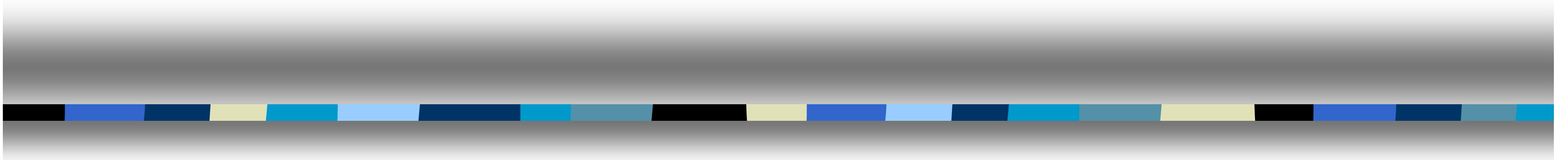
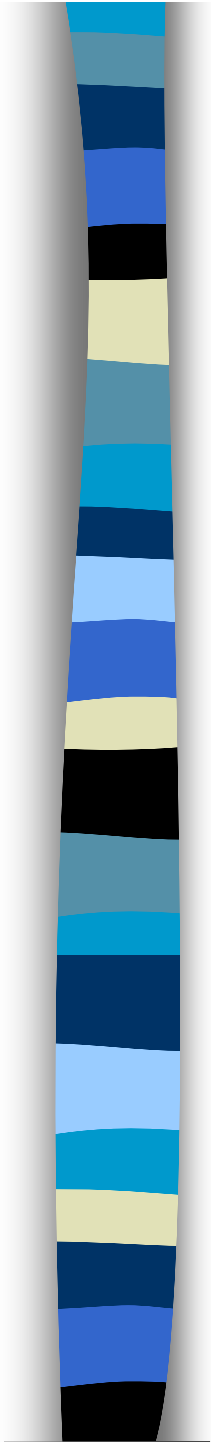
Compiler Design





Lecture-5

Lexical Analyzer

- Tokens
- Attribute
- Patterns
- Lexemes
- Regular Expressions

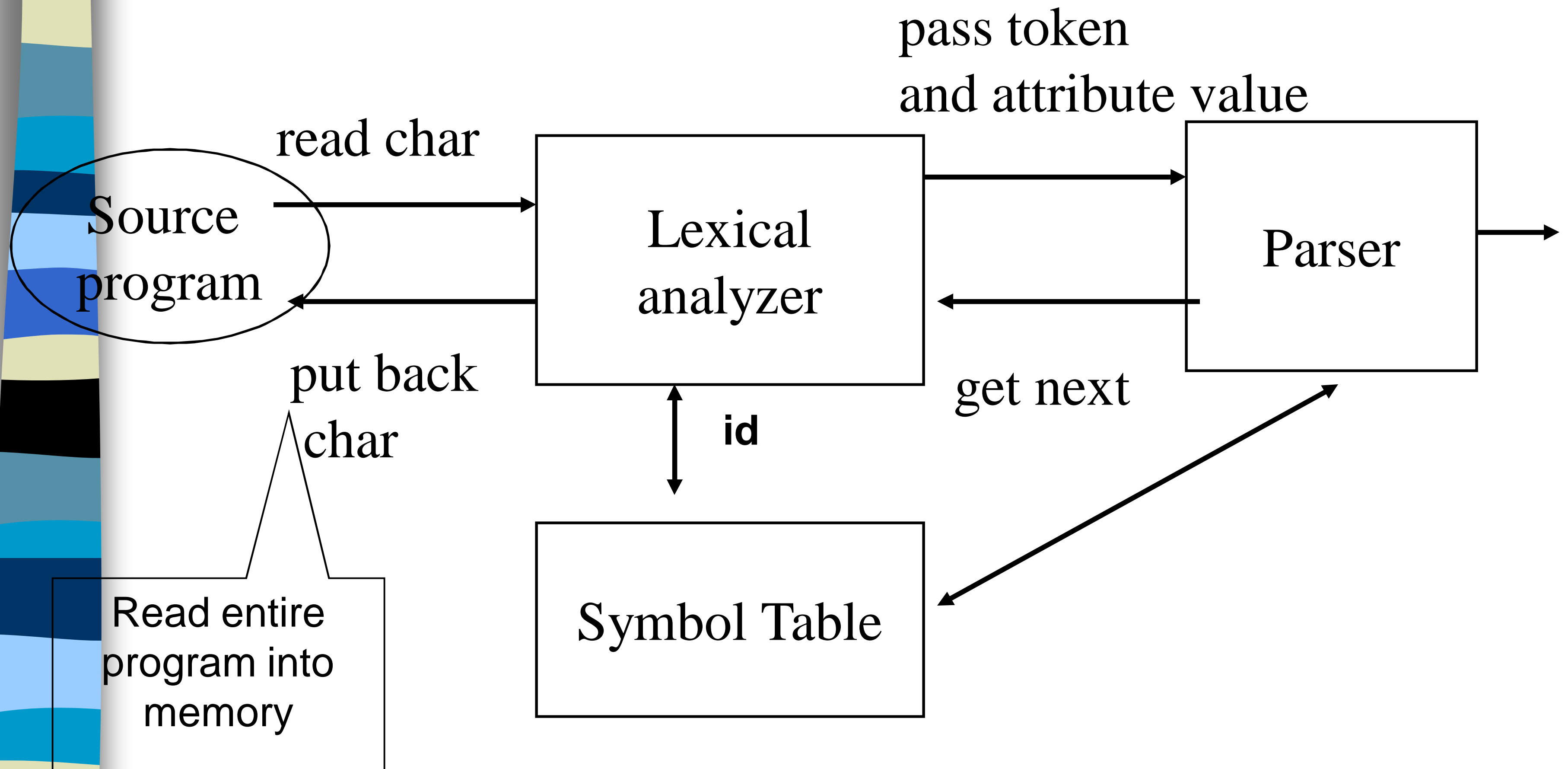
Introduction

- Informal sketch of lexical analysis
  - Identifies tokens in input string
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexemes
  - Regular expressions
  - Examples of regular expressions

Lexical Analyzer

- Functions
  - Grouping input characters into tokens
  - Stripping out comments and white spaces
  - Correlating error messages with the source program
- Issues (why separating lexical analysis from parsing)
  - Simpler design
  - Compiler efficiency
  - Compiler portability (e.g. Linux to Win)

The Role of a Lexical Analyzer



- The input is just a string of characters:

  `\t if (i == j) \n \t \t z = 0;\n \t else \n \t \t z = 1;`
- Goal: Partition input string into substrings
  – Where the substrings are tokens

- A syntactic category
  - In English:
- noun, verb, adjective, …
  - In a programming language:
- Identifier, Integer, Keyword, Whitespace,

What are Tokens For?

- Classify program substrings according to role
- Output of lexical analysis is a stream of tokens . . .which is input to the parser
- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

- Tokens correspond to sets of strings.
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: “else” or “if” or “begin” or …
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

Typical Tokens in a PL

- Symbols: +, -, *, /, =, <, >, ->, …
- Keywords: if, while, struct, float, int, …
- Integer and Real (floating point) literals
  123, 123.45
- Char (string) literals
- Identifiers
- Comments
- White space

Tokens, Patterns and Lexemes

- Pattern: A rule that describes a set of strings
- Token: A set of strings in the same pattern
- Lexeme: The sequence of characters of a token

Token	Sample Lexemes	Pattern
if	if	if
id	abc, n, count,…	letters+digit
NUMBER	3.14, 1000	numerical constant
;	;	;

Token Attribute

- E = C1 ** 10

Token	Attribute
ID	Index to symbol table entry E
=	
ID	Index to symbol table entry C1
**	
NUM	10

- Error detection
- Error reporting
- Error recovery
  - Delete the current character and restart scanning at the next character
  - Delete the first character read by the scanner and resume scanning at the character following it.

Specification of Tokens

- Regular expressions are an important notation for specifying lexeme patterns. While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.

- An alphabet is any finite set of symbols such as letters, digits, and punctuation.
  - The set {0,1) is the binary alphabet
  - If x and y are strings, then the concatenation of x and y is also string, denoted xy, For example, if x = dog and y = house, then xy = doghouse.
  - The empty string is the identity under concatenation; that is, for any string s, ES = SE = s.

- A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
  - In language theory, the terms "sentence" and "word" are often used as synonyms for "string."
  - |s| represents the length of a string s, Ex: banana is a string of length 6
  - The empty string, is the string of length zero.

Strings and Languages (cont.)

- A language is any countable set of strings over some fixed alphabet.

Def. Let Σ be a set of characters. A *language over* Σ is a set of strings of characters drawn from Σ

- Let L = {**A,** . . . , Z}, then{“A”,”B”,”C”, “BF”…,”ABZ”,…] is consider the language defined by L
- Abstract languages like Φ, the empty set, or

{ε},the set containing only the empty string, are languages under this definition.

Terms for Parts of Strings

The following string-related terms are commonly used:

1. A *prefix* of string s is any string obtained by removing zero or more symbols from the end of s. For example, **ban**, **banana**, and ϵ are prefixes of **banana**.
2. A *suffix* of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, **nana**, **banana**, and ϵ are suffixes of **banana**.
3. A *substring* of s is obtained by deleting any prefix and any suffix from s. For instance, **banana**, **nan**, and ϵ are substrings of **banana**.
4. The *proper* prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ϵ or not equal to s itself.
5. A *subsequence* of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, **baan** is a subsequence of **banana**.

Operations on Languages

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Example:

Let L be the set of letters {A, B, . . . , Z, a, b, . . . , z) and let D be the set of digits {0,1,.. .9).

L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits.

other languages can be constructed from L and D, using the operators illustrated above

1. $L \cup D$ is the set of letters and digits - strictly speaking the language with 62 (52+10) strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.(10×52).

