Lexical Analysis

Responsibilities

Scans the input program

Removes white spaces

Removes comments

Extracts and identifies tokens

Generates lexical errors

Passes tokens to parser

Terminologies

Token: classification for a common set of strings

Examples: Identifier, Integer, Float, Operator,....

Pattern: The rules that characterize the set of strings for a token

Examples: [0-9]+

Lexeme: Actual sequence of characters that matches a pattern and has a given Token class

Examples:

Identifier: sum, data, x

Integer: 345, 2, 0, 629,....

Lexical Errors

Error Handling is localized with respect to the input source code

For example: fi (a == f(x)) … generates no lexical error in C

In what situations do errors occur?

Prefix of remaining input does not match any defined token

Possible error recovery actions:

1. **Deleting or Inserting Input Characters**
2. **Replacing or Transposing Characters**
3. **Or, skip over to next separator to *ignore* problem**

Overall strategy

Use one character of *look-ahead*

Perform a case analysis

1) **Based on lookahead char**
2) **Based on current lexeme**

Outcome

1) **If char can extend lexeme, continue.**
2) **If char cannot extend lexeme, find what the complete lexeme is (upto the previous character) and return its token. Put the lookahead back into the symbol stream.**

Regular expressions

ε is a regular expression, L(ε) = {ε}

If a is a symbol in ∑, then a is a regular expression, L(a) = {a}

(r) | (s) is a regular expression denoting the language L(r) ∪ L(s)

(Strings from both languages)

(r)(s) is a regular expression denoting the language L(r)L(s)

(Strings constructed by concatenating a string from the first language with a string from the second language)

(r)* is a regular expression denoting (L(r))*

(Each string in the language is a concatenation of any number of strings in the language of s)

(r) is a regular expression denoting L(r)

Regular expressions

Examples:

letter → A | B | … | Z | a | b | … | Z | _

digit → 0 | 1 | … | 9

id → letter (letter | digit)*

Regular expressions

Extensions

One or more instances: (r)+

Zero of one instances: r?

Character classes: [abc]

Example:

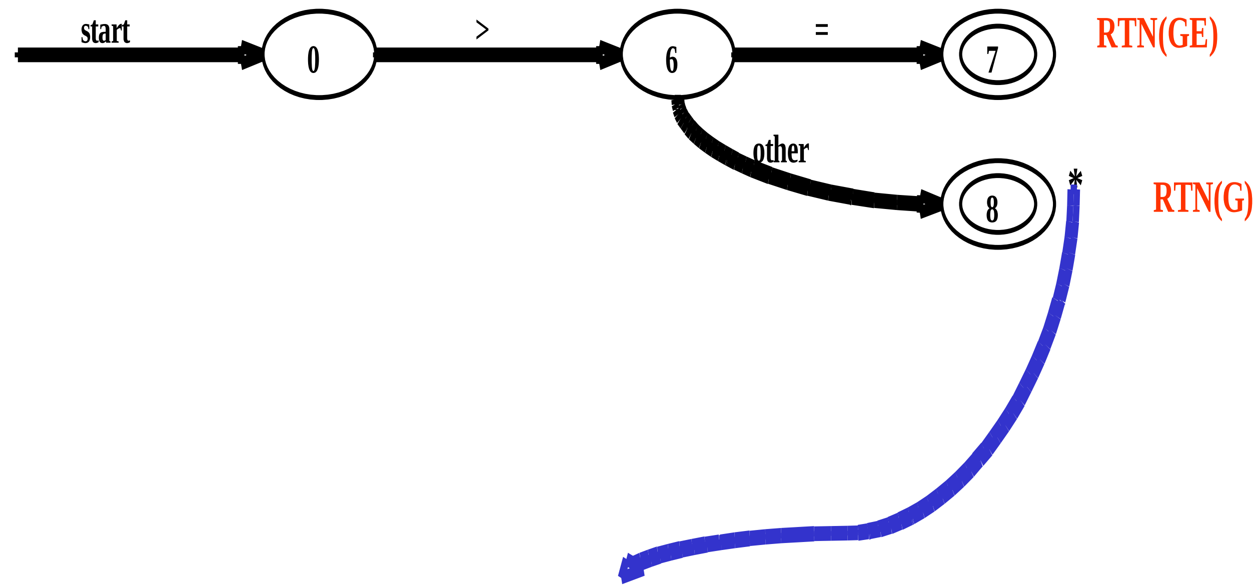
letter → [A-Za-z]

digit → [0-9]

id → letter(letter|digit)*

Transition diagrams

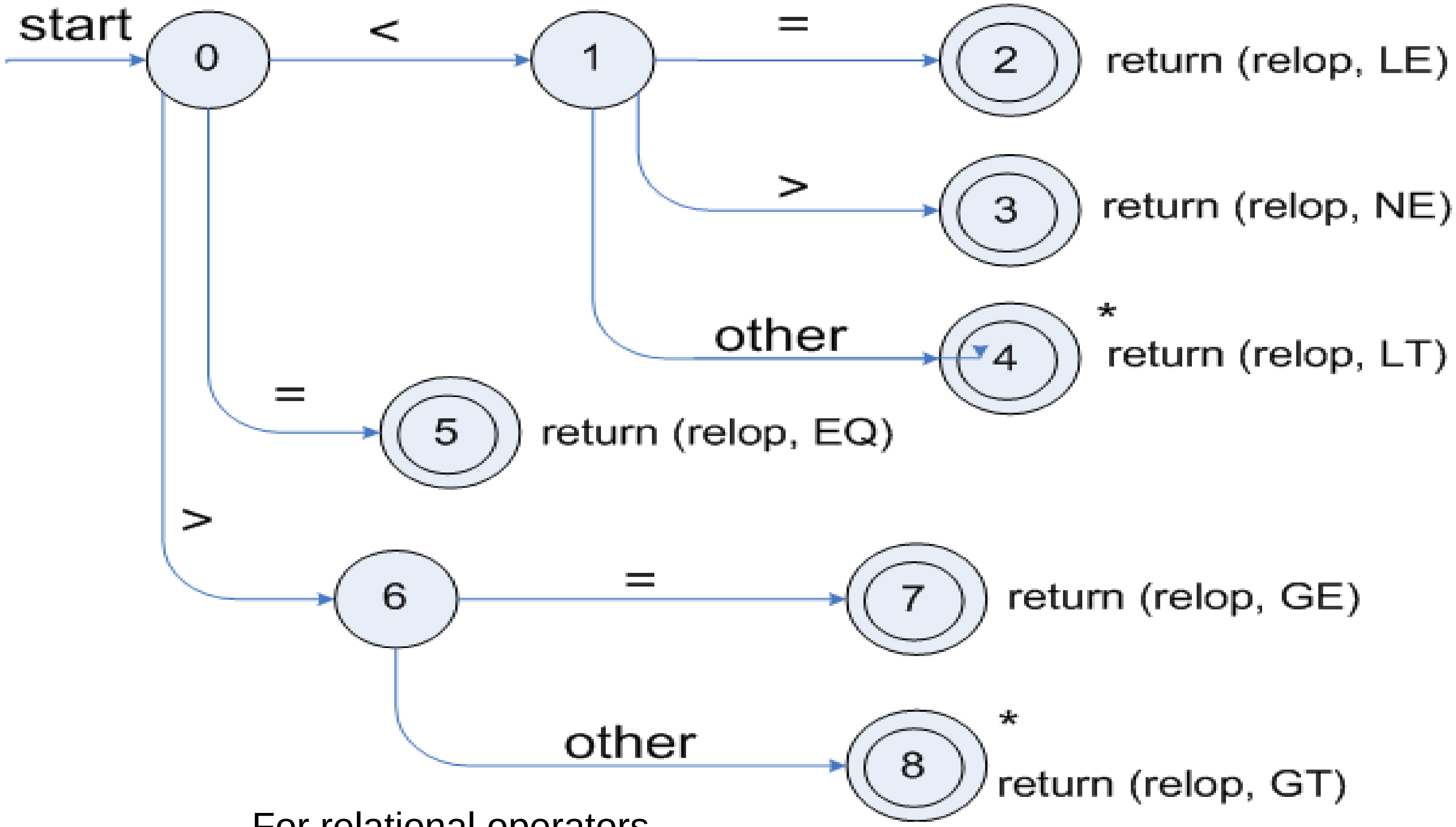
>=



We Õve accepted Ņ > Ó and have read other char that must be unread.

