CS6109 – COMPILER DESIGN

Module – 2

Presented By
Dr. S. Muthurajkumar,
Assistant Professor,
Dept. of CT, MIT Campus,
Anna University, Chennai.

MODULE - 2

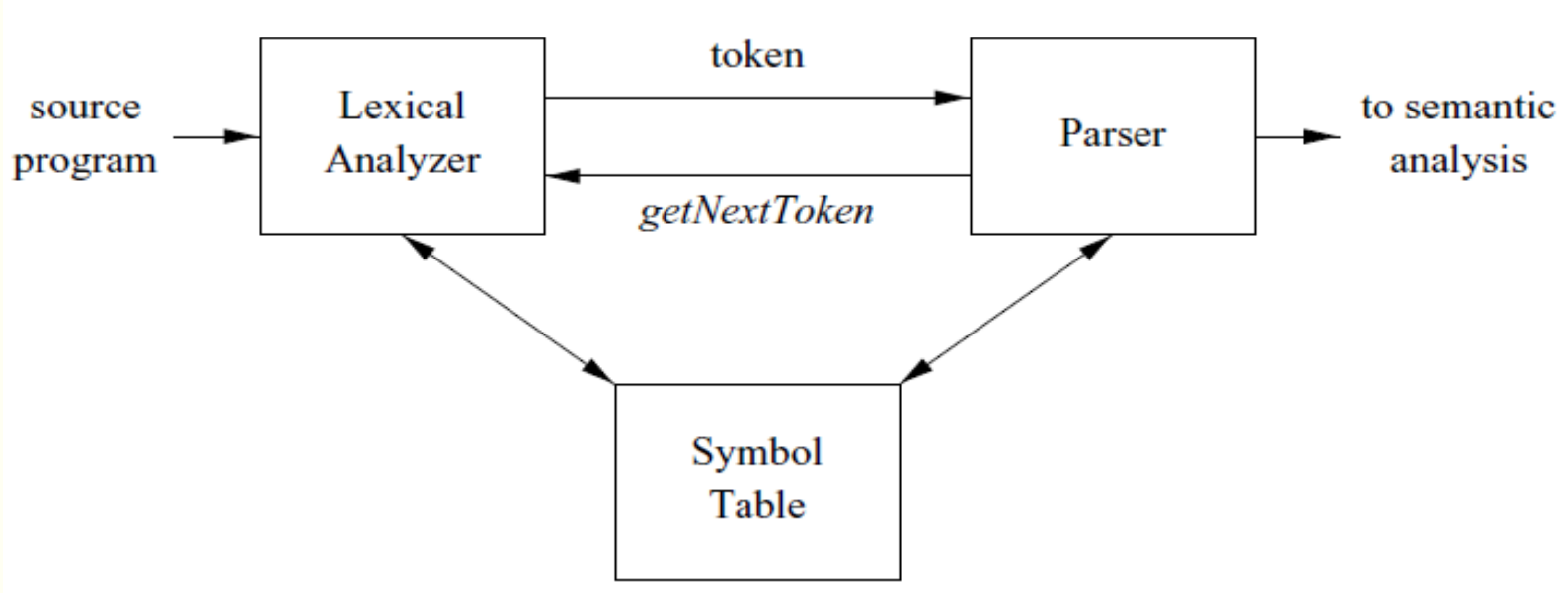
- Role of a lexical analyzer
- Recognition of Tokens
- Specification of Tokens
- Finite Automata (FA)
- Deterministic Finite Automata (DFA)
- Non-deterministic Finite Automata (NFA)
- Finite Automata with Epsilon Transitions
- NFA to DFA conversion
- Minimization of Automata

ROLE OF A LEXICAL ANALYZER

- 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.
- The stream of tokens is sent to the parser for syntax analysis.
- It is common for the lexical analyzer to interact with the symbol table as well.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.
- In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.

ROLE OF A LEXICAL ANALYZER

- The interaction is implemented by having the parser call the lexical analyzer.
- 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.



Interactions between the lexical analyzer and the parser

ROLE OF A LEXICAL ANALYZER

- The lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes.
- One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input).
- Another task is correlating error messages generated by the compiler with the source program.
- **Lexical analyzers are divided into a cascade of two processes:**
  a) Scanning consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
  b) Lexical analysis proper is the more complex portion, which produces tokens from the output of the scanner.

ROLE OF A LEXICAL ANALYZER

- Lexical Analysis Versus Parsing
- Tokens, Patterns, and Lexemes
- Attributes for Tokens
- Lexical Errors

ROLE OF A LEXICAL ANALYZER

- **Lexical Analysis Versus Parsing**
  - Simplicity of design is the most important consideration.
  - Compiler efficiency is improved.
  - Compiler portability is enhanced.

ROLE OF A LEXICAL ANALYZER

- **Tokens, Patterns, and Lexemes**
- **Tokens**
  - A token is a pair consisting of a token name and an optional attribute value.
  - The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier.
- **Patterns**
  - A pattern is a description of the form that the lexemes of a token may take.
  - In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- **Lexemes**
  - 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.

ROLE OF A LEXICAL ANALYZER

- **Tokens, Patterns, and Lexemes**
- printf("Total = %d\n", score);
- both printf and score are lexemes matching the pattern for token id, and
- "Total = %d\n" is a lexeme matching literal.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

ROLE OF A LEXICAL ANALYZER

- **Tokens, Patterns, and Lexemes**

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token comparison mentioned in slide number 9 Figure.
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 parentheses, comma, and semicolon.

ROLE OF A LEXICAL ANALYZER

- **Attributes for Tokens**
  - When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched.
  - For example, the pattern for token number matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program.
  - Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

ROLE OF A LEXICAL ANALYZER

- **Attributes for Tokens**
  - E = M * C ** 2

    are written below as a sequence of pairs.

    <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>

    <number, integer value 2>

ROLE OF A LEXICAL ANALYZER

- **Lexical Errors**
- It is hard for a lexical analyzer to tell, without the aid of other components, that there is a source-code error.
- For instance, if the string fi is encountered for the first time in a C program in the context:

  fi (a == f(x)) ...
- a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier.
- Since fi is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler - probably the parser in this case - handle an error due to transposition of the letters.

ROLE OF A LEXICAL ANALYZER

- **Lexical Errors**
  - However, suppose a situation arises in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches any prefix of the remaining input.
  - The simplest recovery strategy is “panic mode” recovery.
  - Error-recovery actions are:

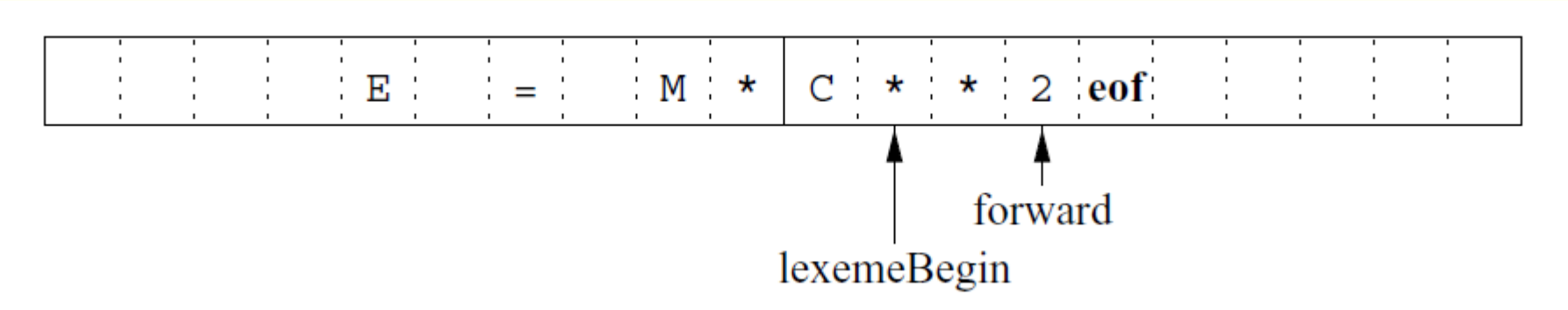
1. Delete one character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

INPUT BUFFERING

- Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded.
- This task is made dificult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.
- Two Methods
  - Buffer Pairs
  - Sentinels

INPUT BUFFERING

- **Buffer Pairs**
- Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.



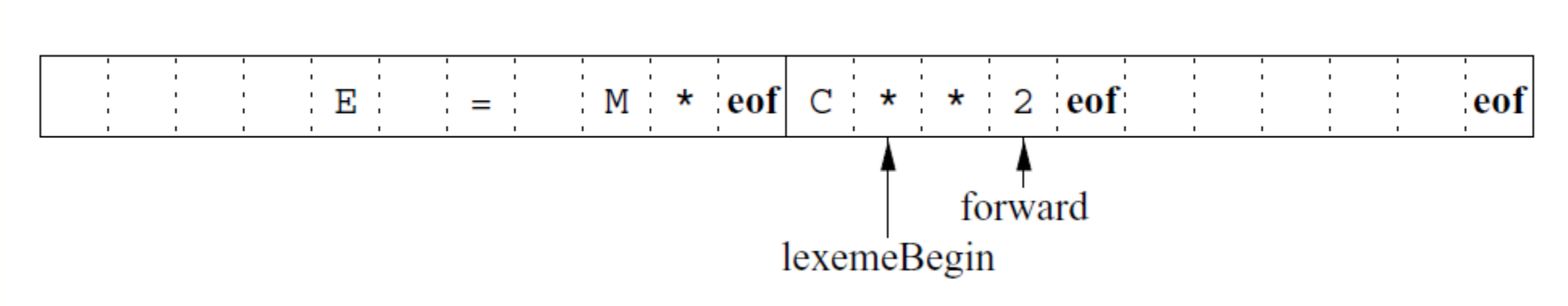
Using a pair of input buffers

INPUT BUFFERING

- **Buffer Pairs**
- Two pointers to the input are maintained:
  1. Pointer lexemeBegin, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  2. Pointer forward scans ahead until a pattern match is found

INPUT BUFFERING

- **Sentinels**
- Each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch).
- 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.



Sentinels at the end of each buffer

SPECIFICATION OF TOKENS

- Regular expressions are an important notation for specifying lexeme patterns.
- While they cannot express all possible patterns, they are very effective in specifying those types of patterns that we actually need for tokens.
  - Strings and Languages
  - Operations on Languages
  - Regular Expressions
  - Regular Definitions
  - Extensions of Regular Expressions

SPECIFICATION OF TOKENS

- **Strings and Languages**
- An alphabet is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0,1} is the binary alphabet.
- ASCII is an important example of an alphabet; it is used in many software systems.
- 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 |s|, is the number of occurrences of symbols in s.

SPECIFICATION OF TOKENS

- **Strings and Languages**
- For example, banana is a string of length six.
- The empty string, denoted ϵ, is the string of length zero.
- A language is any countable set of strings over some fixed alphabet.
- This definition is very broad.
- Abstract languages like Ø, the empty set, or $\{\epsilon\}$, the set containing only the empty string, are languages under this definition.

SPECIFICATION OF TOKENS

- **Terms for Parts of Strings**
- The following string-related terms are commonly used:

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

SPECIFICATION OF TOKENS

- **Operations on Languages**
  - Union
  - Concatenation
  - Closure
- **Union**
  - Union is the familiar operation on sets.
- **Concatenation**
  - The concatenation of languages is all strings formed by taking a string from the first language and a string from the second language, in all possible ways, and concatenating them.

SPECIFICATION OF TOKENS

- **Operations on Languages**
- **Closure**
- The (Kleene) closure of a language L, denoted L*, is the set of strings you get by concatenating L zero or more times.
- Note that L^0, 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.

SPECIFICATION OF TOKENS

- **Operations on Languages**

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

Definitions of operations on languages

SPECIFICATION OF TOKENS

Example:

Let L be the set of letters {A, B, … , Z, a, b, … , z} and let D be the set of digits {0, 1, … 9}. We may think of L and D in two, essentially equivalent, ways. One way is that L and D are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that L and D are languages, all of whose strings happen to be of length one. Here are some other languages that can be constructed from languages L and D, using the operators of slide no. 25:

ANSWER:

1. 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.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.

SPECIFICATION OF TOKENS

ANSWER:

1. 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.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is the set of all 4-letter strings.
4. L* is the set of all strings of letters, including ϵ, the empty string.
5. L(L U D)* is the set of all strings of letters and digits beginning with a letter.
6. D^+ is the set of all strings of one or more digits.

SPECIFICATION OF TOKENS

- **Regular Expressions**
- The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r's subexpressions. Here are the rules that define the regular expressions over some alphabet $\sum$ and the languages that those expressions denote.
- **BASIS: There are two rules that form the basis:**

1. ϵ is a regular expression, and L(ϵ) is {ϵ}, that is, the language whose sole member is the empty string.
2. If a is a symbol in $\sum$, then a is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with a in its one position. Note that by convention, we use italics for symbols, and boldface for their corresponding regular expression.

SPECIFICATION OF TOKENS

- **Regular Expressions**
- **INDUCTION:** There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages L(r) and L(s), respectively.

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

SPECIFICATION OF TOKENS

- **Regular Expressions**
- Regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator * has highest precedence and is left associative.

b) Concatenation has second highest precedence and is left associative.

c) | has lowest precedence and is left associative.

- (a)|((b) * (c)) by a|b*c.
- Both expressions denote the set of strings that are either a single a or are zero or more b's followed by one c.

SPECIFICATION OF TOKENS

- **Regular Expressions - Example 3.4 : Let ∑ = {a, b}.**

1. The regular expression a|b denotes the language {a, b}.
2. (a|b)(a|b) denotes {aa, ab, ba, bb}, the language of all strings of length two over the alphabet ∑. Another regular expression for the same language is aa|ab|ba|bb.
3. a* denotes the language consisting of all strings of zero or more a's, that is, {ϵ, a, aa, aaa, …}.
4. (a|b)* denotes the set of all strings consisting of zero or more instances of a or b, that is, all strings of a's and b's: {ϵ, a, b, aa, ab, ba, bb, aaa, …}. Another regular expression for the same language is (a*b*)*.
5. a|a*b denotes the language {a, b, ab, aab, aaab, … }, that is, the string a and all strings consisting of zero or more a's and ending in b.

SPECIFICATION OF TOKENS

- **Regular Expressions**
- A language that can be defined by a regular expression is called a regular set.
- If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s. For instance, (a|b) = (b|a).
- There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent.

Algebraic laws for regular expressions

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

SPECIFICATION OF TOKENS

- **Regular Definitions**
- Certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If $\sum$ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

  $d_1 \rightarrow r_1$

  $d_3 \rightarrow r_2$

  …

  $d_n \rightarrow r_n$

- where:

1. Each d_i is a new symbol, not in $\sum$ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet $\sum$ U $\{d_1, d_2, \ldots, d_{i-1}\}$.

SPECIFICATION OF TOKENS

- **Extensions of Regular Expressions**
  - One or more instances
  - Zero or one instance
  - Character classes

SPECIFICATION OF TOKENS

- **One or more instances**
- The unary, postfix operator + represents the positive closure of a regular expression and its language.
- That is, if r is a regular expression, then $(r)^+$ denotes the language $((r))^+$.
- The operator + has the same precedence and associativity as the operator *.
- Two useful algebraic laws, $r^* = r^+ \mid \epsilon$ and $r^+ = rr^* = r^*r$ relate the Kleene closure and positive closure.

SPECIFICATION OF TOKENS

- **Zero or one instance**
- The unary postfix operator ? means “zero or one occurrence”.
- That is, r ? is equivalent to r|ϵ, or put another way, L(r?) = L(r) U {ϵ}.
- The ? operator has the same precedence and associativity as * and +.

SPECIFICATION OF TOKENS

- **Character classes**
- A regular expression $a_1|a_2| \ldots |a_n$, where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2 \ldots a_n]$.
- More importantly, when $a_1, a_2, \ldots, a_n$ form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a_1-a_n, that is, just the first and last separated by a hyphen.
- Thus, [abc] is shorthand for a|b|c, and [a-z] is shorthand for a|b| … |z.

RECOGNITION OF TOKENS

- To study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

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

expr → *term* relop *term*
| *term*

term → id
| number

RECOGNITION OF TOKENS

- **Patterns for Tokens**

  digit → [0-9]

  digits → digit+

  number → digits (. digits) ? (E [+-]? digits)?

  letter → [A-Z a-z]

  id → letter (letter | digit)*

  if → if

  then → then

  else → else

  relop → < | > | <= | >= | = | <>

  white-space, by recognizing the \token" ws defined by:

  ws → (blank | tab | newline)+

RECOGNITION OF TOKENS

- **Tokens, their patterns, and attribute values**

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

RECOGNITION OF TOKENS

- Transition Diagrams
- Recognition of Reserved Words and Identifiers
- Completion of the Running Example
- Architecture of a Transition-Diagram-Based Lexical Analyzer

RECOGNITION OF TOKENS

- **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.
- Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.
- We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the *forward* pointer.
- Edges are directed from one state of the transition diagram to another.
- Each edge is *labeled* by a symbol or set of symbols.