   Ex: A1, a1,B0,etc
3. L^4 is the set of all 4-letter strings. (ex: aaba, bcef)

4. L^* is the set of all strings of letters, including e, the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

- The standard notation for regular languages is regular expressions.
- Atomic regular expression:
  - Single character

    $$'c' = \{"c"\}$$

  - Epsilon

    $$\varepsilon = \{""\}$$

- Compound regular expression:
  - Union

    $$A + B = \{s \mid s \in A \text{ or } s \in B\}$$

  - Concatenation

    $$AB = \{ab \mid a \in A \text{ and } b \in B\}$$

  - Iteration

    $$A^* = \bigcup_{i \geq 0} A^i \quad \text{where } A^i = A \ldots i \text{ times} \ldots A$$

Cont.

$$
\begin{aligned}
L(\varepsilon) &= \{""\} \\
L('c') &= \{"c"\} \\
L(A+B) &= L(A) \cup L(B) \\
L(AB) &= \{ab \mid a \in L(A) \text{ and } b \in L(B)\} \\
L(A^*) &= \bigcup_{i \geq 0} L(A^i)
\end{aligned}
$$

larger regular expressions are built from smaller ones. Let **r** and **s** are regular expressions denoting languages L(r) and L(s), respectively.

1. (r) | (s) is a regular expression denoting the language L(r) U L(s).
2. (r) (s) is a regular expression denoting the language L(r) L(s) .
3. (r) * is a regular expression denoting (L (r)) * .
4. (r) is a regular expression denoting L(r). This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

for example, we may replace the regular expression (a) | ((b) * (c)) by a| b^*c.

Example 3.4: Let $\Sigma = \{a, b\}$.

1. The regular expression $\mathbf{a}|\mathbf{b}$ denotes the language $\{a, b\}$.

2. $(\mathbf{a}|\mathbf{b})(\mathbf{a}|\mathbf{b})$ denotes $\{aa, ab, ba, bb\}$, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is $\mathbf{aa}|\mathbf{ab}|\mathbf{ba}|\mathbf{bb}$.

3. $\mathbf{a}^*$ denotes the language consisting of all strings of zero or more a's, that is, $\{\epsilon, a, aa, aaa, \ldots\}$.

4. $(\mathbf{a}|\mathbf{b})^*$ denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. Another regular expression for the same language is $(\mathbf{a}^*\mathbf{b}^*)^*$.

5. $\mathbf{a}|\mathbf{a}^*\mathbf{b}$ denotes the language $\{a, b, ab, aab, aaab, \ldots\}$, that is, the string a and all strings consisting of zero or more a's and ending in b.

Regular Definition

- C identifiers are strings of letters, digits, and underscores. The regular definition for the language of C identifiers.
  - *Letter* → A | B / *C*/…/ Z / a | b | … |z| -
  - *digit* → *0/1/2 /… / 9*
  - *id* → *letter(letter / digit)**
- Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition
  - *digit* → *0/1/2 /… / 9*
  - *digits* → *digit digit**
  - *optionalFraction* → *.digits* / *ε*
  - *optionalExponent* → (E(+ |- / *ε*) *digits*) / *ε*
  - *number* → *digits optionalFraction optionalExponent*

RECOGNITION OF TOKENS

- Given the grammar of branching statement:

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

The **terminals** of the grammar, which are **if, then**, **else**, **relop**, **id**, and **number**, are the names of tokens as used by the lexical analyzer.
The lexical analyzer also has the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^+$$

- The patterns for the given tokens:

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^+ \\
number & \rightarrow & digits\ (.\ digits)?\ (\ \texttt{E}\ \texttt{[+-]}?\ digits\ )? \\
letter & \rightarrow & \texttt{[A-Za-z]} \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^* \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

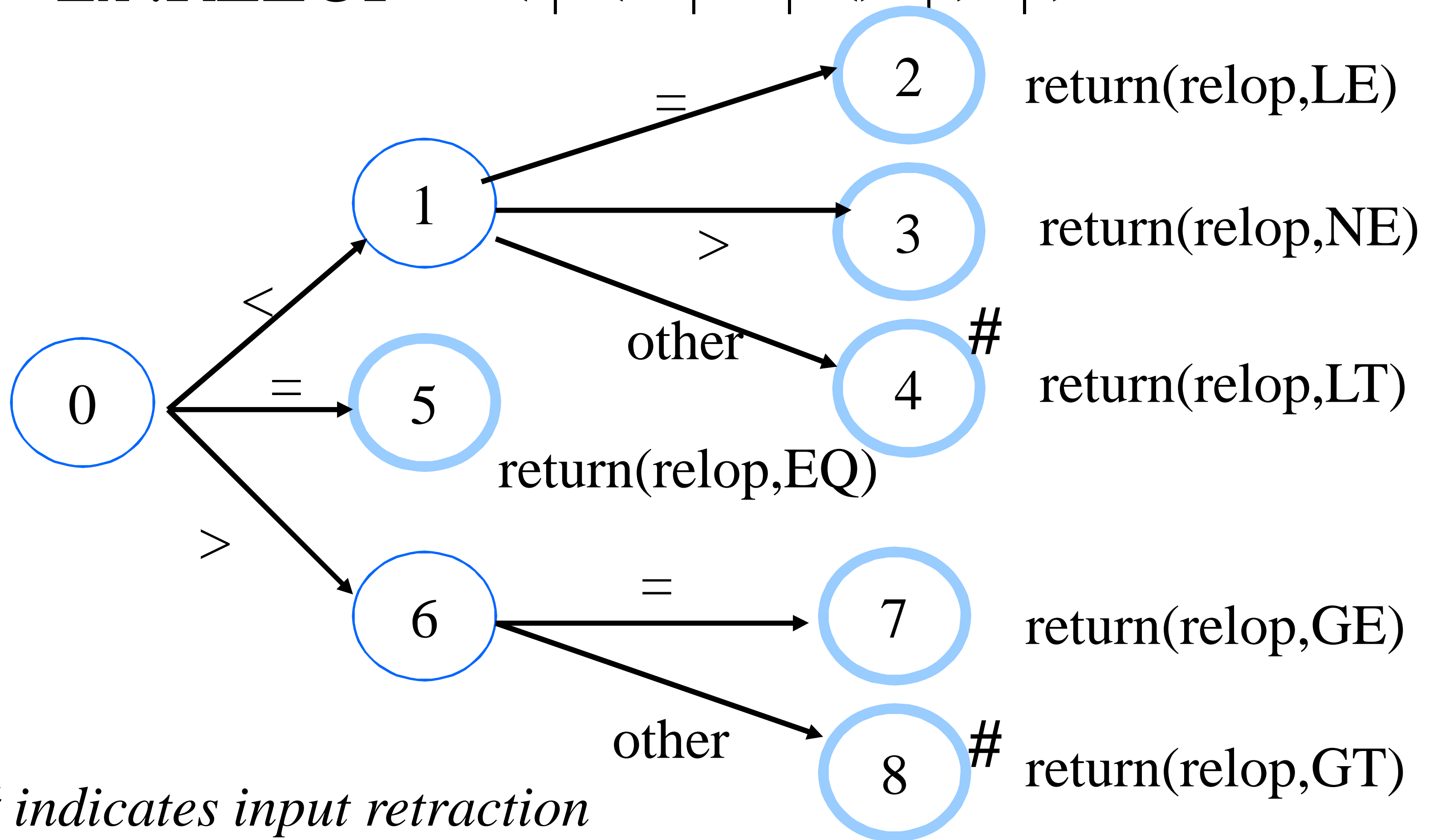
$r?$ is equivalent to $r|\epsilon$

Tokens, their patterns, and attribute values

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Recognition of Tokens: Transition Diagram

