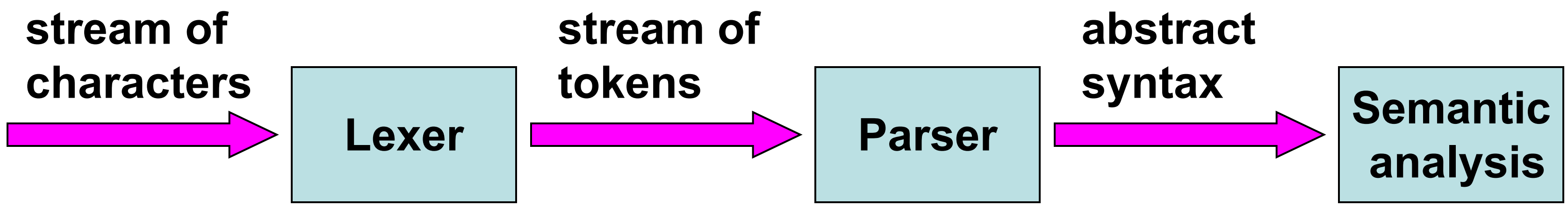
Outline

Today:

– Lexical Analysis

– Reading: Chapter 2 of Appel

The Front End



- Lexical Analysis: Create sequence of tokens from characters
- Syntax Analysis: parsing the phrase structure of the program;Create abstract syntax tree from sequence of tokens
- Semantic analysis (Type Checking): calculating the program’s meaning.

- Lexical Analysis: Breaks stream of ASCII characters (source) into tokens
- Token: An atomic unit of program syntax
  - i.e., a word as opposed to a sentence
- Tokens and their types:

Characters Recognized:	**Type:**	**Token:**
foo, x, listcount	ID	ID(foo), ID(x), ...
10.45, 3.14, -2.1	REAL	REAL(10.45), REAL(3.14), ...
;	SEMI	SEMI
(	LPAREN	LPAREN
50, 100	NUM	NUM(50), NUM(100)
if	IF	IF

x = (y + 4.0) ;

Lexical Analysis

ID(x) ASSIGN LPAREN ID(y) PLUS REAL(4.0) RPAREN SEMI

2.1 Lexical Token

2.1 Lexical Token

A lexical token: A sequence of characters

A unit in the grammar of a programming language

Classification of lexical tokens: A finite set of token types

Some of the token types of a typical programming language:

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

2.1 Lexical Token

Reserved words**,in most languages, not be used as identifiers**

such as IF, VOID, RETURN

Examples of non-tokens:

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

The preprocessor deletes the non-tokens

- **Operates on the source character stream**
- **Producing another character stream to the lexical analyzer**

2.1 Lexical Token

Given a program such as

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

The lexical analyzer will return the stream:

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

The token-type of each token is reported

Some of the tokens attached ***semantic values***

Such as identifiers and literals, with auxiliary information

2.1 Lexical Token

How should the lexical rules of a programming language be described?

In what language should a lexical analyzer be written?

An ad hoc lexer

- **Any reasonable programming language can be used to implement it**

A simpler and more readable lexical analyzers

(1) *Regular expressions* :Specify lexical tokens

(2) *Deterministic finite automata* : Implementing lexers

(3) Mathematics : Connecting the above two

2.2 Regular Expression

2.2 Regular Expression

A *language* is a set of *strings*

A string is a finite sequence of *symbols*

A symbol is taken from a finite *alphabet*

Symbol: a

- **For each symbol a in the alphabet of the language**

The regular expression a denotes the language containing just the string a.

2.2 Regular Expression

Alternation: A vertical bar ||

Given two regular expressions *M* and *N*, the alternation operator makes a new regular expression *M* || *N*.

A string is in the language of *M* || *N* if it is in the language of *M* or in the language of *N*.

Example:

The language of a || b contains the two strings a and b

2.2 Regular Expression

Concatenation: operator ·

Given two regular expressions *M* and *N*, the concatenation makes a new regular expression *M* · *N*.

A string is in the language of *M* · *N* if it is the concatenation of any two strings α and β such that α is in the language of *M* and β is in the language of *N*.

Example:

The regular expression (a || b) · a defines the language containing the two strings aa and ba.

2.2 Regular Expression

Epsilon: ϵ

The regular expression ϵ represents a language whose only string is the empty string.

Example: ($a \cdot b$) || ϵ represents the language {"", "ab"}.

Repetition: *

Given a regular expression *M*, its Kleene closure is *M. A string is in *M** if it is the concatenation of zero or more strings, all of which are in *M*.**

Example: ((a || b) · a)* represents the infinite set {"", "aa", "ba", "aaaa", "baaa", "aaba", "baba", "aaaaaa", …}.

2.2 Regular Expression

In writing regular expressions, sometimes the concatenation symbol or the epsilon will be omitted

Assuming that

- **Kleene closure "binds tighter" than concatenation**
- **Concatenation binds tighter than alternation;**

so that

- **ab | c means (a · b) | c, and (a |) means (a | ∊)**

Introducing some more abbreviations:

[**abcd**] means (**a** | **b** | **c** | **d**),

[**b-g**] means [**bcdefg**],

[**b-gM-Qkr**] means [**bcdefgMNOPQkr**],

***M*?** means **(*M* | ∊)**, and ***M*+** means **(*M*·*M**)**.

2.2 Regular Expression

a	An ordinary character stands for itself.
ϵ	The empty string.
	Another way to write the empty string.
$M \mid N$	Alternation, choosing from M or N.
$M \cdot N$	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.
$''\mathrm{a.+*}''$	Quotation, a string in quotes stands for itself literally.

Figure 2.1: Regular expression notation.

