

# 18IS54 - Compiler Design

Unit 1

Lexical Analysis

# Outline

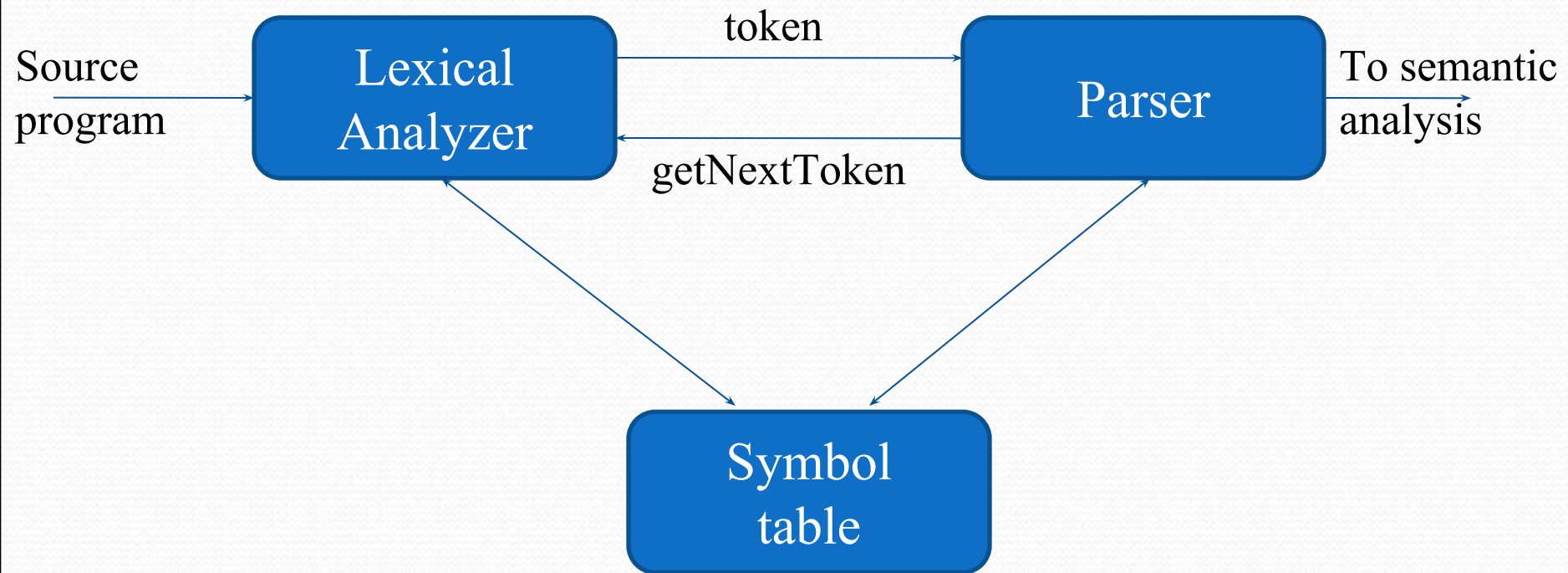
- Role of lexical analyzer
- Specification of tokens
- Recognition of tokens

# The role of 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.
- When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table.



# The role of lexical analyzer



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

# The role of lexical analyzer

Since 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).

Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.

In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions.



# The role of lexical analyzer

- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer.
- Sometimes, 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** which produces the sequence of tokens as output.

# Why to separate Lexical analysis and parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

## 1. Simplicity of design

- The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks.
  - For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex.

## 2. Improving compiler efficiency

- A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing.
- Specialized buffering techniques for reading input characters can speed up the compiler significantly

## 3. Enhancing compiler portability

- Input-device-specific peculiarities can be restricted to the lexical analyzer.



# Tokens, Patterns and Lexemes


- 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 , For example: id and num
- 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.
  - For example: abc and 123
- 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.
  - For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.
    - For example: “letter followed by letters and digits” and “non-empty sequence of digits”



# Example

Token	Informal description	Sample lexemes
<b>n</b>		
<b>if</b>	Characters i, f	if
<b>else</b>	Characters e, l, s, e	else
<b>comparison</b>	< or > or <= or >= or == or !=	<=, !=
<b>id</b>	Letter followed by letter and digits	pi, score, D2
<b>number</b>	Any numeric constant	3.14159, 0, 6.02e23
<b>literal</b>	Anything but “ sorrounded by “	“core dumped”

```
printf(“total = %d\n”, score);
```



In many programming languages, the following classes cover most or all of the tokens:

- One token for each keyword. The pattern for a keyword is the same as the keyword itself.
- Tokens for the operators, either individually or in classes such as the token comparison(< or > or <= or >= or == or !=)
- One token representing all identifiers.
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.



# 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.



# Attributes for tokens

- The token names and associated attribute values for the Fortran statement

$E = M * C ** 2$

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>

Divide the following C++ program:

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

into appropriate lexemes. Which lexemes should get associated lexical values? What should those values be?

Soln :

<float> <id, limitedsquare> <( > <float> <id,x> <)>

<{>

<return> <( > <id,x> <op, <=> <num,-10.0> <op,||> <id,x> <op, >=>

<num, 10.0> <)> <op, ?> <num,100> <:> <id,x> <op, \*> <id,x> <;>

<}>



# 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( x == 0)`

- 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.



# Error recovery

- 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.
- We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left.
- Other possible error-recovery actions are:
  - Delete one character from the remaining input
  - Insert a missing character into the remaining input
  - Replace a character by another character
  - Transpose two adjacent characters

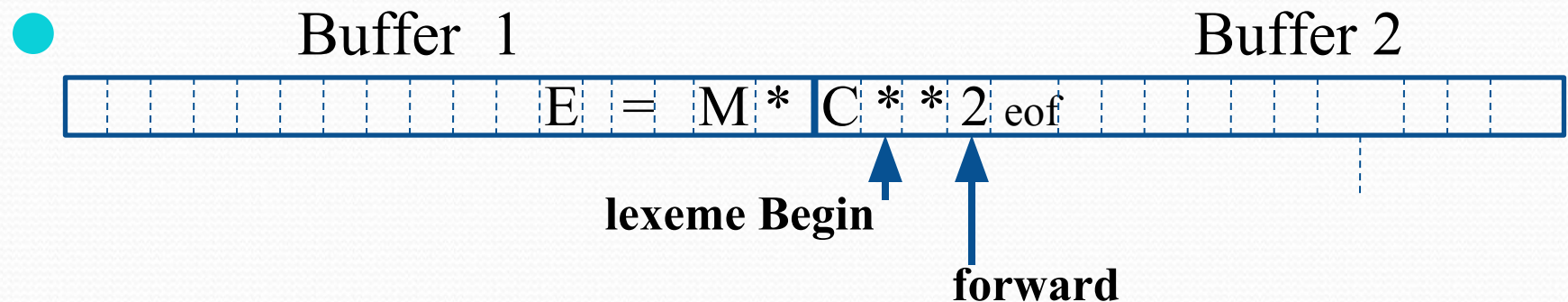
# Input buffering

- Consider task of reading the source program by the LA.
- This task is made difficult 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.
- There are many situations where we need to look at least one additional character ahead.
  - For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id.
  - In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=.
- Thus, we shall introduce a two-buffer scheme that handles large lookaheads safely.



