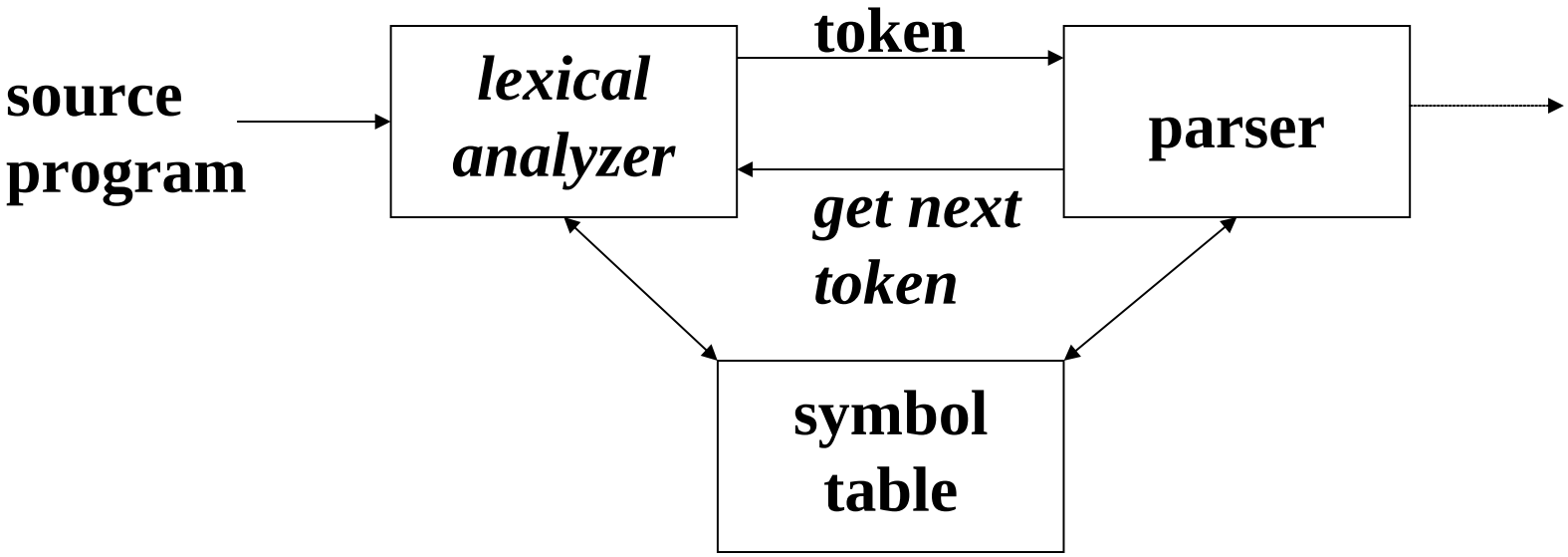
Lexical Analysis

Lexical Analysis

- Basic Concepts & Regular Expressions
  - What does a Lexical Analyzer do?
  - How does it Work?
  - Formalizing Token Definition & Recognition
- Reviewing Finite Automata Concepts
  - Non-Deterministic and Deterministic FA
  - Conversion Process
    - Regular Expressions to NFA
    - NFA to DFA
- Relating NFAs/DFAs /Conversion to Lexical Analysis

Lexical Analyzer in Perspective



Important Issue:

What are Responsibilities of each Box ?

Focus on Lexical Analyzer and Parser.

Lexical Analyzer in Perspective

- **LEXICAL ANALYZER**
  - Scan Input
  - Remove WS, NL, …
  - Identify Tokens
  - Create Symbol Table
  - Insert Tokens into ST
  - Generate Errors
  - Send Tokens to Parser
- **PARSER**
  - Perform Syntax Analysis
  - Actions Dictated by Token Order
  - Update Symbol Table Entries
  - Create Abstract Rep. of Source
  - Generate Errors
  - And More…. (We'll see later)

What Factors Have Influenced the Functional Division of Labor ?

- **Separation of Lexical Analysis From Parsing Presents a Simpler Conceptual Model**
  - A parser embodying the conventions for comments and white space is significantly more complex that one that can assume comments and white space have already been removed by lexical analyzer.
- **Separation Increases Compiler Efficiency**
  - Specialized buffering techniques for reading input characters and processing tokens…
- **Separation Promotes Portability.**
  - Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.

Introducing Basic Terminology

- What are Major Terms for Lexical Analysis?
  - **TOKEN**
    - A classification for a common set of strings
    - Examples Include <Identifier>, <number>, etc.
  - **PATTERN**
    - The rules which characterize the set of strings for a token
    - e.g., digit followed by zero or more digits
  - **LEXEME**
    - Actual sequence of characters that matches pattern and is classified by a token
    - Identifiers: x, count, name, etc…

Introducing Basic Terminology

Token	Sample Lexemes	Informal Description of Pattern
const	const	const
if	if	characters i, f
relation	<, <=, =, < >, >, >=	< or <= or = or < > or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	“core dumped”	any characters between “ and ” except “

Classifies Pattern

Actual values are critical. Info is :

1. Stored in symbol table
2. Returned to parser

Attributes for Tokens

Tokens influence parsing decision; the attributes influence the translation of tokens.

Example: E = M * C ** 2 (Fortran statement)

<id, pointer to symbol-table entry for E>

<assign_op, >

<id, pointer to symbol-table entry for M>

<mult_op, >

<id, pointer to symbol-table entry for C>

<exp_op, >

<num, integer value 2>

Handling Lexical Errors

- Error Handling is very localized, with Respect to Input Source
- For example: ***fi (a == f(x))***
  a lexical analyzer cannot tell whether **fi** is a misspelling of the keyword if or an undeclared function identifier.
- **In what Situations do Errors Occur?**
  - Lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of remaining input.
- **Panic mode Recovery**
  - Delete successive characters from the remaining input until the analyzer can find a well-formed token.
  - May confuse the parser – creating syntax error
- **Possible error recovery actions:**
  - Deleting or Inserting Input Characters
  - Replacing or Transposing Characters

Buffer Pairs

- Lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.
- In C, single-character operators like =, or < could also be the beginning of a two-character operator like ==, or <=.
- Use a function **ungetc** to push lookahead characters back into the input stream.
- Large amount of time can be consumed moving characters.

Special Buffering Technique

Use a buffer divided into two N-character halves

N = Number of characters on one disk block

One system command read N characters

Fewer than N character => **eof**

Buffer Pairs (2)

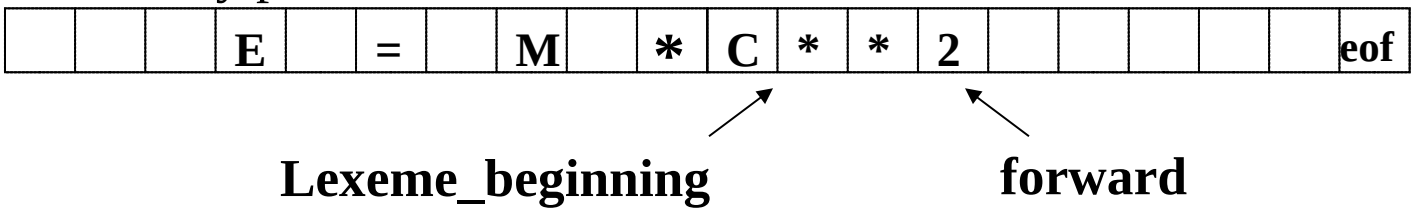
Two pointers **lexeme_beginning** and **forward** to the input buffer are maintained.

The string of characters between the pointers is the current lexeme.

Initially both pointers point to first character of the next lexeme to be found. Forward pointer scans ahead until a match for a pattern is found

Once the next lexeme is determined, the forward pointer is set to the character at its right end.

After the lexeme is processed both pointers are set to the character immediately past the lexeme



Comments and white space can be treated as patterns that yield no token

Code to advance forward pointer

```
if  forward at the end of first half then  begin
        reload second half ;
        forward  : =  forward  + 1;
end
else if  forward  at end of second half  then begin
         reload first half ;
         move  forward  to biginning of first half
end
else  forward  : =  forward  + 1;
```

Pitfalls

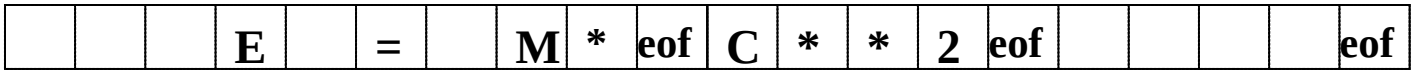
1. This buffering scheme works quite well most of the time but with it amount of lookahead is limited.
2. Limited lookahead makes it impossible to recognize tokens in situations where the distance, forward pointer must travel is more than the length of buffer.

Problem?

Except at the end of buffer halves, for each advance of *forward* requires 2 tests

```
if  forward at the end of first half then  begin
        reload second half ;
        forward  : =  forward  + 1;
end
else if  forward  at end of second half  then begin
          reload first half ;
          move  forward  to biginning of first half
end
else  forward  : =  forward  + 1;
```

Algorithm: Buffered I/O with Sentinels



lexeme beginning

forward (scans ahead to find pattern match)

```
forward : = forward + 1 ;
if forward ↑ = eof then begin
   if forward at end of first half then begin
        reload second half ;        ← Block I/O
        forward : = forward + 1
   end
   else if forward at end of second half then begin
        reload first half ;         ← Block I/O
        move forward to biginning of first half
   end
   else / * eof within buffer signifying end of input * /
        terminate lexical analysis
end                  2nd eof ⇒ no more input !
```

We extend each buffer half to hold a sentinal [*eof*] character at the end

Token Recognition

How can we use concepts developed so far to assist in recognizing tokens of a source language ?

Assume Following Tokens:

if, then, else, relop, id, num

We can express patterns using regular expressions.

Given Tokens, What are Patterns ?

if → if

then → then

else → else

relop → < | <= | > | >= | = | <>

digit → [0-9]

letter → [A-Za-z]

id → letter (letter | digit)*

num → digit $^{+}$ (. digit $^{+}$) ? (E(+ | -) ? digit $^{+}$) ?

Grammar:

stmt **→ |if *expr* then *stmt***
|if *expr* then *stmt* else *stmt*
|∈

expr **→ *term* relop *term* | *term***

term **→ id | num**

What Else Does Lexical Analyzer Do?

Scan away *blanks*, new lines, tabs

Can we Define Tokens For These?

blank → blank

tab → tab

newline → newline

delim → blank | tab | newline

ws → delim $^{+}$

In these cases no token is returned to parser

Overall

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

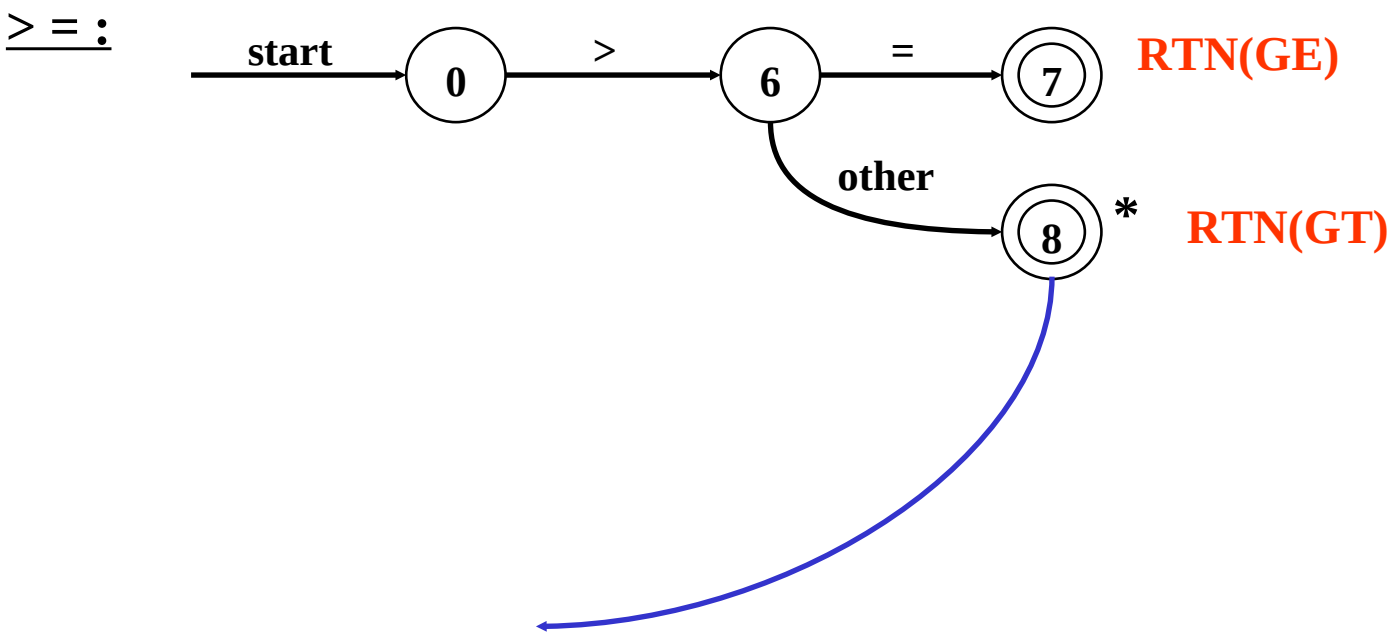
Tokens, their patterns, and attribute values