2.2 Regular Expression

```
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: Regular expressions for some tokens

The fifth line of the description recognizes comments or white space but does not report back to the parser.

Instead:

- **The white space is discarded and the lexer resumed**
- **The comments begin with two dashes, contain only alphabetic characters, and end with new-line.**

2.2 Regular Expression

Ambiguous：

Does if8 match as a single identifier or as the two tokens if and 8?

Does the string if 89 begin with an identifier or a reserved word?

Two important disambiguation rules：

Longest match:

The longest initial substring of the input that can match any regular expression is taken as the next token.

Rule priority:

For a *particular* longest initial substring, the first regular expression that can match determines its token-type.

Thus,

if8 matches as an identifier by the longest-match rule

if matches as a reserved word by rule-priority.

2.3 Finite Automata

2.3 Finite Automata

Regular expressions:

Convenient for specifying lexical tokens

Needing a formalism:

Implemented as a computer program

Using finite automata.

A finite automaton :

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**.

2.3 Finite Automata

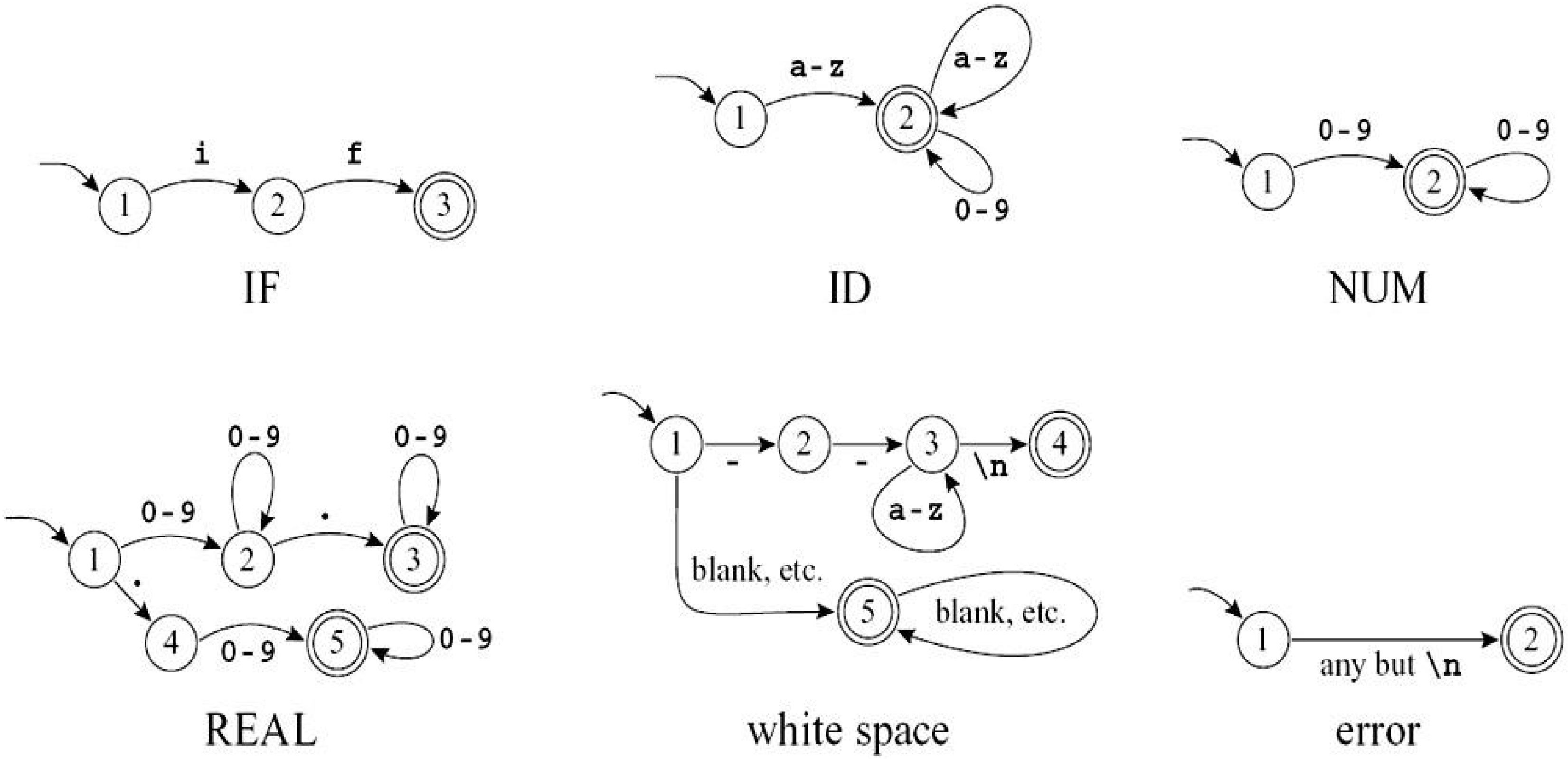


Figure 2.3: Finite automata for lexical tokens.

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.

2.3 Finite Automata

DFA (*deterministic* finite automaton)： no two edges leaving from the same state are labeled with the same symbol.

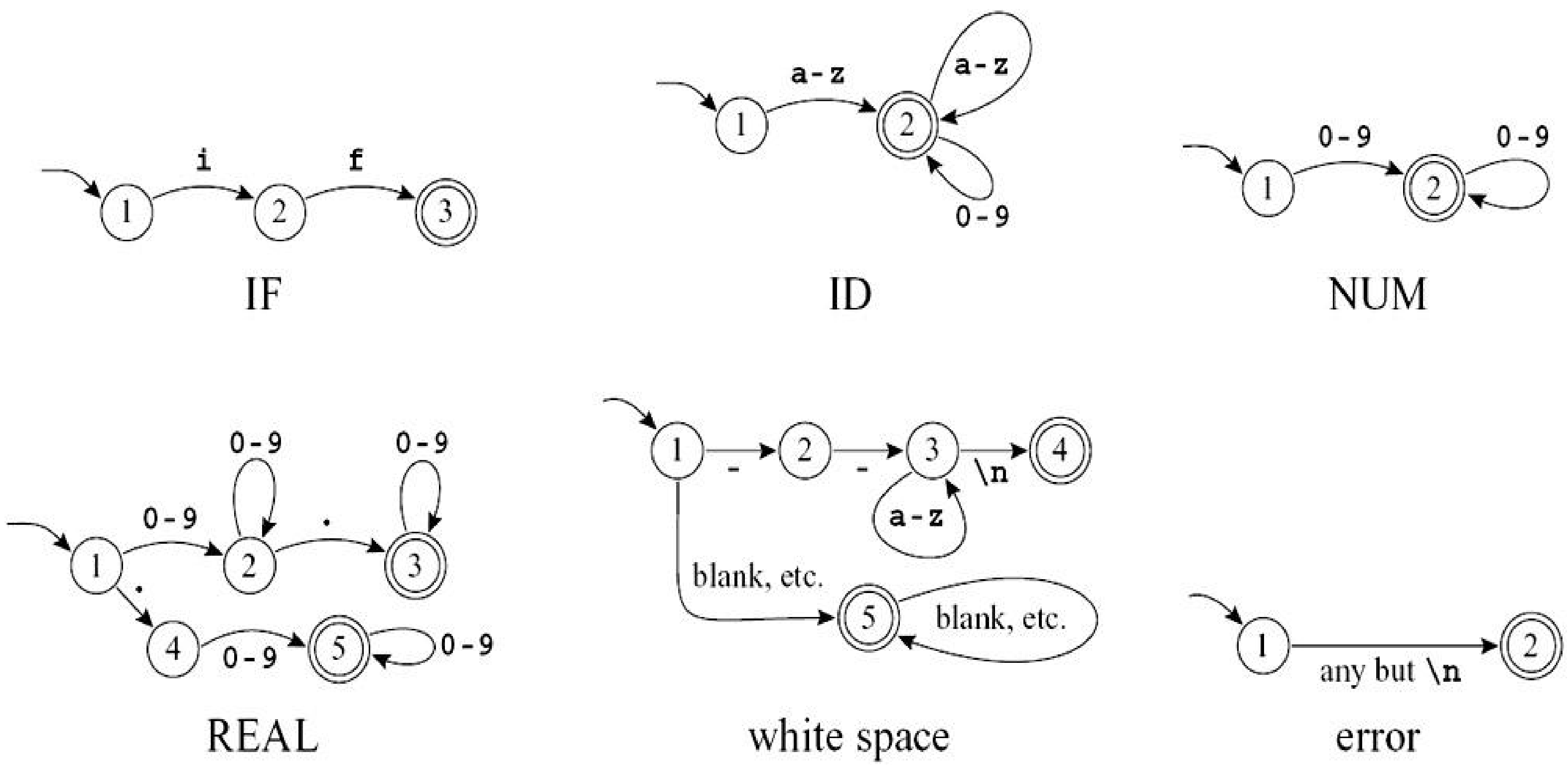
A DFA *accepts* or *rejects* a string as follows.

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 final 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.

2.3 Finite Automata

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



2.3 Finite Automata

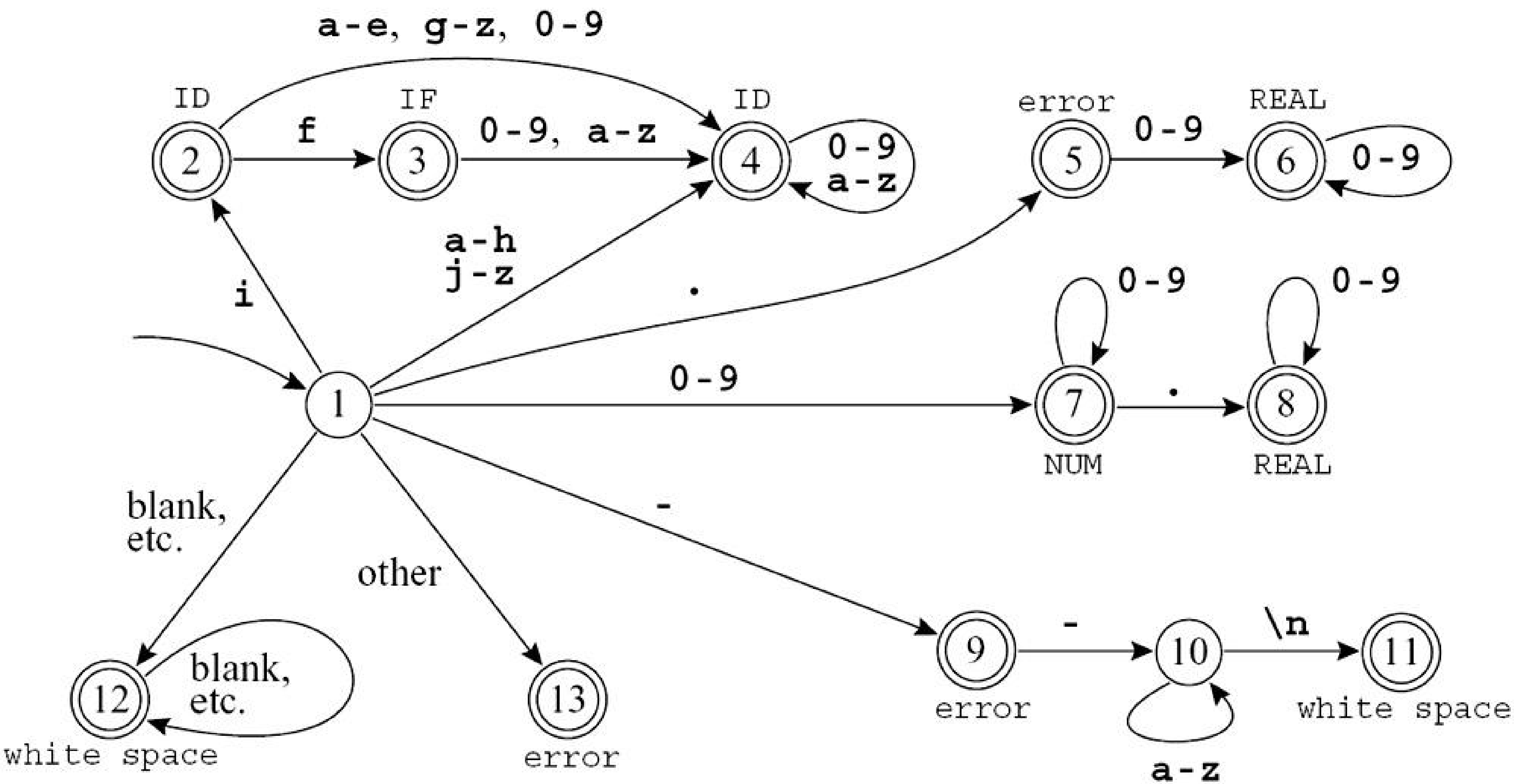


Figure 2.4: Combined finite automaton

（ **study formal ways of doing this in the next section** ）

2.3 Finite Automata

Encoding this machine as a transition matrix:

- a two-dimensional **array** subscripted by state number and input character.
- **“dead” state (state 0)** that loops to itself on all characters： to encode the absence of an edge.
- **a "finality" array:** mapping state numbers to actions
  - - final state 2 maps to action ID, and so on.