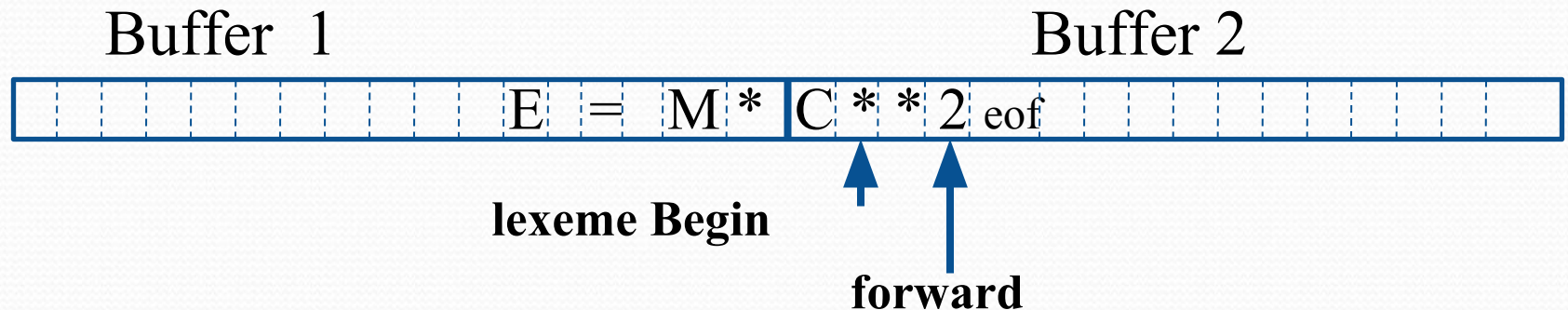
# Input buffering

- 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.
- An important scheme involves two buffers that are alternately reloaded



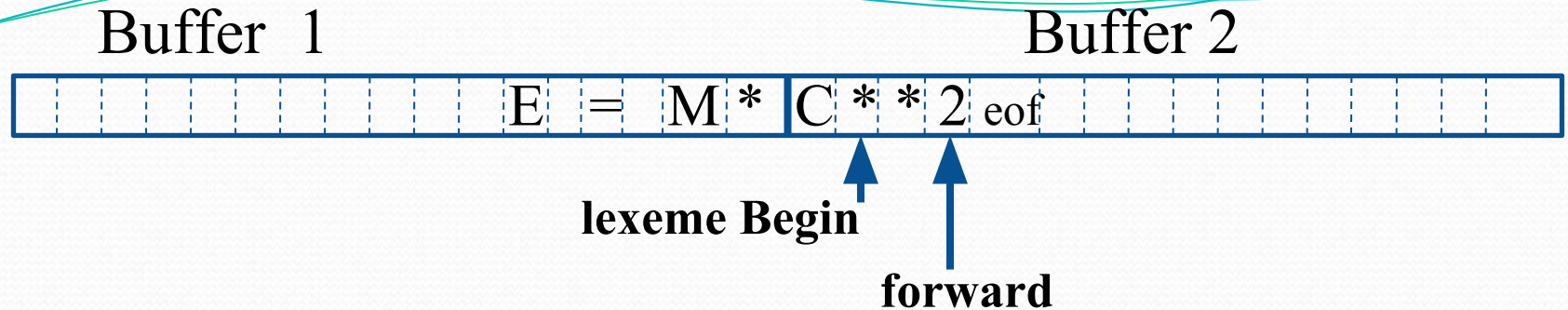


# Input buffering



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

# Input buffering

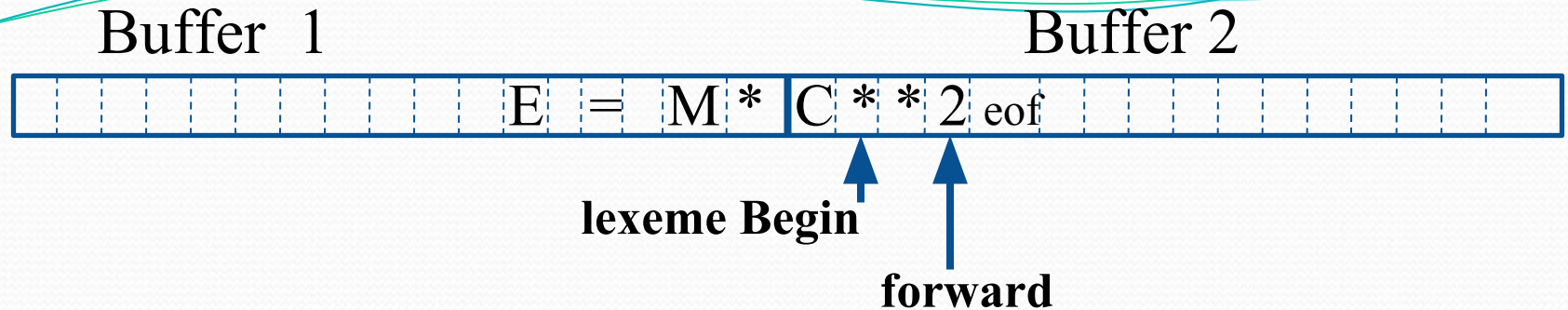


Two pointers to the input are maintained:

- Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
- Pointer **forward** scans ahead until a pattern match is found.
- Once the next lexeme is determined, forward is set to the character at its right end.
- Then, after the lexeme is recorded as an attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found.



# Input buffering



- Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.
- Each time when we advance forward pointer, we must check whether we have not moved off one of the buffers; if we do, then we must also reload the other buffer.
- Thus, for each character read, we make two tests:
  - one for the end of the buffer, and
  - one to determine what character is read.



# Sentinel

- 
- The diagram shows a horizontal buffer divided into two sections: "Buffer 1" and "Buffer 2". The buffer contains the string "E = M eof \* C \* \* 2 eof". The characters are distributed as follows: "E" (1 cell), " " (1 cell), "=" (1 cell), " " (1 cell), "M" (1 cell), " " (1 cell), "eof" (3 cells), " " (1 cell), "\*" (1 cell), " " (1 cell), "C" (1 cell), " " (1 cell), "\*" (1 cell), " " (1 cell), "\*" (1 cell), " " (1 cell), "2" (1 cell), " " (1 cell), "eof" (3 cells). The "lexeme Begin" pointer is at the start of the "C" character. The "forward" pointer is at the start of the second "eof" sequence.

- Any eof that appears other than at the end of a buffer means that the input is at an end.

# Algorithm for advancing forward.

[illegible]

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



# Specification of tokens

- An **alphabet**  $\Sigma$

is a finite set of symbols (characters)

Typical examples of symbols are letters, digits, and punctuation. The set  $\{0,1\}$  is the binary alphabet.

- A **string**  $s$

is a finite sequence of symbols from  $\Sigma$

$|s|$  denotes the length of string  $s$

$\epsilon$  denotes the empty string, thus  $|\epsilon| = 0$

- A **language**

is any countable set of strings over some fixed alphabet. Abstract languages like  $\Phi$ , the empty set, or  $\{\Phi\}$ , the set containing only the empty string, are languages under this definition.

# String Operations

- The concatenation of two strings  $x$  and  $y$  is denoted by  $xy$
- The exponentiation of a string  $s$  is defined by
  - $s^0 = \epsilon$
  - $s^i = s^{i-1}s$  for  $i > 0$       note that  $s \cdot \epsilon = \epsilon \cdot s = s$
- A prefix of string  $s$  is any string obtained by removing zero or more symbols from the end of  $s$ .

For example,  $\text{ban}$ ,  $\text{banana}$ , and  $\epsilon$  are prefixes of  $\text{banana}$ .
- A suffix of string  $s$  is any string obtained by removing zero or more symbols from the beginning of  $s$ .

For example,  $\text{nana}$ ,  $\text{banana}$ , and  $\epsilon$  are suffixes of  $\text{banana}$ .
- A substring of  $s$  is obtained by deleting any prefix and any suffix from  $s$ .

For instance,  $\text{banana}$ ,  $\text{nan}$ , and  $\epsilon$  are substrings of  $\text{banana}$ .



# String Operations

- The proper prefixes, suffixes, and substrings of a string  $s$  are those, prefixes, suffixes, and substrings, respectively, of  $s$  that are not  $\epsilon$  or not equal to  $s$  itself.
- 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`.

# Language Operations

- The most important operations on languages are union, concatenation, and closure.
- Union is the familiar operation on sets.
- 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.
  - 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  $\{\epsilon\}$ , 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,  $\epsilon$  will not be in  $L^+$  unless it is in  $L$  itself.