RECOGNITION OF TOKENS

- **Transition Diagrams**
- Certain states are said to be accepting, or final.
- In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state.
- One state is designated the start state, or initial state;

RECOGNITION OF TOKENS

- **Transition Diagrams -** Important conventions about transition diagrams are:

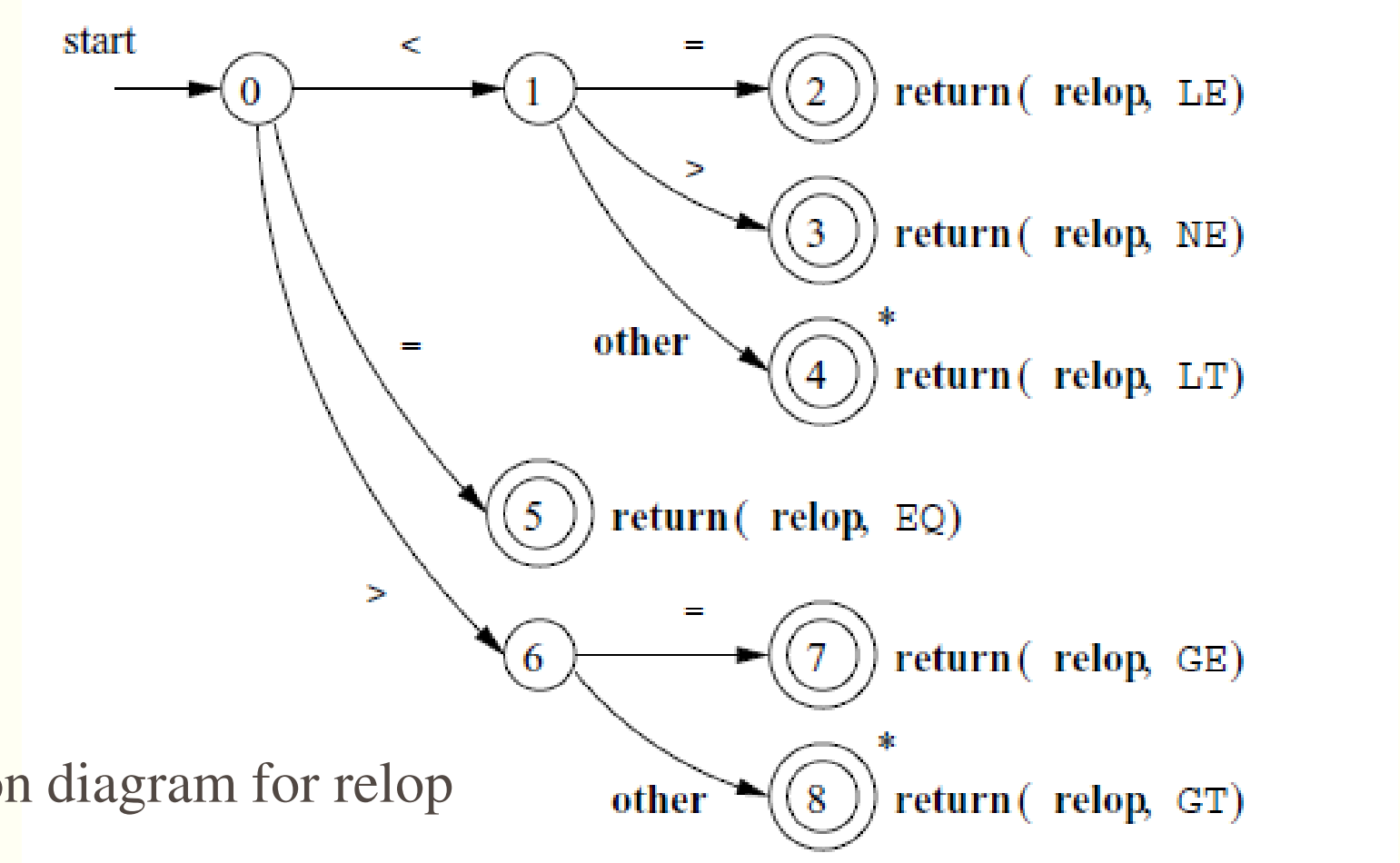
1. Certain states are said to be **accepting, or final**.
   a. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the lexemeBegin and forward pointers.
   b. We always indicate an accepting state by a double circle, and if there is an action to be taken typically returning a token and an attribute value to the parser we shall attach that action to the accepting state.
2. In addition, if it is necessary to retract the **forward pointer** one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state.
   a. In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting state.

RECOGNITION OF TOKENS

- **Transition Diagrams -** Important conventions about transition diagrams are:

3. One state is designated the **start state, or initial state**; it is indicated by an edge, labeled "start", entering from nowhere.
   a. The transition diagram always begins in the start state before any input symbols have been read.

RECOGNITION OF TOKENS



Transition diagram for relop

RECOGNITION OF TOKENS - Transition Diagrams

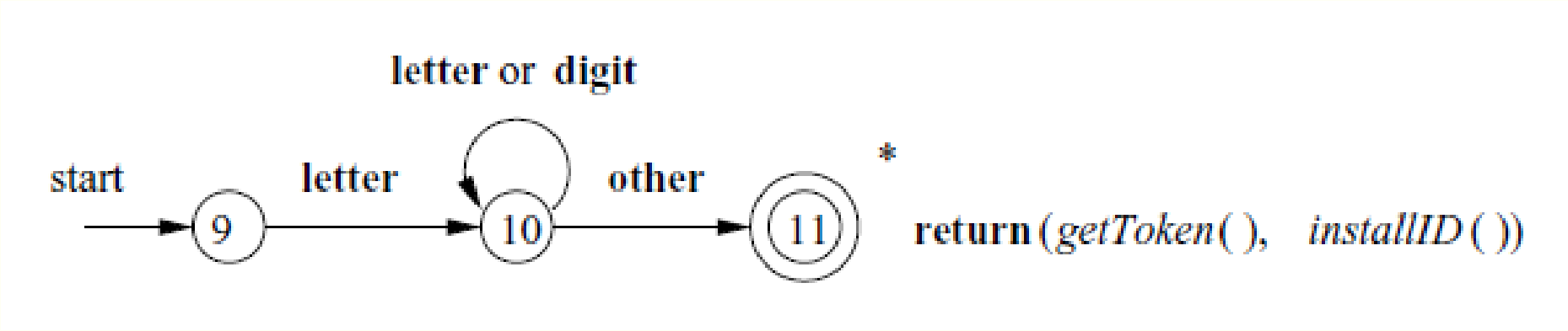
- Transition diagram that recognizes the lexemes matching the token relop.
- We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for relop.
- we can only be looking at <, <>, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token relop with attribute LE, the symbolic constant representing this particular comparison operator.
- If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found.
- On any other character, the lexeme is <, and we enter state 4 to return that information.
- Note, however, that state 4 has a * to indicate that we must retract the input one position.

RECOGNITION OF TOKENS

- On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme.
- We immediately return that fact from state 5.
- The remaining possibility is that the first character is >.
- Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is >= (if we next see the = sign), or just > (on any other character).
- Note that if, in state 0, we see any character besides <, =, or >, we can not possibly be seeing a relop lexeme, so this transition diagram will not be used.

RECOGNITION OF TOKENS

- **Recognition of Reserved Words and Identifiers**
- Keywords like if or then are reserved

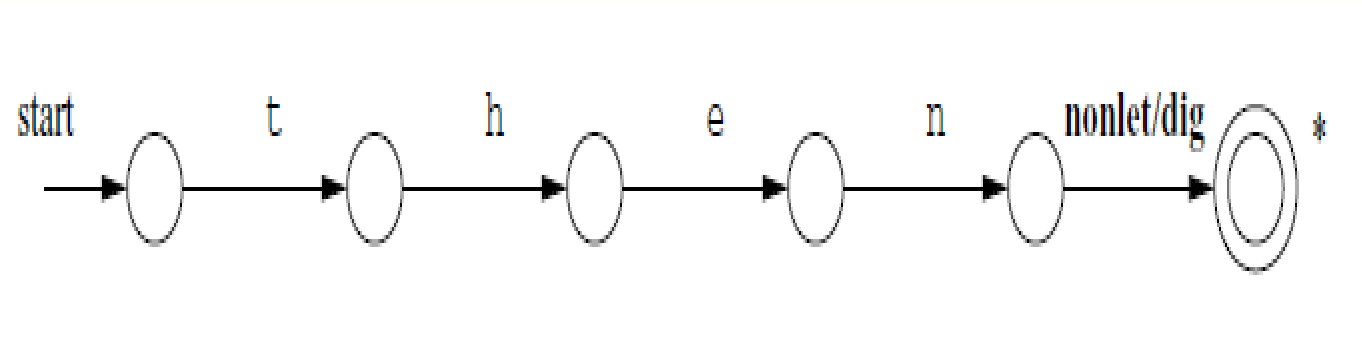


A transition diagram for id's and keywords

RECOGNITION OF TOKENS

- **Recognition of Reserved Words and Identifiers**
- There are two ways that we can handle reserved words that look like identifiers:

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



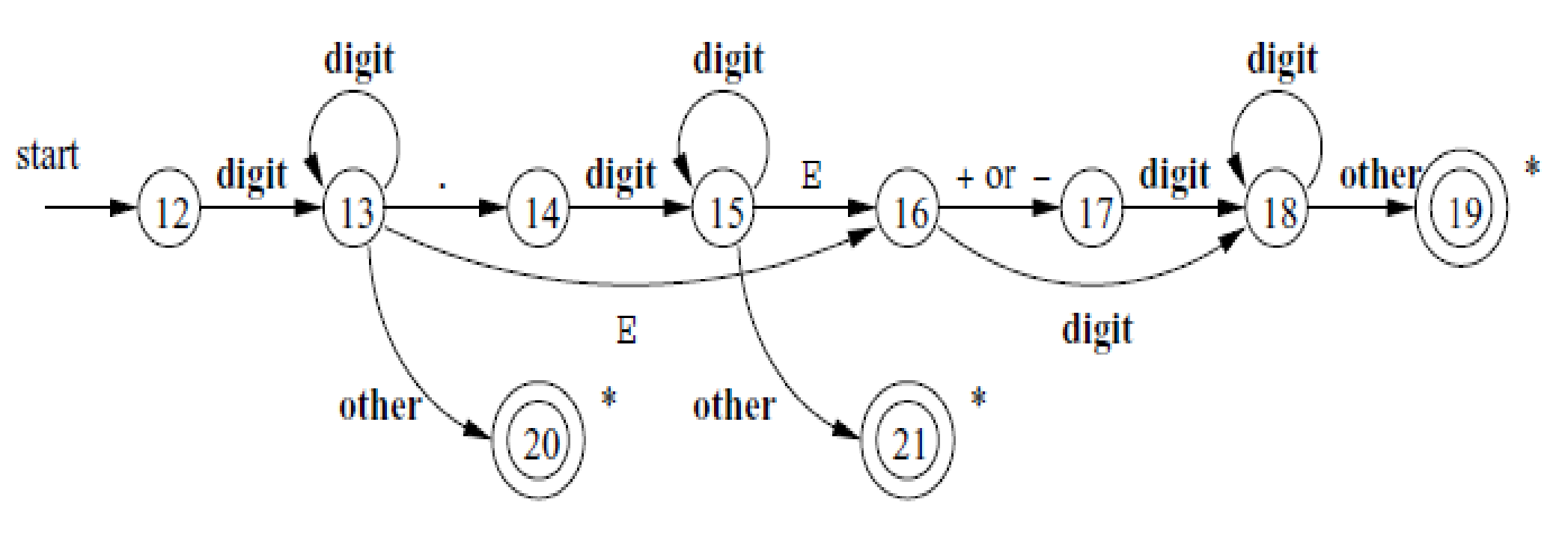
Hypothetical transition diagram for the keyword then

RECOGNITION OF TOKENS

- **Completion of the Running Example**
- The transition diagram for token number is shown in Fig, and is so far the most complex diagram we have seen.
- Beginning in state 12, if we see a digit, we go to state 13.
- In that state, we can read any number of additional digits.
- However, if we see anything but a digit, dot, or E, we have seen a number in the form of an integer; 123 is an example.
- That case is handled by entering state 20, where we return token number and a pointer to a table of constants where the found lexeme is entered.
- These mechanics are not shown on the diagram but are analogous to the way we handled identifiers.

RECOGNITION OF TOKENS

- **Completion of the Running Example**

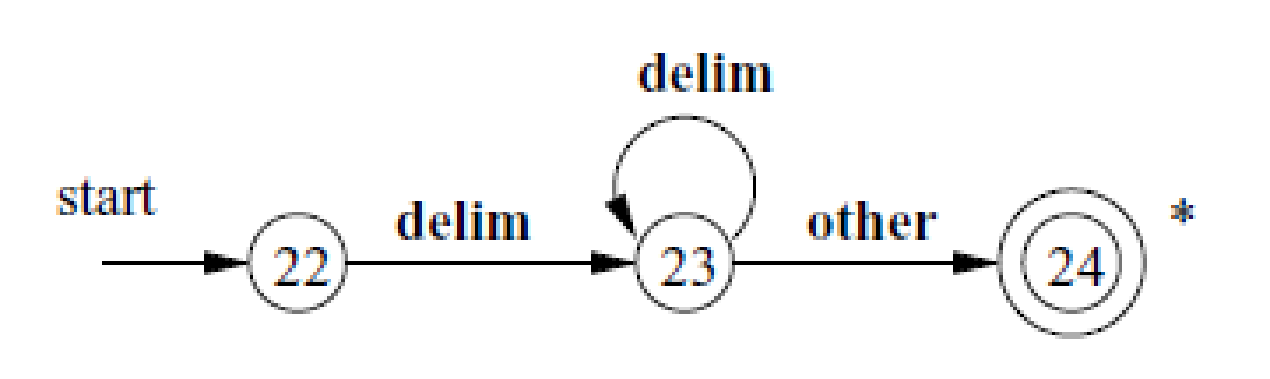


RECOGNITION OF TOKENS

- **Completion of the Running Example**
- If we instead see a dot in state 13, then we have an “optional fraction”.
- State 14 is entered, and we look for one or more additional digits; state 15 is used for that purpose.
- If we see an E, then we have an “optional exponent”, whose recognition is the job of states 16 through 19.
- Should we, in state 15, instead see anything but E or a digit, then we have come to the end of the fraction, there is no exponent, and we return the lexeme found, via state 21.

RECOGNITION OF TOKENS

- **Completion of the Running Example**
- The final transition diagram, shown in Fig. is for whitespace.
- In that diagram, we look for one or more “whitespace" characters, represented by delim in that diagram typically these characters would be blank, tab, newline, and perhaps other characters that are not considered by the language design to be part of any token.



A transition diagram for whitespace

RECOGNITION OF TOKENS

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

RECOGNITION OF TOKENS

- **Architecture of a Transition-Diagram-Based Lexical Analyzer**
- There are several ways that a collection of transition diagrams can be used to build a lexical analyzer.
- Regardless of the overall strategy, each state is represented by a piece of code.
- We may imagine a variable state holding the number of the current state for a transition diagram.
- A switch based on the value of state takes us to code for each of the possible states, where we find the action of that state.
- Often, the code for a state is itself a switch statement or multiway branch that determines the next state by reading and examining the next input character.

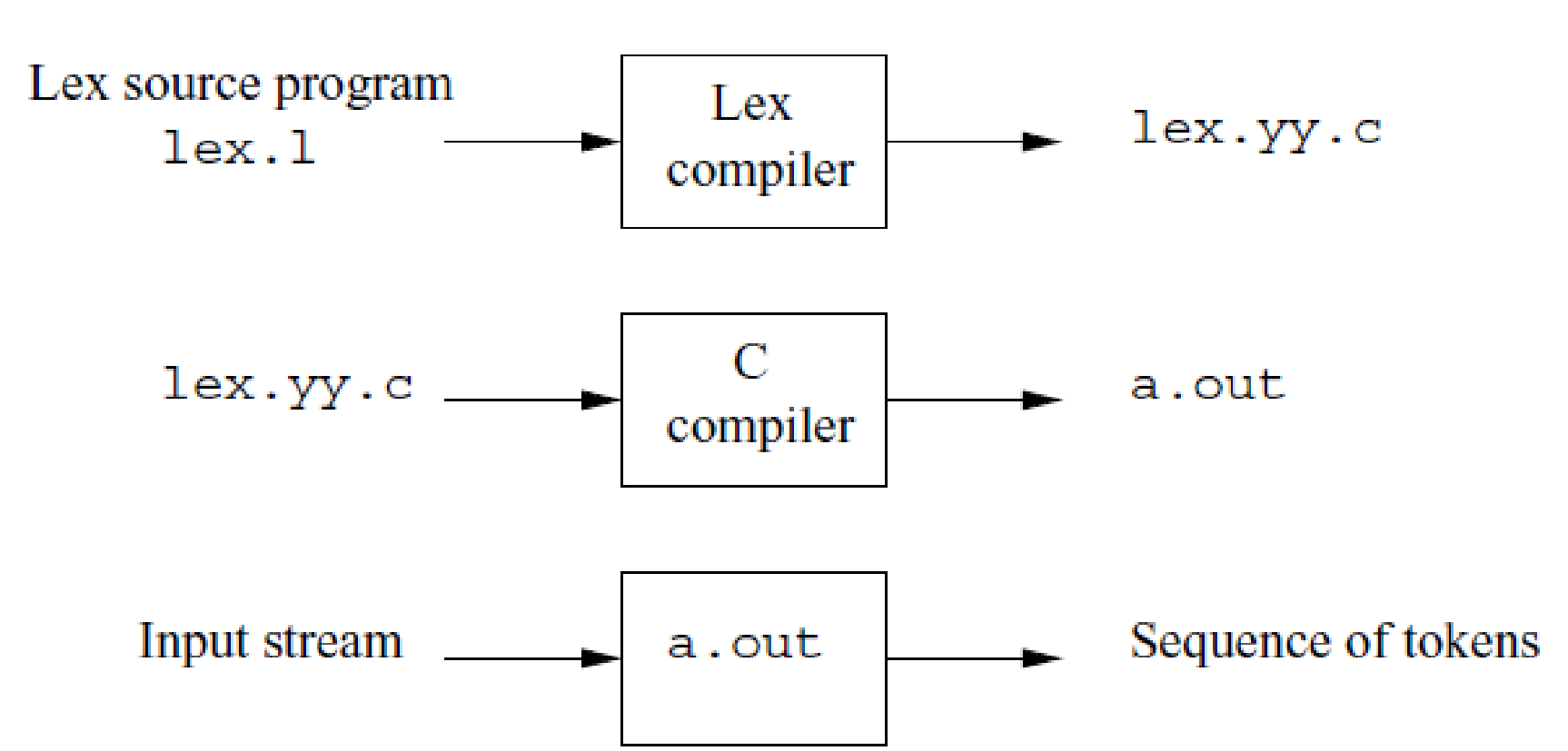
THE LEXICAL-ANALYZER GENERATOR LEX

- The Lexical-Analyzer Generator Lex
- A tool called Lex, or in a more recent implementation Flex, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens.
- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.
- Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c, that simulates this transition diagram.
  - Use of Lex
  - Structure of Lex Programs
  - Conflict Resolution in Lex
  - The Lookahead Operator

THE LEXICAL-ANALYZER GENERATOR LEX – USE OF LEX

- An input file, which we call lex.l, is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c.
- The latter file is compiled by the C compiler into a file called a.out, as always.
- The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

THE LEXICAL-ANALYZER GENERATOR LEX – USE OF LEX



Creating a lexical analyzer with Lex

THE LEXICAL-ANALYZER GENERATOR LEX – STRUCTURE OF LEX PROGRAMS

- A Lex program has the following form:

  declarations

  %%

  translation rules

  %%

  auxiliary functions

- The declarations section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.

