CC Lectures 3-4-5-6

Compiled for: 7th Sem, CE, DDU

Compiled by: Niyati J. Buch

Ref. : Compilers: Principles, Techniques, and Tools, 2nd Edition
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

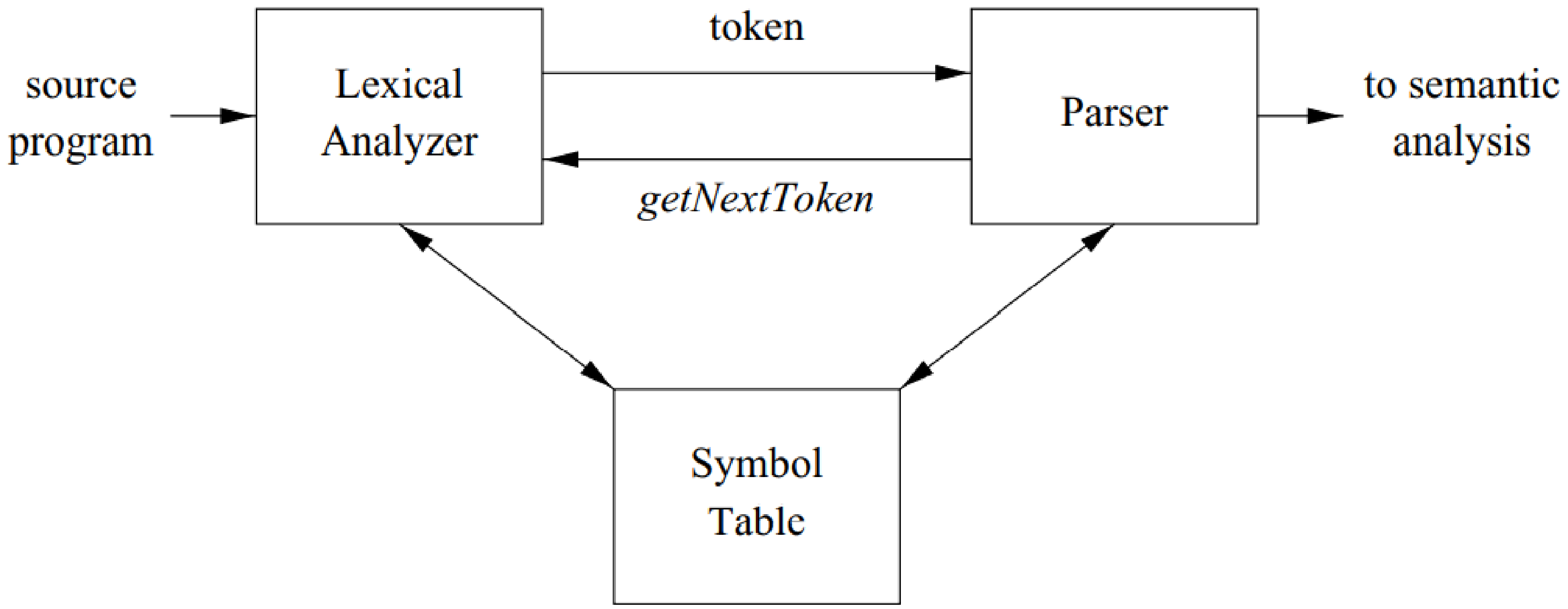
Topics Covered

- The Role of the Lexical Analyzer
  - Why lexical analysis is separated from parsing?
  - Tokens vs. Pattern vs. Lexeme
  - Lexical Errors
- Input Buffering
  - Buffer Pairs & Sentinels
- Specification of Tokens
  - Language Operations
  - Regular Expression and definition
- Recognition of Tokens
  - Transition Diagrams
- Convert finite automata to regular expressions
  - Arden’s Theorem
  - State elimination method

Lexical Analysis

- The main task of the lexical analyzer is
  - to read the input characters of the source program,
  - group them into lexemes and
  - produce as output a sequence of tokens for each lexeme in the source program.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- The stream of tokens is sent to the parser for syntax analysis.

Interactions between the lexical analyzer and the parser



The call, suggested by the ***getNextToken*** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token which it returns to the parser.

Tasks of lexical analyzer

- Scans the input
- Remove comments and whitespaces (also other characters that maybe used to separate tokens)
- Identify the tokens
- Insert the tokens into symbol table
- Keeps track of line numbers and associates error messages from various parts of a compiler with line numbers
- Send the tokens to the parser
- Performs some preprocessor functions such as #define and #include in ’C’

Why lexical analysis is separated from parsing?

1. **Simplicity of design**
   - As the tasks performed by each phase is distinct.
2. **Compiler efficiency is improved**
   - Specialized buffering techniques for reading the input characters speeds up the compiler
3. **Compiler portability is enhanced**
   - Input device specific peculiarities can be restricted to the lexical analyzer.

Tokens vs. Patterns vs. Lexemes

- **Token**
  - A **token** is a pair consisting of a token name and an optional attribute value.
  - It is a classification for a common set of strings.
  - E.g. a particular keyword or a sequence of input characters denoting an identifier, etc
  - The token names are the input symbols that parser processes.

Tokens vs. Patterns vs. Lexemes

- **Pattern**
  - A **pattern** is a description of the form that the lexemes of a token may take.
  - They are the rules which characterize the set of strings for a token.
  - E.g. ([A-Z]*.*)

Tokens vs. Patterns vs. Lexemes

- **Lexeme**
  - A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.
  - It is the actual sequence of characters that matches pattern and is classified by a token
  - E.g. Identifiers: x, count, name, etc…

Examples

Token	Pattern	Lexeme
if	if	if
const	const	const
id	letter followed by letters of digit	pi, name, num1
number	any numeric constant	3.14, 100, 6.1e9
literal	Anything between quotes " except quotes "	"core dumped"

For many programming languages, there are these following classes of tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators
3. One token representing all identifiers
4. One or more tokens representing constants, such as numbers and literal strings
5. Tokens for each punctuation symbol, such as left and right parenthesis, comma, and semicolon.

Example 1: Identify tokens

$E = mc^2$
in FORTRAN

E = M * C ** 2

<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>

Example 2: Identify tokens

```
float limitedSquare(x){float x;
  /* returns x-squared, nut never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

Example 2: Identify tokens

```
float limitedSquare(x){float x;
  /* returns x-squared, nut never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

```
<float> <id, limitedSquare> <(> <id, x> <)> <{> <float> <id, x>
    <return> <(> <id, x> <op,"<="> <num, -10.0>
     <op, "||"> <id, x> <op, ">="> <num, 10.0> <)>
    <op, "?"> <num, 100> <op, ":"> <id, x> <op, "*"> <id, x>
 <}>
```

Example 2: Identify tokens

```
float limitedSquare(x){float x;
  /* returns x-squared, nut never more than 100 */
  return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

```
<float> <id, limitedSquare> <(> <id, x> <)> <{> <float> <id, x>
     <return> <(> <id, x> <op,"<="> <num, -10.0>
      <op, "||"> <id, x> <op, ">="> <num, 10.0> <)>
     <op, "?"> <num, 100> <op, ":"> <id, x> <op, "*"> <id, x>
 <}>
```

Example 3: Count the no. of tokens

```
int main() {
    // 2 variables
    int a, b;
    a = 10;
    return 0;
}
```

Example 3: Count the no. of tokens

```
int main() {
    // 2 variables
    int a, b;
    a = 10;
    return 0;
}
```

```
<int> <main> <(> <)> <{>


<int> <id, "a"> <,> <id, "b"> <;>
<id, "a"> <op, "="> <num, "10"> <;>
<return> <num, "0"> <;>
<}>
```

Example 3: Count the no. of tokens

```
int main() {
    // 2 variables
    int a, b;
    a = 10;
    return 0;
}
```

18

1. <int>
2. <main>
3. <(>
4. <)>
5. <{>
6. <int>
7. <id, "a">
8. <,>
9. <id, "b">
10. <;>
11. <id, "a">
12. <op, "=">
13. <num, "10">
14. <;>
15. <return>
16. <num, "0">
17. <;>
18. <}>

Lexical Errors

- Error handling is very localized, with respect to input source.
- **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 .
- **Possible error recovery actions**:
  - Deleting or inserting input characters
  - Replacing or transposing characters

Tricky situations...

- Certain languages do not have any reserved words.
  - e.g., while, do, if, else, etc., are reserved in 'C', but not in PL/1.
- Blanks are not significant in FORTRAN and can appear in the midst of identifiers, but not so in 'C'.
- In FORTRAN, some keywords are context-dependent.
  - In the statement, DO 10 I = 10.86,

    DO10I is an identifier, and DO is not a keyword
  - But in the statement, DO 10 I = 10, 86,

    DO is a keyword
  - Such features require substantial look ahead for resolution.
- LA cannot catch any significant errors except for simple errors such as, illegal symbols, etc.
- In such cases, LA skips characters in the input until a well-formed token is found.

Need of Buffering

- For instance, we cannot be sure we have seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id.

- In C/C++, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=.

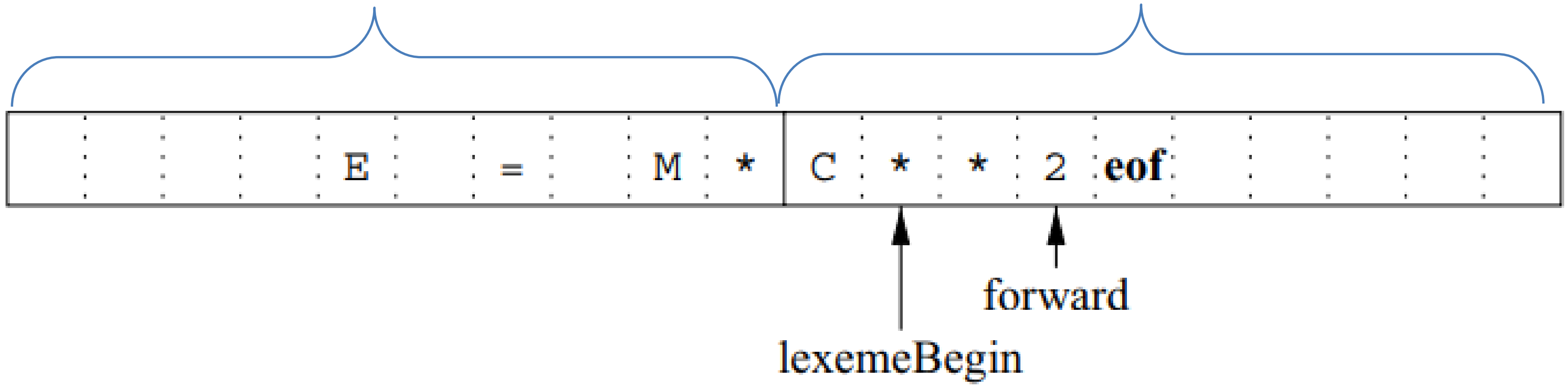
- Lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.

- But, a large amount of time can be consumed moving characters.

Specialized Buffering Technique

- The technique involves two buffers that are alternately reloaded.
- Each buffer is of the same size N.
- N is usually the size of a disk block, e.g., 4096 bytes.
- Using one system read command we can read N characters into a buffer, rather than using one system call per character.
- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file and is different from any possible character of the source program.

Using a pair of input buffers



- Two pointers to the input are maintained:
  1. Pointer **lexemeBegin** points the beginning of the current lexeme being discovered.
  2. Pointer **forward** scans ahead until a pattern match is found for the lexeme.
- The string of characters between the pointers in the current lexeme.

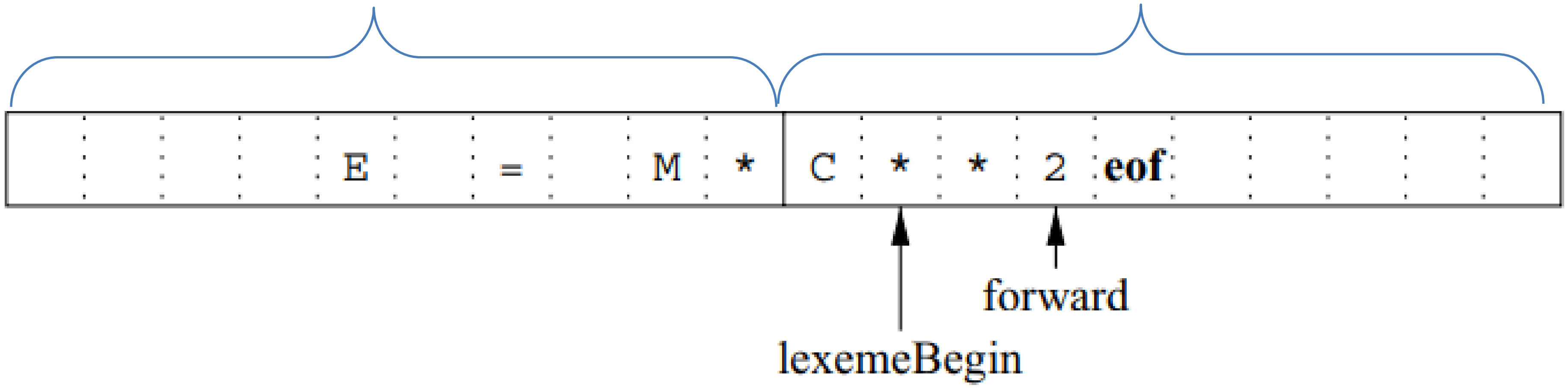
Code to advance forward pointer:

```
if forward at 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 beginning of first half
end
else
    forward := forward + 1;
```

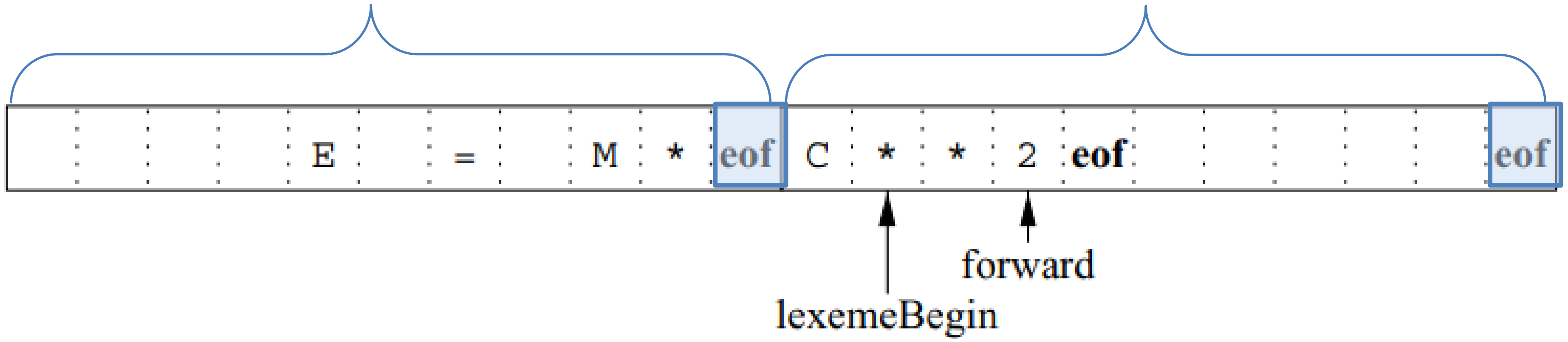
Need of Sentinel

- Each time we advance forward, we must check that we have not moved one of the buffers; if we do, then we must also reload the other buffer.
- Thus, for each character read, we make **two tests:**
  - one for the end of the buffer, and
  - one to determine what character is read
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The **sentinel** is a special character that cannot be part of the source program, and a natural choice is the character **eof.**

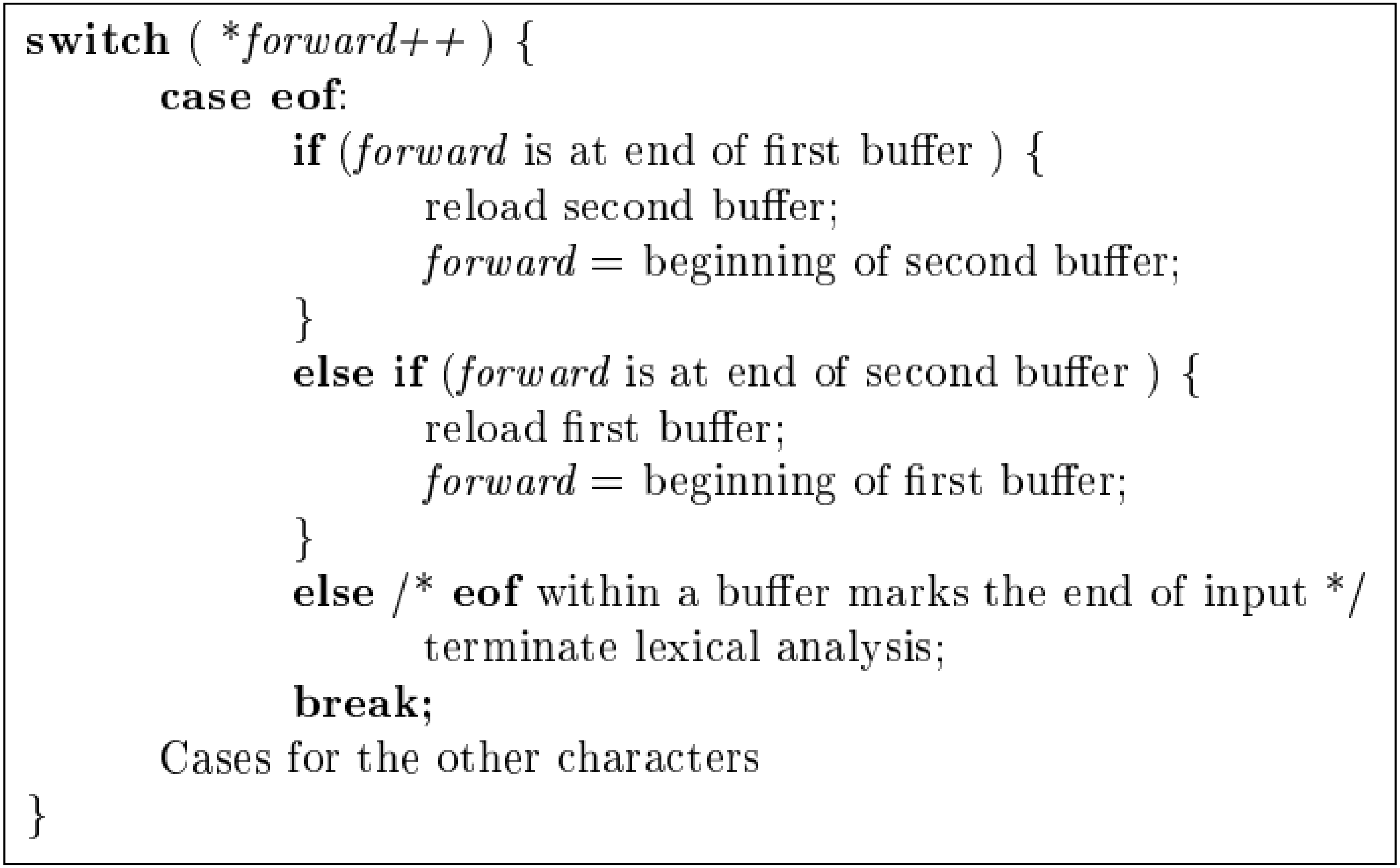
Using a pair of input buffers



Sentinels at the end of each buffer



Code to advance forward pointer:



