

Compiler Construction

Chapter 2: Scanning

Slides modified from Loudon Book and Dr. Scherger

Contents

- ▶ Scanning
- ▶ Tokens, Patterns, and Lexemes
 - ▶ Lexical Errors
- ▶ Scope
- ▶ Symbol Tables and Hash Tables
- ▶ Tokens, Languages, and Regular Definitions
- ▶ Recognition of Tokens
- ▶ Finite Automata
- ▶ NFA \rightarrow DFA
- ▶ Regular Expression \rightarrow NFA

Scanning, or Lexical Analysis.

- ▶ A scanner is an implementation of a deterministic finite automaton (DFA, finite state machine).
- ▶ That is, looping but no recursion is allowed.

The Role of the Lexical Analyzer

- ▶ The lexical analyzer or scanner is the first phase of a compiler:
 - ▶ Its main task is to read the input characters and produce a sequence of tokens for the syntax analyzer.
 - ▶ More compact representation of input and easier to deal with later
- ▶ All Scanners do basically the same thing, only recognize different tokens
- ▶ It is usually implemented as a subroutine which the syntax analyzer calls whenever it wants the next token:
 - ▶ The lexical analyzer returns a token and then waits for the next call.
- ▶ A secondary task of the lexical analyzer is to ignore white space (spaces, tabs, and newline characters in the source) and comments.
- ▶ Another task might be to keep track of line numbers so meaningful error messages can be generated.

Tokens, Patterns, and Lexemes

- ▶ The terms token, pattern, and lexeme have specific meanings.
 - ▶ The lexical analyzer returns a token of a certain type to the parser whenever it sees a sequence of input characters, a lexeme, that matches the pattern for that type of token.

- ▶ An Example of some lexemes

- ▶ The pattern for the RELOP token contains six different lexemes (=, <>, <, <=, >=, and >)
 - ▶ Many tokens such as IFTOK, THENTOK have single-lexeme patterns.
 - ▶ The patterns for the ID and NUM tokens are described by regular expressions .
 - ▶ Informally, a lexeme for the ID token must start with a letter followed by zero or more letters and digits.

Token	Lexeme
IFTOK	if
THENTOK	then
ELSETOK	else
RELOP	= <> < <= >= >
ASSIGNOP	:=
ID	letter (letter digit)*
NUM	digit*(. digit+)?
SEMITOK	;

Tokens, Patterns, and Lexemes

- Lexical analysis is complicated in some languages. FORTRAN, for example, allows white space inside of lexemes. The FORTRAN statement:

DO 5 I = 1.25

- is an assignment statement with three tokens:

ID	ASSIGNOP	NUM
DO5I	=	1.25

- but the FORTRAN statement:

DO 5 I = 1,25

- is a DO-statement with seven tokens:

DOTOK	NUM	ID	ASSIGNOP	NUM	COMMA	NUM
DO	5	I	=	1	,	25

- Before the lexical analyzer can produce the first token it must look ahead to see if there is a dot or a comma in this statement.

Tokens, Patterns, and Lexemes

- ▶ Most languages have keywords, strings of letters that have pre-defined meanings so they can not be used as identifiers.
 - ▶ For example, keywords in Pascal are: if, then, else, etc.
 - ▶ The PL/I language does not reserve its keywords so distinguishing between a keyword and an identifier is very complicated.
 - ▶ For example,

```
IF THEN THEN
    THEN = ELSE;
ELSE
    ELSE = THEN;
```

is a legal PL/I statement.

Typical Token Classes

- ▶ Reserved words (keywords):
if while do ...
- ▶ Identifiers:
interest x23 __z_over
- ▶ Literals (constants):
42 3.14159 “Hello”
- ▶ Special symbols (operators):
+ += ;, ==
- ▶ White space: blanks, tabs, comments, newlines, other control characters

TINY Tokens

Reserved words	Special symbols	Other
<code>if</code> <code>then</code> <code>else</code> <code>end</code> <code>repeat</code> <code>until</code> <code>read</code> <code>write</code>	<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>=</code> <code><</code> <code>(</code> <code>)</code> <code>;</code> <code>:=</code>	<i>number</i> (1 or more digits) <i>identifier</i> (1 or more letters)

Attributes for Tokens

- ▶ The pattern for the RELOP token in Pascal contains six lexemes:
 - ▶ The syntax analyzer only needs to know that the token is a relational operator
 - ▶ But later phases will have to know which of the six relational operators is specified
 - ▶ So the lexical analyzer must return this attribute with the RELOP token.

Attributes for Tokens

- ▶ The lexical analyzer must also return attributes with other multi-lexeme tokens:
 - ▶ The UNARYOP, ADDOP, MULOP, BCONST, ID, and NUM tokens.
 - ▶ A pointer to the symbol table entry is all that is required for an ID token.
 - ▶ An easy way to handle a NUM token is to store its lexeme in the symbol table and return a pointer to that entry.
 - ▶ One way to handle the other multi-lexeme tokens is to pre-store all their lexemes in the symbol table so the lexical analyzer can return a pointer to a symbol table entry in all cases.

Lexical Errors

- ▶ A lexical error is a sequence of characters that does not match the pattern of any token.
- ▶ The most common causes of a lexical error are:
 - ▶ The addition of an extraneous character;
 - ▶ The removal of a character that should be present;
 - ▶ The replacement of a correct character with an incorrect character; and
 - ▶ The transposition of two characters.

Lexical Errors

- ▶ The theoretically best way of handling a lexical error is to find the closest character sequence that does match a pattern but this will take too much time and is never used in a practical compiler.
- ▶ A way to handle a lexical error is to store the bad character in the symbol table and return an ERROR token to the syntax analyzer with a pointer to the symbol table entry.

Symbol Tables

- ▶ The symbol table, sometimes called a dictionary, keeps track of information about the symbols (identifiers) that the source program is using:
 - ▶ Names of scalars, arrays, and functions, etc.
- ▶ That is it associates attributes with names
 - ▶ Attributes are a representation of the semantics of the names
 - ▶ What are useful attributes?
- ▶ Numbers can go in the symbol table - values must be stored somewhere:
 - ▶ They can also be stored in a separate table.
 - ▶ A language like C allows unions, so you can return the value instead of a STptr.
- ▶ Keywords like if, while, and real, can also go in the symbol table .
 - ▶ The keywords are pre-loaded into the symbol table before any of the source program is read so the first time the source program uses a keyword it will be assigned the proper token-type.
- ▶ The lexical analyzer can be simplified by also pre-loading the other lexemes like (,], :=, and <=, into the symbol table.

Symbol Tables - Attributes

- ▶ **Name**

- ▶ What do we store

- ▶ **Type**

- ▶ What do we store?
 - ▶ Think about C
 - ▶ Boolean, char, integer, real
 - ▶ Pointer, array, function
 - ▶ How could we encode this?



Symbol Tables – Encoding Attributes

► How could we encode this?

Boolean	00	pointer	01
Char	01	array	10
Integer	10	function	11
Real	11		

► So:

char foo	01
char *foo	0101
char foo()	1101
*(char foo())	011101



Data Structures of a Symbol Table

- ▶ A fixed size array?
- ▶ Unordered list
 - ▶ Simple
 - ▶ $O(1)$ insertion, but $O(n)$ search
- ▶ Ordered list
 - ▶ $O(\log(n))$ search, $O(n)$ insertion
- ▶ Binary search trees
 - ▶ $O(\log(n))$ search, $O(\log(n))$ insertion
 - ▶ Performance not guaranteed. Why?
 - ▶ Easy to use. coding can be difficult,
- ▶ Hash tables
 - ▶ Searching close to $O(1)$, quite common

Block-Structured Symbol Table

- ▶ *Name scopes* are units of code
 - ▶ can be nested
- ▶ line of code can be in multiple scopes
 - ▶ innermost scope is the *current scope*
- ▶ *Open scopes* enclose a line of code
- ▶ *Closed scopes* do not contain line of code
- ▶ Visibility rules
 - ▶ Only current & open scopes accessible
 - ▶ When scope closes all vars inaccessible
 - ▶ Innermost declaration used
 - ▶ New declaration only in current scope



Block-Structured Symbol Table

- ▶ Individual table for each scope
- ▶ Single symbol table



Block-Structured Symbol Table

- ▶ Individual table for each scope
 - ▶ Can maintain a scope stack, search each table in turn until find symbol
 - ▶ Push a new table when a scope is opened
 - ▶ Pop when the scope is closed
 - ▶ Can be destroyed in a one-pass, but not in multi
 - ▶ Searches can be slow
 - ▶ Can waste space (if using hash, or arrays)
- ▶ Single symbol table



Block-Structured Symbol Table

- ▶ Individual table for each scope
- ▶ Single symbol table
 - ▶ Use unique scope numbers
 - ▶ A symbol can appear >1 if different scope number
 - ▶ With a hash, always insert at front of list
 - ▶ Searches are easy, always get correct scope
 - ▶ Delete all entries for scope when scope is left
 - ▶ Visit all chains, but only as deep as different scope
 - ▶ BSPs will be slow to search and delete
 - ▶ Why?



Symbol Table: Other issues

A,R:record

A: Integer;

X: record

A: Real;

C: Boolean;

end;

end;

- ▶ Here field names are not unique within the scope, only within the record.
- ▶ Records become a mini-scope



Symbol Table: Other issues

- ▶ C static
- ▶ Exporting names outside of scope
- ▶ Separate compilation
- ▶ Hiding type
- ▶ Importing and nonimporting scopes
 - ▶ Does a scope see containing scopes?
- ▶ Implicit declarations
 - ▶ Labels, for loop indices
- ▶ Overloading
- ▶ Forward References



For Your Compiler

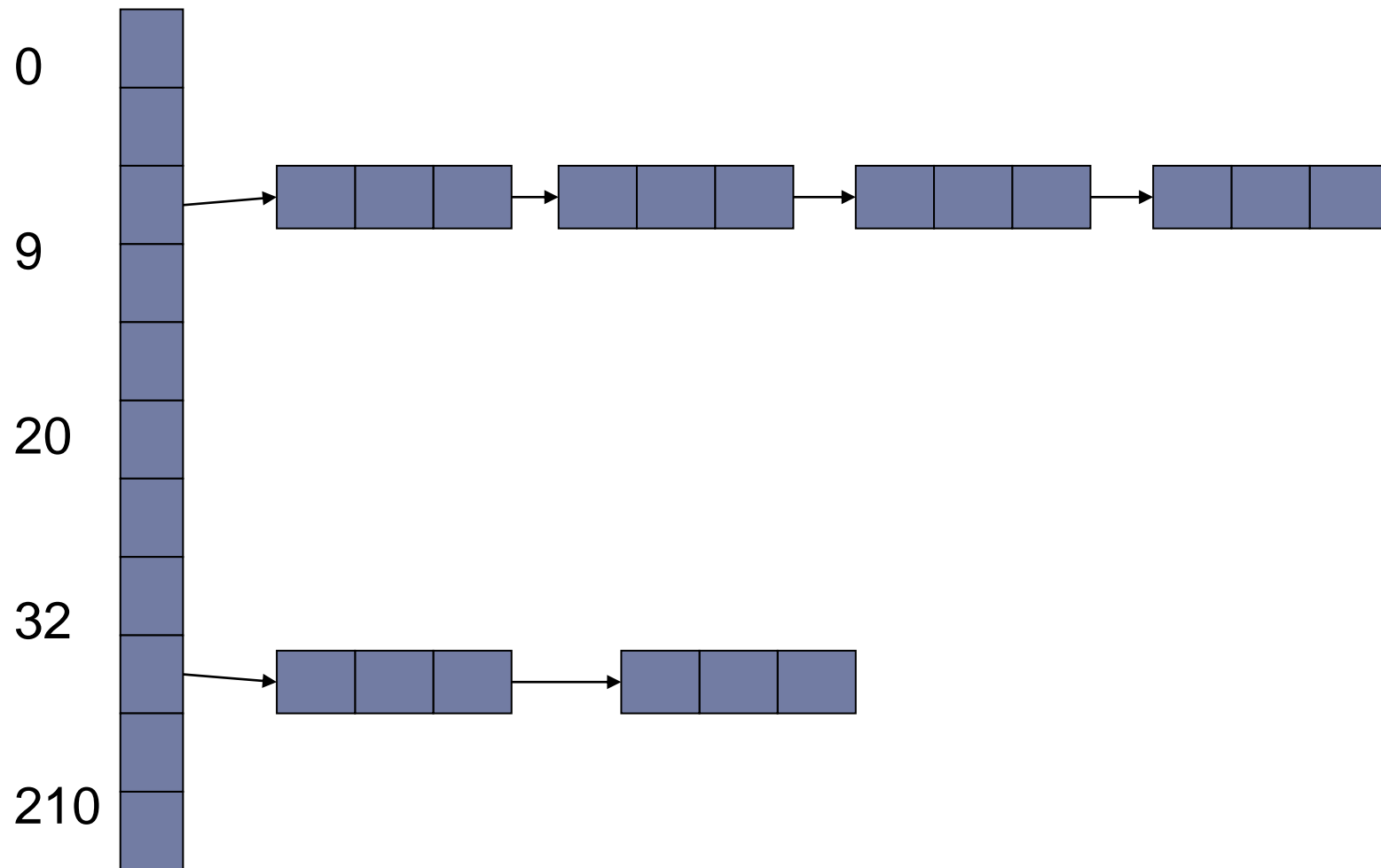
- ▶ Hash function for symbol table
- ▶ When do you add a symbol to the table?
- ▶ When do you access a symbol from the symbol table?
- ▶ Semantic checking
 - ▶ How will you check types of an expression?



Hash Tables

- ▶ Usually the symbol table is implemented with a hash function to steer each entry to one of a number of linear linked lists:
 - ▶ A good hash function will sprinkle the entries across many lists so every list is short enough to be searched efficiently.
 - ▶ The following figure illustrates a hash table, an array of 211 list headers with indices ranging from 0 to 210.
 - ▶ Each list header points to the start of a linked list of symbol table entries (a list header is NULL if its list is empty.)
 - ▶ To find an entry for a given lexeme:
 - ▶ the lexeme is sent to a hash function which returns an integer in the range of 0 to 210;
 - that integer is used as an index to select one of the headers in the hash table;
 - and that header is used to search the corresponding linked list sequentially for an entry with the given lexeme.