THE LEXICAL-ANALYZER GENERATOR LEX – CONFLICT RESOLUTION IN LEX

- We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:
  1. Always prefer a longer prefix to a shorter prefix.
  2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

THE LEXICAL-ANALYZER GENERATOR LEX – THE LOOKAHEAD OPERATOR

- Lex automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input.
- However, sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters.

FINITE AUTOMATA

- We shall now discover how Lex turns its input program into a lexical analyzer.
- At the heart of the transition is the formalism known as finite automata. These are essentially graphs, like transition diagrams, with a few differences:
    1. Finite automata are recognizers; they simply say “yes” or “no” about each possible input string.
    2. Finite automata come in two favors:
        a) Nondeterministic Finite Automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ, the empty string, is a possible label.
        b) Deterministic Finite Automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

FINITE AUTOMATA

- Nondeterministic Finite Automata (NFA)
- Transition Table
- Acceptance of Input Strings by Automata
- Deterministic Finite Automata (DFA)

NONDETERMINISTIC FINITE AUTOMATA (NFA)

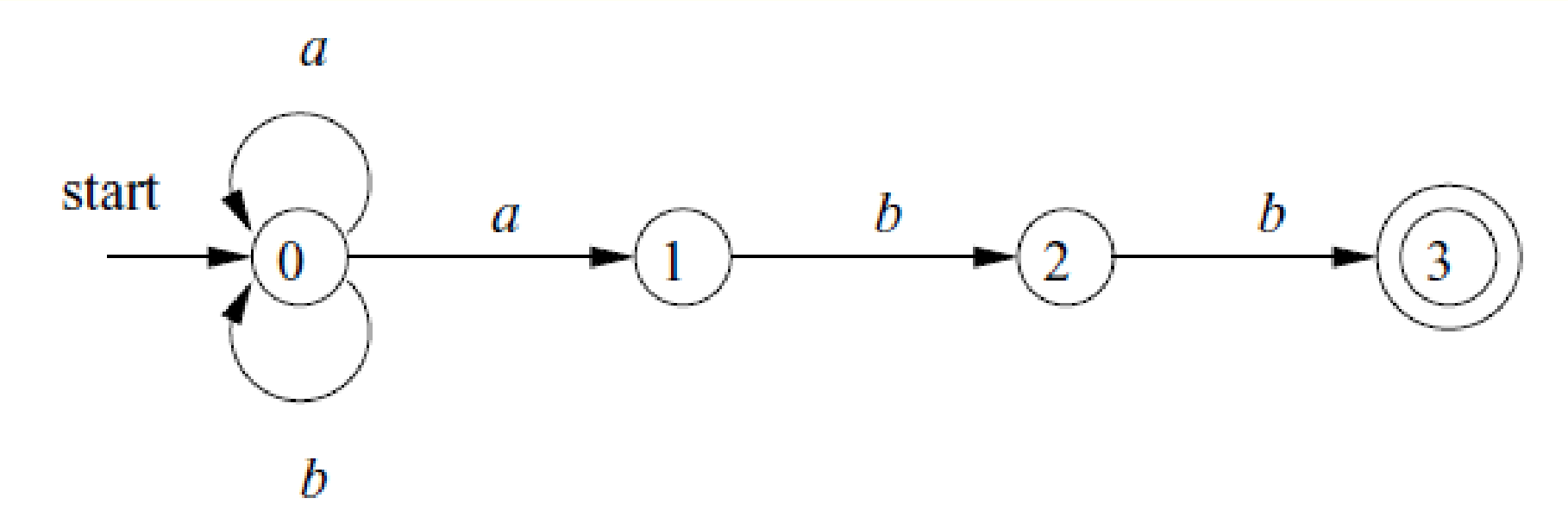
- A nondeterministic finite automaton (NFA) consists of: $S = \{S, \Sigma, \Sigma \cup \{\epsilon\}, s0, F\}$
  1. A finite set of states S.
  2. A set of input symbols Σ, the input alphabet. We assume that ϵ, which stands for the empty string, is never a member of Σ.
  3. A transition function that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of next states.
  4. A state s0 from S that is distinguished as the start state (or initial state).
  5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

NONDETERMINISTIC FINITE AUTOMATA (NFA)

- We can represent either an NFA or DFA by a transition graph, where the nodes are states and the labeled edges represent the transition function.
- There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a.
- This graph is very much like a transition diagram, except:
  a) The same symbol can label edges from one state to several different states, and
  b) An edge may be labeled by ϵ, the empty string, instead of, or in addition to, symbols from the input alphabet.

NONDETERMINISTIC FINITE AUTOMATA (NFA)

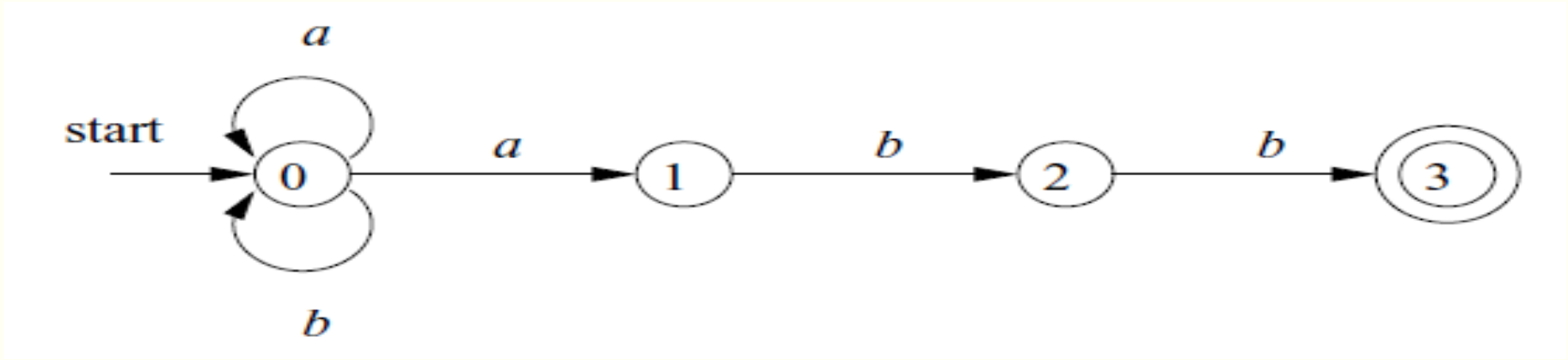
- Example:
- Regular expression (a | b)*abb



A nondeterministic finite automaton

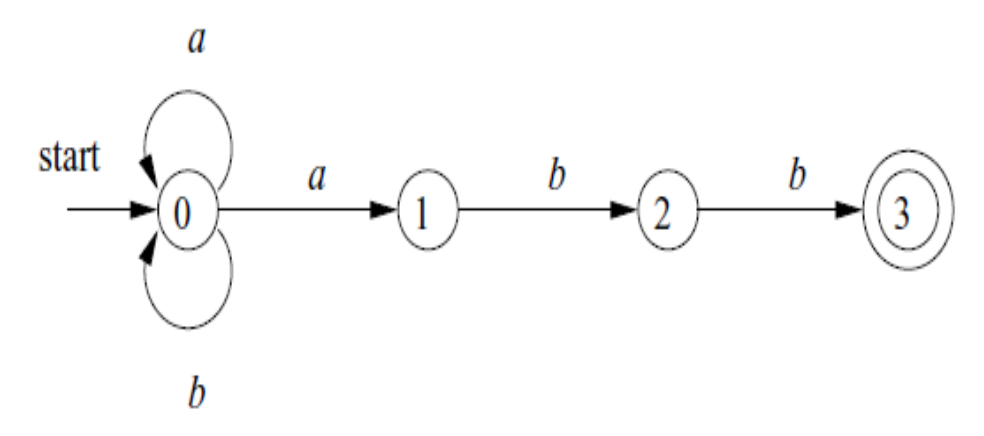
NONDETERMINISTIC FINITE AUTOMATA (NFA)

- Example:
- Regular expression (a | b)*abb
- The double circle around state 3 indicates that this state is accepting.
- Notice that the only ways to get from the start state 0 to the accepting state is to follow some path that stays in state 0 for a while, then goes to states 1, 2, and 3 by reading **abb** from the input.
- Thus, the only strings getting to the accepting state are those that end in **abb**.



TRANSITION TABLE

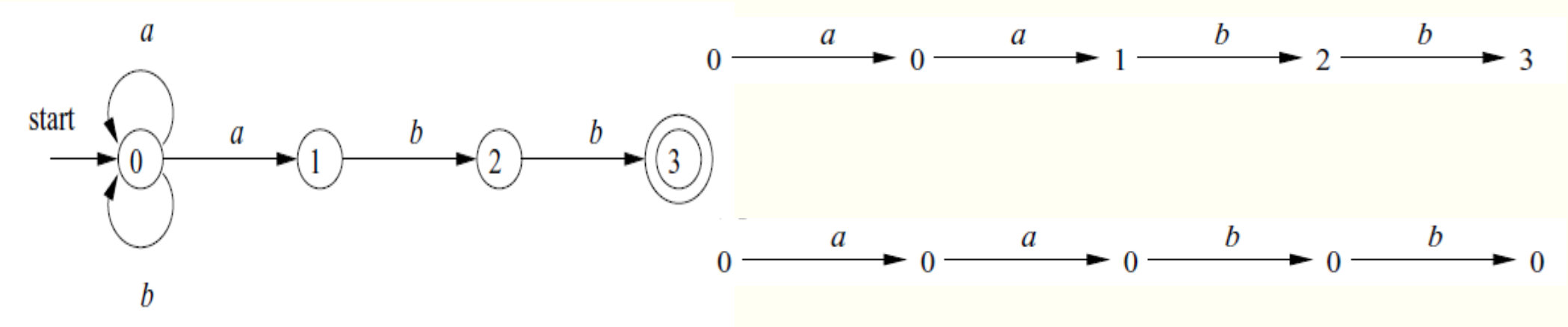
- Represent an NFA by a transition table, whose rows correspond to states, and whose columns correspond to the input symbols and ϵ.
- The entry for a given state and input is the value of the transition function applied to those arguments.
- If the transition function has no information about that state-input pair, we put Ø; in the table for the pair.



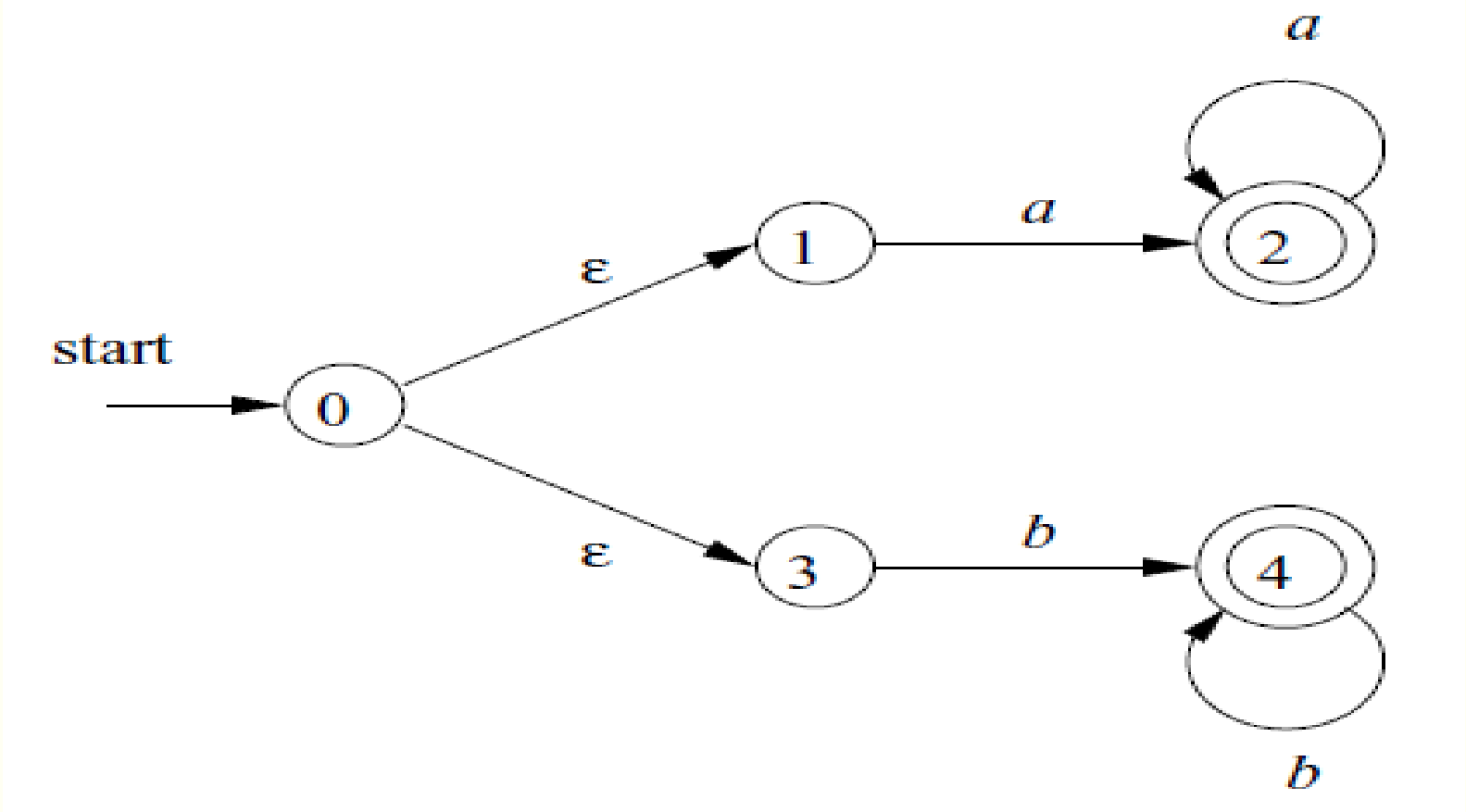
STATE	a	b	ϵ
0	$\{0, 1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

ACCEPTANCE OF INPUT STRINGS BY AUTOMATA

- An NFA accepts input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x.
- Note that ϵ labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.
- The string **aabb** is accepted by the NFA.



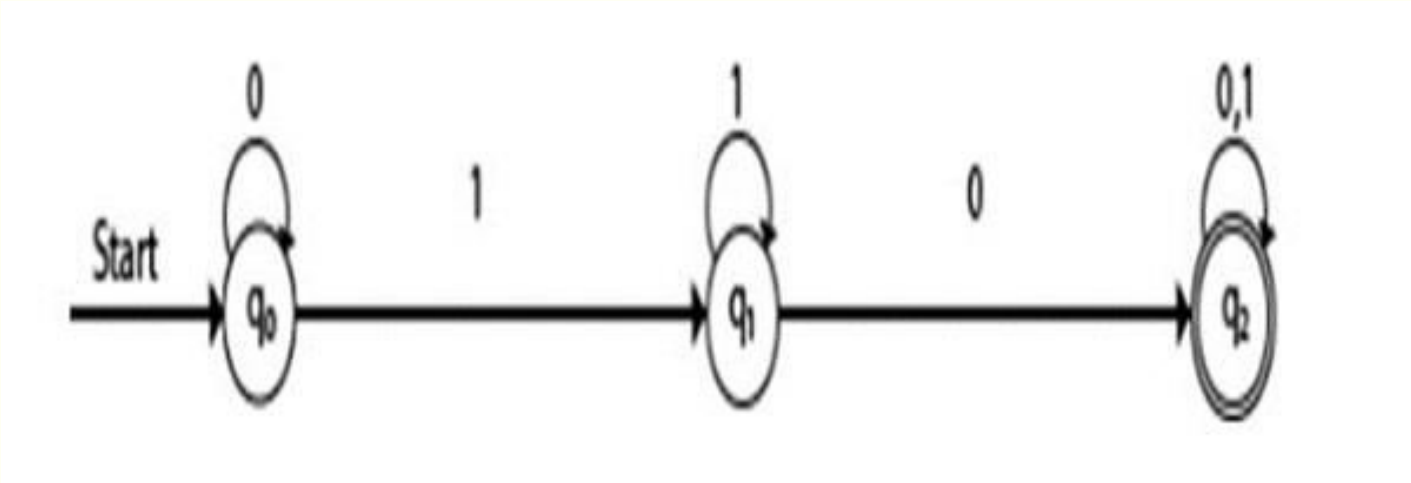
ACCEPTANCE OF INPUT STRINGS BY AUTOMATA



NFA accepting aa* | bb*

DETERMINISTIC FINITE AUTOMATA (DFA)

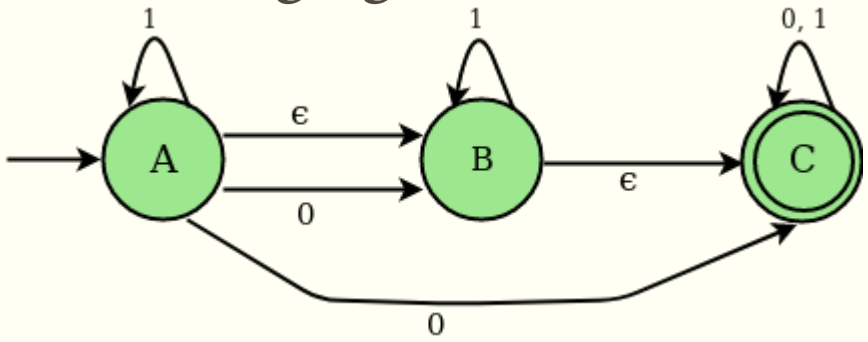
- A deterministic finite automaton (DFA) is a special case of an NFA where:
- S = {S, Σ, Σ ∪ {input}, s0, F}
  1. There are no moves on input ϵ, and
  2. For each state s and input symbol a, there is exactly one edge out of s labeled a.