```
switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}
```

Specification of Tokens

- An alphabet is any finite set of symbols.
  - The set {0,1} is the binary alphabet.

- 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."
  - The length of a string s, usually written as|s|, is the number of occurrences of symbols in s.
  - The empty string, denoted ε (epsilon), is the string of length zero.

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

Language Operations

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	$L^+ = \cup_{i=1}^{\infty} L^i$

- The (Kleene) closure of language L, denoted L* , is the set of strings you get by concatenating L zero or more times.
- Note that L0 , the “concatenation of L zero times,” is defined to be {ε}, and inductively, L^i is $L^{i-1}L$.
- Finally, the positive closure, denoted L^+, is the same as the Kleene closure, but without the term L^0 . That is, ε will not be in L^+ unless it is in L itself.

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 U D** is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
- **LD** is the set of 520 strings of length two, each consisting of one letter followed by one digit.
- **L^4** is the set of all 4-letter strings.
- **L*** is the set of all strings of letters, including ε, the empty string.
- **L(L U D)*** is the set of all strings of letters and digits beginning with a letter.
- **D^+** is the set of all strings of one or more digits.

Regular Language

- A regular expression is a set of rules / techniques for constructing sequences of symbols (strings) from an alphabet.

- Let ∑ be an alphabet, r a regular expression then L(r) is the language that is characterized by the rules of r .

Algebraic laws for regular expressions

LAW	DESCRIPTION
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt;\ (s \mid t)r = sr \mid tr$	Concatenation distributes over $\mid$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r \mid \epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Some Examples

- All strings that start with "tab" or end with "bat" .
- All strings in which digits 1,2,3 exist in ascending numerical order.
- All strings of lowercase letters in which the letters are in ascending lexicographic order.

Some Examples

- All strings that start with "tab" or end with "bat" .

  tab{A,…,Z,a,...,z}*|{A,…,Z,a,....,z}*bat

- All strings in which digits 1,2,3 exist in ascending numerical order.

  {A,…,Z}*1 {A,…,Z}*2 {A,…,Z}*3 {A,…,Z}*

- All strings of lowercase letters in which the letters are in ascending lexicographic order.

  a*b*c*………z*

Regular Definition

- Regular definition for C language identifiers