Note: Each token has a unique token identifier to define category of lexemes

Constructing Transition Diagrams for Tokens

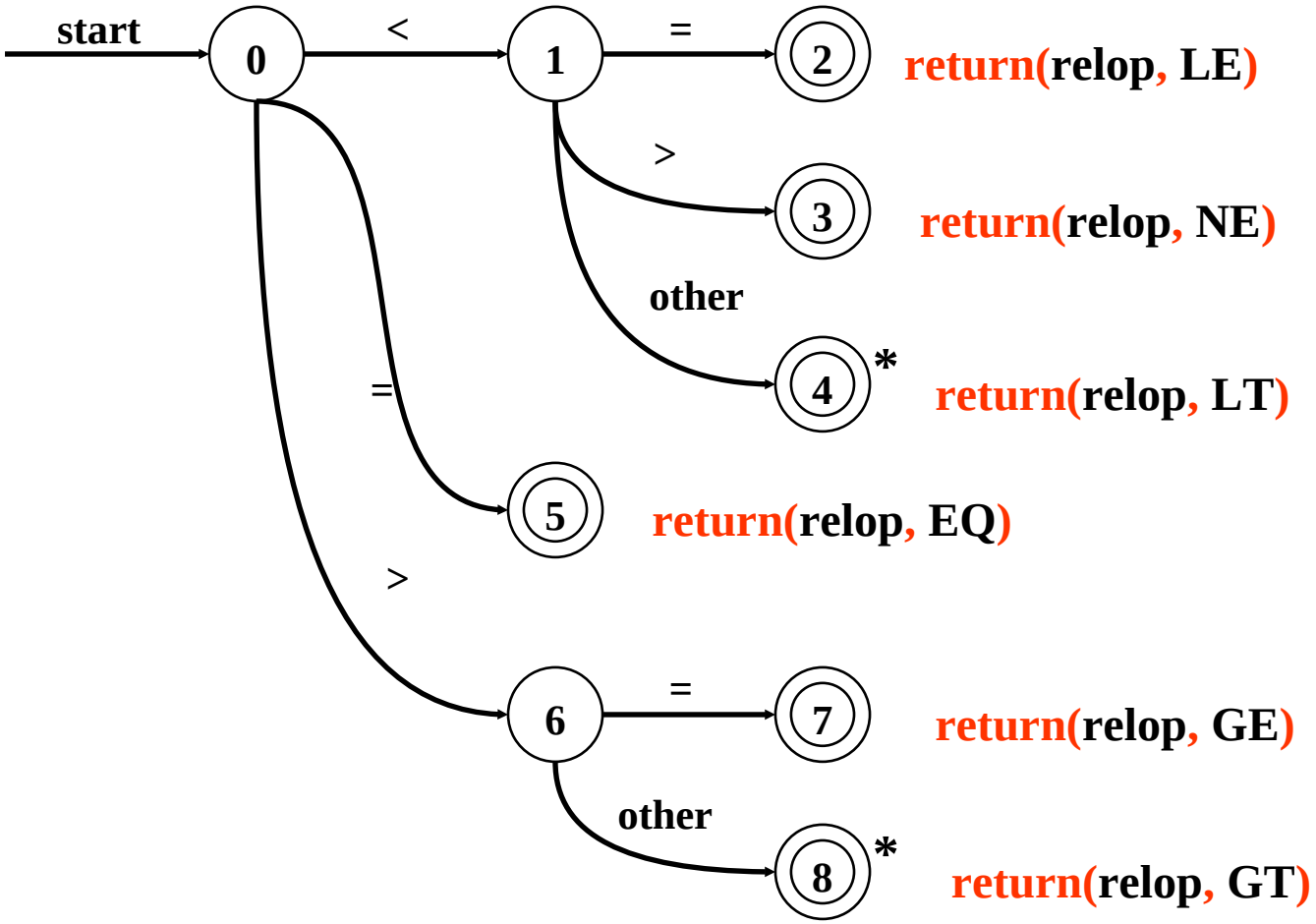
- **Transition Diagrams (TD) are used to represent the tokens**
- **As characters are read, the relevant TDs are used to attempt to match lexeme to a pattern**
- **Each TD has:**
  - **States : Represented by Circles**
  - **Actions : Represented by Arrows between states**
  - **Start State : Beginning of a pattern (Arrowhead)**
  - **Final State(s) : End of pattern (Concentric Circles)**
- **Each TD is Deterministic (assume) - No need to choose between 2 different actions !**

Example TDs



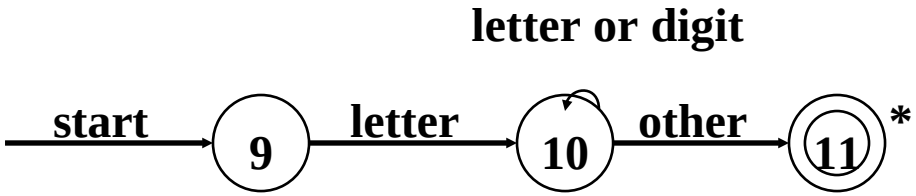
We've accepted ">" and have read one extra char that must be unread.

Example : All RELOPs



Example TDs : id and delim

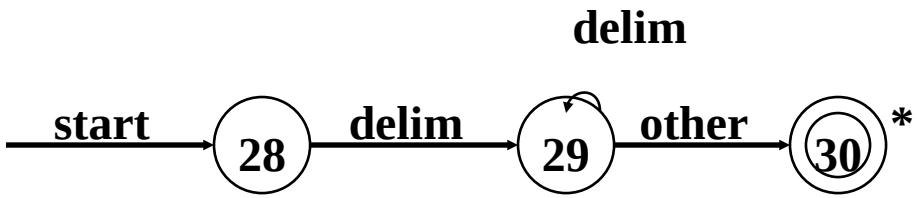
id :



return(get_token(), install_id())

Either returns ptr or "0" if reserved

delim :



Question: Why are there no TDs for `then, else, if` ?

What Else Does Lexical Analyzer Do?

All Keywords / Reserved words are matched as ids !!

- **So recognizing keywords and identifiers presents a problem.**
- **There are two ways that we can handle reserved words that look like identifiers.**

What Else Does Lexical Analyzer Do?

First Approach (Preferred):

- **Install the reserved words in the symbol table initially.**
- **After the match, the symbol table or a special keyword table is consulted.**
- **Keyword table contains string versions of all keywords and associated token values**

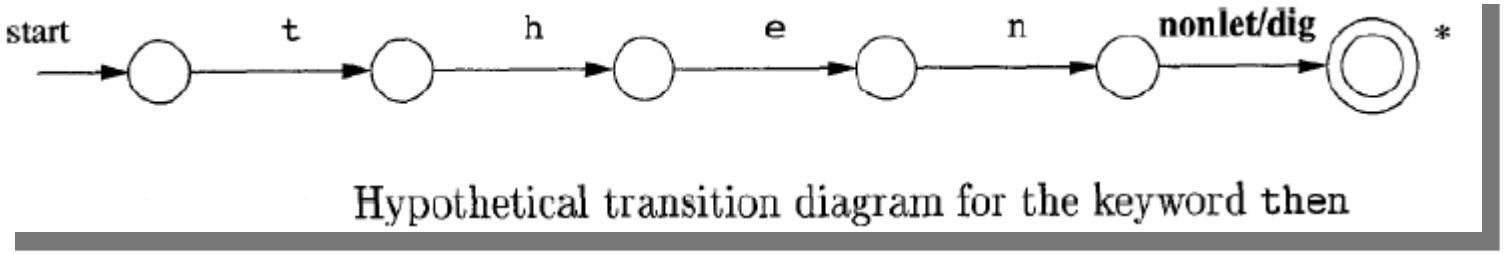
if	**15**
then	**16**
begin	**17**
...	**...**

- **When a match is found, the token is returned, along with its symbolic value, i.e., "then", 16**
- **If a match is not found, then it is assumed that an id has been discovered**

What Else Does Lexical Analyzer Do?

Second Approach :

- **Create separate transition diagrams for each keyword.**



Hypothetical transition diagram for the keyword then

Example TDs : Unsigned Number

digit → 0 | 1 | 2 | … | 9

digits → digit digit*

1240, 39.45, 6.33E15, or 1.578E-41

optional_fraction → . digits | ∈

optional_exponent → (E (+ | -| ∈) digits) | ∈

num → digits optional_fraction optional_exponent

Shorthand

digit → 0 | 1 | 2 | … | 9

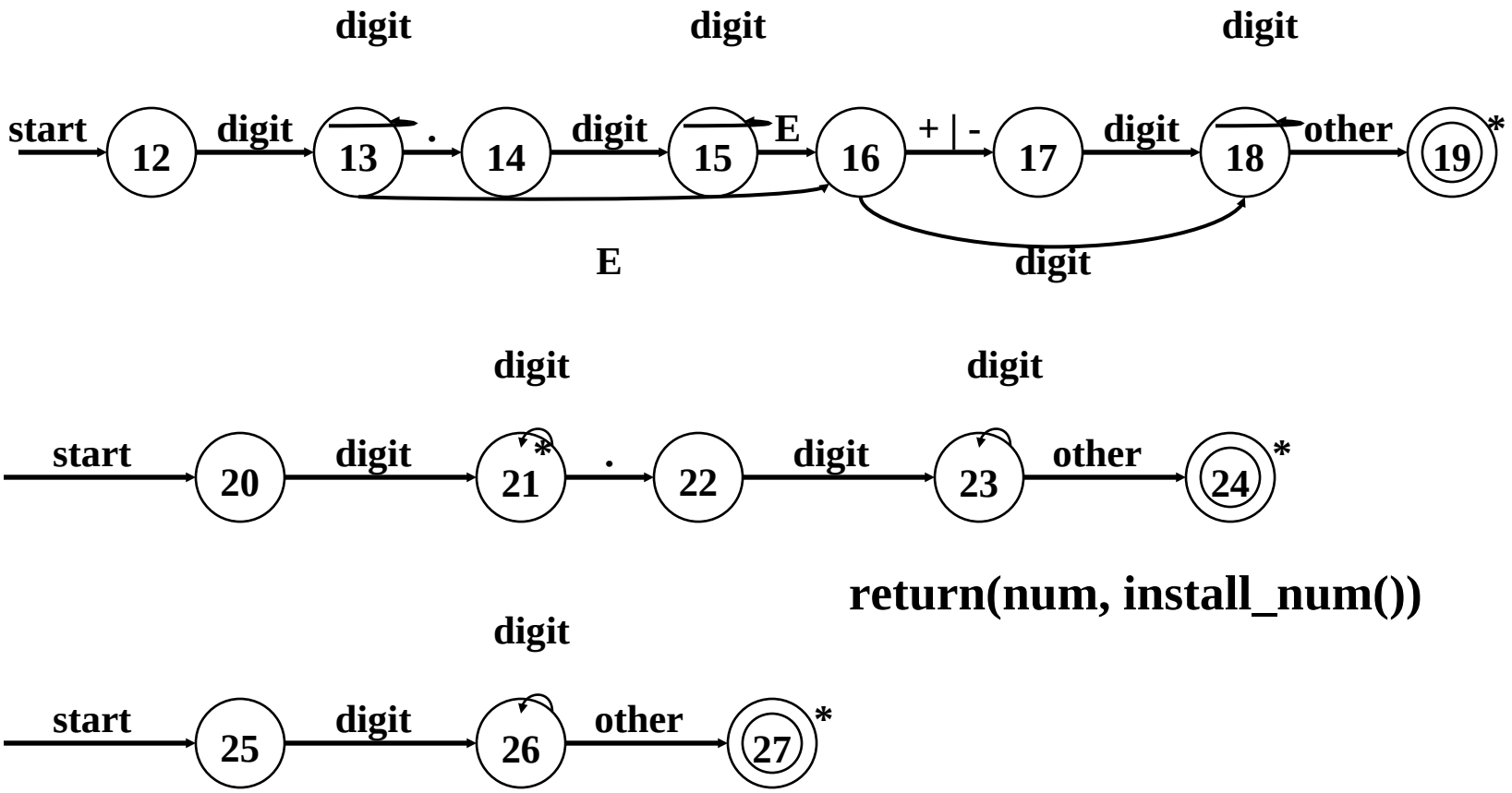
digits → digit^{+}

optional_fraction → (. digits) ?

optional_exponent → (E (+ | -) ? digits) ?

num → digits optional_fraction optional_exponent

Example TDs : Unsigned #s



Questions: **Is ordering important for unsigned #s ?**

Implementing Transition Diagrams

A sequence of transition diagrams can be converted into a program to look for the tokens specified by the grammar