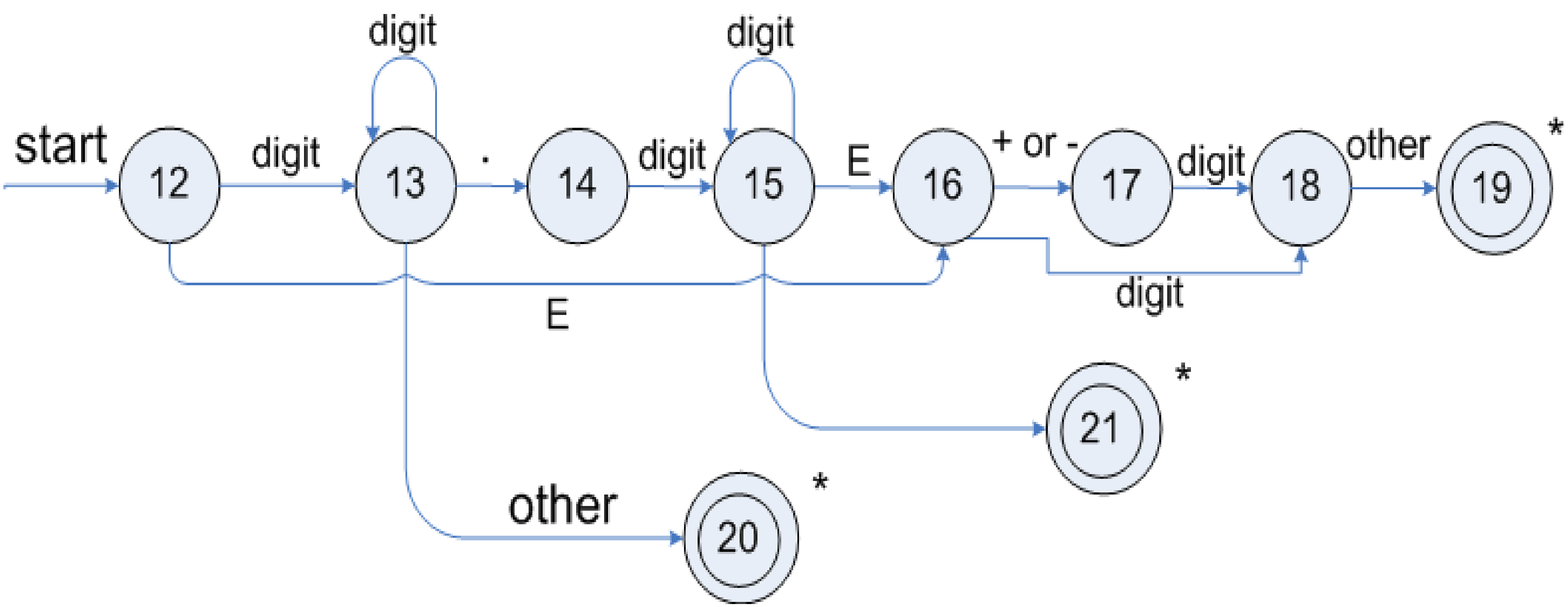
An additional character is read, that needs to be unread(return to input buffer

Transition diagrams



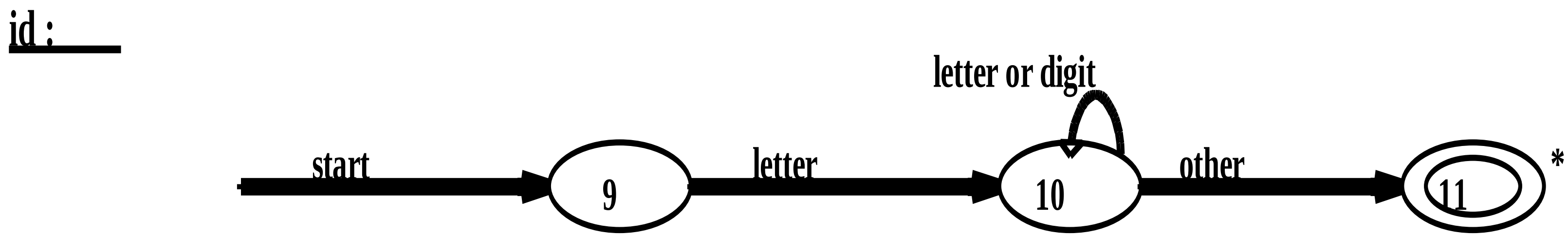
For relational operators

Transition diagrams



For unsigned numbers

Transition diagrams



For identifiers

What to do for keywords?

- Use the “Identifier” token
- After a match, lookup the keyword table
- If found, return a token for the matched keyword
- If not, return a token for the *true* identifier

Recognising Tokens

Regular expressions provide specifications for the tokens in a language

Finite automaton is used to recognise a token – implementation

A finite automaton consists of

An input alphabet Σ

A set of states S

A start state n

A set of accepting states $F \subseteq S$

A set of transitions state $\xrightarrow{\text{input}}$ state

Finite Automaton

Deterministic Finite Automata (DFA)

One transition per input per state

No ε-moves

Nondeterministic Finite Automata (NFA)

Can have multiple transitions for one input in a given state

Can have ε-moves

***Finite* automata have *finite* memory**

Need only to encode the current state

Conversion can be automated

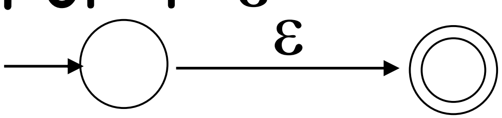
NFAs and DFAs recognize the same set of languages (regular languages)

DFAs are easier to implement

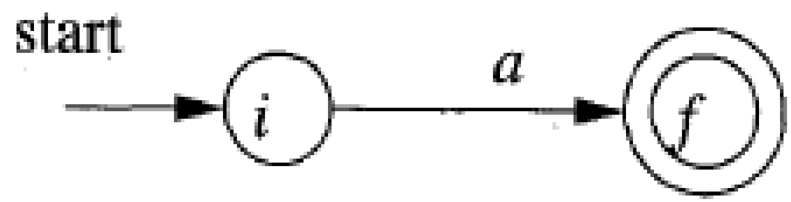
Regular Expressions to NFA
McNaughton-Yamada-Thompson Algorithm