  letter_ → A |B| …| Z | a | b |…| z | _

  digit → 0 |1|…|9

  id → letter_(letter_ | digit) *

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & [\texttt{A-Za-z\_}] \\
\mathit{digit} & \rightarrow & [\texttt{0-9}] \\
\mathit{id} & \rightarrow & \mathit{letter\_}\ (\ \mathit{letter\_} \mid \mathit{digit}\ )^*
\end{array}
$$

Regular Definition

- Regular definition for unsigned numbers (integer or floating point)

$$\begin{aligned}
digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits &\rightarrow digit\ digit^* \\
optionalFraction &\rightarrow \texttt{.}\ digits \mid \epsilon \\
optionalExponent &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ digits\ ) \mid \epsilon \\
number &\rightarrow digits\ optionalFraction\ optionalExponent
\end{aligned}$$

- E.g. 5280, 0.01234, 6.336E4, or 1.89E-4

$$\begin{aligned}
digit &\rightarrow [\texttt{0}-\texttt{9}] \\
digits &\rightarrow digit^{+} \\
number &\rightarrow digits\ (\texttt{.}\ \ digits)?\ \ (\ \texttt{E}\ [\texttt{+-}]?\ \ digits\ )?
\end{aligned}$$

Describe the languages denoted by the following regular expressions

- a(a|b)*a
- ((ε|a)b*)*
- (a|b)*a(a|b)(a|b)
- a*ba*ba*ba*
- (aa|bb) *((ab|ba)(aa|bb)* (ab|ba)(aa|bb)*)*

Describe the languages denoted by the following regular expressions

- a(a|b)*a
  - String of a’s and b’s that start and end with a
- ((ε|a)b*)*
  - String of a’s and b’s
- (a|b)*a(a|b)(a|b)
  - String of a’s and b’s that has the third character from the last as a.
- a*ba*ba*ba*
  - String of a’s and b’s that contains only three b’s
- (aa|bb) *((ab|ba)(aa|bb)* (ab|ba)(aa|bb)*)*
  - String of a’s and b’s that has even number of a’s and b’s

Write character classes for the following sets of characters:

- The first ten letters (up to "j") in either upper or lower case.
- The lowercase consonants.
- The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).
- The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).

Write character classes for the following sets of characters:

- The first ten letters (up to "j") in either upper or lower case.
  - [A-Ja-j]
- The lowercase consonants.
  - [bcdfghjklmnpqrstvwxzyz]
- The "digits" in a hexadecimal number (choose either upper or lower case for the "digits" above 9).
  - [0-9a-f]
- The characters that can appear at the end of a legitimate English sentence (e.g., exclamation point).
  - [.?!]

Transition Diagrams

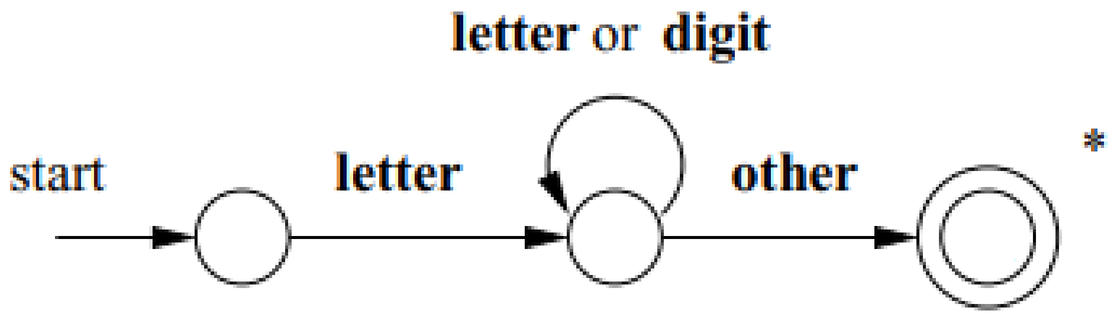
- As an intermediate step in the construction of a lexical analyzer, we first convert patterns into stylized flowcharts, called "transition diagrams.“
- Transition diagrams have a collection of **nodes** or circles, called **states**.