```
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
}
```

2.3 Finite Automata

RECOGNIZING 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:

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

2.3 Finite Automata

<table>
<tr><th>Last Final</th><th>Current State</th><th>Current Input</th><th>Accept Action</th></tr>
<tr><td>0</td><td>1</td><td>if --not-a-com</td><td></td></tr>
<tr><td>2</td><td>2</td><td>|if --not-a-com</td><td></td></tr>
<tr><td>3</td><td>3</td><td>|if --not-a-com</td><td></td></tr>
<tr><td>3</td><td>0</td><td>|if --not-a-com</td><td>return IF</td></tr>
<tr><td>0</td><td>1</td><td>if --not-a-com</td><td></td></tr>
<tr><td>12</td><td>12</td><td>if| --not-a-com</td><td></td></tr>
<tr><td>12</td><td>0</td><td>if| --not-a-com</td><td>found white space; resume</td></tr>
<tr><td>0</td><td>1</td><td>if --not-a-com</td><td></td></tr>
<tr><td>9</td><td>9</td><td>if |--not-a-com</td><td></td></tr>
<tr><td>9</td><td>10</td><td>if |--not-a-com</td><td></td></tr>
<tr><td>9</td><td>10</td><td>if |--not-a-com</td><td></td></tr>
<tr><td>9</td><td>10</td><td>if |--not-a-com</td><td></td></tr>
<tr><td>9</td><td>10</td><td>if |--not-a-com</td><td></td></tr>
<tr><td>9</td><td>0</td><td>if |--not-a-com</td><td>error, illegal token ‘-’; resume</td></tr>
<tr><td>0</td><td>1</td><td>if -not-a-com</td><td></td></tr>
<tr><td>9</td><td>9</td><td>if -|-not-a-com</td><td></td></tr>
<tr><td>9</td><td>0</td><td>if -|-not-a-com</td><td>error, illegal token ‘-’; resume</td></tr>
</table>

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

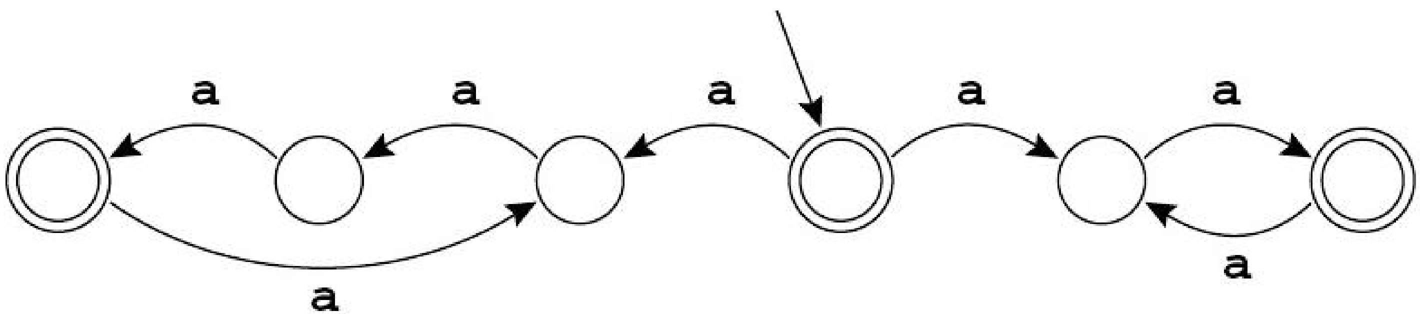
2.4 Nondeterministic Finite Automata

2.4 Nondeterministic Finite Automata

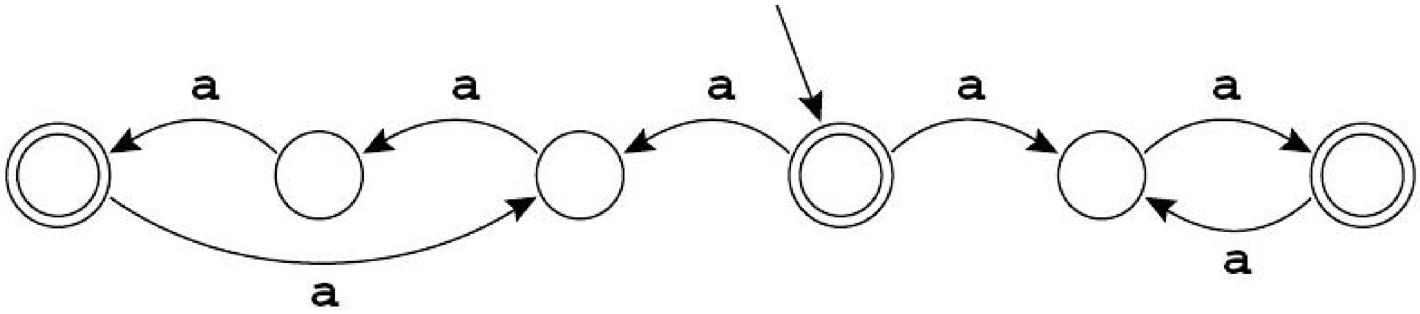
A nondeterministic finite automaton (NFA):

- Have to choose one from the edges (- labeled with the same symbol -) to follow out of a state
- Have special edges labeled with ∈ (the Greek letter epsilon) without eating any symbol from the input.

Here is an example of an NFA:



2.4 Nondeterministic Finite Automata



The language recognized by this NFA:

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

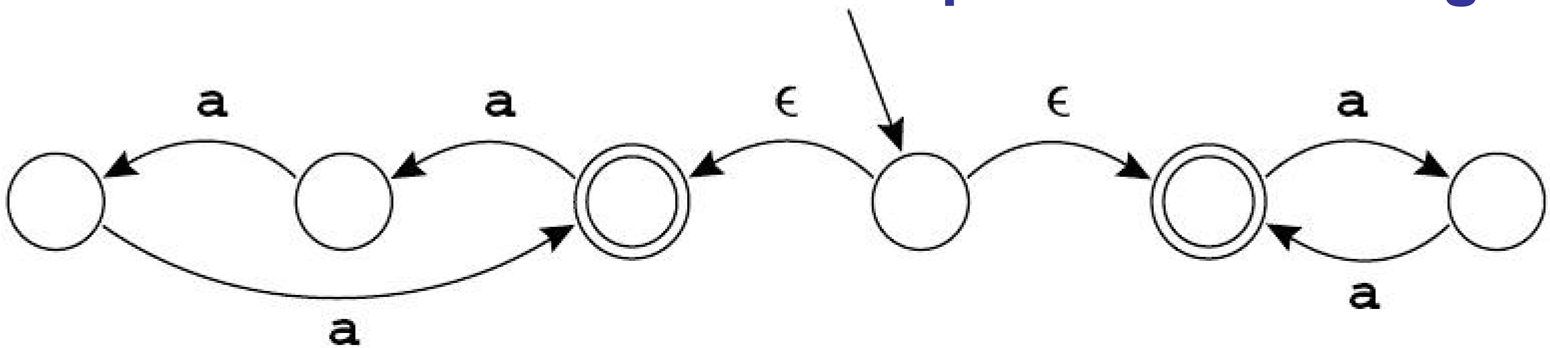
How the NFA works?

On character a, the automaton can move either right or left

- **If left is chosen -- strings’ length is a multiple of three**
- **If right is chosen – the length of strings is even**

2.4 Nondeterministic Finite Automata

Here is another NFA that accepts the same language:



edges labeled with ϵ may be taken without using up a symbol from the input.

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

2.4 Nondeterministic Finite Automata

Why NFA?

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

2.4 Nondeterministic Finite Automata

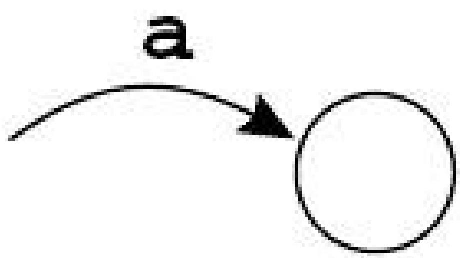