FINITE AUTOMATA WITH EPSILON TRANSITIONS

- **NFA with (null) or ∈ move :** If any finite automata contains ε (null) move or transaction, then that finite automata is called NFA with ∈ moves

- **Example :**
  Consider the following figure of NFA with ∈ move :



- **Transition state table for the above NFA**

STATES	0	1	epsilon
A	B, C	A	B
B	–	B	C
C	C	C	–

FINITE AUTOMATA WITH EPSILON TRANSITIONS

- **Epsilon (∈) – closure:** Epsilon closure for a given state X is a set of states which can be reached from the states X with only (null) or ε moves including the state X itself. In other words, ε-closure for a state can be obtained by union operation of the ε-closure of the states which can be reached from X with a single ε move in recursive manner.
- **For the above example ∈ closure are as follows :**
- ∈ closure(A) : {A, B, C}
- ∈ closure(B) : {B, C}
- ∈ closure(C) : {C}

FINITE AUTOMATA WITH EPSILON TRANSITIONS

- **Steps to Convert NFA with ε-move to DFA :**

Step 1: Take ∈ closure for the beginning state of NFA as beginning state of DFA.

Step 2: Find the states that can be traversed from the present for each input symbol (union of transition value and their closures for each states of NFA present in current state of DFA).

Step 3: If any new state is found take it as current state and repeat step 2.

Step 4: Do repeat Step 2 and Step 3 until no new state present in DFA transition table.

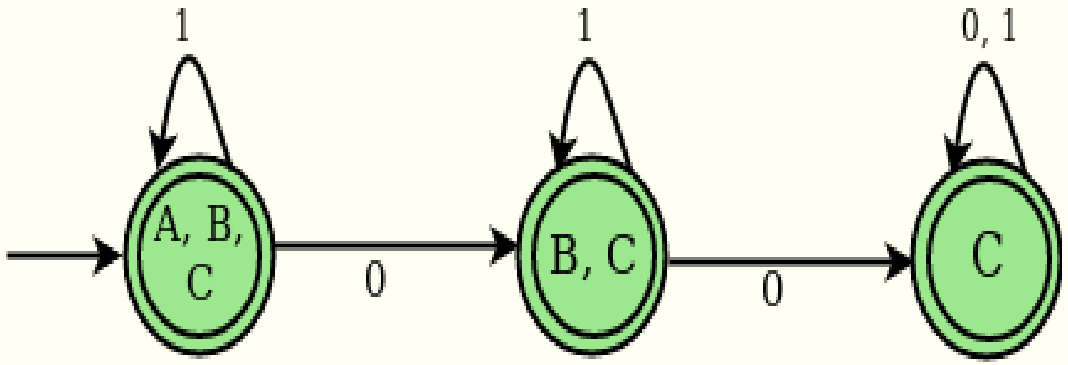
Step 5: Mark the states of DFA which contains final state of NFA as final states of DFA.

FINITE AUTOMATA WITH EPSILON TRANSITIONS

- Transition State Table for DFA corresponding to above NFA

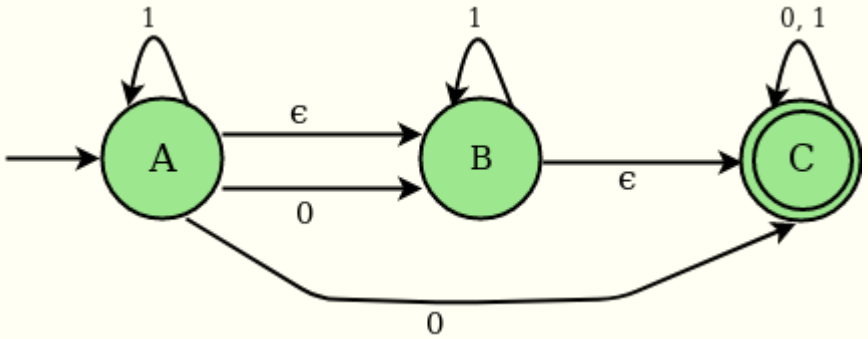
DFA

STATES	0	1
A, B, C	B, C	A, B, C
B, C	C	B, C
C	C	C



NFA

STATES	0	1	epsilon
A	B, C	A	B
B	–	B	C
C	C	C	–



NFA TO DFA CONVERSION

- Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M).
- There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').
- The following steps are followed to convert a given NFA to a DFA-

Step 1: Initially Q' = ϕ

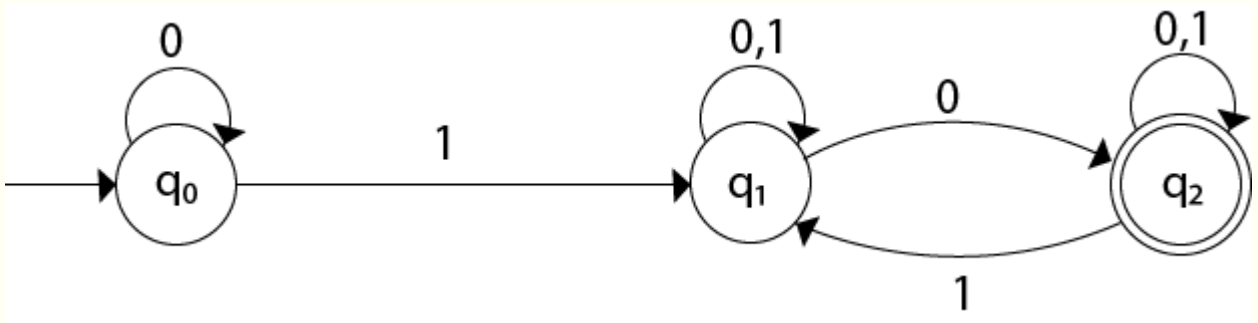
Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 4: In DFA, the final state will be all the states which contain F(final states of NFA)

NFA TO DFA CONVERSION - Example 1

- Convert the given NFA to DFA.



- For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

NFA TO DFA CONVERSION - Example 1

- Now we will obtain δ' transition for state q0.
  - δ'([q0], 0) = [q0]
  - δ'([q0], 1) = [q1]
- The δ' transition for state q1 is obtained as:
  - δ'([q1], 0) = [q1, q2] (new state generated)
  - δ'([q1], 1) = [q1]
- The δ' transition for state q2 is obtained as:
  - δ'([q2], 0) = [q2]
  - δ'([q2], 1) = [q1, q2]

NFA TO DFA CONVERSION - Example 1

- Now we will obtain δ' transition on [q1, q2].

$$\delta'([q1, q2], 0) = \delta(q1, 0) \cup \delta(q2, 0)$$
$$= \{q1, q2\} \cup \{q2\}$$
$$= [q1, q2]$$
$$\delta'([q1, q2], 1) = \delta(q1, 1) \cup \delta(q2, 1)$$
$$= \{q1\} \cup \{q1, q2\}$$
$$= \{q1, q2\}$$
$$= [q1, q2]$$

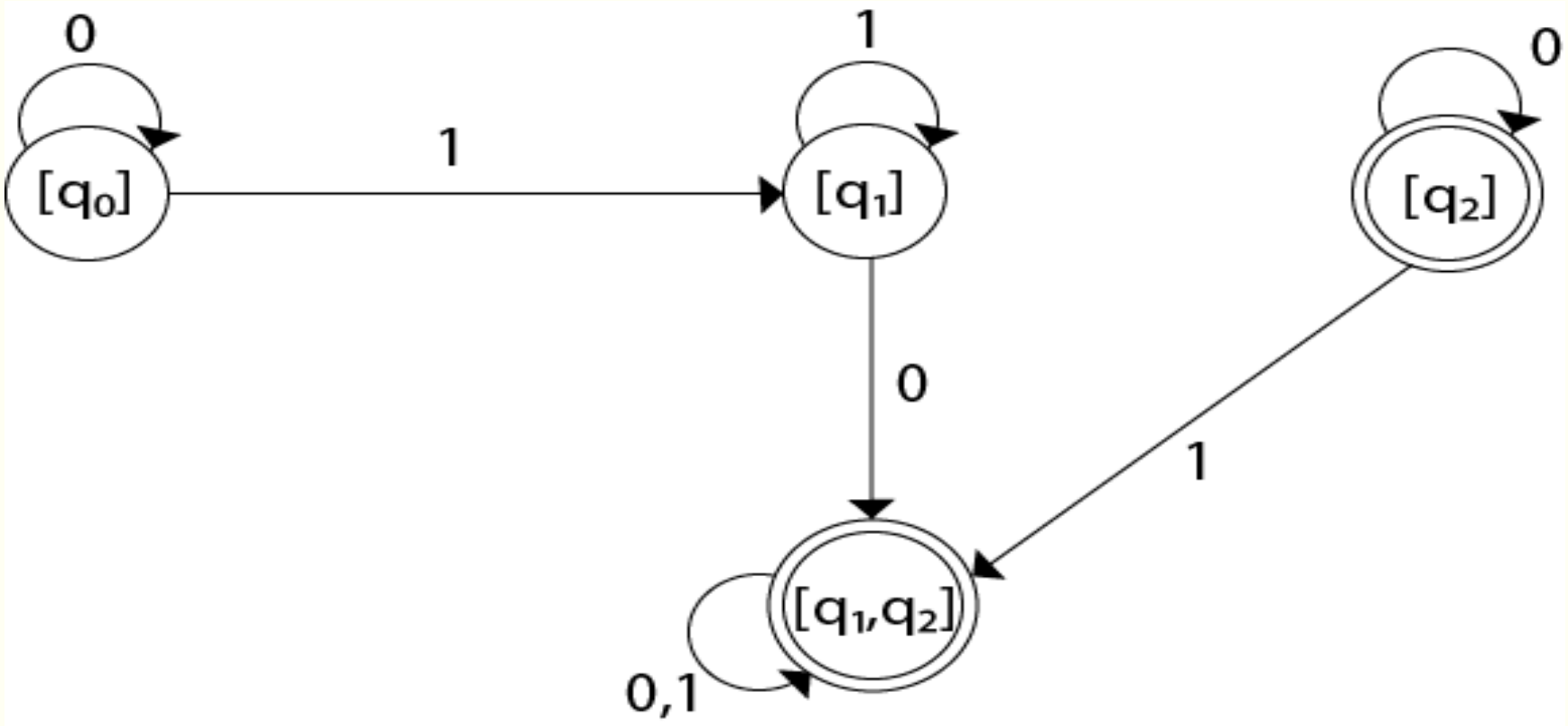
NFA TO DFA CONVERSION - Example 1

- The state [q1, q2] is the final state as well because it contains a final state q2.
- The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

NFA TO DFA CONVERSION - Example 1

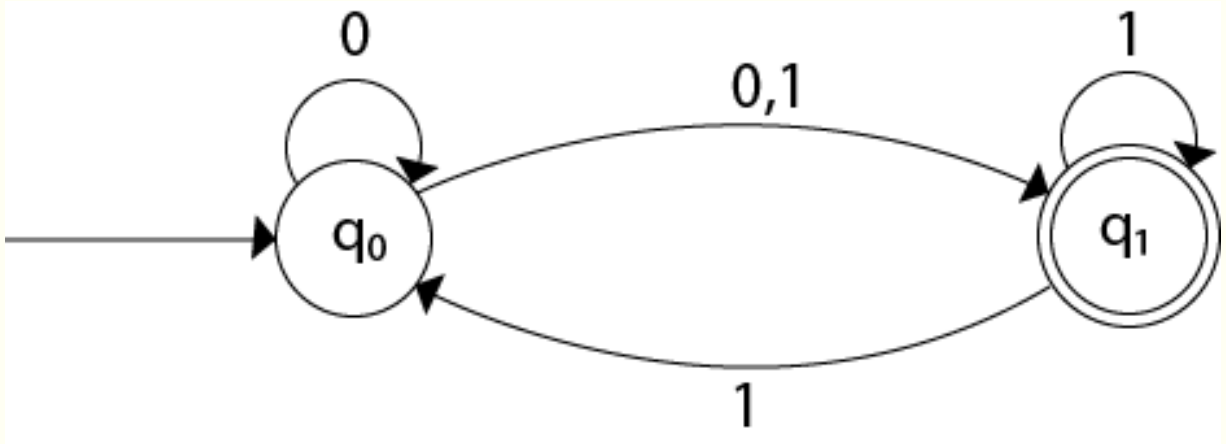
- The Transition diagram will be:



- The state q2 can be eliminated because q2 is an unreachable state.

NFA TO DFA CONVERSION - Example 2

- Convert the given NFA to DFA.



- For the given transition diagram we will first construct the transition table.

State	0	1
→q0	{q0, q1}	{q1}
*q1	ϕ	{q0, q1}

NFA TO DFA CONVERSION - Example 2

- Now we will obtain δ' transition for state q0.

  $\delta'([q0], 0) = \{q0, q1\}$

  $= [q0, q1]$ (new state generated)

  $\delta'([q0], 1) = \{q1\} = [q1]$

- The δ' transition for state q1 is obtained as:

  $\delta'([q1], 0) = \phi$

  $\delta'([q1], 1) = [q0, q1]$

- Now we will obtain δ' transition on [q0, q1].

  $\delta'([q0, q1], 0) = \delta(q0, 0) \cup \delta(q1, 0)$

  $= \{q0, q1\} \cup \phi$

  $= \{q0, q1\}$

  $= [q0, q1]$

NFA TO DFA CONVERSION - Example 2

- Similarly,

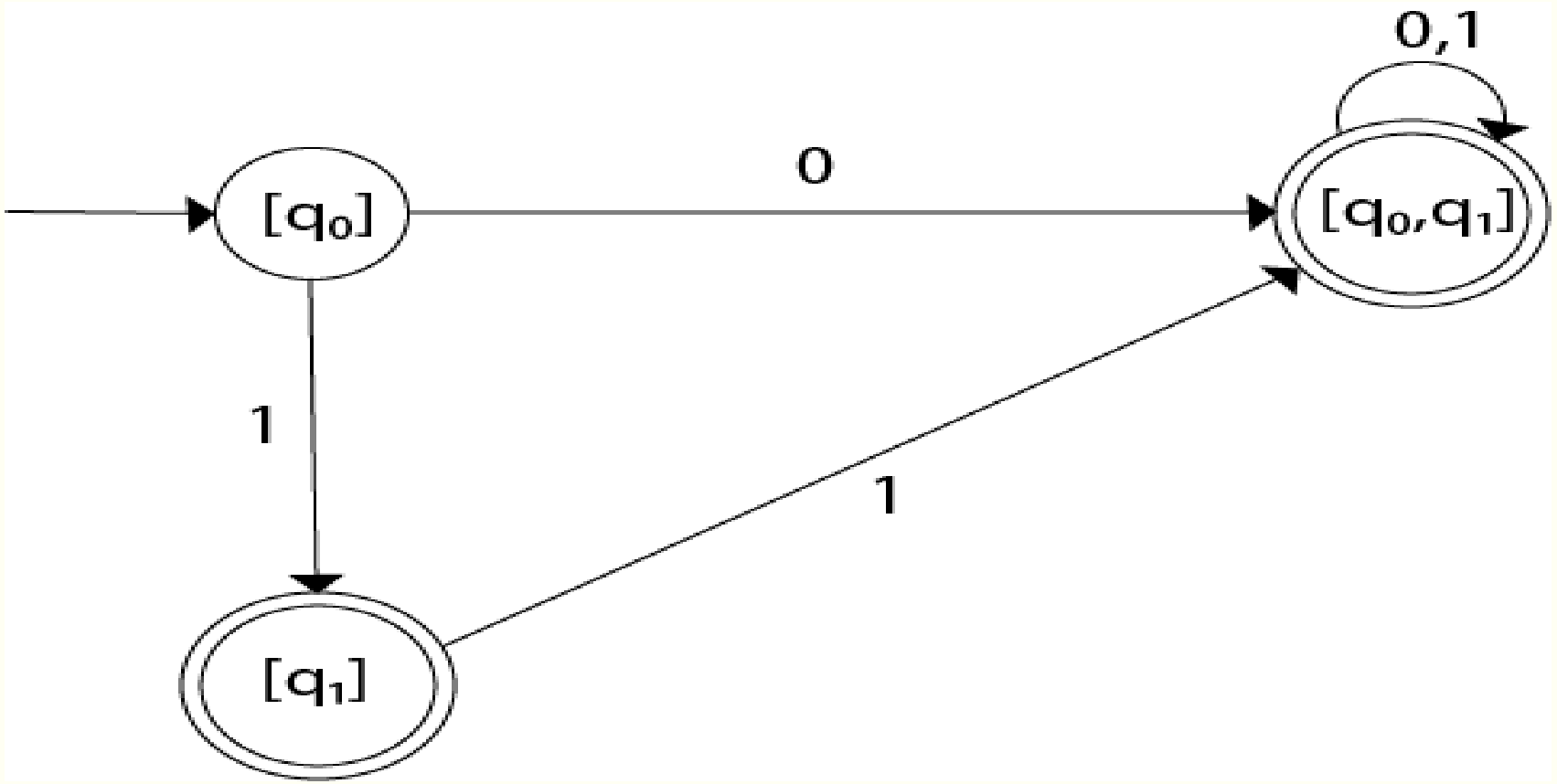
$$\delta'([q0, q1], 1) = \delta(q0, 1) \cup \delta(q1, 1)$$
$$= \{q1\} \cup \{q0, q1\}$$
$$= \{q0, q1\}$$
$$= [q0, q1]$$