Each state gets a segment of code

FUNCTIONS USED

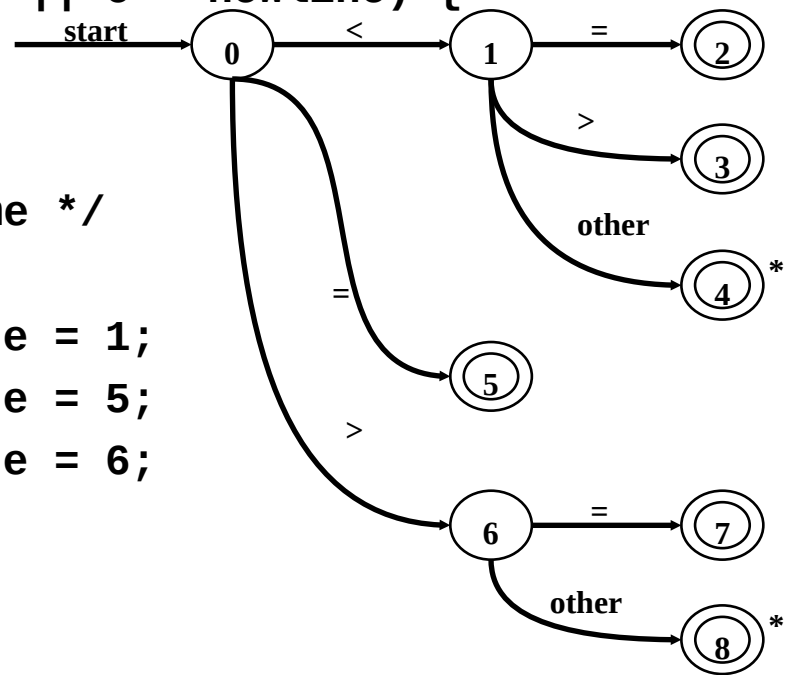
- nextchar(),
- retract(),
- install_num(),
- install_id(),
- gettoken(),
- isdigit(),
- isletter(),
- recover()

Implementing Transition Diagrams

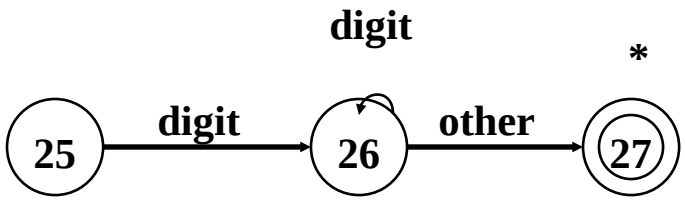
```
int state = 0, start = 0

lexeme_beginning = forward;
token nexttoken()
{   while(1) {
        switch (state) {
        case 0:   c = nextchar();
           /* c is lookahead character */
           if (c== blank || c==tab || c== newline) {
              state = 0;
              lexeme_beginning++;
                 /* advance
                   beginning of lexeme */
           }
           else if (c == '<') state = 1;
           else if (c == '=') state = 5;
           else if (c == '>') state = 6;
           else state = fail();
           break;
           … /* cases 1-8 here */
```

repeat until a "return" occurs



Implementing Transition Diagrams, II




```
.............
case 25;  c = nextchar();
          if (isdigit(c)) state = 26;
          else state = fail();
          break;
case 26;  c = nextchar();
          if (isdigit(c)) state = 26;
          else state = 27;
          break;
case 27;  retract(1); lexical_value = install_num();
          return ( NUM );
.............
```

advances forward

Case numbers correspond to transition diagram states !

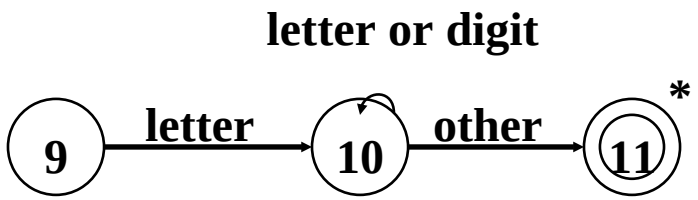
retracts forward

looks at the region lexeme_beginning ... forward

Implementing Transition Diagrams, III

```
.............
 case 9:   c = nextchar();
           if (isletter(c)) state = 10;
           else state = fail();
           break;
 case 10;  c = nextchar();
           if (isletter(c)) state = 10;
           else if (isdigit(c)) state = 10;
           else state = 11;
           break;
 case 11;  retract(1); lexical_value = install_id();
           return ( gettoken(lexical_value) );
.............
```

reads token name from ST



When Failures Occur:

```
int fail()
{
     forward = lexeme beginning;
     switch (start) {
        case 0:   start = 9;  break;
        case 9:   start = 12; break;
        case 12:  start = 20; break;
        case 20:  start = 25; break;
        case 25:  recover();  break;
        default:   /* lex error */
     }
     return start;
}
```

Switch to next transition diagram

Finite State Automata (FSAs)

- **AKA "Finite State Machines", "Finite Automata", "FA"**
- A *recognizer* for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise.
  - The regular expression is compiled into a recognizer by constructing a generalized transition diagram called a finite automaton.
- One start state
- Many final states
- Each state is labeled with a state name
- Directed edges, labeled with symbols
- Two types
  - Deterministic (DFA)
  - Non-deterministic (NFA)

Simulation of a Finite Automata (FA)

```
s ← s0
c ← nextchar;
while c ≠ eof do
  s ← move(s,c);
  c ← nextchar;
end;
if s is in F then return “yes”
             else return “no”
```

DFA simulation

```
S ← ∈-closure({s0})
c ← nextchar;
while c ≠ eof do
  S ← ∈-closure(move(S,c));
  c ← nextchar;
end;
if S∩F≠∅ then return “yes”
         else return “no”
```

NFA simulation

Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) is mathematical model that consists of

1. A set of states S
2. A set of input symbols S
3. A transition function that maps state/symbol pairs to a set of states

1. A special state s_0 called the start state
2. A set of states F (subset of S) of final states

INPUT: string

OUTPUT: yes or no

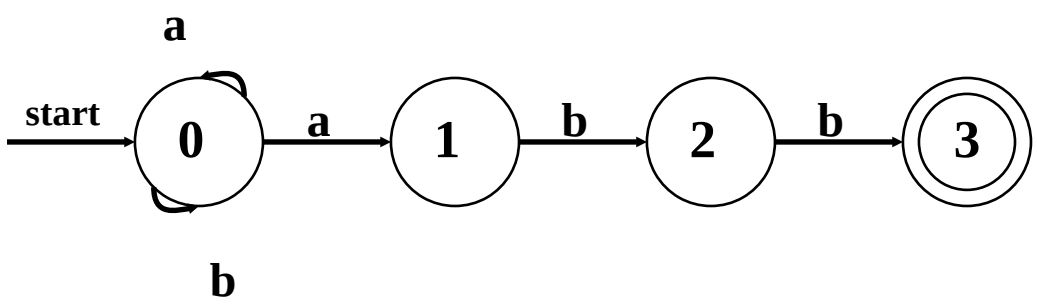
Example – NFA : (a|b)*abb

$S = \{ 0, 1, 2, 3 \}$

$s_0 = 0$

$F = \{ 3 \}$

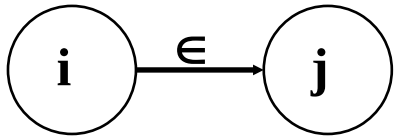
$\Sigma = \{ a, b \}$



state \ input	a	b
0	{ 0, 1 }	{ 0 }
1	--	{ 2 }
2	--	{ 3 }

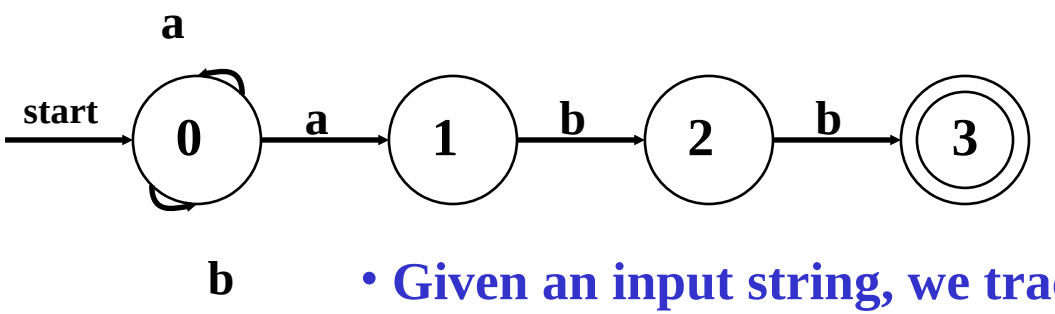
Transition Table

∈ (null) moves possible



Switch state but do not use any input symbol

How Does An NFA Work ?



- Given an input string, we trace moves
- If no more input & in final state, ACCEPT

EXAMPLE:

Input: ababb

move(0, a) = 1

move(1, b) = 2

move(2, a) = ? (undefined)

REJECT !

-OR-

move(0, a) = 0

move(0, b) = 0

move(0, a) = 1

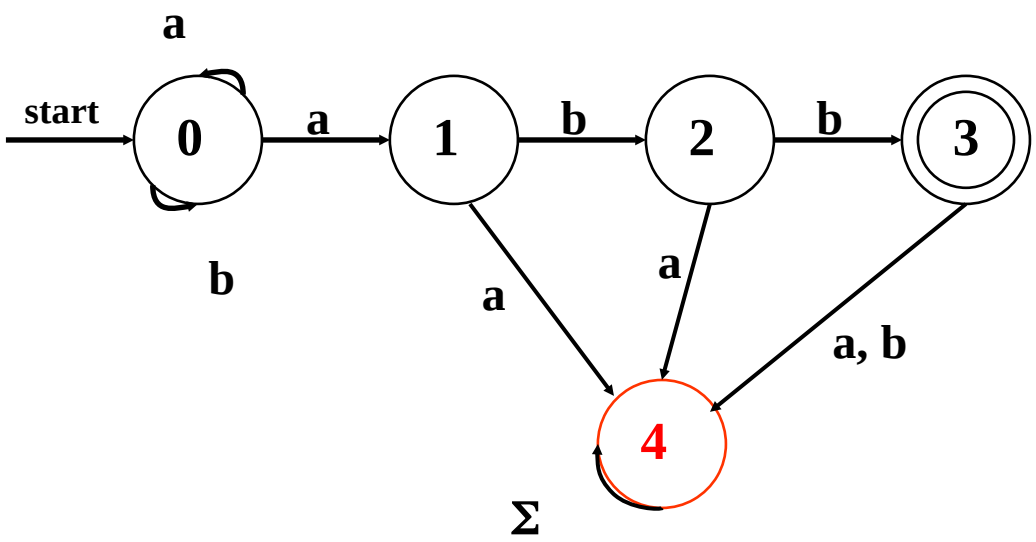
move(1, b) = 2

move(2, b) = 3

ACCEPT !

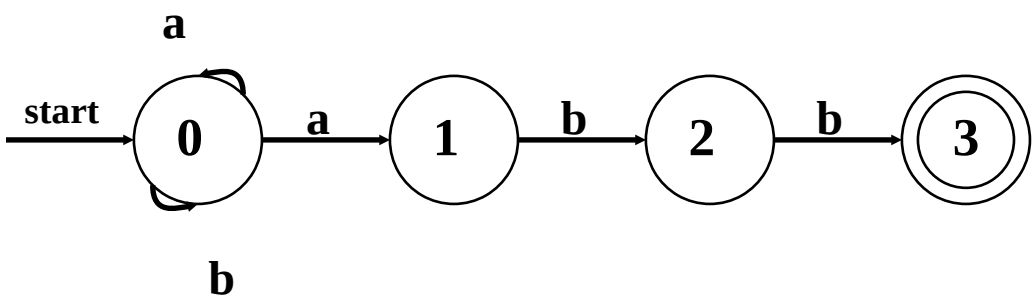
Handling Undefined Transitions

We can handle undefined transitions by defining one more state, a “death” state, and transitioning all previously undefined transition to this **death state.**



Other Concepts

Not all paths may result in acceptance.



aabb is accepted along path : 0 → 0 → 1 → 2 → 3

BUT… it is not accepted along the valid path:

0 → 0 → 0 → 0 → 0

Deterministic Finite Automata

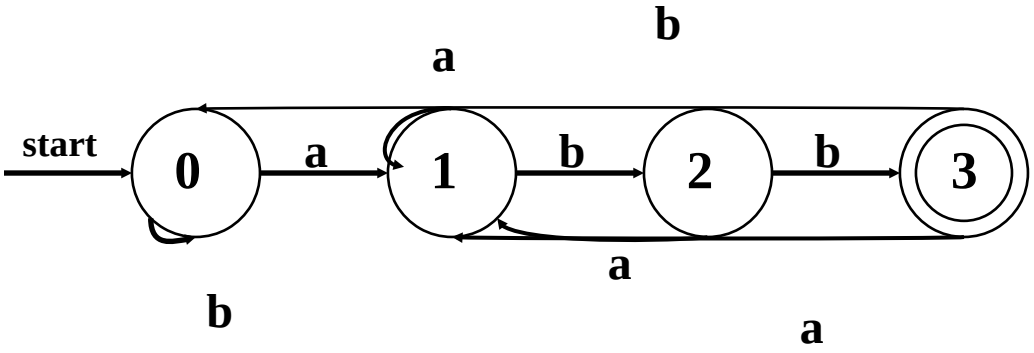
A DFA is an NFA with the following restrictions:

- $\in$ moves are not allowed
- For every state s $\in$S, there is one and only one path from s for every input symbol a $\in \Sigma$.

