Lexical Analysis

Mrs. Sunita M Dol

(sunitaaher@gmail.com)

Assistant Professor, Dept of Computer Science and Engineering

Walchand Institute of Technology, Solapur
(www.witsolapur.org)

Lexical Analysis

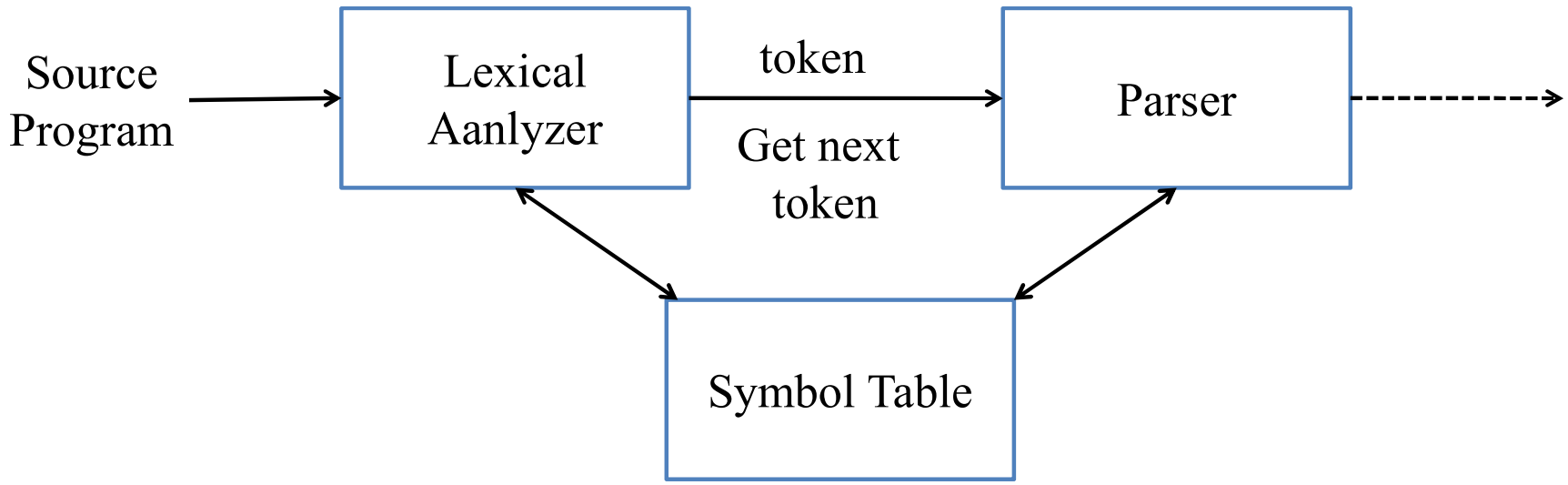
- Role of a Lexical analyzer
- Input buffering
- Specification and recognition of tokens
- Finite automata implications
- Designing a lexical analyzer generator

Introduction

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

The Role of Lexical Analyzer

- The Lexical Analyzer is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



Interaction of lexical analyzer with parser

The Role of Lexical Analyzer

- Upon receiving a ‘get next token’ command form the parser, the lexical analyzer reads the input character until it can identify the next token. The Lexical Analyzer return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.
- Lexical Analyzer may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

The Role of Lexical Analyzer

- **Issues in Lexical Analyzer**
  1. **Simpler Design** - e.g. Parser for comments and white space is significantly more complex than one that can assume comments and white spaces are already removed.
  2. **Compiler Efficiency** is Improved - Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of compiler.
  3. **Compiler Portability** is Improved – The representation of special or non-standard symbols, such as ↑ in Pascal, can be isolated in the lexical analyzer.

The Role of Lexical Analyzer

- **Token, Lexeme and Pattern**
  - **Token**: Token is a sequence of characters that can be treated as a single logical entity.

    Typical tokens are Identifiers, keywords,, operators, special symbols and constants
  - **Pattern**: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
  - **Lexeme**: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

The Role of Lexical Analyzer

- **Token, Lexeme and Pattern**

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

Info is :

1.Stored in symbol table

2.Returned to parser

The Role of Lexical Analyzer

- **Attributes for token**
  - Tokens influence parsing decision; the attributes influence the translation of tokens.
  - Example: 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>

The Role of Lexical Analyzer

- **Lexical errors**
  - Lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of remaining input.

    For example:

    fi (a == f(x))….

  - Panic mode recovery - deleting successive characters until a well formed token is formed.
  - Error-recovery actions are:
    - ✓ Delete one character from the remaining input.
    - ✓ Insert a missing character in to the remaining input.
    - ✓ Replace a character by another character.
    - ✓ Transpose two adjacent characters.

Input Buffering

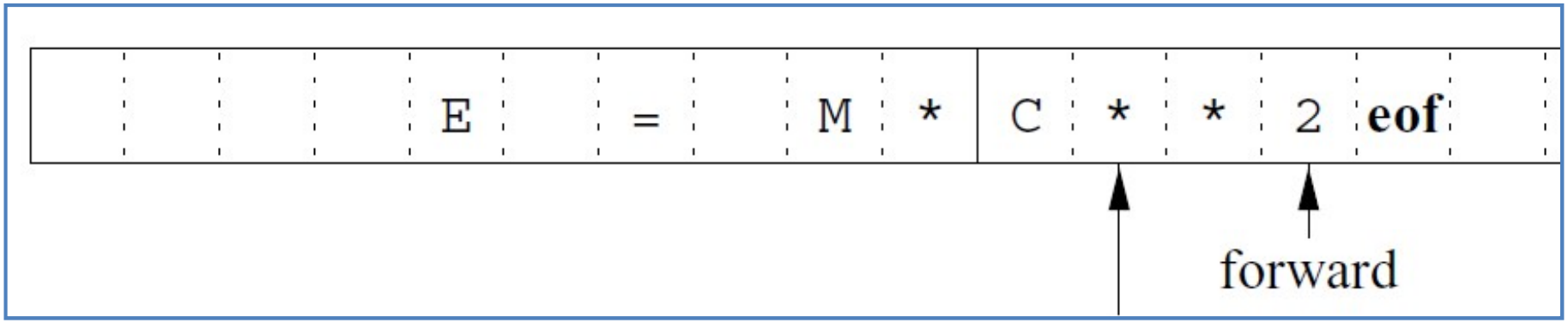
- Three approaches to implement lexical analyzer
  - Use lexical analyzer generator
  - Write lexical analyzer in a conventional system programming language using IO facilities of that language to read the input.
  - Write a lexical analyzer in assembly language

Input Buffering

- **Buffer pairs**
  - Buffer is divided into two N-character halves where N is the number of character on one disk block.
  - Two pointer namely forward pointer& lexeme beginning are maintained.
  - Initially both pointer points to the first character of the next lexeme to be found.
  - Forward pointer scans ahead until a match for a pattern is found.
  - Once the next lexeme is determined, the forward pointer is set to the character at its right end.
  - After lexeme is processed, both pointers are set to the character immediately past the lexeme.

Input Buffering

- **Buffer pairs**



Input Buffering

- **Buffer pairs**

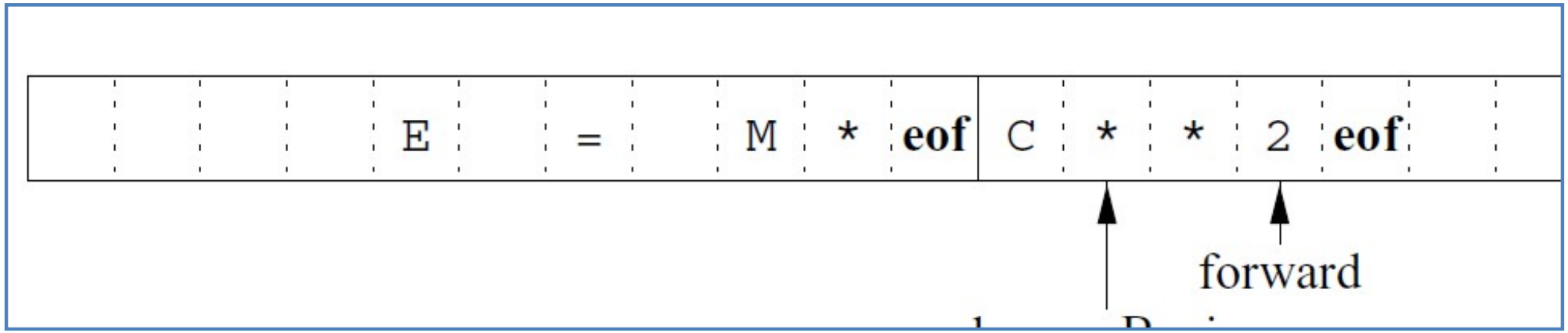
```
Algorithm
    If forward at end of first half then
    begin
        reload second half;
        forward=forward+1;
    end
    else if forward at end of second half then
    begin
        reload first half;
        move forward to the beginning of first half.
    end
    else forward=forward+1;
```

Input Buffering

- **Sentinels**
  - We must check each time, we move the forward pointer that we have moved off one half of the buffer then we must reload the other half.
  - Except at the ends of the buffer halves, the code requires two tests for each advance of the forward pointer.
  - We can reduce the two tests to one if we extend each buffer half to hold a sentinel character at the end.
  - The sentinels is a special character that can not be part of the source program.