- As in the given NFA, q1 is a final state, then in DFA wherever, q1 exists that state becomes a final state. Hence in the DFA, final states are [q1] and [q0, q1]. Therefore set of final states F = {[q1], [q0, q1]}.
- The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0, q1]	[q1]
*[q1]	ϕ	[q0, q1]
*[q0, q1]	[q0, q1]	[q0, q1]

NFA TO DFA CONVERSION - Example 2

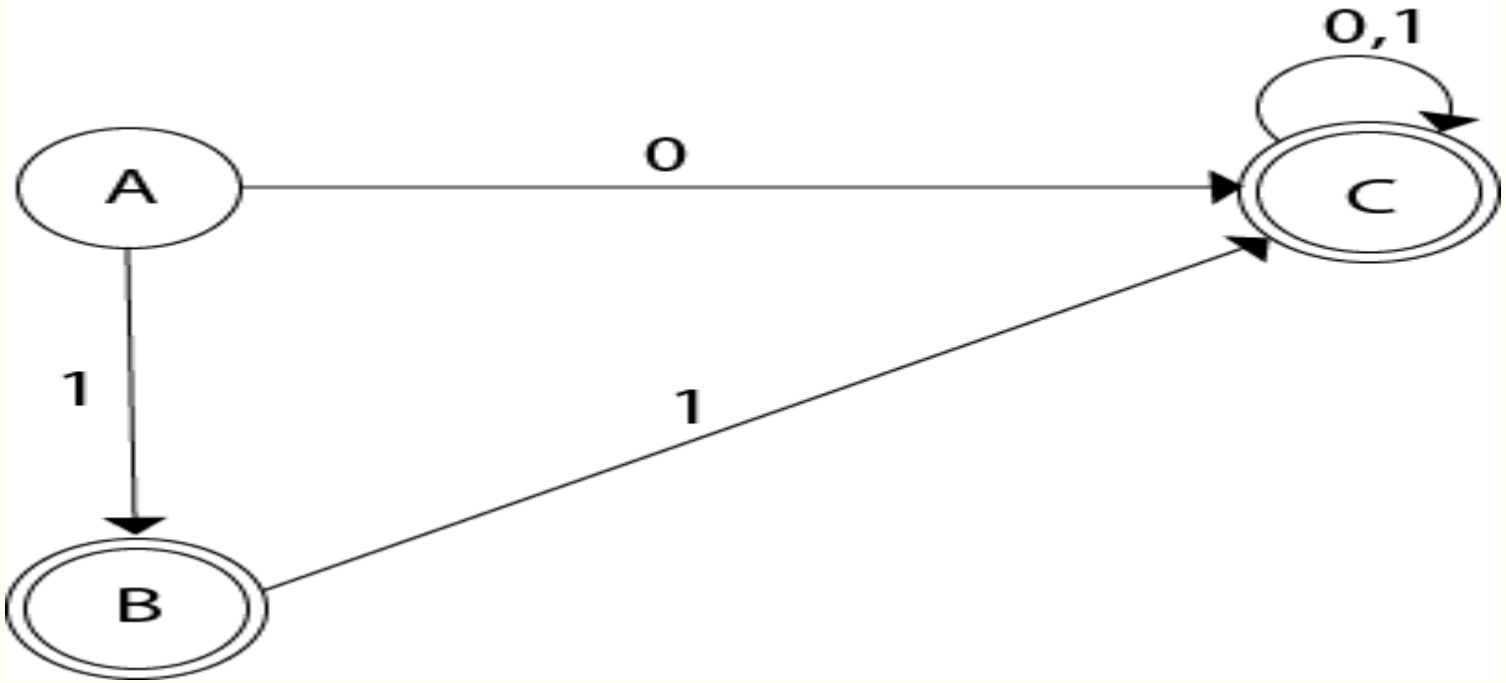
- The Transition diagram will be:



- Even we can change the name of the states of DFA.

NFA TO DFA CONVERSION - Example 2

- A = [q0]
- B = [q1]
- C = [q0, q1]
- With these new names the DFA will be as follows:



NFA TO DFA CONVERSION - Example 3

- Following are the various parameters for NFA.
- Q = { q0, q1, q2 }
- ∑ = (a, b)
- F = { q2 }
- δ (Transition Function of NFA)

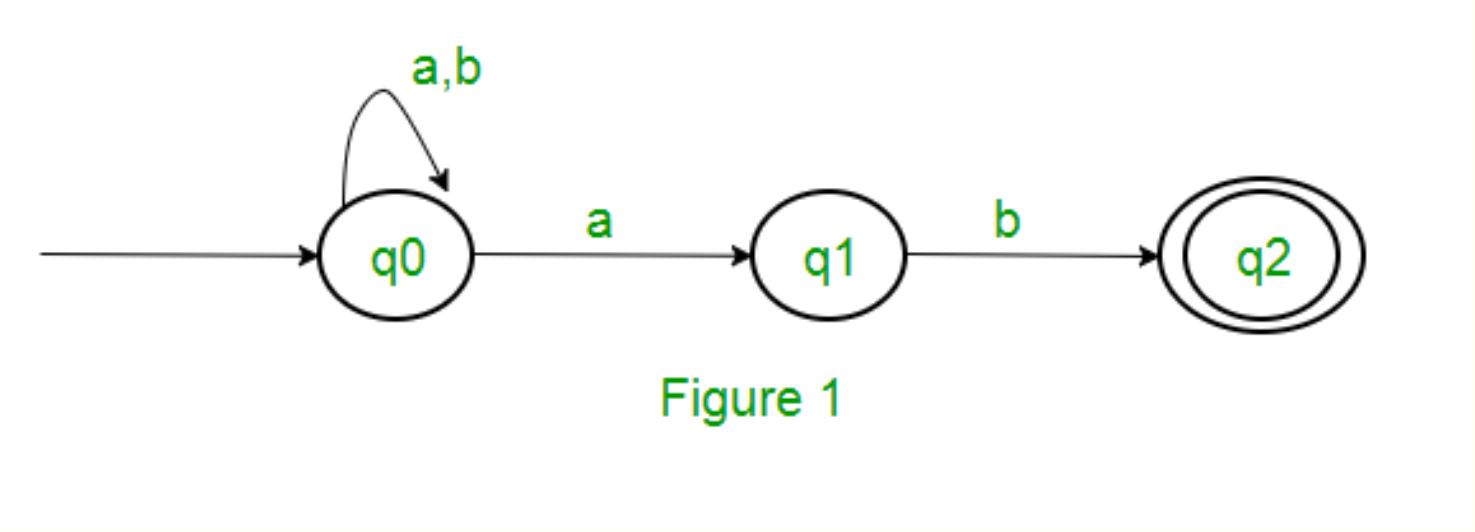


Figure 1

NFA TO DFA CONVERSION - Example 3

- Step 1: Q’ = ϕ
- Step 2: Q’ = {q0}
- Step 3: For each state in Q’, find the states for each input symbol.
- Currently, state in Q’ is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA.
- δ’ (Transition Function of DFA)

State	a	b
q0	q0,q1	q0
q1		q2
q2		

NFA TO DFA CONVERSION - Example 3

- Now { q0, q1 } will be considered as a single state. As its entry is not in Q', add it to Q'.
- So Q' = { q0, { q0, q1 } }

State	a	b
q0	{q0,q1}	q0

NFA TO DFA CONVERSION - Example 3

- Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like:
- δ' ({ q0, q1 }, a) = δ (q0, a) $\cup$ δ (q1, a) = { q0, q1 }
- δ' ({ q0, q1 }, b) = δ (q0, b) $\cup$ δ (q1, b) = { q0, q2 }
- Now we will update the transition table of DFA.
- δ' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

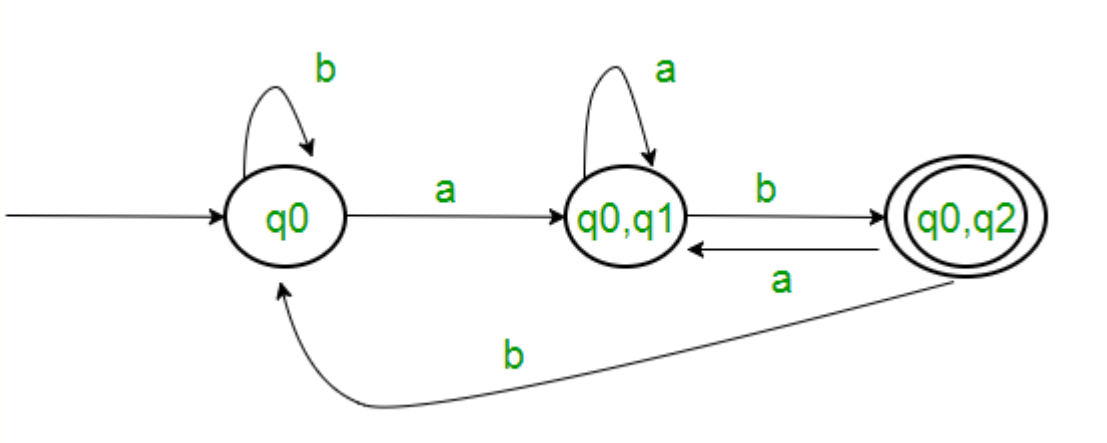
NFA TO DFA CONVERSION - Example 3

- Now { q0, q2 } will be considered as a single state. As its entry is not in Q', add it to Q'.
- So Q' = { q0, { q0, q1 }, { q0, q2 } }
- Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like:
- δ' ({ q0, q2 }, a) = δ (q0, a) $\cup$ δ (q2, a) = { q0, q1 }
- δ' ({ q0, q2 }, b) = δ (q0, b) $\cup$ δ (q2, b) = { q0 }
- Now we will update the transition table of DFA.
- δ' (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

NFA TO DFA CONVERSION - Example 3

- As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., { q0, q2 }
- Following are the various parameters for DFA.
- Q' = { q0, { q0, q1 }, { q0, q2 } }
- ∑ = (a, b)
- F = { { q0, q2 } } and transition function δ' as shown above. The final DFA for above NFA has been shown in Figure.

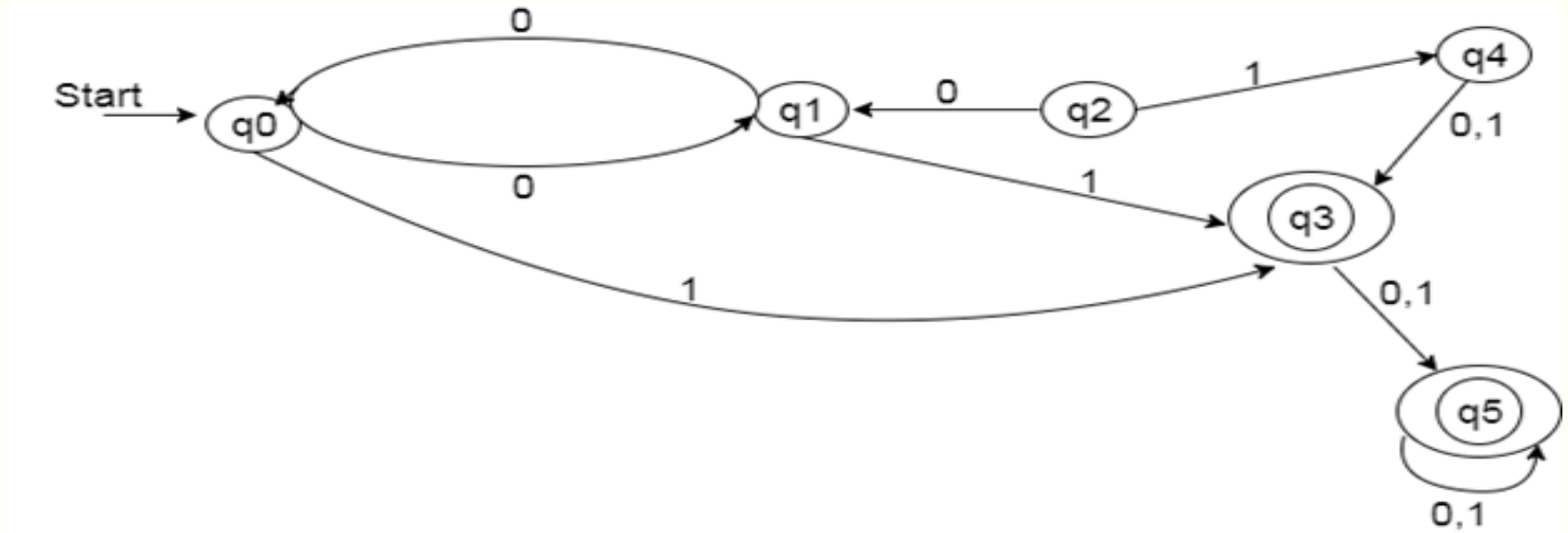


MINIMIZATION OF AUTOMATA

- Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.
- Step 2: Draw the transition table for all pair of states.
- Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states and T2 contains non-final states.
- Step 4: Find the similar rows from T1 such that:
  - δ (q, a) = p
  - δ (r, a) = p
- That means, find the two states which have same value of a and b and remove one of them.

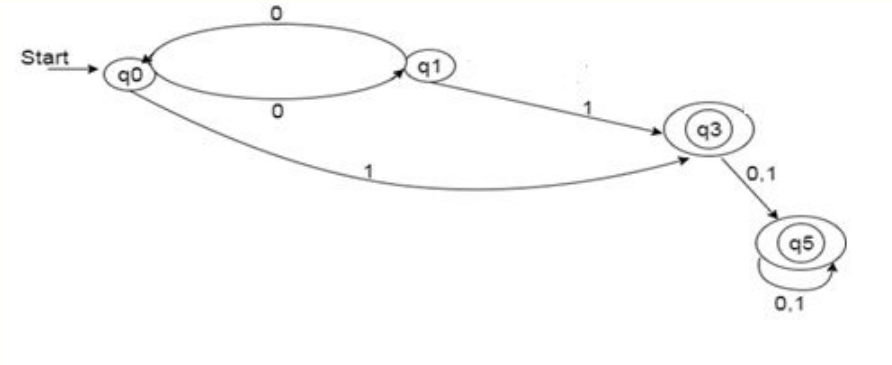
MINIMIZATION OF AUTOMATA

- Step 5: Repeat step 3 until there is no similar rows are available in the transition table T1.
- Step 6: Repeat step 3 and step 4 for table T2 also.
- Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.



MINIMIZATION OF AUTOMATA

- Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.
- Step 2: Draw the transition table for rest of the states.



State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

MINIMIZATION OF AUTOMATA

- Step 3:
- Now divide rows of transition table into two sets as:
- 1. One set contains those rows, which start from non-final sates:

State	0	1
q0	q1	q3
q1	q0	q3

MINIMIZATION OF AUTOMATA

- 2. Other set contains those rows, which starts from final states.

State	0	1
q3	q5	q5
q5	q5	q5

MINIMIZATION OF AUTOMATA

- Step 4: Set 1 has no similar rows so set 1 will be the same.
- Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

MINIMIZATION OF AUTOMATA

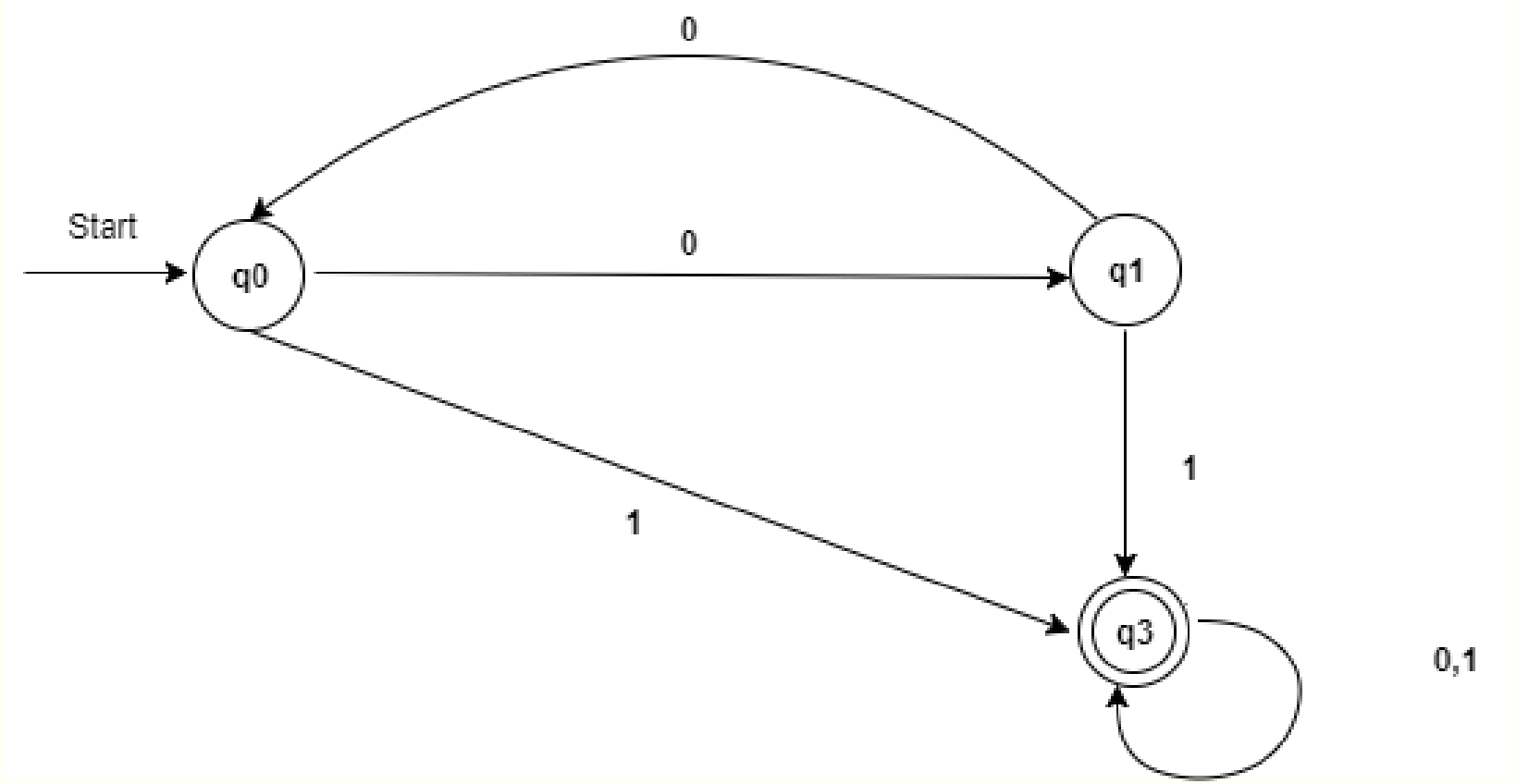
- Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