Since transition tables don't have any alternative options, DFAs are easily simulated via an algorithm.

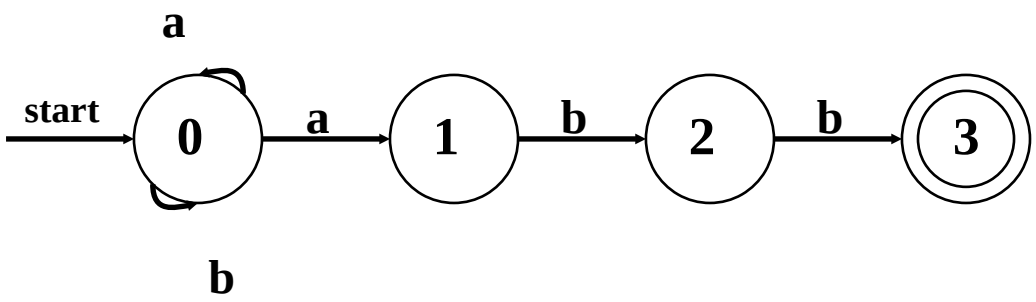
```
s ← s0
c ← nextchar;
while c ≠ eof do
  s ← move(s,c);
  c ← nextchar;
end;
if s is in F then return "yes"
             else return "no"
```

Example – DFA : (a|b)*abb



DFA

Recall the original NFA:



Relation between RE, NFA and DFA

1. There is an algorithm for converting any RE into an NFA.
2. There is an algorithm for converting any NFA to a DFA.
3. There is an algorithm for converting any DFA to a RE.

These facts tell us that **REs, NFAs and DFAs have equivalent expressive power.**

All three describe the class of **regular languages**.

NFA vs DFA

An NFA may be simulated by algorithm, when NFA is constructed from the R.E

Algorithm run time is proportional to |N| * |x| where |N| is the number of states and |x| is the length of input

Alternatively, we can construct DFA from NFA and uses it to recognize input

The space requirement of a DFA can be large. The RE (a+b)*a(a+b)(a+b) ….(a+b) [n-1 (a+b) at the end] has no DFA with less than 2^n states. Fortunately, such RE in practice does not occur often

	space required	time to simulate
NFA	**O(|r|)**	**O(|r|*|x|)**
DFA	**$O(2^{|r|})$**	**O(|x|)**

where |r| is the length of the regular expression.

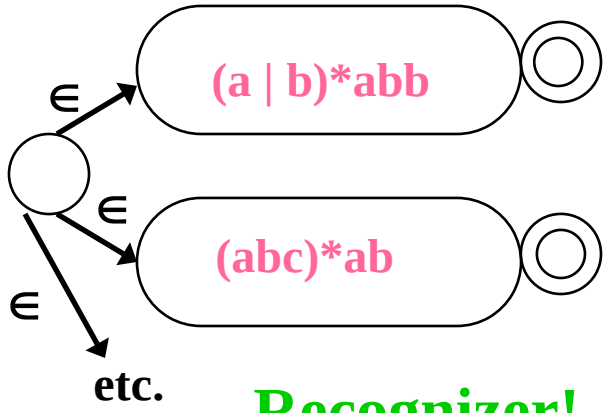
Lexical Analyzer

- **Designing Lexical Analyzer Generator**

  Reg. Expr. → NFA construction

  NFA → DFA conversion

  DFA simulation for lexical analyzer



Recognizer!

- Each pattern recognizes lexemes. Longest prefix of input is matched. In case of tie the first pattern listed is chosed

- Each pattern described by regular expression

Lex Specification → Lexical Analyzer

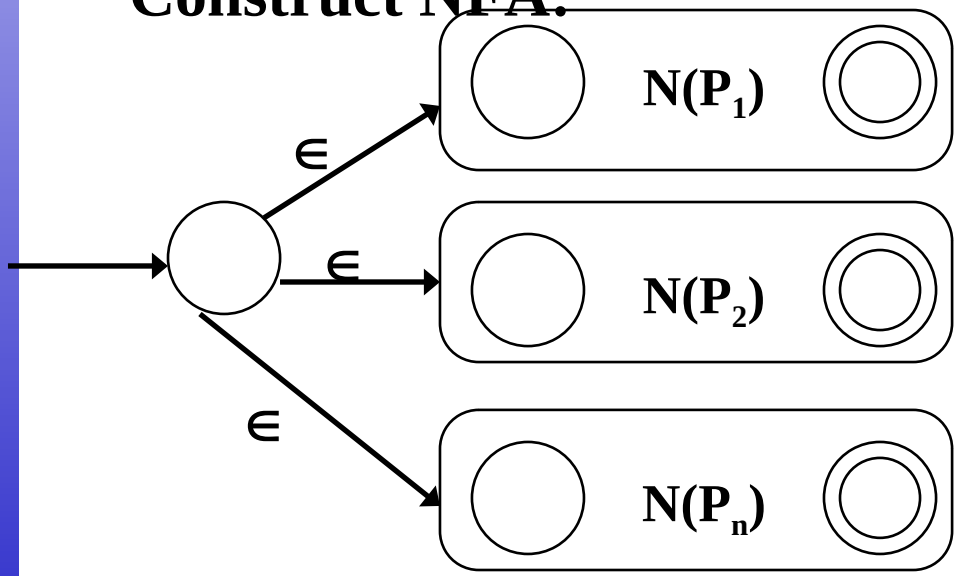
- **Let $P_1, P_2, \ldots, P_n$ be Lex patterns**

  (regular expressions for valid tokens in prog. lang.)

- **Construct $N(P_1), N(P_2), \ldots N(P_n)$**

- **Note: accepting state of $N(P_i)$ will be marked by P_i**

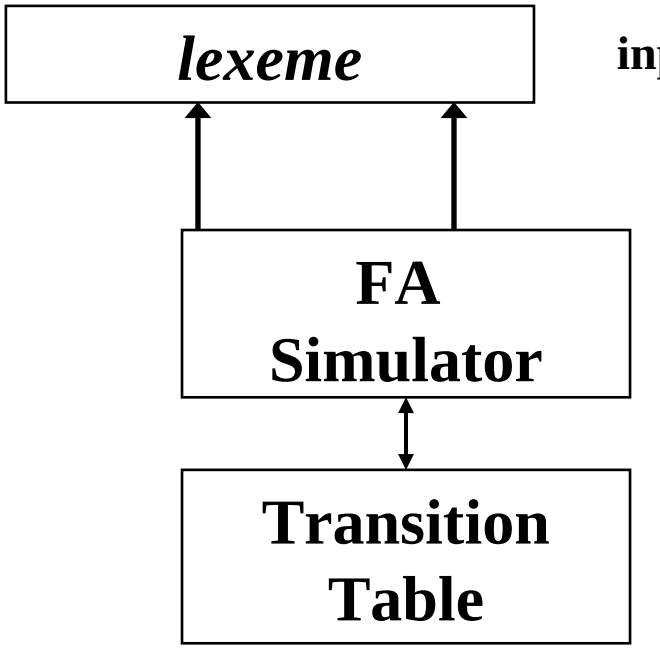
- **Construct NFA:**



- **Lex applies conversion algorithm to construct DFA that is equivalent!**

Pictorially

(a) Lex Compiler



(b) Schematic lexical analyzer

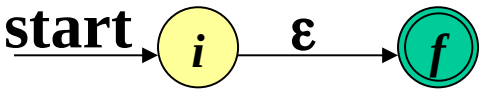
Converting Regular Expressions to NFAs

- How do we define an NFA that accepts a regular expression?
- It is very simple. Remember that a regular expression is formed by the use of alternation, concatenation, and repetition.
- Thus all we need to do is to know how to build the NFA for a single symbol, and how to compose NFAs.

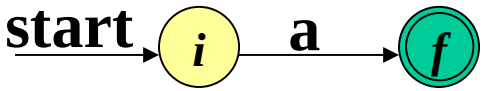
Converting Regular Expressions to NFAs

Thompson's Construction

- Empty string ε is a regular expression denoting { ε }



- *a* is a regular expression denoting {*a*} for any *a* in Σ

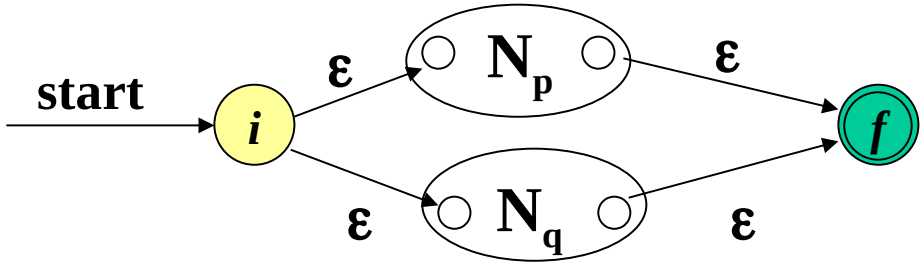


Converting Regular Expressions to NFAs

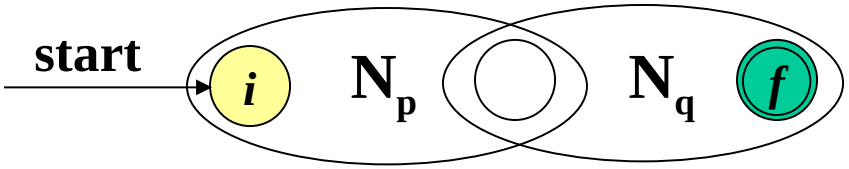
Composing NFAs with Alternation and Concatenation

If P and Q are regular expressions with NFAs N_p, N_q:

P | Q (union)



PQ (concatenation)

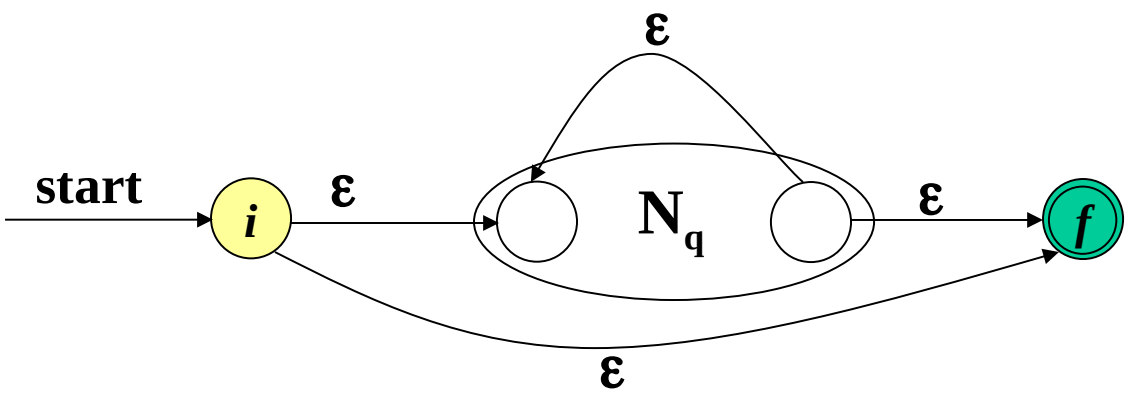


Converting Regular Expressions to NFAs

Composing NFAs with Repeation

If Q is a regular expression with NFA N_q:

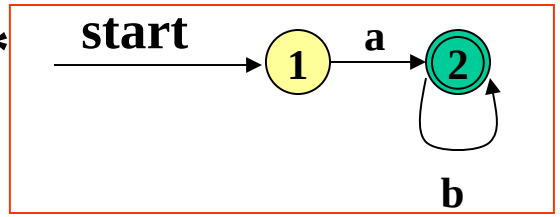
Q* (closure)



