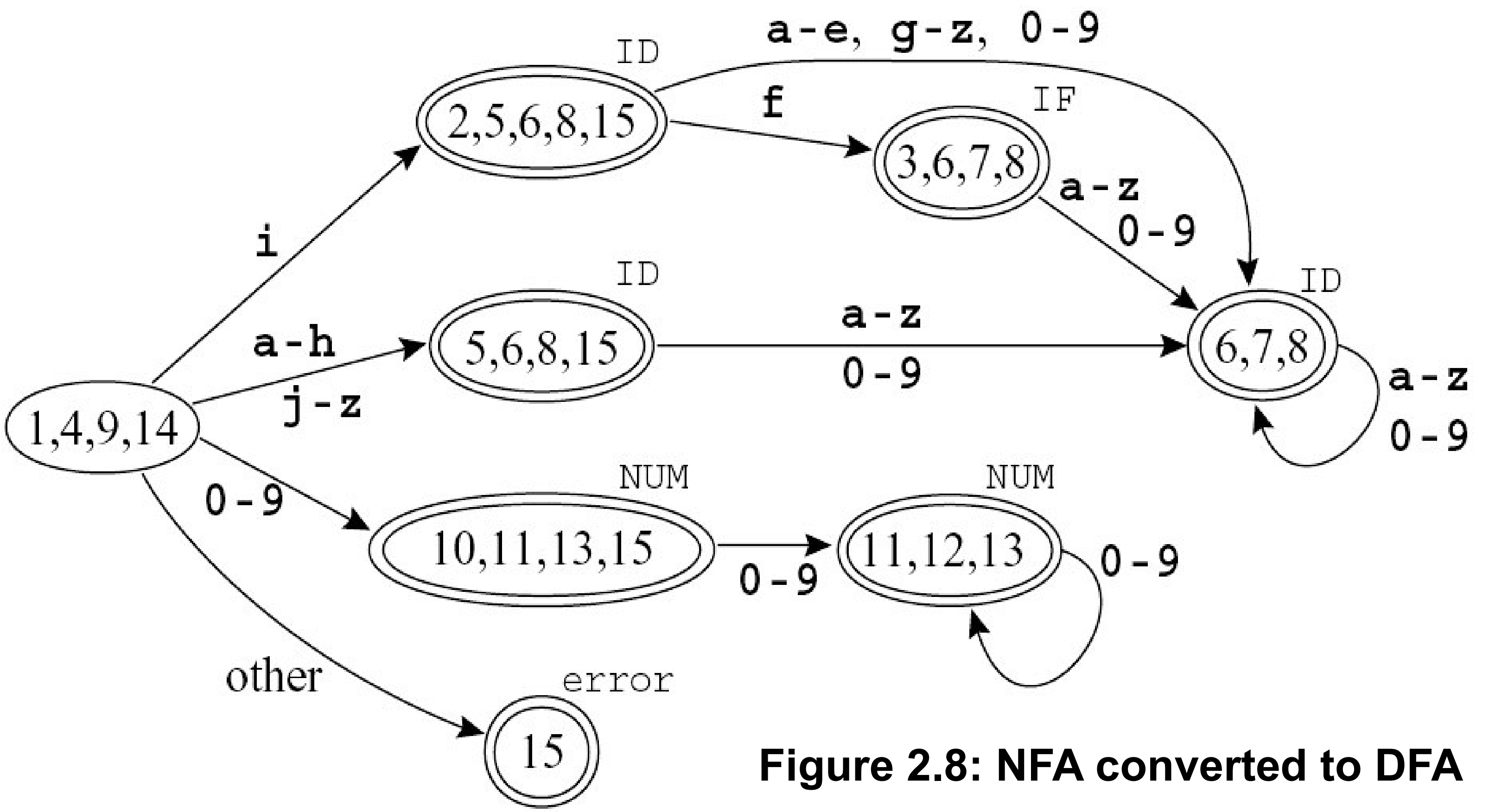


Figure 2.8: NFA converted to DFA

This automaton is suboptimal: not the smallest one that recognizes the same language

The equivalent states

Two states s_1 and s_2 are equivalent

- The machine starting in s_1 accepts a string σ if and only if starting in s_2 it accepts σ.
- Such as ： the states labeled {5,6,8,15} and {6,7,8} in Figure 2.8, and of the states labeled{10,11,13,15} and {11,12,13}