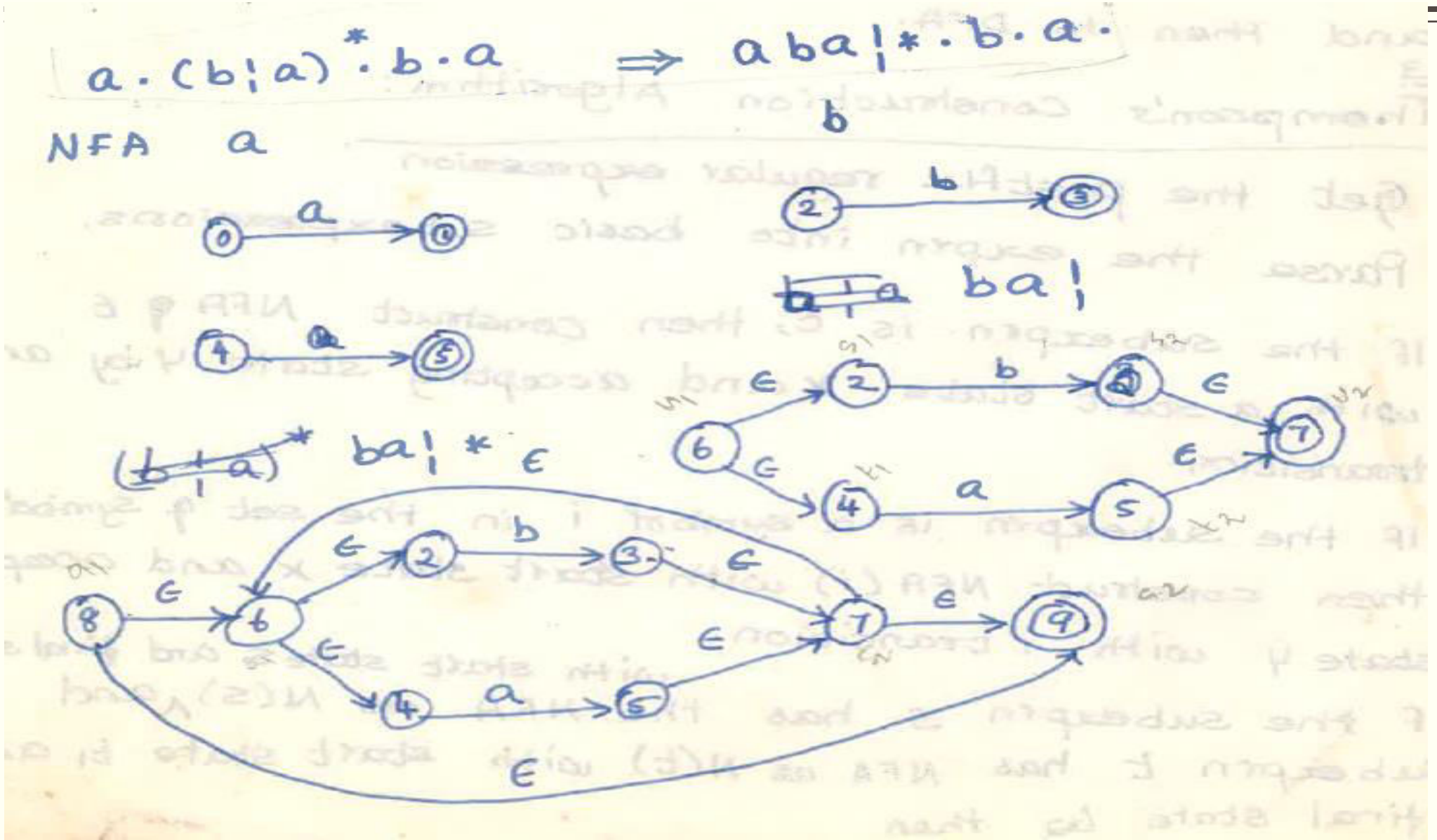
- Now it is the transition table of minimized DFA.

MINIMIZATION OF AUTOMATA

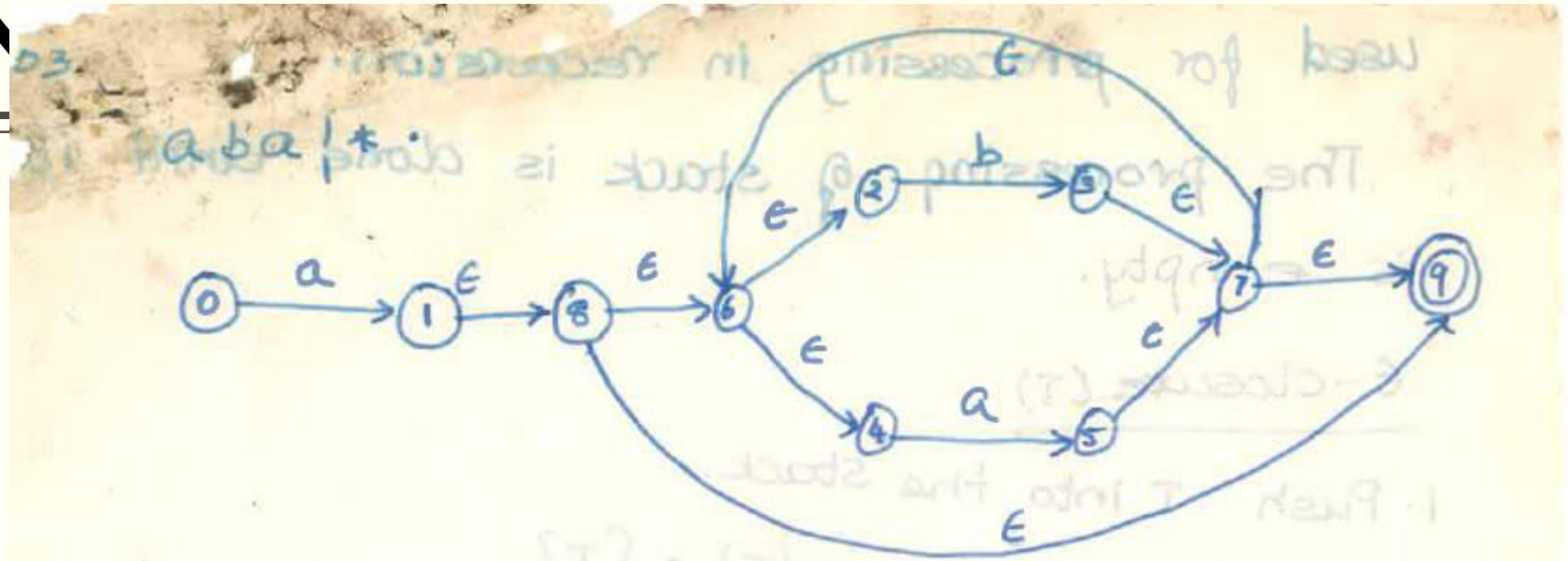
- Transition diagram of minimized DFA:

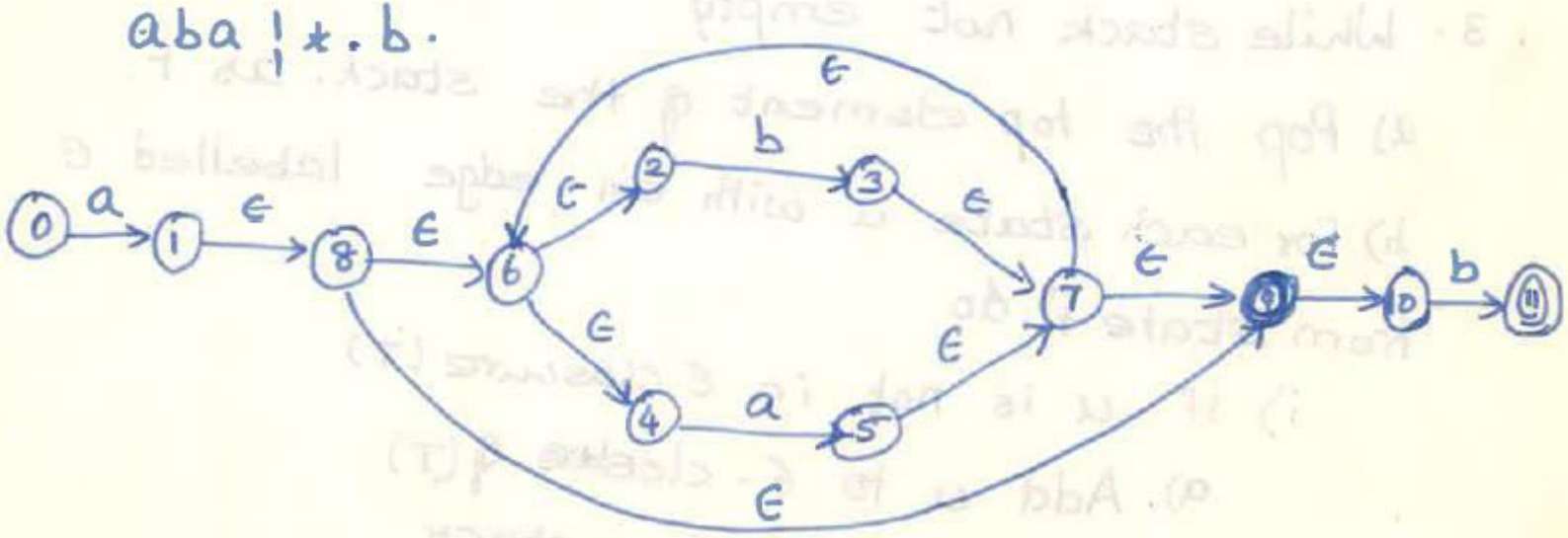


CONVERSION OF RE TO FA

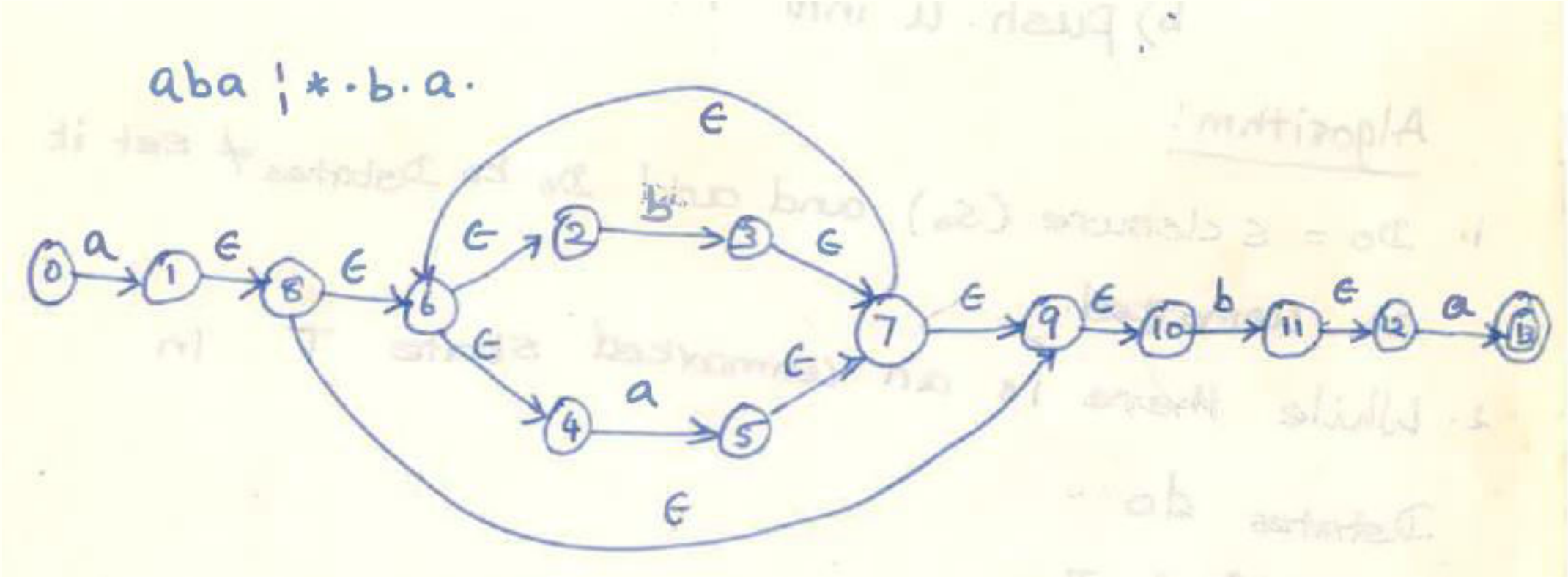


CON


ab a ! *.
0
a
1
ε
8
ε
6
ε
2
b
3
ε
7
ε
9
ε
ε
4
a
5
c
ε


aba ! *. b.
0
a
1
ε
8
ε
6
ε
2
b
3
ε
7
ε
ε
ε
4
a
5
ε
ε
ε
b

CONVERSION OF RE TO FA

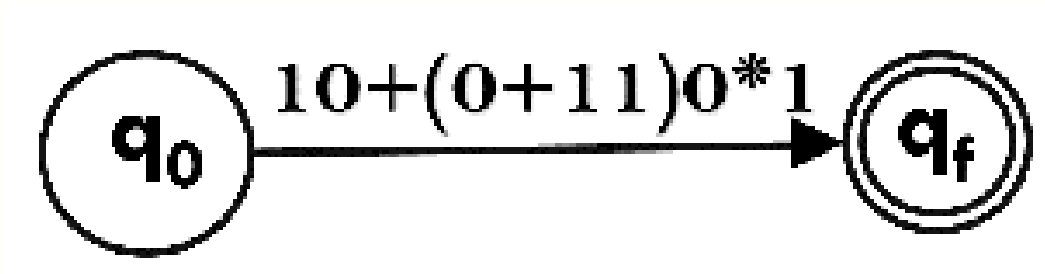


CONVERSION OF RE TO FA

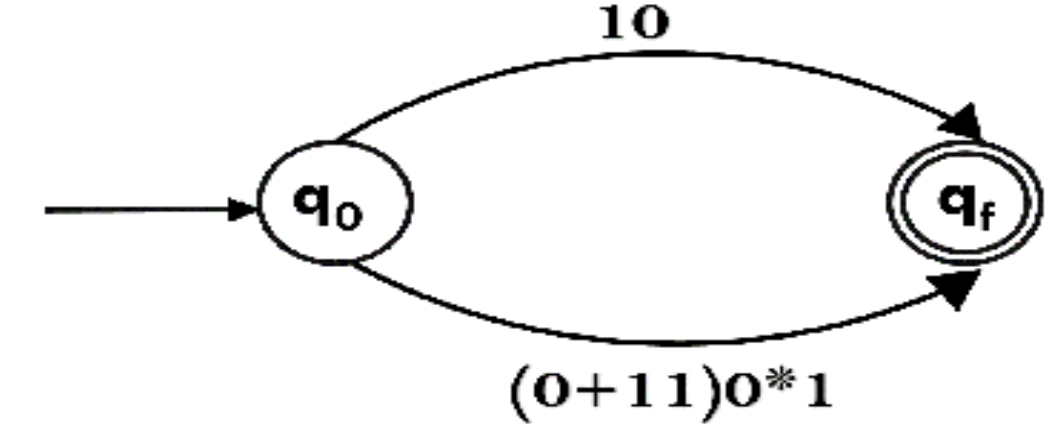
- To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:
- Step 1: Design a transition diagram for given regular expression, using NFA with ε moves.
- Step 2: Convert this NFA with ε to NFA without ε.
- Step 3: Convert the obtained NFA to equivalent DFA.
- Example 1:
- Design a FA from given regular expression 10 + (0 + 11)0* 1.