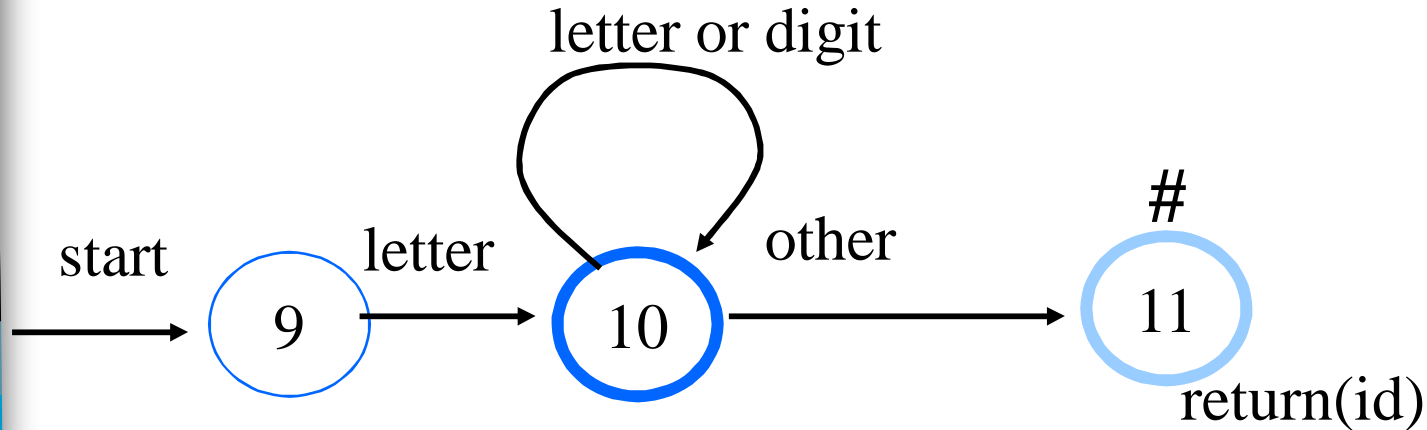
Ex :RELOP = < | <= | = | <> | > | >=



Recognition of Identifiers

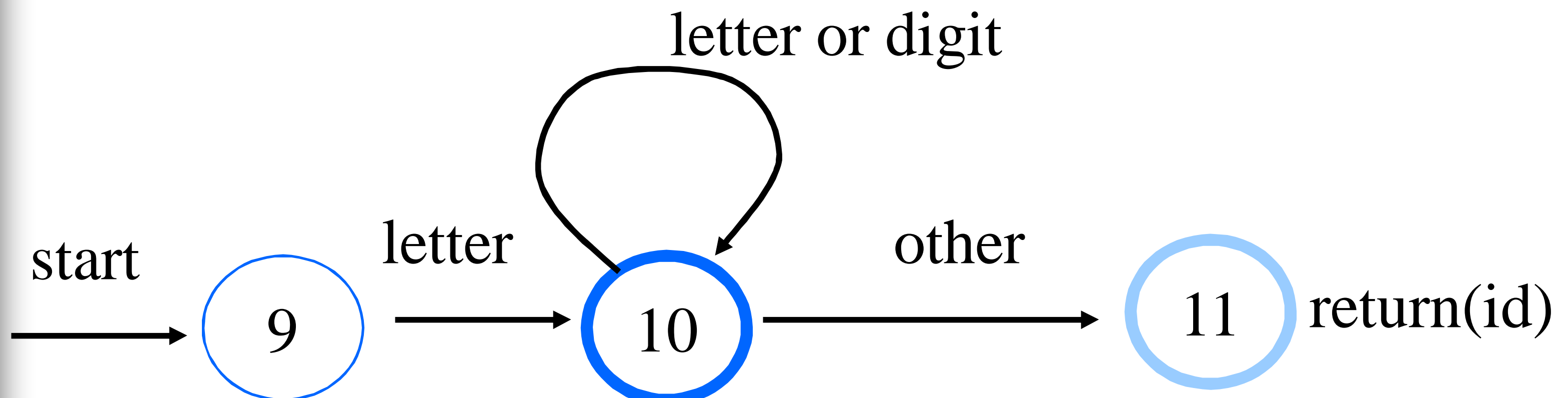
- Ex2: ID = letter(letter | digit) *

Transition Diagram:



 # indicates input retraction

Mapping transition diagrams into C code




```
switch (state) {
case 9:
        if (isletter( c) ) state = 10; else state =
failure();
        break;
case 10:        c = nextchar();
                if (isletter( c) || isdigit( c) ) state = 10;  else state 11
case 11: retract(1); insert(id); return;
```

Lexical analyzer loop

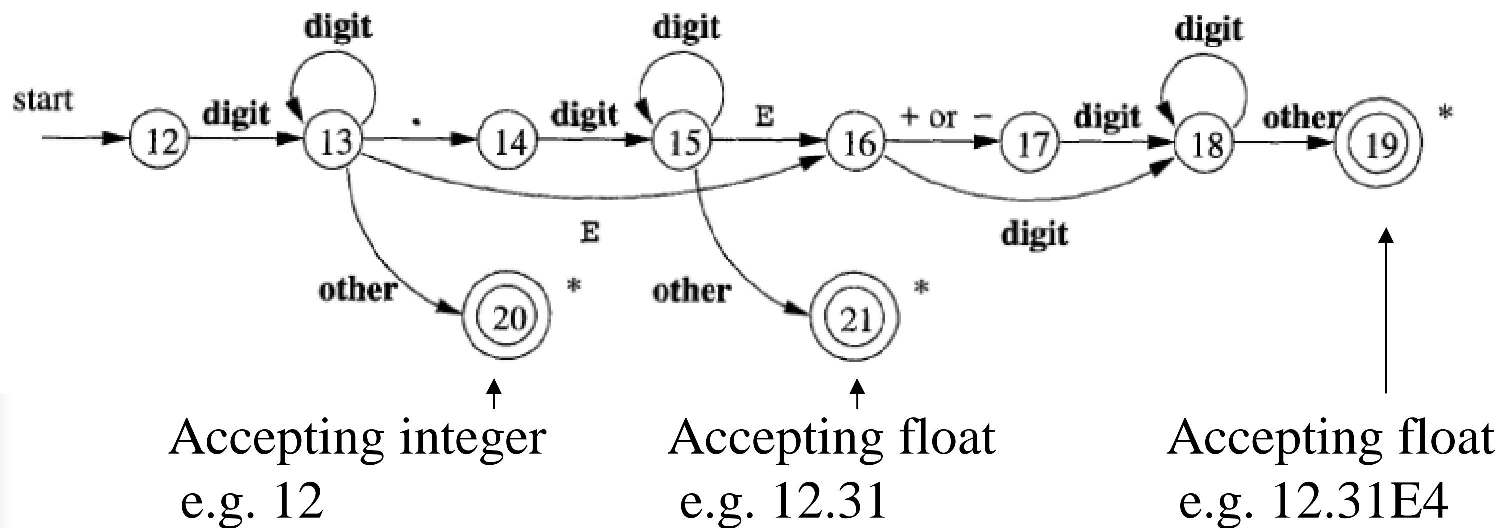
```
Token nexttoken() {
while (1) {
    switch (state) {
    case 0:       c = nextchar();
                  if (c is white space) state = 0;
                  else if (c == ‘<‘) state = 1;
                  else if (c == ‘=‘) state = 5;
                  …
    case 9:       c = nextchar();
                  if (isletter( c) ) state = 10; else state =fail();
                  break;
    case 10:      ….
    case 11:      retract(1); insert(id);
                  return;
```

Recognition of Reserved Words

- Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.
  - Create separate transition diagrams for each keyword; the transition diagram for the reserved word then

The transition diagram for token number

Multiple accepting state



Accepting integer
e.g. 12

Accepting float
e.g. 12.31

Accepting float
e.g. 12.31E4

- Two ways to implement:
  - Implement it as multiple regular expressions.

    each with its own start and accepting states. Starting with the longest one first, if failed, then change the start state to a shorter RE, and re-scan. See example of Fig. 3.15 and 3.16 in the textbook.
  - Implement it as a transition diagram with multiple accepting states.

    When the transition arrives at the first two accepting states, just remember the states, but keep advancing until a failure is occurred. Then backup the input to the position of the last accepting state.

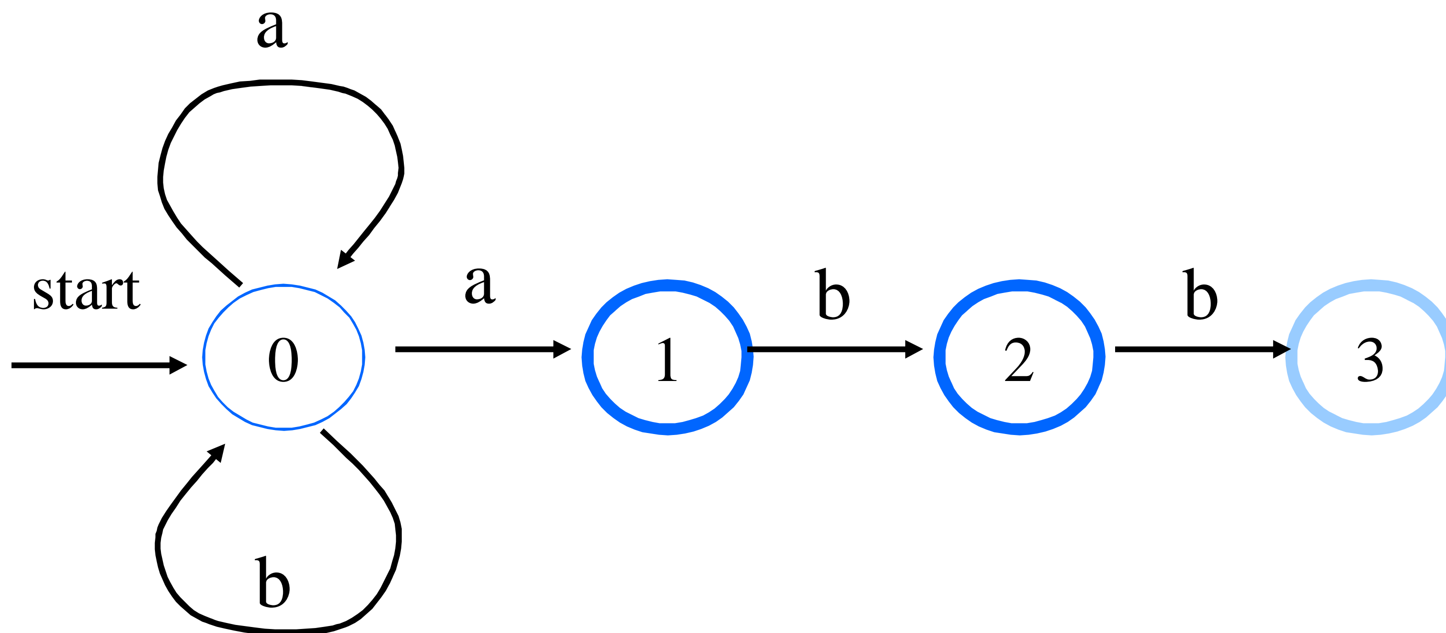
Lexical Analyzer Generator

- Lexical analyzer generator is to transform RE into a stade transition table (i.e. Finite Automation)
- Theory of such tralsformation
- Some practical consideration

Finite Automata

- Transition diagram is finite automation
- Nondeterministic Finite Automation (NFA)
  - A set of states
  - A set of input symbols
  - A transition function, *move*(), that maps state-symbol pairs to sets of states.
  - A start state S_0
  - A set of states F as accepting (Final) states.

Example



The set of states = {0,1,2,3}
Input symbol = {a,b}
Start state is S0, accepting state is S3

Transition Function

- Transition function can be implemented as a transition table.