Input Buffering

- **Sentinels**



Input Buffering

- **Sentinels**

```
forward=forward+1;
        If forward pointer = eof then
        begin
        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 the beginning of first half.
        end
        else
                terminate lexical analysis
    end
```

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.

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.
  - A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
  - The length of a string s, usually written |S|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Specification of Tokens

- **Strings and Languages**
  - A language is any countable set of strings over some fixed alphabet. Abstract languages like Φ, the empty set, or {ε}, the set containing only the empty string, are languages under this definition
  - If x and y are strings, then the concatenation of x and y, denoted xy, is the string formed by appending y to x. For example, if x = dog and y = house, then xy = doghouse.

Specification of Tokens

- **Strings and Languages**
  - 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.

Specification of Tokens

- **Strings and Languages**
  - The following string-related terms are commonly used:
    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**
  - In lexical analysis, the most important operations on languages are union, concatenation, and closure.

OPERATION	DEFINITION AND N
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ o}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$

Specification of Tokens

- **Operations on Languages**
  - Let L be the set of letters {A, B, C,…, Z, a, b, c,…,z} and let D be the set of digits {0, 1, 2,…, 9}.
  - L ∪ 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.
  - L4 is the set of all 4-letter strings.
  - L* is the set of all strings of letters, including ε, the empty string.
  - L(L ∪ 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.

Specification of Tokens

- **Regular Expression**
  - 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 subexpression. Here are the rules that define the regular expressions over some alphabet Σ 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 Σ, 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 Expression**
  - The regular expressions are built recursively out of smaller regular expressions, using the rules described below.

    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) ∪ 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 Expression**
  - The unary operator * has highest precedence and is left associative.
  - Concatenation has second highest precedence and is left associative.
  - | has lowest precedence and is left associative.

Specification of Tokens

- **Regular Expression**
  - Let Σ = {a, b}.
    - ✓ The regular expression a|b denotes the language {a, b}.
    - ✓ (a|b)(a|b) denotes {aa, ab, ba, bb}, the language of all strings of length two over the alphabet Σ. Another regular expression for the same language is aa|ab|ba|bb.
    - ✓ a* denotes the language consisting of all strings of zero or more a's, thatis, {ε, a, aa, aaa, ...}.
    - ✓ (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,...}. Another regular expression for the same language is (a*b*)*.
    - ✓ a|a*b denotes the language {a, ab, aab, ...}, that is, the string a and all strings consisting of zero or more a's and ending in b.

Specification of Tokens

- **Regular Expression**
  - 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.

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is assoc
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distribu
$\epsilon r = r\epsilon = r$	ϵ is the identity for cor
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a clo

Specification of Tokens

- **Regular Definition**
  - If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

    $d_1 \rightarrow r_1$

    $d_2 \rightarrow r_2$

    …

    $d_n \rightarrow r_n$

    where:

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

Specification of Tokens

- **Regular Definition**
  - Example - C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers.

    letter →A | B |…| Z | a | b |… | z |-

    digit → 0 |1 |…|9

    id → letter (letter | digit)*

Specification of Tokens

- **Regular Definition**
  - Example - Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

    digit → 0 | 1 …| 9

    digits → digit digit*

    optionalFraction → . digits | ε

    optionalExponent → (E (+ | - | ε) digits) | ε

    number → digits optionalFraction optionalExponent

Specification of Tokens

- **Extensions of Regular Expressions**
  - **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 (L(r))+ . The operator + has the same precedence and associativity as the operator *. Two useful algebraic laws, r* = r+| ε and r+ = rr* relate the Kleene closure and positive closure.

Specification of Tokens

- **Extensions of Regular Expressions**
  - **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) ∪{ε}. The ? operator has the same precedence and associativity as * and +.
  - **Character classes**. A regular expression a1|a2|…|an, where the ai's are each symbols of the alphabet, can be replaced by the shorthand [a1a2 … an]. More importantly, when a1,a2, …,an form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a1-an, 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.

Specification of Tokens

- **Extensions of Regular Expressions**
  - **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) ∪{ε}. The ? operator has the same precedence and associativity as * and +.
  - **Character classes**. A regular expression a1|a2|…|an, where the ai's are each symbols of the alphabet, can be replaced by the shorthand [a1a2 … an]. More importantly, when a1,a2, …,an form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by a1-an, 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.

Specification of Tokens

- **Extensions of Regular Expressions**
  - Example - C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers.

    letter →[A-Za-z_]

    digit → [0-9]

    id → letter (letter | digit)*

Specification of Tokens

- **Extensions of Regular Expressions**
  - Example - Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. The regular definition

    digit → [0-9]

    digits → digit+

    optionalFraction → (. digits)+

    optionalExponent → (E [+-]?digits)?

    number → digits optionalFraction optionalExponent

Recognition of Tokens

- We 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
- Consider the example

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

expr → term relop term
| term

term → id
| number

Recognition of Tokens

- We 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
- Consider the example

A grammar for branching statements

stmt → if expr then stmt
| if expr then stmt else stmt
| ε
expr → term relop term
| term
term → id
| number

Patterns for tokens

digit → [0-9]
digits → digit+
number → digits (. digits)? (E [+-]? digits)?
letter → [A-Za-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

Recognition of Tokens

- For this language, the lexical analyzer will recognize the keywords if, then, and else, as well as lexemes that match the patterns for relop, id, and number. To simplify matters, we make the common assumption that keywords are also reserved words: that is, they are not identifiers, even though their lexemes match the pattern for identifiers.

Recognition of Tokens

- In addition, we assign the lexical analyzer the job of stripping out white-space, by recognizing the \token" ws defined by:

  ws → (blank | tab | newline)+

  Here, blank, tab, and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is di erent from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.

Recognition of Tokens

- The table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value

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

Recognition of Tokens

- **Transition Diagram**
  - 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 lexeme Begin 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

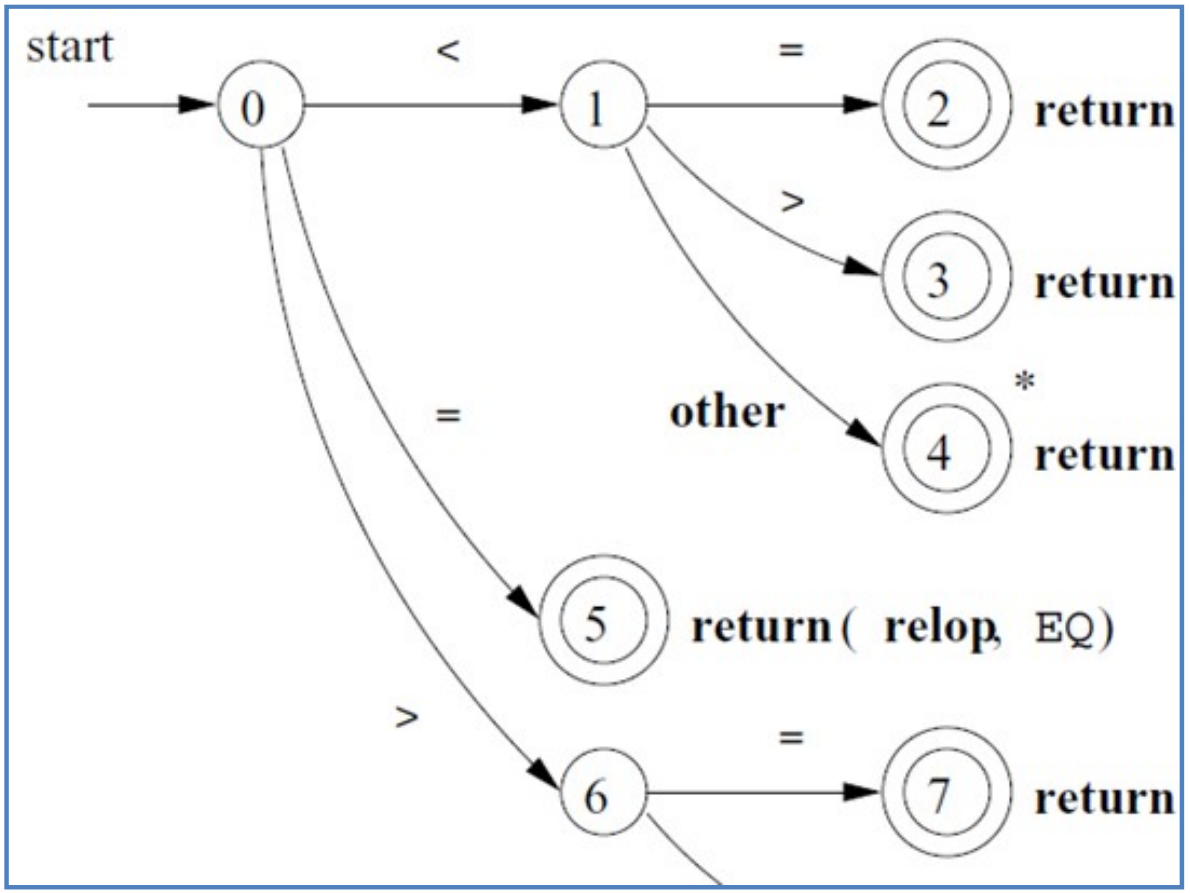
- **Transition Diagram**
  - Some important conventions about transition diagrams are:
    - ✓ Certain states are said to be accepting, or final. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the lexemeBegin and forward pointers. 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.