# Language Operations

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

# Language Operations

- Let  $L$  be the set of letters  $\{A, B, \dots, Z, a, b, \dots, z\}$  and
  - Let  $D$  be the set of digits  $\{0, 1, \dots, 9\}$ .
1.  $L \cup D$   
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  $\epsilon$ , the empty string.
  5.  $L(L \cup D)^*$   
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

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - Letter\_(letter\_ | digit)\*
- Each regular expression is a pattern specifying the form of strings

# Regular expressions

- $\epsilon$  is a regular expression,  $L(\epsilon) = \{\epsilon\}$
- If  $a$  is a symbol in  $\Sigma$  then  $a$  is a regular expression,  $L(a) = \{a\}$  that is, the language with one string, of length one, with  $a$  in its one position.
- Suppose  $r$  and  $s$  are regular expressions denoting languages  $L(r)$  and  $L(s)$ , respectively.
  - $(r) \mid (s)$  is a regular expression denoting the language  $L(r) \cup L(s)$
  - $(r)(s)$  is a regular expression denoting the language  $L(r)L(s)$
  - $(r)^*$  is a regular expression denoting  $(L(r))^*$
  - $(r)$  is a regular expression denoting  $L(r)$
- A language defined by a regular expression is called a **regular set**



# 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

Ex :  $(a) | ((b) * (c))$  by  
 $a | b * c.$

# Regular expressions

Let  $C = \{a, b\}$ .

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  $C$ . 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,  $\{\epsilon, a, aa, aaa, \dots\}$ .
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:  
$$\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}.$$
 Another regular expression for the same language is  $(a^*b^*)^*$ .
5.  $a|a^*b$  denotes the language  $\{a, b, ab, aab, aaab, \dots\}$ , that is, the string  $a$  and all strings consisting of zero or more  $a$ 's and ending in  $b$ .



# Regular expressions

Describe the languages defined by the following regular expression.

1.  $a(a|b)^* a$

Solution : Strings of a's and b's starting and ending with a

2.  $((\epsilon | a)^* b^*)^*$

Solution : Strings of a's and b's

3.  $(a|b)^* a (a|b) (a|b)$

Solution : Strings of a's and b's where third symbol from last is a.

4.  $a^* b a^* b a^* b a^*$

Solution : Strings of a's and b's having only three b's

5.  $(aa|bb)^* (ab|ba) (aa|bb)^* (ab|ba) (aa|bb)^*$

Solution : Strings of a's and b's having even number of a's and b's

# Regular definitions

Give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols.

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



# Regular definitions

- By restricting  $r_i$  to  $C$  and the previously defined  $d$ 's, we avoid recursive definitions.
- We can construct a regular expression over  $\Sigma$  alone, for each  $r_i$ .
- We do so by first replacing uses of  $d_1$  in  $r_2$ , then replacing uses of  $d_1$  and  $d_2$  in  $r_3$  by  $r_1$  and  $r_2$ , and so on.
- Finally, in  $r_n$  we replace each  $d_i$ , for  $i = 1, 2, \dots, n - 1$ , by the substituted version of  $r_i$ , each of which has only symbols of  $\Sigma$ .
- Example : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifier.

letter\_  $\rightarrow A|B|\dots\dots\dots|Z|a|b|\dots\dots\dots|z|_$

digit  $\rightarrow 0|1|2|\dots\dots\dots|9|$

id  $\rightarrow$  letter\_

$(\text{letter\_}| \text{digit})^*$

# Regular definitions

Write the regular definition for the unsigned number

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

number  $\rightarrow$  digits optionalFraction optionalExponent

digit  $\rightarrow$  0 | 1 | ..... | 9

digits  $\rightarrow$  digit digit\*

optionalFraction  $\rightarrow$  . digits |  $\epsilon$

optionalExponent  $\rightarrow$  ( E ( + | - |  $\epsilon$ ) digits ) |  $\epsilon$



# Extensions

Many extensions have been added to regular expressions to enhance their ability to specify string patterns. Some of them are :

- **One or more instances:**  $(r)^+$  : 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  $(L(r))^+$
- **Zero or one instances:**  $r?$  : The unary postfix operator  $?$  means "zero or one occurrence." That is,  $r?$  is equivalent to  $r|\epsilon$ .
- **Character classes:** A regular expression  $a_1|a_2|a_3\dots|a_n$ , where the  $a_i$ 's are each symbols of the alphabet, can be replaced by the shorthand  $[a_1,a_2 \dots a_n]$ . When  $a_1, a_2, \dots, a_n$  form a logical sequence, e.g., consecutive uppercase letters, lowercase letters, or digits, we can replace them by  $[a_1-a_n]$ .
- **Example:**  
letter\_  $\rightarrow [A-Za-z\_]$   
digit  $\rightarrow [0-9]$                       id  $\rightarrow \text{letter\_}(\text{letter\_}|\text{digit})^*$

# Regular definitions

Write the regular definition for the unsigned number

- Solution :
- Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4.
- The regular definition
$$\begin{aligned}\text{digit} &\rightarrow [0-9] \\ \text{digits} &\rightarrow \text{digit}^+ \\ \text{number} &\rightarrow \text{digits} ( . \text{digits} ) ? ( E [+ - ] ? \text{digits} ) ?\end{aligned}$$



# Regular definitions

Write the regular definition for all strings in which the letters are in ascending lexicographic order.

Solution :

Regular definition:

$\text{str} \rightarrow a^* b^* c^* \dots z^*$

# Regular definitions

Write the regular definition for all strings of lowercase letters that contain 5 vowels in order.

Solution : Five vowels are a,e,i,o and u

Regular definition:

Cons -> [bcd fghjklmnpqrstvwxyz]

str -> Cons<sup>\*</sup> a (Cons|a)<sup>\*</sup> e (Cons|e)<sup>\*</sup> i (Cons|i)<sup>\*</sup> o (Cons|o)<sup>\*</sup>  
u (Cons|u)<sup>\*</sup>



# Regular definitions

Since SQL is case insensitive, a keyword can be written either in lowercase or in uppercase, or in any mixture of cases. Show how to write a regular expression for the keywords “select” and “except”

Solution :

Regular definition:

Select -> [Ss][Ee][Ll][Ee][Cc][Tt]

Except -> [Ee][Xx][Cc][Ee][Pp][Tt]

# Recognition of tokens

- We learnt how to express patterns using regular expressions.
- Now, we must 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 if then else statement in pascal :
- stmt -> **if** expr **then** stmt  
      | **if** expr **then** stmt **else** stmt  
      |  $\epsilon$
- expr -> term relop term  
      | term
- term -> id  
      | number

A grammar for branching statements



# Recognition of tokens (cont.)

- The terminals of the grammar, if, then, else, relop, id, and number, are discovered by the lexical analyzer as tokens.
- The patterns for these tokens are described using regular definitions:

*digit* -> [0-9]

*Digits* ->  $\text{digit}^+$

*number* ->  $\text{Digits}(\text{.Digits})? (\text{E}[+ -]? \text{Digits})?$

*letter* -> [A-Za-z]

*id* ->  $\text{letter}(\text{letter}|\text{digit})^*$

*If* -> if

*Then* -> then

*Else* -> else

*Relop* ->  $< | > | <= | >= | = | <>$

- We also need to handle whitespaces:

*ws* ->  $(\text{blank} | \text{tab} | \text{newline})^+$

# Recognition of tokens (cont.)

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

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	—	—
<b>if</b>	<b>if</b>	—
<b>then</b>	<b>then</b>	—
<b>else</b>	<b>else</b>	—
Any <i>id</i>	<b>id</b>	Pointer to table entry
Any <i>number</i>	<b>number</b>	Pointer to table entry
<b>&lt;</b>	<b>relop</b>	LT
<b>&lt;=</b>	<b>relop</b>	LE
<b>=</b>	<b>relop</b>	EQ
<b>&lt;&gt;</b>	<b>relop</b>	NE
<b>&gt;</b>	<b>relop</b>	GT
<b>&gt;=</b>	<b>relop</b>	GE

Figure 3.12: Tokens, their patterns, and attribute values



# Transition diagrams

- An intermediate step in the construction of a lexical analyzer is the construction of "transition diagram" .
- We perform the conversion from regular-expression patterns to transition diagrams by hand.
- 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.
- Edges are directed from one state of the transition diagram to another.
- Each edge is labeled by a symbol or set of symbols.
- If we are in some state  $s$ , and the next input symbol is  $a$ , we look for an edge out of state  $s$  labeled by  $a$ .