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

Simulation of NFA

- Given an NFA *N* and an input string *x*, determine whether N accepts *x*

S:= e-closure({s0}) ; a := nextchar;

While a <> eof do begin

    S:= e-closure(move(S

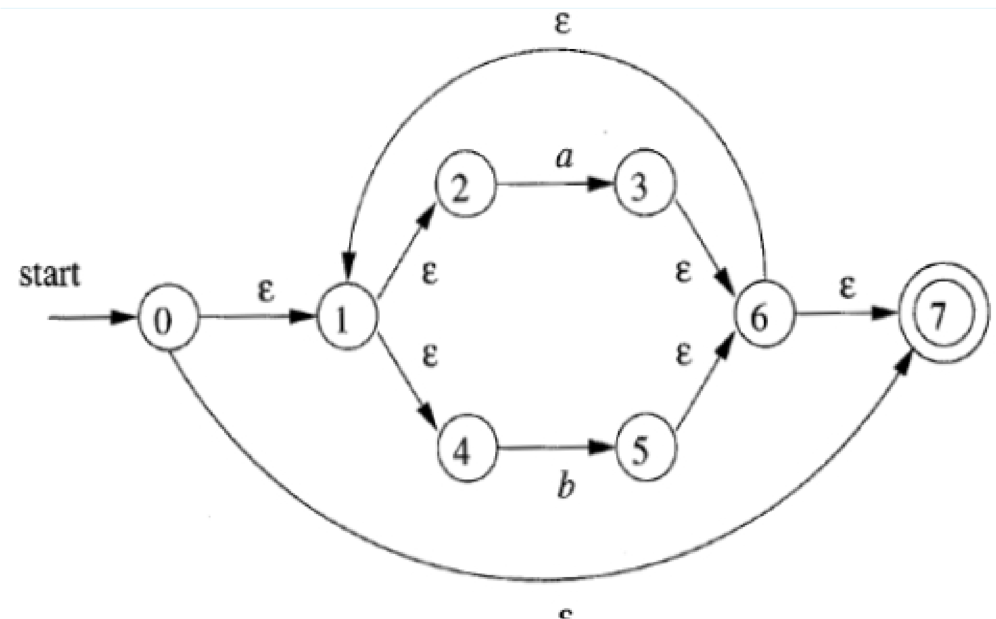
    a:= nextchar;

end

if (an accepting state s in S

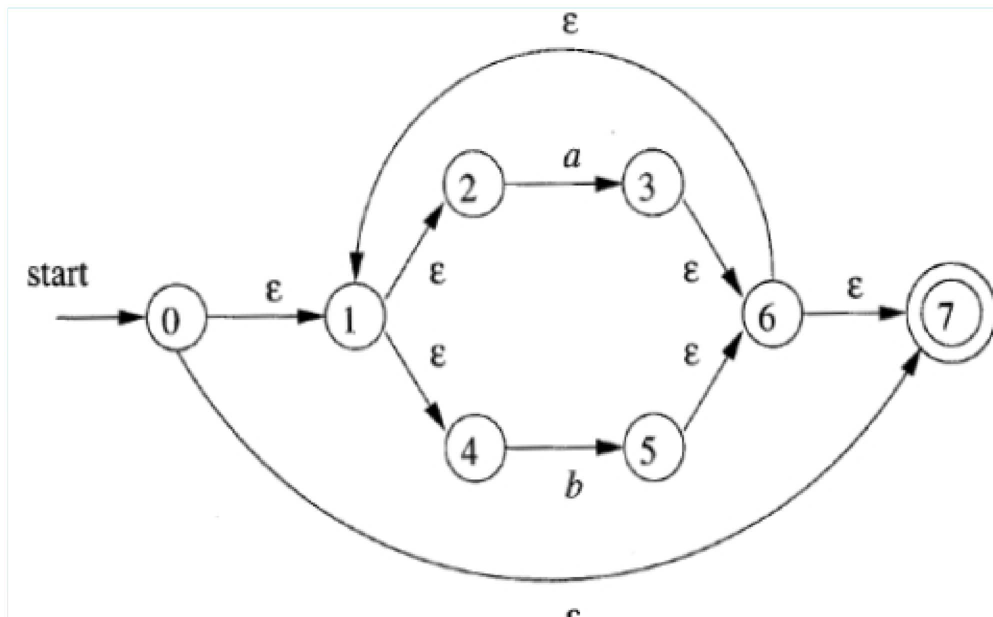
  return(yes)

otherwise return (no)

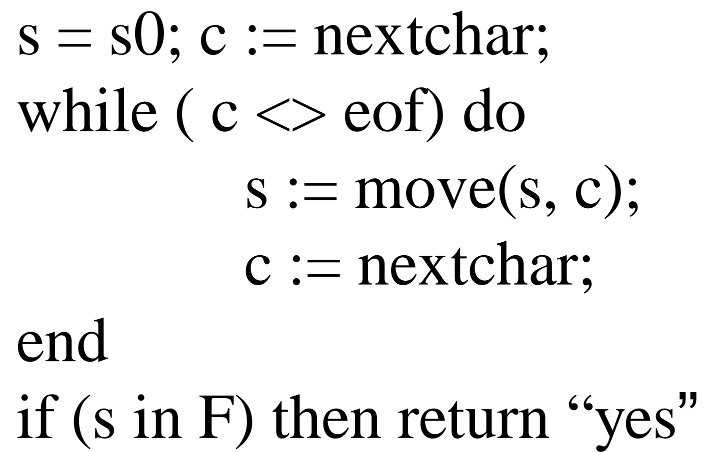


Computing the ε-closure (T)

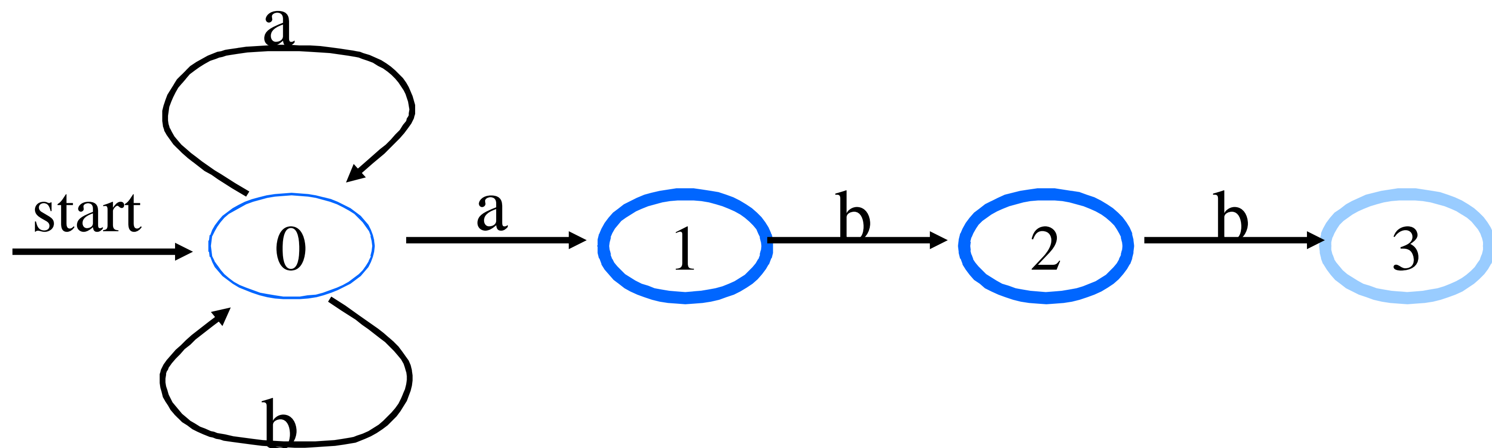
push all states of T onto *stack*;
initialize ϵ-*closure*(T) to T;
while (*stack* is not empty) {
    pop t, the top element, off *stack*;
    for (each state u with an edge from t to u labeled ϵ)
        if (u is not in ϵ-*closure*(T)) {
            add u to ϵ-*closure*(T);
            push u onto *stack*;
        }
}



- Non-deterministic Finite Automata (NFA)
  - An NFA accepts an input string *x* iff there is a path in the transition graph from the start state to some accepting (final) states.
  - ThE language defined by an NFA is the set of strings it accepts
- Deterministic Finite Automata (DFA)
- A DFA is a special case of NFA in which
  - There is no e-transition
  - Always have unique successor states.



```
s = s0; c := nextchar;
while ( c <> eof) do
        s := move(s, c);
        c := nextchar;
end
if (s in F) then return “yes”
```

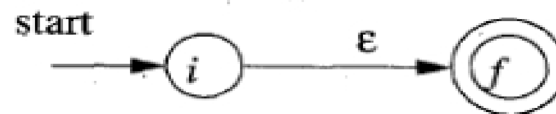


- For each kind of RE, there is a corresponding NFA To convert any regular expression to a NFA that defines the same language.

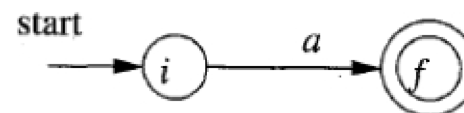
- The algorithm is syntax-directed, in the sense that it works recursively up the parse tree for the regular expression.

- For each sub-expression the algorithm constructs an NFA with a single accepting state.

- **INPUT: A** regular expression r over alphabet Σ.
- **OUTPUT:** An NFA **N** accepting L(r).
- **Method**: Begin by parsing r into its constituent sub-expressions. The rules for constructing an NFA consist of basis rules for handling sub-expressions with no operators, and inductive rules for constructing larger NFA's from the NFA's for the immediate sub-expressions of a given expression.
  - For expression e construct the NFA

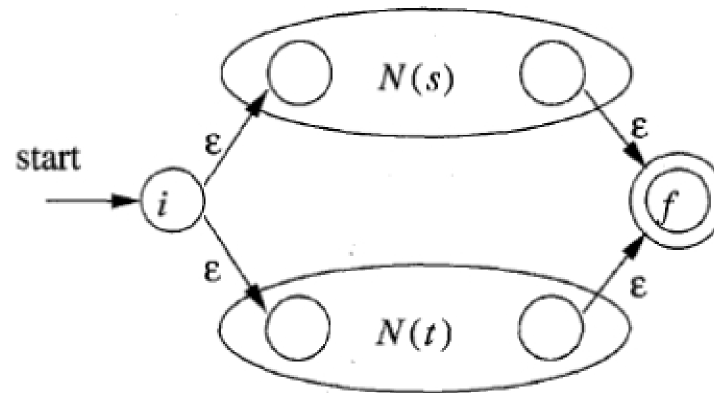


  - For any sub-expression a in C, construct the NFA

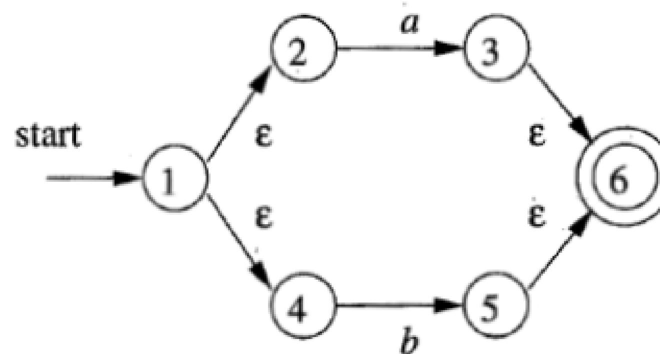


RE to NFA (cont.)