For each kind of regular expression, define an NFA

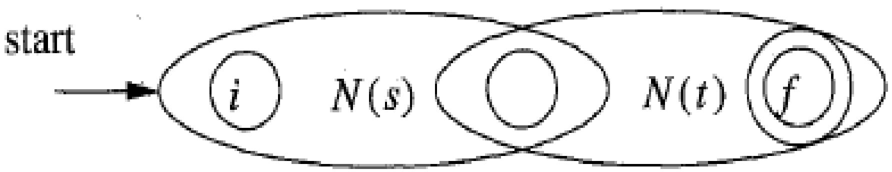
For r=ε



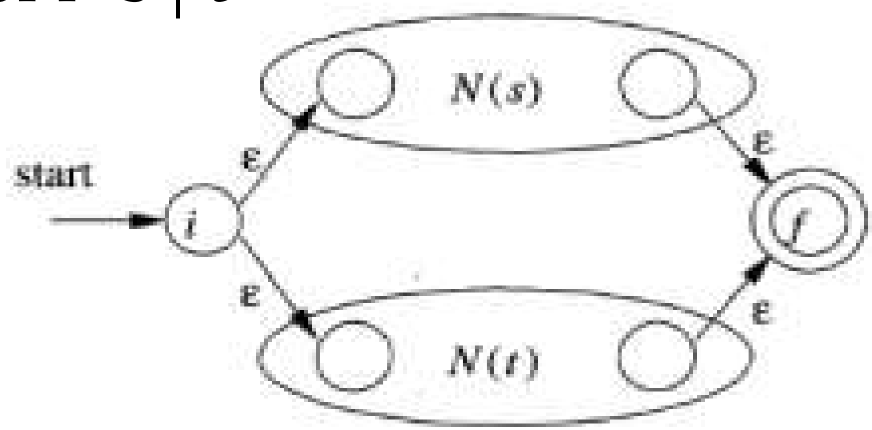
For r=a



For r=st

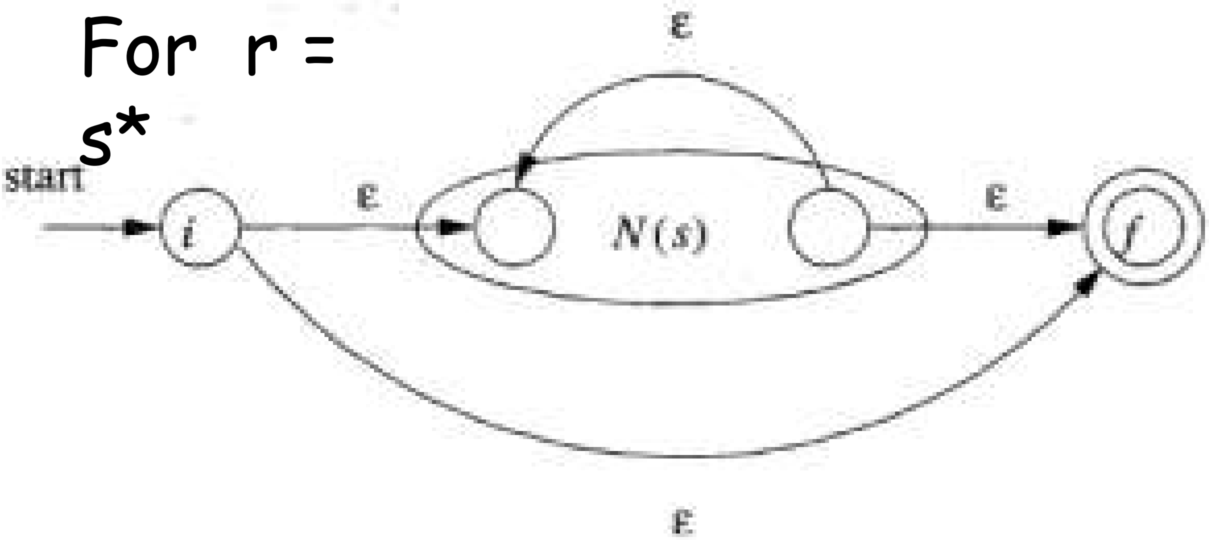


For r=s | t



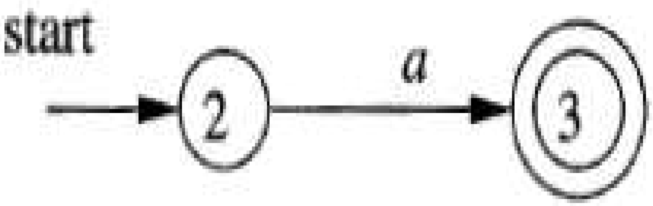
Regular Expressions to NFA
McNaughton-Yamada-Thompson Algorithm