- Edges are directed from one state of the transition diagram to another.
- Each **edge** is labeled by **a symbol or set of symbols**.
- We shall assume that all our transition diagrams are deterministic (DFA’s), meaning that there is never more than one edge out of a given state with a given symbol among its labels.

Transition Diagrams

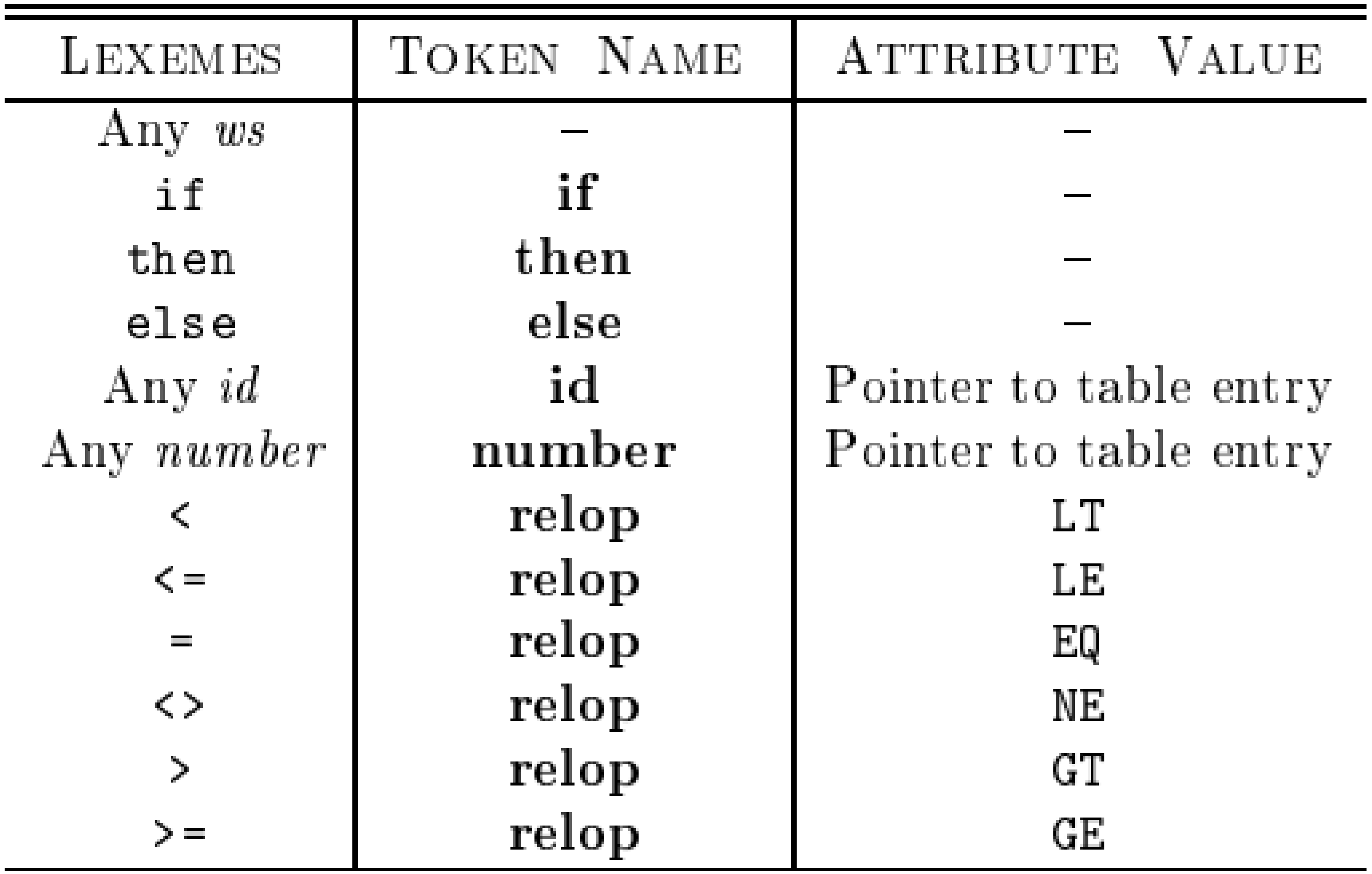
- Start state or initial state is labelled by start
- Accepting state or final state means that the lexeme is found. It is represented by double circle.
- If it is necessary to retract forward pointer one position, then we shall additionally place * near the accepting state.

A transition diagram for id

$$\begin{array}{rcl} \mathit{letter\_} & \rightarrow & \texttt{[A-Za-z\_]} \\ \mathit{digit} & \rightarrow & \texttt{[0-9]} \\ \mathit{id} & \rightarrow & \mathit{letter\_}\ (\ \mathit{letter\_} \mid \mathit{digit}\ )^{*} \end{array}$$

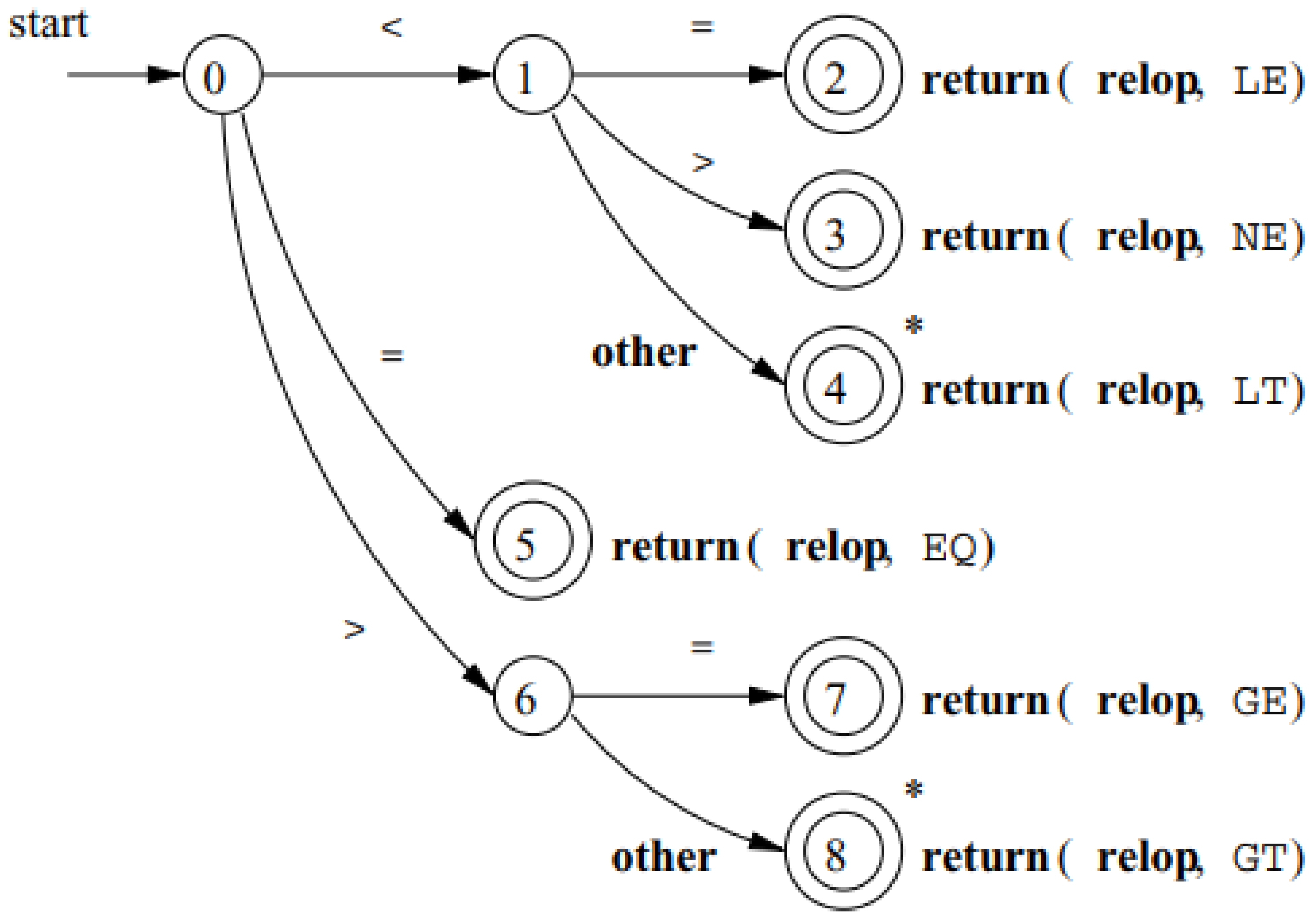


Tokens, their patterns, and attribute values

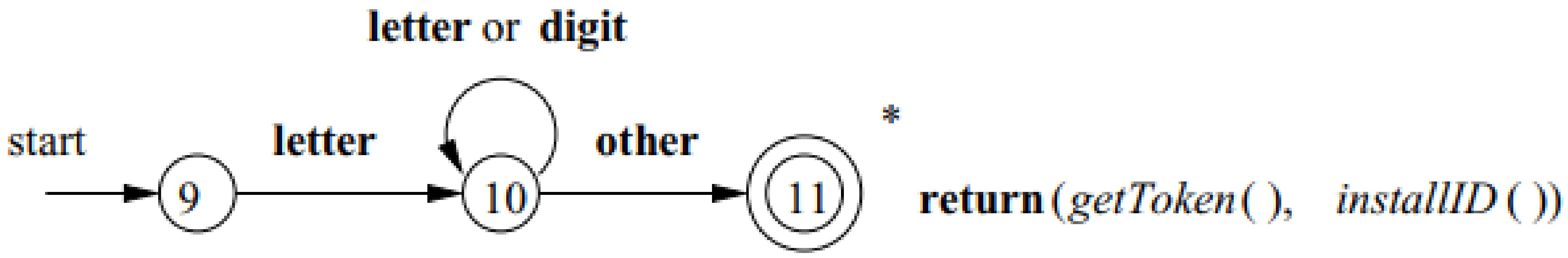


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Transition diagram that recognizes the lexemes matching the token **relop**



(cont.)Transition diagram for id’s and keyword



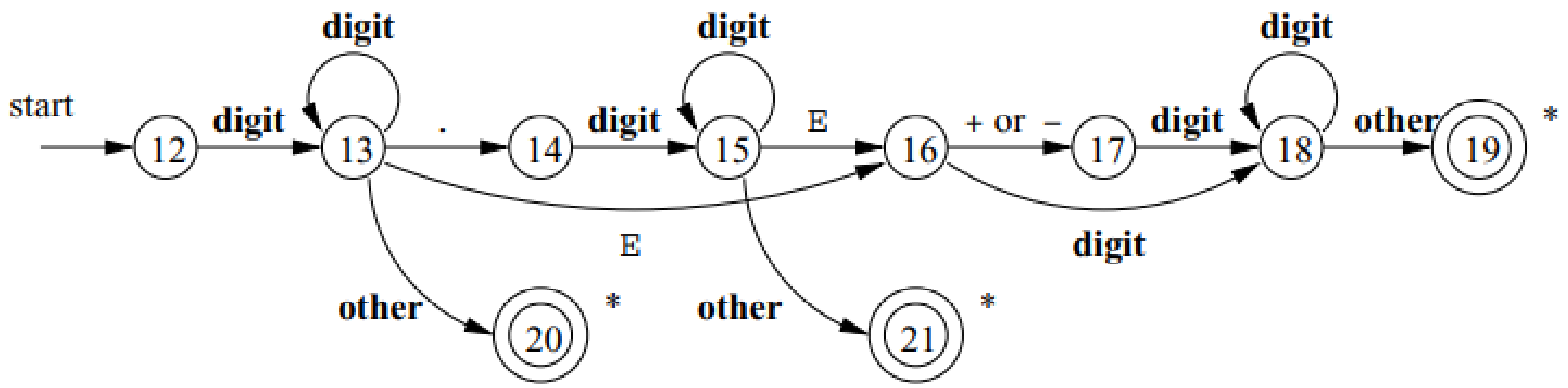
Handling the reserved words

- When we find an identifier, a call to **installID()** places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found.

- Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id**.

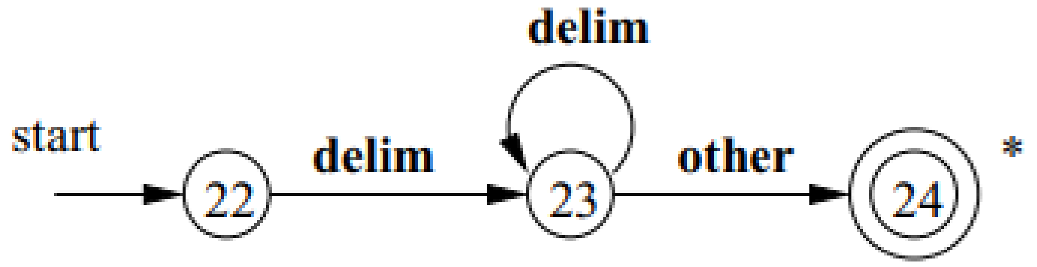
- The function **getToken()** examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents - either id or one of the keyword tokens that was initially installed in the table.

(cont.) transition diagram for unsigned numbers



$$\begin{array}{rcl} digit & \rightarrow & [0\text{-}9] \\ digits & \rightarrow & digit^{+} \\ number & \rightarrow & digits\ (.\ digits)?\ (\ \mathtt{E}\ [+-]?\ digits\ )? \end{array}$$

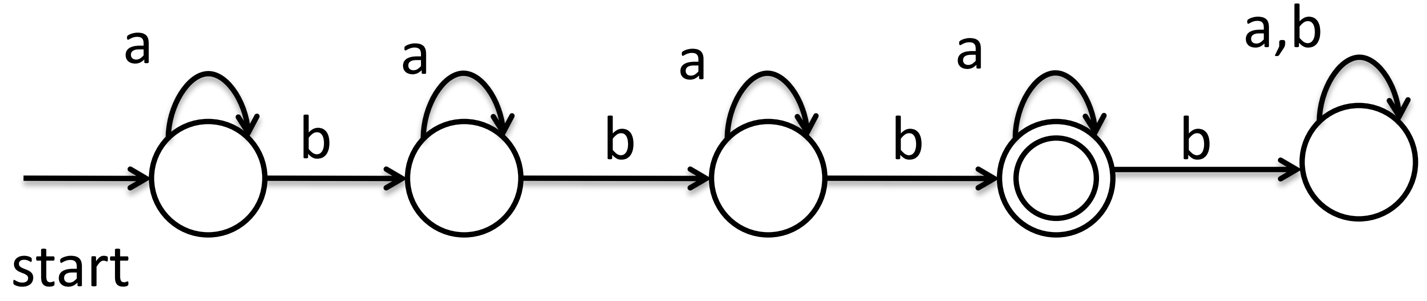
(cont.) transition diagram for whitespaces



$$ws \rightarrow (\ \mathbf{blank}\ |\ \mathbf{tab}\ |\ \mathbf{newline}\ )^{+}$$

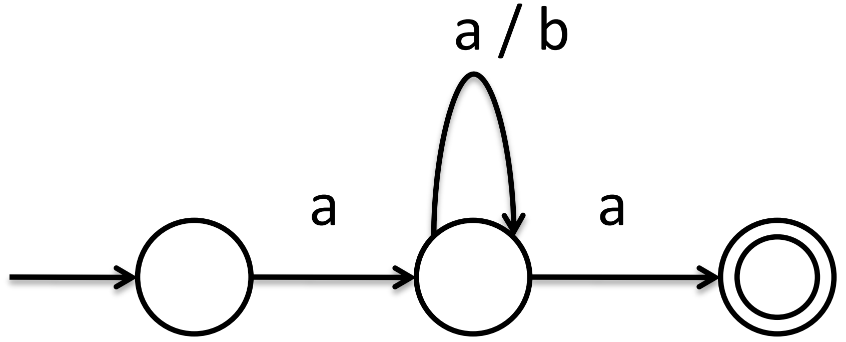
- Note that in state 24, we have found a block of consecutive whitespace characters, followed by a nonwhite space character.
- We retract the input to begin at the nonwhite space, but we do not return to the parser.
- Rather, we must restart the process of lexical analysis after the whitespace.

Transition Diagram for a*ba*ba*ba*

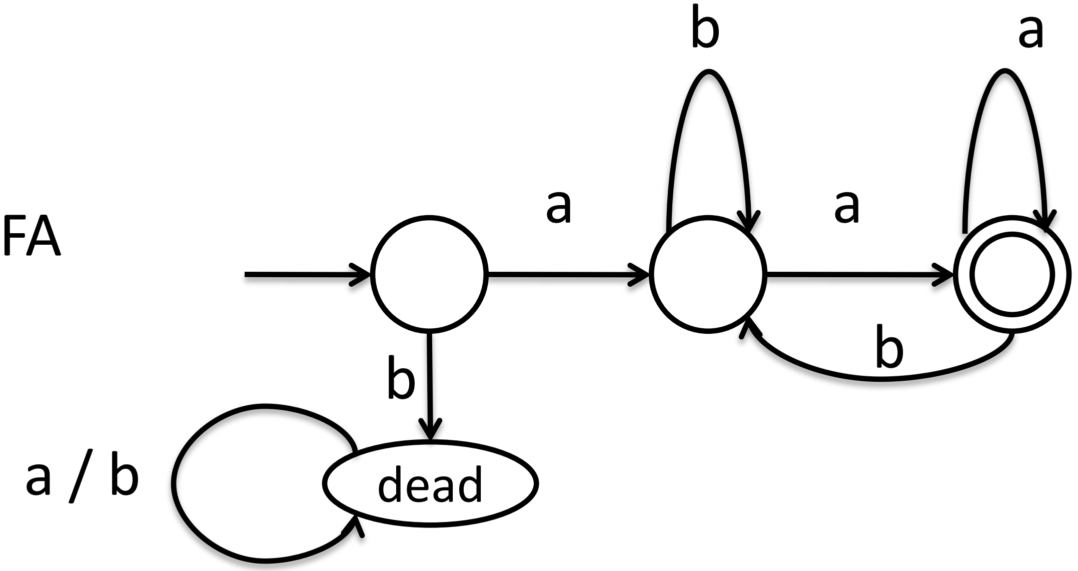


Transition Diagram for a(a|b)*a

NFA



DFA



Try the following:

- (a|b)*a(a|b)(a|b)

- (aa|bb)* ((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