CONVERTING A REGULAR EXPRESSION TO AN NFA?

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

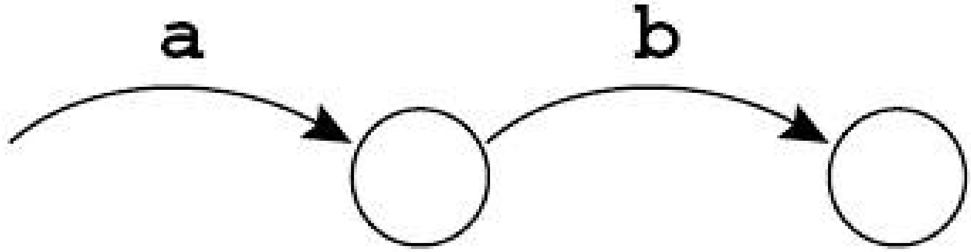


2.4 Nondeterministic Finite Automata

The regular expression ab:

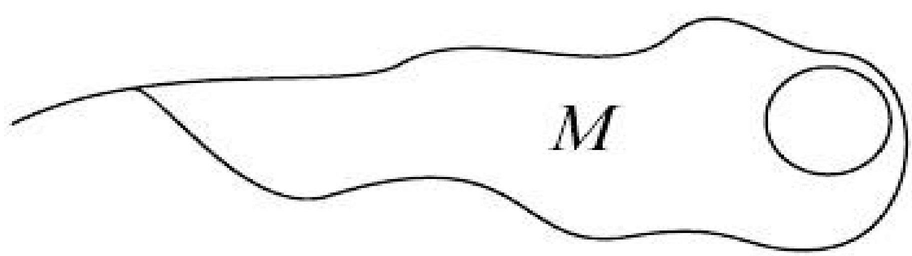
Combining the two NFAs

Hooking the head of a to the tail of b



In general, any regular expression *M*

Some NFA with a tail and head:



2.4 Nondeterministic Finite Automata

The rules for translating regular expressions to nondeterministic automata

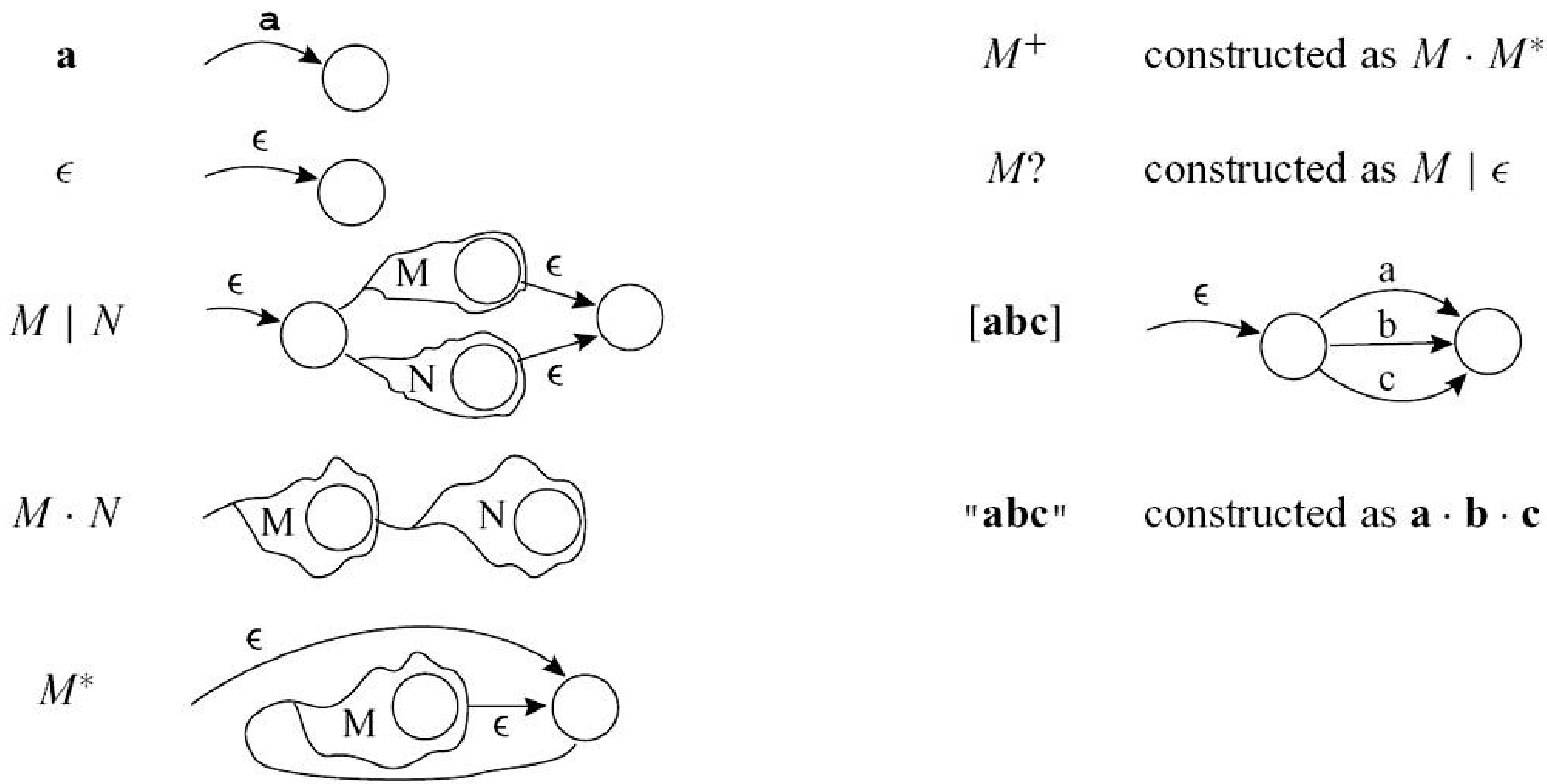


Figure 2.6: Translation of regular expressions to NFAs.

2.4 Nondeterministic Finite Automata

The result for the tokens IF, ID, NUM, and error
- after some merging of equivalent NFA states

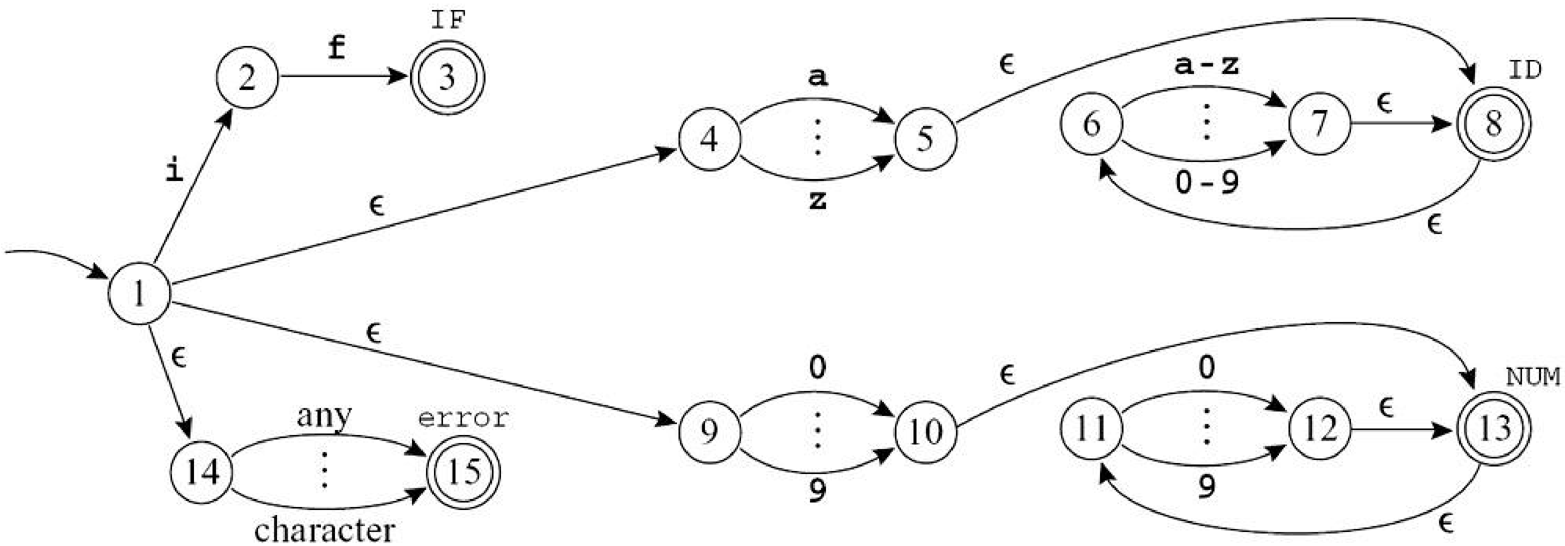


Figure 2.7: Four regular expressions translated to an NFA

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

2.4 Nondeterministic Finite Automata

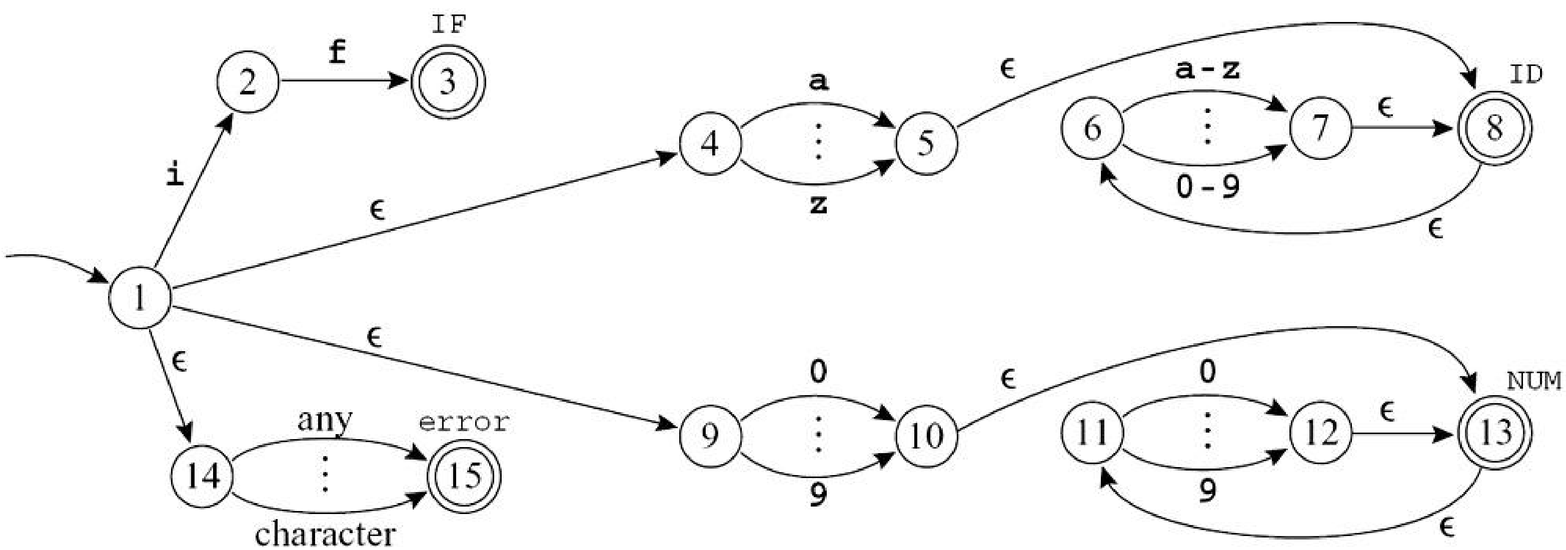
THE REASON FOR CONVERTING AN NFA TO A DFA

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

2.4 Nondeterministic Finite Automata

Simulating the NFA of Figure 2.7 on the string "in" 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, we compute the ∈ -*closure* of {1}**
- **No other states reachable without eating the first character**



2.4 Nondeterministic Finite Automata

Making the transition on the character i

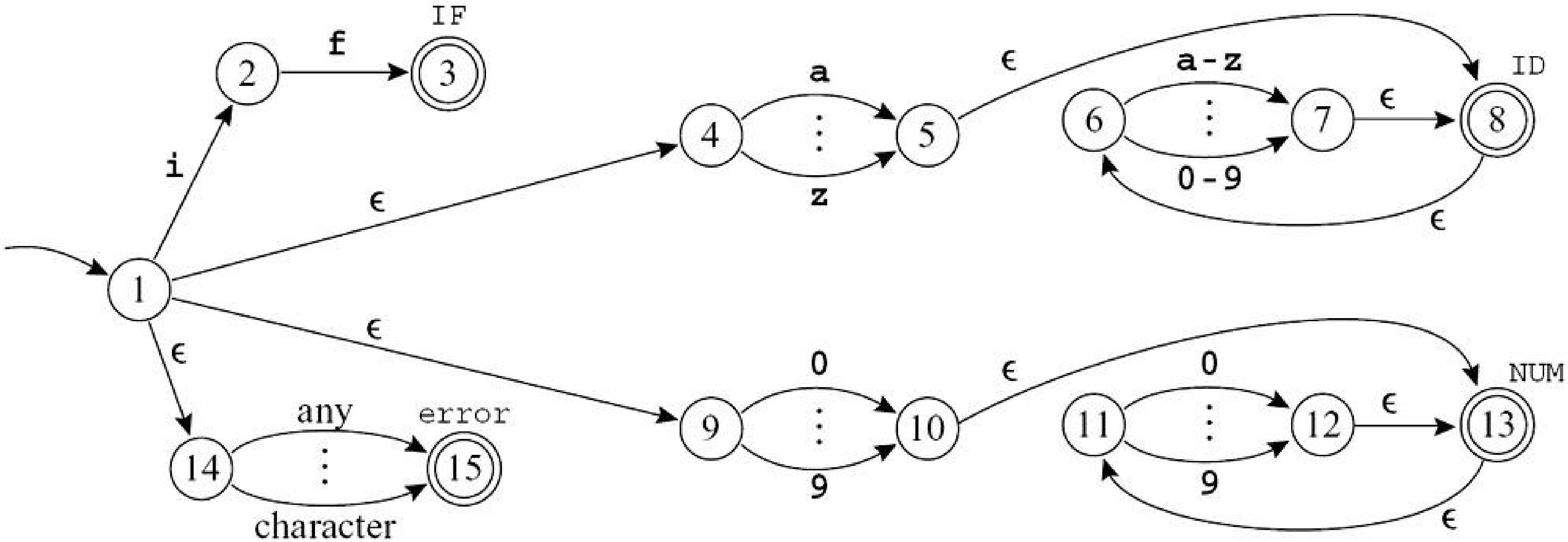
- **From state 1 to reach 2; from 4 to 5, from 9 to nowhere, and from 14 to 15**

  So the set {2, 5, 15}.

- **Again compute the ∈ -closure:**

  From 5 there is an ∈ -transition to 8 and From 8 to 6

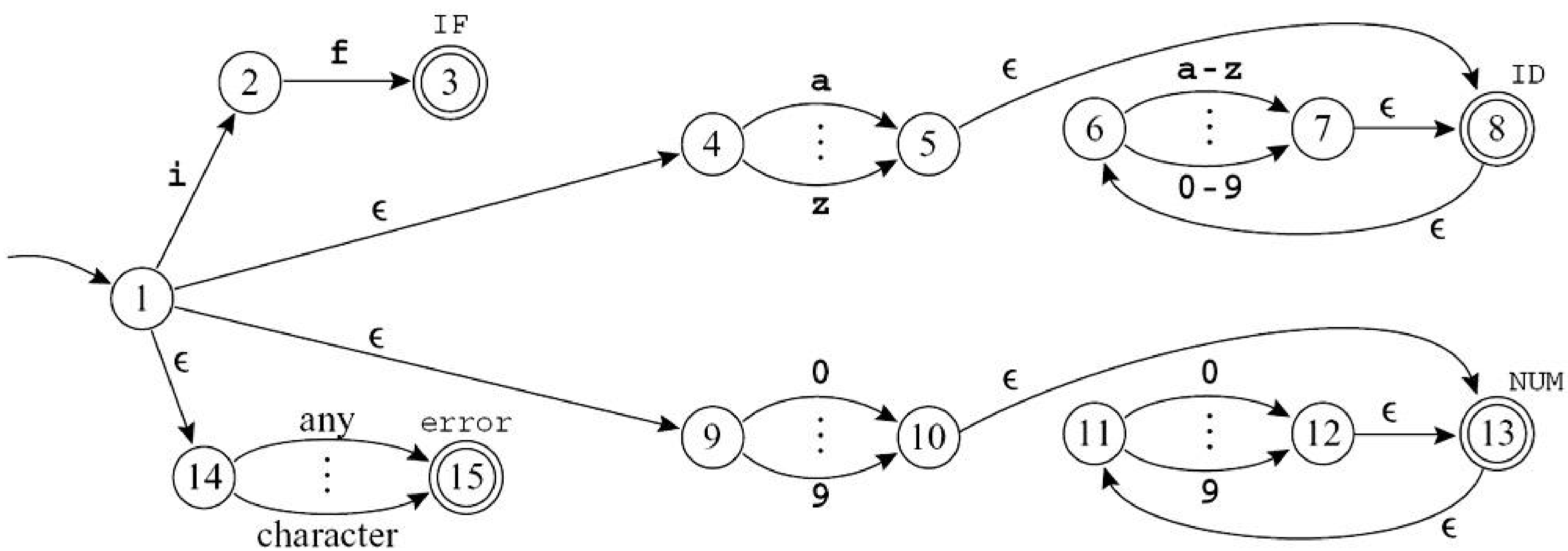
  So the NFA in one of the states {2, 5, 6, 8, 15}.



2.4 Nondeterministic Finite Automata

On the character n

- **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}.**



2.4 Nondeterministic Finite Automata

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*

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

2.4 Nondeterministic Finite Automata

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)**

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

2.4 Nondeterministic Finite Automata

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

$$
\begin{array}{l}
\text{states}[0] \leftarrow \{\}; \qquad \text{states}[1] \leftarrow \textbf{closure}(\{s_1\}) \\
p \leftarrow 1; \qquad j \leftarrow 0 \\
\textbf{while } j \le p \\
\quad \textbf{foreach } c \in \Sigma \\
\qquad e \leftarrow \textbf{DFAedge}(\text{states}[j], c) \\
\qquad \textbf{if } e = \text{states}[i] \text{ for some } i \le p \\
\qquad\quad \textbf{then } \text{trans}[j, c] \leftarrow i \\
\qquad\quad \textbf{else } p \leftarrow p + 1 \\
\qquad\qquad \text{states}[p] \leftarrow e \\