For r = s*

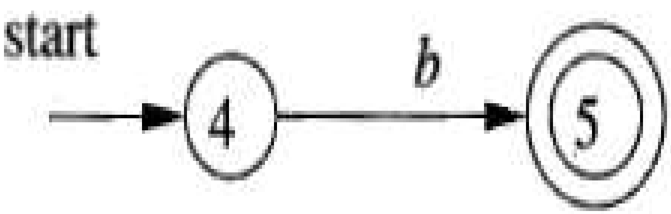


Example: (a|b)*abb

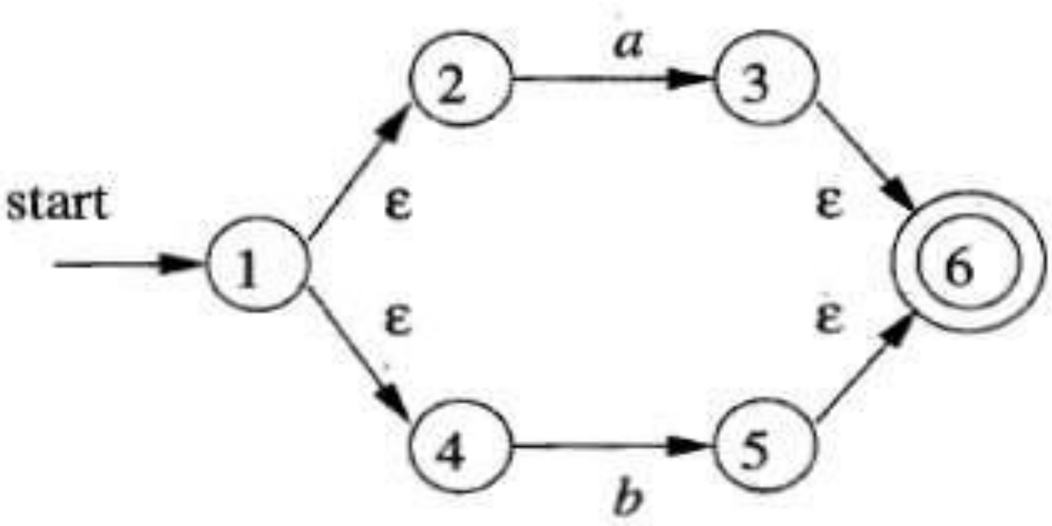
a



b

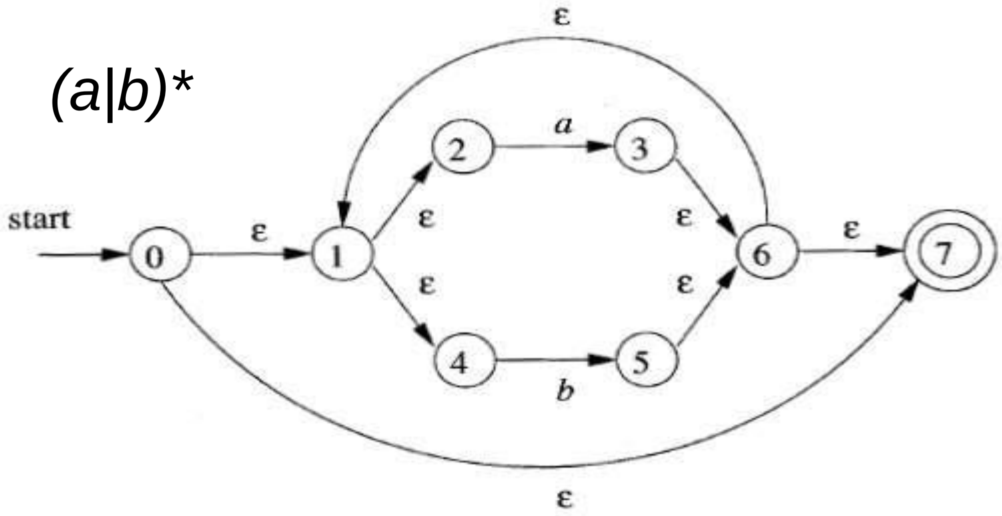


a | *b*

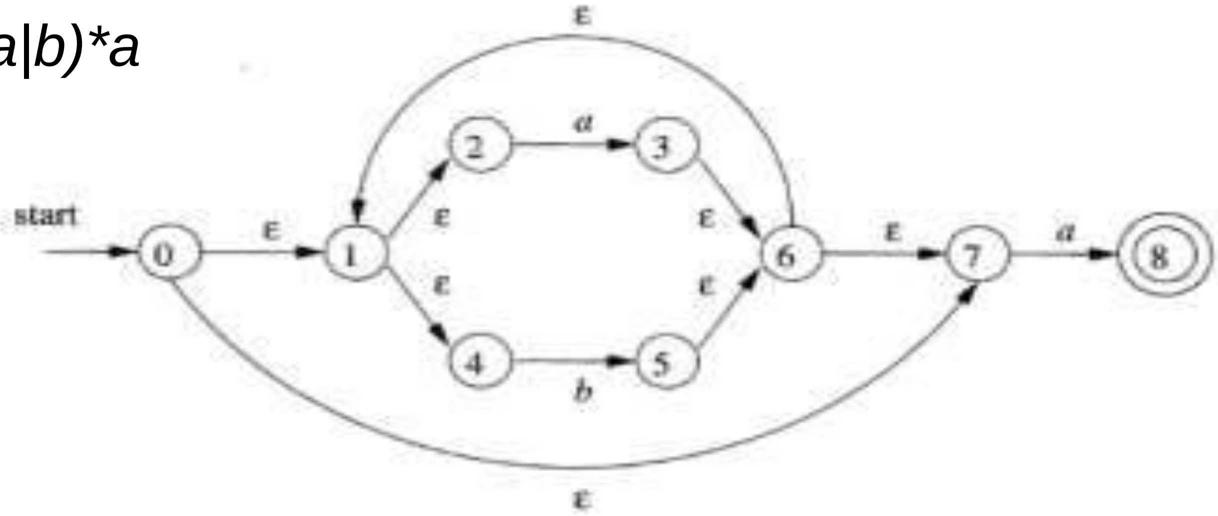


Example: (a|b)*abb

*(a|b)**

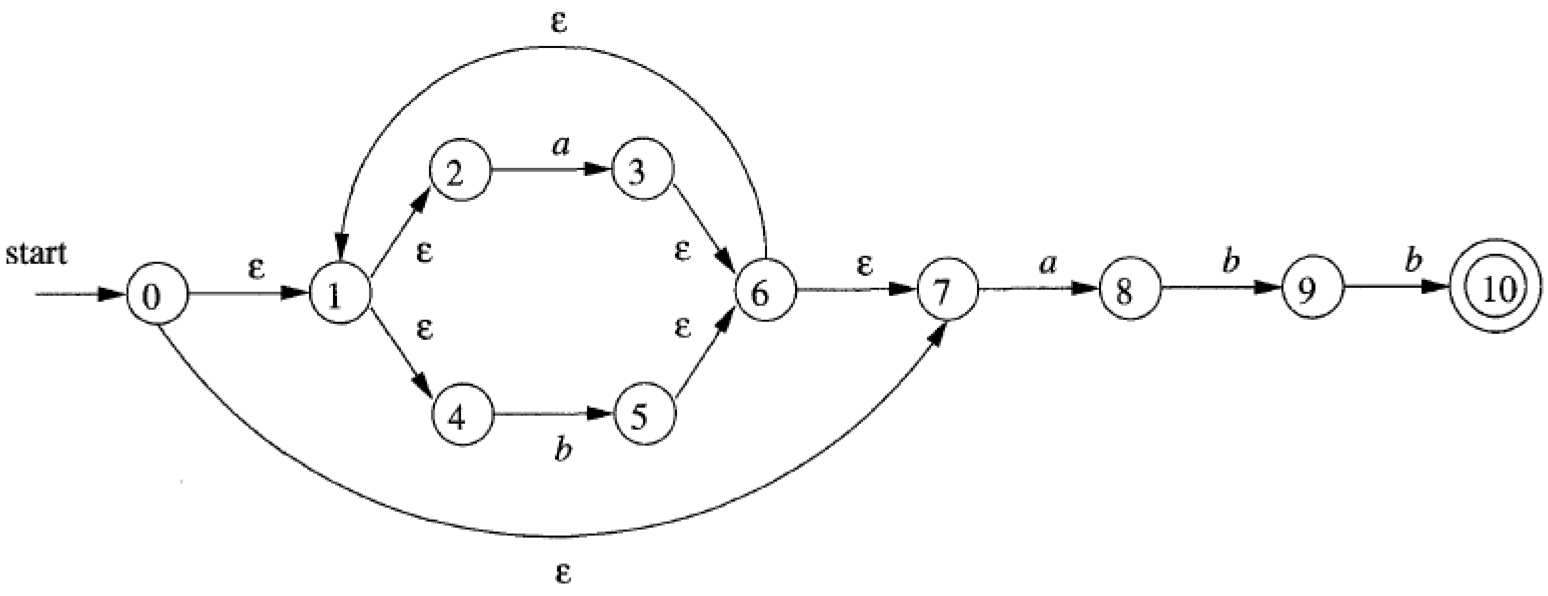


*(a|b)*a*



Example: (a|b)*abb

*(a|b)*abb*



Converting NFA to DFA

- Subset construction: each state of DFA corresponds to a set of NFA states
- For real languages NFA and DFA have approximately the same number of states (though not theoretically)

Definitions

Operation	Description
$\epsilon\text{-}closure(s)$	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
$\epsilon\text{-}closure(T)$	Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone; $= \cup_{s \text{ in } T}\ \epsilon\text{-}closure(s)$.
$move(T, a)$	Set of NFA states to which there is a transition on input symbol a from some state s in T.

Simulating NFA

```
S = ε-closure(s0);
c = nextChar();
while ( c != eof ) {
        S = ε-closure(move(S, c));
        c = nextChar();
}
if ( S ∩ F != ∅ ) return "yes";
else return "no";
```

Computing ε-closure

- Push all states of T onto stack;
- Initialize ε-closure(T) to T;
- while (stack is not empty) {
  - pop t, the top element of the 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
    - }
- }