Convert Finite Automata to Regular Expression

1. Arden’s Theorem
2. State Elimination Method

- Assumptions to use **Arden’s Theorem**:
    1. The transition diagram must not have any ε transitions.
    2. There must be only a single initial state in the finite automata.

Arden’s Theorem

- **Statement:**

  Let **P** and **Q** be two regular expressions.
  If **P** does not contain null string, then **R = Q + RP** has a unique solution that is **R = QP***

- **Proof :**

  $R = Q + (Q + RP)P$ [After putting the value $R = Q + RP$]

  $= Q + QP + RPP$

  When we put the value of **R** recursively again and again, we get the following equation :−

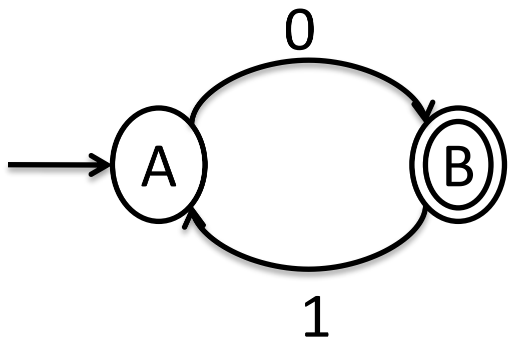
  $R = Q + QP + \mathbf{QP^2 + QP^3}$**…..**

  $R = Q (\varepsilon + P + P^2 + P^3 + ….)$

  $R = QP^*$ [As P^* represents $(\varepsilon + P + P^2 + P^3 + ….)$]

  Hence, proved.

1. Construct a Regular Expression from the given Finite Automata by Algebraic Method using Arden's Theorem



For State A:

A= ε + B1

For State B:

B = A0

Bring accepting state in the form R=Q+RP,

B= (ε + B1)0

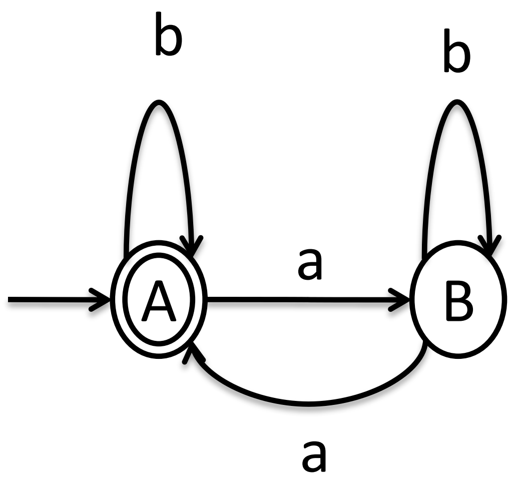
B= ε0 + B10

B= 0+B10

Here, R=B, Q=0 and P=10

As per R=QP*, solution will be : **0(10)***

2. Construct a Regular Expression from the given Finite Automata by Algebraic Method using Arden's Theorem



$$\begin{tikzcd} \arrow[r] & A \arrow[loop above, "b"] \arrow[r, "a"] & B \arrow[loop above, "b"] \arrow[l, bend left, "a"] \end{tikzcd}$$

For State A: A= ε + Ab + Ba

For State B: B = Aa + Bb

In state B, applying Arden's Theorem (R=Q+RP),

R=B, P=b and Q=Aa

As per R=QP*, B=Aab*

Bring accepting state in the form R=Q+RP,

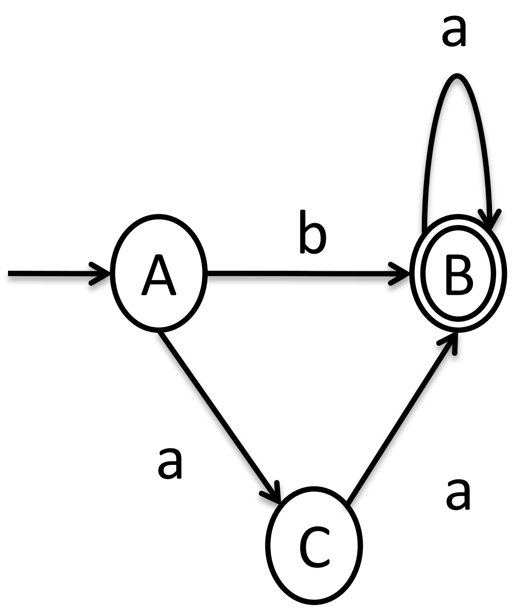
A= ε + Ab + Aab*a

A = ε + A(b + ab*a)

Here, R=A, Q = ε and P=b+ab*a

As per R=QP*, solution will be **: A=(b+ab*a)***

3. Construct a Regular Expression from the given Finite Automata by Algebraic Method using Arden's Theorem



For State A: A = ε

For State B: B = Ab + Ba + Ca

For State C: C = Aa

Bring accepting state in the form R=Q+RP,

B = Ab + Ba + Aaa [C = Aa]

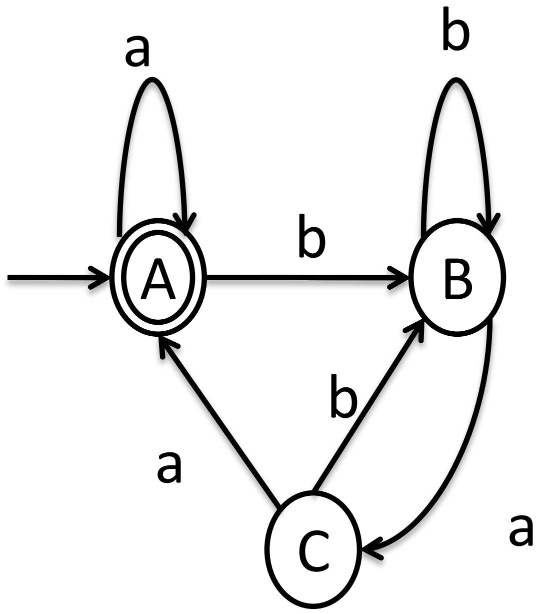
B = εb + Ba + εaa

B = b + aa + Ba

Here, R = B, Q = b + aa, P = a

As per R=QP*, solution will be **(b + aa)a***

4. Construct a Regular Expression from the given Finite Automata by Algebraic Method using Arden's Theorem



For State A: A = ε + Aa + Ca

For State B: B = Ab + Bb + Cb

For State C: C = Ba

B = Ab + Bb + Cb

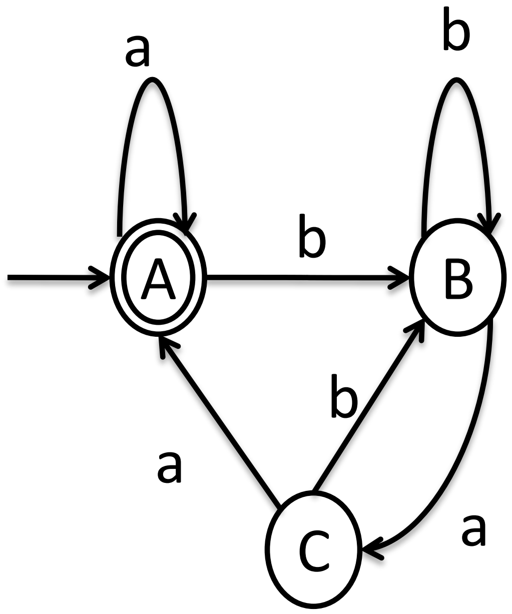
B = Ab + Bb + Bab

Here, R = B, P = (b + ab), Q = Ab

So, by Arden's Theorem (R=QP*),

B = Ab(b+ab)*

4. Construct a Regular Expression from the given Finite Automata by Algebraic Method using Arden's Theorem (cont.)