Recognition of Tokens

- **Transition Diagram**
  - Some important conventions about transition diagrams are:
    - ✓ 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. 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.
    - ✓ One state is designated the start state, or initial state; it is indicated by an edge, labeled “start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Recognition of Tokens

- **Transition Diagram**

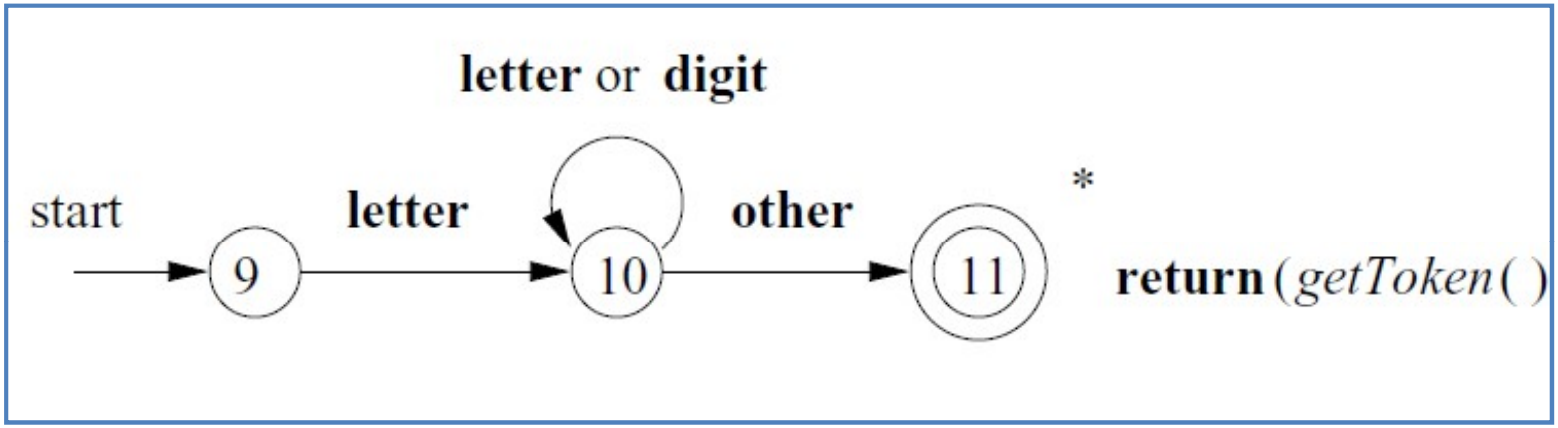


Recognition of Tokens

- **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.
       1. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent.
       2. 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.
       3. Any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id.
       4. 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.

Recognition of Tokens

- **Recognition of reserved words and identifiers**



Recognition of Tokens

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

    2. Create separate transition diagrams for each keyword.
        1. Such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier.
        2. It is necessary to check that the identifier has ended, or else we would return token then in situations where the correct token was id, with a lexeme like the next value that has then as a proper prefix.
        3. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.

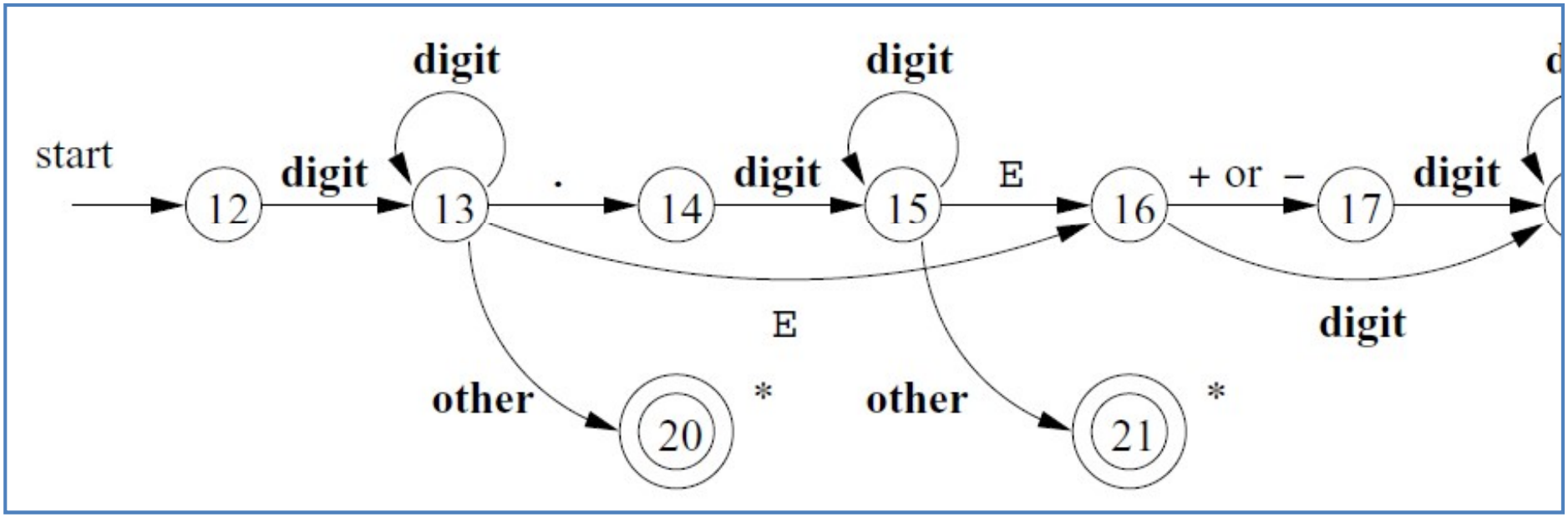
Recognition of Tokens

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

    2. Create separate transition diagrams for each keyword.
        1. Such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "nonletter-or-digit," i.e., any character that cannot be the continuation of an identifier.
        2. It is necessary to check that the identifier has ended, or else we would return token then in situations where the correct token was id, with a lexeme like the next value that has then as a proper prefix.
        3. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.

Recognition of Tokens

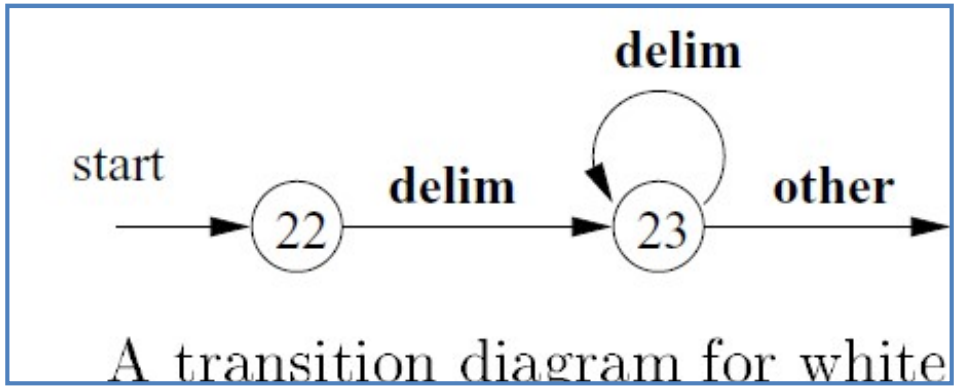
- **Transition Diagram for number**



Recognition of Tokens

- **Transition Diagram for white space**

  We look 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 white

Recognition of Tokens

- **Implementation of Transition Diagram**
  - 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.

Recognition of Tokens

- **Implementation of Transition Diagram**

```
Token nexttoken() {
        while(1) {
                switch (state) {
                        ……
                        case 10: c=nextchar();
                                if(isletter(c)) state=10;
                                elseif (isdigit(c)) state=10;
                                else state=11;
                                break;
                        ……
                }
        }
}
```

Finite Automata

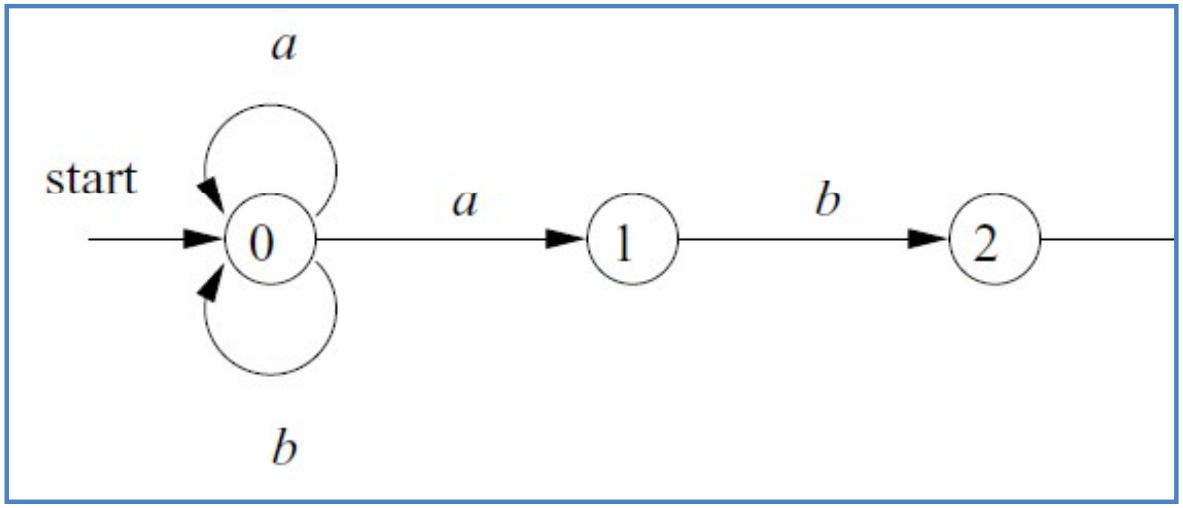
- Finite automata are **recognizers**; they simply say yes or no about each possible input string. Finite automata come in two favors:
  - **Nondeterministic finite automata** (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ε, the empty string, is a possible label.
  - **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.
- Both deterministic and nondeterministic finite automata are capable of recognizing the same languages.