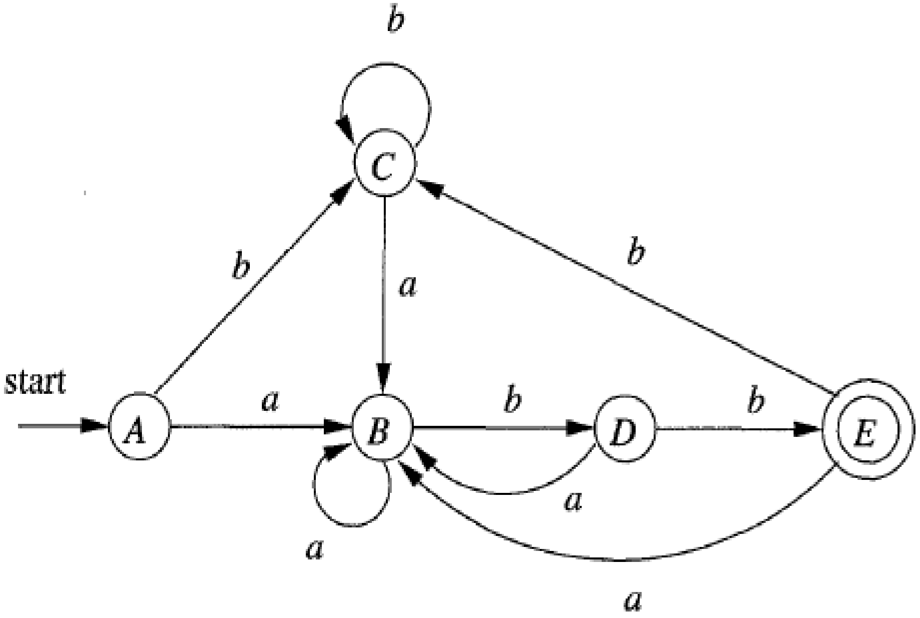
Subset Constructions

```
initially, ε-closure(s0) is the only state in Dstates, and it is unmarked;
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```

States of the DFA we are constructing

The resulting DFA

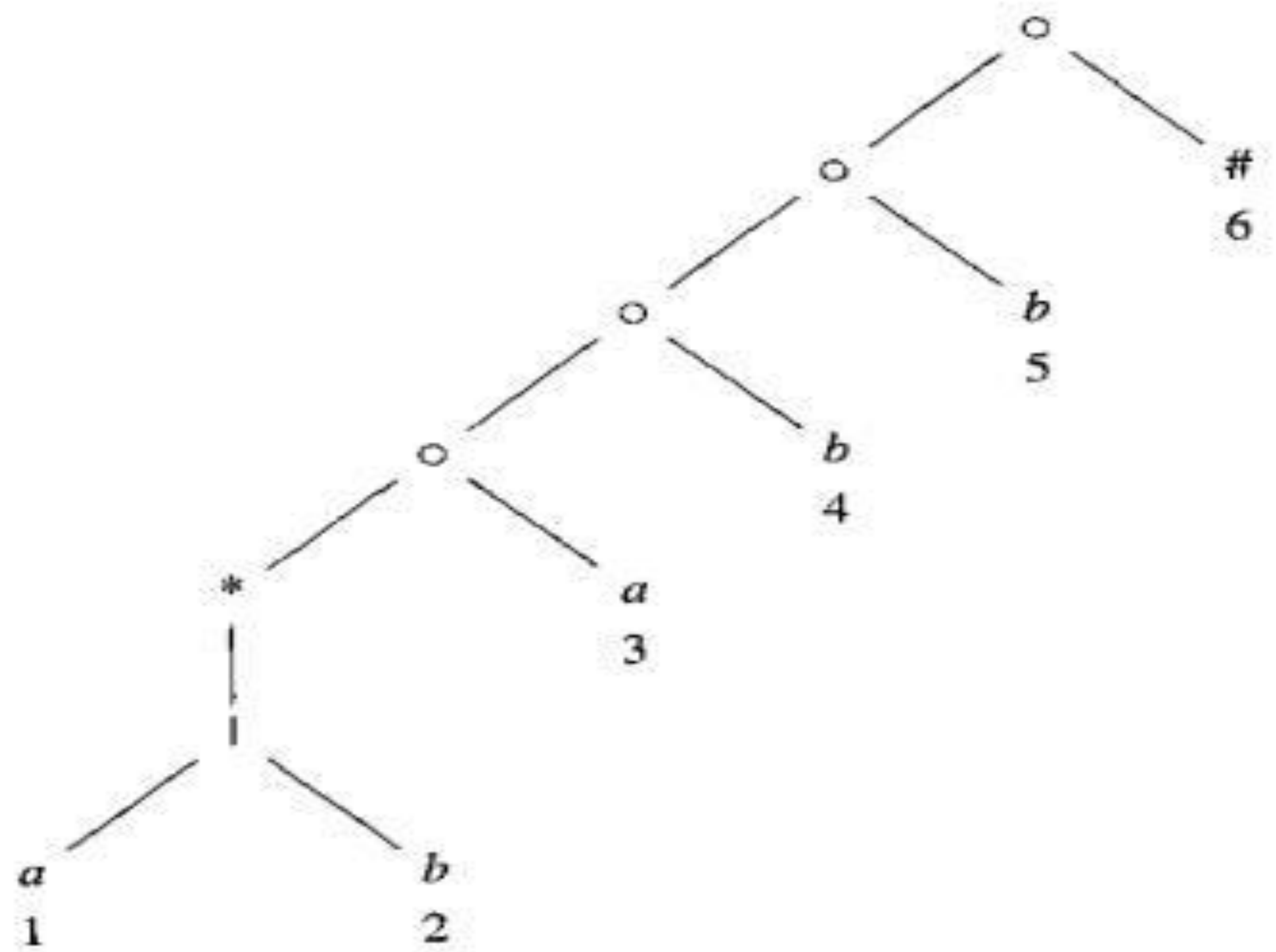
NFA STATE	DFA STATE	a	b
$\{0,1,2,4,7\}$	A	B	C
$\{1,2,3,4,6,7,8\}$	B	B	D
$\{1,2,4,5,6,7\}$	C	B	C
$\{1,2,4,5,6,7,9\}$	D	B	E
$\{1,2,3,5,6,7,10\}$	E	B	C



Construction of DFA directly from RE

- Augment the regular expression r with a special end symbol # to make accepting states important
  - the new expression is r#
- Construct a syntax tree for r#
- Attach a unique integer to each node that is not labeled by ε

Syntax tree for regular expression (a|b)*abb

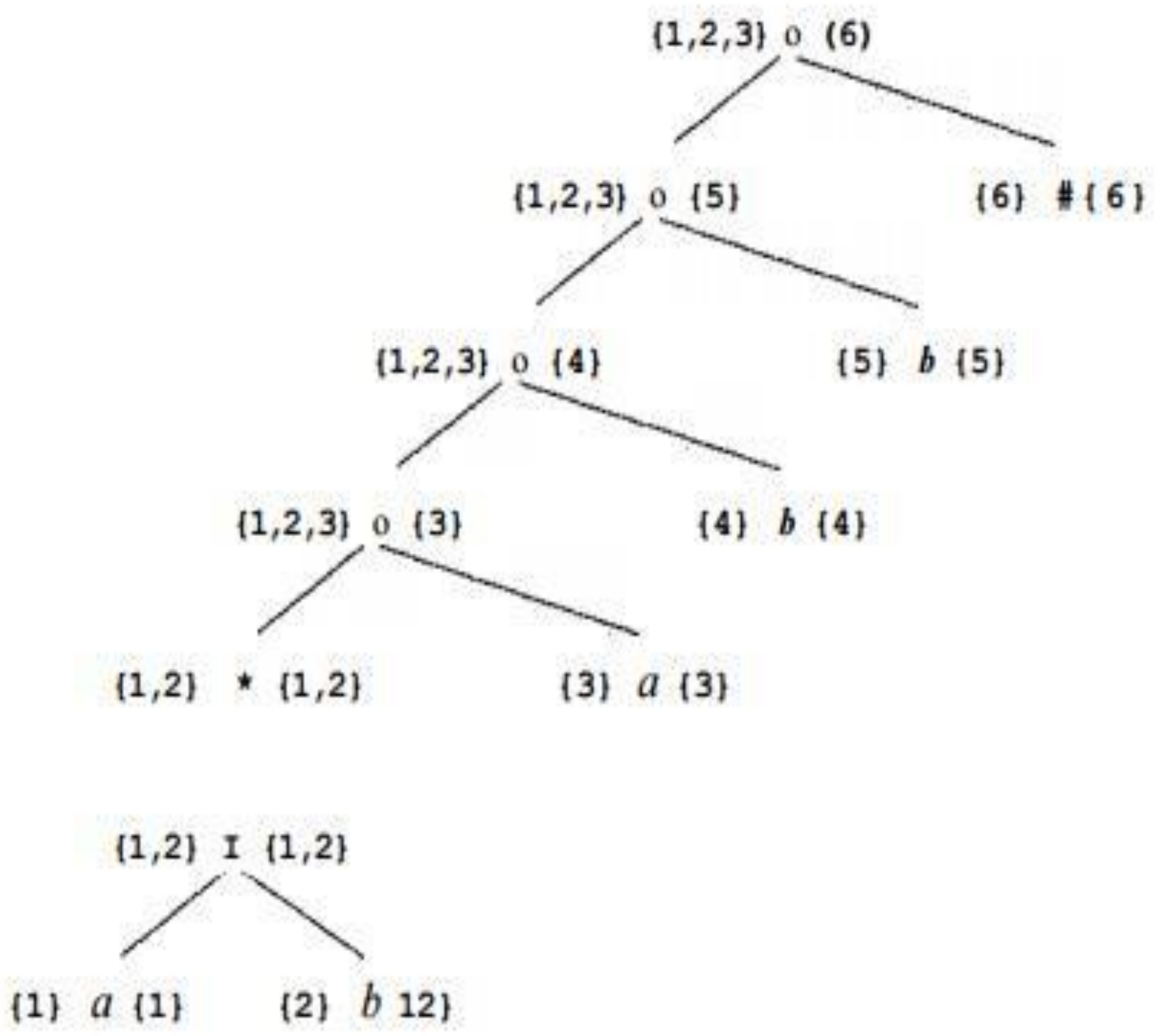


Annotating the tree

- Traverse the tree to construct functions nullable, firstpos, lastpos, and followpos
- For a node n, let L(n) be the language generated by the subtree with root n
- nullable(n): L(n) contains the empty string ε
- firstpos(n): set of positions under n that can match the first symbol of a string in L(n)