# Transition diagrams

- If we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads.
- We shall assume that all our transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels.
- Some important conventions about transition diagrams are:
  - 1. 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 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.



# Transition diagrams

- 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
- 3. One state is designated as 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.

# Transition Diagrams

- Transition diagram for the token **relop** (Pascal)
  - 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.
- 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.s



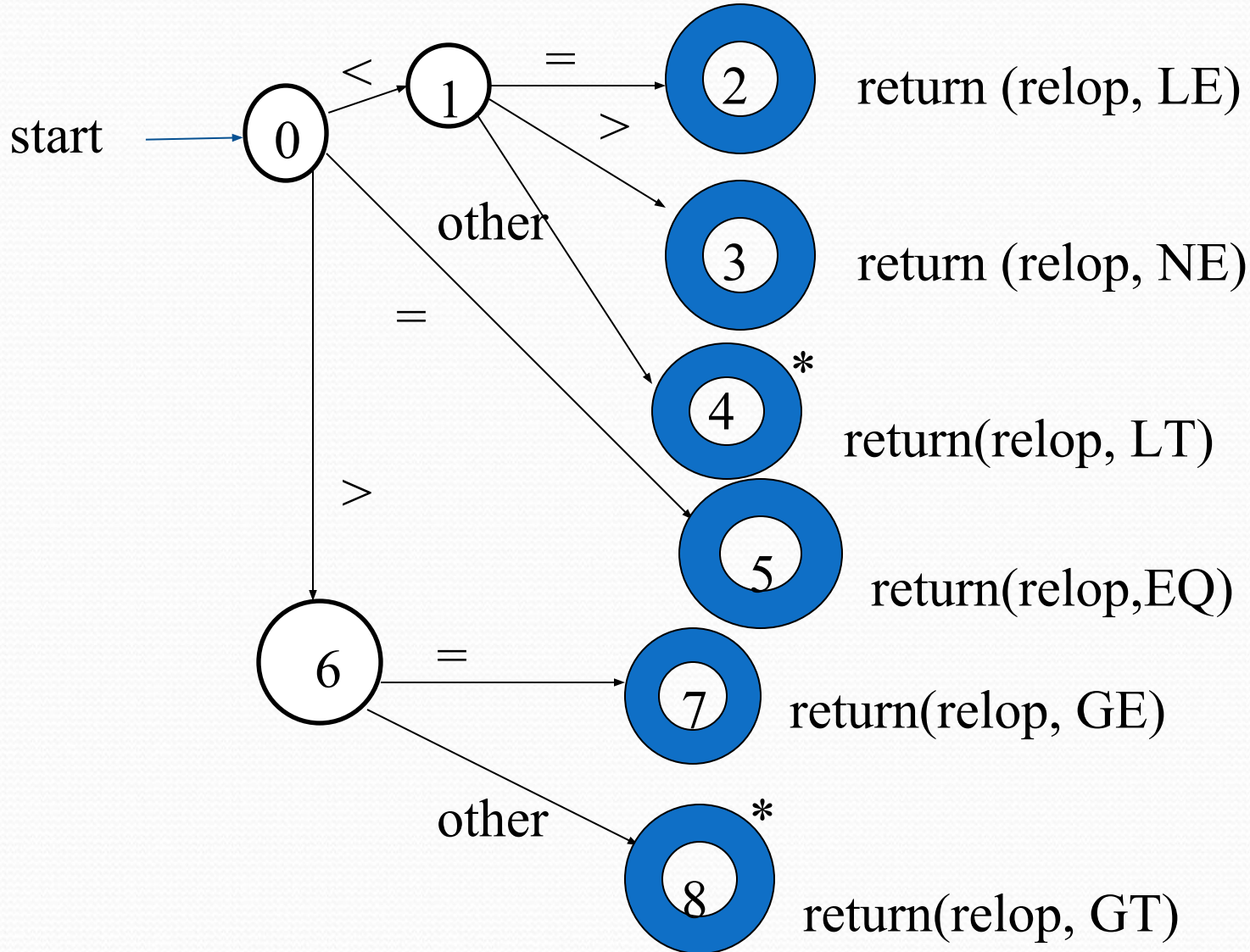
# Transition Diagrams

- Transition diagram for the token **relop**
- 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 C, =, or >, we can not possibly be seeing* a relop lexeme, so this transition diagram will not be used

# Transition Diagrams

- Transition diagram for the token

*Relop* -> < | > | <= | >= | = | <>

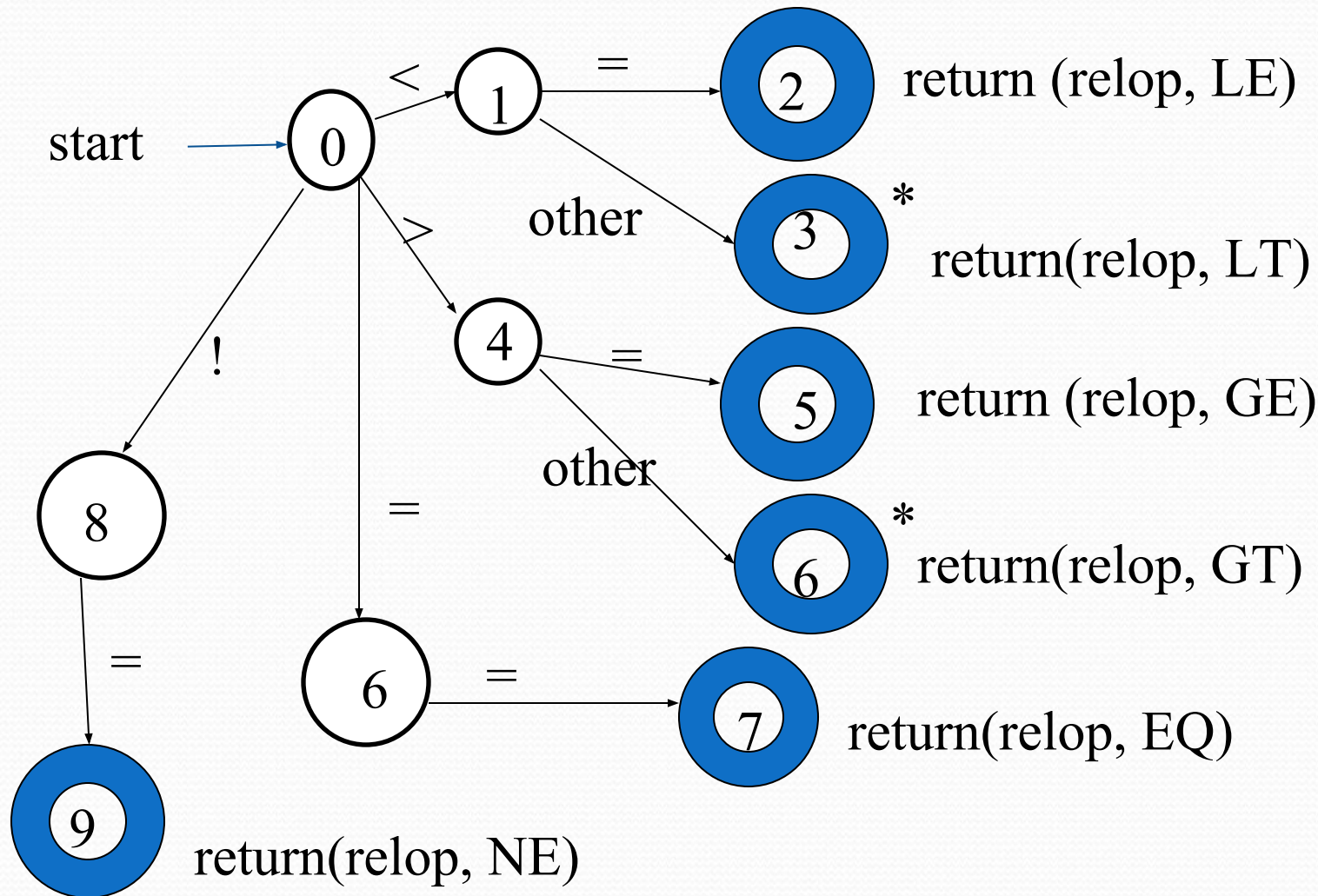




# Transition Diagrams

- Transition diagram for the token **relop** of a C language

*Relop* -> < | > | <= | >= | == | !=



# Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers
- Recognizing keywords and identifiers presents a problem.
- Usually, keywords like `if` or `then` are reserved, so they are not identifiers even though they look like identifiers.
- The use a transition diagram to search for identifier lexemes, also recognize the keywords `if`, `then`, and `else`.
- 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.
    - 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.