Finite Automata

- **Nondeterministic finite automaton (NFA)**
  - It consists of:
    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 Σ∪{ε} 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).

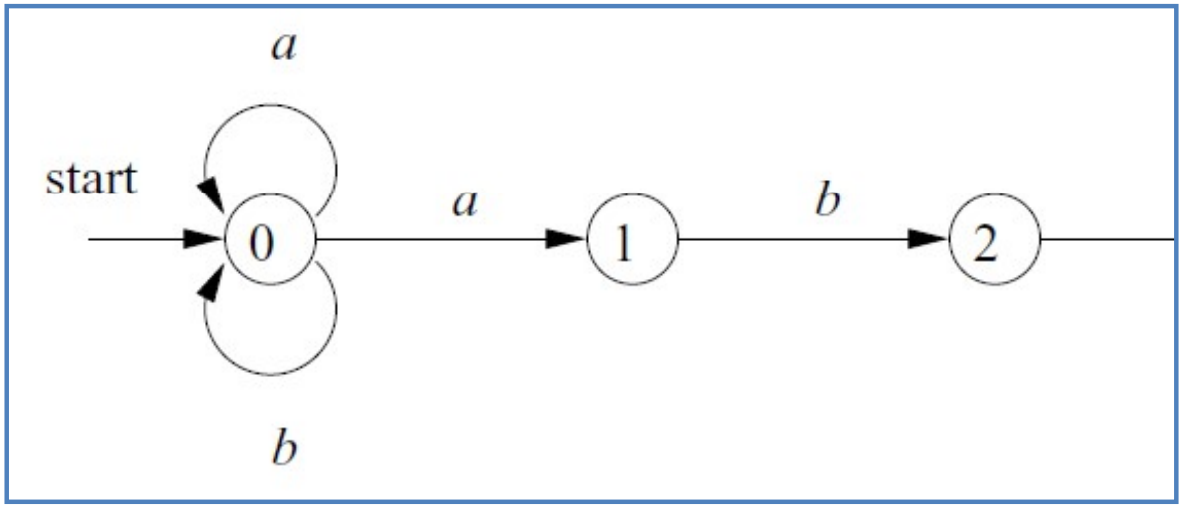
Finite Automata

- The transition graph for an NFA recognizing the language of regular expression (a|b)*abb is shown



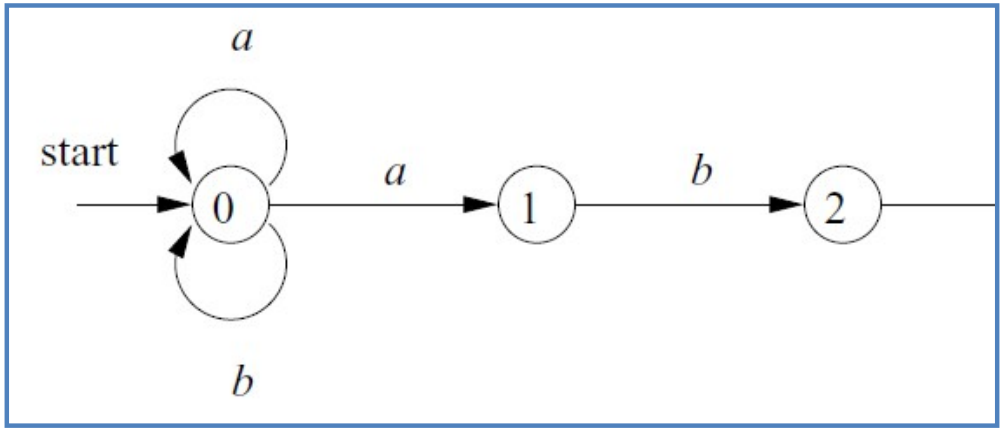
Finite Automata

- The transition graph for an NFA recognizing the language of regular expression (a|b)*abb is shown



Finite Automata

- **Transition Table**
  - We can also 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
0	$\{0, 1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

Finite Automata

- **Acceptance of 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.
  - ε labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.
  - The language defined (or accepted) by an NFA is the set of strings labeling some path from the start to an accepting state.

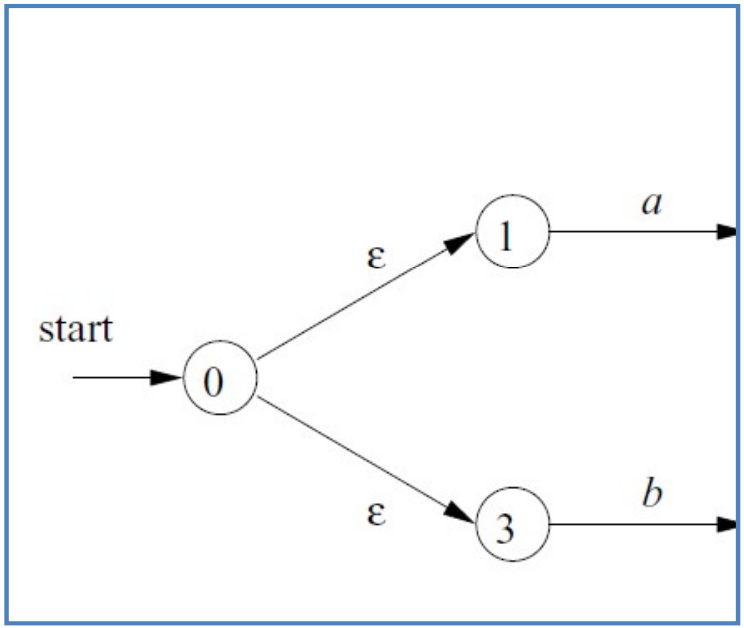
Finite Automata

- **Implementing NFA**

```
S ← ε-closure({s0}) { set all of states can be accessible from s0 by ε-transitions }
c ← nextchar
while (c != eos) {
begin
        s ← ε-closure(move(S,c))  { set of all states can be accessible from a state in S
        c ← nextchar                by a transition on c }
end
if (S∩F != Φ) then { if S contains an accepting state }
        return "yes"
else
        return "no"
```

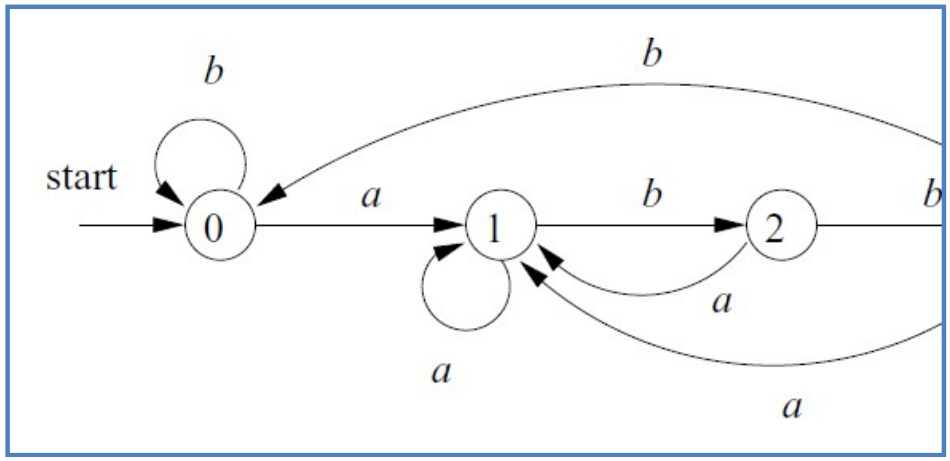
Finite Automata

- **Acceptance of strings by automata**
  - String aaa is accepted



Finite Automata

- **Deterministic Finite Automata**
  - A deterministic finite automaton (DFA) is a special case of an NFA where:
    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.
  - If we are using a transition table to represent a DFA, then each entry is a single state.



Finite Automata

- **Deterministic Finite Automata**

  Algorithm: Simulating a DFA.

  INPUT: An input string x terminated by an end-of-file character eof. A DFA D with start state s0, accepting states F, and transition function move.

  OUTPUT: Answer “yes” if D accepts x; “no” otherwise.