- lastpos(n): the set of positions under n that can match the last symbol of a string in L(n)
- followpos(i): the set of positions that can follow position i in any generated string

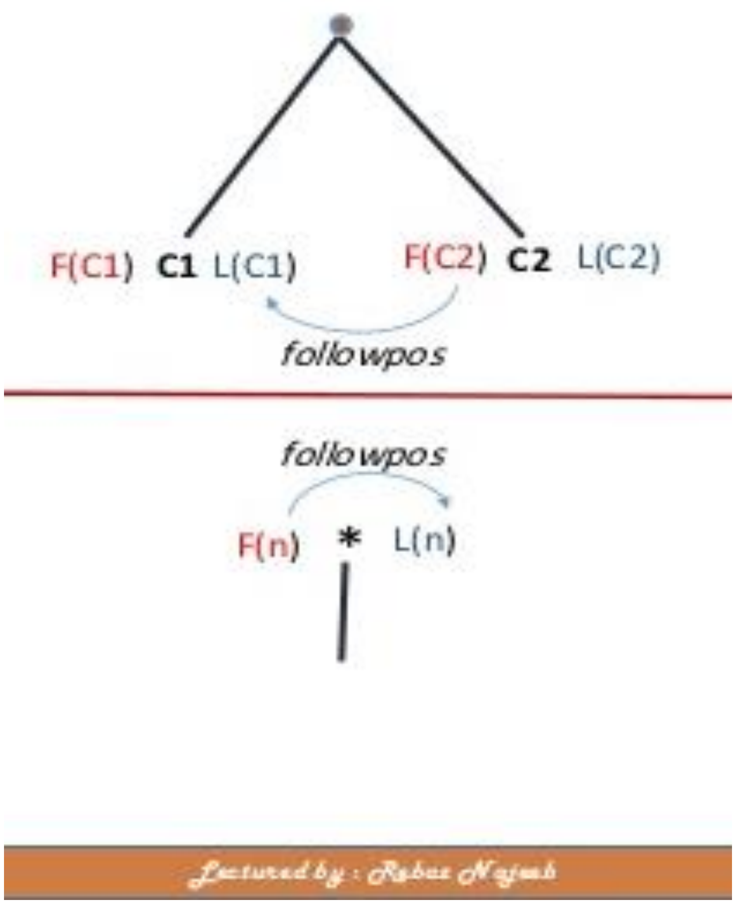
Computation of nullable, firstpos, lastpos

Node n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
Leaf ε	true	$\varnothing$	$\varnothing$
Leaf i	false	$\{i\}$	$\{i\}$
| / \\ c_1 c_2	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1)$ $\cup$ $firstpos(c_2)$	$lastpos(c_1)$ $\cup$ $lastpos(c_2)$
• / \\ c_1 c_2	$nullable(c_1)$ and $nullable(c_2)$	**if** $nullable(c_1)$ **then** $firstpos(c_1) \cup$ $firstpos(c_2)$ **else** $firstpos(c_1)$	**if** $nullable(c_2)$ **then** $lastpos(c_1) \cup$ $lastpos(c_2)$ **else** $lastpos(c_2)$
* | c_1	true	$firstpos(c_1)$	$lastpos(c_1)$


{1,2,3} o {6}
{1,2,3} o {5}
{6} # {6}
{1,2,3} o {4}
{5} b {5}
{1,2,3} o {3}
{4} b {4}
{1,2} * {1,2}
{3} a {3}
{1,2} I {1,2}
{1} a {1}
{2} b 12}

Evaluating followpos

- Two-rules for followpos:
  - If *n* is concatenation-node with left child c_1 and right child c_2, and *i* is a position in lastpos (c_1), then all positions in firstpos(c_2) are in followpos(*i*).