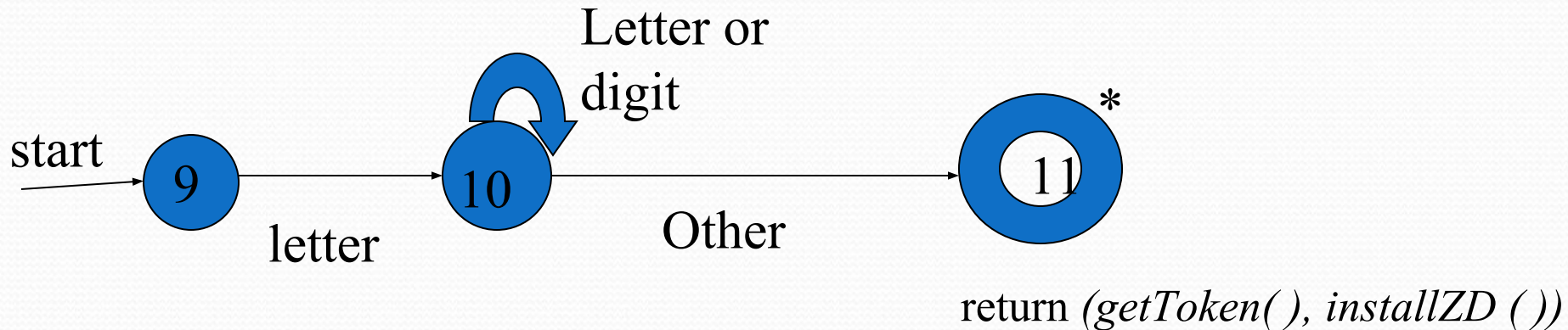
# Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers
  - Any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id.
  - The function *getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents - either id or one of the keyword tokens that was initially installed in the table.

# Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers

*id*  $\rightarrow$  letter ( letter | digit ) \*





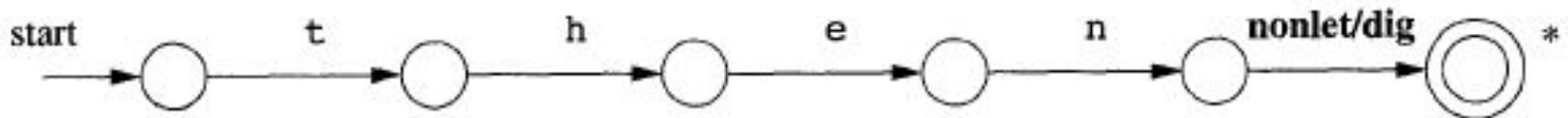
# Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers

2. Create separate transition diagrams for each keyword;

**Note that 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.

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 thenextvalue that has then as a proper prefix. 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.



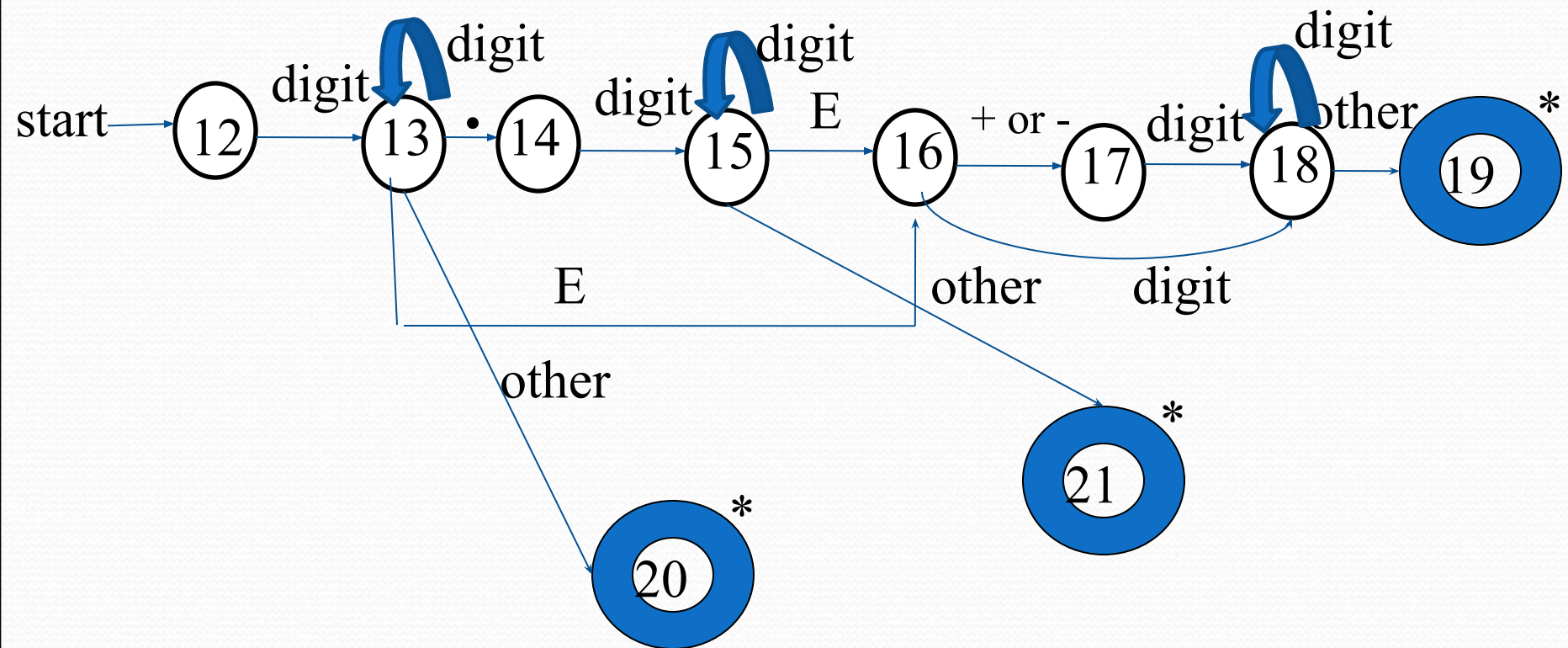
# Transition diagrams (cont.)

- Transition diagram for unsigned numbers
  - The transition diagram for token **number** is shown in Fig.
  - 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.
  - If we see anything but a digit or a dot, 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.
- 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.



# Transition diagrams

- Transition diagram for unsigned numbers



*digit*  $\rightarrow [0-9]$

*Digits*  $\rightarrow \text{digit}^+$

*number*  $\rightarrow \text{Digits}(\text{.Digits})? (\text{E}[+-]? \text{Digits})?$

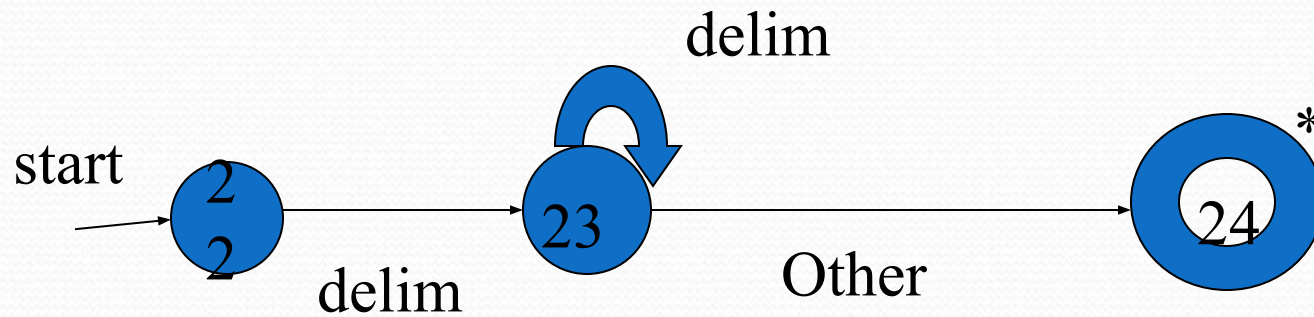
# Transition diagrams (cont.)