CONVERSION OF RE TO FA – Example 1

- Solution: First we will construct the transition diagram for a given regular expression.
- Step 1:

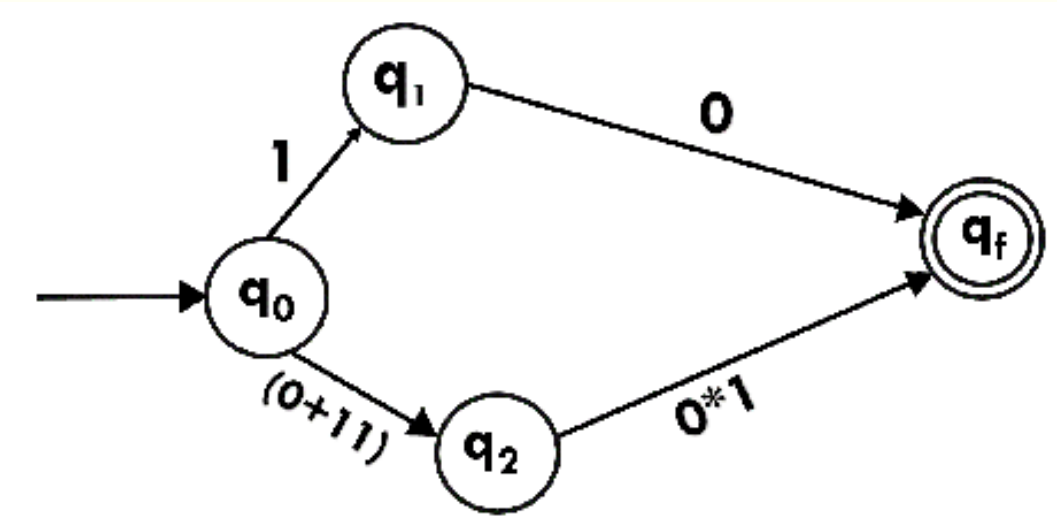


- Step 2:

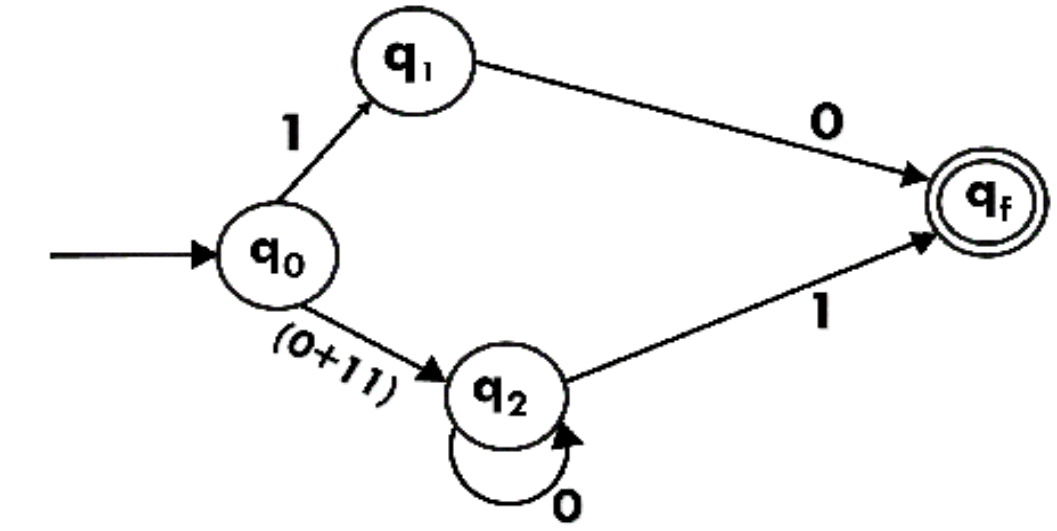


CONVERSION OF RE TO FA – Example 1

- Step 3:

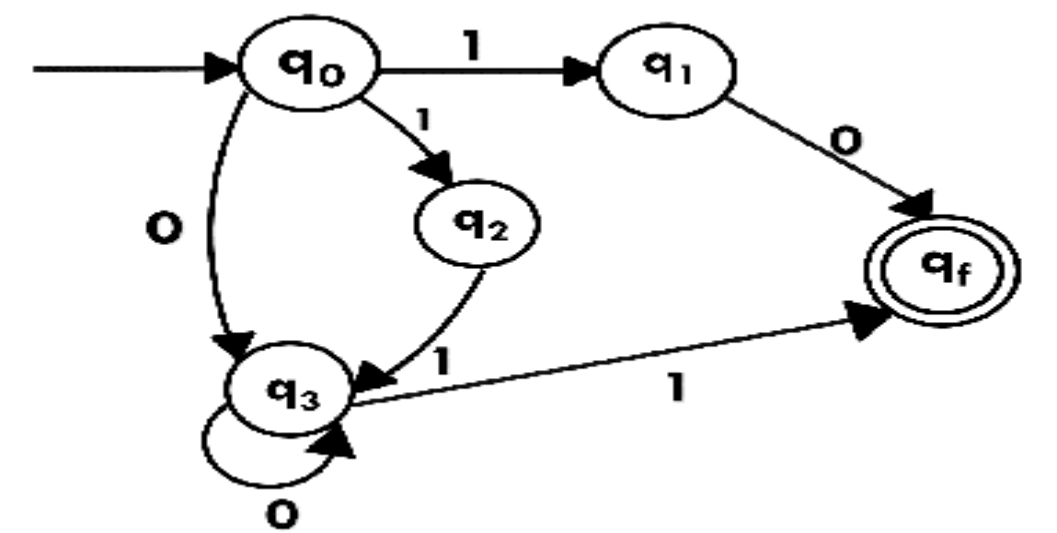


- Step 4:



CONVERSION OF RE TO FA – Example 1

- Step 5:



- Now we have got NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

CONVERSION OF RE TO FA – Example 1

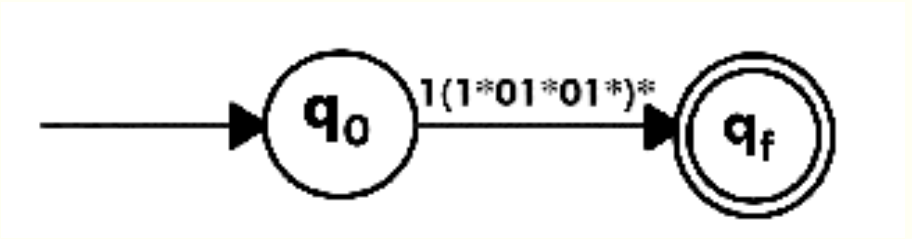
State	0	1
→q0	q3	{q1, q2}
q1	qf	ϕ
q2	ϕ	q3
q3	q3	qf
*qf	ϕ	ϕ

- The equivalent DFA will be:

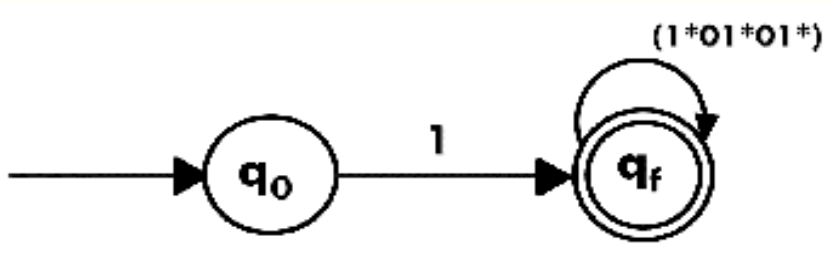
State	0	1
→[q0]	[q3]	[q1, q2]
[q1]	[qf]	ϕ
[q2]	ϕ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
*[qf]	ϕ	ϕ

CONVERSION OF RE TO FA – Example 2

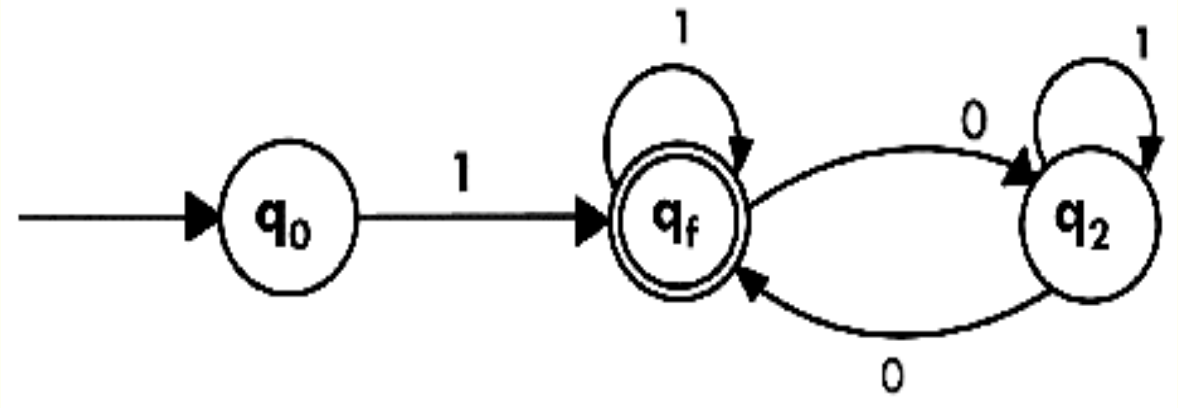
- Design a NFA from given regular expression 1 (1* 01* 01*)*.
- Solution: The NFA for the given regular expression is as follows:
- Step 1:



Step 2:

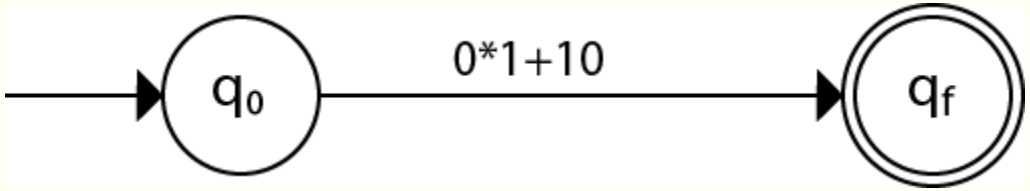


- Step 3:

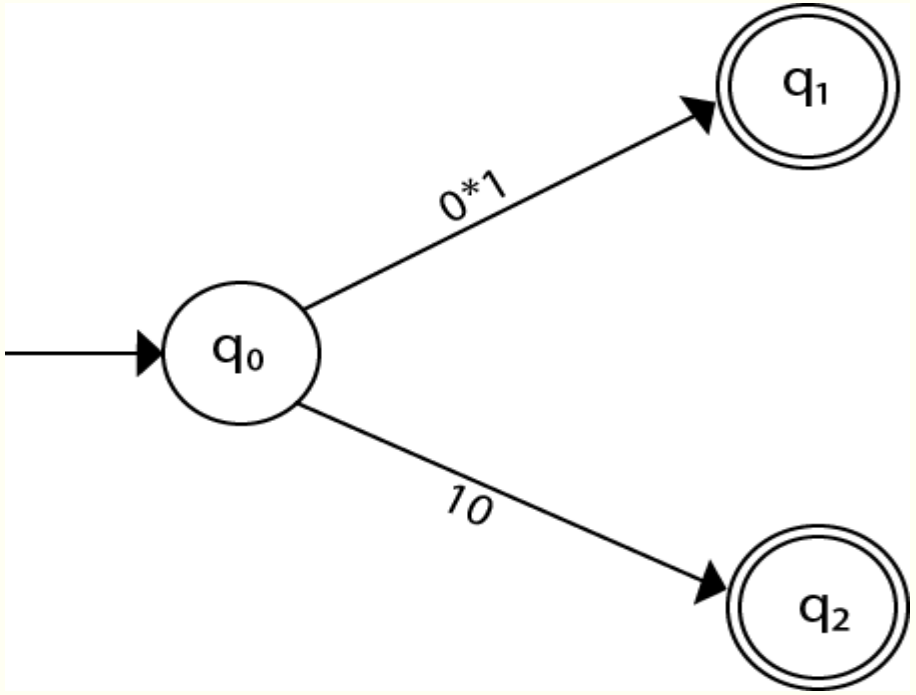


CONVERSION OF RE TO FA – Example 3

- Construct the FA for regular expression 0*1 + 10.
- Solution:
- We will first construct FA for R = 0*1 + 10 as follows:
- Step 1:

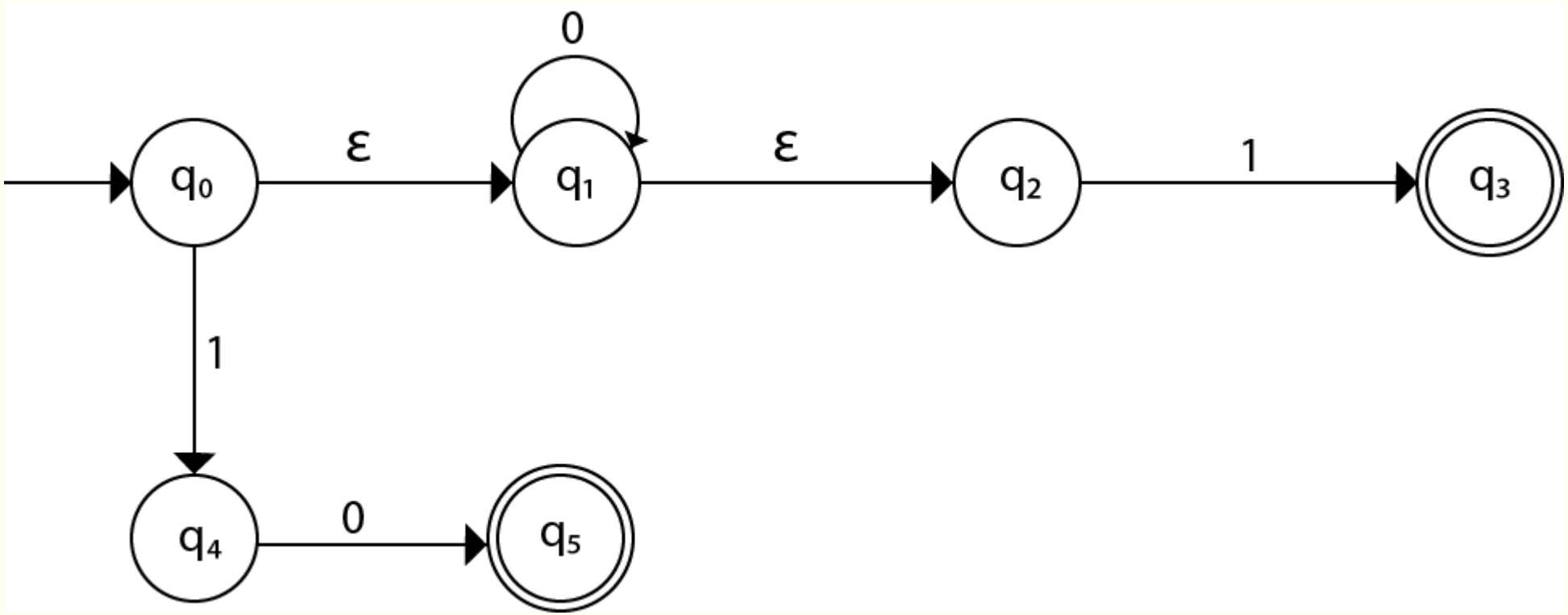


Step 2:



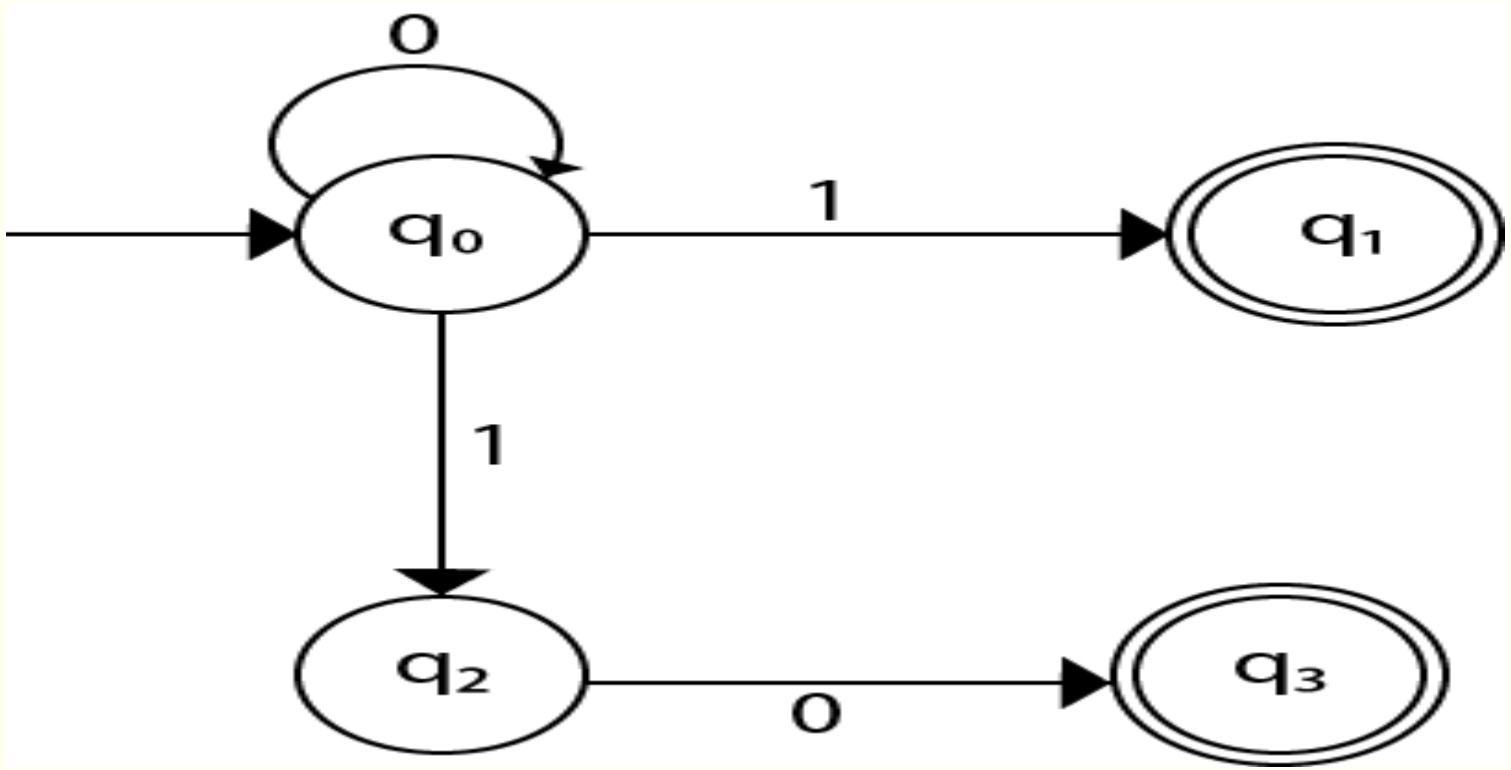
CONVERSION OF RE TO FA – Example 3

- Step 3:



CONVERSION OF RE TO FA – Example 3

- Step 4:



Conversion from NFA with ε to DFA

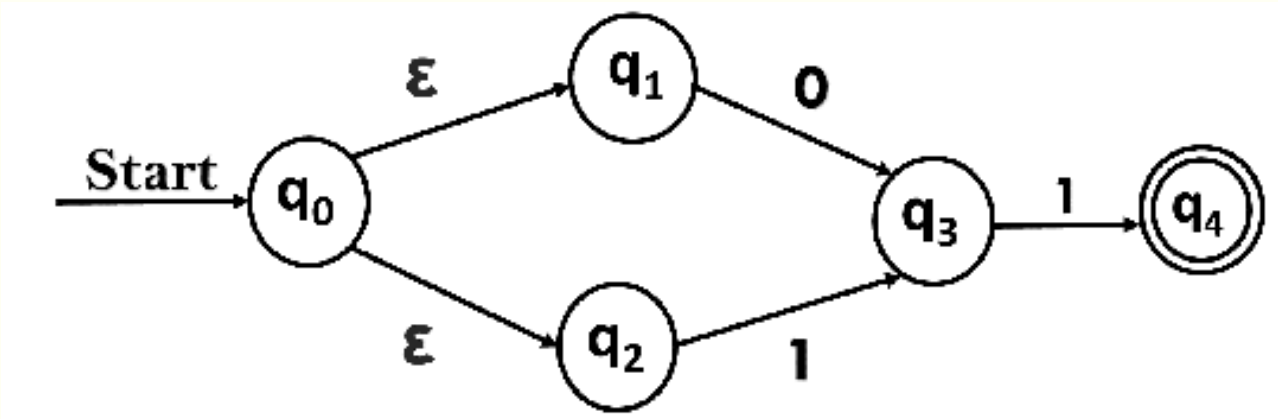
- Non-deterministic finite automata(NFA) is a finite automata where for some cases when a specific input is given to the current state, the machine goes to multiple states or more than 1 states. It can contain ε move. It can be represented as M = { Q, ∑, δ, q0, F}.
- Where
  - Q: finite set of states
  - ∑: finite set of the input symbol
  - q0: initial state
  - F: final state
  - δ: Transition function
- NFA with ∈ move: If any FA contains ε transaction or move, the finite automata is called NFA with ∈ move.
- ε-closure: ε-closure for a given state A means a set of states which can be reached from the state A with only ε(null) move including the state A itself.