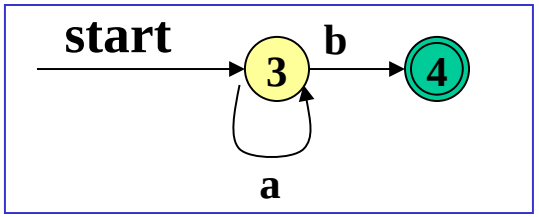
Example (ab* | a*b)*

Starting with:

ab*

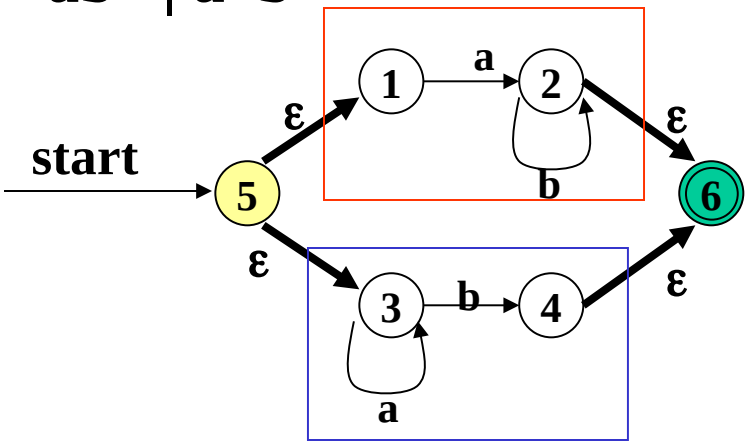


a*b



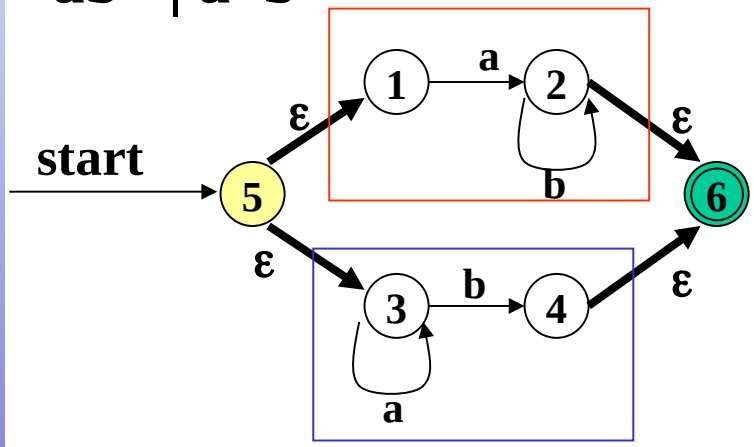
---

ab* | a*b

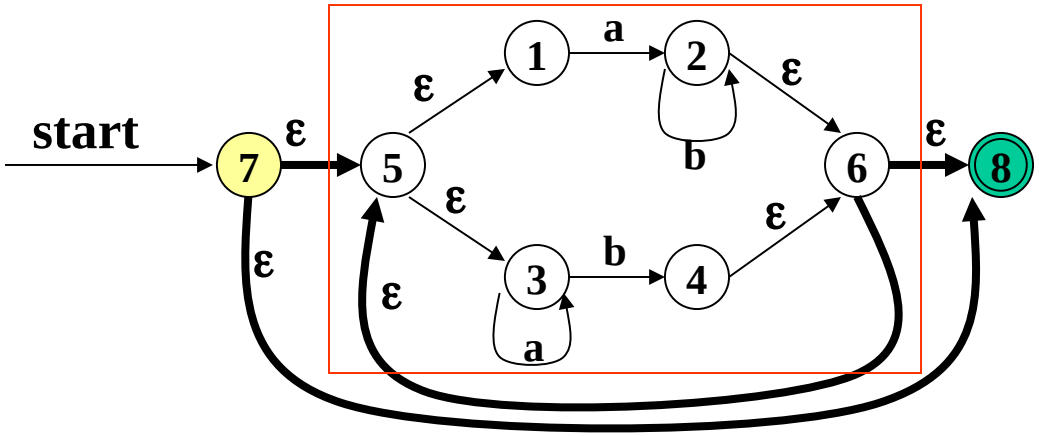


Example (ab* | a*b)*

ab* | a*b



(ab* | a*b)*



Properties of Construction

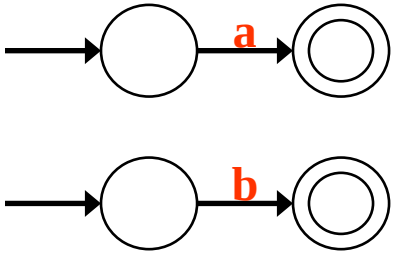
Let r be a regular expression, with NFA N(r), then

1. N(r) has #of states $\leq$ 2*(#symbols + #operators) of r
2. N(r) has exactly one start and one accepting state
3. Each state of N(r) has at most one outgoing edge $a \in \Sigma$ or at most two outgoing $\in$-transition

Detailed Example – NFA Construction(1)

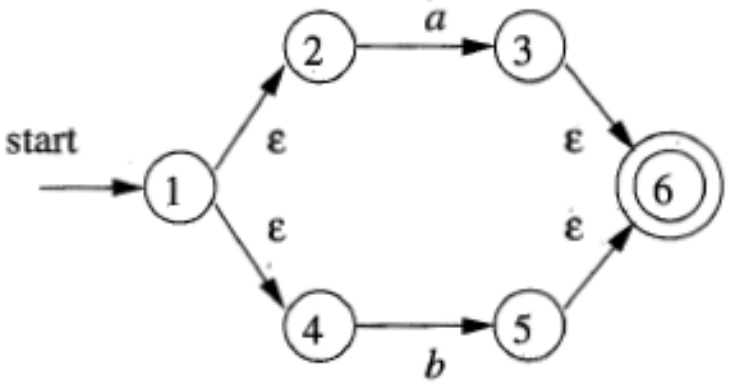
(a|b)*abb

r_1:

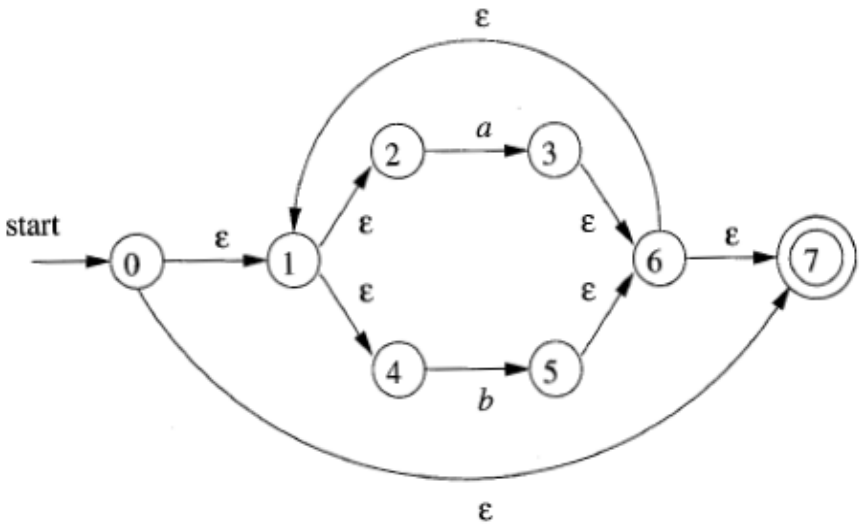


r_2:

r_3:r1|r2



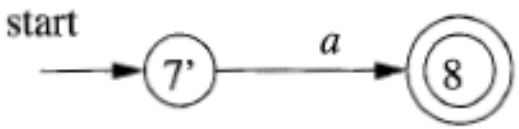
r_4:(r3)*



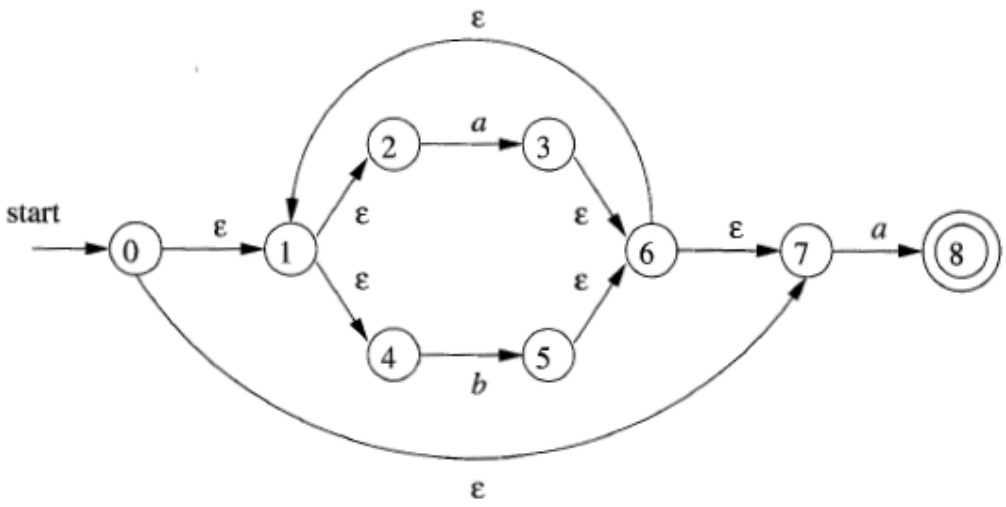
Detailed Example – NFA Construction(2)

(a|b)*abb

r_5



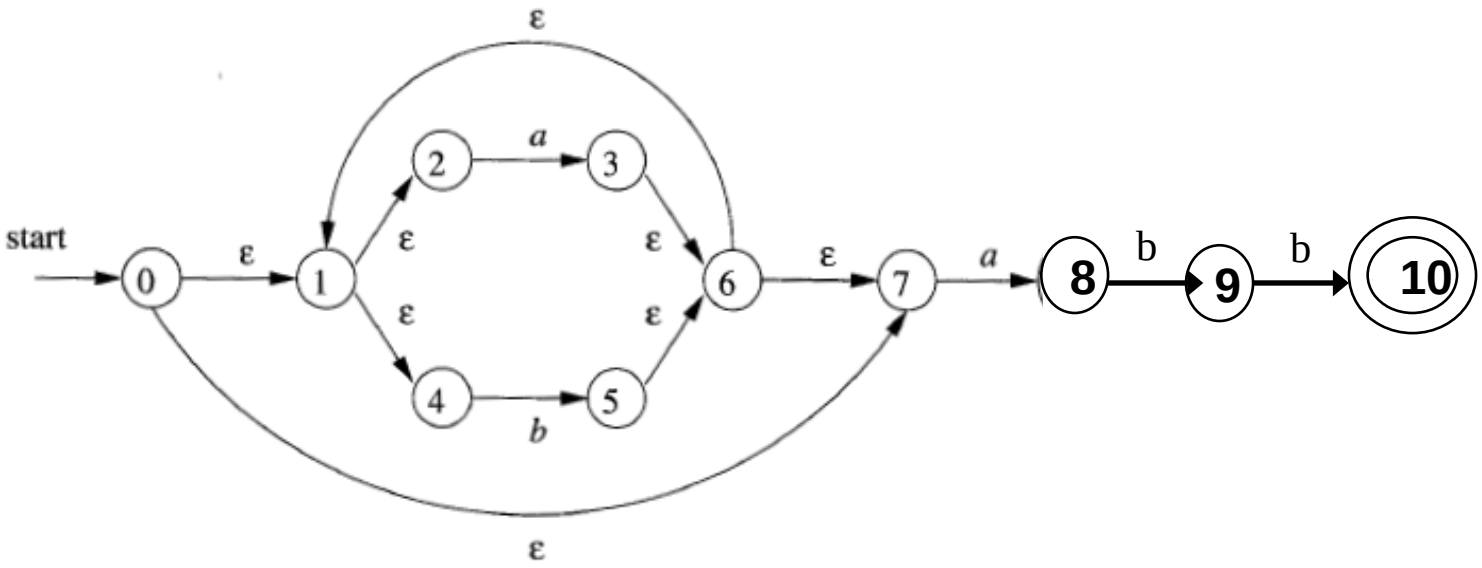
r_6: r4 r5



Detailed Example – Final Step

(a|b)*abb

r_{10}



Converting NFAs to DFAs (subset construction)

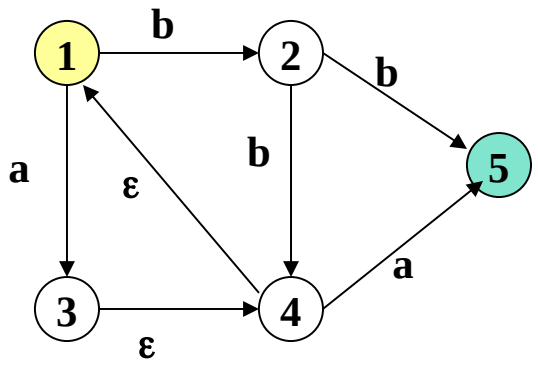
- **Idea**: Each state in the new DFA will correspond to some set of states from the NFA. The DFA will be in state $\{s_0,s_1,\ldots\}$ after input if the NFA could be in *any* of these states for the same input.