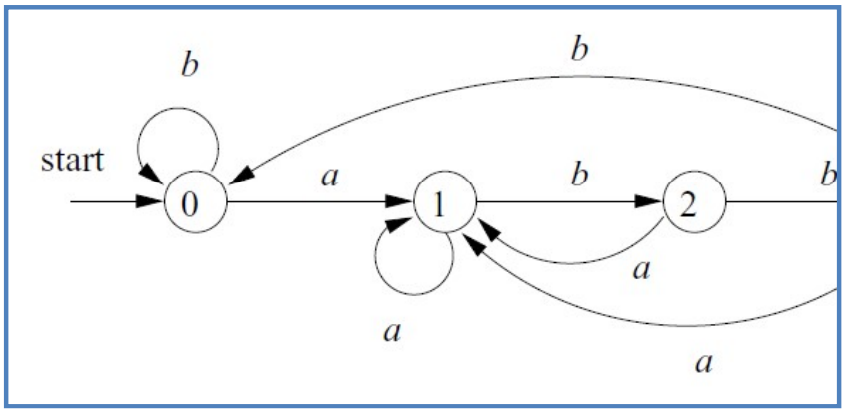
  METHOD: The function move(s, c) gives the state to which there is an edge from state s on input c. The function nextChar returns the next character of the input string x.

Finite Automata

- **Deterministic Finite Automata**

  Algorithm: Simulating a DFA.

```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no“;
```



Given the input string ababb, this DFA enters the sequence of states 0,1,2,1,2,3 and returns “Yes”.

Finite Automata

- **Conversion of an NFA into a DFA**

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

Finite Automata

- **Conversion of an NFA into a DFA**

  Algorithm: The subset construction of a DFA from an NFA.

  INPUT: An NFA N.

  OUTPUT: A DFA D accepting the same language as N.

  METHOD: Our algorithm constructs a transition table Dtran for D. Each state of D is a set of NFA states, and we construct Dtran so D will simulate "in parallel" all possible moves N can make on a given input string. Our first problem is to deal with ε -transitions of N properly. Note that s is a single state of N, while T is a set of states of N.

Finite Automata

- **Conversion of an NFA into a DFA**

  Algorithm: The subset construction of a DFA from an NFA.

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

Finite Automata

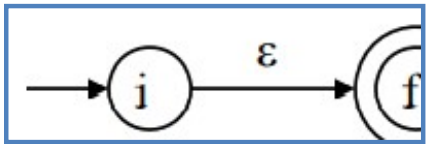
- **Conversion of an NFA into a DFA**

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

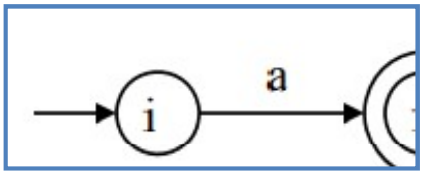
Computing ε-closure(T)

Finite Automata

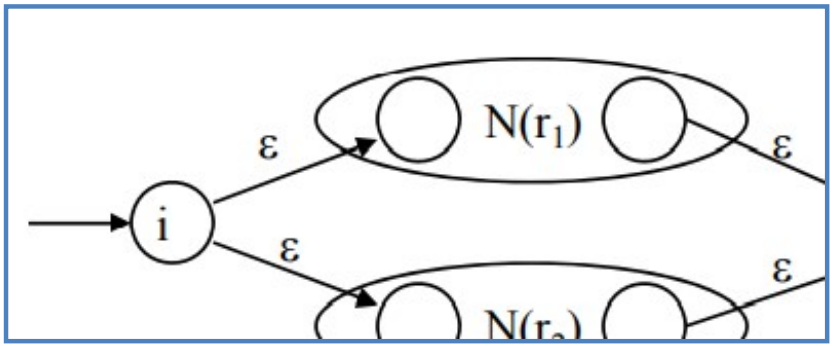
- **Conversion of an NFA into a DFA**
  - **Thompson's Construction**
    - ✓ To recognize an empty string ε



    - ✓ To recognize a symbol a in the alphabet Σ

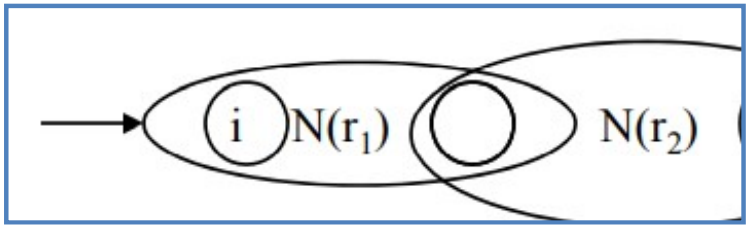


    - ✓ If N(r1) and N(r2) are NFAs for regular expressions r1 and r2

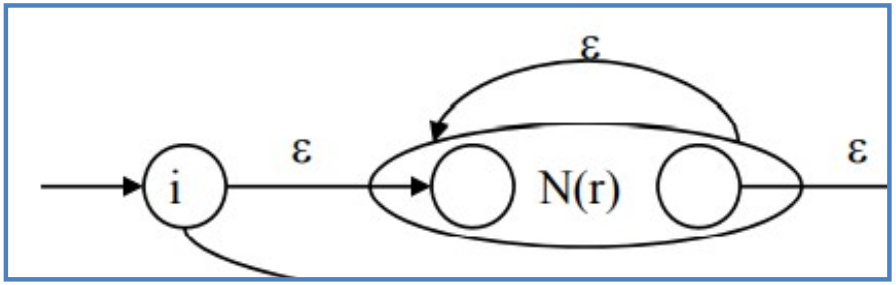


Finite Automata

- **Conversion of an NFA into a DFA**
  - **Thompson's Construction**
    - ✓ For regular expression r1 r2