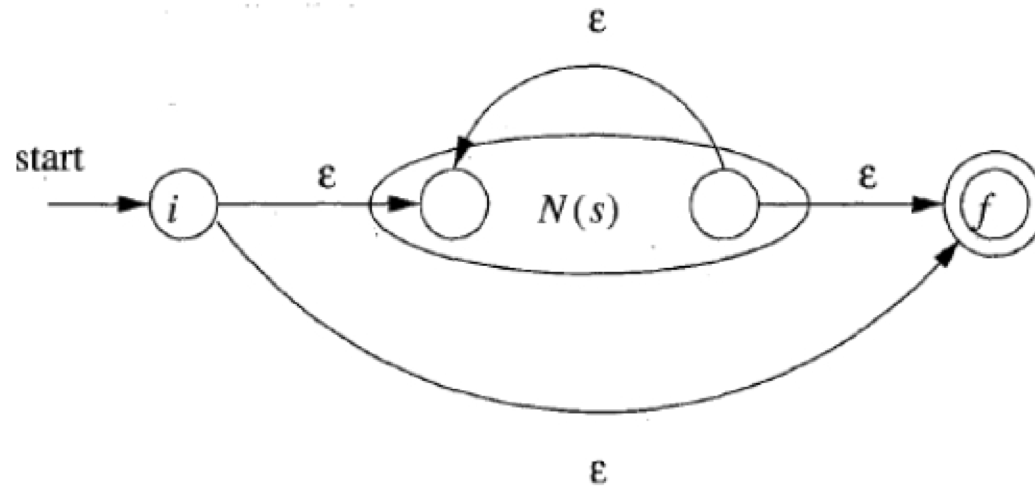
- NFA for the union of two regular expressions



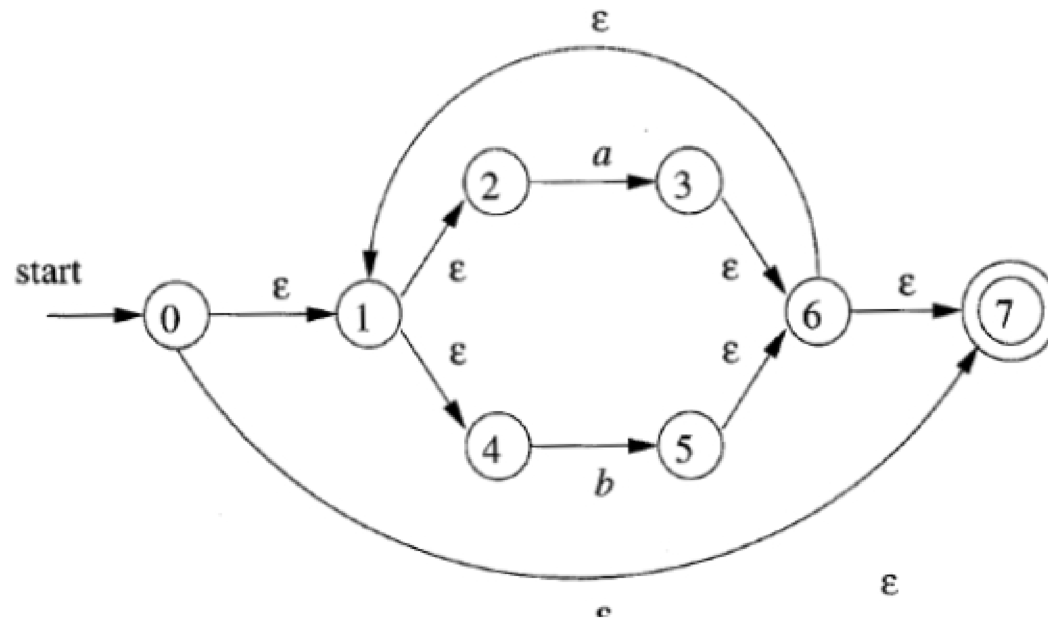
- Ex: $a|b$



NFA for the closure of a regular expression

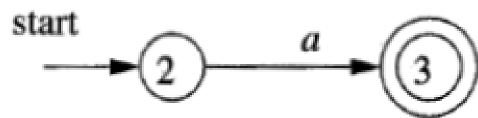


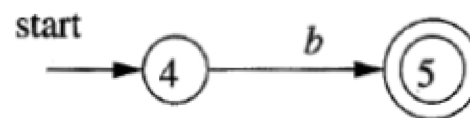
(a|b)*

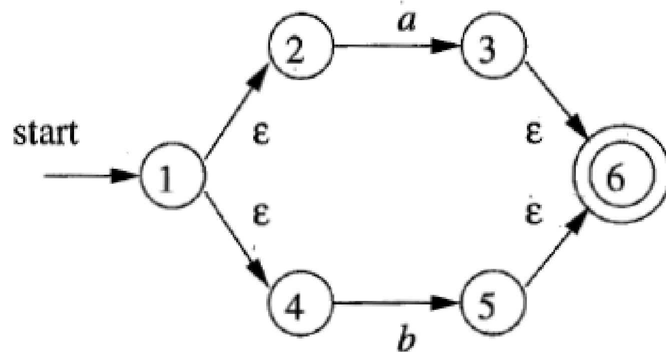


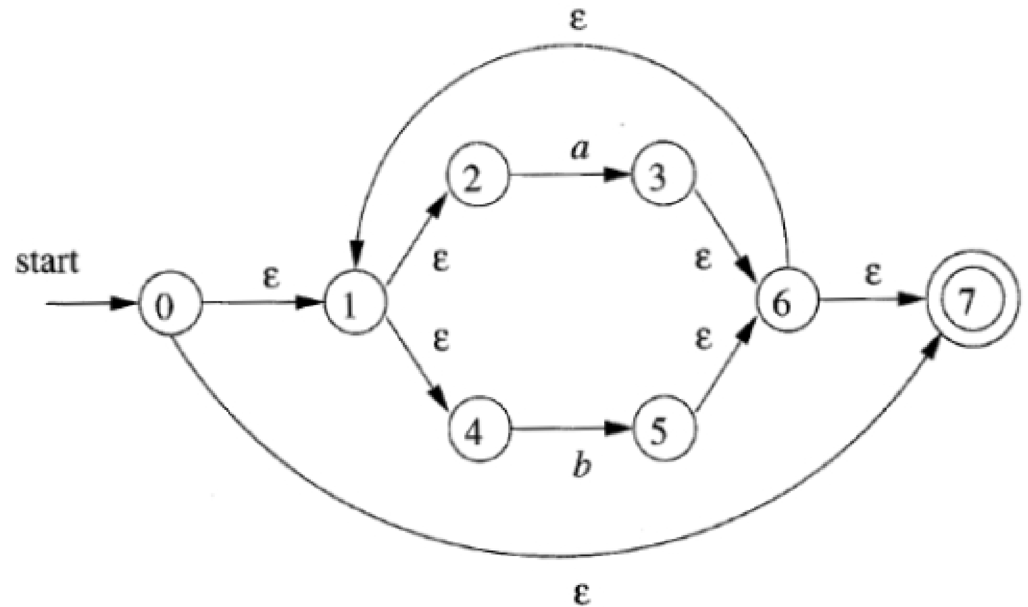
Example: Constructing NFA for regular expression r= (a|b)*abb