- **Input**: NFA N with state set S_N, alphabet S, start state s_N, final states F_N, transition function T_N: S_N x {S U e} → S_N

- **Output**: DFA D with state set S_D, alphabet S, start state s_D = e-closure(s_N), final states F_D, transition function T_D: S_D x S → S_D

Terminology: ε-closure

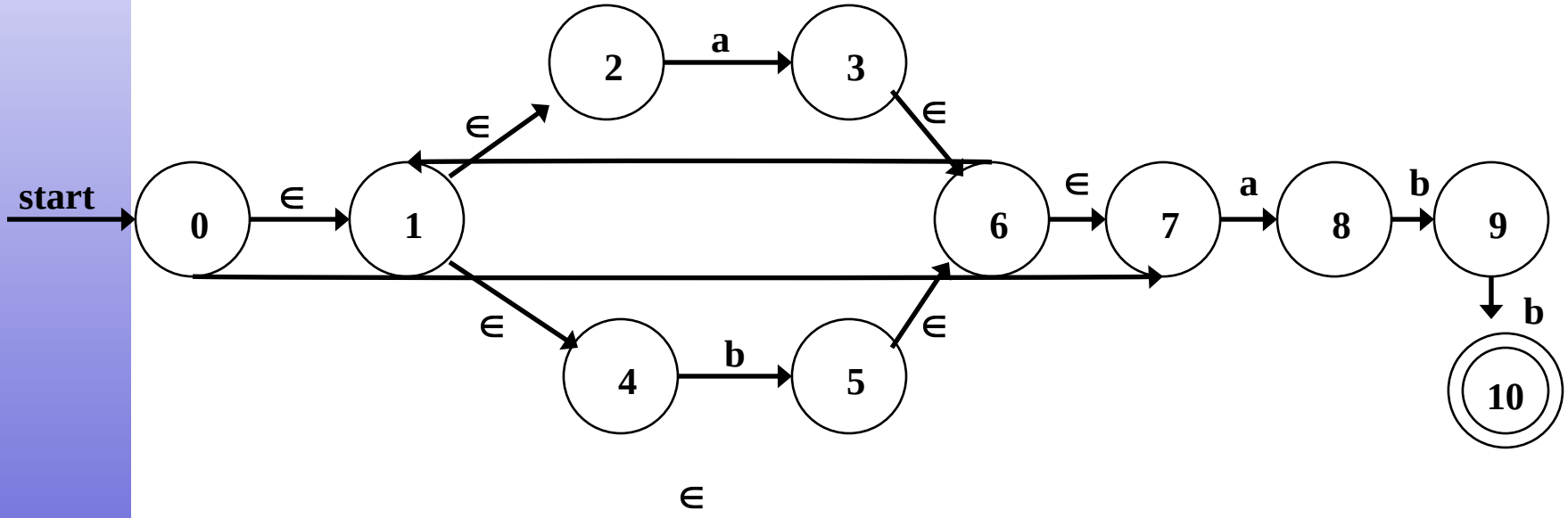
ε-closure(T) = T + all NFA states reachable from any state in T using only ε transitions.



ε-closure({1,2,5}) = {1,2,5}
ε-closure({4}) = {1,4}
ε-closure({3}) = {1,3,4}
ε-closure({3,5}) = {1,3,4,5}

Illustrating Conversion – An Example

Start with NFA: **(a | b)*abb**



First we calculate: ∈-closure(0) **(i.e., state 0)**

∈-closure(0) = {0, 1, 2, 4, 7} **(all states reachable from 0 on ∈-moves)**

Let A={0, 1, 2, 4, 7} be a state of new DFA, D.

Conversion Example – continued (1)

2nd , we calculate : a : ∈-closure(*move*(A,a)) and
b : ∈-closure(*move*(A,b))

a : ∈-closure(*move*(A,a)) = ∈-closure(*move*({0,1,2,4,7},a))}
adds {3,8} (since *move*(2,a)=3 and *move*(7,a)=8)

From this we have : ∈-closure({3,8}) = {1,2,3,4,6,7,8}
(since 3→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let B={1,2,3,4,6,7,8} be a new state. Define Dtran[A,a] = B.

b : ∈-closure(*move*(A,b)) = ∈-closure(*move*({0,1,2,4,7},b))

adds {5} (since *move*(4,b)=5)

From this we have : ∈-closure({5}) = {1,2,4,5,6,7}
(since 5→6 →1 →4, 6 →7, and 1 →2 all by ∈-moves)

Let C={1,2,4,5,6,7} be a new state. Define Dtran[A,b] = C.

Conversion Example – continued (2)

3rd , we calculate for state B on {a,b}

a : ∈-closure(*move*(B,a)) = ∈-closure(*move*({1,2,3,4,6,7,8},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[B,a] = B.

b : ∈-closure(*move*(B,b)) = ∈-closure(*move*({1,2,3,4,6,7,8},b))}
= {1,2,4,5,6,7,9} = D

Define Dtran[B,b] = D.

4th , we calculate for state C on {a,b}

a : ∈-closure(*move*(C,a)) = ∈-closure(*move*({1,2,4,5,6,7},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[C,a] = B.

b : ∈-closure(*move*(C,b)) = ∈-closure(*move*({1,2,4,5,6,7},b))}
= {1,2,4,5,6,7} = C

Define Dtran[C,b] = C.

Conversion Example – continued (3)

5th , we calculate for state D on {a,b}

a : ∈-closure(*move*(D,a)) = ∈-closure(*move*({1,2,4,5,6,7,9},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[D,a] = B.

b : ∈-closure(*move*(D,b)) = ∈-closure(*move*({1,2,4,5,6,7,9},b))}
= {1,2,4,5,6,7,10} = E

Define Dtran[D,b] = E.

Finally, we calculate for state E on {a,b}

a : ∈-closure(*move*(E,a)) = ∈-closure(*move*({1,2,4,5,6,7,10},a))}
= {1,2,3,4,6,7,8} = B

Define Dtran[E,a] = B.

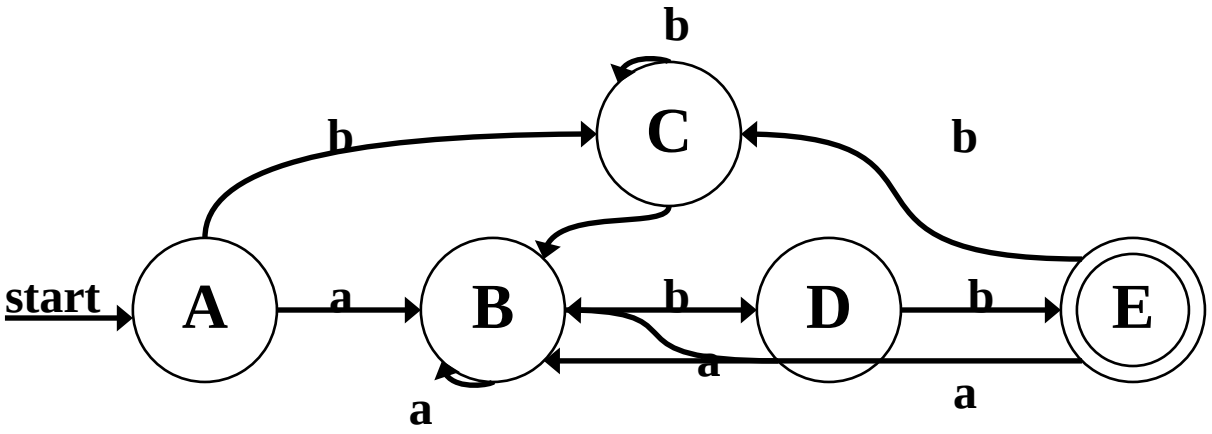
b : ∈-closure(*move*(*E*,b)) = ∈-closure(*move*({1,2,4,5,6,7,10},b))}
= {1,2,4,5,6,7} = C

Define Dtran[E,b] = C.

Conversion Example – continued (4)

This gives the transition table Dtran for the DFA of:

Dstates	Input Symbol	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



Algorithm For Subset Construction

computing the ∈-closure

```
push all states in T onto stack;
initialize ∈-closure(T) to T;
while stack is not empty do begin
    pop t, the top element, off the stack;
    for each state u with edge from t to u labeled ∈ do
        if u is not in ∈-closure(T) do begin
            add u to ∈-closure(T) ;
            push u onto stack
    end
end
```

Algorithm For Subset Construction – (2)

initially, $\in$-closure(s_0) is only (unmarked) state in Dstates;

`while` there is unmarked state T in Dstates `do begin`

    mark T;

    `for` each input symbol *a* `do begin`

        U := $\in$-closure(*move*(T,*a*));

        `if` U is not in Dstates `then`

            add U as an unmarked state to Dstates;

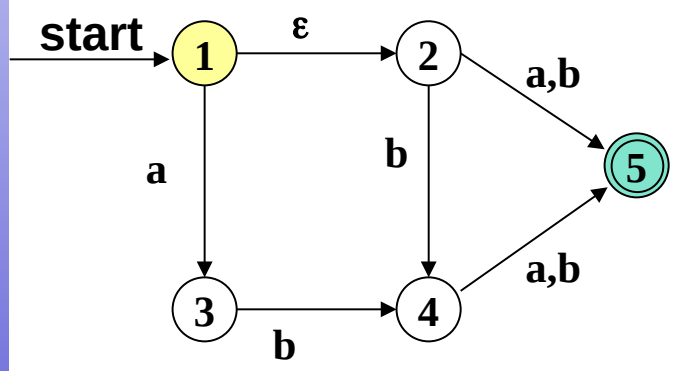
        Dtran[T,*a*] := U

    `end`

`end`

Example 2: Subset Construction

NFA



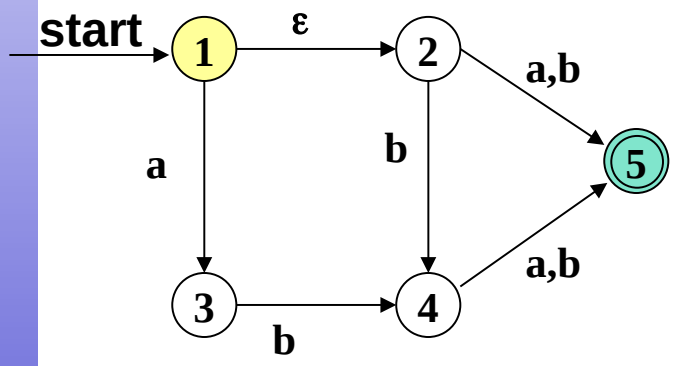
NFA N with

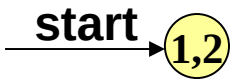
- **State set S_N = {1,2,3,4,5},**
- **Alphabet Σ = {a,b}**
- **Start state s_N=1,**
- **Final states F_N={5},**
- **Transition function T_N: S_N x {Σ ∪ ε} → S_N**