Bring accepting state in the form R=Q+RP,

A = ε + Aa + Ca

A = ε + Aa + Baa [C= Ba]

A = ε + Aa + Ab(b+ab)*aa [B= Ab(b+ab)*]

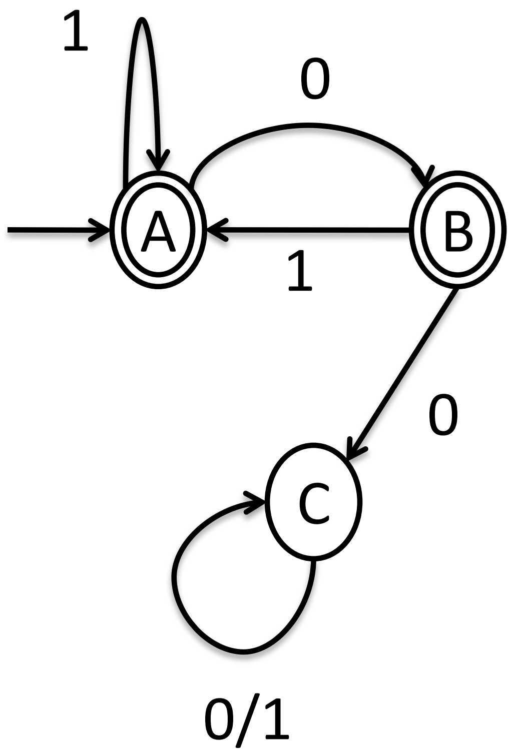
A = ε + A(a + b(b+ab)*aa)

Here, R = A, Q = ε and P = a + b(b+ab)*aa

As per R=QP*, solution will be

A = (a + b(b+ab)*aa)*

5. Construct a Regular Expression from the given Finite Automata by Algebraic Method using Arden's Theorem



For State A: A = ε + A1 + B1

For State B: B = A0

Bring accepting state A in the form R=Q+RP,

A = ε + A1 + B1

A = ε + A1 + A01 = ε + (1 + 01)A

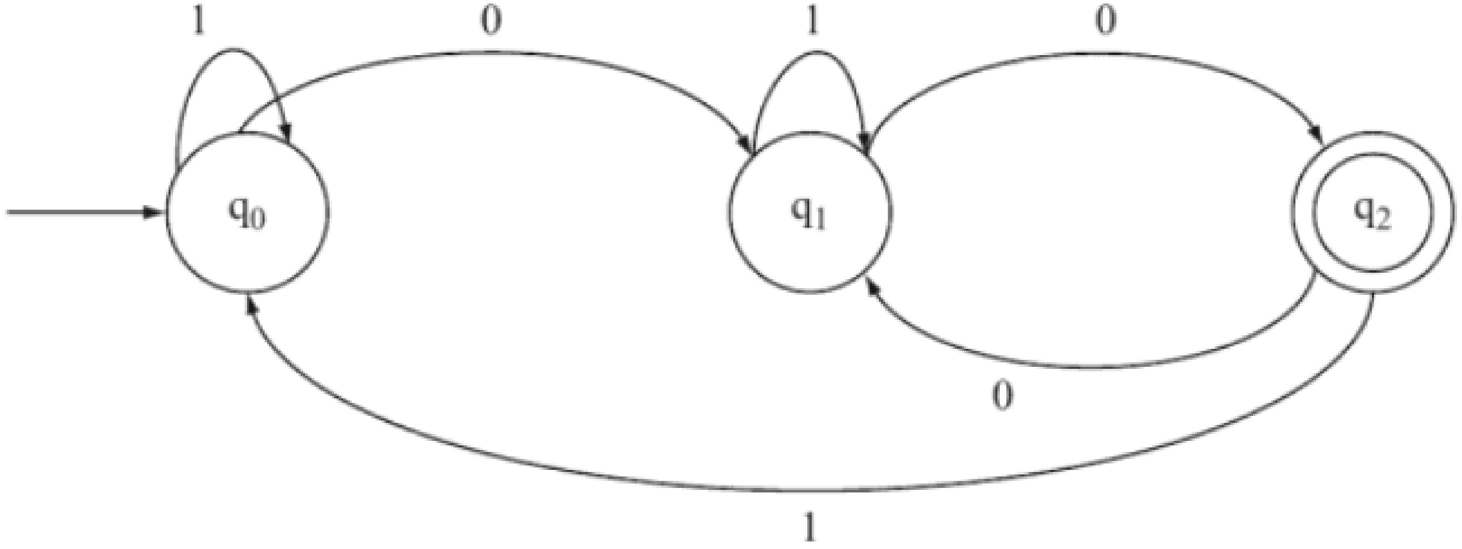
Here, R = A, Q = ε and P = (1 + 01)

As per R=QP*, solution will be **A = (1 + 01)***

And so, **B = (1 + 01)*0**

Hence, solution is **(1 + 01)* + (1 + 01)*0**

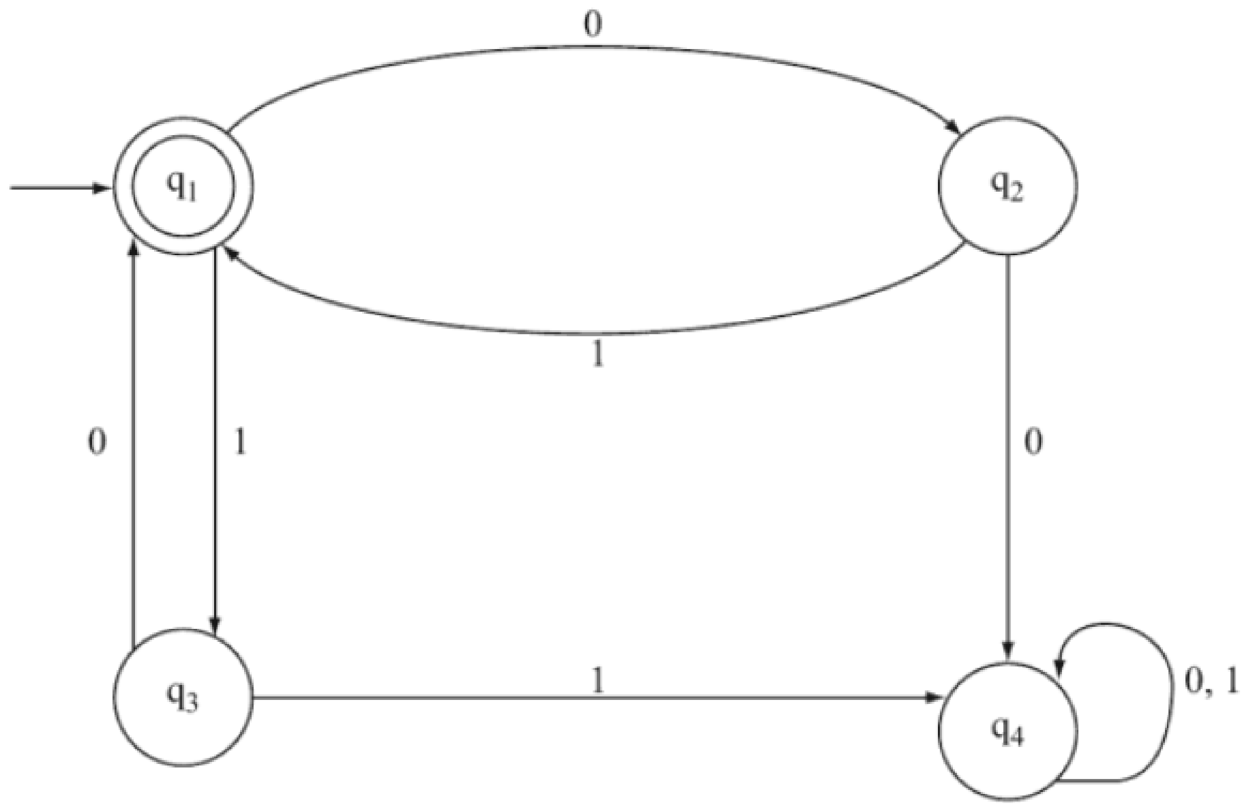
Examples to try



Solution: q2 = [1 + 0.(1+0.0)*.0.1]*.0.(1+0.0)*.0

Example from : **Automata Theory and Formal Languages:** By Shyamalendu Kandar

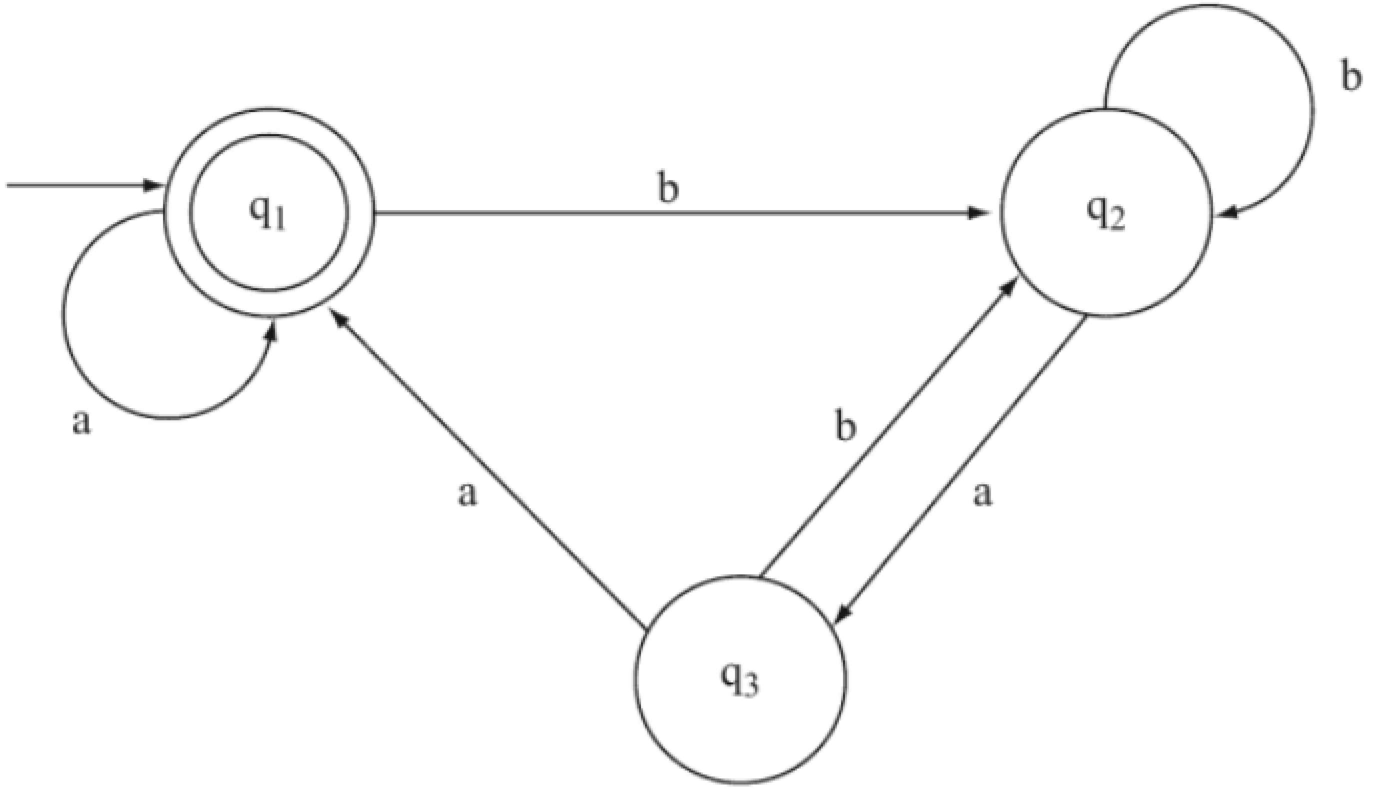
Example to try



Solution: q1 =(01 + 10)*

Example from : **Automata Theory and Formal Languages:** By Shyamalendu Kandar

Example to try



Solution: q1 = [a + b.(b + ab)*.a.a]*

Example from : **Automata Theory and Formal Languages:** By Shyamalendu Kandar

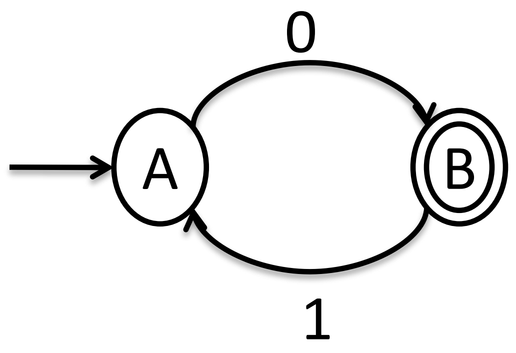
State Elimination Method

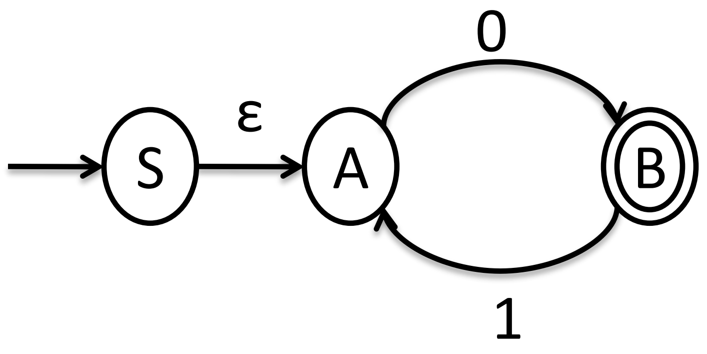
- Rules:
  - The initial state of the DFA must not have any incoming edge.
  - There must exist only one final state in the DFA.
  - The final state of the DFA must not have any outgoing edge.

- The state elimination method can be applied to any finite automata. (NFA, ∈-NFA, DFA etc)

Step 1 of State Elimination Method

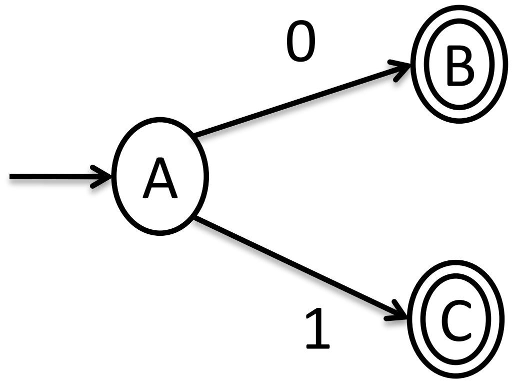
- RULE : The initial state of the DFA must not have any incoming edge.
- If there exists any incoming edge to the initial state, then create a new initial state having no incoming edge to it.

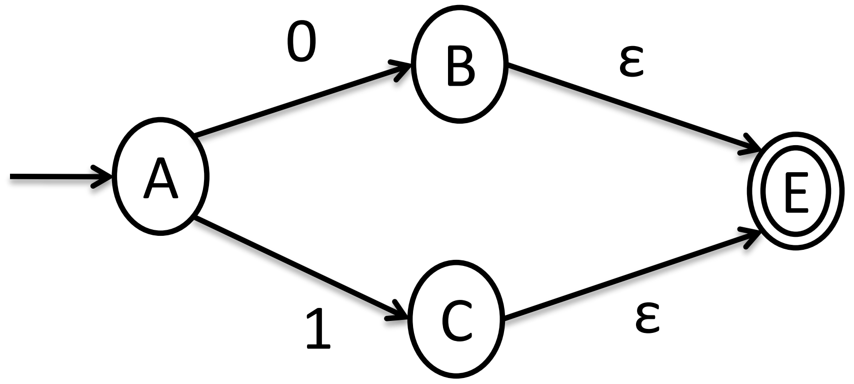




Step 2 of State Elimination Method

- RULE : There must exist only one final state in the DFA.
- If there exists multiple final states in the DFA, then convert all the final states into non-final states and create a new single final state.

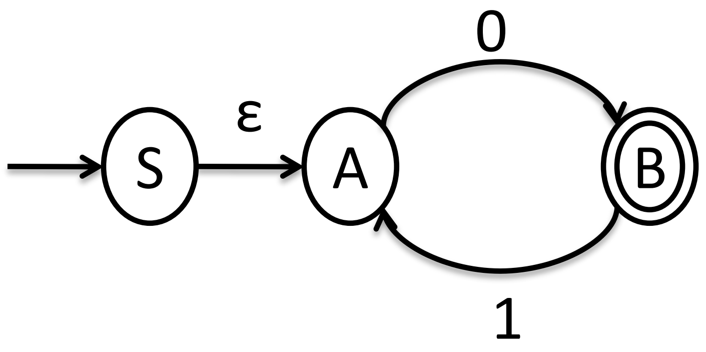


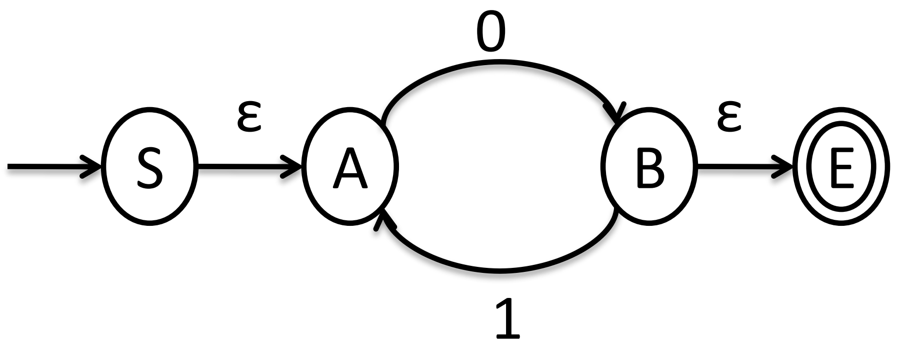


Step 3 of State Elimination Method

- RULE : The final state of the DFA must not have any outgoing edge.

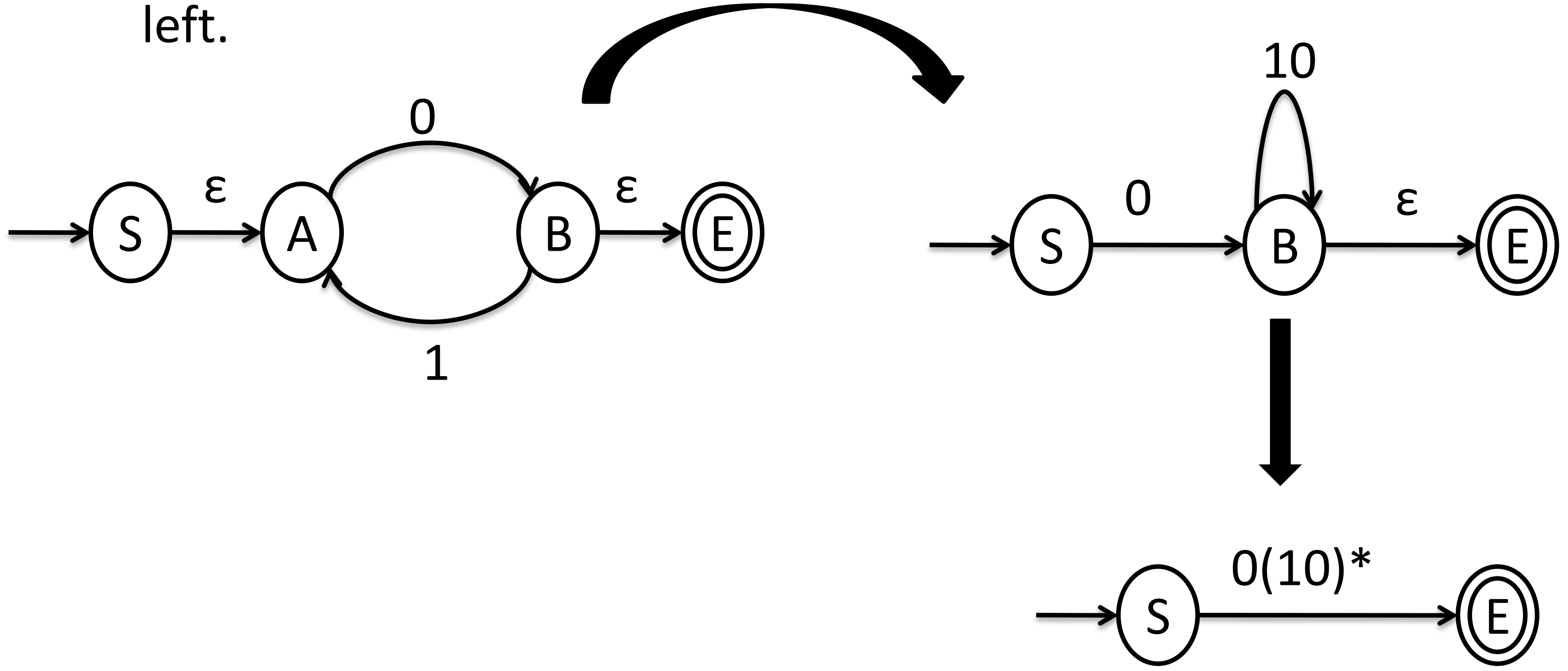
- If there exists any outgoing edge from the final state, then create a new final state having no outgoing edge from it.



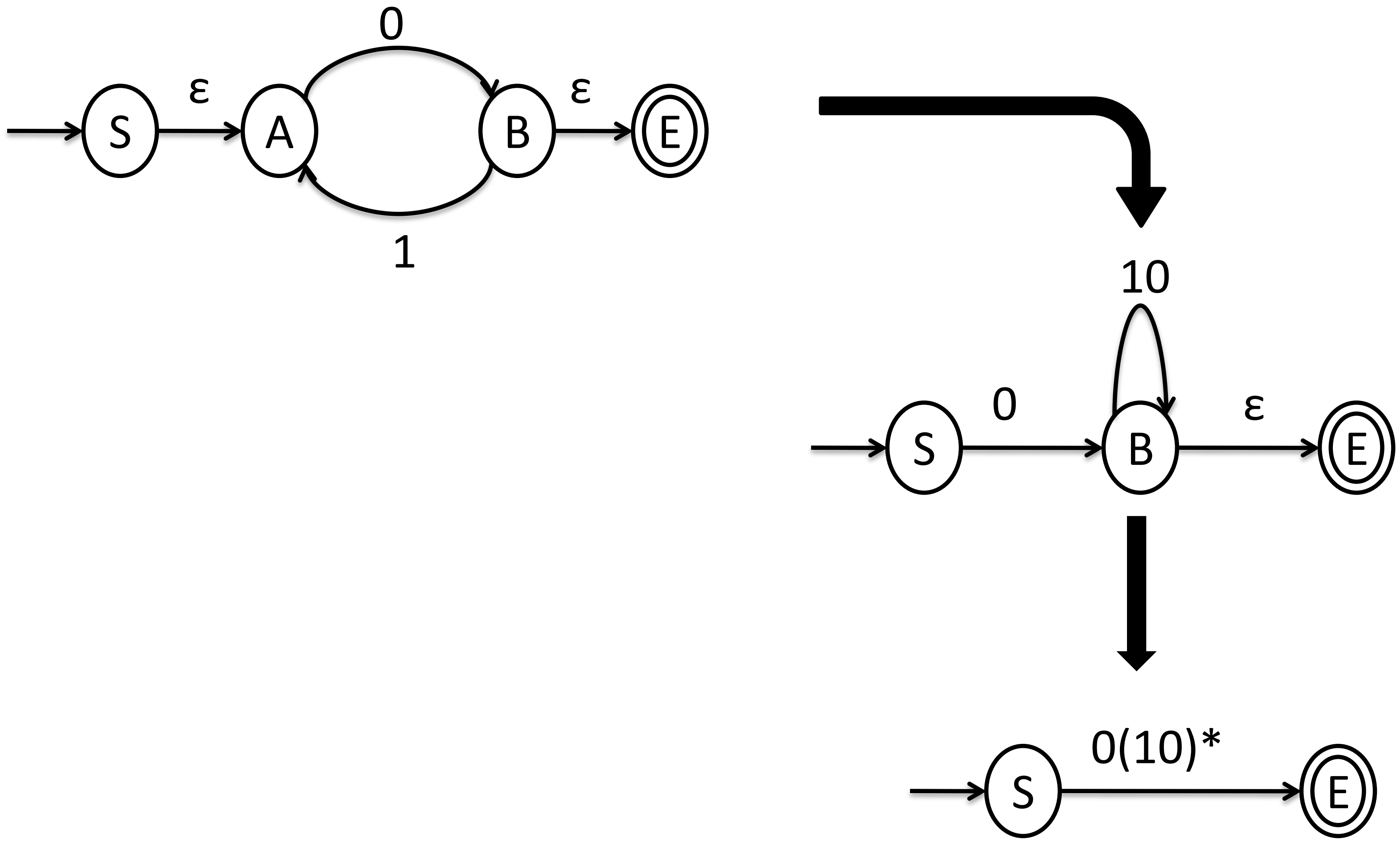