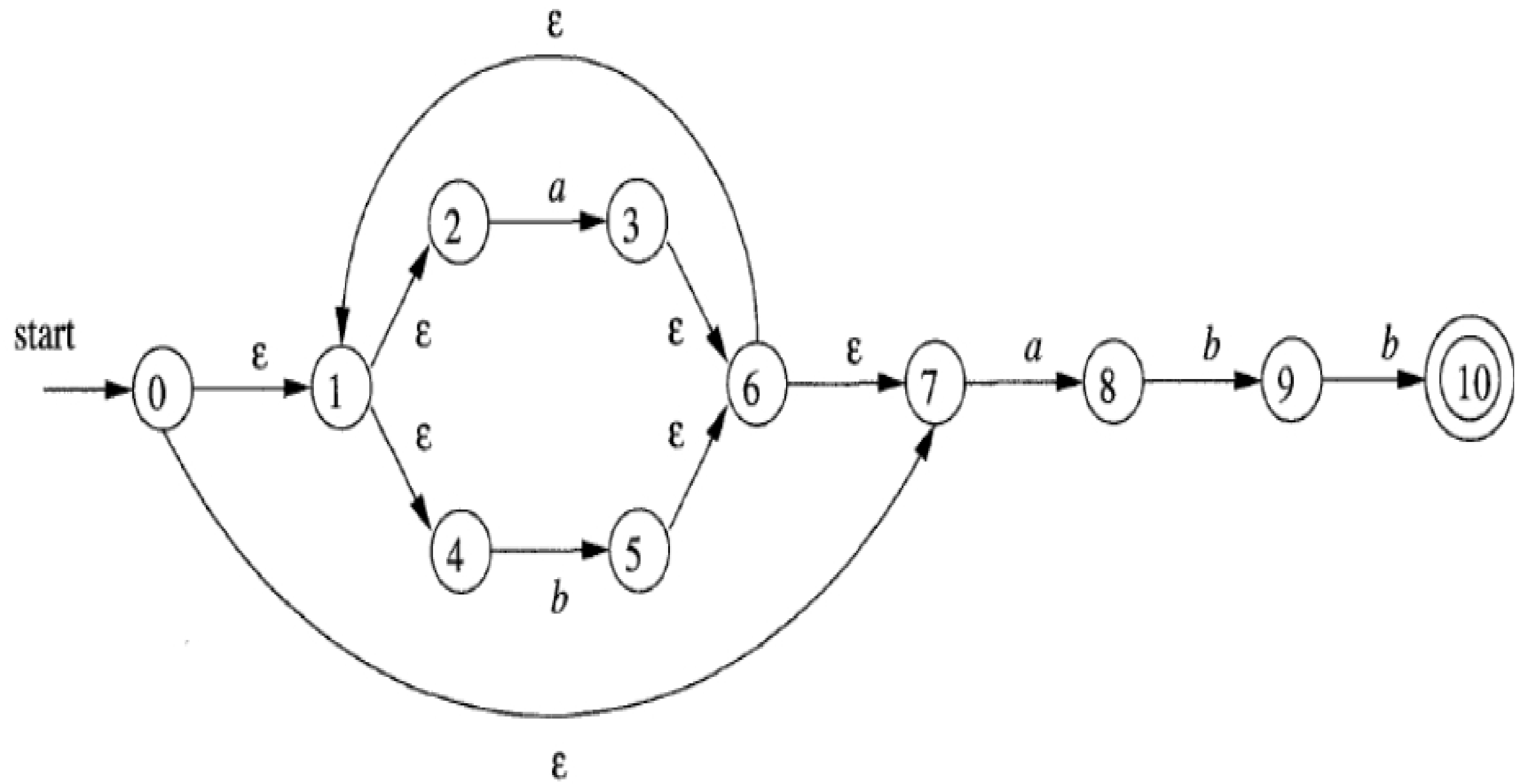
Step 1: construct a, b
Step 2: constructing a | b
Step3: construct (a|b)*
Step4: concat it with a, then, b, then b











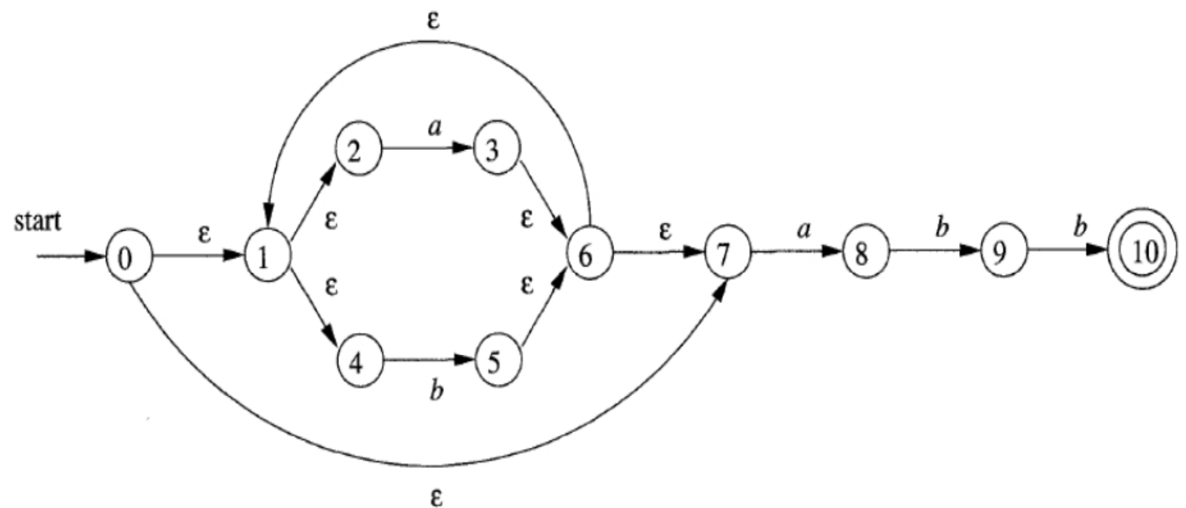
start
ε
ε
a
ε
ε
ε
ε
b
ε
a
b
b
ε
0
1
2
3
4
5
6
7
8
9
10

Conversion of NFA to DFA

- Why?
  - DFA is difficult to construct directly from RE’s
  - NFA is difficult to represent in a computer program and inefficient to compute
- Conversion algorithm: subset construction
  - The idea is that each DFA state corresponds to a set of NFA states.
  - After reading input a1, a2, …, an, the DFA is in a state that represents the subset T of the states of the NFA that are reachable from the start state.

Subset Construction Algorithm

```
Dstates := e-closure (s0)
While there is an unmarked state T in Dstates do
begin
  mark T;
  for each input symbol a do
  begin
     U := e-closure ( move(T,a) );
     if  U is not in Dstates then
        add U as an unmarked state to Drtates;
        Dtran [T, a] := U;
     end
end
```



Example NFA to DFA

- The start state A of the equivalent DFA is ε-closure(0),
  - A = {0,1,2,4,7},
- since these are exactly the states reachable from state 0 via a path all of whose edges have label ε. Note that a path can have zero edges, so state 0 is reachable from itself by an ε -labeled path.
- The input alphabet is {a, b). Thus, our first step is to mark A and compute

  Dtran[A, a] = ε -closure(moue(A, a)) and

  Dtran[A, b] = ε **-** closure(moue(A, b)) .
- Among the states 0, 1, 2, 4, and 7, only 2 and 7 have transitions on a, to **3** and 8, respectively. Thus,

  move(A, a) = {3,8). Also, ε -closure({3,8})= {1,2,3,4,6,7,8), so we conclude call this set B,

$$Dtran[A,a] = \epsilon\text{-}closure(move(A,a)) = \epsilon\text{-}closure(\{3,8\}) = \{1,2,3,4,6,7,8\}$$

let Dtran[A, a] = B

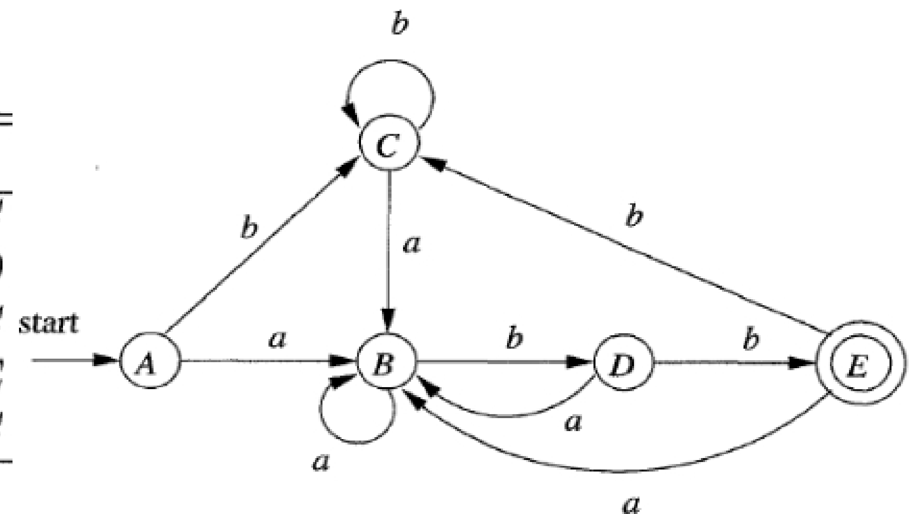
NFA to DFA (cont.)

- compute Dtran[A, b]. Among the states in A, only 4 has a transition on *b,* and it goes to 5.

$$Dtran[A, b] = \epsilon\text{-}closure(\{5\}) = \{1, 2, 4, 6, 7\}$$

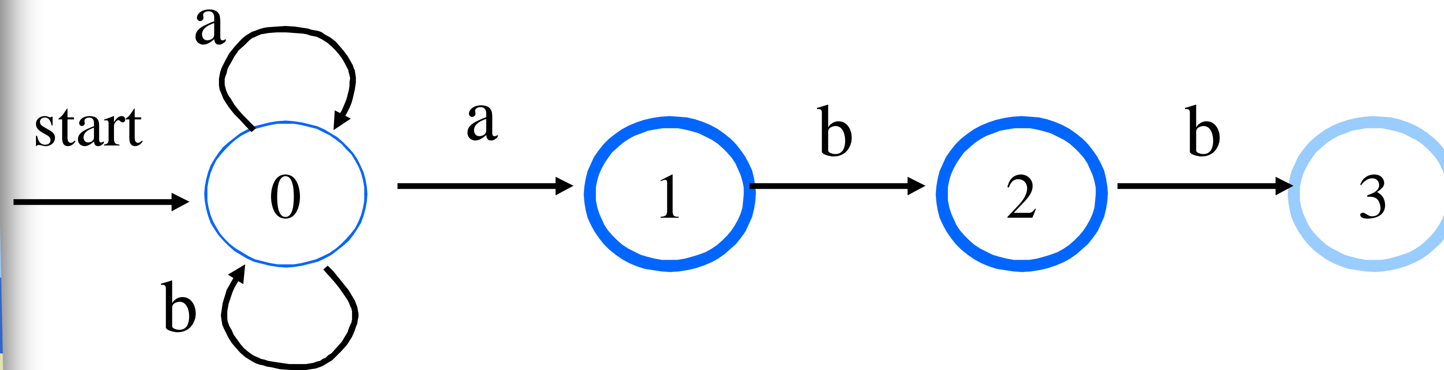
- Call it C
- If we continue this process with the unmarked sets B and C, we eventually reach a point where all the states of the DFA are marked.