\qquad\qquad \text{trans}[j, c] \leftarrow p \\
\quad j \leftarrow j + 1
\end{array}
$$

Not visit unreachable states of the DFA

- In principle the DFA has 2*n* states.
- 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

2.4 Nondeterministic Finite Automata

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

- **we must 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.**

2.4 Nondeterministic Finite Automata

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

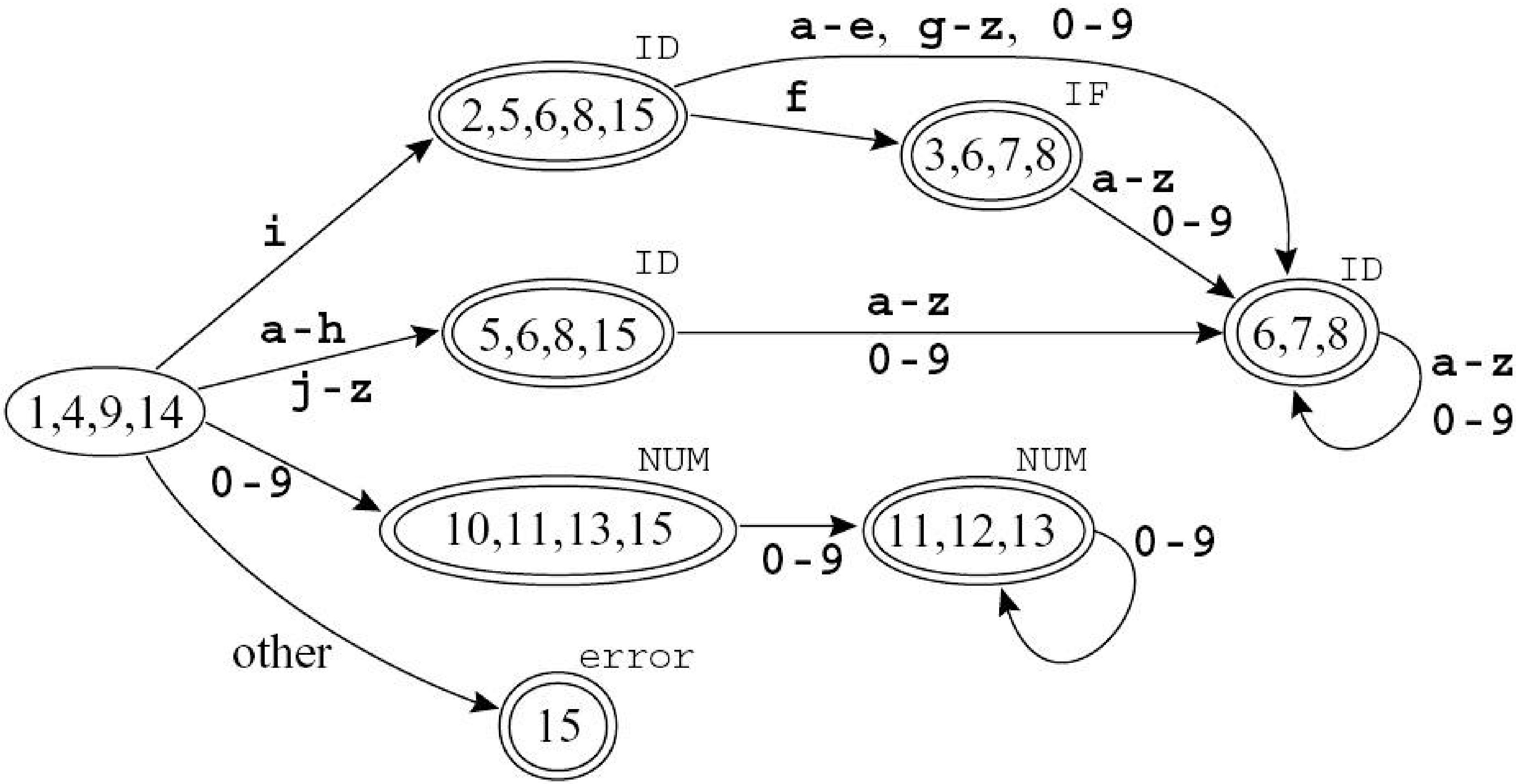


Figure 2.8: NFA converted to DFA

2.4 Nondeterministic Finite Automata

This automaton is suboptimal

It is not the smallest one that recognizes the same language

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.

equivalent:

the states labeled {5,6,8,15} and {6,7,8}

the states labeled{10,11,13,15} and {11,12,13}

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.

2.4 Nondeterministic Finite Automata

minimize the DFA

How can we find equivalent states?

***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.

(1) It begins with the most optimistic assumption possible: it creates two sets, one consisting of all the final states and the other consisting of all the non-final states.

2.4 Nondeterministic Finite Automata

minimize the DFA

(2) Given this partition of the states of the original DFA, consider the transitions on each character *a* of the alphabet.

if there are two states *s* and *t* in a set that have transitions on *a* that land in different sets,

character *a* distinguishes the states *s* and *t*.

(3) We must also consider error transitions to an error state that is non-final state.

If there are states *s* and *t* in a set such that *s* has an a transition to another set, while *t* has no a transition at all (i.e., an error transition), then a distinguishes *s* and *t.*

2.4 Nondeterministic Finite Automata

minimize the DFA

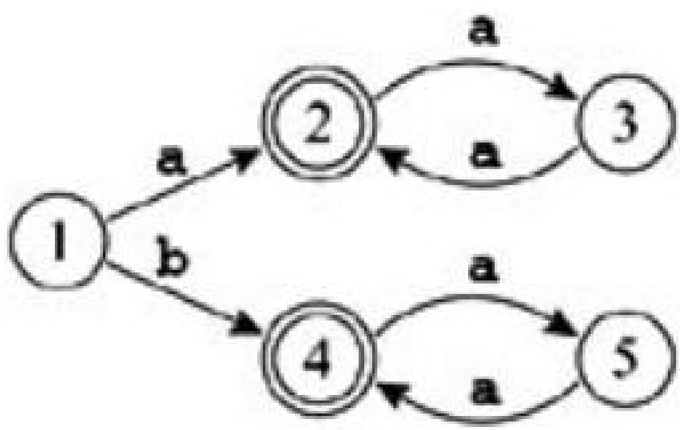
(4）If any further sets are split, we must return and repeat the process from the beginning.

This process continues until

- all sets contain only one element (in which case, we have shown the original DFA to be minimal)
- until no further splitting of sets occurs.

2.4 Nondeterministic Finite Automata

minimize the DFA



	states
The first process	{ 1,3,5} , { 2,4 }
The second process	{ 1} , { 3,5} , {2,4}

2.4 Nondeterministic Finite Automata

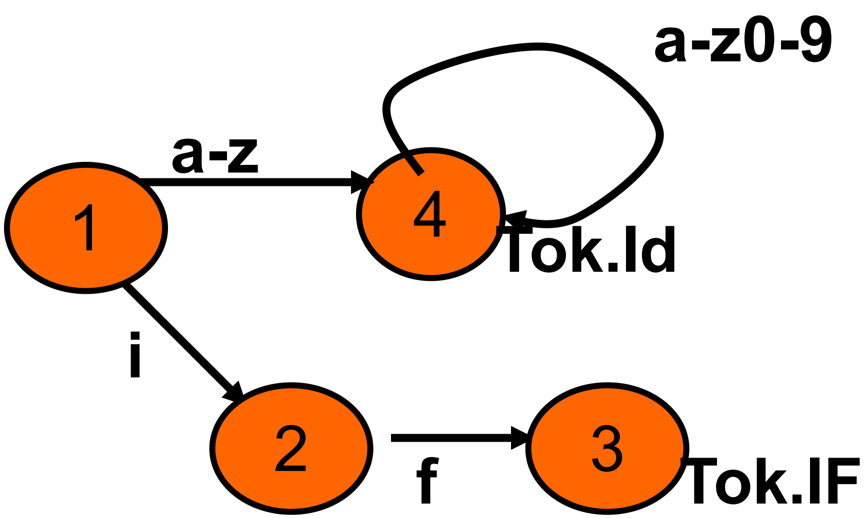
RE ==> NFA(NDFA) ==> DFA

Lex rules:

if => (Tok.IF)

[a-z][a-z0-9]* => (Tok.Id)

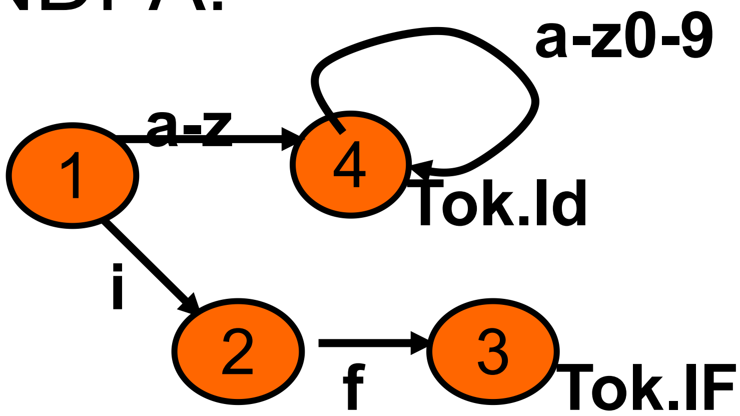
NDFA:



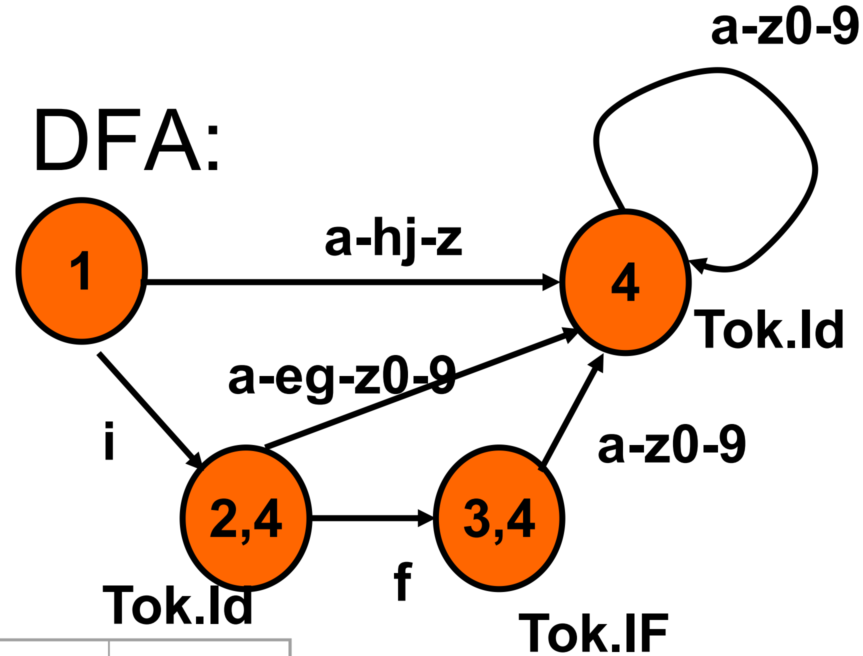
2.4 Nondeterministic Finite Automata

RE ==> NDFA ==> DFA

NDFA:



DFA:

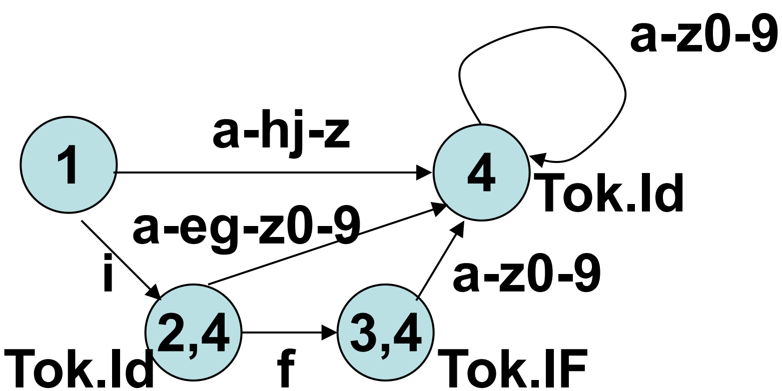


State	closure	i	a - e gh j-z	f	0-9
{1}	{1}	{2,4}	{4}	{4}	
{4}	{4}	{4}	{4}	{4}	{4}