Hash Tables



Hash Functions

- ▶ Four hash functions have good performance
 - ▶ Hashpjlw
 - ▶ X65599
 - ▶ Quad
 - ▶ xl6

Hash Functions - hashpjw

```
#define PRIME 211
#define EOS '\\0'
int hashpjw( char *s )
{
    char *p;
    unsigned h=0, g;

    for( p=s; *p != EOS; p=p+1 )
    {
        h = (h<<4) + (*p);
        if (g = h&0xf0000000)
        {
            h = h ^ (g >> 24);
            h = h ^ g;
        }
    }
    return h % PRIME;
}
```

Hash Functions – Other Functions

- ▶ `x65599` initializes `h` to 0, performs $h = (65599 * h + (\text{ASCII-value of character})) \bmod 211$ for each character in the lexeme, and then returns `h`
- ▶ `x16` is similar to `x65599` except that the constant multiplier is 16 instead of 65599
- ▶ `quad` splits the lexeme into 4-character groups, adds the groups together as though they were integers, and returns their sum modulo 211.
- ▶ Regardless of the hash function, hashing seems to work much better when the size of the hash table is a prime number like 211.

Storing Lexemes

- ▶ Most source languages do not impose any limit on the length of a symbol name.
 - ▶ Most programmers use only short symbol names (e.g., 10 characters or less) but the compiler must also be capable of handling a very long symbol name with 100 characters or more.
- ▶ Another method is that Lexemes are stored serially as they are first encountered in a large array of a few thousand characters with each lexeme followed by an end-of-string character.
 - ▶ The lexeme field of each symbol table entry is only a short 2-byte integer to hold the starting location of the lexeme in the large array.
 - ▶ This method works as long as the sum of all lexeme lengths (including their end-of-string characters) does not exceed the length of the large array.
- ▶ Alternatively, if the compiler is written in C or C++, each lexeme can be stored in a dynamic storage item that is just large enough to hold it and its end-of-string character.
 - ▶ Each symbol table entry is stored in another dynamic storage item of fixed size containing a pointer to the lexeme item.

The Scope of Symbols

- ▶ The scope of symbols is best illustrated with an example like the Pascal source program:

```
program main(input, output);  
var x, y : integer;  
var z : real;  
  
function foo(a : real) : real;  
  var x : real;  
  begin (* body of foo function *)  
    x := 2.0 * a;  
    y := y + 1;  
    foo := x + 5.0  
  end;  
  
Begin (* body of main program *)  
  y := 0 ;  
  x := 0 ;  
  z := foo(6.0)  
end.
```

The Scope of Symbols

- ▶ The scope of symbols is best illustrated with an example like the Pascal source program:
- ▶ Function foo has two internal real variables, a and x , and a real return value, foo .
- ▶ A global integer and a real variable local to foo share the same name, x :
 - ▶ The scope rules for Pascal tell us that x in the body of foo denotes the local real variable while x in the body of the main program denotes the global integer.
 - ▶ Both the global integer and the local real must have an entry in the symbol table and the two entries will have the same lexeme:
 - ▶ The symbol table search operation must be designed to always return a pointer to the correct entry.

```
program main(input, output);  
var x, y : integer;  
var z : real;
```

```
function foo(a : real) : real;  
    var x : real;  
    begin (* body of foo function *)  
        x := 2.0 * a;  
        y := y + 1;  
        foo := x + 5.0  
    end;
```

```
Begin (* body of main program *)  
    y := 0 ;  
    x := 0 ;  
    z := foo(6.0)  
end.
```


The Scope of Symbols

- ▶ When compiling the body of a function the search operation should return the local variable instead of the global variable whenever there are duplicated lexemes.
- ▶ When searching a linked list the FIND function will return the entry closest to the start of the list and globals are inserted before local variables so the INSERT function should always insert each entry at the start of the list.

The Scope of Symbols

- ▶ When the body of a function has been completely compiled all variables local to that function should be flagged as unsearchable.
 - ▶ One way to do this is to use an integer Scope field in each entry to keep track of its scope:
 - ▶ The scope field will be 0 for a global, positive for a local variable, and negative for an entry that is unsearchable.
- ▶ The syntax analyzer maintains the current scope of the compilation in a global integer, CurrentScope , that the INSERT function copies into the Scope field of each entry it inserts.
 - ▶ The syntax analyzer also negates the Scope field of the local entries whenever a function has been completely compiled.
 - ▶ The FIND function should ignore any entry with a negative Scope field.

Specification of Tokens

- ▶ An alphabet is a finite set of symbols.
 - ▶ Typical examples of symbols are letters and characters.
 - ▶ The set $\{0, 1\}$ is the binary alphabet .
 - ▶ ASCII and EBCDIC are two examples of computer alphabets.

Strings

- ▶ A string over some alphabet is a sequence of symbols drawn from that alphabet.
- ▶ For example, *banana* is a sequence of six symbols drawn from the ASCII alphabet.
- ▶ The empty string, denoted by epsilon, ε , is a special string with zero symbols.
- ▶ If x and y are two strings then the concatenation of x and y , written xy , is the string formed by appending y after x .
 - ▶ For example if $x = \mathbf{dog}$ and $y = \mathbf{house}$ then $xy = \mathbf{doghouse}$.
- ▶ String exponentiation concatenates a string with itself a given number of times:
 - ▶ $x^2 = xx$; $x^3 = xxx$; $x^4 = xxxx$; etc.
 - ▶ By definition x^0 is the empty string, ε , and $x^1 = x$.
 - ▶ For example, if $x = ba$ and $y = na$ then $xy^2 = banana$.

Languages

- ▶ A language is a set of strings over some fixed alphabet:
 - ▶ The set may contain a finite number or an infinite number of strings.
- ▶ If L and M are two languages then their union, denoted by $L \cup M$, is the language containing every string in L and every string in M (any string that is in both L and M appears only once in $L \cup M$).
- ▶ If L and M are two languages then their concatenation, denoted by LM , is the language containing every string in L concatenated with every string in M .
 - ▶ For example if $L = \{\text{dog, ba, na}\}$ and $M = \{\text{house, ba}\}$ then $L \cup M = \{\text{dog, ba, na, house}\}$ and $LM = \{\text{doghouse, dogba, bahouse, baba, nahouse, naba}\}$.
- ▶ Language exponentiation concatenates a language with itself a given number of times: $L^2 = LL$, $L^3 = LLL$, $L^4 = LLLL$, etc.
 - ▶ By definition $L^0 = \{ \}$ and $L^1 = L$.

Languages

- ▶ The Kleene closure of a language L , denoted by L^* , equals
 - ▶ $L^0 \cup L^1 \cup L^2 \cup L^3 \cup L^4 \cup \dots$
- ▶ For example, if $L = \{a, b\}$ then $L^* = \{ \varepsilon, a, b, aa, ab, ba, bb, aaa, aba, baa, bba, aab, abb, bab, bbb, \dots \}$.
- ▶ The positive closure of a language L , denoted by L^+ , equals $L^1 \cup L^2 \cup L^3 \cup L^4 \cup \dots$
- ▶ For example, if $L = \{a, b\}$ then $L^+ = \{a, b, aa, ab, ba, bb, aaa, aba, baa, bba, aab, abb, bab, bbb, \dots \}$.

Languages

- ▶ Let $L = \{A, B, \dots, Z, a, b, \dots, z\}$ be the set of all capital and lower-case letters and let $D = \{0, 1, \dots, 9\}$ be the set of all digits. Then:
 - ▶ $L \cup D$ is the set of all letters and digits.
 - ▶ LD is the set of strings consisting of a letter followed by a digit.
 - ▶ L^4 is the set of all four-letter strings.
 - ▶ L^* is the set of all strings of letters including the empty string, ϵ .
 - ▶ $L(L \cup D)^*$ is the set of all strings of letters and digits that begin with a letter.
 - ▶ D^+ is the set of all strings of one or more digits.

Regular Expressions

- Specify simple (possibly infinite) strings
- The set of strings defined by a RE are called *regular sets*
- Start with a finite character set called a *vocabulary*, or V
- Empty strings are allowed
- Strings are created by concatenating characters from the vocabulary
- Just like used for lex
- REs are great for defining tokens



Regular Expressions

- ▶ Every regular expression specifies a language according to the following rules:
 - ▶ ε is a regular expression that denotes $\{ \}$, the set containing just the empty string.
 - ▶ If a is a symbol in the alphabet then the regular expression a denotes $\{a\}$, the set containing just that symbol.
 - ▶ Suppose s and t are regular expressions denoting languages $L(s)$ and $L(t)$, respectively. Then:
 - ▶ $(s) \mid (t)$ is a regular expression denoting $L(s) \cup L(t)$.
 - ▶ $(s)(t)$ is a regular expression denoting $L(s)L(t)$.
 - ▶ $(s)^*$ is a regular expression denoting $L(s)^*$.
 - ▶ (s) is another regular expression denoting $L(s)$ so we can put extra pairs of parentheses around regular expressions if we desire.
- ▶ Unnecessary parentheses can be avoided in regular expressions using the convention that Kleene closure, $*$, is left-associative and takes precedence over concatenation which is left-associative and takes precedence over union, \mid , which is also left-associative.

Regular Definitions

- ▶ A regular definition gives names to certain regular expressions and uses those names in other regular expressions. As an example:
 - ▶ letter $\rightarrow A \mid B \mid \dots \mid Z \mid a \mid b \mid \dots \mid z$
 - ▶ digit $\rightarrow 0 \mid 1 \mid \dots \mid 9$
 - ▶ id $\rightarrow \text{letter} (\text{letter} \mid \text{digit})^*$
- ▶ This defines letter to be the regular expression for the set of all upper-case and lower case letters in the English alphabet, digit to be the regular expression for the set of all decimal digits, and id to be the regular expression for all strings of letters and digits that begin with a letter.
 - ▶ Note that id is the pattern for the Pascal identifier token.

Regular Definitions

- ▶ The pattern for the Pascal number token can be specified as follows:
- ▶ $\text{digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$
- ▶ $\text{digits} \rightarrow \text{digit} \text{ digit}^*$
- ▶ $\text{optional_fraction} \rightarrow . \text{digits} \mid \varepsilon$
- ▶ $\text{optional_exponent} \rightarrow (E (+ \mid - \mid \varepsilon) \text{digits}) \mid \varepsilon$
- ▶ $\text{num} \rightarrow \text{digits} \text{ optional_fraction} \text{ optional_exponent}$
- ▶ This definition says that an `optional_fraction` is either a decimal point followed by one or more digits or it is missing (the empty string).
- ▶ An `optional_exponent` is either the empty string or it is the letter E followed by an optional + or - sign, followed by one or more digits.

Notational Shorthands

- ▶ If r is a regular expression then:

$$(r)^+ = r r^*$$

and

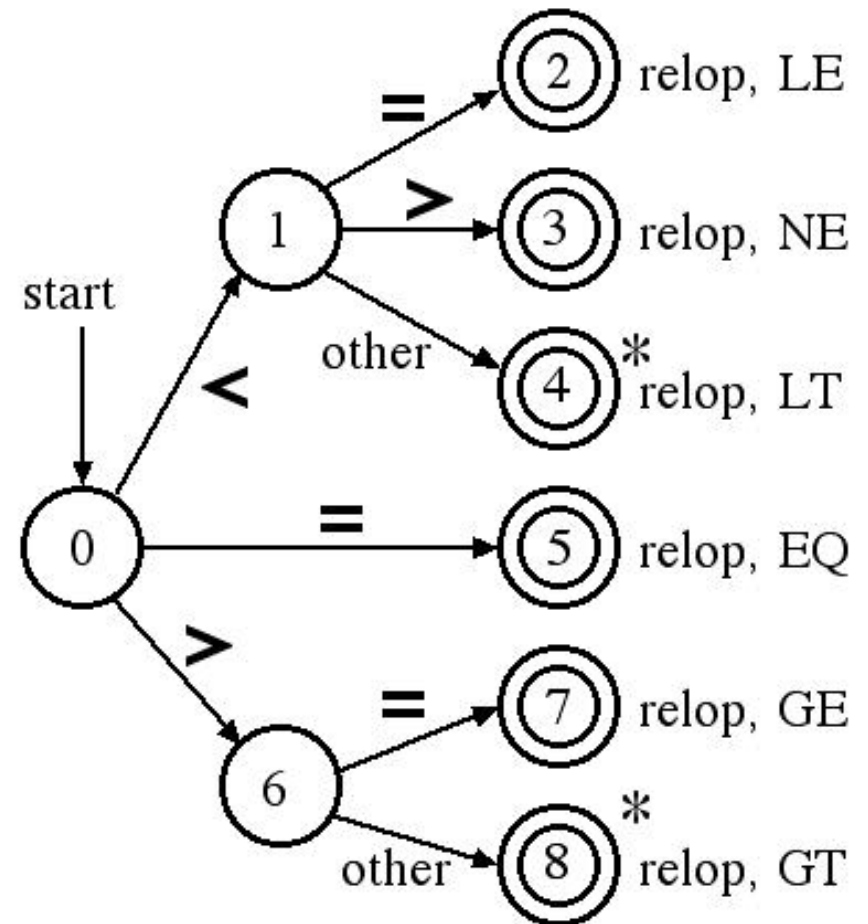
$$r? = (r \mid \varepsilon)$$

- ▶ With these shorthand notations one can write:

$$\begin{aligned} \text{num} &= \text{digits} (. \text{digits})? (E (+ \mid -)? \text{digits})? \\ &= \text{digit}^+ (. \text{digit}^+)? (E (+ \mid -)? \text{digit}^+)? \end{aligned}$$

Recognition of Tokens

- ▶ One way to recognize a token is with a finite state automaton following a particular transition diagram.
- ▶ For example, the transition diagram for the RELOP token of Pascal is:
- ▶ The start state for this diagram is state 0.
- ▶ If the input character agrees with one of the arrows leaving that state then the analyzer goes to the state at the end of that arrow and examines the next input character.
- ▶ If none of the arrows leaving a state agrees with the input character then this particular transition diagram does not apply and the analyzer should return to the first input character and try the diagram for some other token.
- ▶ States with double circles are accept states where the analyzer can return a token to the caller.
- ▶ States with stars, *, indicate places where the last input character read must be returned to the input.