NFA State	DFA State	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,3,5,6,7,10\}$	E	B	C



igure 3.35: Transition table *Dtran* for DFA *D*

EX(2) NFA to DFA conversion



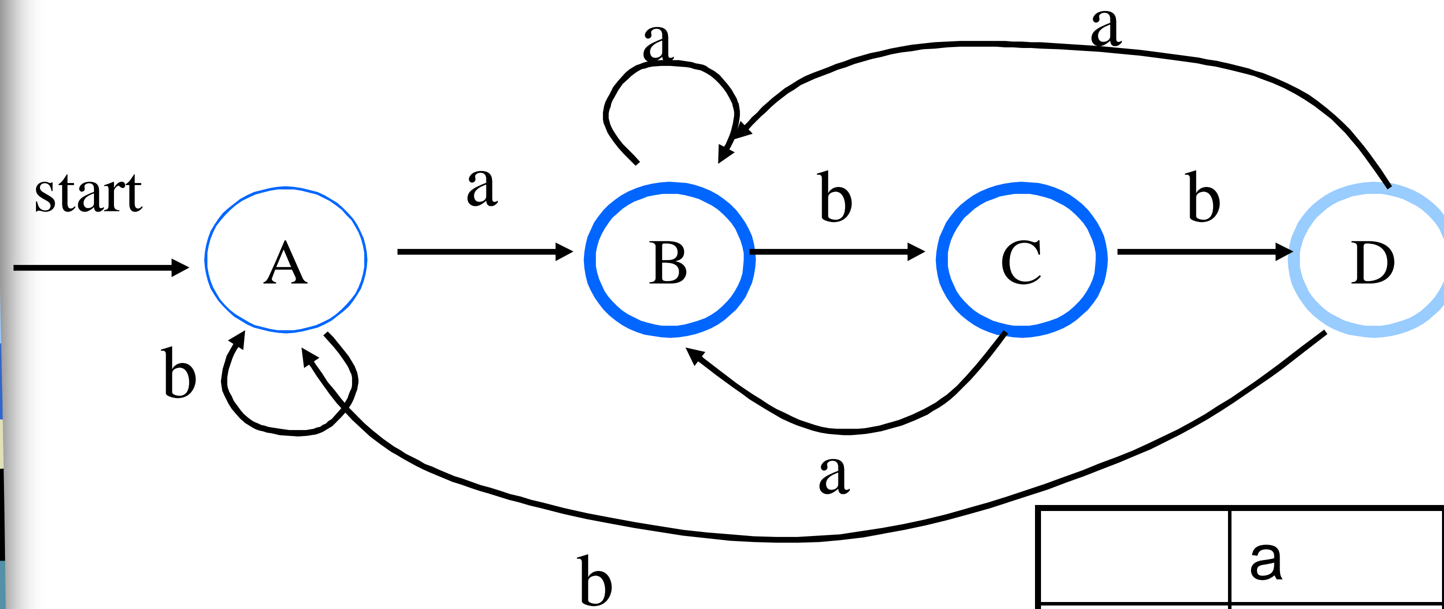
(0,a) = {0,1}
(0,b) = {0}
({0,1}, a) = {0,1}
({0,1}, b) = {0,2}
({0,2}, a) = {0,1}
({0,2}, b) = {0,3}

New states

A = {0}
B = {0,1}
C = {0,2}
D = {0,3}

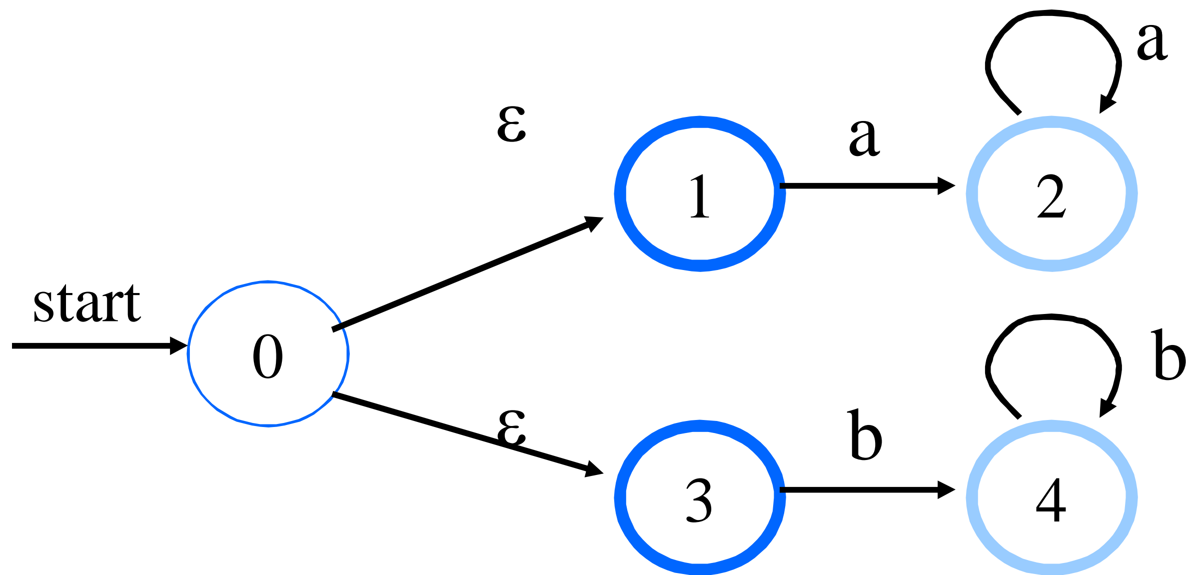
	a	b
A	B	A
B	B	C
C	B	D
D	B	A

NFA to DFA conversion (cont.)



	a	b
A	B	A
B	B	C
C	B	D
D	B	A

NFA to DFA conversion (cont.)

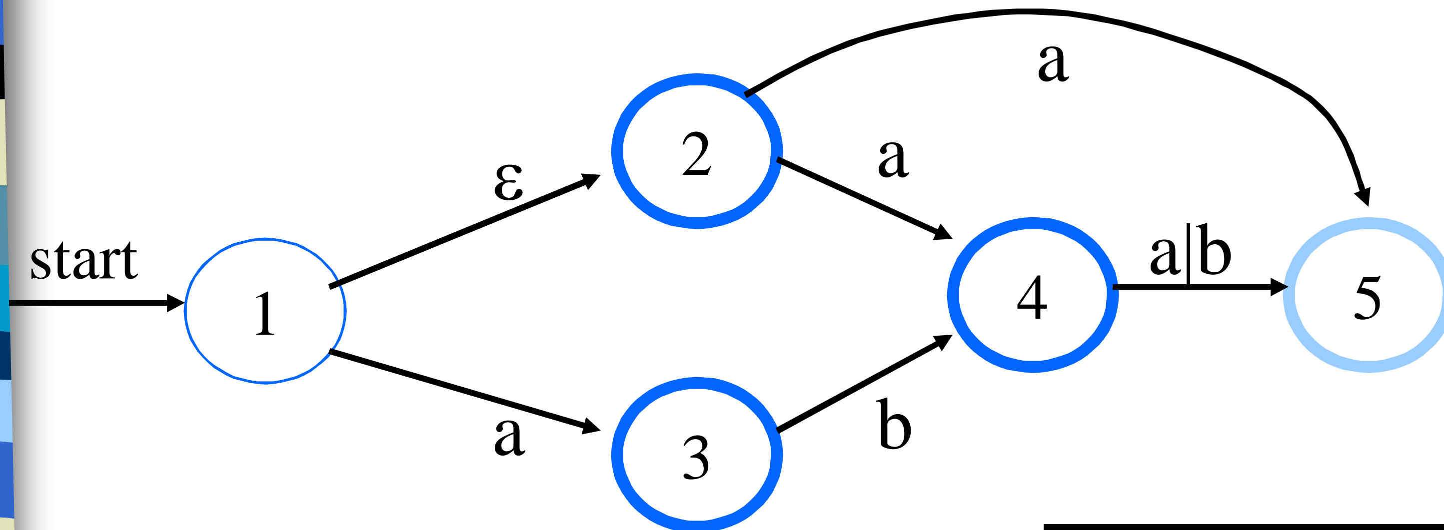


How about e-transition?

Due to e-transitions, we must compute e-closure(S) which is the set of NFA states reachable from NFA state S on e-transition, and e-closure(T) where T is a set of NFA states.

Example: e-closure (0) = {1,3}

Example

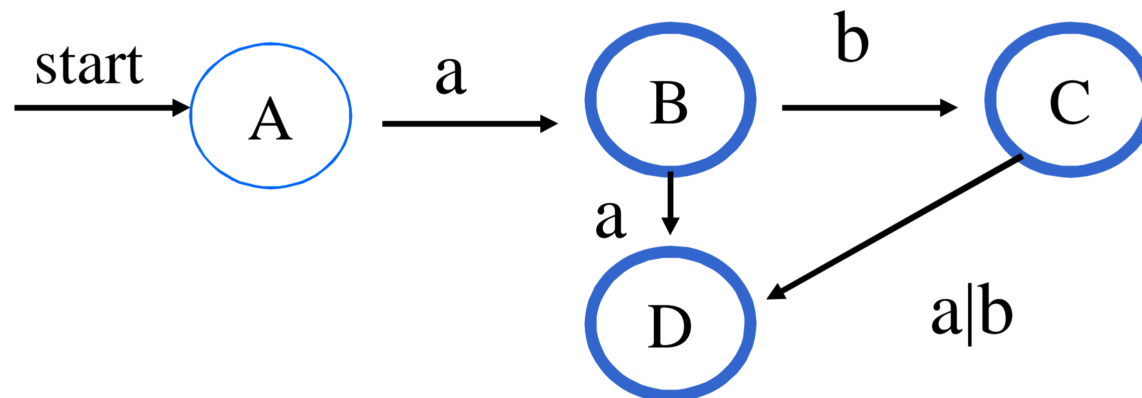


Dstates := ε-closure(1) = {1,2}
U:= ε-closure (move({1,2}, a)) = {3,4,5}
Add {3,4,5} to Dstates
U:= ε-closure (move({1,2}, b)) = { }
ε-closure (move({3,4,5}, a)) = {5}
ε-closure (move({3,4,5}, b)) = {4,5}
ε-closure (move({4,5}, a)) = {5}
ε-closure (move({4,5}, b)) = {5}

	a	b
A{1,2}	B	--
B{3,4,5}	D	C
C{4,5}	D	D
D{5}	--	--

DFA after conversion

	a	b
A{1,2}	B	--
B{3,4,5}	D	C
C{4,5}	D	D
D{5}	--	--



Minimization of DFA

- If we implement a lexical analyzer as a DFA, we would generally prefer a DFA with as few states as possible, since each state requires entries in the table that describes the lexical analyzer.
- There is always a unique minimum state DFA for any regular language. Moreover, this minimum-state DFA can be constructed from any DFA for the same language by grouping sets of equivalent states.