- Transition diagram for whitespace
- The 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.
- 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



# Transition diagrams (cont.)

- Transition diagram for whitespace



$ws \rightarrow (\text{blank} \mid \text{tab} \mid \text{newline})^+$

# Writing code for recognizing a token

- 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.



# Writing code for recognizing a token- **Relop**

- `getRelop()`, a C++ function whose job is to simulate the transition diagram of **relop** and **return an object** of type **TOKEN**, that is, a pair consisting of the token name and an attribute value
- `getRelop()` first creates a new object `retToken` and initializes its first component to **RELOP**, the symbolic code for token **relop**.
- We see the typical behavior of a state in case 0, the case where the current state is 0.
- A function `nextchar()` obtains the next character from the input and assigns it to local variable `c`.
- We then check `c` for the three characters we expect to find, making the state transition dictated by the transition diagram

**For example, if the next input character is =, we go to state 5.**

# Writing code for recognizing a token- Relop

- If the next input character is not one that can begin a comparison operator, then a function `fail ()` is called.
- What **`fail ()`** does depends on the **global error recovery** strategy of the lexical analyzer.
- It should reset the forward pointer to `lexemeBegin`, in order to allow another transition diagram to be applied to the true beginning of the unprocessed input.
- It might then change the value of state to be the start state for another transition diagram, which will search for another token.
- Alternatively, if there is no other transition diagram that remains unused, `fail()` could initiate an error-correction phase that will try to repair the input and find a lexeme.

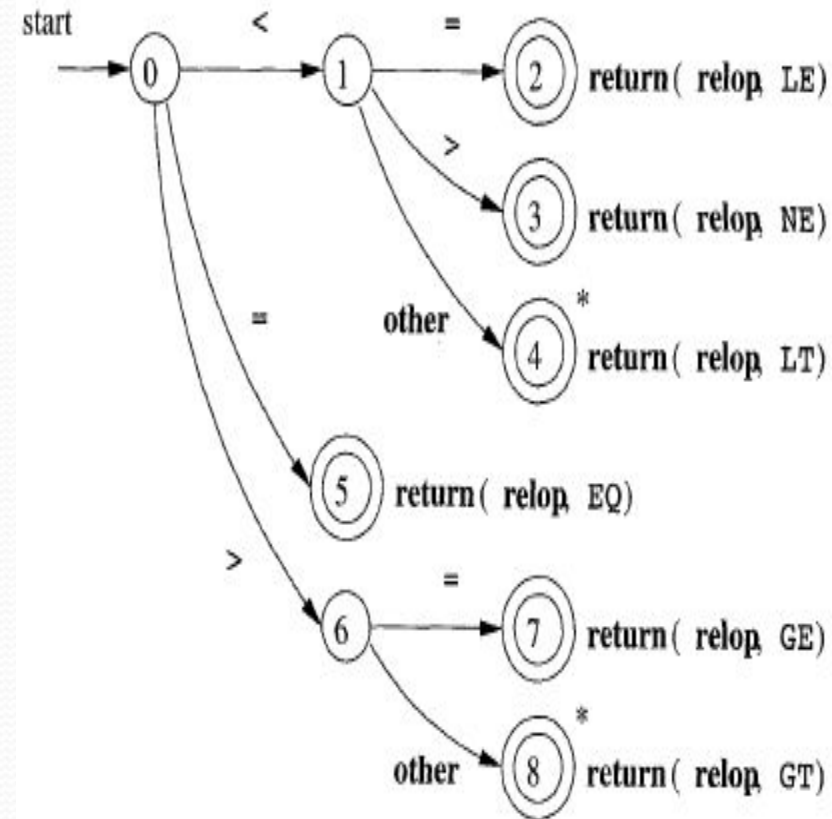


# Writing code for recognizing a token- Relop

- We also show the action for state 8.
- Because state 8 bears a \*, we must retract the input pointer one position That task is accomplished by the function `r e t r a c t ()` .
- Since state 8 represents the recognition of lexeme `>=`, we set the second component of the returned object, which we suppose is named `a t t r i b u t e` , to `GT`, the code for this operator.

# Writing code for recognizing a token- Relop

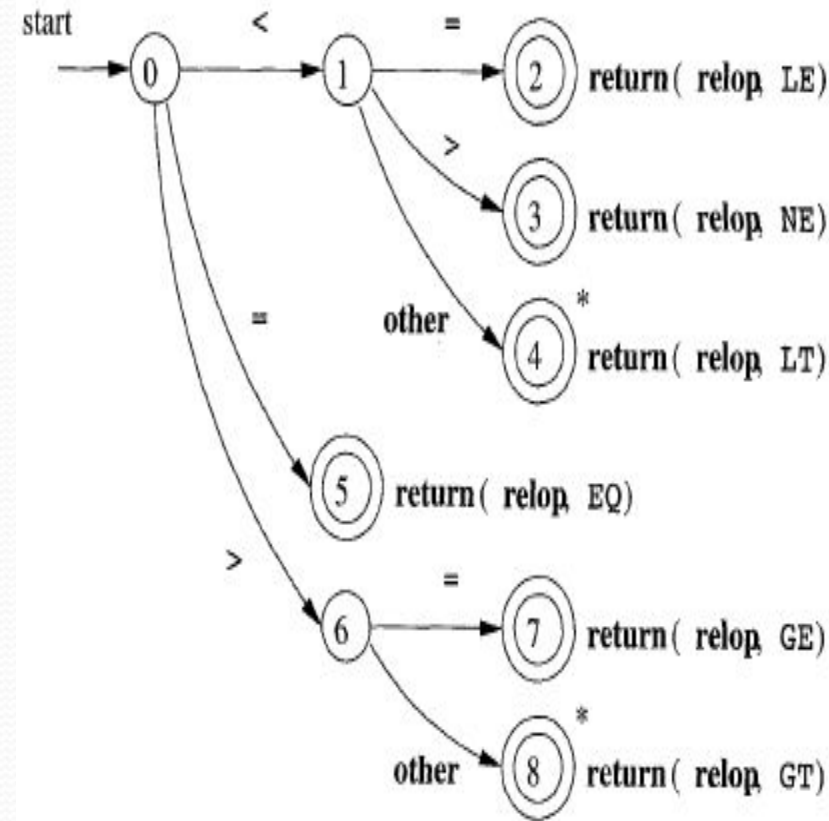
```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) { /* repeat character processing
                until a return or failure occurs */
        switch(state) {
            case 0: c= nextchar();
                    if (c == '<') state = 1;
                    else if (c == '=') state = 5;
                    else if (c == '>') state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: c= nextchar();
                    if (c == '=') state = 2;
                    else if (c == '>') state = 3;
                    else state = 4; break;
        }
    }
}
```





# Writing code for recognizing a token- Relop

```
case 2: reToken.attribute = LE;  
        return(reToken);  
case 3: reToken.attribute = NE;  
        return(reToken);  
case 4: retract();  
        reToken.attribute = LT;  
        return(reToken);  
case 5: reToken.attribute = EQ;  
        return(reToken);  
case 6: c= nextchar();  
        if (c == '=') state = 7;  
        else state = 8; break;  
case 7: reToken.attribute = GE;  
        return(reToken);  
case 8: retract();  
        reToken.attribute = GT; return(reToken); }
```