Finite Automaton

- A *finite automaton* (FA), is a simple idealized computer that recognizes strings belonging to regular sets
- FAs can be used to recognize tokens defined by REs
- An FA consists of
 - A finite set of states
 - A set of input symbols V
 - A set of transitions from one state to another labeled by characters in V
 - A special start state
 - A set of final or accepting states



Deterministic FAs

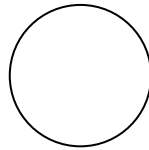
- If an FA always has a unique next state it is *deterministic* or a DFA, otherwise it is *nondeterministic* or an NFA
- DFAs can be represented by transition tables (these can be compressed)
- Any RE can be translated into a DFA that accepts the set of string denoted by RE
- Translation can be done automatically or manually
- DFAs are faster than NFAs, but NFAs can be smaller.



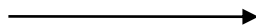
Finite Automaton

- *Transition diagrams* are a graphical representation of FAs

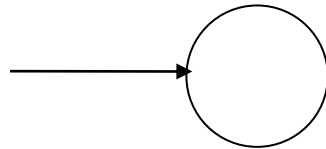
A state



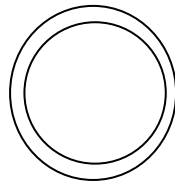
A transition



Start state

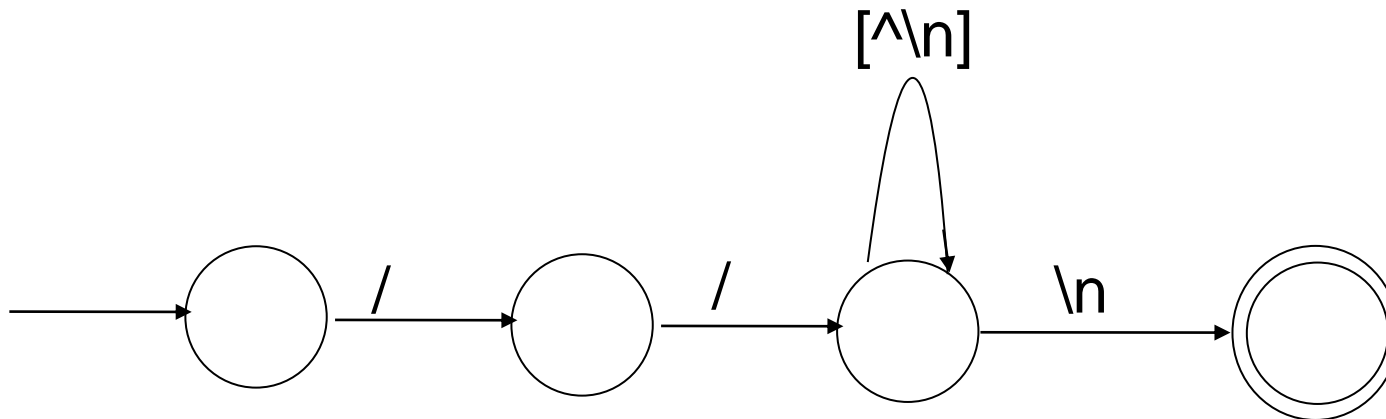


Final state



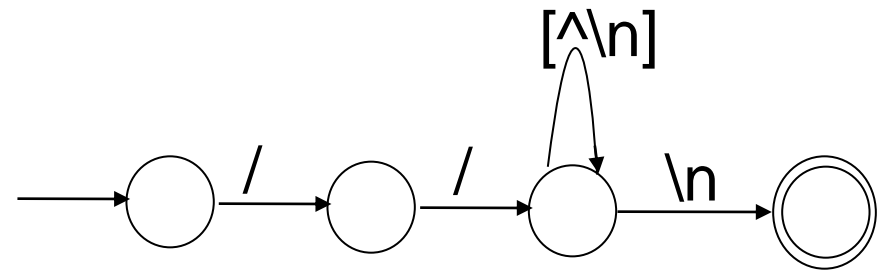
Finite Automaton

- Let's do a c++ comment



Finite Automaton

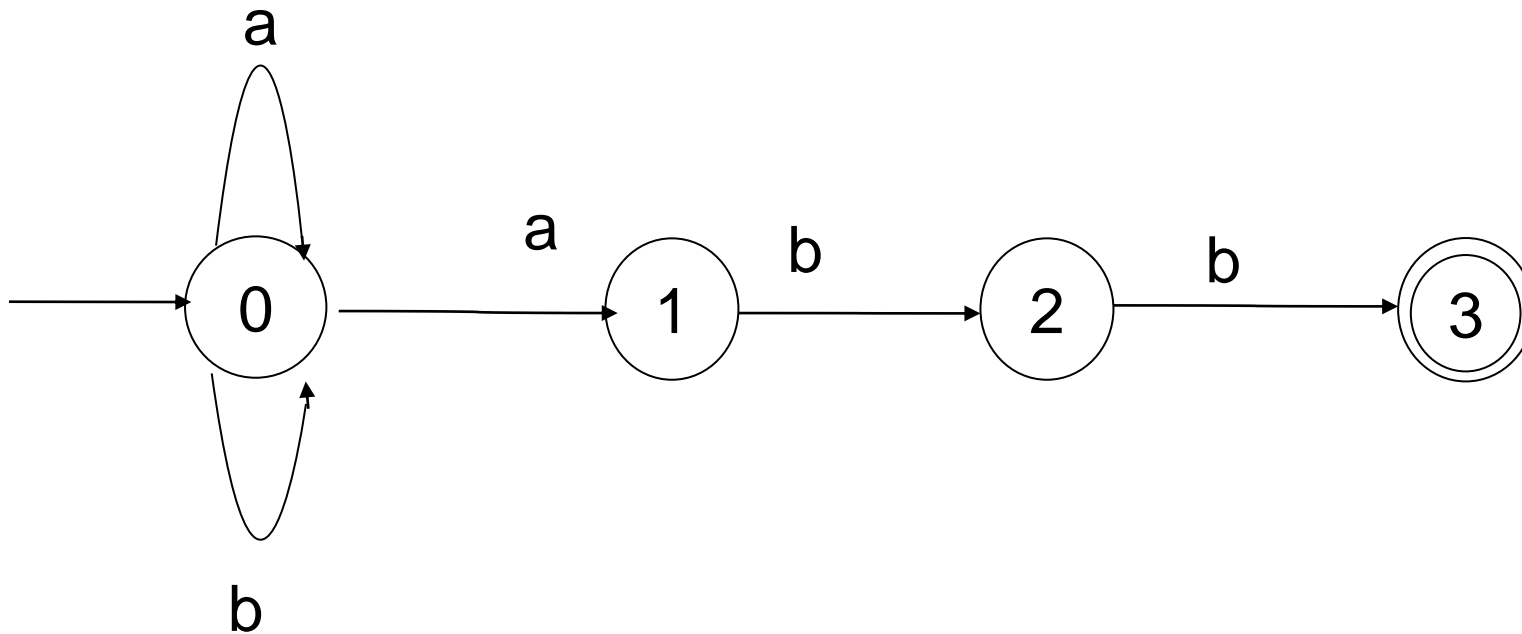
- Let's do a c++ comment



State	Character				
	/	\\n	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Error entries are blank

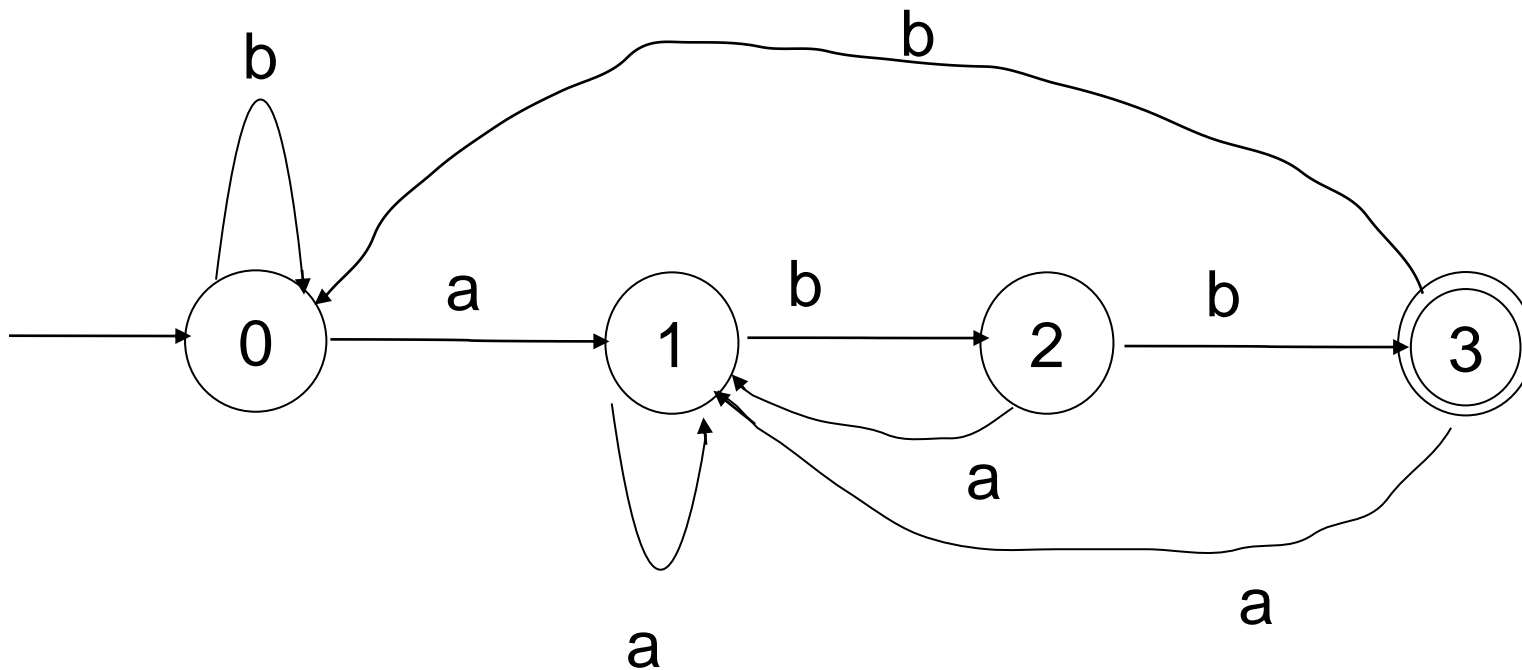
Nondeterministic FA



NFA accepting $(a|b)^*abb$



Another FA

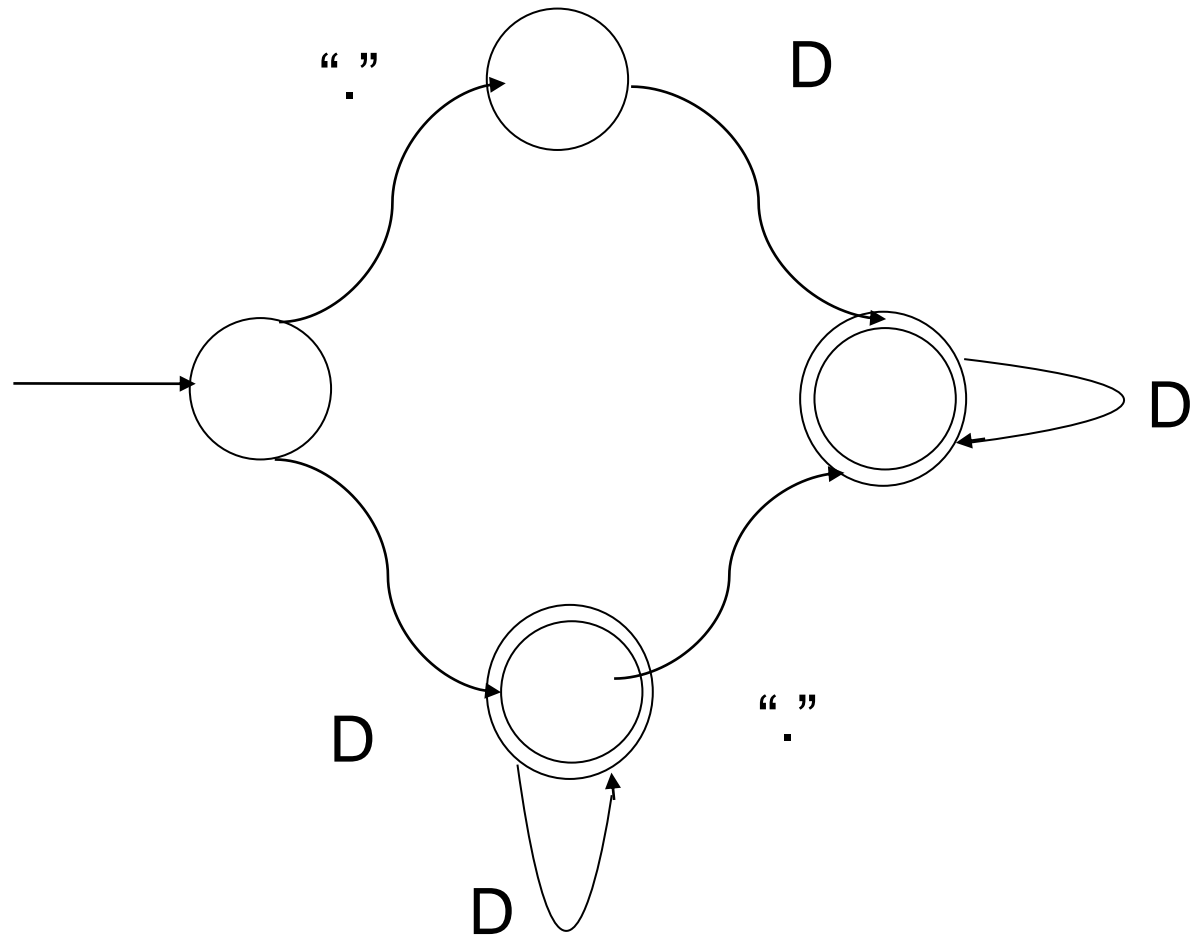


What strings to I accept?
Am I a DFA or an NFA?