{2,4}	{2,4}	{4}	{4}	{3,4}	{4}
{3,4}	{3,4}	{4}	{4}	{4}	{4}

2.4 Nondeterministic Finite Automata

Table-driven algorithm

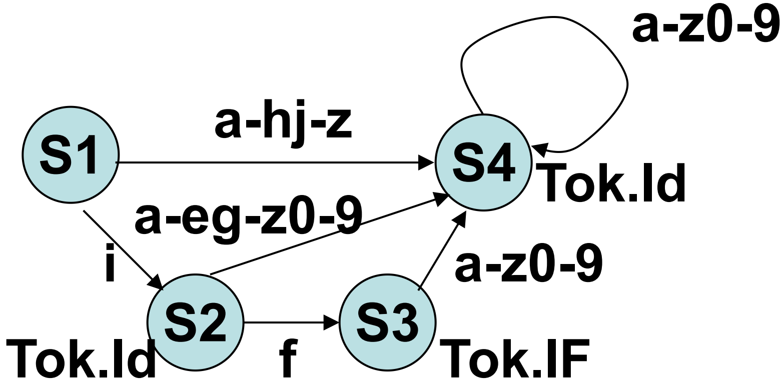
- **NDFA:**



2.4 Nondeterministic Finite Automata

Table-driven algorithm

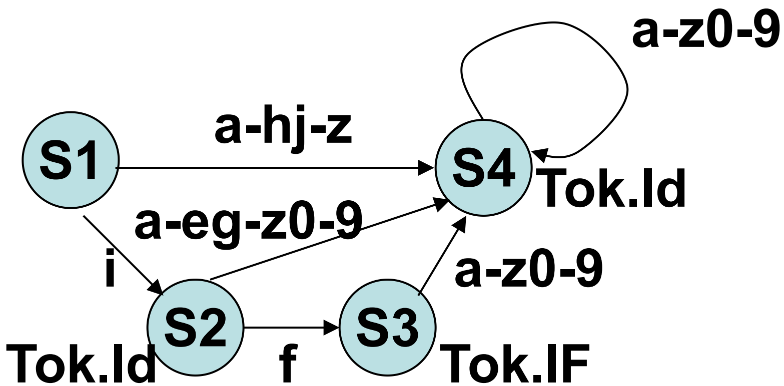
- NDFA (states conveniently renamed):



2.4 Nondeterministic Finite Automata

Table-driven algorithm

- DFA:



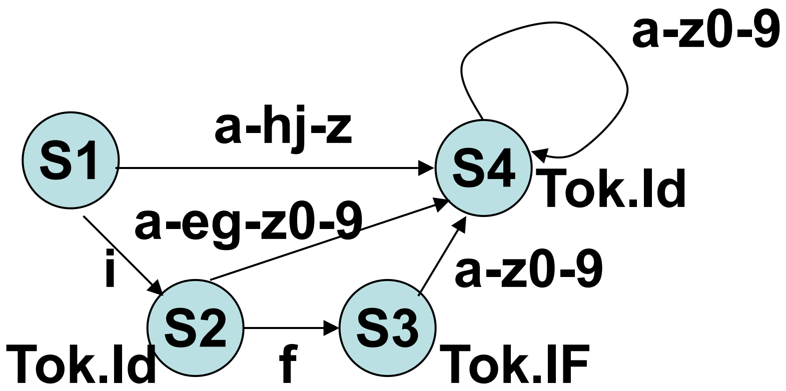
Transition Table:

	S1	S2	S3	S4
a	S4	S4	S4	S4
b	S4	S4	S4	S4
...				
i	S2	S4	S4	S4
...				

2.4 Nondeterministic Finite Automata

Table-driven algorithm

- DFA:



Transition Table:

	S1	S2	S3	S4
a	S4	S4	S4	S4
b	S4	S4	S4	S4
…				
i	S2	S4	S4	S4
…				

Final State Table:

S1	S2	S3	S4
-	Tok.Id	Tok.IF	Tok.Id

作业：

第二章　　**2.1, 2.2，2.5(a,b)，2.6**

2.5 Lex: A Lexical Analyzer Generator

2.5 Lex: A Lexical Analyzer Generator

- **Implementation Options:**
  1. Write a Lexer from scratch
     - Boring, error-prone and too much work

2.5 Lex: A Lexical Analyzer Generator

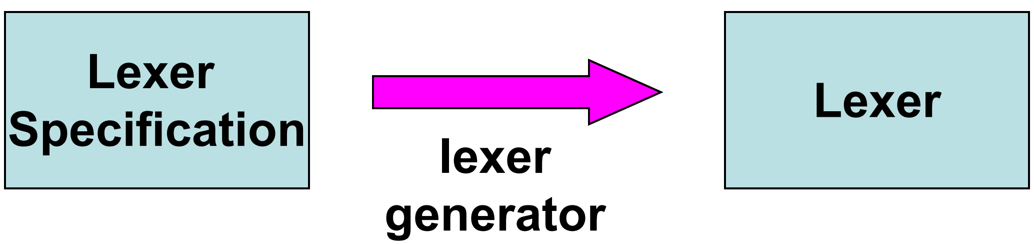
- **Implementation Options:**

1. **Write a Lexer from scratch**
   - Boring, error-prone and too much work
2. **Use a Lexer Generator**
   - Quick and easy. Good for lazy compiler writers.

Lexer
Specification

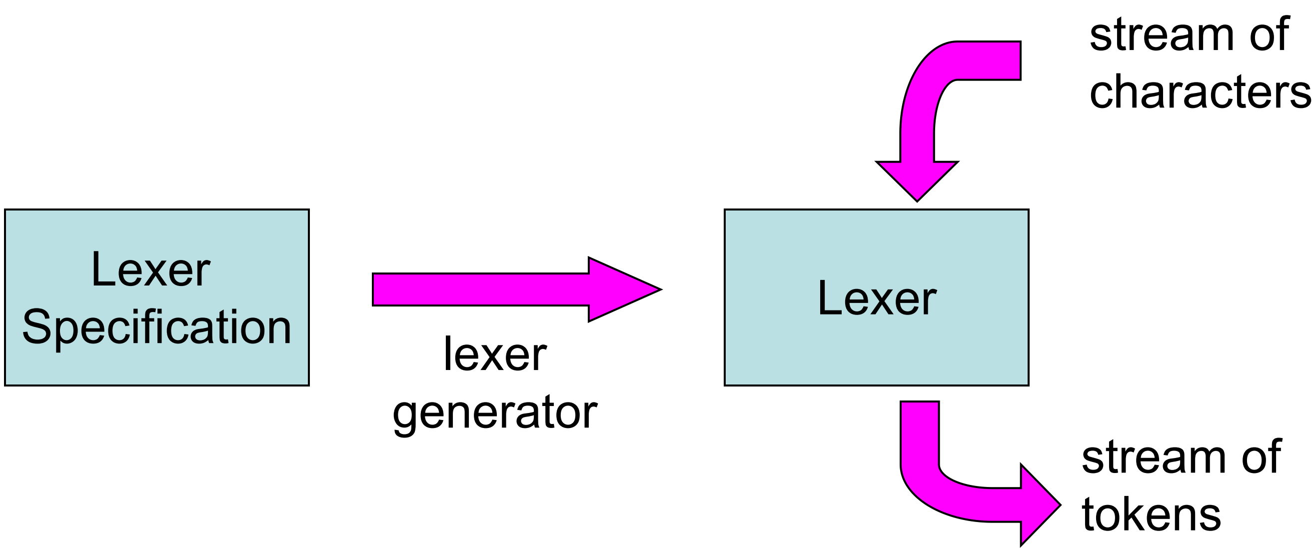
2.5 Lex: A Lexical Analyzer Generator

- **Implementation Options:**
  1. **Write a Lexer from scratch**
     - Boring, error-prone and too much work
  2. **Use a Lexer Generator**
     - Quick and easy. Good for lazy compiler writers.



2.5 Lex: A Lexical Analyzer Generator

- **Implementation Options:**
  1. **Write a Lexer from scratch**
     – Boring, error-prone and too much work
  2. **Use a Lexer Generator**
     – Quick and easy. Good for lazy compiler writers.



2.5 Lex: A Lexical Analyzer Generator

- **How do we specify the lexer?**
  - Develop another language
  - We'll use a language involving regular expressions to specify tokens

- **What is a lexer generator?**
  - Another compiler ....

2.5 Lex: A Lexical Analyzer Generator

- **we will use the Lex Analyzer generator to generate a C program from.**
- **The most popular version of Lex is called flex {for Fast Lex).**

It is distributed as part of the **Gnu compiler package** produced by the Free Software Foundation, and is also freely available at many Internet sites.

2.5 Lex: A Lexical Analyzer Generator

Lex is a program:

- **Input : a text file containing regular expressions, together with the actions to be taken when each expression is matched**
- **Output : Contains C source code defining a procedure *yylex* that is *a* table-driven implementation of a DFA corresponding to the regular expressions of the input file.**

2.5 Lex: A Lexical Analyzer Generator

- The format of a Lex input file