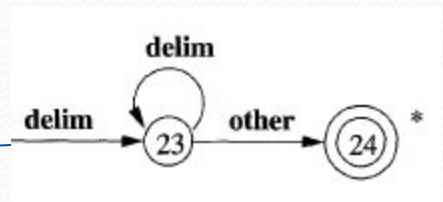
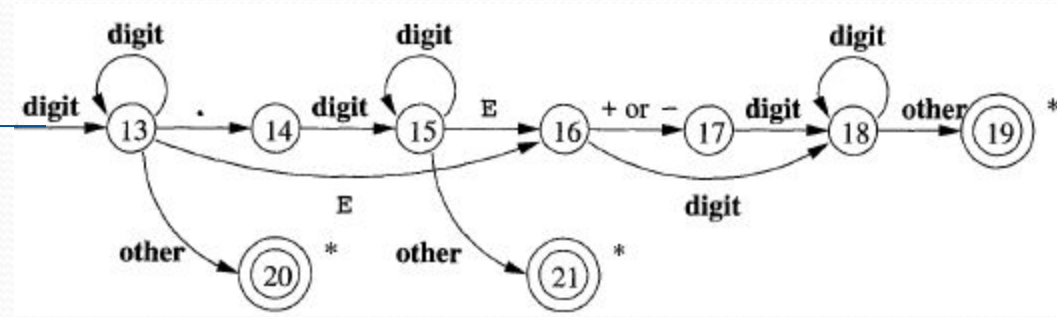
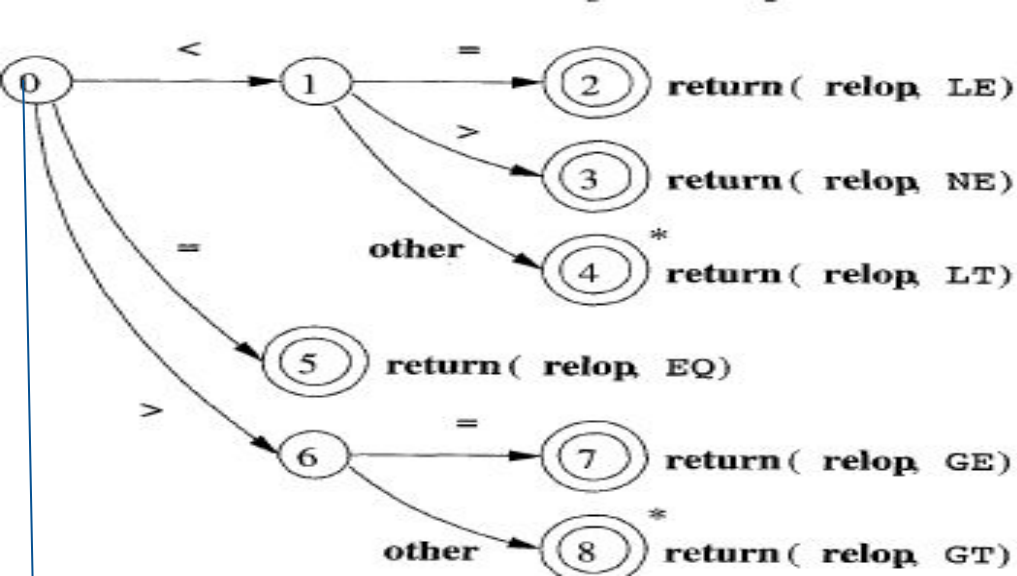


# Architecture of a transition-diagram-based lexical analyzer

There are 3 different ways to use the transition diagrams to build lexical analyzer.

1. We could arrange for the transition diagrams for each token to be tried **sequentially**. Then, the function **fail()** resets the pointer forward and starts the next transition diagram, each time it is called. This method allows us to use transition diagrams for the individual keywords,. We have only to use these before we use the diagram for **id**, **in order for the keywords to be reserved** words.
2. We could run the various transition diagrams **"in parallel,"** feeding the next input character to all of them and allowing each one to make whatever transitions it required. If we use this strategy, we must be careful to resolve the case where one diagram finds a lexeme that matches its pattern, while one or more other diagrams are still able to process input. The normal strategy is to take the longest prefix of the input that matches any pattern. That rule allows us to prefer identifier thenext to keyword then.
3. The preferred approach is to **combine all the transition diagrams into one**. We allow the transition diagram to read input until there is no possible next state, and then take the longest lexeme that matched any pattern. This combination is easy because no two tokens can start with the same character; i.e., the first character immediately tells us which token we are looking for. Thus, we could simply combine states 0, 9, 12, and **22 into one start state, leaving** other transitions intact. However, in general, the problem of combining transition diagrams for several tokens is more complex.





# Assignment 1 :

Write the code for recognizing the following tokens using Transition diagram.

- Unsigned number
- Identifier
- Whitespace
- C language relational operators



# Lab Program 8

- Write a C program to test whether a given identifier is valid or not.

- Write a C program to test whether a given identifier is valid or not.

```
#include<stdio.h>
#include<ctype.h>
void main()
{
char a[20];
int flag,i=1;
printf("\n Enter an identifier:\n");
gets(a);
if(!isalpha(a[0]))
printf("\n Not a valid identifier\n");
else
{
```



# ● Write a C program to test whether a given identifier is valid or not.

```
Else { flag=1;
while(a[i]!='\0')
{
if(!isdigit(a[i])||!isalpha(a[i]))
{ flag=0; break; }
i++;
}
if(flag==1)    printf("\n Valid identifier\n");
else printf("\n Not a Valid identifier\n");
} }
```

/\* Output :

Enter an identifier:

12qa

Not a valid identifier

Enter an identifier:

Azxs

Valid identifier

Enter an identifier:

As123

Valid identifier

Enter an identifier:

asd+

Not a Valid identifier

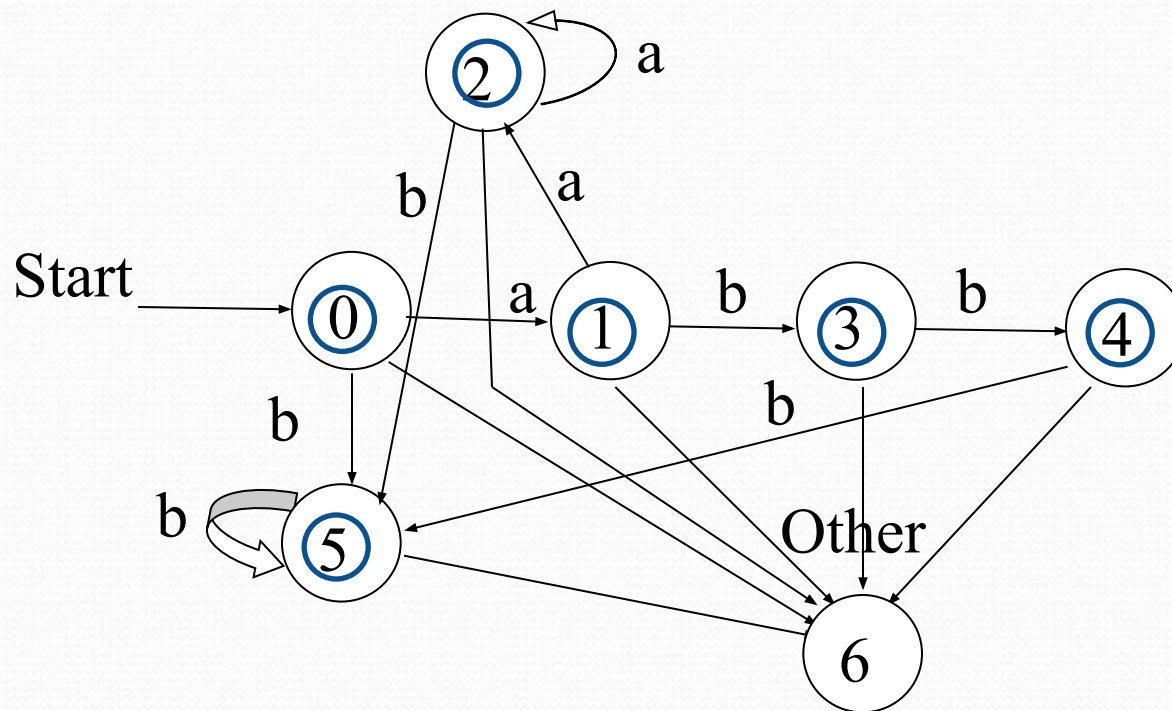
# Lab Program 7

- Write a C program to recognize strings under ' $a^*$ ', ' $a^*b^+$ ', ' $abb$ '.
- Valid strings are :
  - $a^*$  : ,a,aa,aa...a.
  - $a^*b^+$  : b,ab,aab,abbb, aaa..abb.....b
  - $abb$  : abb



● Write a C program to recognize strings under 'a\*', 'a\*b+', 'abb'.