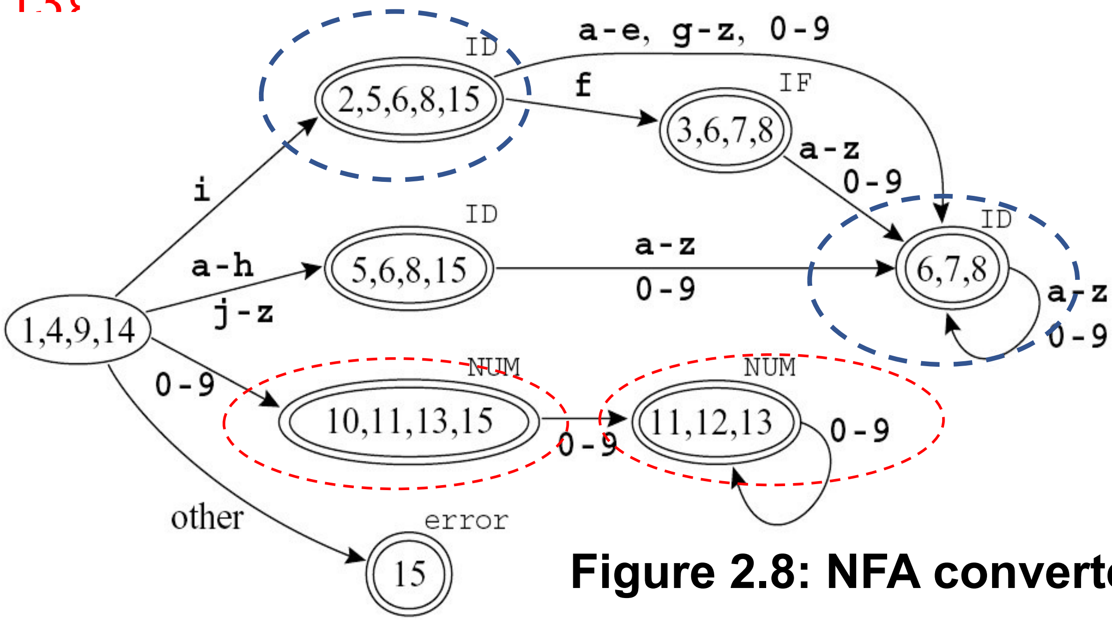


Figure 2.8: NFA converted to DFA

In an automaton with two equivalent states s_1 and s_2

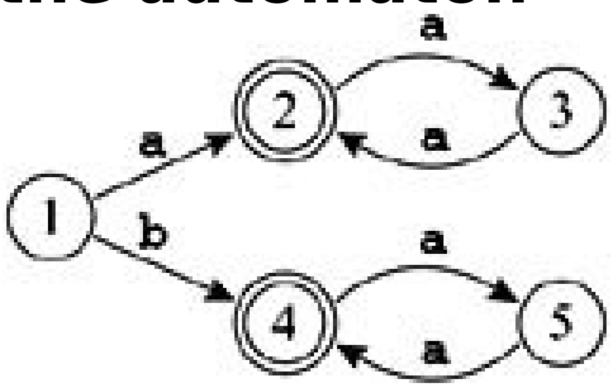
- Make all of s_2's incoming edges point to s_1 instead and delete s_2

How can we find equivalent states?

Certainly, *s*1 and *s*2 are equivalent if they are both final or both nonfinal and, for any symbol *c*, trans[*s*1, *c*] = trans[*s*2, *c*];

{10,11,13,15} and {11,12,13} satisfy this criterion.

This condition is not sufficiently general; consider the automaton



Here, states 2 and 4 are equivalent, but trans[2, *a*] ≠ trans[4, *a*].

2.5 Lex: A Lexical Analyzer Generator

DFA construction

- A mechanical task easily performed by computer
- An automatic *lexical-analyzer generator* to translate regular expressions into a DFA

Lex:

- A lexical analyzer generator that produces a C program from a lexical specification.

```
%{
/* C Declarations:*/
#include “tokens.h”  /*definition of IF, ID ,NUM, …*/
#include “errormsg.h”
union {int ival; string sval; double fval;} yylval;
Int charPos=1
#define ADJ  (EM_tokPos=charPos, charPos+=yyleng)
%}

/* Lex Definitions; */
Digits [0-9]+

%%
/* Regular Expressions and Actions:*/
If                                          {ADJ; return IF;}
[a-z][]a-z0-9]*                             {ADJ; yyval.sval=String(yytext); return ID;}
{digits}                                    {ADJ; yyval.ival=atoi(yytext); return NUM;}
({digits}”.”[0-9]*)|([0-9]*”.”{digits})     {ADJ; yyval.fval=atof(yytext); return REAL;}
(“--”[a-z*”\n”])|(“ “|”\n”|”\t”)+           {ADJ;}
.                                           {ADJ; EM-error(“illegal character”)}
```

For each token type in the programming language

- The specification contains a regular expression and an action.

- The action: Communicates the token type to the next phase of the compiler

The output of Lex is a program in C

- A lexical analyzer that interprets a DFA using the algorithm described in Section 2.3 and executes the action fragments on each match.

The action fragments

- The C statements that return token values

The end of Chapter 2(2)