```
{ definitions }
%%
{ rules }
%%
{ auxiliary routines}
```

Lex :A Lexical Analyzer Generator

1、*The first section :definitions*

The definition section occurs before the first %%.

1) **any C code that must be inserted external to any function should appear in this section between the delimiters %{and %},** ***(Note the order of these characters!)***
2) **names for regular expressions must also be defined in this section.**

2.5 Lex: A Lexical Analyzer Generator

2、 The second section : rules

These consist of a sequence of regular expressions followed by the C code that is to be executed when the corresponding regular expression is matched.

3、 The third section: auxiliary routines

Routines are called in the second section and not defined elsewhere.

2.5 Lex: A Lexical Analyzer Generator

1、**Lex allows the matching of single characters, or strings of characters, simply by writing the characters in sequence.**

- Quotes can also be written around characters that are not metacharacters, where they have no effect. For example : **if and “if” are same meaning.**
- to match the character sequence (* we would have to write \(* or " (* ".
- a special meaning: \n matches a newline and \t matches a tab.

2、**Complementary sets—that is, sets that do *not* contain certain characters**

the carat ^ as the first character inside the brackets.

example: [^0-9abc] means any character that is not a digit and is not one of the letters *a, b,* or *c.*

2.5 Lex: A Lexical Analyzer Generator

3、 One curious feature in Lex is that inside square brackets (representing a character class), most of the metacharacters lose their special status and do not need to be quoted.

(1) we could have written **[-+]** instead of (**"+" | "-"**) . (but not **[+-]** because of the metacharacter use of - to express a range of characters).

(2) **[."?]** means any of the three characters period, quotation mark, or question mark (all three of these characters have lost their metacharacter meaning inside the brackets).

(3) ***Some characters, however, are still metacharacters even inside the square brackets.*** we must precede the character by a backslash (quotes cannot be used as they have lost their metacharacter meaning). Thus, [**^** \ \] means either of the actual characters **^** or \.

2.5 Lex: A Lexical Analyzer Generator

4、A further important metacharacter convention in Lex is the use of curly brackets to denote names of regular expressions.

these names can be used in other regular expressions as long as there are **no recursive references.**

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

2.5 Lex: A Lexical Analyzer Generator

metacharacter conventions in lex

Pattern	Meaning
a	the character a
“a”	the character a, even if a is a metacharacter
\a	the character a when a is a metacharacter
a*	zero or more repetitions of a
a+	one or more repetitions of a
a?	an optional a
a|b	a or b
(a)	a itself
[abc]	any of the characters a, b, or c .
[a-d]	any of the characters a. b, c or d
[^ab]	any character except a or b
.	any character except a newline
{xxx}	the regular expression that the name xxx represents

2.5 Lex: A Lexical Analyzer Generator

yytext: string matched by the regular expression

yyin: Lex input file (default: stdin)

yyout: Lex output file (default: stdout)

yyleng: the length of the matched string

yylineno: the line number

charPos: keep track of the position of each token.

yylval: a union of the different types of semantic values.

2.5 Lex: A Lexical Analyzer Generator

example