Finite Automaton

- What do I do?
- D is [0-9]



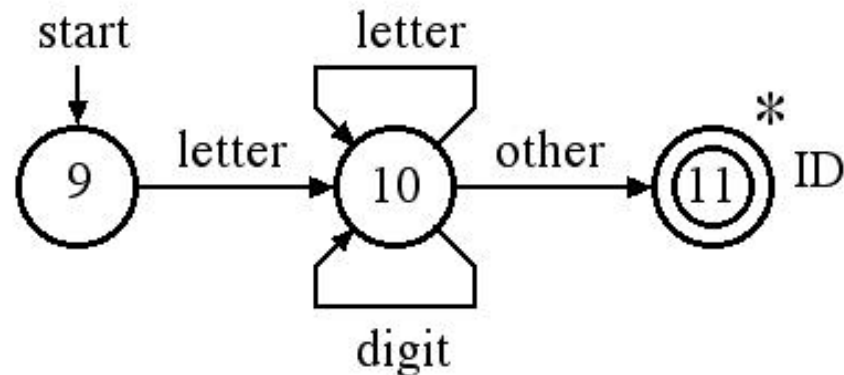
Creating scanners

- **Keywords**
 - Create a regular expression for each
 - This increases the size of the tables
 - Treat them as variables and use LUT
 - More effort but smaller transition tables
 - Can create a RE the recognizes only valid vars, that is doesn't match keywords
 - Smaller tables, but RE is VERY complex
- **Programming Listing**
- **Arguments**



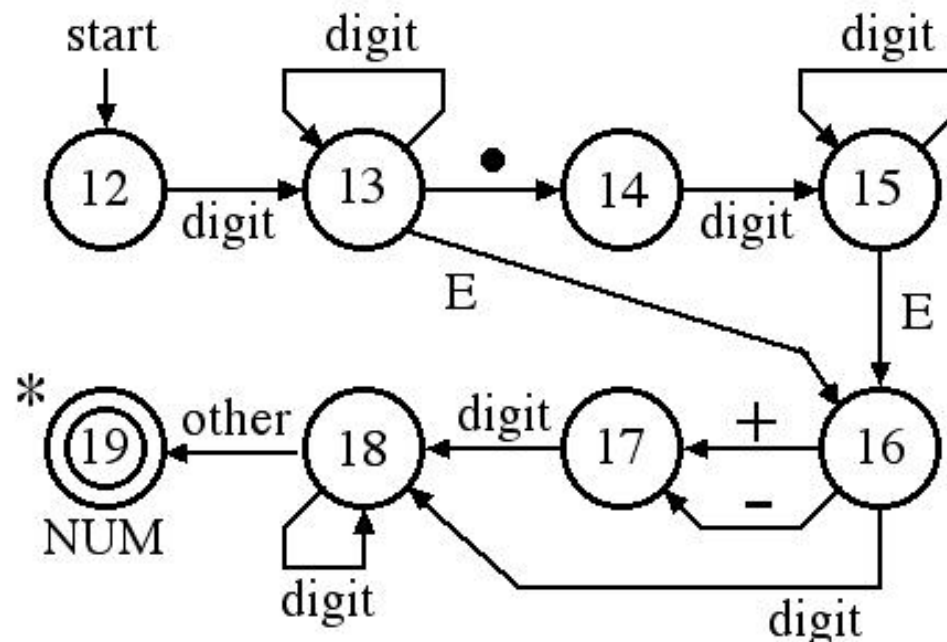
Recognition of Tokens

- ▶ The transition diagram for the ID token of Pascal is:



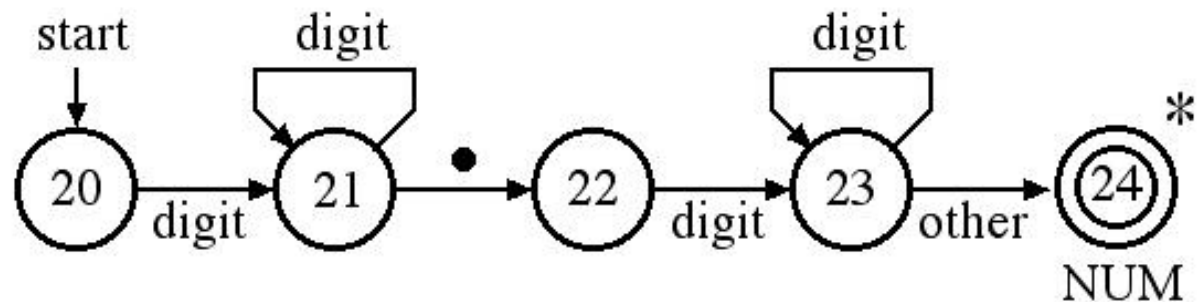
Recognition of Tokens

- ▶ The recognition of the NUM token of Pascal can be done with three different transition diagrams.
- ▶ The first diagram accepts real numbers with exponents:



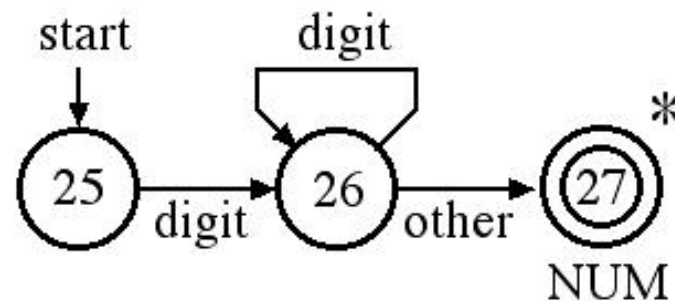
Recognition of Tokens

- ▶ If the first diagram fails to reach its accept state then we should try the second diagram which accepts real numbers with fractions but no exponents:



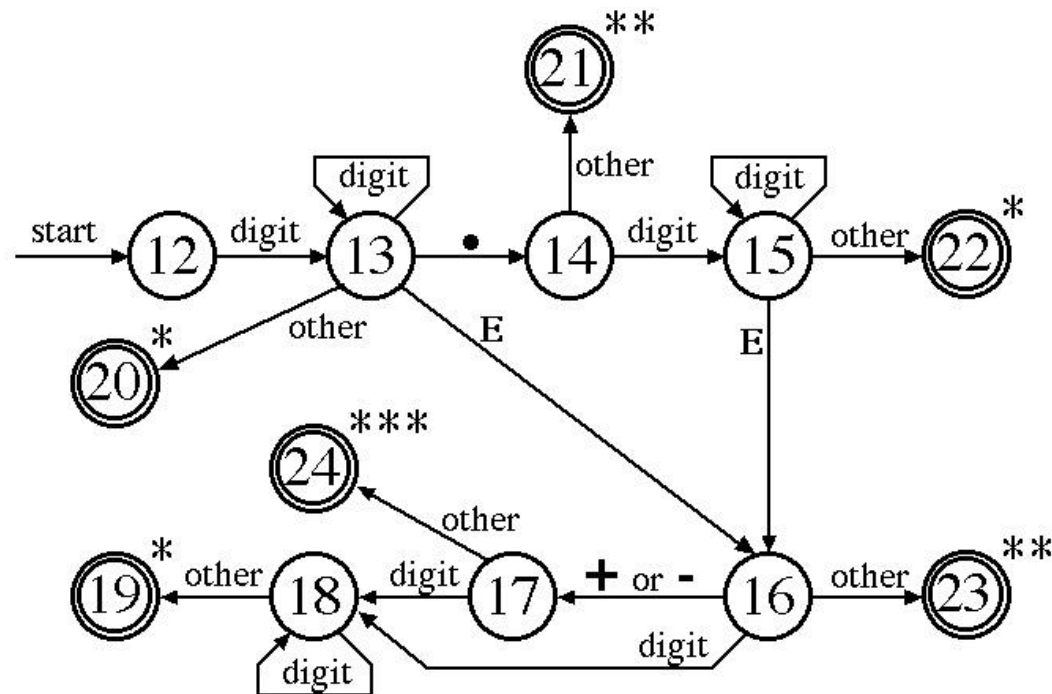
Recognition of Tokens

- ▶ If the second diagram also fails then we should try the third diagram which accepts integers:



Recognition of Tokens

- One can combine the three diagrams for unsigned numbers into one diagram but some of the accept states in this diagram have multiple stars to indicate that two or more characters must be pushed back onto the input:



Recognition of Tokens

- ▶ As shown on the previous diagram, the number of characters that must be pushed back onto the input is:
 - ▶ one if accept state 19, 20, or 22 is reached;
 - ▶ two if accept state 21 or 23 is reached;
 - ▶ and three if accept state 24 is reached.
- ▶ The number is real if state 15 or 18 (or both) is encountered by the transition - if neither state 15 nor state 18 is encountered then the number is an integer.
- ▶ One can use transition diagrams to implement a lexical analyzer:
 - ▶ the analyzer has a large case statement with a case for each state in the diagrams.

A Language for Specifying Lexical Analyzers

- ▶ `lex` is one of a number of tools one can use to create a lexical analyzer from a description based on regular expressions.
- ▶ `lex` creates a C code file, `lex.yy.c`, from a description of the analyzer written in the Lex language;
- ▶ `lex.yy.c` is fed into a C compiler to produce the object program of the lexical analyzer, `a.out` ;
- ▶ `a.out` will analyze an input source file lexically to produce a sequence of tokens.
- ▶ Details forthcoming...

Finite Automata

- ▶ A recognizer for a language is a program that takes an input string x and answers
 - ▶ Yes – if x is in the language or
 - ▶ No – if x is not in the language.
- ▶ One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton.

Nondeterministic Finite Automata

▶ Nondeterministic Finite Automata:

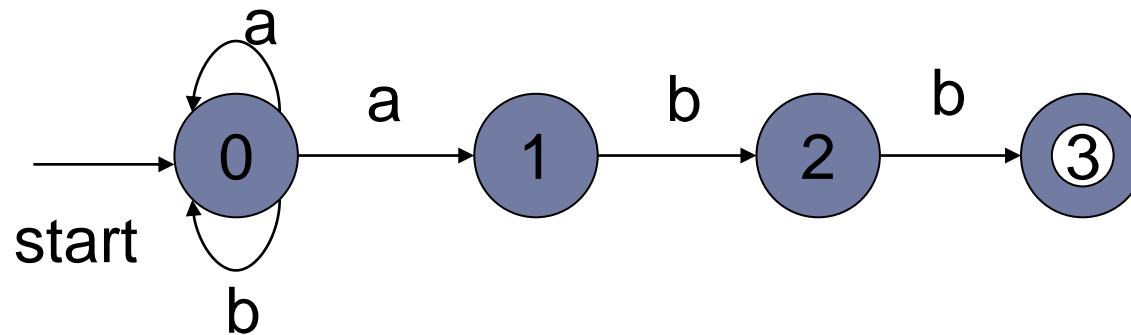
- ▶ A Nondeterministic Finite Automaton or NFA for short consists of:
 - ▶ a set of states \underline{S} ;
 - ▶ a set of input symbols called the input symbol alphabet;
 - ▶ a transition function move that maps state-symbol pairs to sets of states;
 - ▶ a state \underline{s}_0 called the initial state or the start state;
 - ▶ a set of states \underline{F} called the accepting or final states.

Nondeterministic Finite Automata

- ▶ An NFA can be represented by a labeled directed graph (a **transition graph**) where the nodes are states and the labeled edges show the transition function.
- ▶ The label on each edge is either a symbol in the alphabet or ε denoting the empty string.

Nondeterministic Finite Automata

- ▶ The following figure shows an NFA that recognizes the language: $(a \mid b)^* a b$



- ▶ This automaton is nondeterministic because when it is in state-0 and the input symbol is a, it can either go to state-1 or stay in state-0.
- ▶ The transition table for this NFA is:

State	Input Symbol	
	a	b
0	{0,1}	{0}
1	-	{2}
2	-	{3}

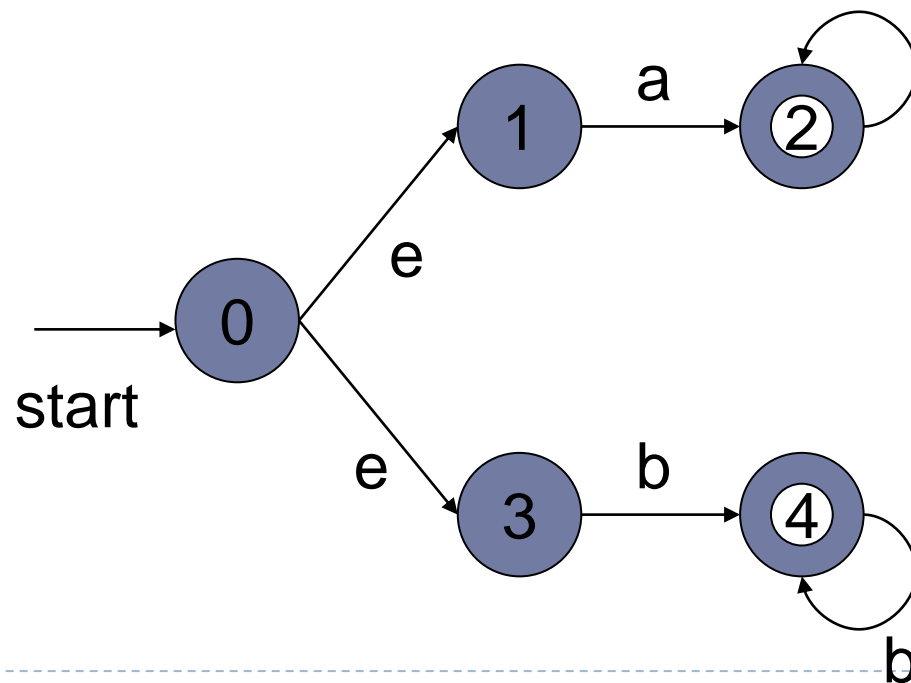
Nondeterministic Finite Automata

- ▶ An NFA accepts an input string x if and only if there is some path in the transition graph from the start state to some final state such that the edge labels along this path spell out x .
- ▶ The following table shows the paths for three of the strings that the previous NFA accepts:

x	Path
a b b	0→1→2→3
a a b b	0→0→1→2→3
b a b b	0→0→1→2→3

Nondeterministic Finite Automata

- ▶ Note that there may be several paths that a given input string might take through an NFA
 - ▶ if at least one such path ends in a final state then the string is accepted.
 - ▶ The language defined by an NFA is the set of input strings it accepts.
 - ▶ The following figure shows an NFA that uses ϵ -transitions to define the language: $a a^* \mid b b^*$
 - ▶ The transition table for this NFA is:



State	Input Symbol		
	a	b	ϵ
0	-	-	{1,3}
1	{2}	-	-
2	{2}	-	-
3	-	{4}	-
4	-	{4}	-

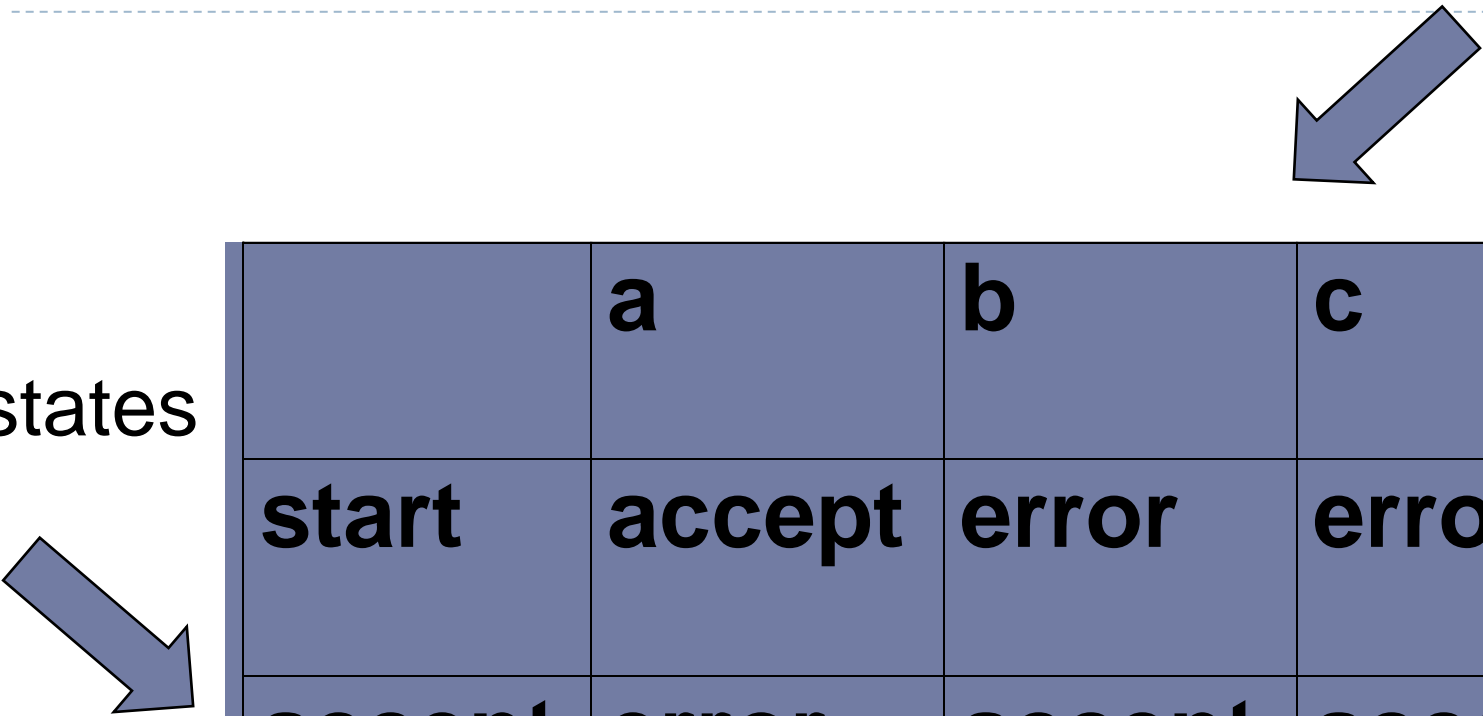
Deterministic Finite Automata

- ▶ A deterministic finite automaton or DFA for short is a special case where:
 - ▶ there are no ε -transitions; and
 - ▶ for each state s and input symbol a there is at most one edge labeled a leaving s .
- ▶ The transition table of a DFA has no ε -column and every entry is a single state.
- ▶ It is very easy to determine whether a DFA accepts a given input string x or not since there is at most a single path for each string.

Example of a DFA

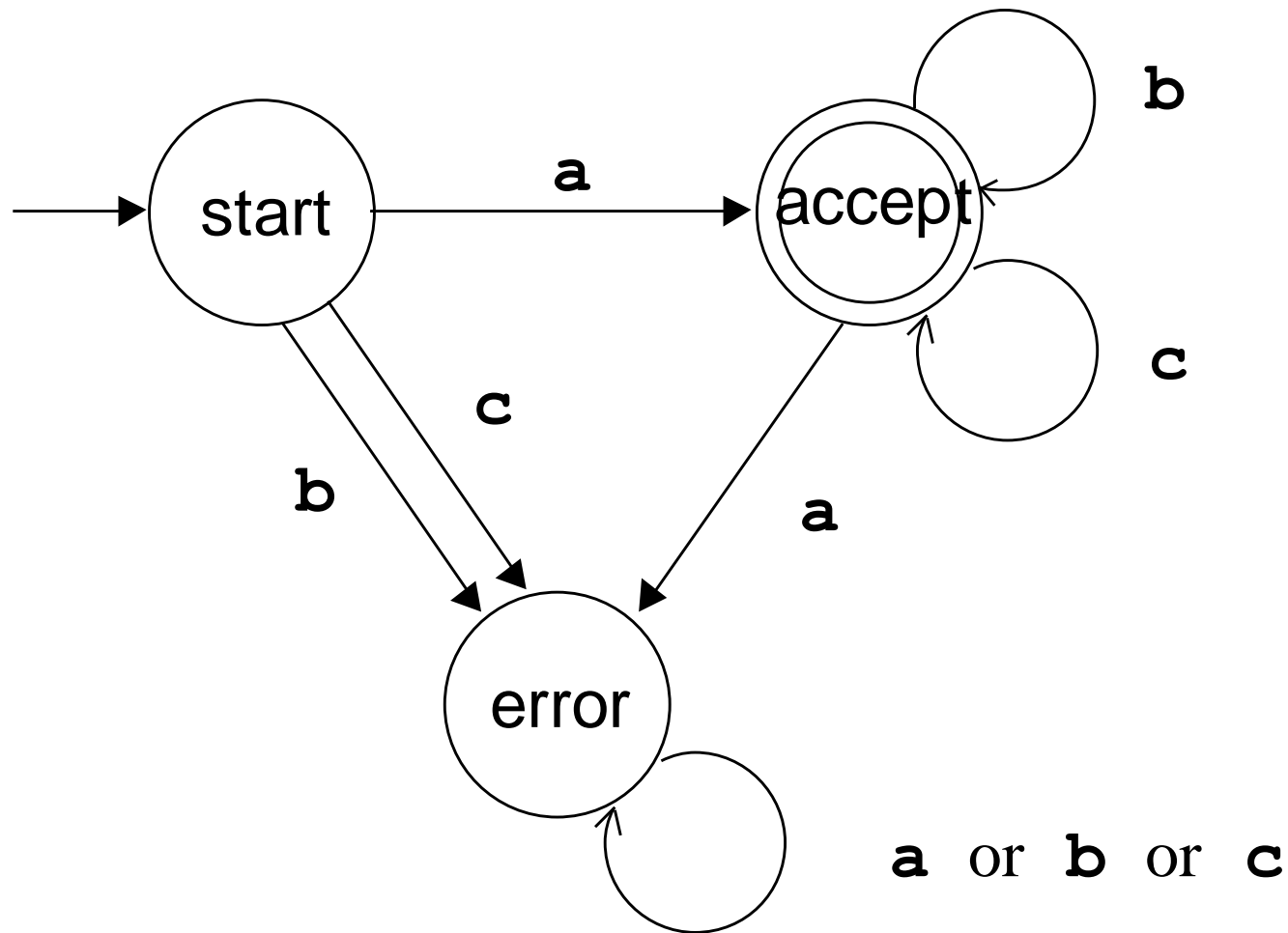
input

states

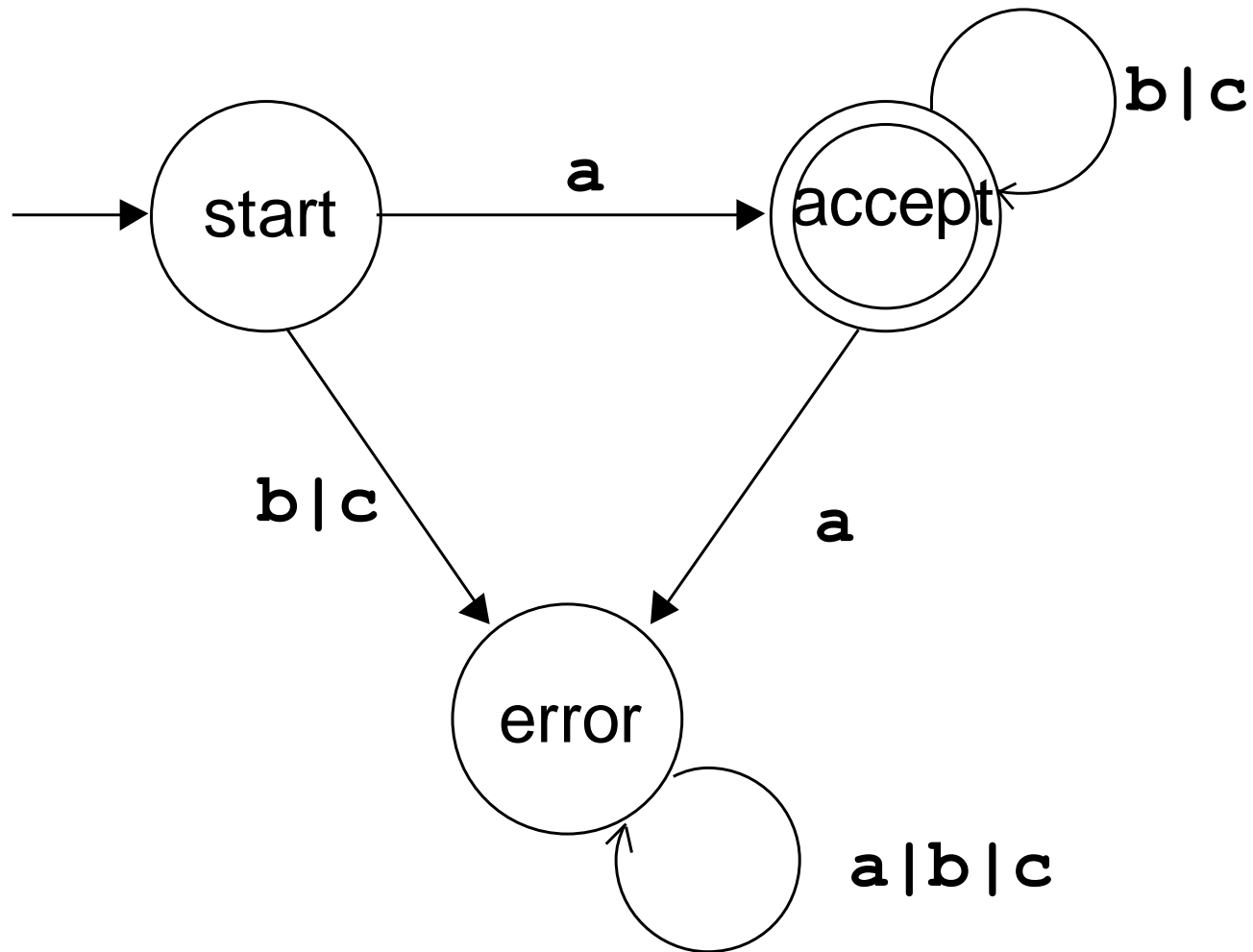


	a	b	c
start	accept	error	error
accept	error	accept	accept
error	error	error	error

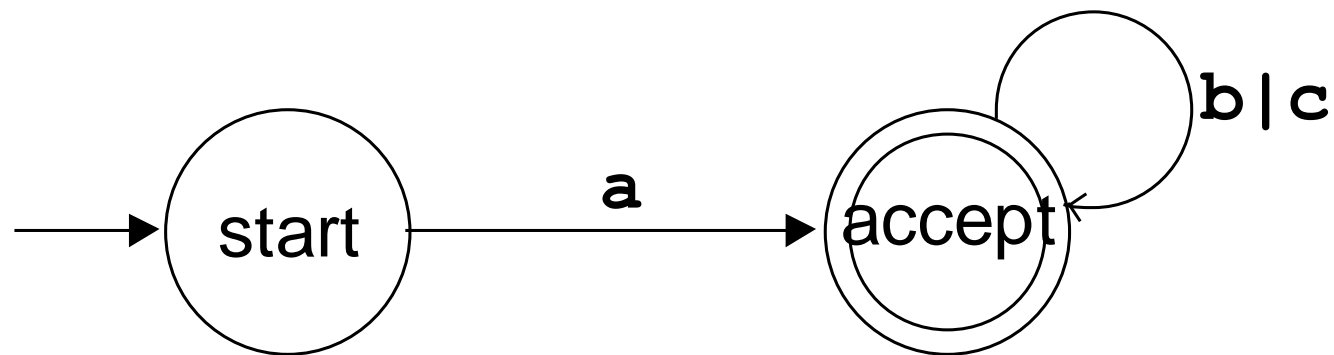
Another View of the Same DFA



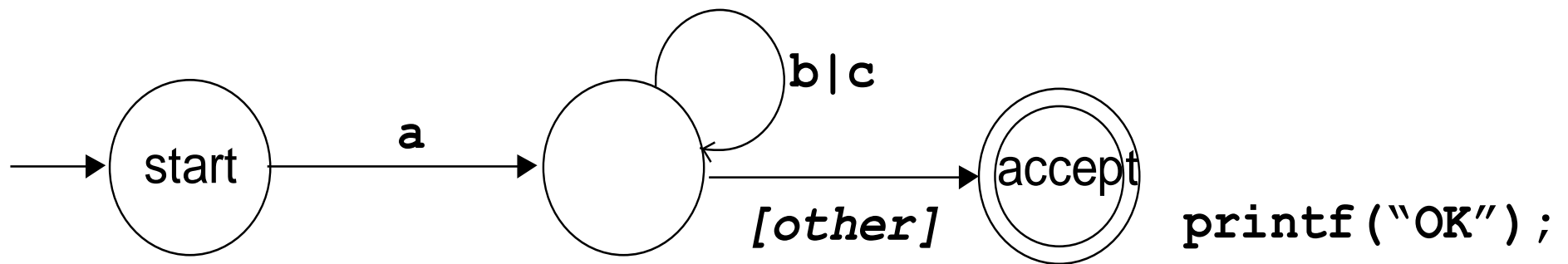
Yet Another View of the Same DFA



Yet Another View of the Same DFA

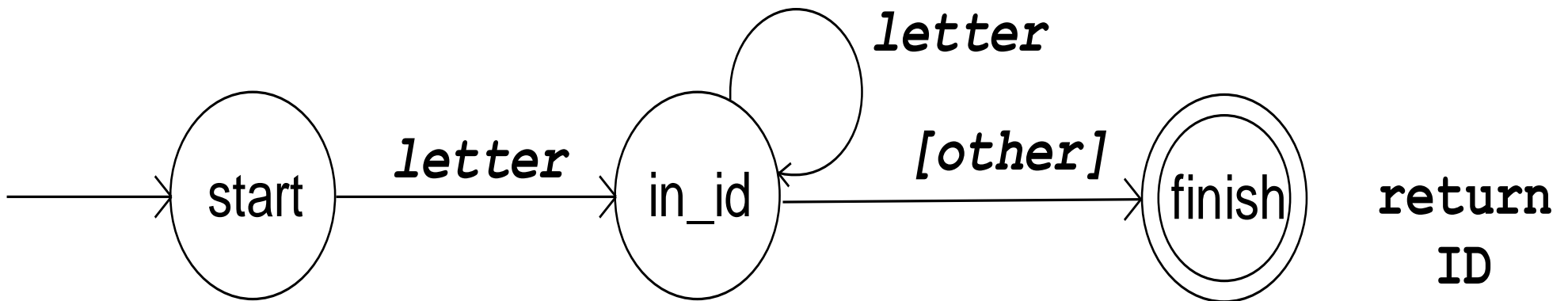


Previous DFA with Stopping Condition and Action



- ▶ This diagram expresses the Principle of Longest Substring (or Maximal Munch):
 - ▶ the DFA matches the longest possible input string before stopping.