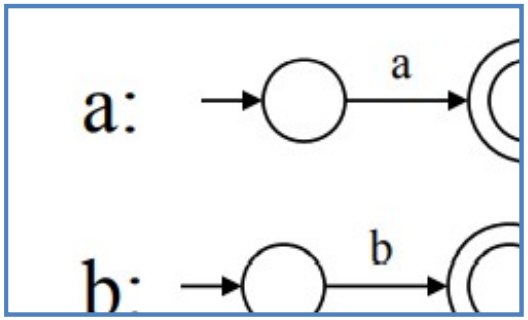


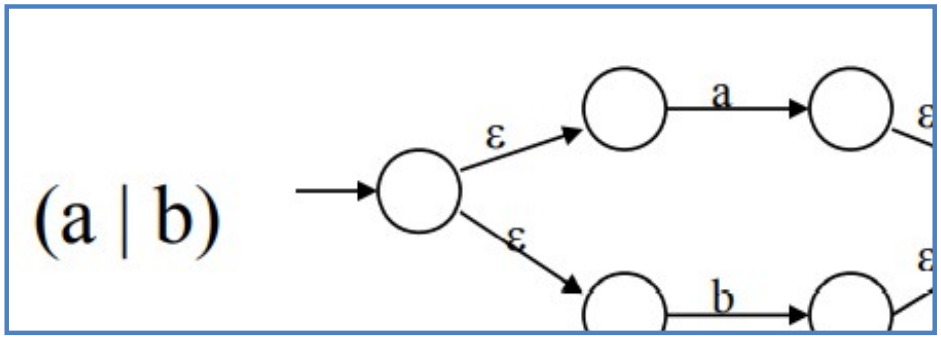
    - ✓ For regular expression r*

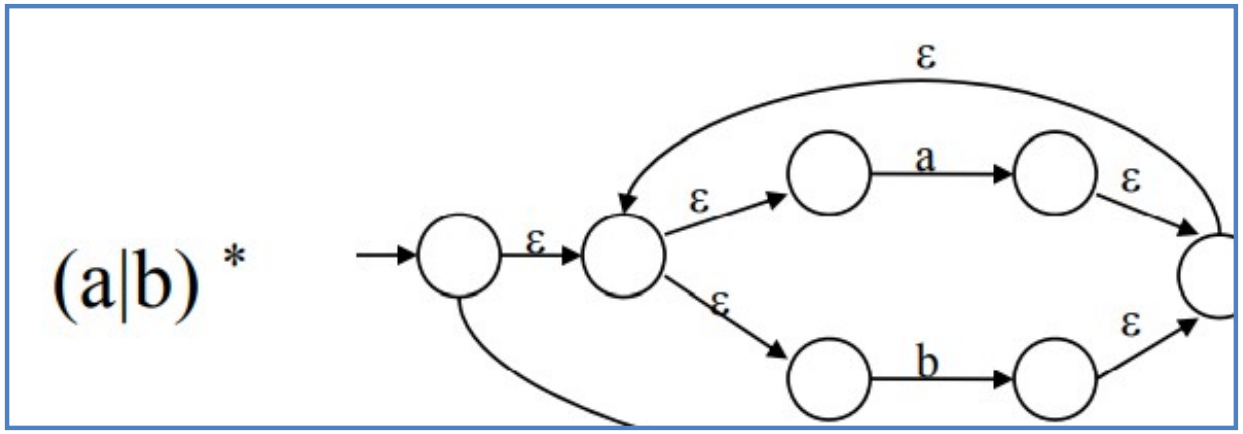


Finite Automata

- **Conversion of an NFA into a DFA**

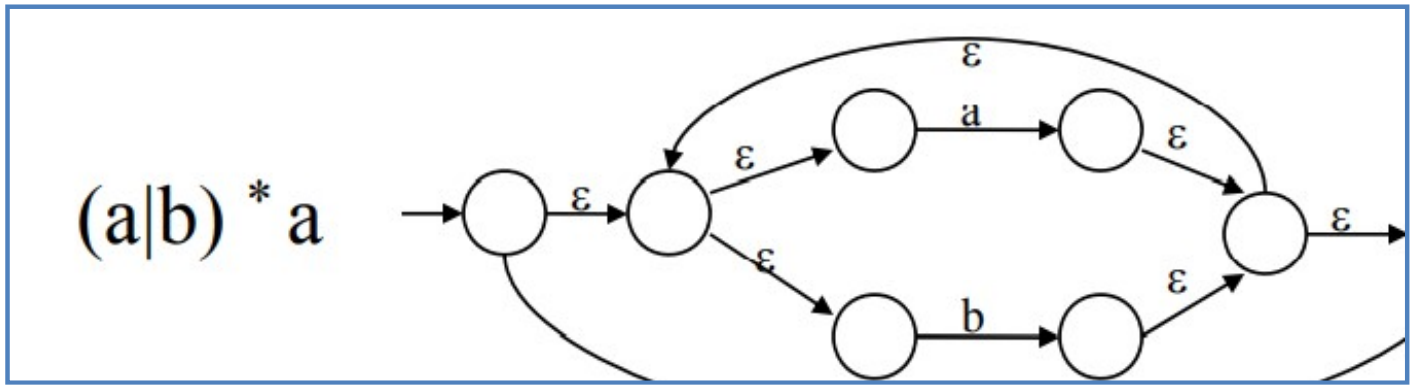






Finite Automata

- **Conversion of an NFA into a DFA**



Finite Automata

- **Conversion of an NFA into a DFA**

S0 = ε-closure({0}) = {0,1,2,4,7} S0 into DS as an unmarked state

⇓ mark S0

ε-closure(move(S0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S1 S1 into DS

ε-closure(move(S0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S2 S2 into DS

transfunc[S0,a] ← S1 transfunc[S0,b] ← S2

⇓ mark S1

ε-closure(move(S1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S1

ε-closure(move(S1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S2

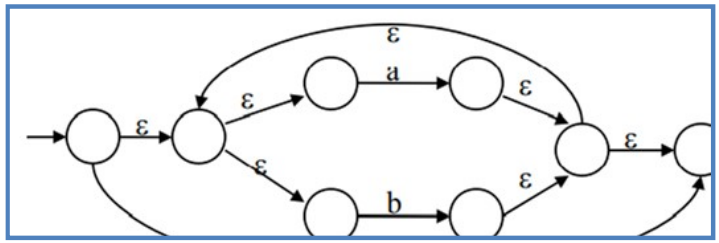
transfunc[S1,a] ← S1 transfunc[S1,b] ← S2

⇓ mark S2

ε-closure(move(S2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S1

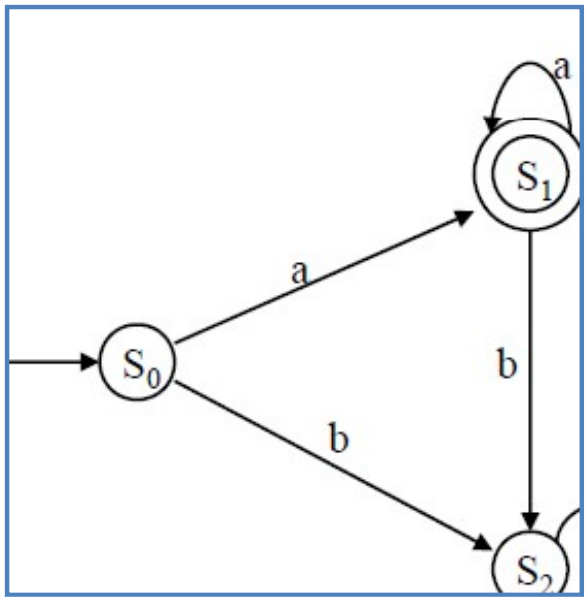
ε-closure(move(S2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S2

ε ε transfunc[S2,a] ← S1 transfunc[S2,b] ← S2

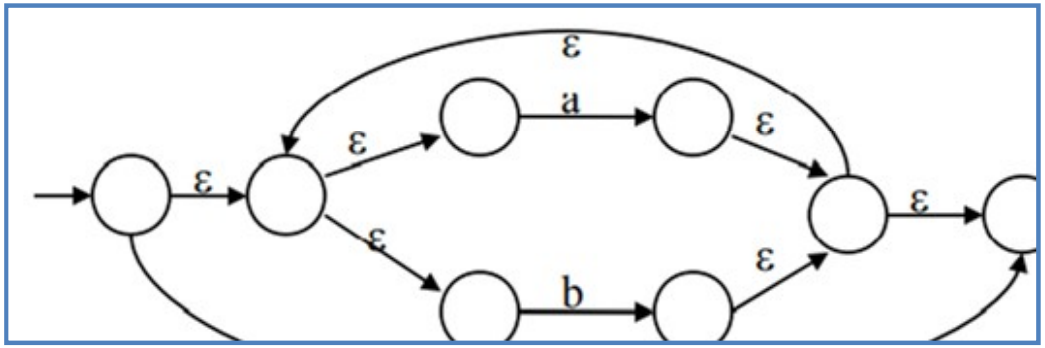


Finite Automata

- **Conversion of an NFA into a DFA**
  - S0 is the start state of DFA since 0 is a member of S0={0,1,2,4,7}
  - S1 is an accepting state of DFA since 8 is a member of S1 = {1,2,3,4,6,7,8}