	a	b	ε
1	3	-	2
2	5	5, 4	-
3	-	4	-
4	5	5	-
5	-	-	-

Example 2: Subset Construction

NFA

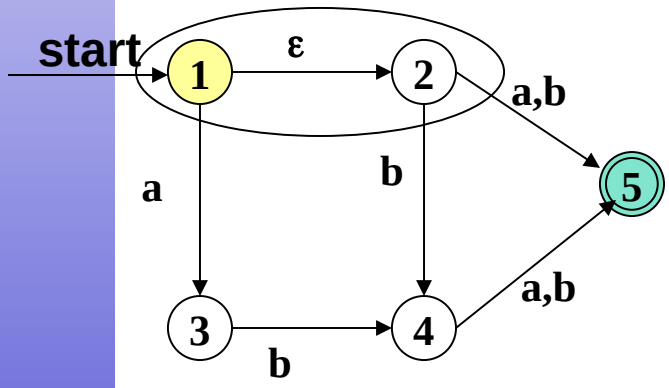


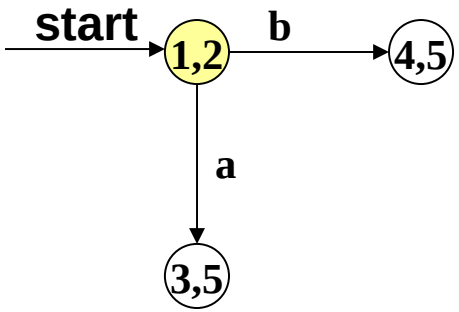


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}		

Example 2: Subset Construction

NFA

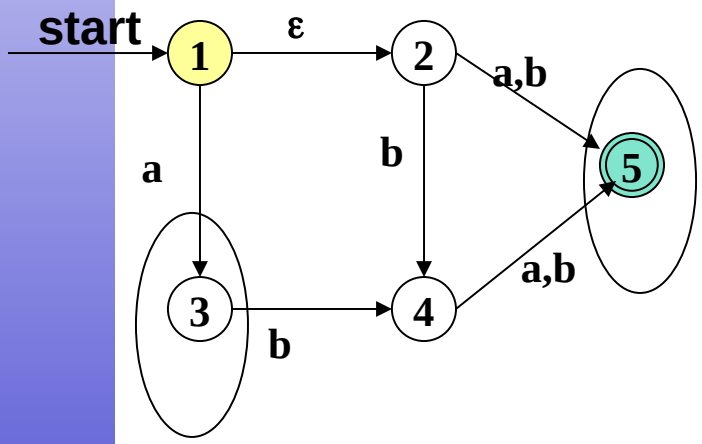


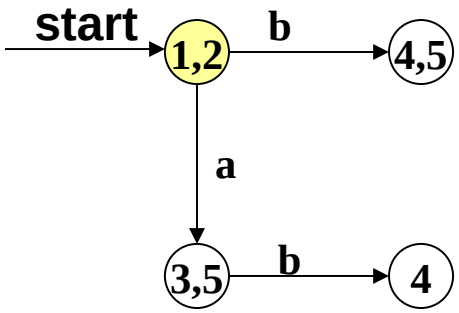


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}		
{4,5}		

Example 2: Subset Construction

NFA

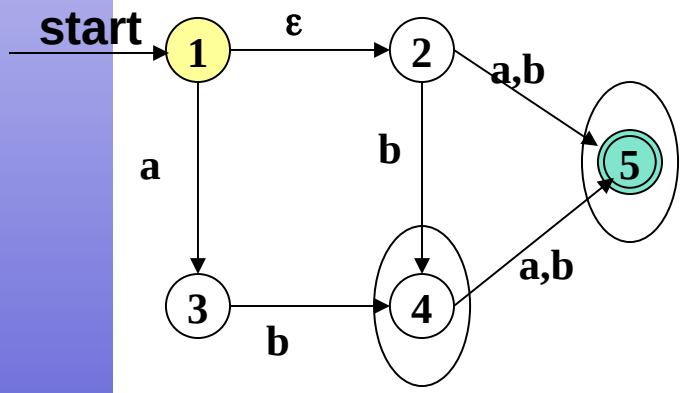


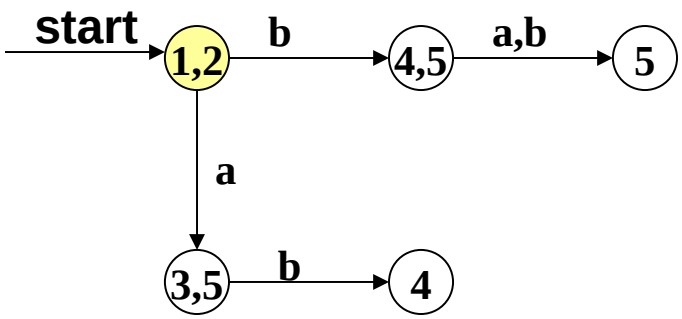


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}		
{4}		

Example 2: Subset Construction

NFA

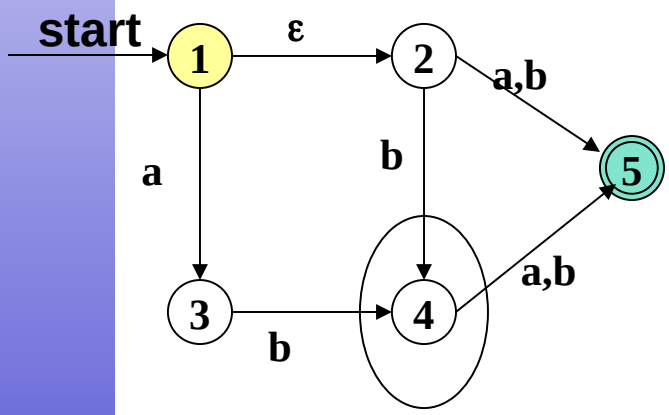


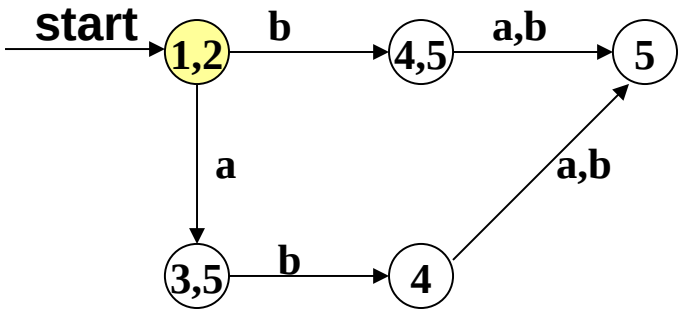


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}		
{5}		

Example 2: Subset Construction

NFA

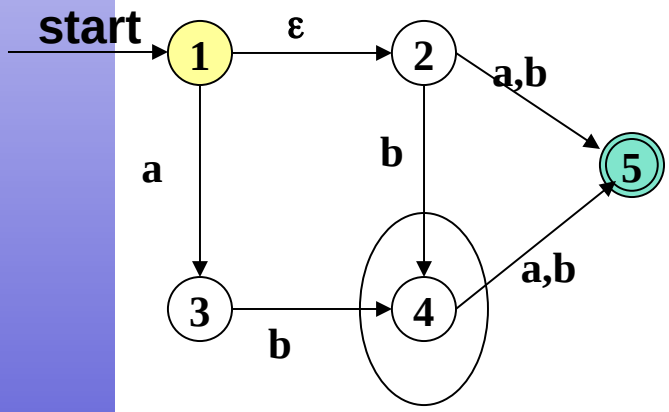


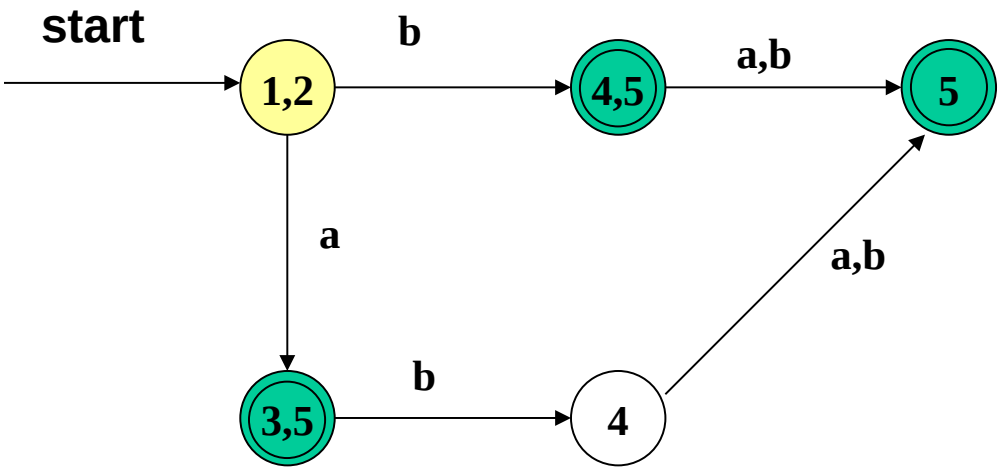


T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

Example 2: Subset Construction

NFA



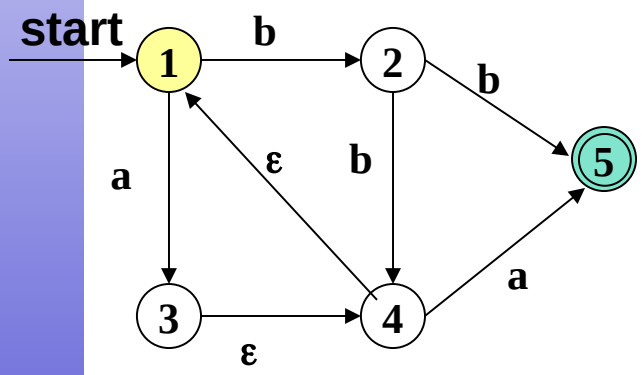


All final states since the NFA final state is included

T	∈-closure(move(T, a))	∈-closure(move(T, b))
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

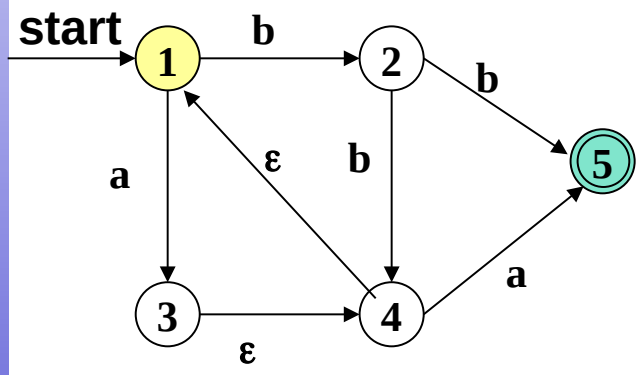
Example 3: Subset Construction

NFA

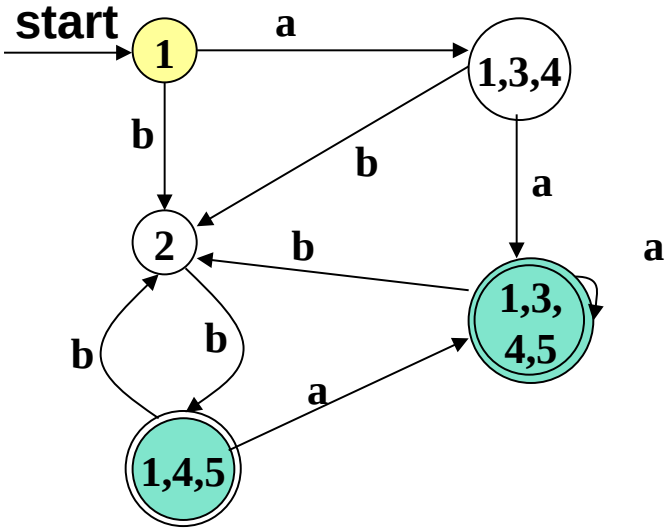


Example 3: Subset Construction

NFA

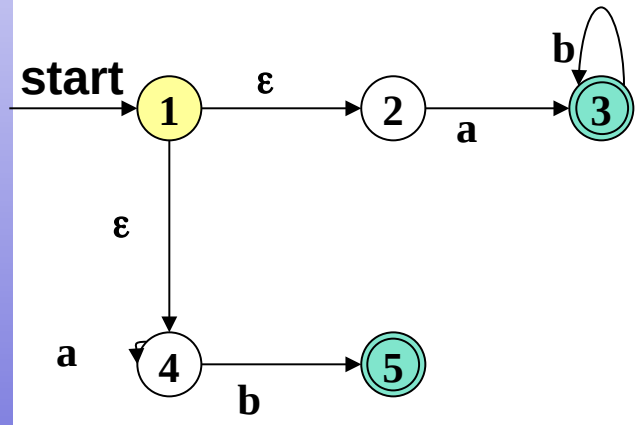


DFA

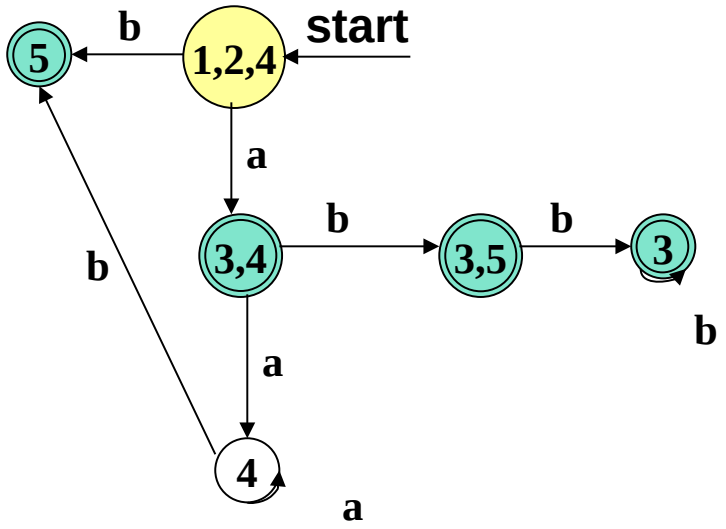


Example 4: Subset Construction

NFA

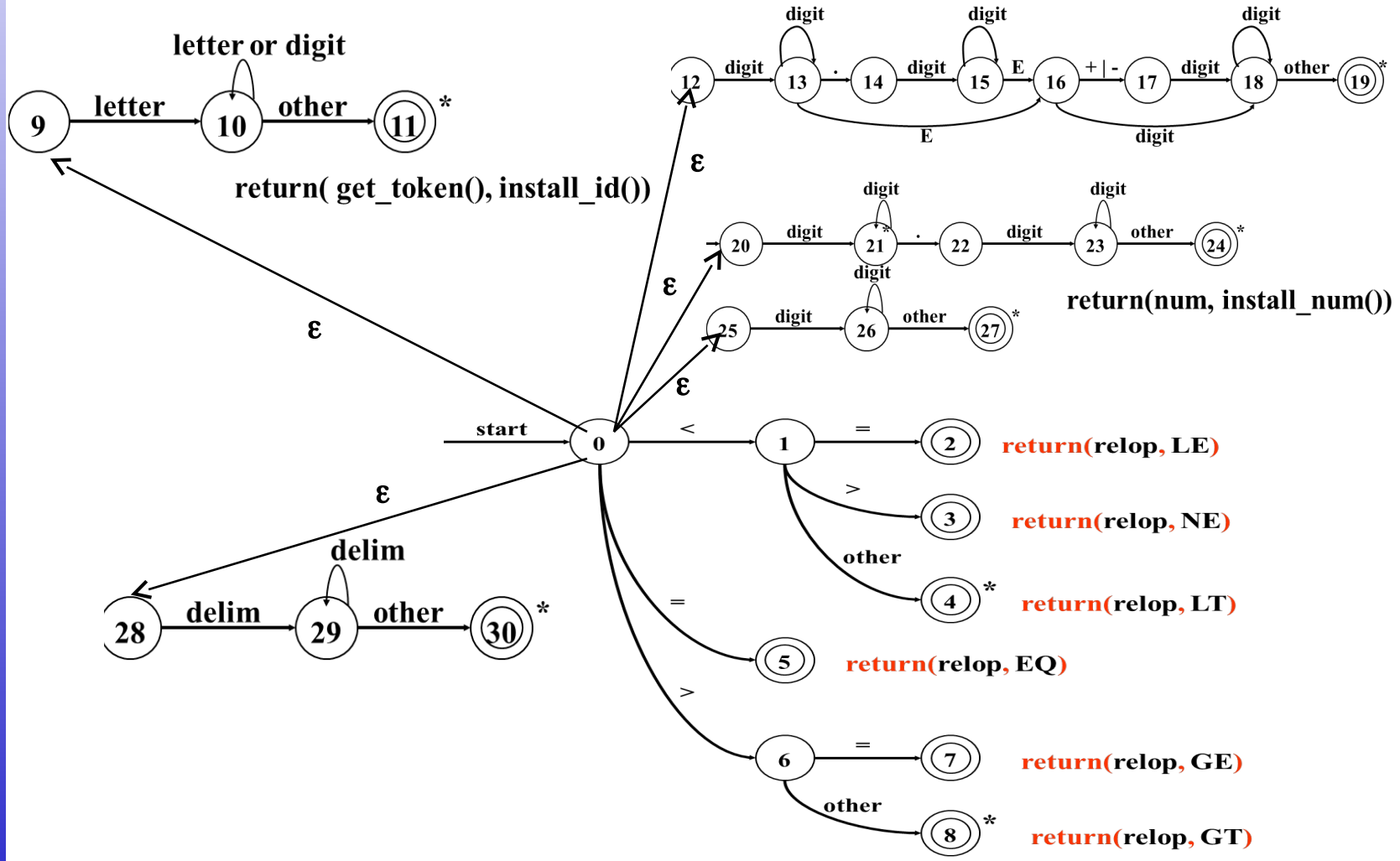


DFA



NFA: Transition Diagram capturing Multiple Tokens

Questions: **Draw the transition diagram that will recognize id, keywords, number, whitespaces and relational operators?**