A DFA for Identifiers in TINY



DFA Conventions

- ▶ Error transitions are not explicitly shown
- ▶ Input symbols that result in the same transition are grouped together (this set can even be given a name)
- ▶ Still not displayed: stopping conditions and actions

Deterministic Finite Automata

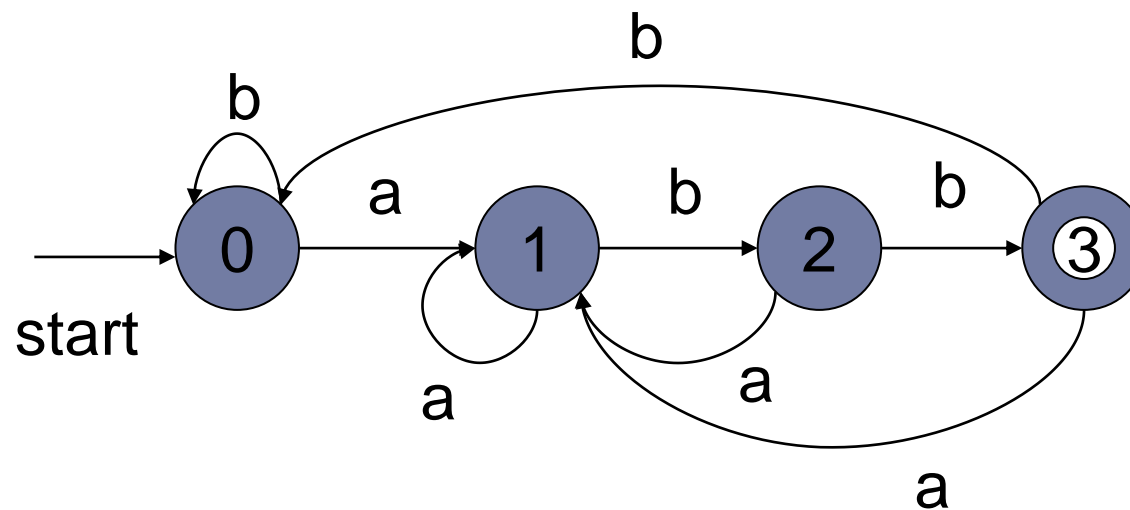
- ▶ Here is an algorithm that simulates a DFA:

- ▶ INPUT: An input string x terminated by an end-of-file character eof .
 - ▶ A DFA D with start state s_0 and a set of accepting states F .
- ▶ OUTPUT: The answer “yes” if D accepts x ; “no” otherwise
- ▶ METHOD: Apply the algorithm to the input string x .
 - ▶ The function $\text{move}(s, c)$ gives the state to which there is a transition from state s on input character c .
 - ▶ The function nextchar returns the next character of the input string x .

```
s := s0;
c := nextchar;
while c != eof do
begin
    s := move(s, c);
    c := nextchar;
end;
if s is in F then
    return "yes"
else
    Return "no"
```

Deterministic Finite Automata

- ▶ This figure shows a DFA that recognizes the language
 - ▶ $(a \mid b)^* a b b$.
 - ▶ Note that it is more complicated than an NFA for the same language



Turning a DFA into Code

- ▶ States are an enum (or defined constants)
- ▶ Code is a single outermost loop
- ▶ Inside the loop is a single switch statement on the state
- ▶ The code for each state is a single switch (or if) statement on the input symbol

Sample Code for a TINY ID

```
state = start;      advance(input);
while (state != finish && state != error)
    switch (state) {
        case start: if (isalpha(input)) {
            advance(input); state = in_id;}
            else state = error; break;
        case in_id: if (!isalpha(input))
            state = finish;
            else advance(input); break;
        default: break;
    }
if (state == finish) return ID;
else return ERROR;
```

Alternative ID Code

```
state = start;          advance(input);
while (TRUE)
    switch (state) {
        case start: if (isalpha(input)) {
            advance(input); state = in_id;}
            else return ERROR; break;
        case in_id: if (!isalpha(input))
            return ID;
            else advance(input); break;
        default: break;
    }
```

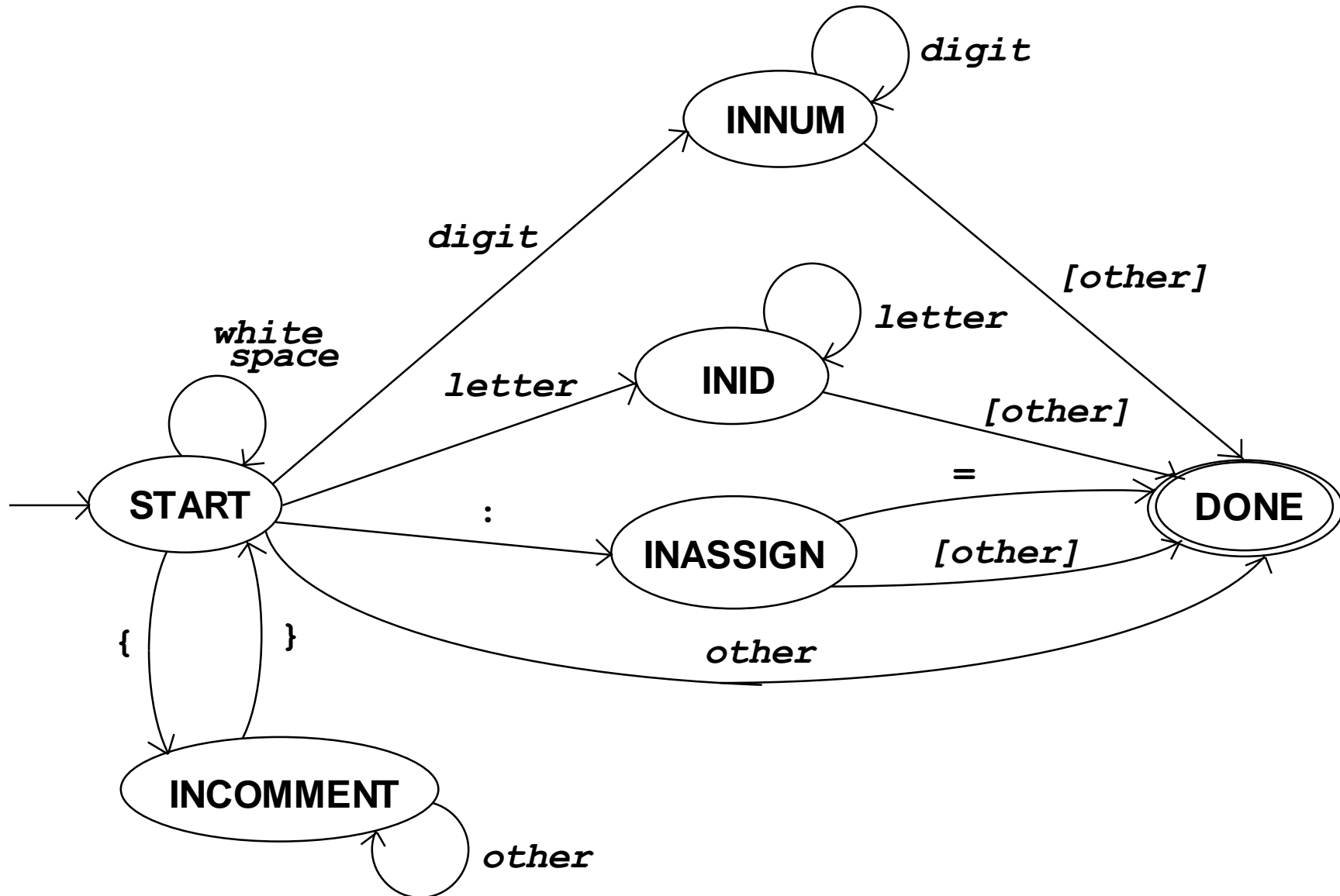

Never write this:

```
advance(input);  
if (!isalpha(input)) return ERROR;  
while (isalpha(input))  
    advance(input);  
return ID;
```

Another Alternative: Table-driven Code

```
state = start; advance(input);  
while (!accept[state] && !err[state])  
{ newstate = t[state][input];  
  if (adv[state][input])  
    advance(input);  
  state = newstate;  
}  
...
```

TINY DFA:



What happened to reserved words?

- ▶ Recognize them as identifiers first
- ▶ Then look them up in a table of reserved words
- ▶ Linear list (TINY) is bad.
- ▶ Binary search (ordered list) is better.
- ▶ Hash table is even better.
 - ▶ Hash table with size I buckets (perfect hash function) is best.

Additional Pragmatics

- ▶ `void advance(char input)`
- ▶ `{ store input in the lexeme* buffer;`
- ▶ `advance the buffer index`
- ▶ `or refill the buffer and setindex to 0`
- ▶ `(watching out for EOF!);`
- ▶ `}`

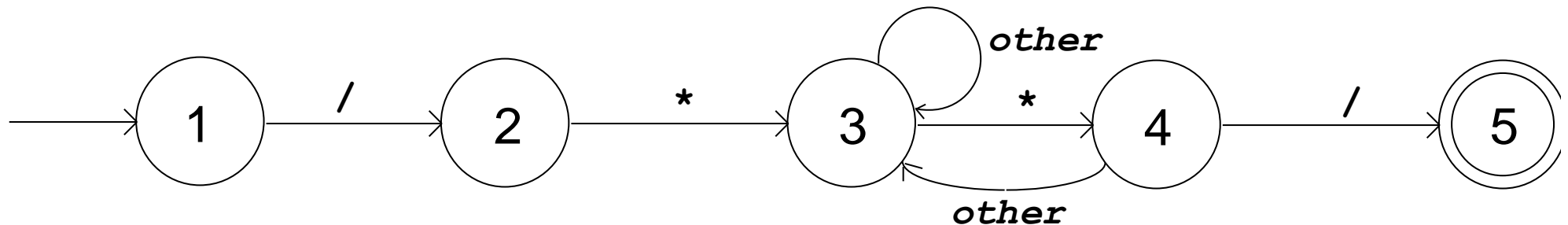
- ▶ ***Lexeme**: the token's actual character string

But How to Turn Token Descriptions into Such a DFA?

- ▶ Ad hoc methods are error-prone
- ▶ Maintaining determinism can be tricky
- ▶ Even the token descriptions themselves might be ambiguous or subject to interpretation

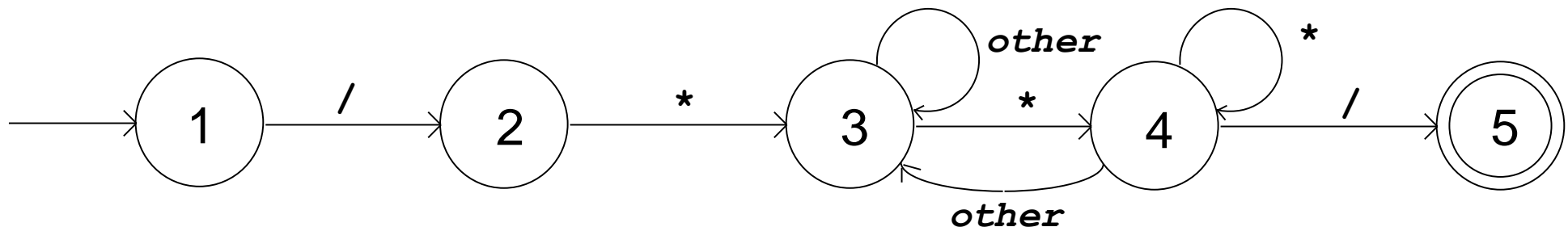
Example

- ▶ A comment in C is any sequence of characters between the delimiters “/*” and “*/”
- ▶ The following DFA is wrong!