```
%{
    /*  a Lex program that changes all numbers from decimal
   to hexadecimal notation,  printing a summary statistic to
   stdeer
   */
#include  <stdlib.h>
#include  <stdio.h>
int  count=0;
 %}
  digit   [0-9]
  number   {digit}+
 %%
 {number}  { int  n = atoi(yytext);
                printf("%x",n);
               if (n > 9) count++;}
%%
 main( )
{ yylex();
 fprintf(stderr, "number of replacements = %d", count);
 return 0;
}
```

2.5 Lex: A Lexical Analyzer Generator

Implementing Lex

- **By compiling, of course:**
  - **convert REs into non-deterministic finite automata**
  - **convert non-deterministic finite automata into deterministic finite automata**
  - **convert deterministic finite automata into a blazingly fast *table-driven algorithm***

Summary

- **A Lexer:**
  - **input: stream of characters**
  - **output: stream of tokens**
- **Writing lexers by hand is boring, so we use a lexer generator: lex**
  - **lexer generators work by converting REs through automata theory to efficient table-driven algorithms.**

以下实验二选一：

实验一、利用LEX计算文本文件的字符数等 （5分）

实验目的：了解LEX的基本编程方法。

实验要求：编写一个LEX输入文件，使之生成可计算文本文件的字符、单词和行数且能报告这些数字的程序。单词为不带标点或空格的字母和/数字的序列。标点和空白格不计算为单词。

提交截止时间：春学期的第三周的周日22：00之前

实验二、利用LEX进行字母的大小写转换 （5分）

实验目的：了解LEX的基本编程方法

实验要求：编写一个LEX输入文件，使之可生成将程序(语言事先定义）注释之外的所有关键字（保留字）均大写的程序。该LEX生成的程序要能够对源程序进行分析，将不是大写的关键字均转换为大写。

提交截止时间：春学期的第三周的周日22：00之前

Expriement Evironment

Linux环境下的编译和运行

- **Linux 2.6以上版本**
- **GCC3.4以上版本**
- **Bison 2.2以上版本**
- **Flex 2.5.33以上版本**
- **发行版可以采用Ubuntu, Gentoo, Fedora Core等。**

Expriement Evironment

- Windows环境下的编译和运行
  - Visual Studio 6.0
  - Masm 6.0以上版本
  - ParseGenerator 4.0 （Lex和Yacc的集成开发包）