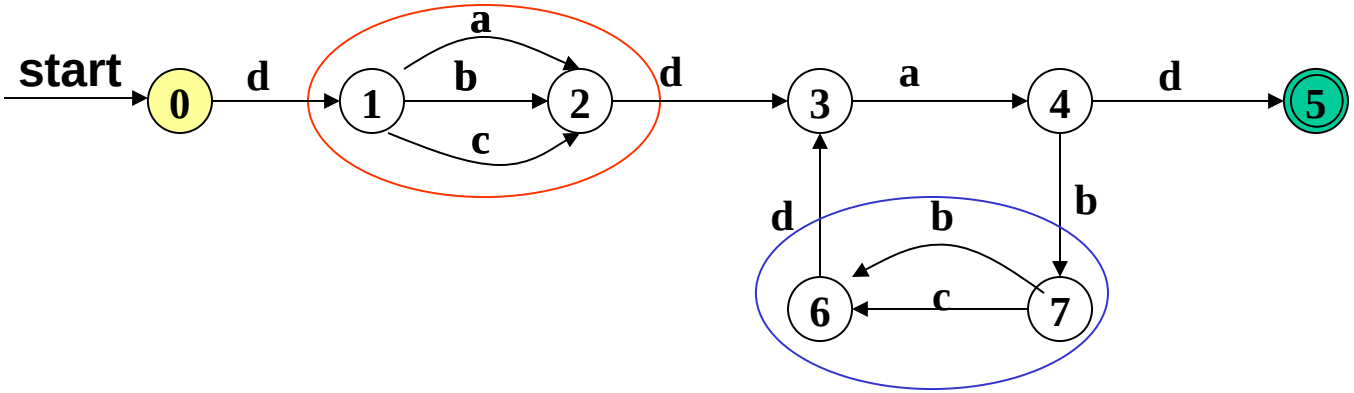
Corresponding DFA: Transition Diagram capturing Multiple Tokens

Questions: **Draw the transition diagram that will recognize id, keywords, number, whitespaces and relational operators?**

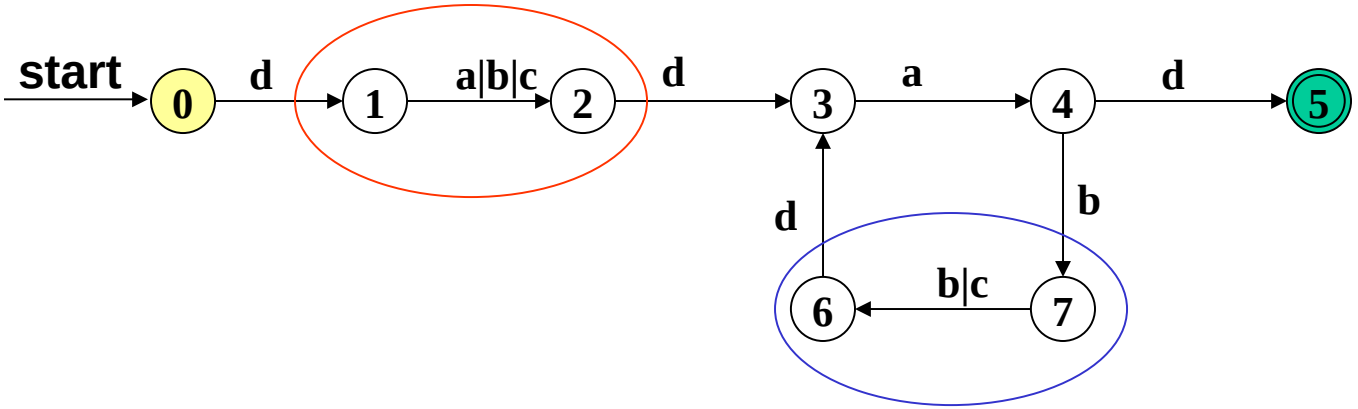
Converting DFAs to REs

1. Combine serial links by concatenation
2. Combine parallel links by alternation
3. Remove self-loops by Kleene closure
4. Select a node (other than initial or final) for removal. Replace it with a set of equivalent links whose path expressions correspond to the in and out links
5. Repeat steps 1-4 until the graph consists of a single link between the entry and exit nodes.

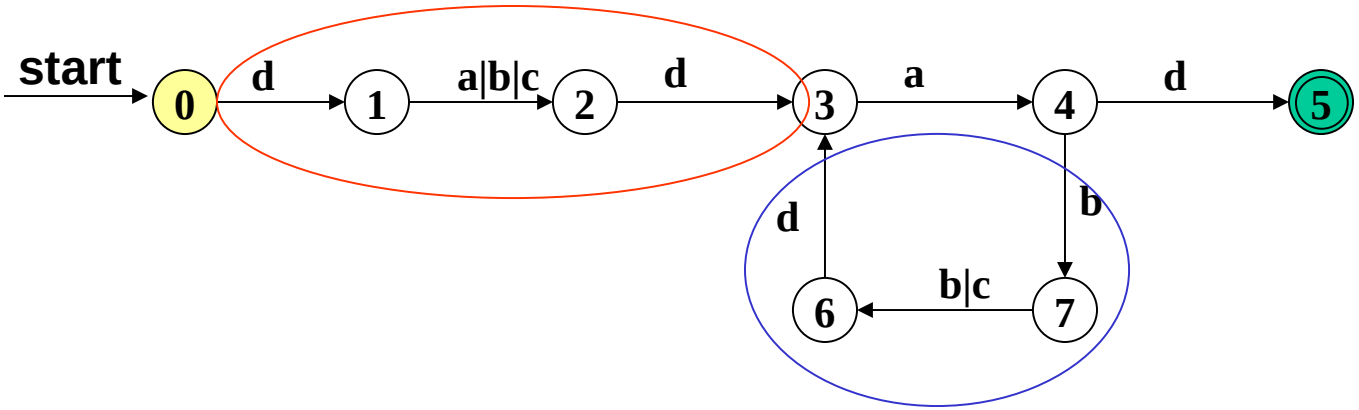
Example: DFAs to REs



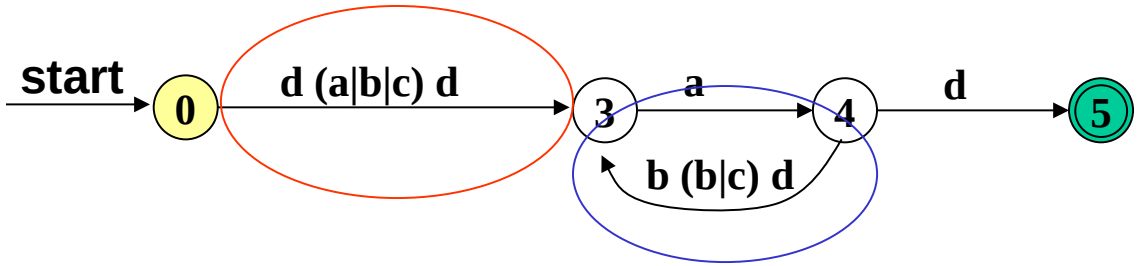
parallel edges become alternation



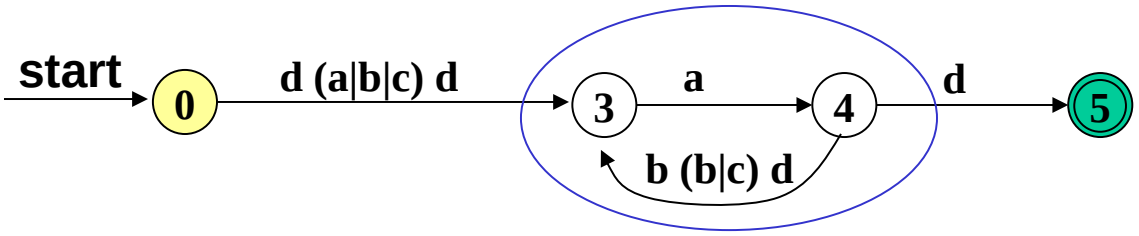
Example: DFAs to REs



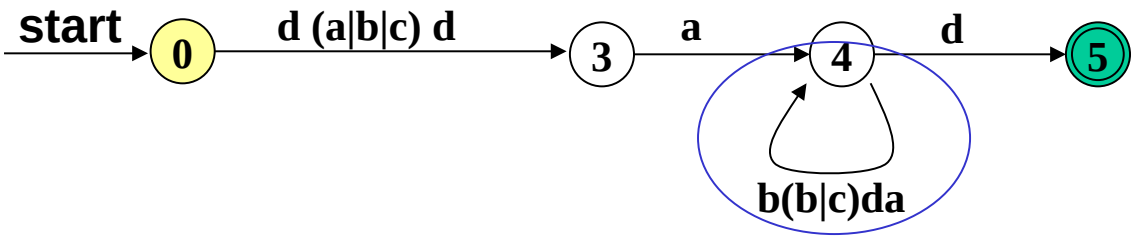
serial edges become concatenation



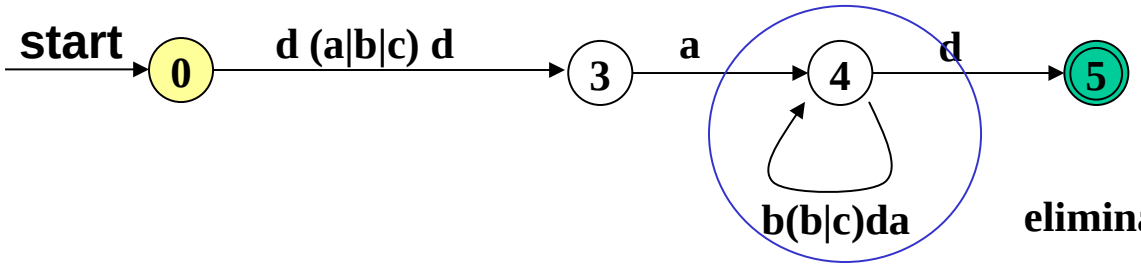
Example: DFAs to REs



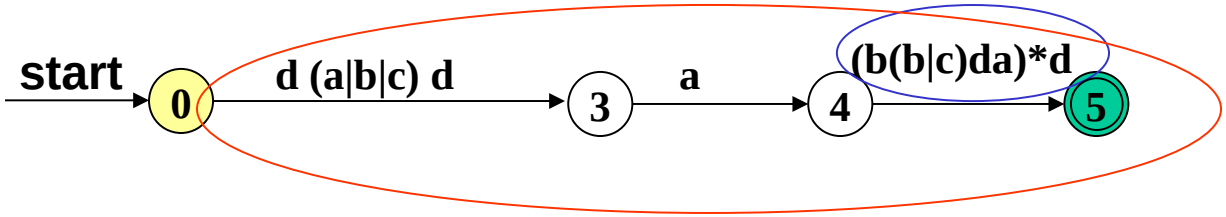
Find paths that can be "shortened"



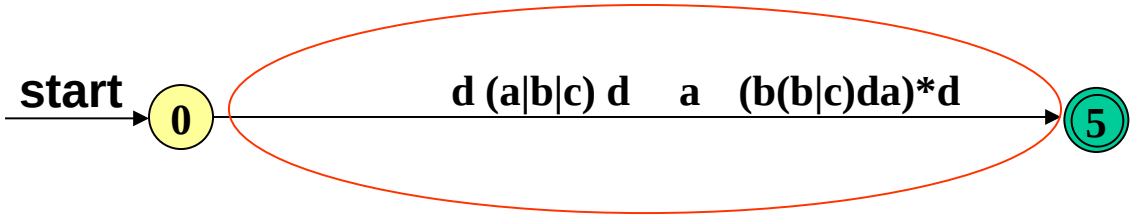
Example: DFAs to REs



eliminate self-loops



serial edges become concatenation



Exercises for Practice

Dragon Book (Chapter-3):

3.3.5, 3.4.1, 3.4.2, 3.6.2, 3.6.3, 3.6.4, 3.6.5, 3.7.1, 3.7.2, 3.7.3