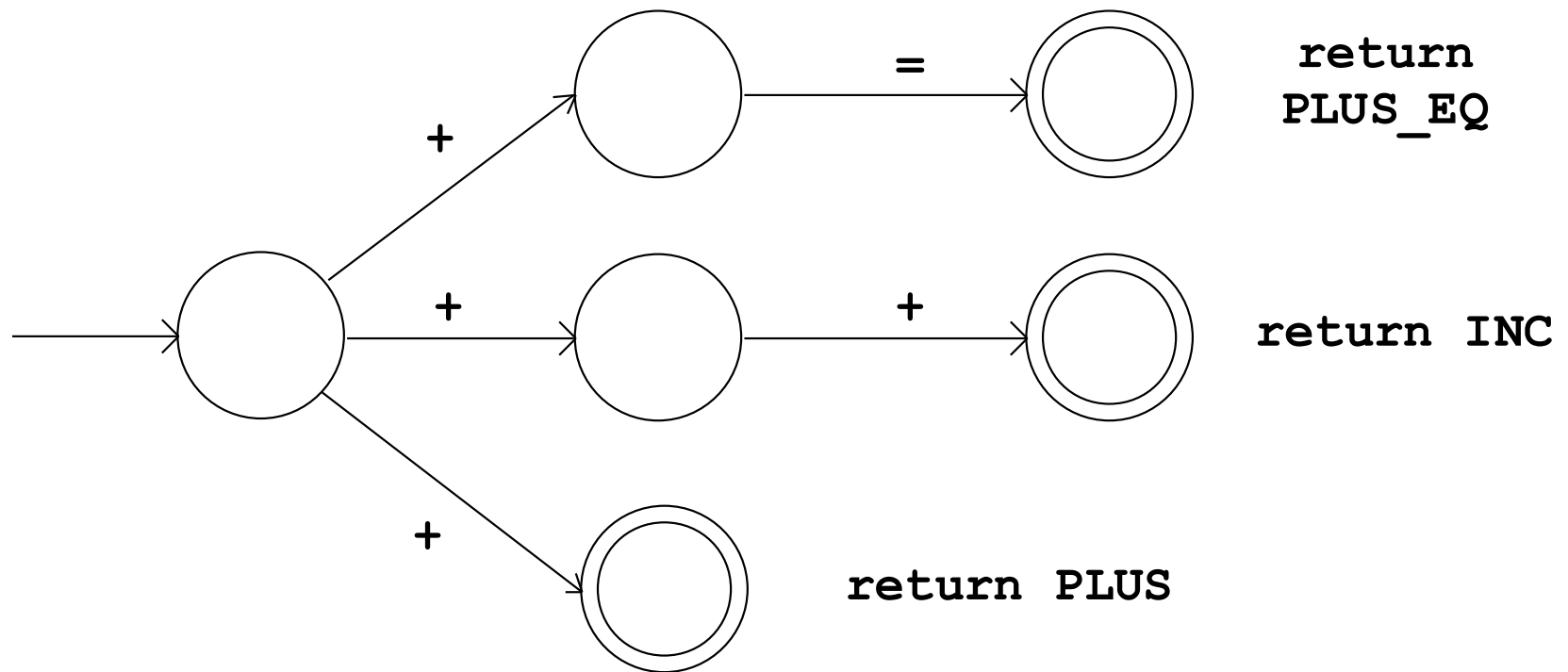
Example (cont.)

- ▶ Here is the correct DFA:
- ▶ But still:
 - ▶ What if EOF is encountered inside?
 - ▶ What if a comment is encountered in a string?
 - ▶ What if a string is encountered in a comment?
 - ▶ Do comments count as white space or not?



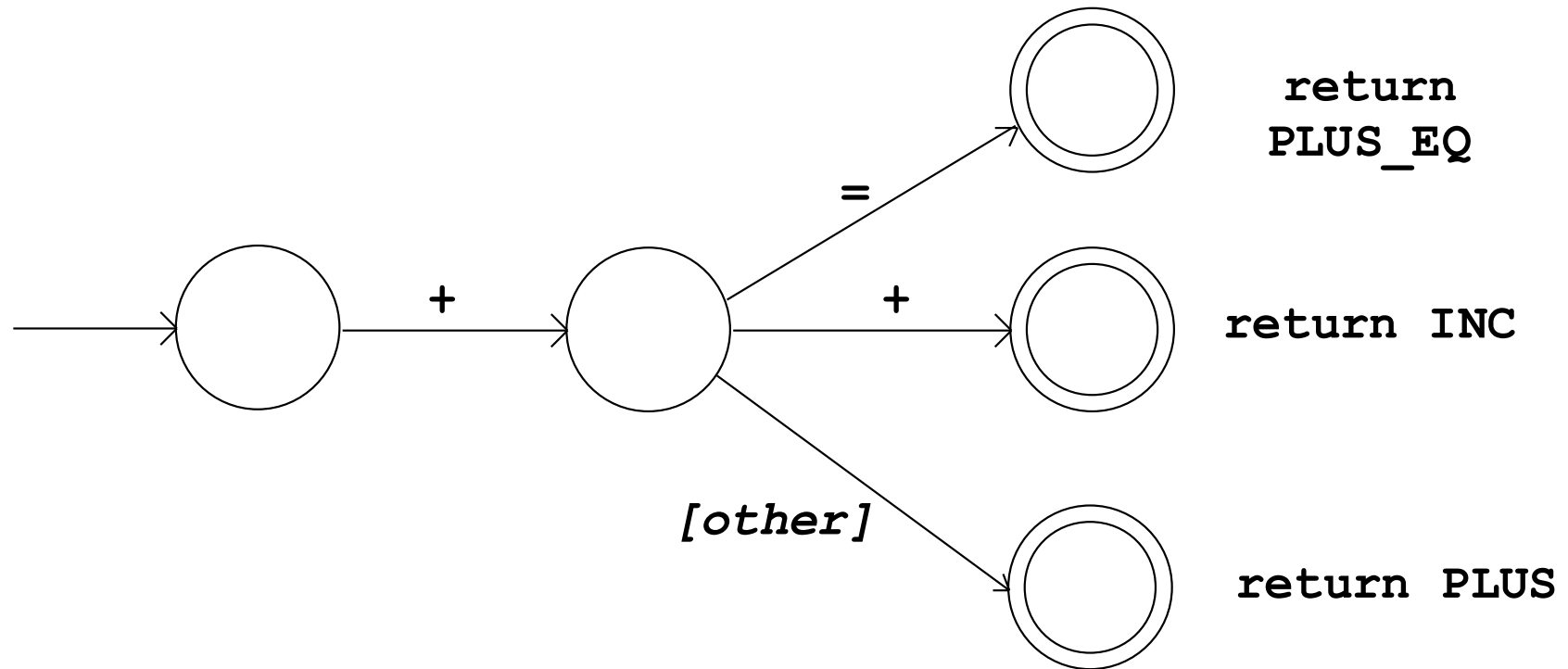
Another Example

- ▶ The following automaton is nondeterministic:



Another Example, Continued

- ▶ It must be replaced by this:



A Better Approach:

- ▶ Use regular expressions to define the tokens.
- ▶ Use well-known (and correct) mathematical algorithms to convert the tokens into a DFA.
- ▶ Even better: automate the entire process from regular expressions to code:
 - ▶ lex (unix) - 1975 (Mike Lesk)
 - ▶ flex (usenix) - 1985 (Van Jacobson, Vern Paxson)

Conversion of an NFA to a DFA

- ▶ It is hard for a computer program to simulate an NFA because the transition function is multi-valued.
- ▶ Fortunately, the **Subset Construction Algorithm** will convert an NFA for any language into a DFA that recognizes the same language
 - ▶ (it is closely related to an algorithm for constructing LR parsers.

Conversion of an NFA to a DFA

- ▶ Each entry in the transition table of an NFA is a set of states;
 - ▶ Each entry in the table of a DFA is just a single state
 - ▶ The basic idea behind the NFA-to-DFA construction is that each DFA state corresponds to a set of NFA states.
 - ▶ For example, let T be the set of all states that an NFA could reach after reading input: a_1, a_2, \dots, a_n
 - ▶ Then the state that the DFA reaches after reading a_1, a_2, \dots, a_n corresponds to set T
- ▶ Theoretically, if the number of states in the NFA is n then the DFA might have $\Theta(2^n)$ states but this worst case is a very rare occurrence.

Conversion of an NFA to a DFA

- ▶ Subset construction algorithm uses certain operations to keep track of sets of NFA states
 - ▶ If s is an NFA state, T is a set of NFA states, and a is a symbol in the input alphabet then:

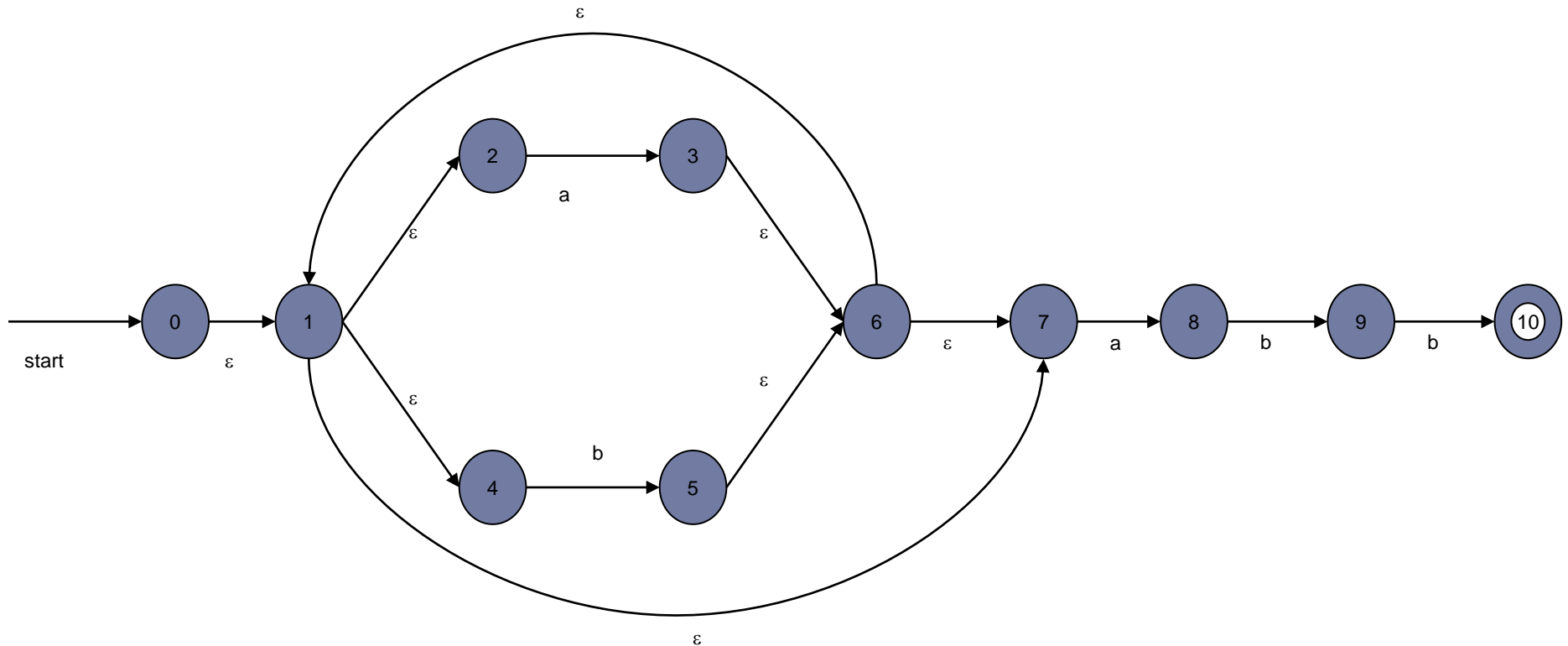
Operation	Description
ε -closure	Set of NFA states reachable by ε -transitions from state s .
ε -closure	Set of NFA states reachable by ε -transitions from every state s in T .
$\text{Move}(T, a)$	Set of NFA states to which there is a transition on input symbol a from some state s in T .

Conversion of an NFA to a DFA

- ▶ ε -closure algorithm
- ▶ push all states in T onto *stack*;
- ▶ initialize $\varepsilon\text{-closure}(T)$ to T ;
- ▶ while *stack* is not empty do begin
 - ▶ pop t , the top element, off the *stack*;
 - ▶ for each state u with an edge from t to u labeled ε do
 - ▶ if u is not in $\varepsilon\text{-closure}(T)$ do begin
 - add u to $\varepsilon\text{-closure}(T)$;
 - push u onto *stack*
 - ▶ end
- ▶ end
- ▶ Subset Construction Algorithm
- ▶ initially, $\varepsilon\text{-closure}(s_0)$ is the only state in $Dstates$ and it is unmarked;
- ▶ while there is an unmarked state T in $Dstates$ do begin
 - ▶ mark T ;
 - ▶ for each input symbol a do begin
 - ▶ $u := \varepsilon\text{-closure}(\text{move}(T, a))$;
 - ▶ If u is not in $Dstates$ then
 - Add u as an unmarked state to $Dstates$
 - $Dtran[T, a] := u$;
 - ▶ end
- ▶ end

Conversion of an NFA to a DFA - Example

- ▶ Consider the NFA N accepting the language
 - ▶ $(a \mid b)^*abb$
 - ▶ Mechanically convert this to a DFA



Conversion of an NFA to a DFA - Example

- ▶ The start state of the NFA is state 0
- ▶ *Dstates* is initialized with the DFA state,
 - ▶ $\varepsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$, which is called A.

States	Dtran	
	Input Symbol	
	a	b
$A=\{0,1,2,4,7\}$	-	-

Conversion of an NFA to a DFA - Example

- ▶ The first iteration of the while-loop:
 - ▶ marks A ;
 - ▶ computes $\varepsilon\text{-closure}(\text{move}(A, a)) = \varepsilon\text{-closure}(\text{move}(\{0, 1, 2, 4, 7\}, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$;
 - ▶ enters $B = \{1, 2, 3, 4, 6, 7, 8\}$ as an unmarked state in $Dstates$;
 - ▶ Sets $Dtran[A, a] = B$;
 - ▶ computes $\varepsilon\text{-closure}(\text{move}(A, b)) = \varepsilon\text{-closure}(\text{move}(\{0, 1, 2, 4, 7\}, b)) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\}$;
 - ▶ enters $C = \{1, 2, 4, 5, 6, 7\}$ as an unmarked state in $Dstates$; and
 - ▶ sets $Dtran[A, b] = C$.