  - If *n* is a star-node, and *i* is a position in lastpos(*n*), then all positions in firstpos(*n*) are in followpos(*i*).



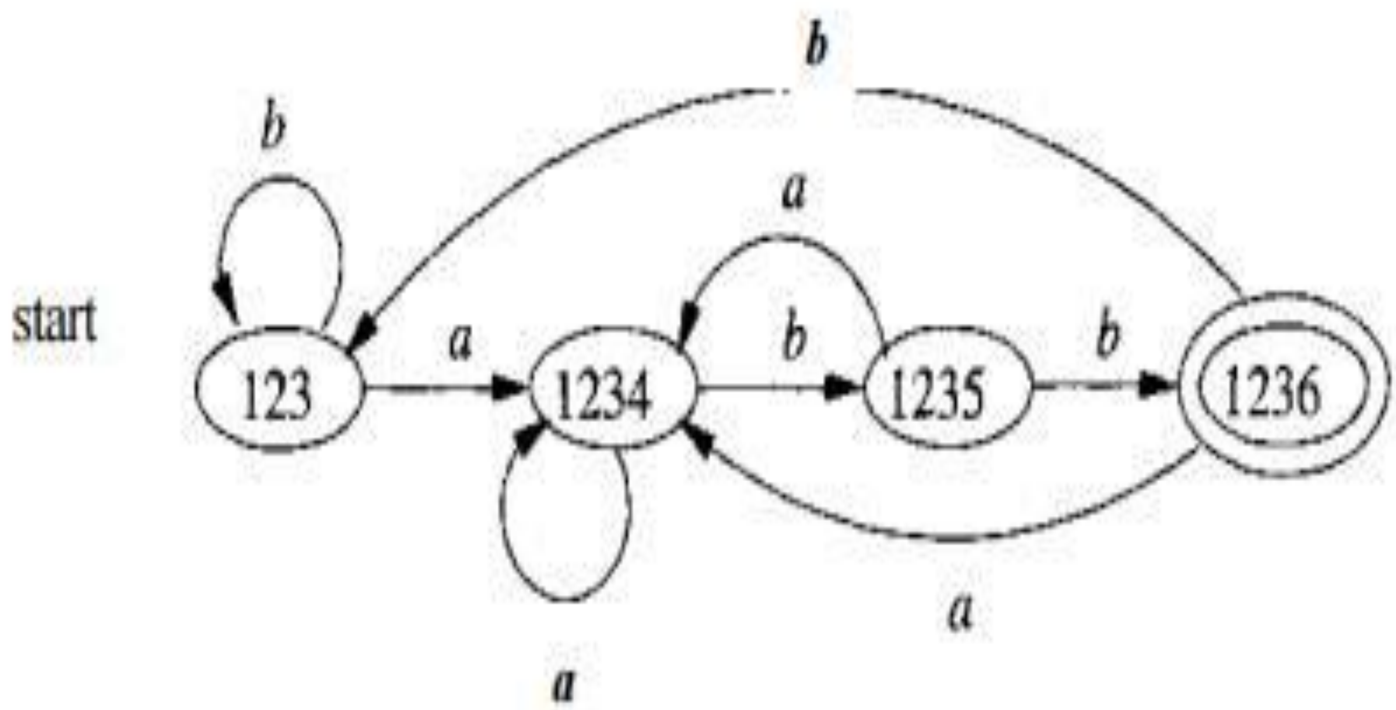
- After computing firstpos and lastpos for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

NODE n	*followpos(n)*
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	0

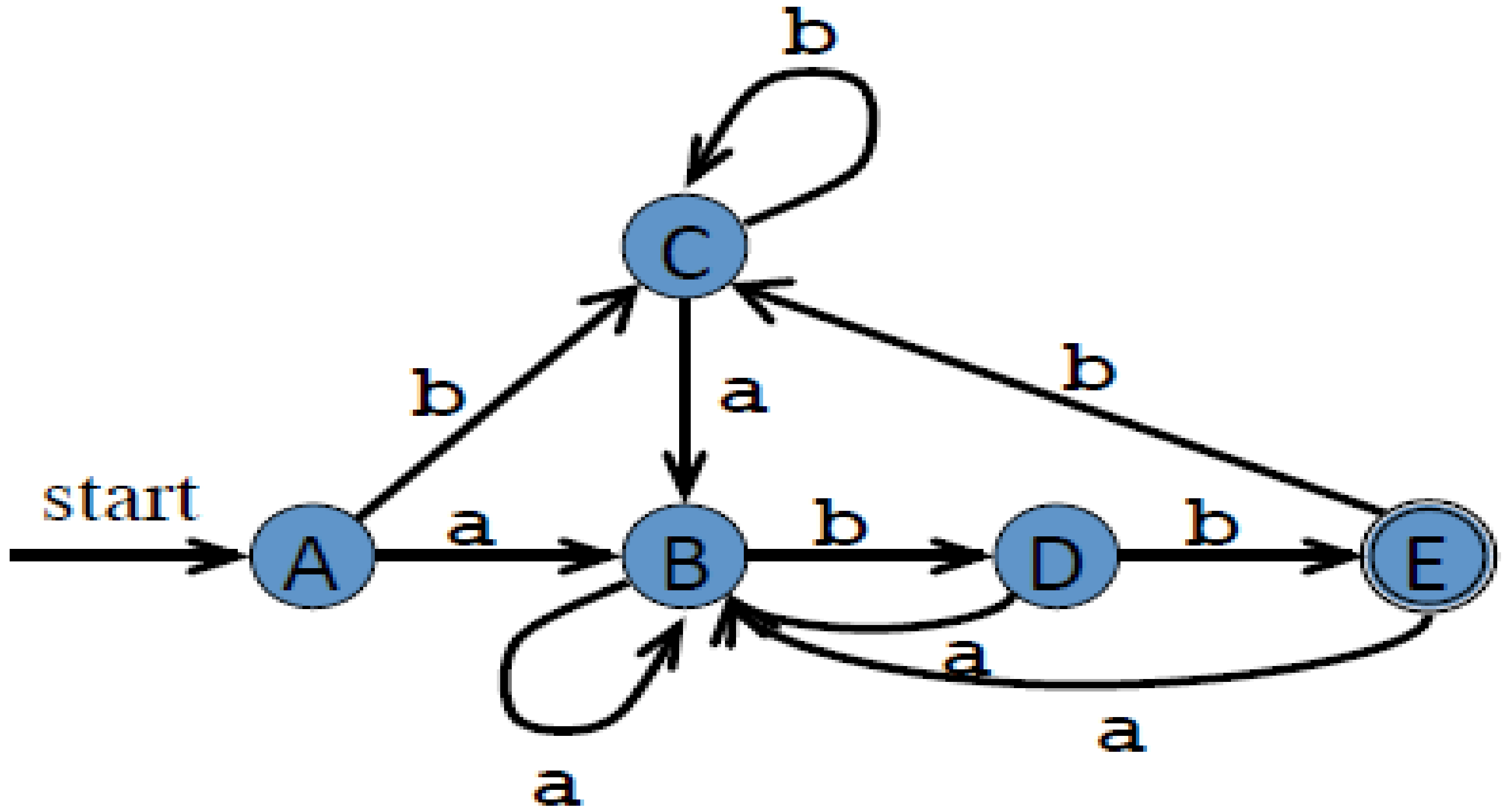
The Algorithm

- Initialize Dstates to contain only the unmarked state firstpos(n_o), where n_o is the root of syntax tree T for (r) #
- while (there is an unmarked state S in Dstates) {

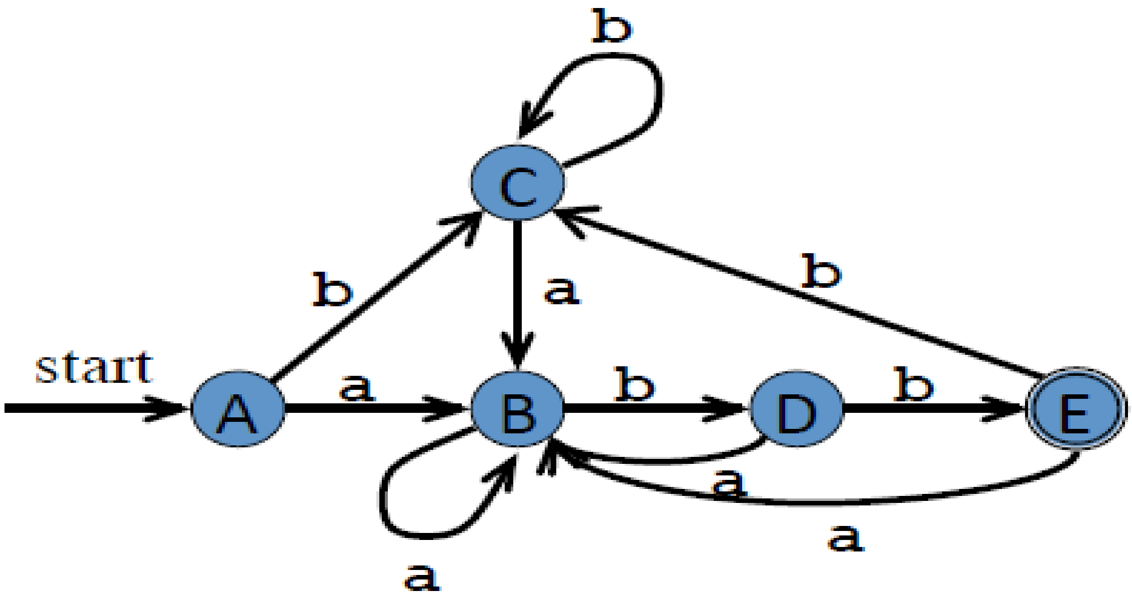
```
 mark  S;
 for ( each input symbol a ) {
      let  U be the union of followpos(p)  for all p in S
      that correspond to a;
      if ( U is not in Dstates )
            add U as an unmarked state to Dstates; Dtran[S, a] = U;
      }
 }
```

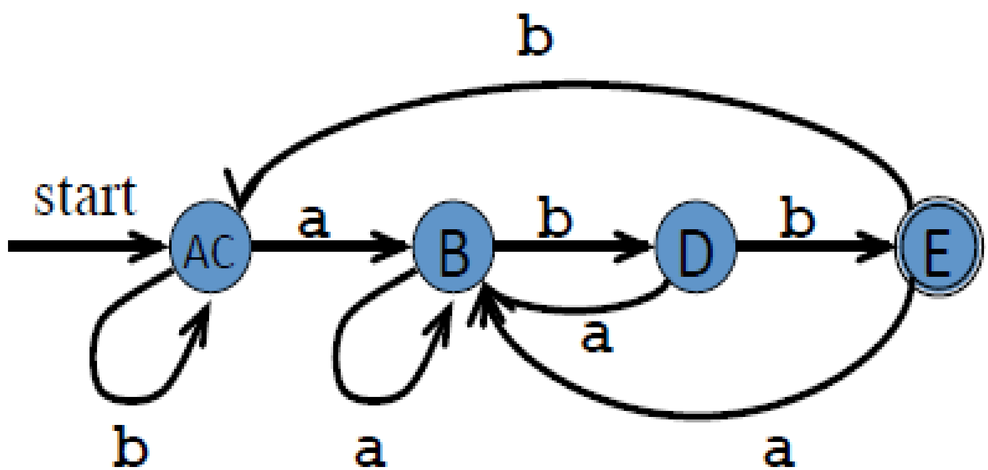


start
123
1234
1235
1236
a
b



b
C
b
a
b
start
a
b
b
A
B
D
E
a
a
a


start
A
B
C
D
E
a
b


start
AC
B
D
E
a
b

Minimizing number of DFA states

For any regular language, there is always a unique minimum state DFA, which can be constructed from any DFA of the language.