Algorithm 3.39 : Minimizing the number of states of a DFA.

INPUT: A DFA **D** with set of states S, input alphabet Σ, start state 0, and set of accepting states F.

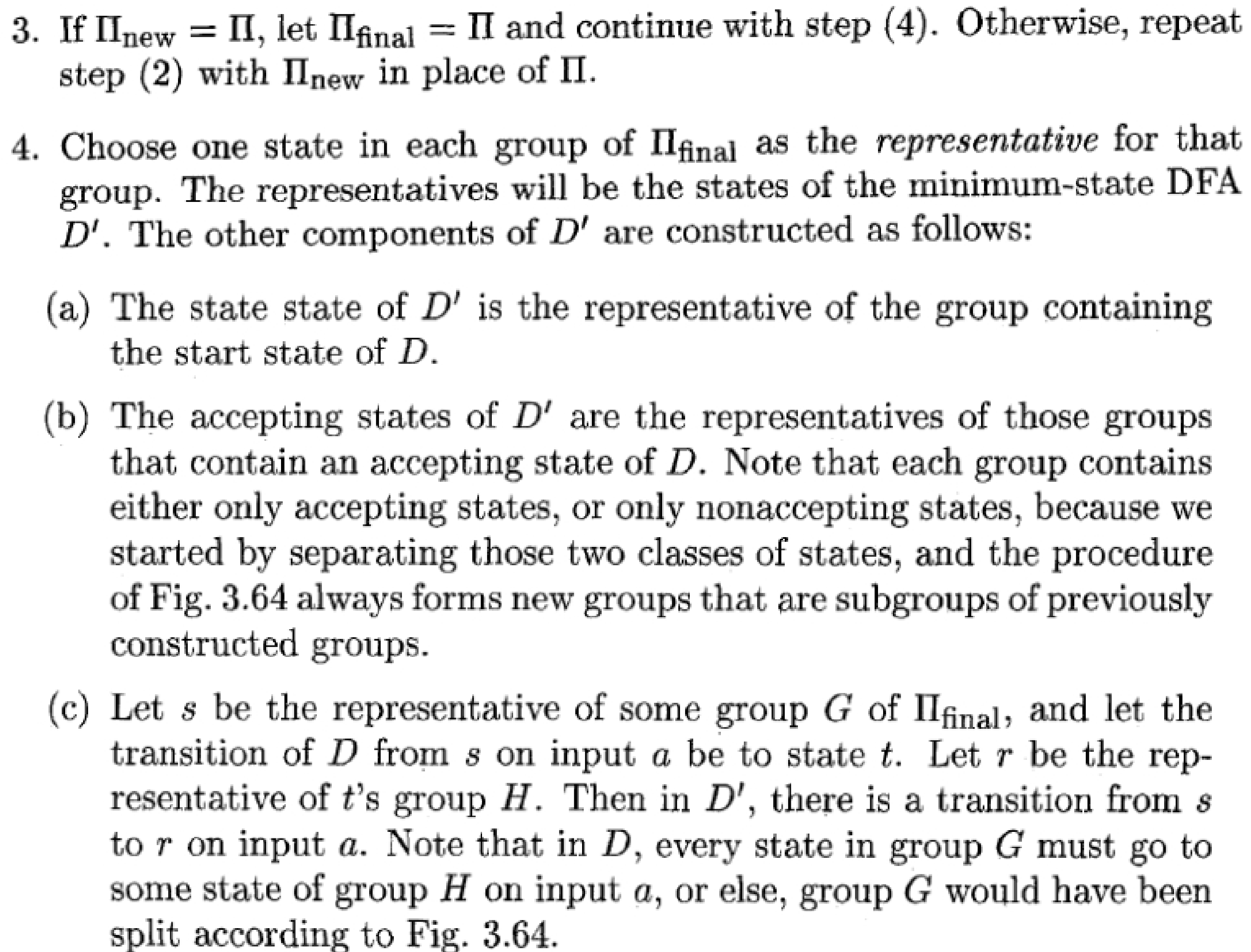
OUTPUT: A DFA **D'** accepting the same language as D and having as few states as possible.

METHOD:

1. Start with an initial partition Π with two groups, F and $S - F$, the accepting and nonaccepting states of D.

2. Apply the procedure of Fig. 3.64 to construct a new partition Π_{new}.

```
initially, let Π_new = Π;
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself */
        replace G in Π_new by the set of all subgroups formed;
}
```



3. If $\Pi_{\text{new}} = \Pi$, let $\Pi_{\text{final}} = \Pi$ and continue with step (4). Otherwise, repeat step (2) with Π_{new} in place of Π.

4. Choose one state in each group of Π_{final} as the *representative* for that group. The representatives will be the states of the minimum-state DFA D'. The other components of D' are constructed as follows:

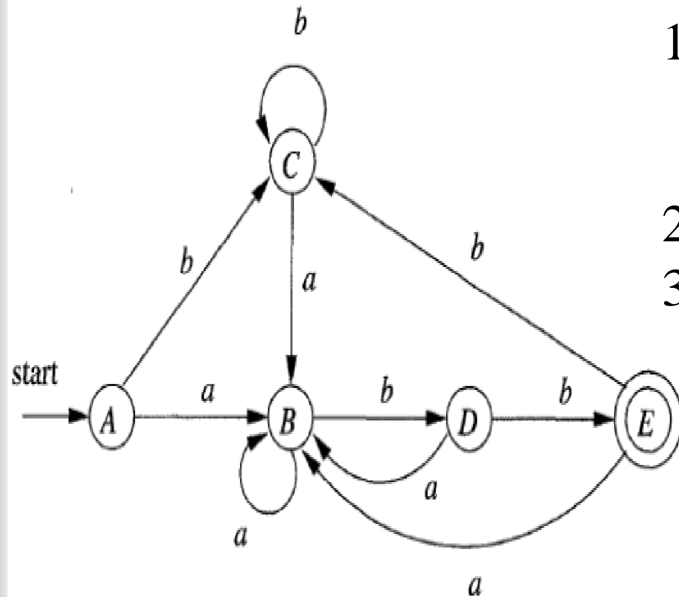
   (a) The state state of D' is the representative of the group containing the start state of D.

   (b) The accepting states of D' are the representatives of those groups that contain an accepting state of D. Note that each group contains either only accepting states, or only nonaccepting states, because we started by separating those two classes of states, and the procedure of Fig. 3.64 always forms new groups that are subgroups of previously constructed groups.

   (c) Let s be the representative of some group G of Π_{final}, and let the transition of D from s on input a be to state t. Let r be the representative of t's group H. Then in D', there is a transition from s to r on input a. Note that in D, every state in group G must go to some state of group H on input a, or else, group G would have been split according to Fig. 3.64.

Step 2

Example: input set is {a,b}, with DFA`Z2



1. Initially partition consists of the two groups
   - •non-final states {A, B, C, D},
   - •final state{E}
2. To construct Π_{new} , group {E} cannot be split
3. group {A, B, C, D} can be split into

   {A, B, C}{D}, and

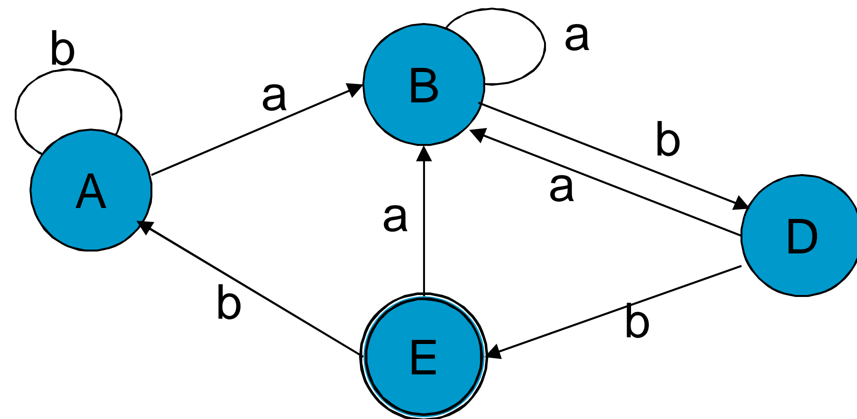
   Π_{new} for this round is {A, B, C){D){E

In the next round, split {A, B, C} into {A, C}{B}, since A and C each go to a member of {A, B, C) on input b, while B goes to a member of another group, {D}. Thus, after the second round, Π_{new} = {A, C} {B} {D} {E).

For the third round, we cannot split the one remaining group with more than one state, since A and C each go to the same state (and therefore to the same group) on each input. Π_{final} = {A, C}{B){D){E). The minimum-state of the given DFA has four states.

Minimized DFA

STATE	a	b
A	B	A
B	B	D
D	B	E
E	B	A



- Parser Generators : Produce Syntax Analyzers
- Scanner Generators : Produce Lexical Analyzers <= Lex (Flex)
- Syntax-directed Translation Engines : Generate Intermediate Code <= Yacc (Bison)
- Automatic Code Generators : Generate Actual Code
- Data-Flow Engines : Support Optimization