Step 4 of State Elimination Method

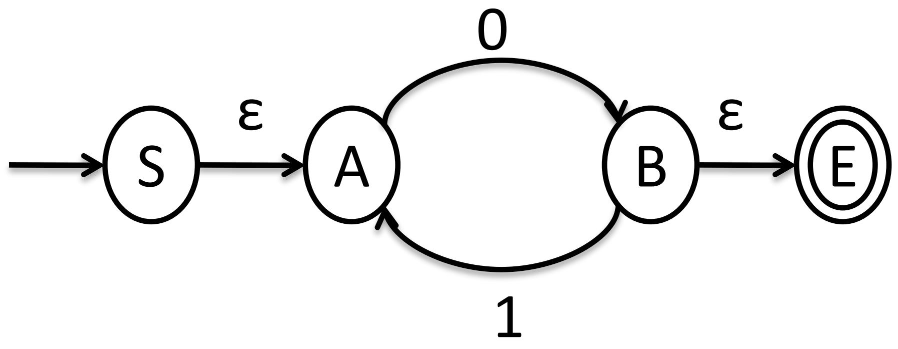
- Eliminate all the intermediate states one by one.
- These states may be eliminated in any order.
- In the end, only an initial state going to the final state will be left.



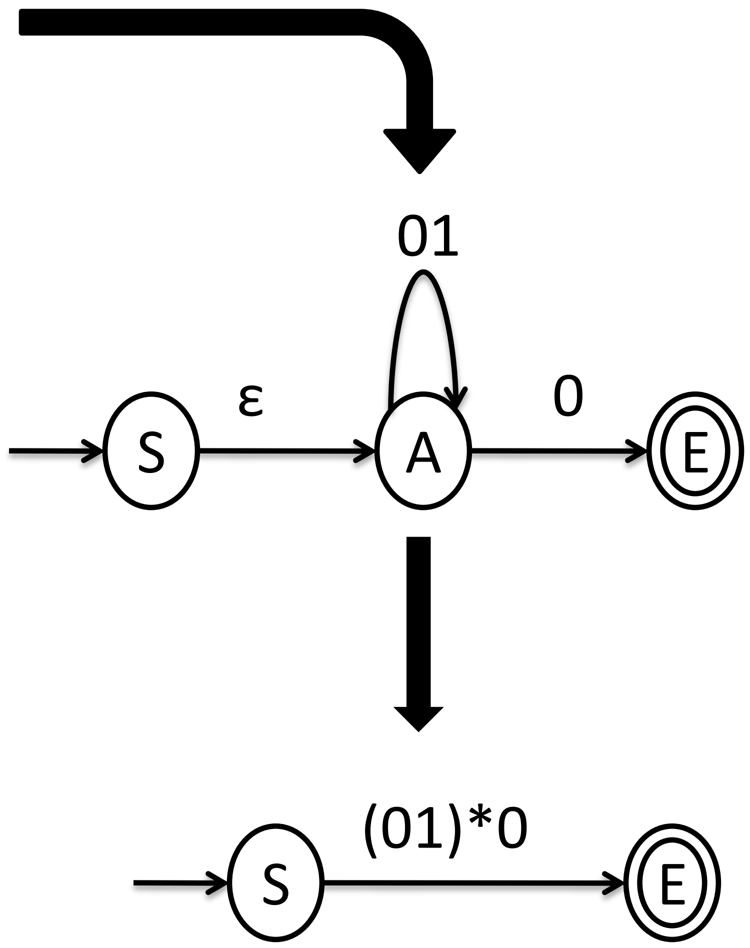
Step 4 of State Elimination Method



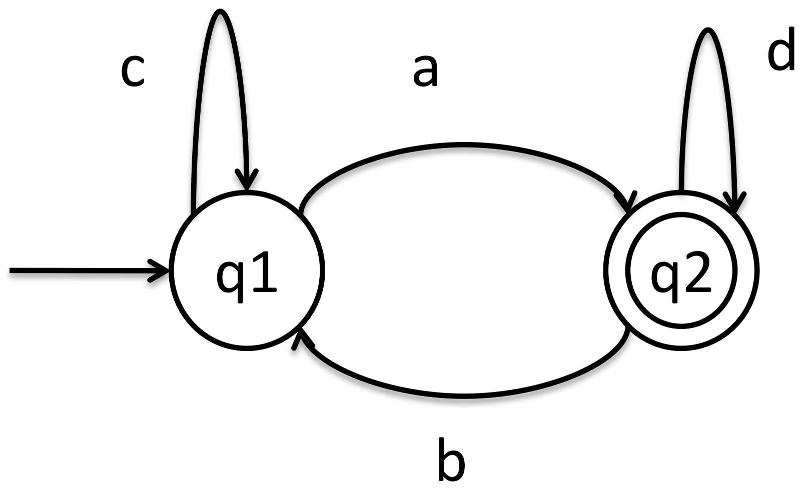
Step 4 of State Elimination Method



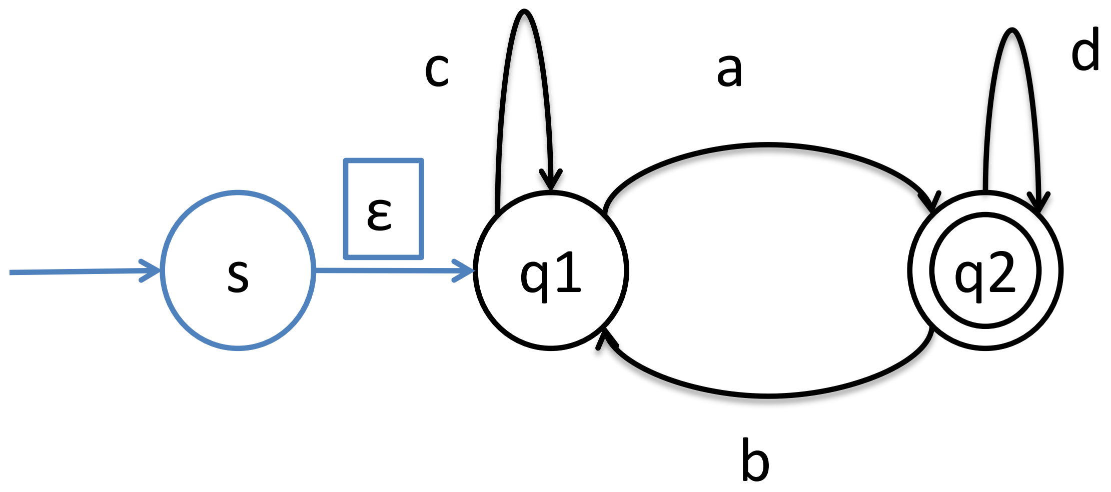
OR



Example 2 : State Elimination Method

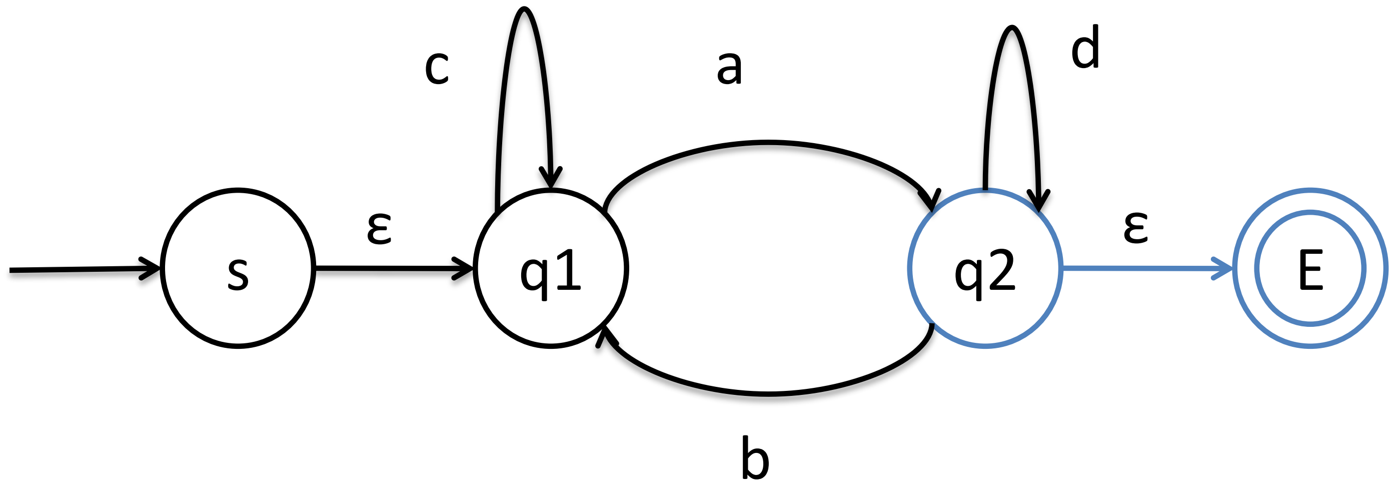


Example 2 : State Elimination Method



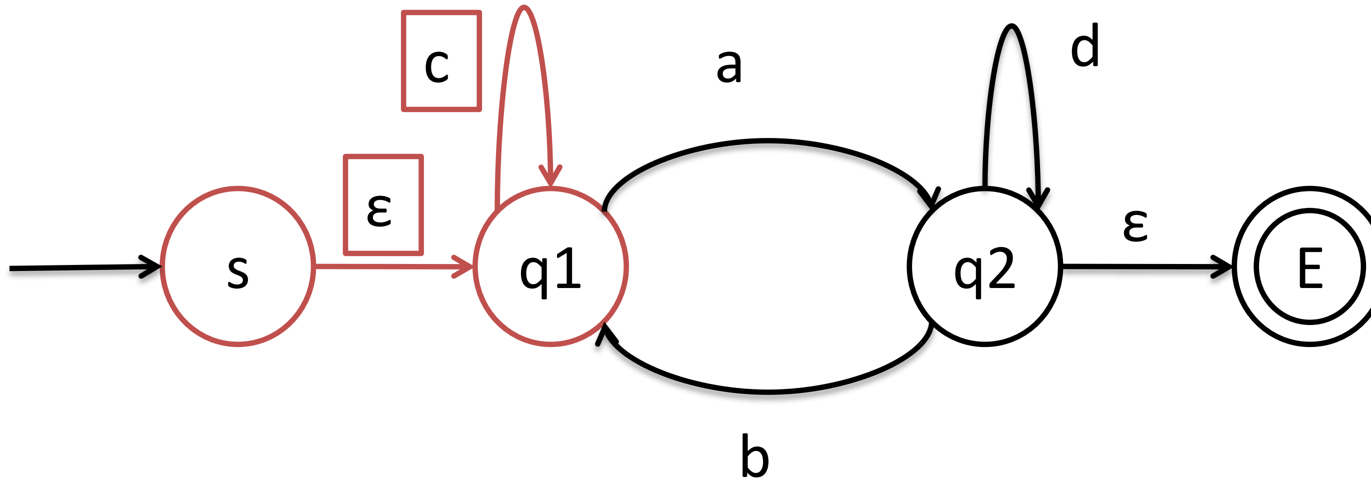
If there exists any incoming edge to the initial state, then create a new initial state having no incoming edge to it.

Example 2 : State Elimination Method



If there exists any outgoing edge from the final state, then create a new final state having no outgoing edge from it.

Example 2 : State Elimination Method



Eliminating state q1:

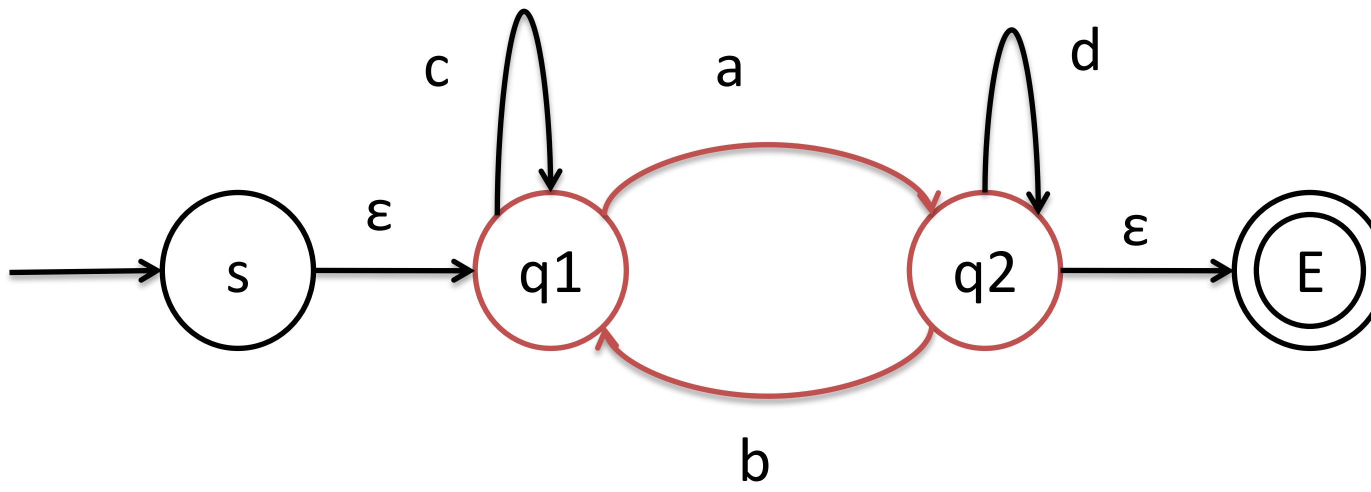
There is a path from state S to state q2 via state q1.

So, when state q1 is eliminated, the **direct path** will have cost **ε.c*.a = c*a**

Also, there is a loop on state q2 using state q1.

So, after eliminating state q1, there will be a direct loop on state q2 with cost bc*.a = bc*a

Example 2 : State Elimination Method



Eliminating state q1:

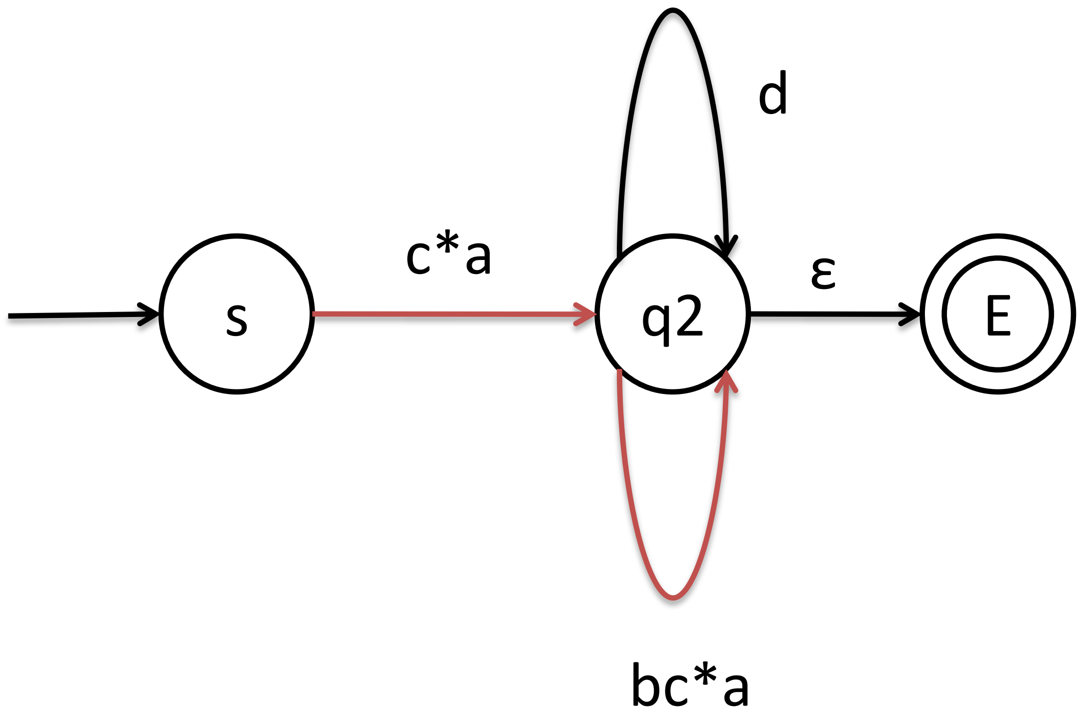
There is a path from state S to state q2 via state q1.

So, when state q1 is eliminated, the **direct path** will have cost **ε.c*.a = c*a**

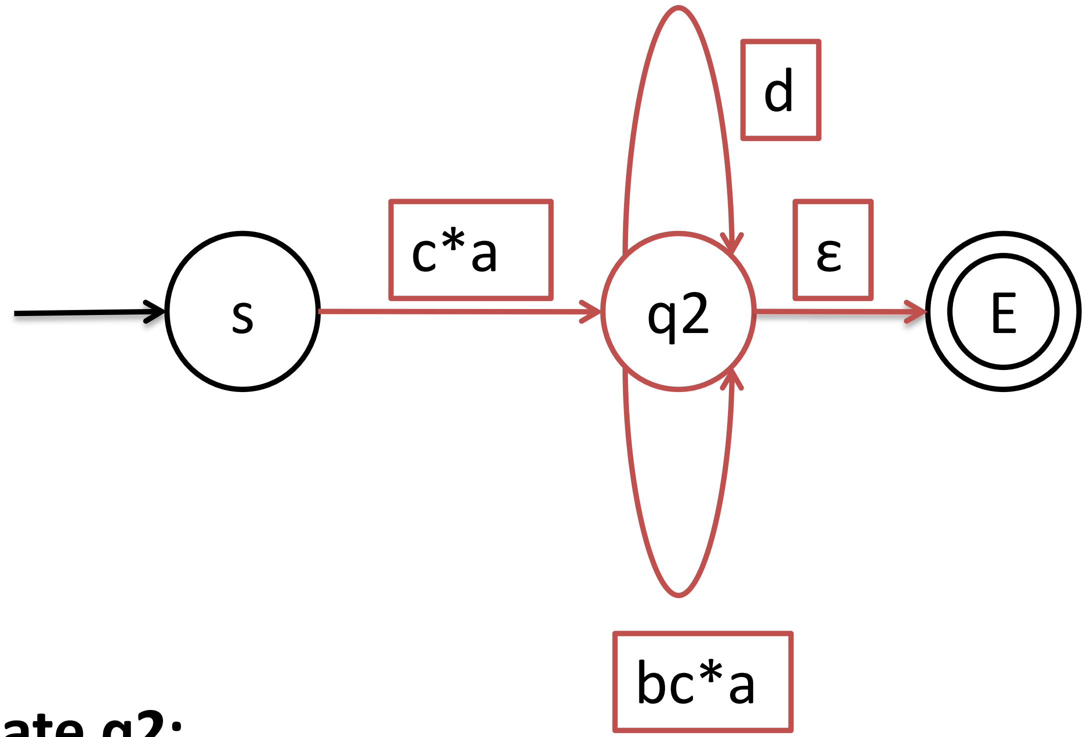
Also, there is a loop on state q2 using state q1.

So, after eliminating state q1, there will be a **direct loop** on state q2 with cost **bc*.a = bc*a**

Example 2 : State Elimination Method



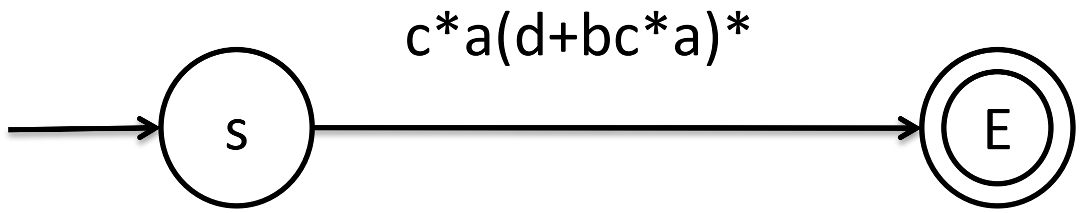
Example 2 : State Elimination Method



Eliminating state q2:

As there is a path going from S to E via state q2, so after eliminating q2, there will be a direct path from S to E with cost c*a.(d+bc*a)*ε = c*a(d+bc*a)*

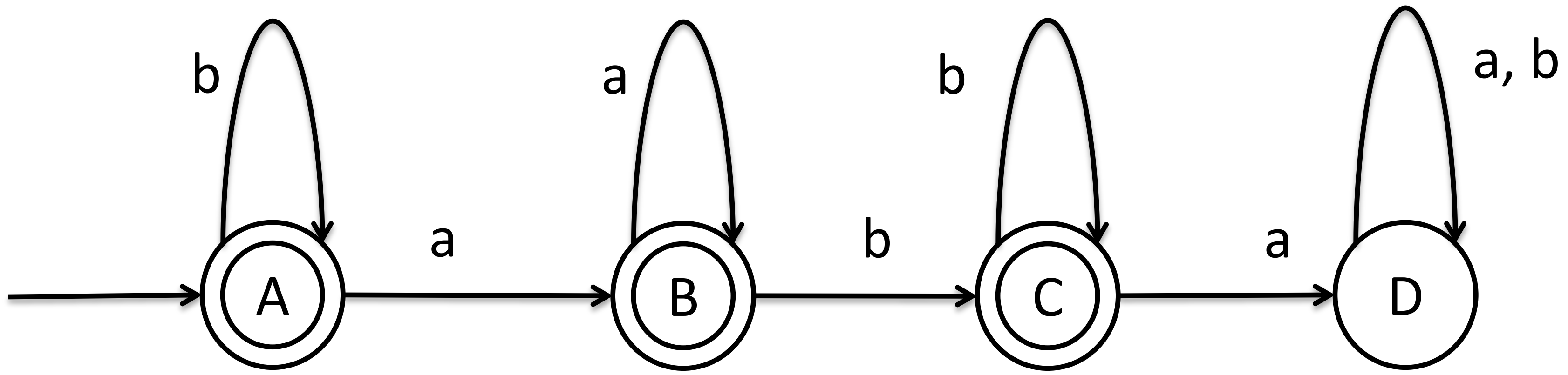
Example 2 : State Elimination Method



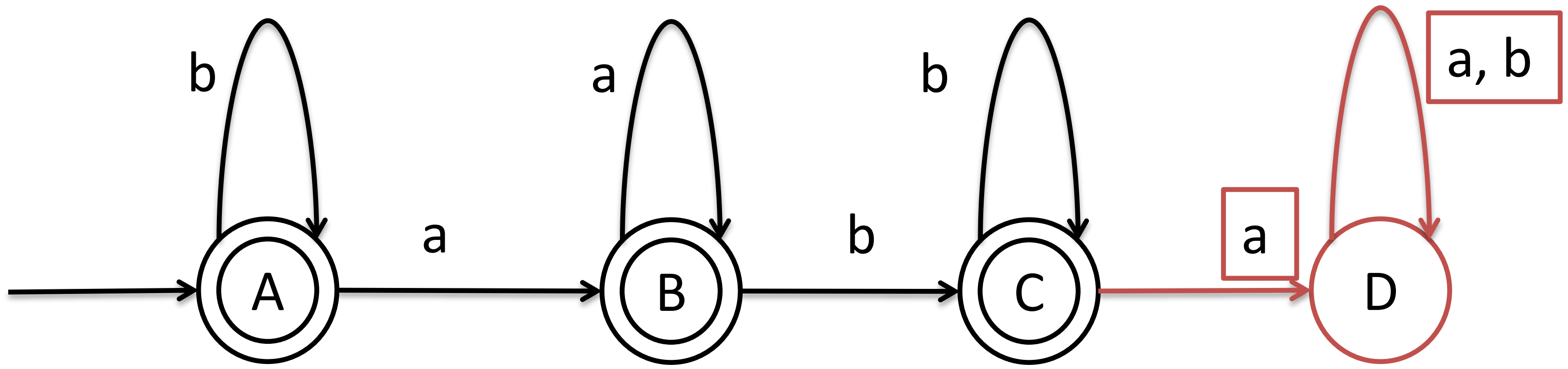
Eliminating state q2:

As there is a path going from S to E via state q2, so after eliminating q2, there will be a direct path from S to E with cost c*a.(d+bc*a)*ε = c*a(d+bc*a)*

Example 3: State Elimination Method

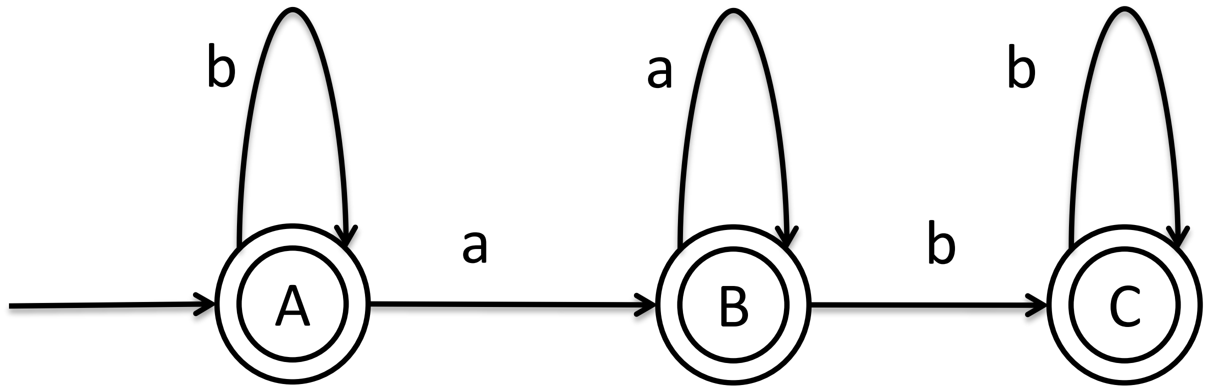


Example 3: State Elimination Method



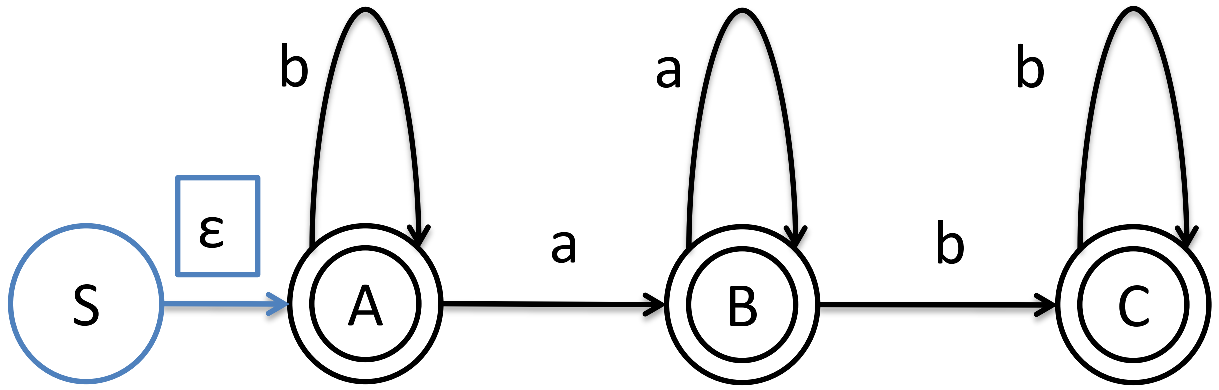
Eliminating the dead state

Example 3: State Elimination Method


$$\begin{tikzcd}
\arrow[r] & A \arrow[loop above, "b"] \arrow[r, "a"] & B \arrow[loop above, "a"] \arrow[r, "b"] & C \arrow[loop above, "b"]