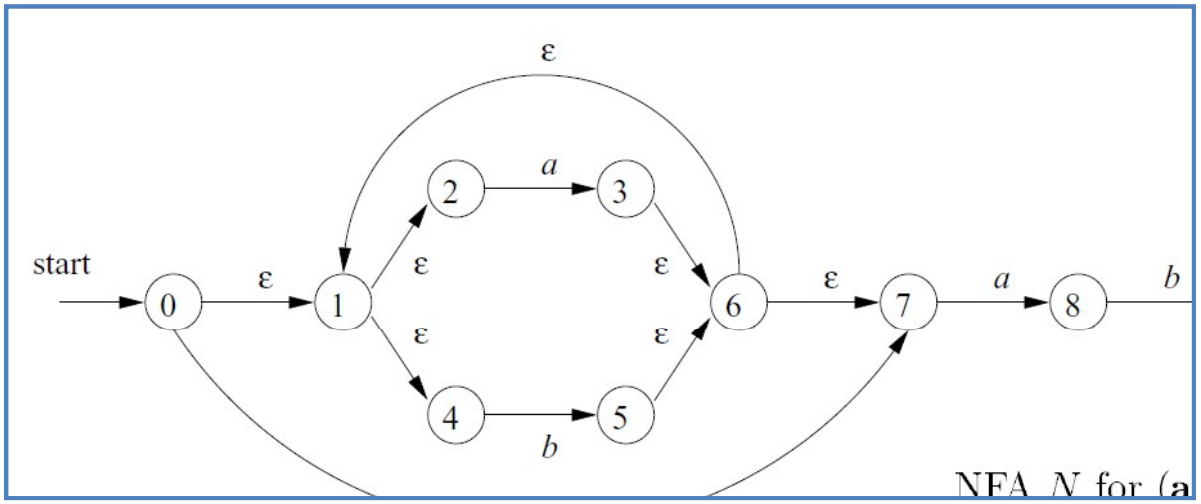
DFA



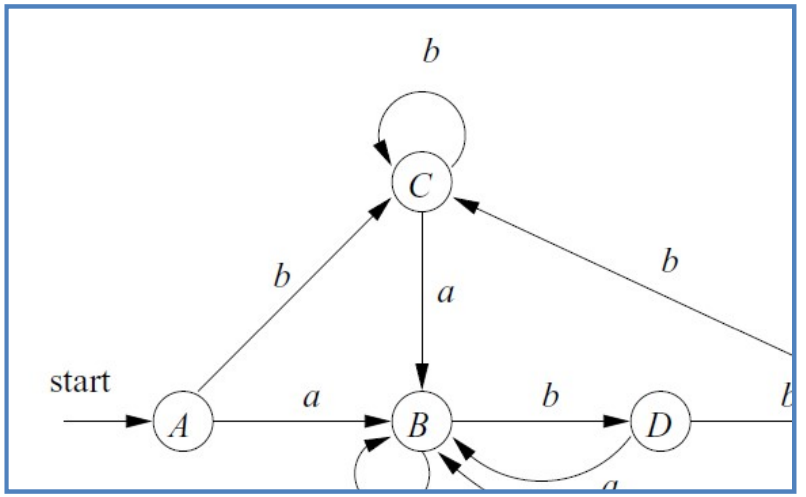
NFA

Finite Automata

- **Conversion of an NFA into a DFA**



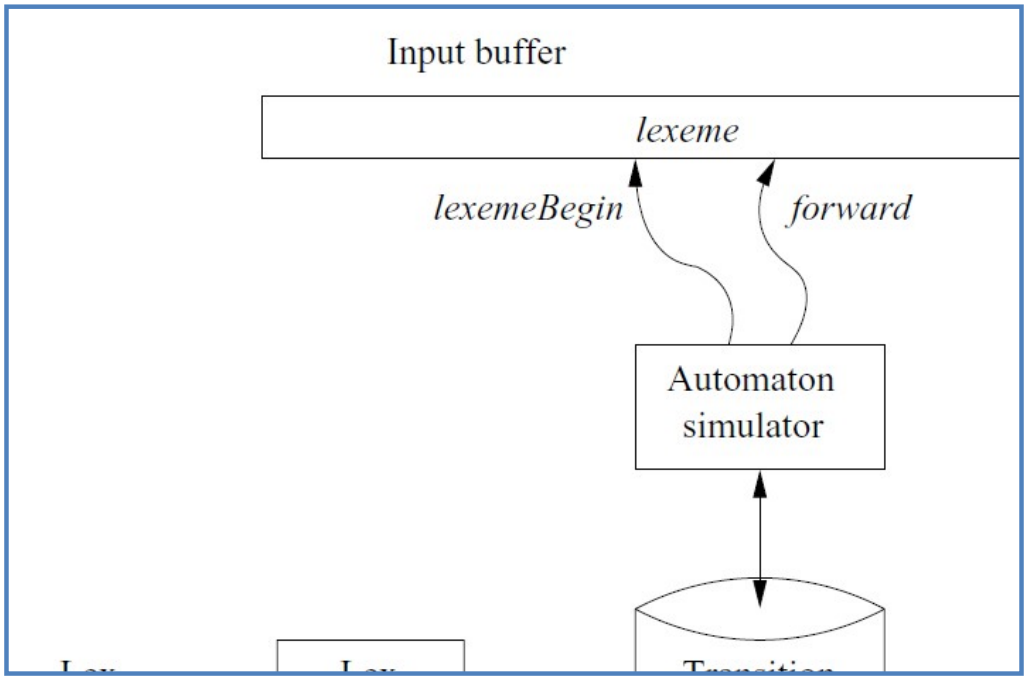
NFA N for (a



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

Design of Lexical Analyzer Generator

- Figure overviews the architecture of a lexical analyzer generated by Lex. The program that serves as the lexical analyzer includes a fixed program that simulates an automaton. The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself.



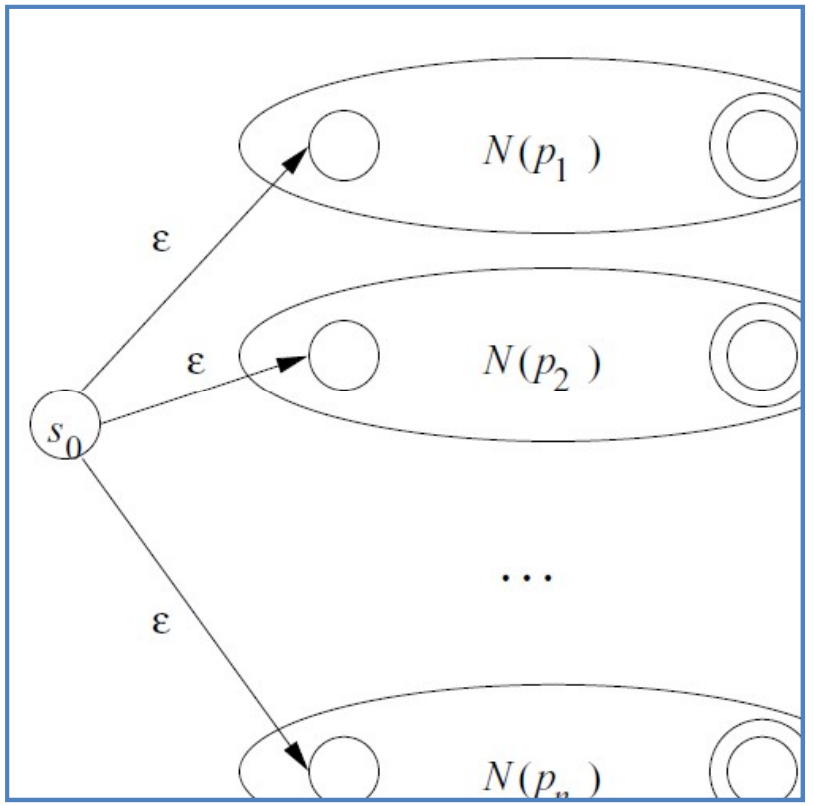
A Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

Design of Lexical Analyzer Generator

- These components are:
  1. A transition table for the automaton.
  2. Those functions that are passed directly through Lex to the output
  3. The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

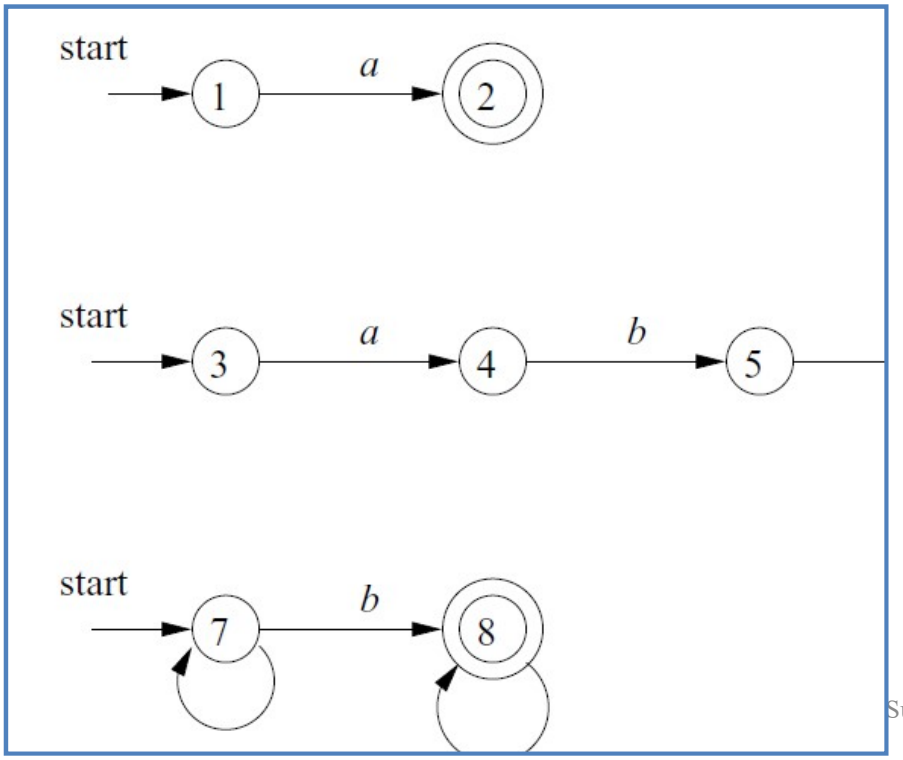
Design of Lexical Analyzer Generator

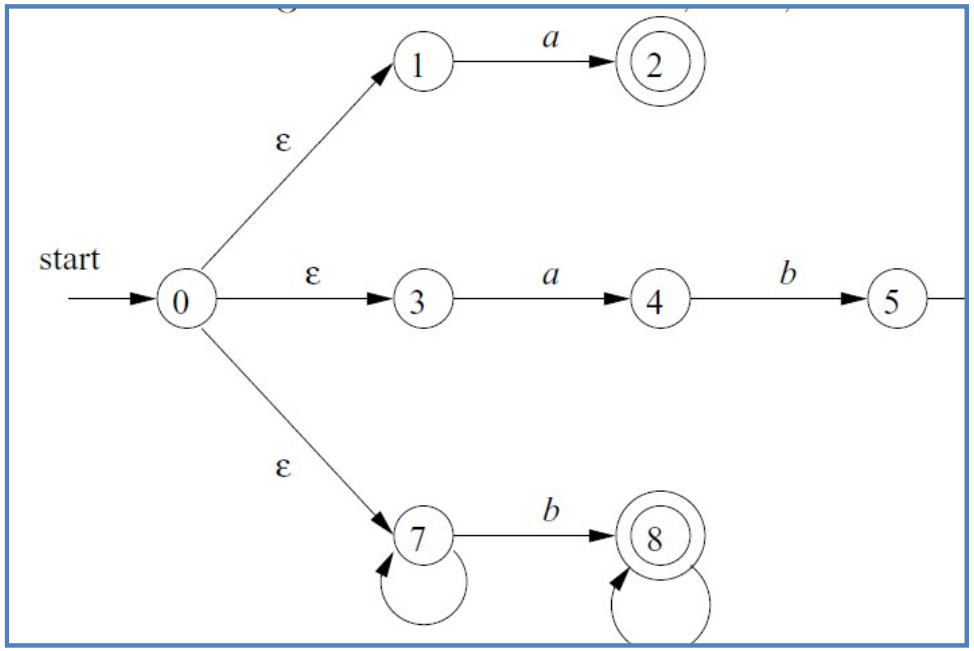
- To construct the automaton, we begin by taking each regular-expression pattern in the Lex program and converting it, to an NFA. We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with ε-transitions to each of the start states of the NFA's Ni for pattern pi. This construction is shown in Fig.