States	Dtran	
	Input Symbol	
	a	b
$A=\{0,1,2,4,7\}$	B	C
$B=\{1,2,3,4,6,7,8\}$	-	-
$C=\{1,2,4,5,6,7\}$	-	-

Conversion of an NFA to a DFA - Example

- ▶ The second iteration of the while-loop:
 - ▶ marks B ;
 - ▶ computes $\epsilon\text{-closure}(\text{move}(B, a)) = \epsilon\text{-closure}(\text{move}(\{1, 2, 3, 4, 6, 7, 8\}, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$;
 - ▶ sets $Dtran[B, a] = B$;
 - ▶ computes $\epsilon\text{-closure}(\text{move}(B, b)) = \epsilon\text{-closure}(\text{move}(\{1, 2, 3, 4, 6, 7, 8\}, b)) = \epsilon\text{-closure}(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\}$;
 - ▶ enters $D = \{1, 2, 4, 5, 6, 7, 9\}$ as an unmarked state in $Dstates$; and
 - ▶ sets $Dtran[B, b] = D$.

States	Dtran	
	Input Symbol	
	a	b
$A=\{0,1,2,4,7\}$	B	C
$B=\{1,2,3,4,6,7,8\}$	B	D
$C=\{1,2,4,5,6,7\}$	-	-
$D=\{1,2,4,5,6,7,9\}$	-	-

Conversion of an NFA to a DFA - Example

- ▶ The third iteration of the while-loop:
 - ▶ marks C ;
 - ▶ computes ε -closure(move(C, a)) = e -closure(move($\{1, 2, 4, 5, 6, 7\}, a$)) = ε -closure($\{3, 8\}$) = $\{1, 2, 3, 4, 6, 7, 8\} = B$;
 - ▶ sets $Dtran[C, a] = B$;
 - ▶ computes e -closure(move(C, b)) = e -closure(move($\{1, 2, 4, 5, 6, 7\}, b$)) = e -closure($\{5\}$) = $\{1, 2, 4, 5, 6, 7\} = C$;
 - ▶ sets $Dtran[C, b] = C$.

States	Dtran	
	Input Symbol	
	a	b
$A=\{0,1,2,4,7\}$	B	C
$B=\{1,2,3,4,6,7,8\}$	B	D
$C=\{1,2,4,5,6,7\}$	B	C
$D=\{1,2,4,5,6,7,9\}$	-	-

Conversion of an NFA to a DFA - Example

- ▶ The fourth iteration of the while-loop:
 - ▶ marks D ;
 - ▶ computes $\varepsilon\text{-closure}(\text{move}(D, a)) = \varepsilon\text{-closure}(\text{move}(\{1, 2, 4, 5, 6, 7, 9\}, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$;
 - ▶ sets $Dtran[D, a] = B$;
 - ▶ computes $\varepsilon\text{-closure}(\text{move}(D, b)) = \varepsilon\text{-closure}(\text{move}(\{1, 2, 4, 5, 6, 7, 9\}, b)) = \varepsilon\text{-closure}(\{5, 10\}) = \{1, 2, 4, 5, 6, 7, 10\}$;
 - ▶ enters $E = \{1, 2, 4, 5, 6, 7, 10\}$ as an unmarked state in $Dstates$; and
 - ▶ sets $Dtran[D, b] = E$.

States	Dtran	
	Input Symbol	
	a	b
$A = \{0, 1, 2, 4, 7\}$	B	C
$B = \{1, 2, 3, 4, 6, 7, 8\}$	B	D
$C = \{1, 2, 4, 5, 6, 7\}$	B	C
$D = \{1, 2, 4, 5, 6, 7, 9\}$	B	E
$E = \{1, 2, 4, 5, 6, 7, 10\}$	-	-

Conversion of an NFA to a DFA - Example

- ▶ The fifth iteration of the while-loop:
 - ▶ marks E ;
 - ▶ computes $\varepsilon\text{-closure}(\text{move}(E, a)) = \varepsilon\text{-closure}(\text{move}(\{1, 2, 4, 5, 6, 7, 10\}, a)) = \varepsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$;
 - ▶ sets $Dtran[E, a] = B$;
 - ▶ computes $\varepsilon\text{-closure}(\text{move}(E, b)) = \varepsilon\text{-closure}(\text{move}(\{1, 2, 4, 5, 6, 7, 10\}, b)) = \varepsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$;
 - ▶ sets $Dtran[E, b] = C$.

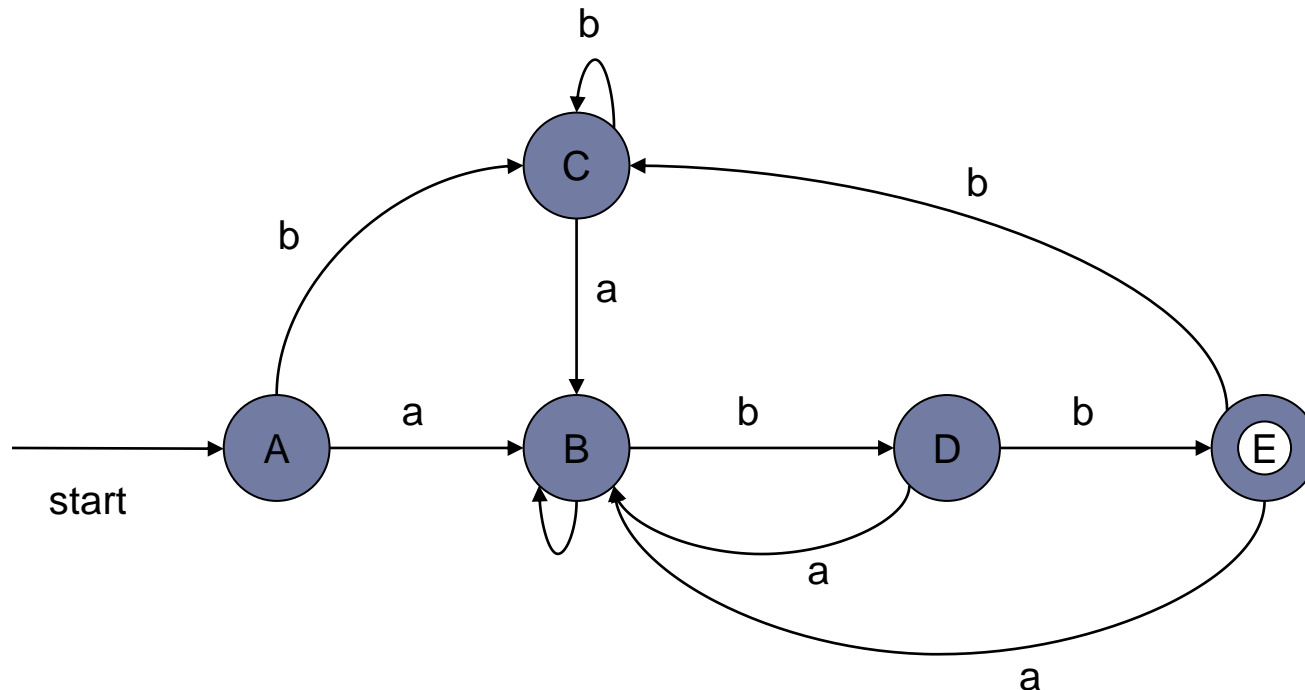
States	Dtran	
	Input Symbol	
	a	b
$A = \{0, 1, 2, 4, 7\}$	B	C
$B = \{1, 2, 3, 4, 6, 7, 8\}$	B	D
$C = \{1, 2, 4, 5, 6, 7\}$	B	C
$D = \{1, 2, 4, 5, 6, 7, 9\}$	B	E
$E = \{1, 2, 4, 5, 6, 7, 10\}$	B	C

BTW...There were actually “only” 2^{11} different subsets of a set of eleven states, and a set, once marked is marked forever.

The final table shows the five different sets of states

Conversion of an NFA to a DFA - Example

- ▶ All five states in *Dstates* are now marked so the Subset Construction Algorithm is done.
- ▶ Here is the resultant DFA...



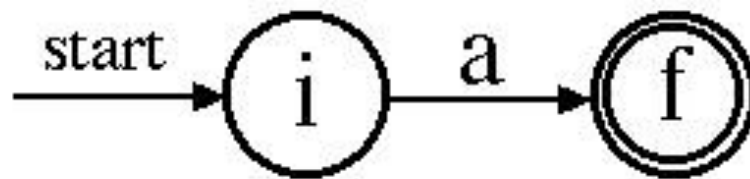
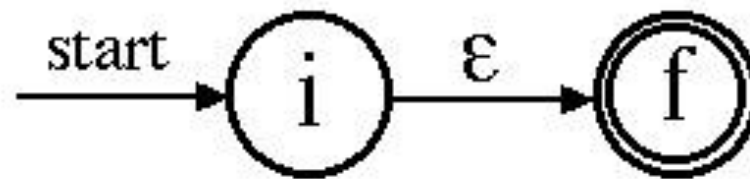
From a Regular Expression to an NFA

- ▶ This section shows ***Thompson's Construction Algorithm*** for constructing an NFA from a regular expression.
- ▶ One can then use the subset construction algorithm to construct a DFA which can then be easily simulated to obtain a recognizer for the regular expression.

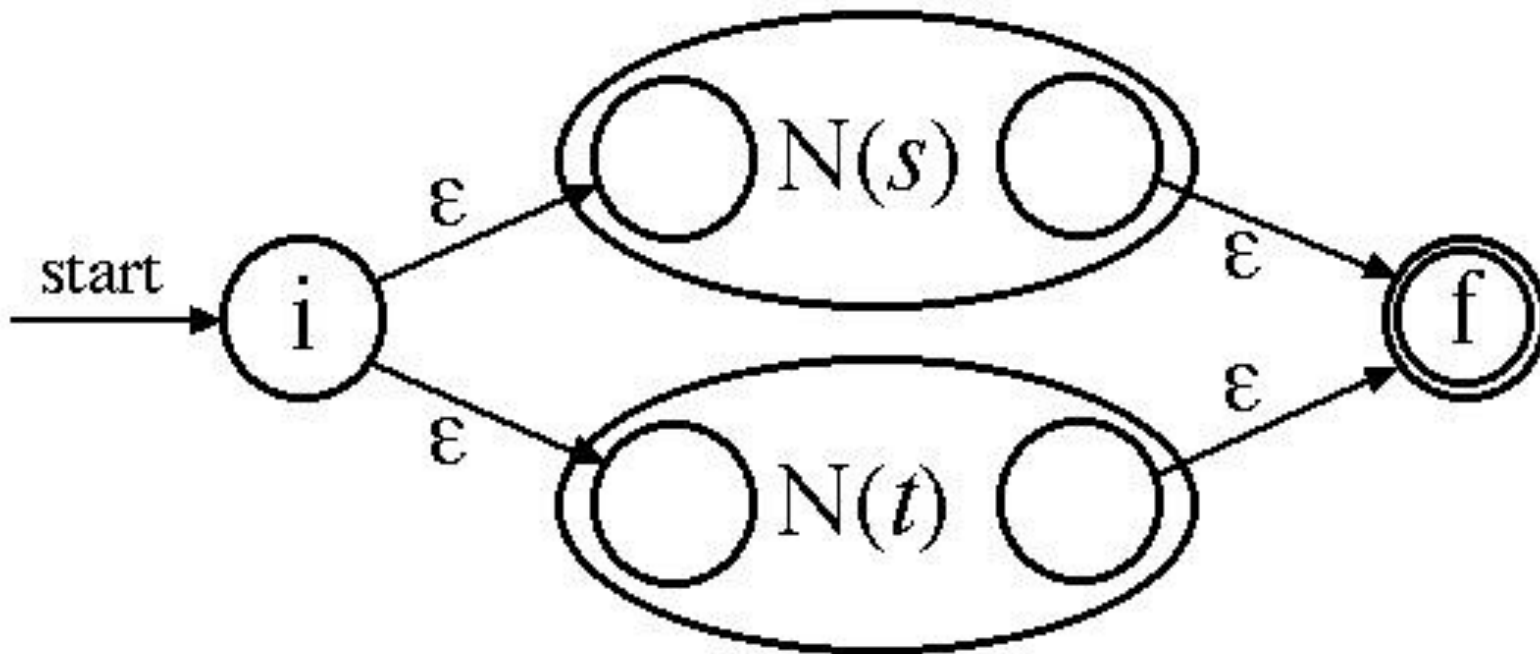
Construction of an NFA from a Regular Expression

- ▶ The construction is guided by the syntax of the regular expression with cases following the cases in the definition of a regular expression.
- ▶ The definition of a regular expression is:
 - ▶ ε is a regular expression that denotes $\{\varepsilon\}$, the set containing just the empty string.
 - ▶ If a is a symbol in the alphabet then the regular expression a denotes $\{a\}$, the set containing just that symbol.
 - ▶ Suppose s and t are regular expressions denoting languages $L(s)$ and $L(t)$, respectively. Then:
 - ▶ a): $(s) \mid (t)$ is a regular expression denoting $L(s) \cup L(t)$;
 - ▶ b): $(s)(t)$ is a regular expression denoting $L(s)L(t)$;
 - ▶ c): $(s)^*$ is a regular expression denoting $L(s)^*$; and
 - ▶ d): (s) is another regular expression denoting $L(s)$.

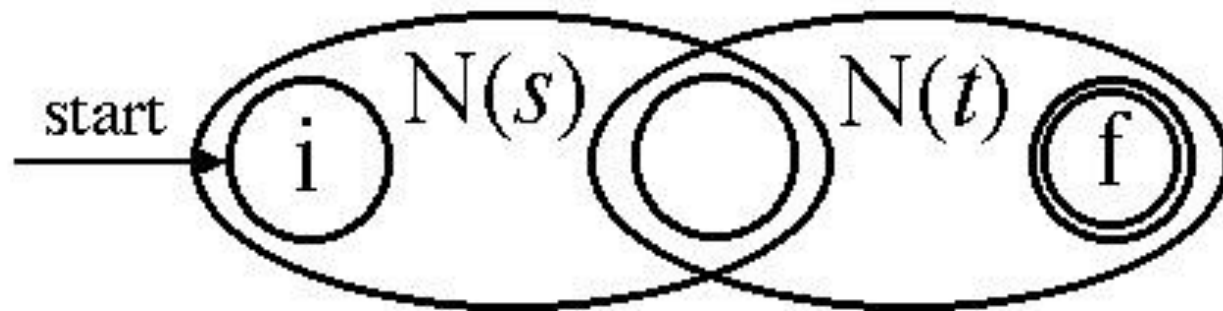
Construction of an NFA from a Regular Expression



Construction of an NFA from a Regular Expression



Construction of an NFA from a Regular Expression



Construction of an NFA from a Regular Expression