\end{tikzcd}$$

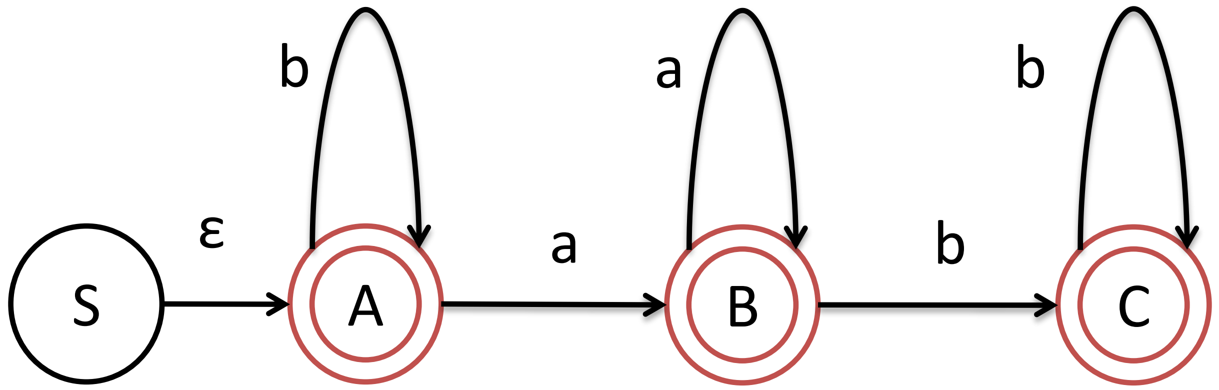
Eliminating the dead state

Example 3: State Elimination Method



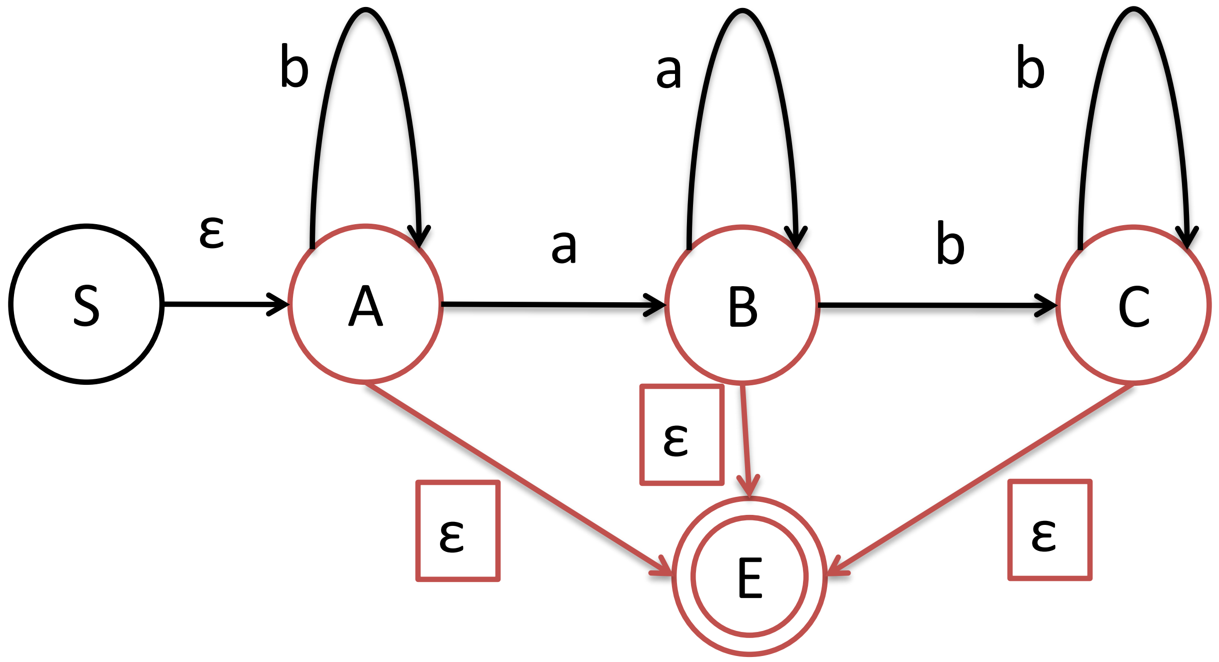
If there exists any incoming edge to the initial state, then create a new initial state having no incoming edge to it.

Example 3: State Elimination Method



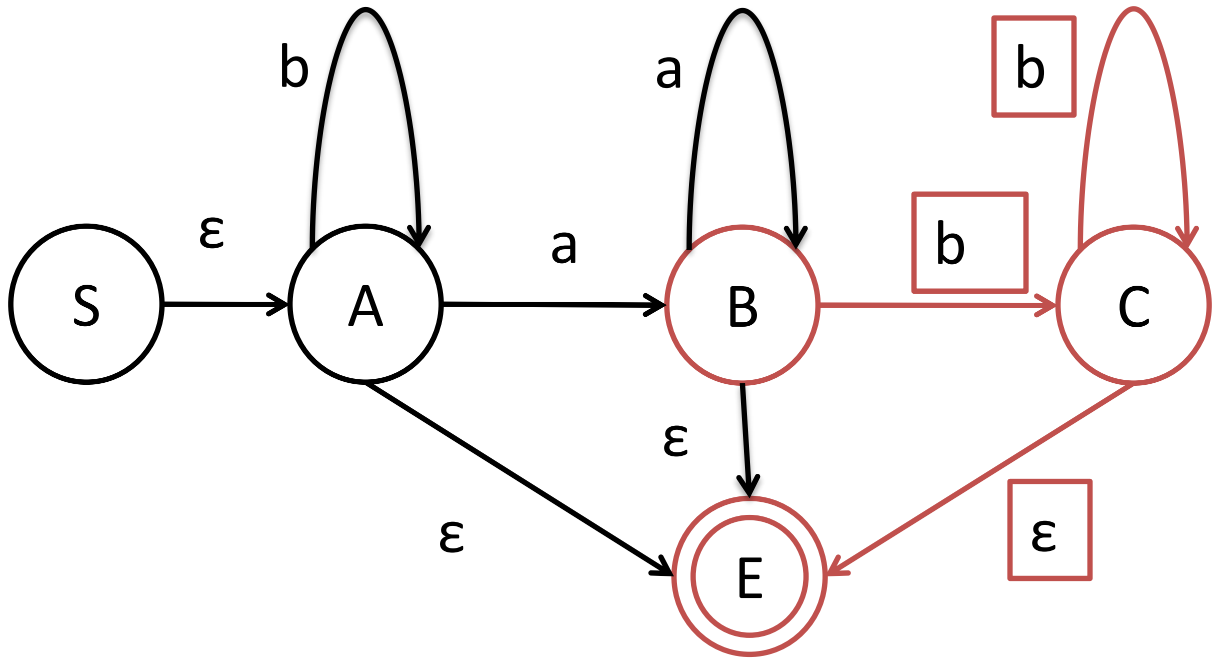
If there exists multiple final states in the DFA, then convert all the final states into non-final states and create a new single final state.

Example 3: State Elimination Method



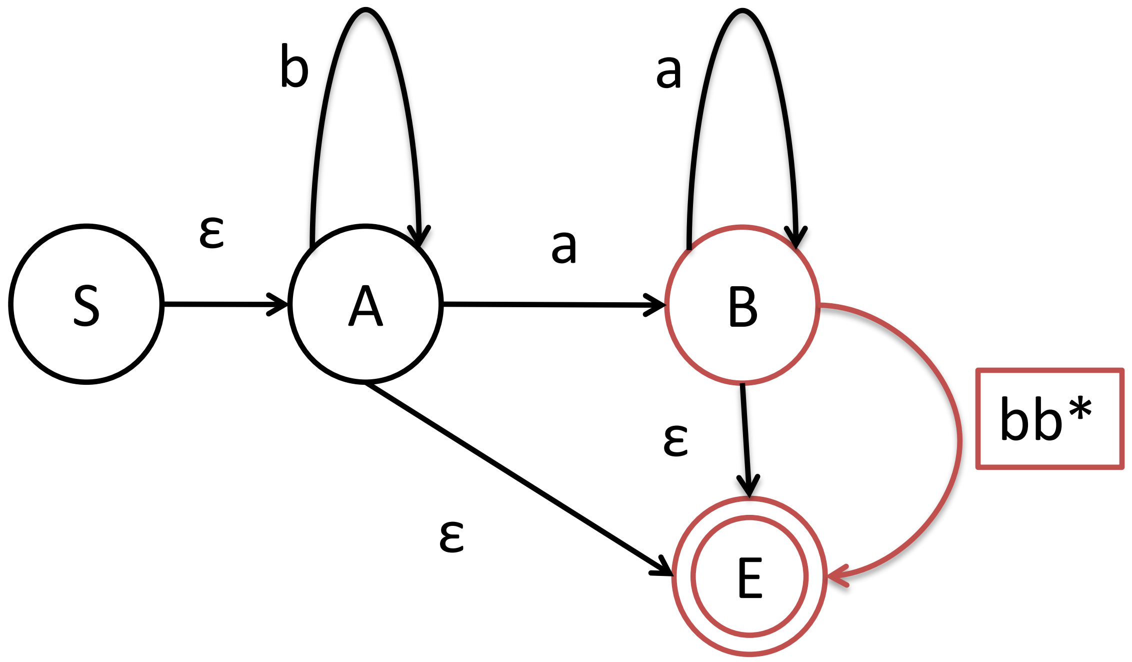
If there exists multiple final states in the DFA, then convert all the final states into non-final states and create a new single final state.

Example 3: State Elimination Method



Eliminating state C:
There is a path from state B to state E via state C.
On eliminating state C, there is a direct path from B to E with cost **bb*ε = bb***

Example 3: State Elimination Method

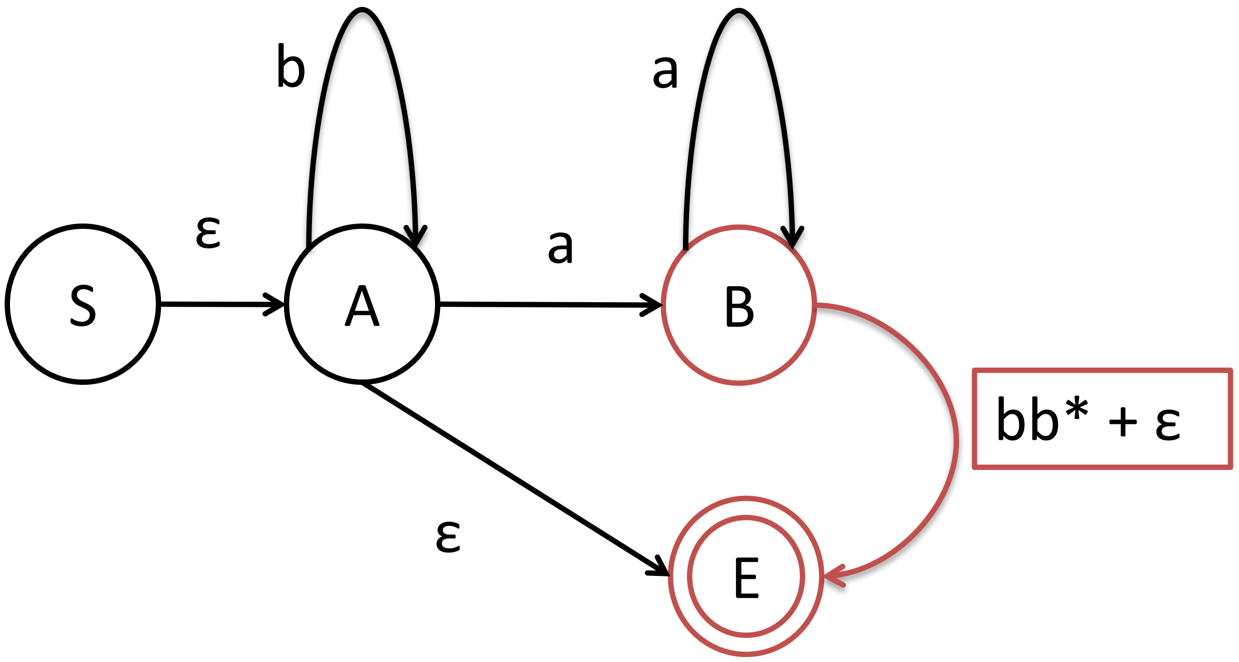


Eliminating state C:

There is a path from state B to state E via state C.

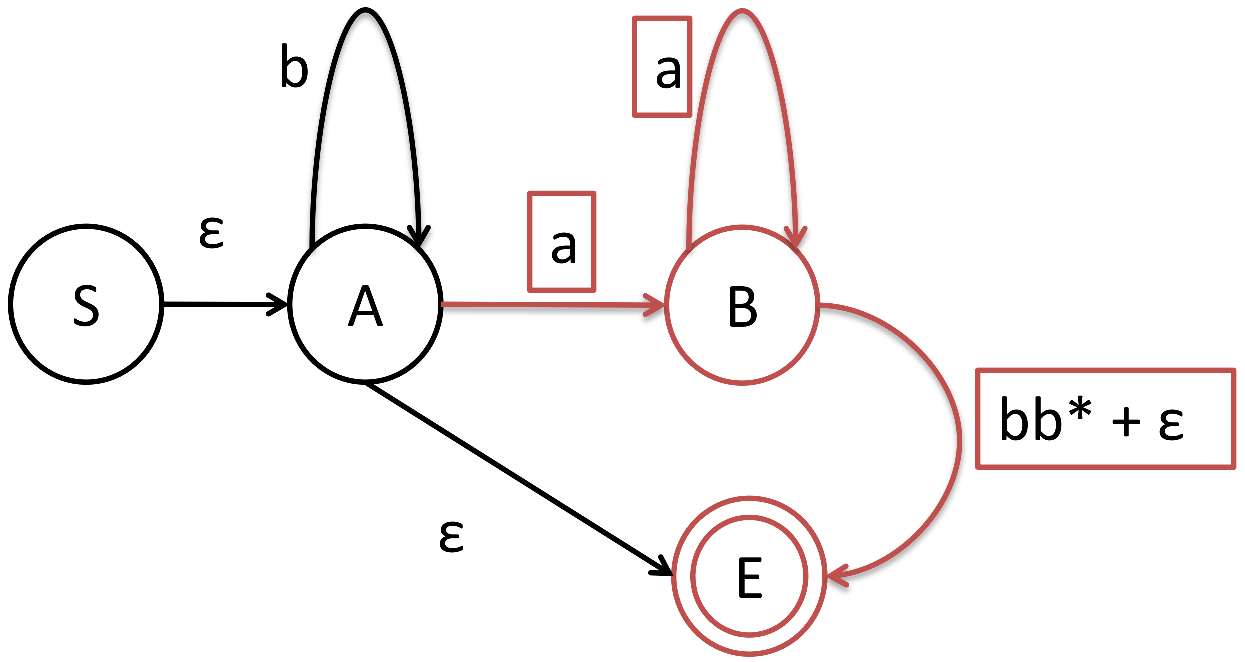
On eliminating state C, there is a direct path from B to E with cost **bb*ε = bb***

Example 3: State Elimination Method



After Eliminating state C:
There are two paths from B to E
So, the cost will be **bb* + ε**

Example 3: State Elimination Method

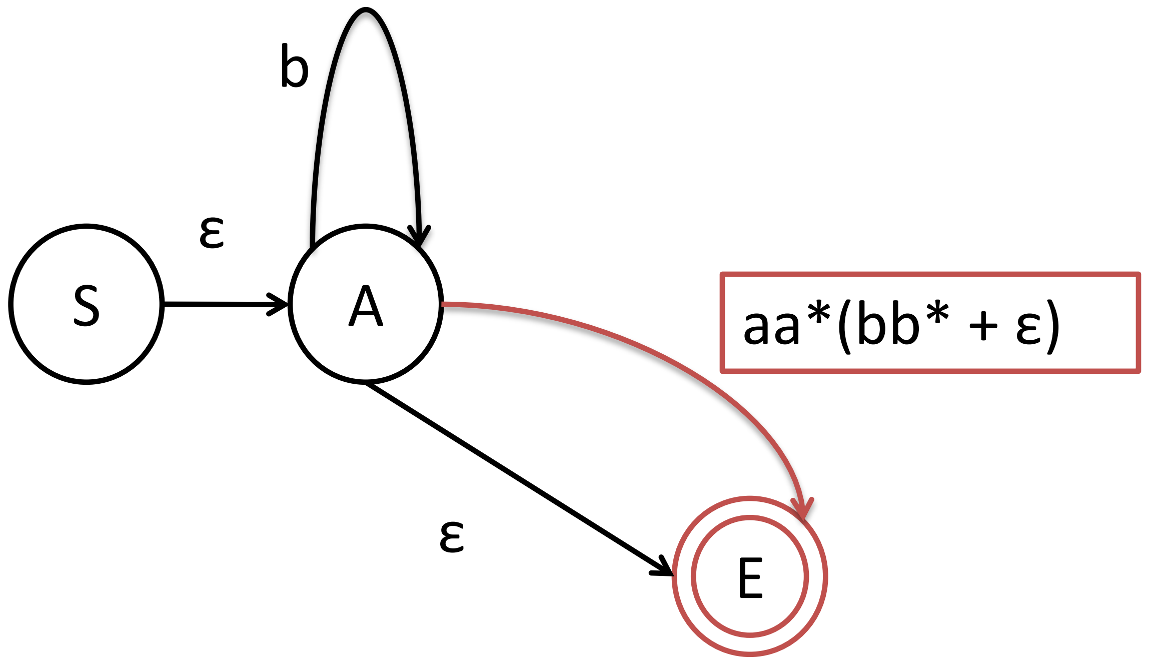


Eliminating state B:

There is a path from state A to state E via state B.

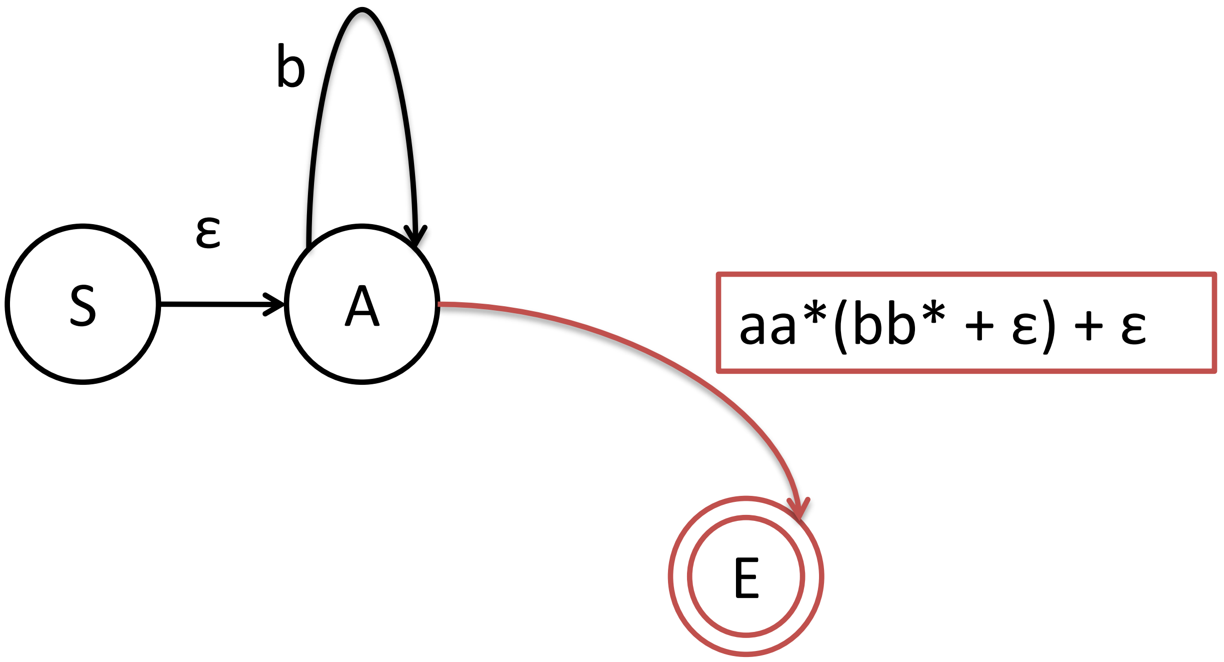
On eliminating state B, there is a direct path from A to E with cost **aa*(bb* + ε)**

Example 3: State Elimination Method



Eliminating state B:
There is a path from state A to state E via state B.
On eliminating state B, there is a direct path from A to E with cost **aa*(bb* + ε)**

Example 3: State Elimination Method

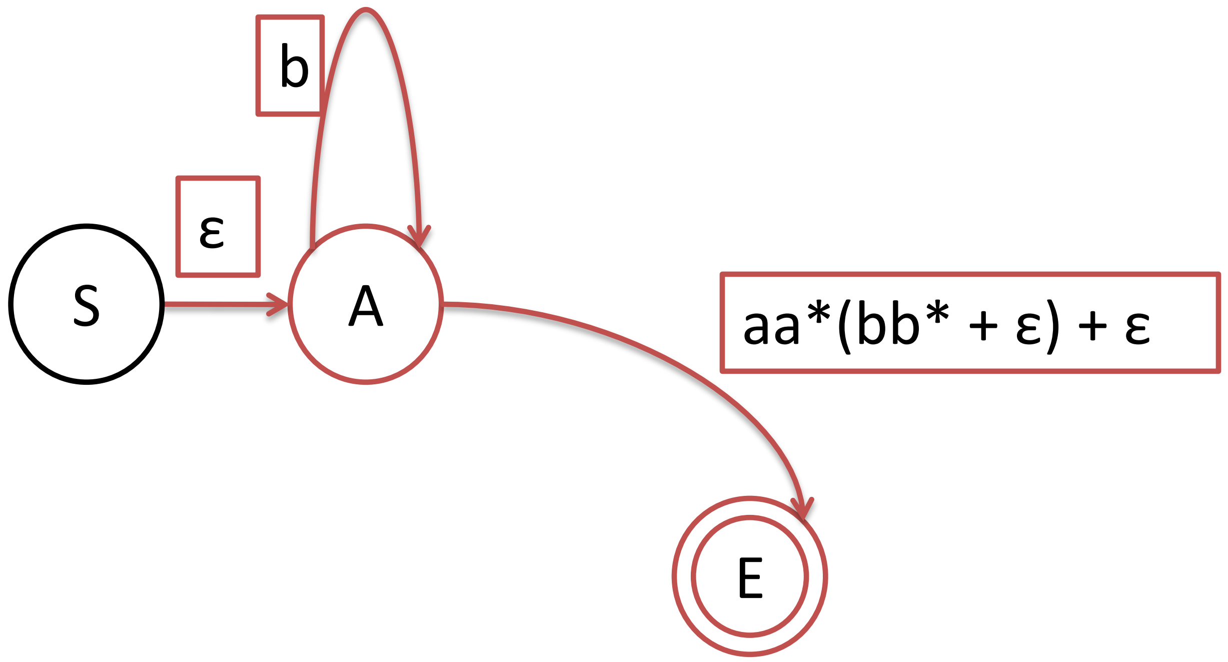


After Eliminating state B:

There are two paths from A to E

So, the cost will be **aa*(bb* + ε) + ε**

Example 3: State Elimination Method

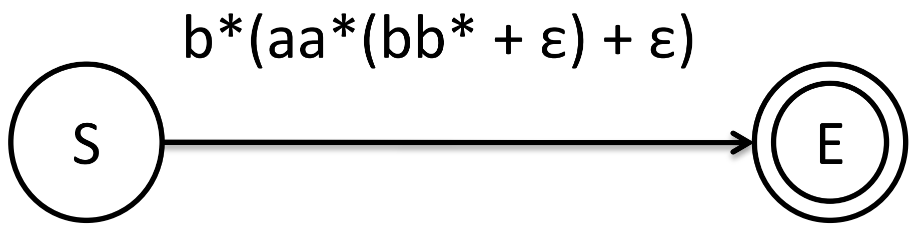


Eliminating state A:

There is a path from state S to state E via state A.

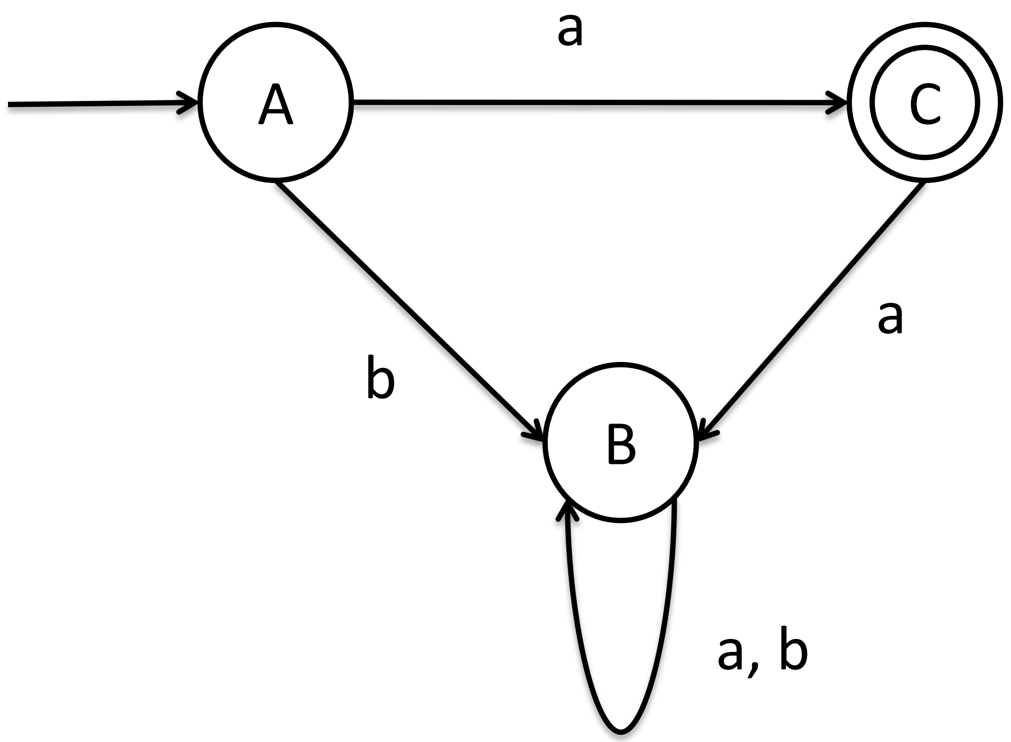
On eliminating state A, there is a direct path from S to E with cost **ε.b*.(aa*(bb* + ε)+ε)= b*(aa*(bb*+ε) +ε)**

Example 3: State Elimination Method

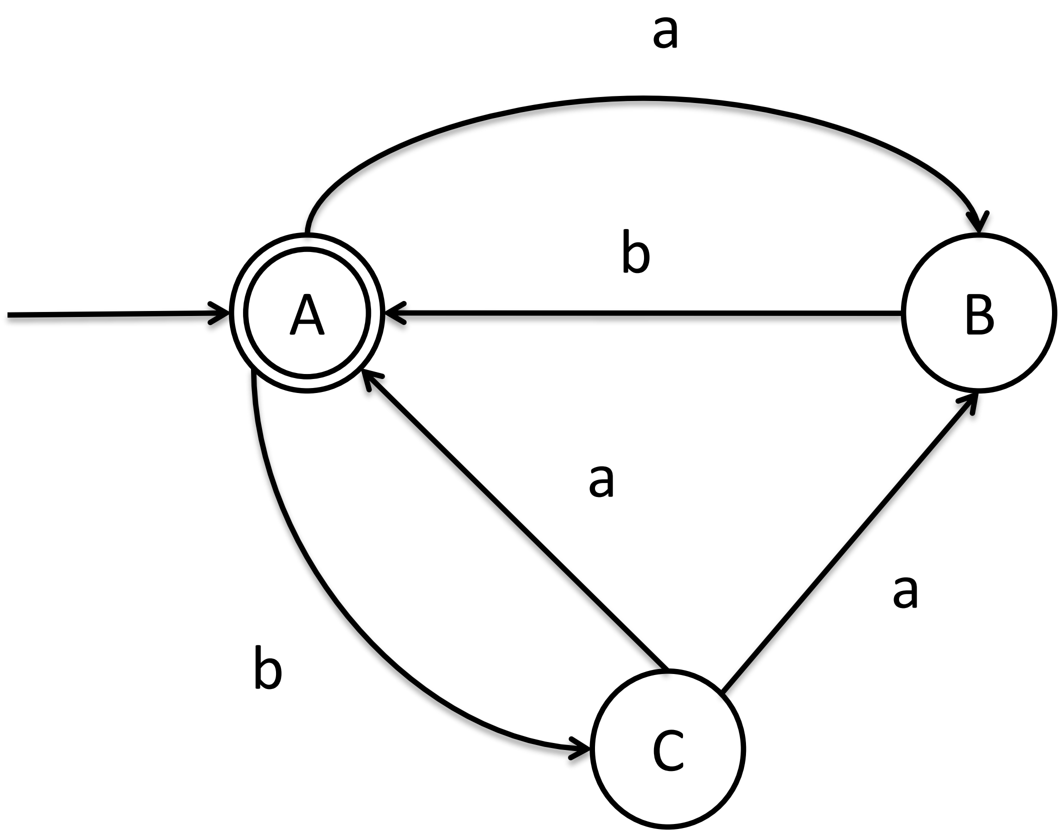


Eliminating state A:
There is a path from state S to state E via state A.
On eliminating state A, there is a direct path from S to E with cost **ε.b*.(aa*(bb* + ε)+ε)= b*(aa*(bb*+ε) +ε)**

Example to Try



Example to Try



Example to Try