Conversion from NFA with ε to DFA

- **Steps for converting NFA with ε to DFA:**
- Step 1: We will take the ε-closure for the starting state of NFA as a starting state of DFA.
- Step 2: Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.
- Step 3: If we found a new state, take it as current state and repeat step 2.
- Step 4: Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.
- Step 5: Mark the states of DFA as a final state which contains the final state of NFA.

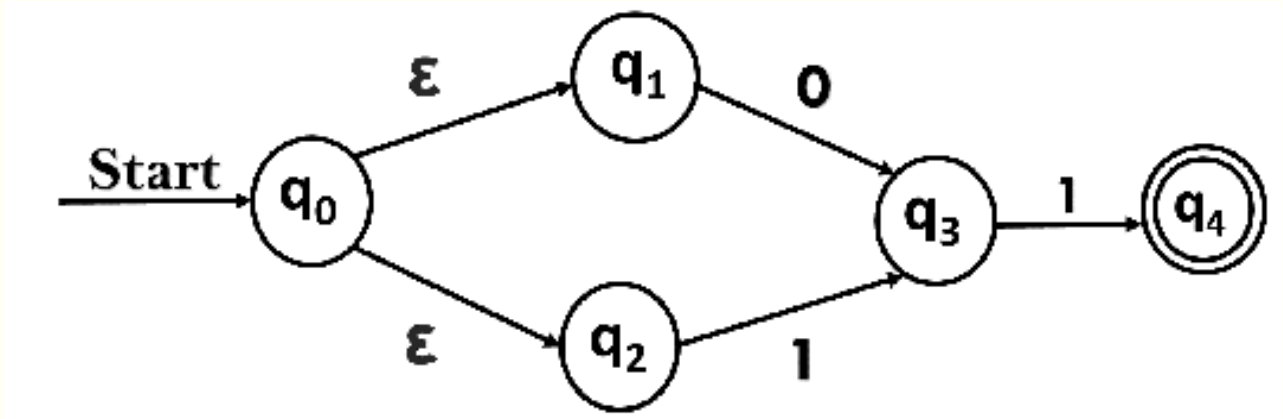
Conversion from NFA with ε to DFA – Example 1

- Convert the NFA with ε into its equivalent DFA.



Conversion from NFA with ε to DFA – Example 1

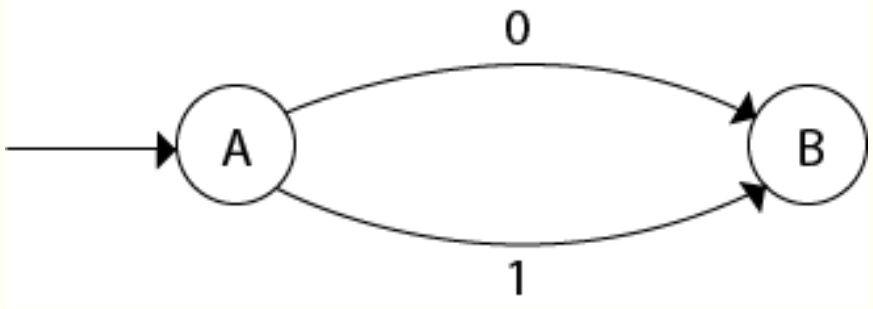
- Solution:



- Let us obtain ε-closure of each state.
- ε-closure {q0} = {q0, q1, q2}
- ε-closure {q1} = {q1}
- ε-closure {q2} = {q2}
- ε-closure {q3} = {q3}
- ε-closure {q4} = {q4}

Conversion from NFA with ε to DFA – Example 1

- The partial DFA will be



- Now,

$\delta'(B, 0) = \varepsilon\text{-closure}\{\delta(q3, 0)\}$

$= \phi\ \delta'(B, 1)$

$= \varepsilon\text{-closure}\{\delta(q3, 1)\}$

$= \varepsilon\text{-closure}\{q4\}$

$= \{q4\}$ i.e. state C

Conversion from NFA with ε to DFA – Example 1

- For state C:

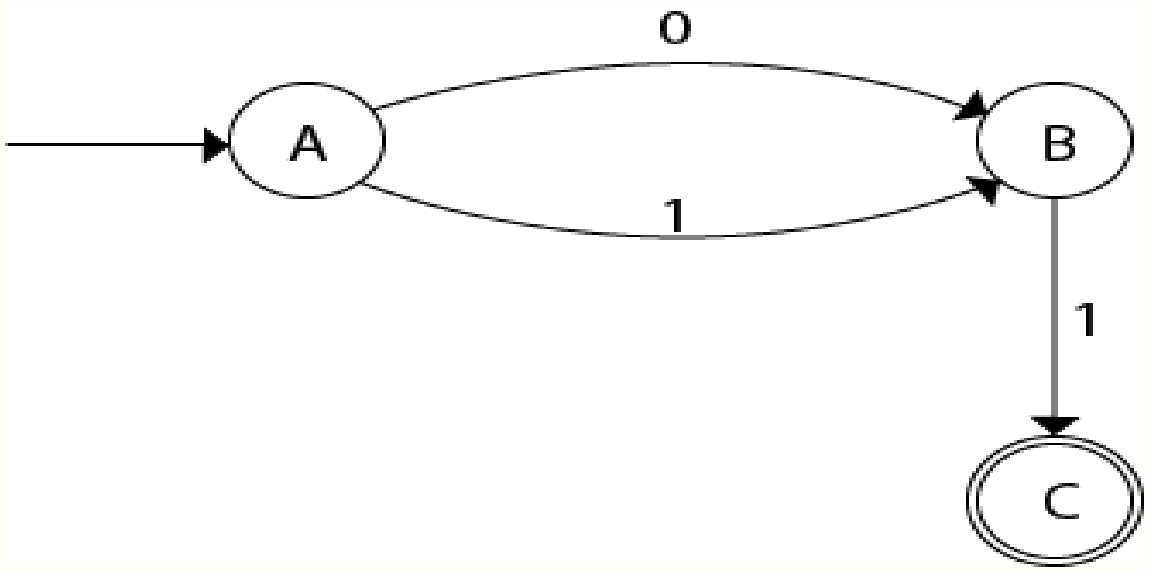
$\delta'(C, 0) = \varepsilon\text{-closure } \{\delta(q4, 0)\}$

$= \phi$

$\delta'(C, 1) = \varepsilon\text{-closure } \{\delta(q4, 1)\}$

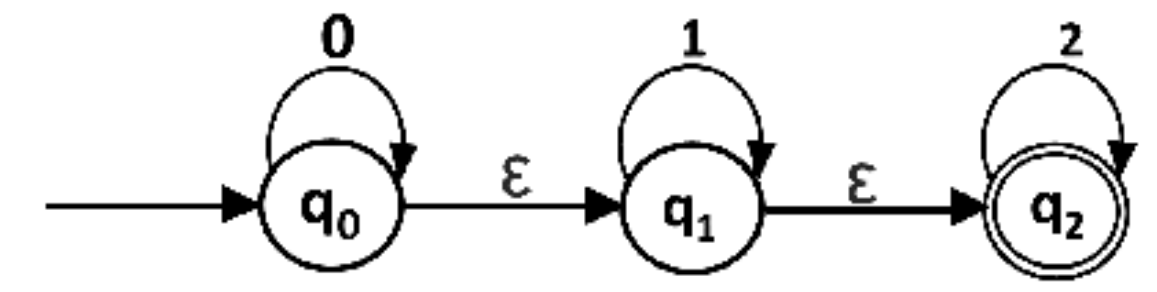
$= \phi$

- The DFA will be,



Conversion from NFA with ε to DFA – Example 2

- Convert the given NFA into its equivalent DFA.



- Solution: Let us obtain the ε-closure of each state.

- ε-closure(q0) = {q0, q1, q2}
- ε-closure(q1) = {q1, q2}
- ε-closure(q2) = {q2}

Conversion from NFA with ε to DFA – Example 2

- Now we will obtain δ' transition.
- Let ε-closure(q0) = {q0, q1, q2} call it as state A.

$\delta'(A, 0) = \varepsilon\text{-closure}\{\delta((q0, q1, q2), 0)\}$

$= \varepsilon\text{-closure}\{\delta(q0, 0) \cup \delta(q1, 0) \cup \delta(q2, 0)\}$

$= \varepsilon\text{-closure}\{q0\}$

$= \{q0, q1, q2\}$

$\delta'(A, 1) = \varepsilon\text{-closure}\{\delta((q0, q1, q2), 1)\}$

$= \varepsilon\text{-closure}\{\delta(q0, 1) \cup \delta(q1, 1) \cup \delta(q2, 1)\}$

$= \varepsilon\text{-closure}\{q1\}$

$= \{q1, q2\}$ call it as state B

Conversion from NFA with ε to DFA – Example 2

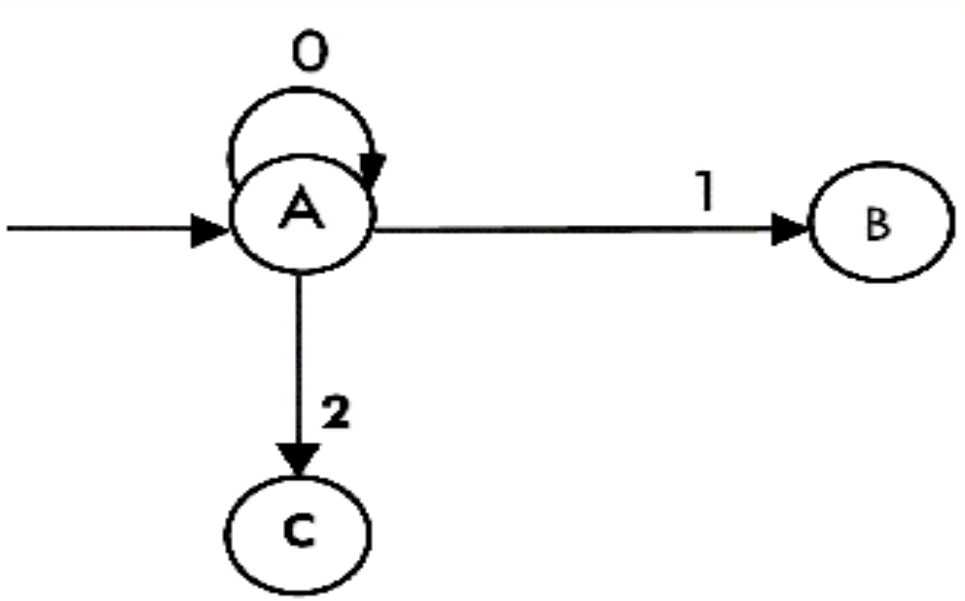
$\delta'(A, 2) = \varepsilon\text{-closure}\{\delta((q0, q1, q2), 2)\}$

$= \varepsilon\text{-closure}\{\delta(q0, 2) \cup \delta(q1, 2) \cup \delta(q2, 2)\}$

$= \varepsilon\text{-closure}\{q2\}$

$= \{q2\}$ call it state C

- Thus we have obtained
- $\delta'(A, 0) = A$
- $\delta'(A, 1) = B$
- $\delta'(A, 2) = C$
- The partial DFA will be:



Conversion from NFA with ε to DFA – Example 2

- Now we will find the transitions on states B and C for each input.
- Hence

$$\delta'(B, 0) = \varepsilon\text{-closure}\{\delta((q1, q2), 0)\}$$
$$= \varepsilon\text{-closure}\{\delta(q1, 0) \cup \delta(q2, 0)\}$$
$$= \varepsilon\text{-closure}\{\phi\}$$
$$= \phi$$

$$\delta'(B, 1) = \varepsilon\text{-closure}\{\delta((q1, q2), 1)\}$$
$$= \varepsilon\text{-closure}\{\delta(q1, 1) \cup \delta(q2, 1)\}$$
$$= \varepsilon\text{-closure}\{q1\}$$
$$= \{q1, q2\} \qquad \text{i.e. state B itself}$$

Conversion from NFA with ε to DFA – Example 2

- Now we will obtain transitions for C:

$\delta'(C, 0) = \varepsilon\text{-closure}\{\delta(q2, 0)\}$

$= \varepsilon\text{-closure}\{\phi\}$

$= \phi$ $\delta'(C, 1)$

$= \varepsilon\text{-closure}\{\delta(q2, 1)\}$

$= \varepsilon\text{-closure}\{\phi\}$

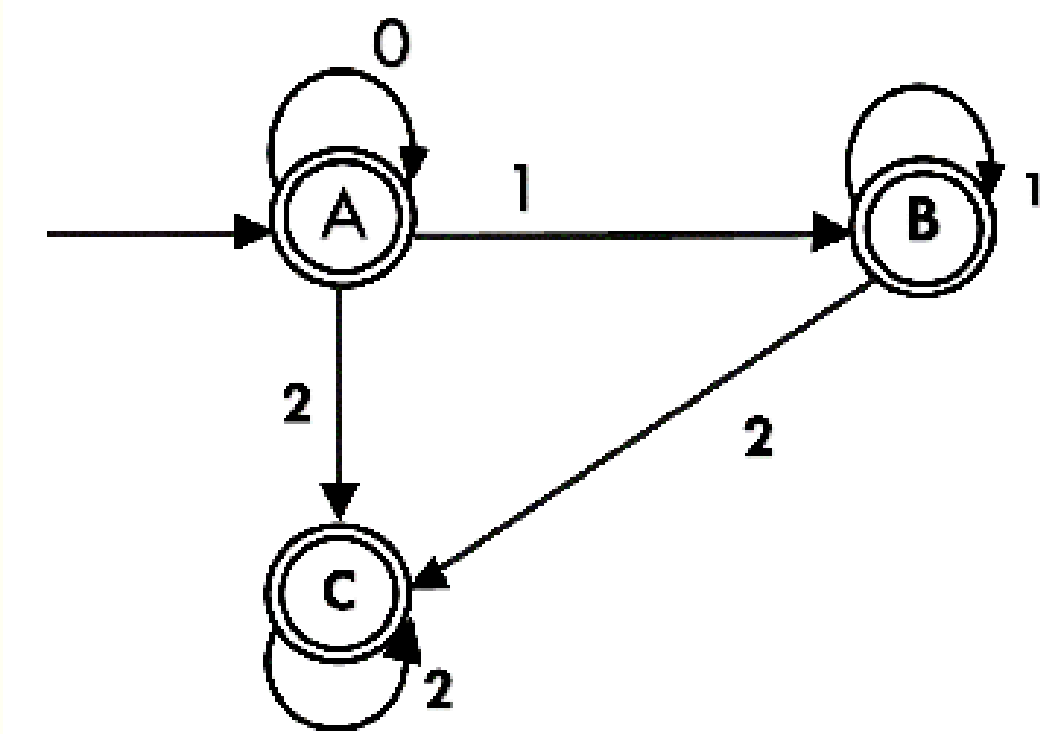
$= \phi$

$\delta'(C, 2) = \varepsilon\text{-closure}\{\delta(q2, 2)\}$

$= \{q2\}$

Conversion from NFA with ε to DFA – Example 2

- Hence the DFA is



- As A = {q0, q1, q2} in which final state q2 lies hence A is final state. B = {q1, q2} in which the state q2 lies hence B is also final state. C = {q2}, the state q2 lies hence C is also a final state.

REFERENCE

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques and Tools”, Second Edition, Pearson Education Limited, 2014.
2. Randy Allen, Ken Kennedy, “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”, Morgan Kaufmann Publishers, 2002.
3. Steven S. Muchnick, “Advanced Compiler Design and Implementation”, Morgan Kaufmann Publishers - Elsevier Science, India, Indian Reprint, 2003.
4. Keith D Cooper and Linda Torczon, “Engineering a Compiler”, Morgan Kaufmann Publishers, Elsevier Science, 2004.
5. V. Raghavan, “Principles of Compiler Design”, Tata McGraw Hill Education Publishers, 2010.
6. Allen I. Holub, “Compiler Design in C”, Prentice-Hall Software Series, 1993.


Thank you