Algorithm:

Partition the set of states into two groups:

G 1 : set of accepting states

G 2 : set of non accepting states

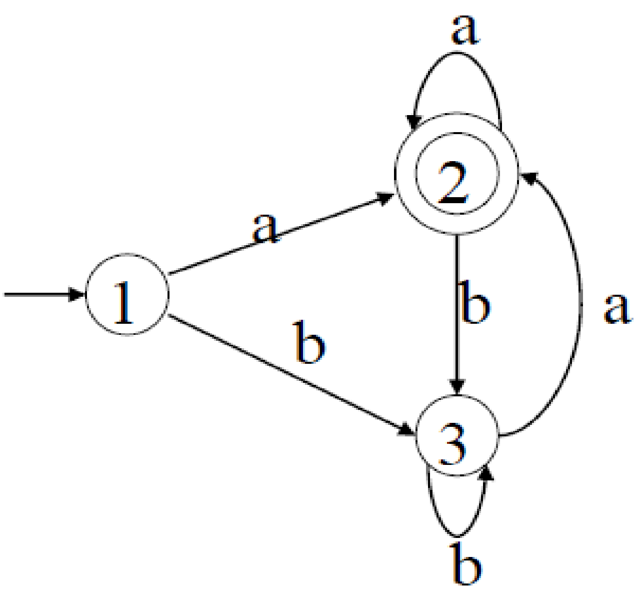
For each new group G

partition G into subgroups such that states s1 and s2 are in the same group, iff for all input symbols *a*, states s1 and s2 have transitions to states in the same group

Start state of the minimized DFA is the group containing the start state of the original DFA

Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Minimizing DFA –Example (1)

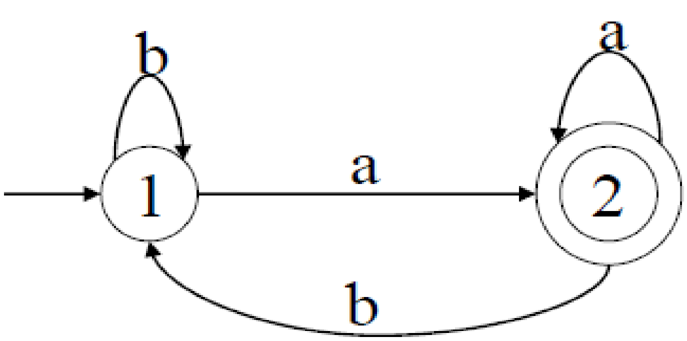


$G_1 = \{2\}$
$G_2 = \{1,3\}$

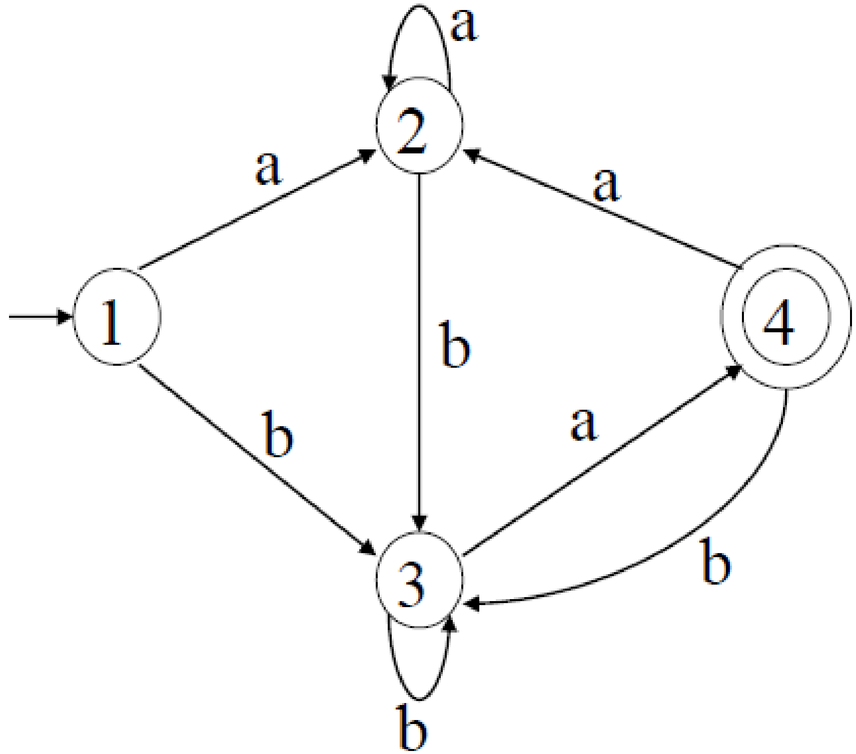
G_2 cannot be partitioned because

Dtran[1,a]=2 Dtran[1,b]=3
Dtran[3,a]=2 Dtran[3,b]=3

So, the minimized DFA (with minimum states) is



Minimizing DFA –Example (2)



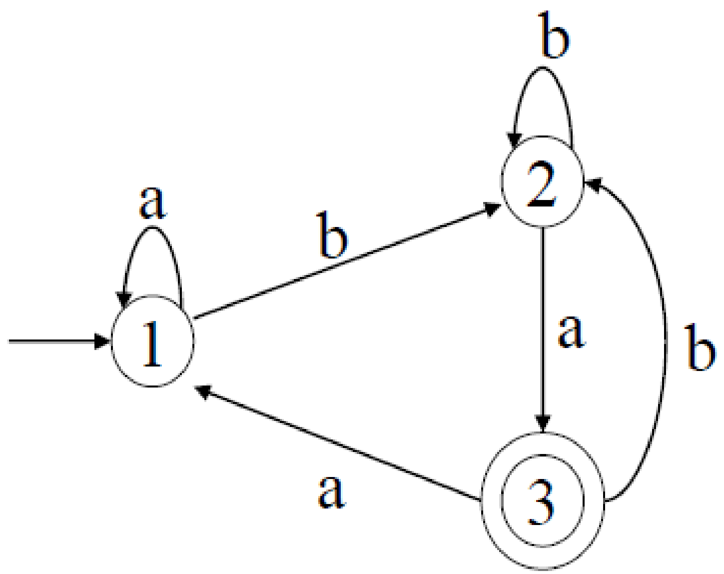
Groups: {1,2,3} {4}

{1,2} {3}

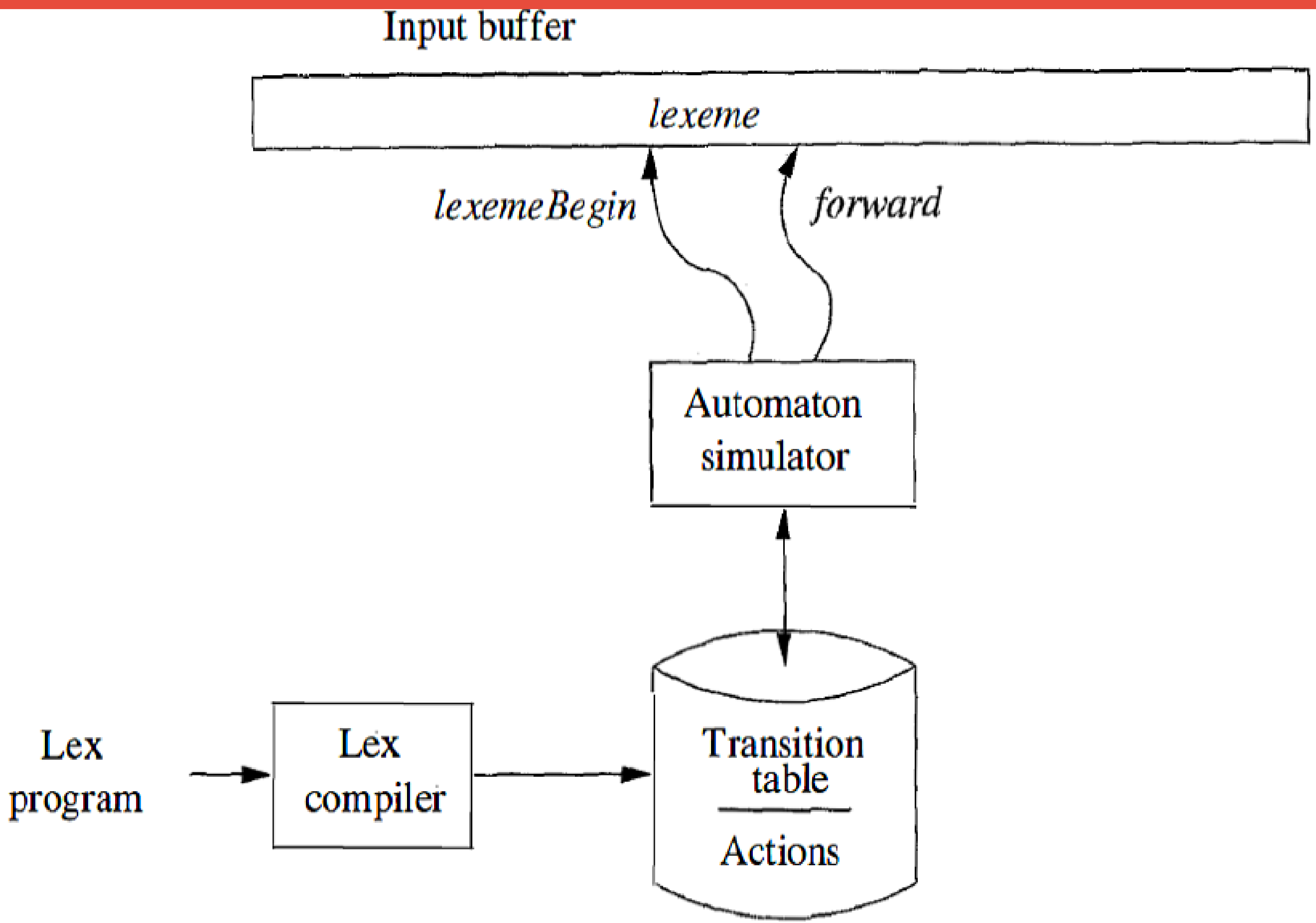
no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

Minimized DFA



Architecture of a Lexical Analyzer



C(L|D)*LLS

1) Construct NFA and then DFA using subset construction
2) Construct DFA directly from the RE