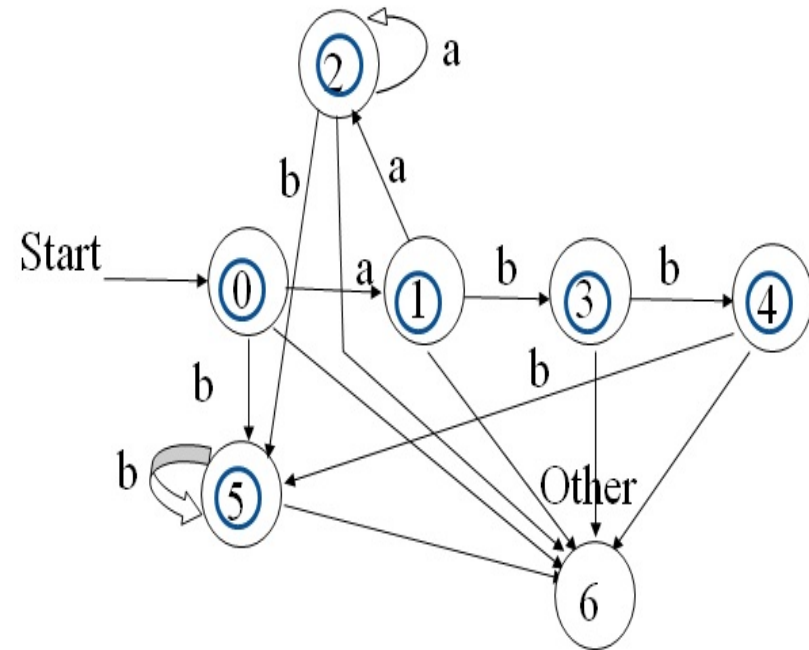
● Transition Diagram :



# Write a C program to recognize strings under 'a\*', 'a\*b+', 'abb'.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
char s[20],c;
int state=0,i=0;
printf("\n Enter a string:");
gets(s);
while(s[i]!='\0')
{
switch(state)
{
case 0: c=s[i++];
        if(c=='a')      state=1;
        else if(c=='b')  state=5;
        else              state=6;
        break;

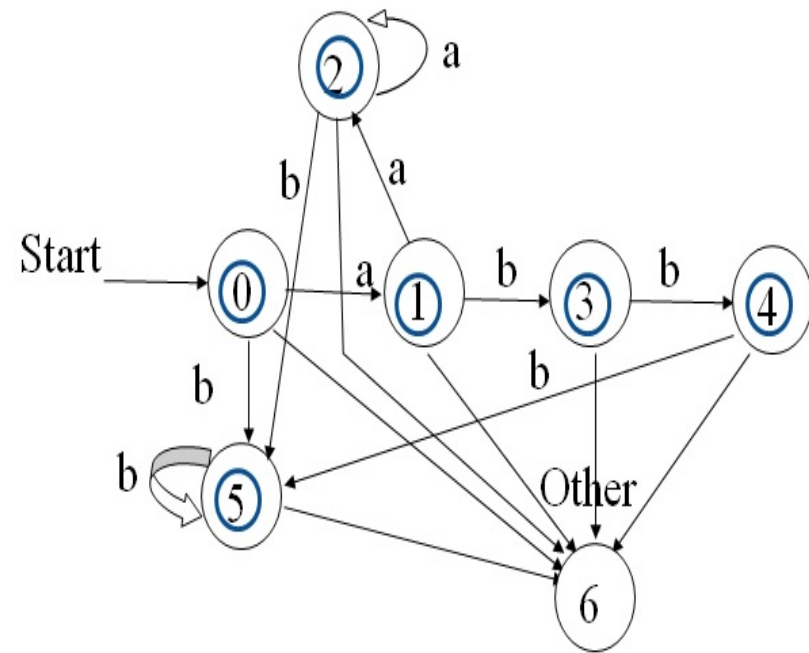
```





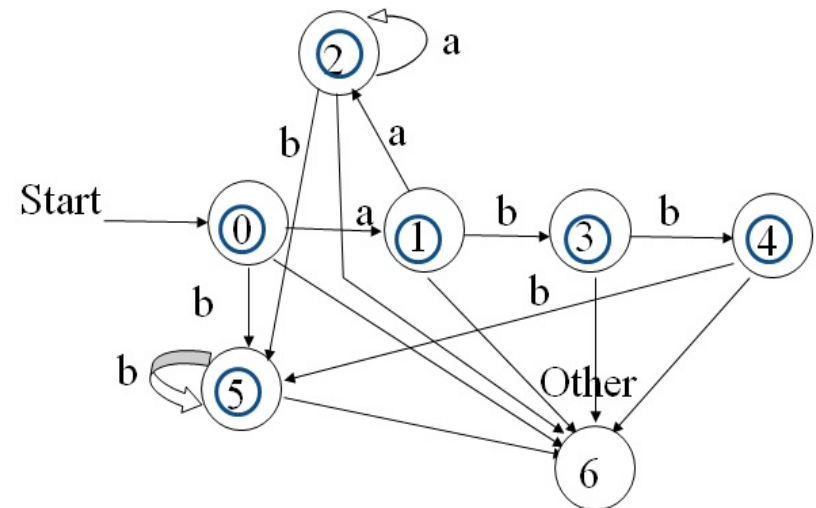
# Write a C program to recognise 'a\*', 'a\*b+', 'abb'.

```
case 1: c=s[i++];  
    if(c=='a') state=2;  
    else if (c=='b') state=3;  
    else state=6; break;  
case 2: c=s[i++];  
    if(c=='a') state=2;  
    else if (c=='b') state=5;  
    else state=6; break;  
case 3: c=s[i++];  
    if (c=='b') state=4;  
    else state=6; break;  
case 4: c=s[i++];  
    if (c=='b') state=5;  
    else state=6; break;  
case 5: c=s[i++];  
    if (c=='b') state=5;  
    else state=6; break;  
case 6: printf("\n %s is not recognised.",s);  
    exit(0);
```



# Write a C program to recognize strings under 'a\*', 'a\*b+', 'abb'.

```
if(state==0 || state==1 || state==2)
printf("\n %s is accepted under rule 'a*'\n",s);
else if(state==3 || state==5)
printf("\n %s is accepted under rule 'a*b+'\n",s);
else if(state==4)
printf("\n %s is accepted under rule 'abb'\n",s);
else
printf("\n %s is not recognised.",s); }
```





# Write a C program to recognize strings under 'a\*', 'a\*b+', 'abb'.

O/P:

Enter a string:

is accepted under rule 'a\*'

Enter a string:a

a is accepted under rule 'a\*'

Enter a string:aaaa

aaaa is accepted under rule 'a\*'

Enter a string:ab

ab is accepted under rule 'a\*b+'

Enter a string:abb

abb is accepted under rule 'abb'

Enter a string:aabb

aabb is accepted under rule 'a\*b+'

Enter a string:aaaabbbb

aaaabbbb is accepted under rule 'a\*b+'

Enter a string:b

b is accepted under rule 'a\*b+'

Enter a string:bb

bb is accepted under rule 'a\*b+'

Enter a string:bbbbbbbbbbbb

bbbbbbbbbbbb is accepted under rule 'a\*b+'

Enter a string:ababa

ababa is not recognised.

Enter a string:ba

ba is not recognised