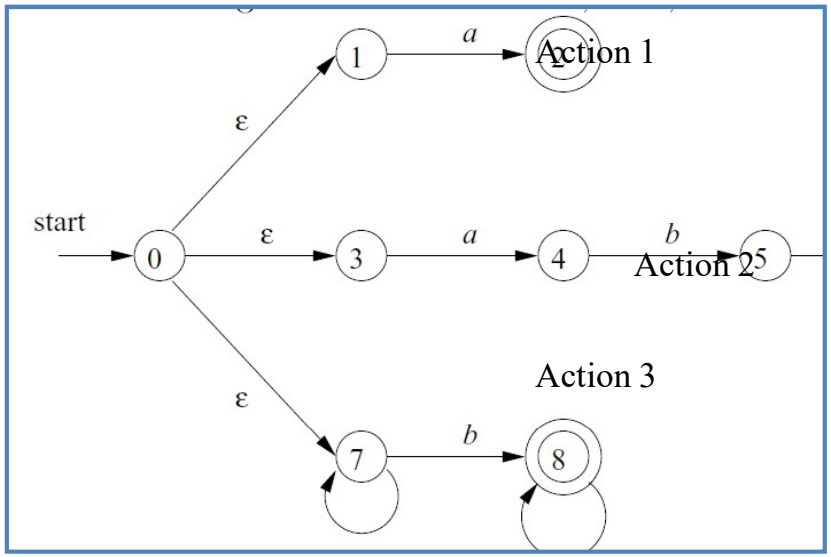
Design of Lexical Analyzer Generator

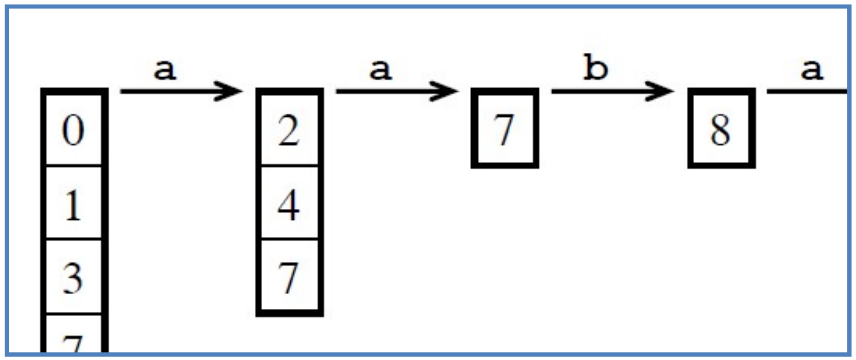
a	{action A1 for pattern p1 }
abb	{ action A2 for pattern p2 }
a*b+	{ action A3 for pattern p3 }





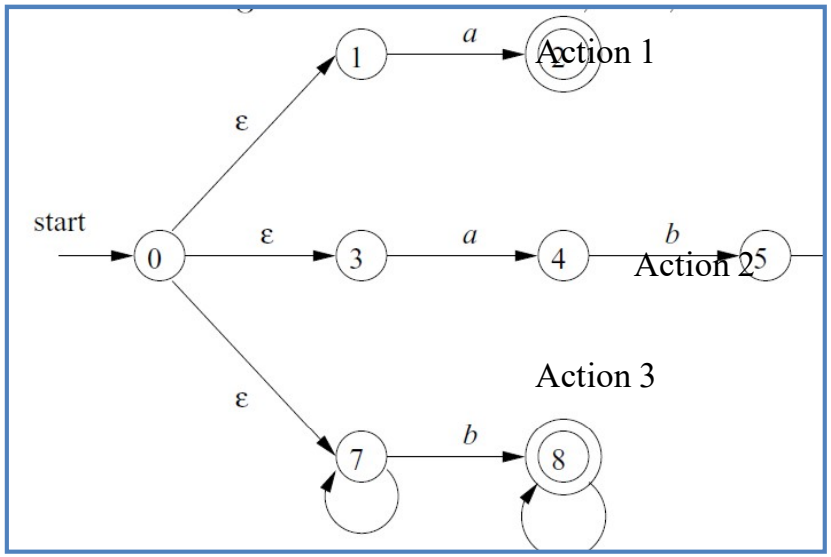
Design of Lexical Analyzer Generator

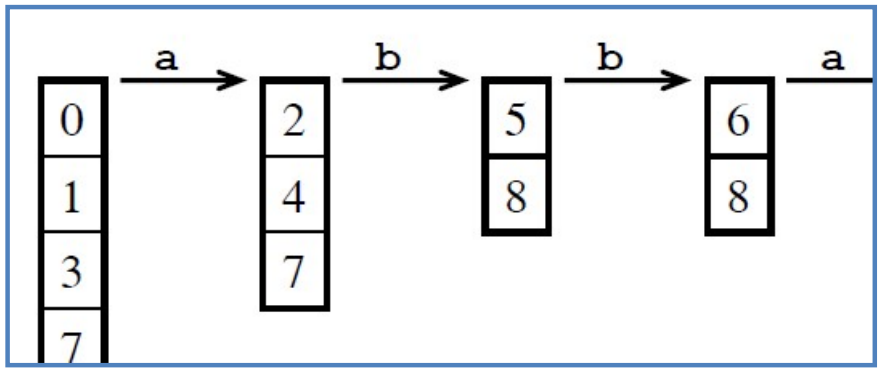




Must find the longest match:
Action3 Continue until no further moves are possible
When last state is accepting: execute action

Design of Lexical Analyzer Generator





action2
action3
When two or more accepting states are reached,
The first action given in the Lex specification is executed

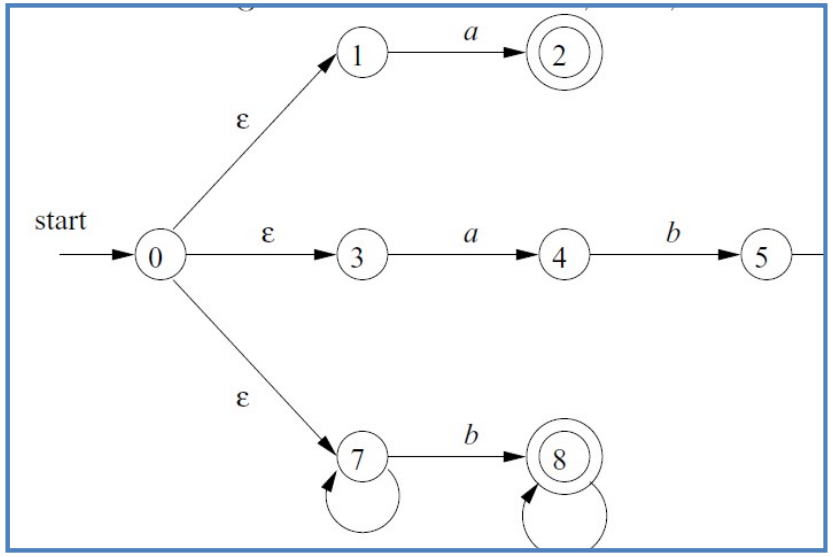
Design of Lexical Analyzer Generator

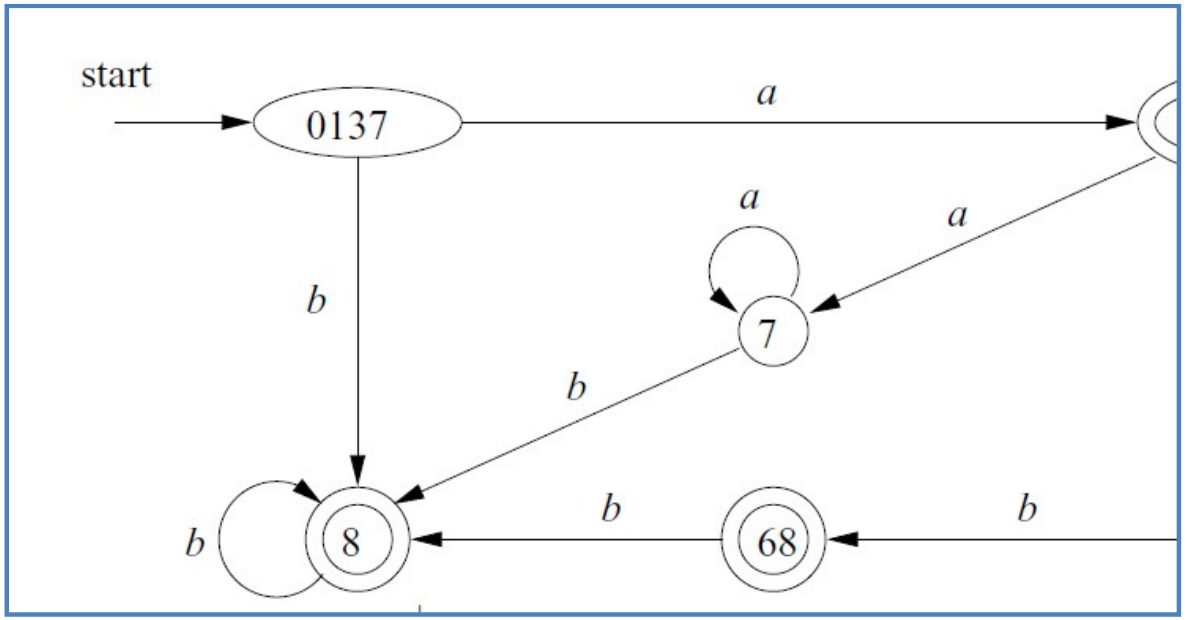
- **DFA's for Lexical Analyzers**

  Another architecture, resembling the output of Lex, is to convert the NFA for all the patterns into an equivalent DFA, using the subset construction. Within each DFA state, if there are one or more accepting NFA states, determine the first pattern whose accepting state is represented, and make that pattern the output of the DFA state.

Design of Lexical Analyzer Generator

- **DFA's for Lexical Analyzers**





References

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullmann “Compilers- Principles, Techniques and Tools”, Pearson Education.

Thank You !!!