Unit II - Syllabus:

Programming language basics - lexical analysis – role of lexical analyzer – input buffering - specification of tokens – recognition of tokens using finite automata. (15 Hrs)

LEXICAL ANALYSIS

A simple way to build lexical analyzer is to construct a diagram that illustrates the structure of the tokens of the source language, and then to hand-translate the diagram into a program for finding tokens. Efficient lexical analyzers can be produced in this manner.

Role of Lexical Analyzer

The lexical analyzer is the first phase of compiler. Its main task is to read the input characters and produces output a sequence of tokens that the parser uses for syntax analysis. As in the figure, upon receiving a "get next token" command from the parser the lexical analyzer reads input characters until it can identify the next token.

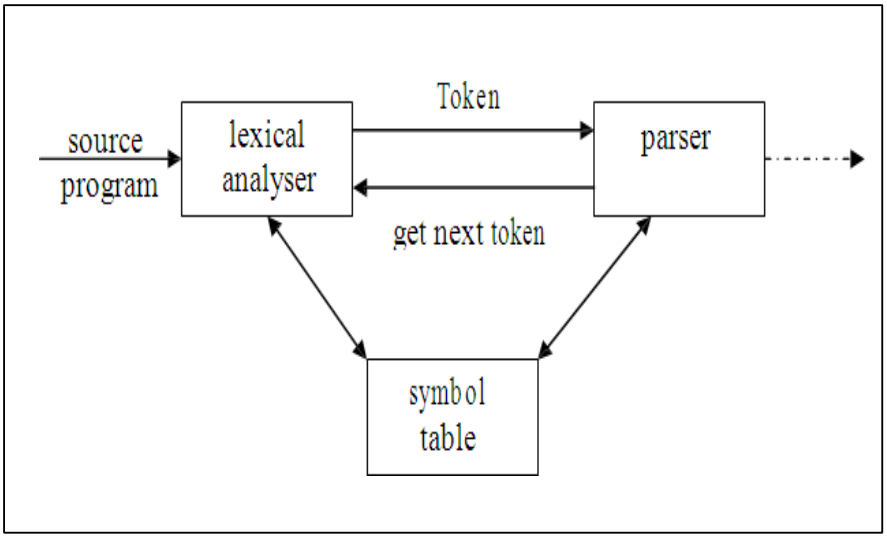


Fig. 1.8 Interaction of lexical analyzer with parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white space in the form of blank, tab, and new line character. Another is correlating error messages from the compiler with the source program.

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing

1 Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.

2 Compiler efficiency is improved: A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task. Specialized buffering technique for reading input character and processing tokens can speed up the performance of compiler.

3 Compiler portability is enhanced: Input alphabet thing and other device specific anomalies can be restricted to the lexical analyzer.

Tokens Patterns and Lexemes.

There are 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. The pattern is set to match each string in the set.

In most programming languages, the following constructs are treated as tokens: keywords, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.

Lexeme

Collection or group of characters forming tokens is called Lexeme. A lexeme is a sequence of characters in the source program that is matched by the pattern for the token. For example in the Pascal's statement const pi = 3.1416; the substring pi is a lexeme for the token identifier.

Patterns

A pattern is a rule describing a set of lexemes that can represent a particular token in source program. The pattern for the token const in the above table is just the single string const that spells out the keyword.

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"

Certain language conventions impact the difficulty of lexical analysis. Languages such as FORTRAN require a certain constructs in fixed positions on the input line. Thus the alignment of a lexeme may be important in determining the correctness of a source program.

Attributes of Token

The lexical analyzer returns to the parser a representation for the token it has found. The representation is an integer code if the token is a simple construct such as a left parenthesis, comma, or colon. The representation is a pair consisting of an integer code and a pointer to a table if the token is a more complex element such as an identifier or constant.

The integer code gives the token type, the pointer points to the value of that token. Pairs are also returned whenever we wish to distinguish between instances of a token.

The Lexical analyzer collects information about tokens and their associated attributes. The tokens influence the parsing decisions and the attributes influence the translation of tokens. As a practical matter, a token usually has only a single attribute, a pointer to the symbol table entry, in which the information about the token is kept; the pointer becomes the attributes for the tokens.

The attributes influence the translation of tokens.

i Constant : value of the constant

ii Identifiers: pointer to the corresponding symbol table entry.

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>

Error Recovery Strategies In Lexical Analysis

Few errors are discernible at the lexical level alone, because a lexical analyzer has a much localized view of the source program. For eg: if the string fi is encountered in a c program, for the first time, the lexical analyzer cannot tell whether 'fi' is a misspelling of the keyword 'if' or if it is an undeclared function identifier:

fi(a == 10)

The following are the error-recovery actions in lexical analysis:

1 Deleting an extraneous character.
2 Inserting a missing character.
3 Replacing an incorrect character by a correct character.
4 Transforming two adjacent characters.
5 **Panic mode recovery**: Deletion of successive characters from the token until error is resolved.

General approaches to the implementation of lexical analyzer

1. Use lexical analyzer generator, such as the lex compiler to produce the lexical analyzer from a regular expression-based specification. In this case, the generator provides routines for reading and buffering the input.
2. Write the lexical analyzer in a conventional system programming language, using the input facilities of that language to read input.
3. Write the lexical analyzer in assembly language and exlicitely manage the reading of input.

1.1 INPUT BUFFERING

The lexical analyzer scans the characters of the source program one at a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for the lexical analyzer to read its input from an input buffer. Fig. 1.9 shows a buffer divided into two halves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the

token is discovered . We view the position of each pointer as being between the character last read and the character next to be read. In practice each buffering scheme adopts one convention either a pointer is at the symbol last read or the symbol it is ready to read.

The distance which the lookahead pointer may have to travel past the actual token may be large. For example, in a PL/I program we may see:

DECLARE (ARG1, ARG2… ARG *n*)

Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file.

Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, if the look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been grouped into tokens. While we can make the buffer larger if we chose or use another buffering scheme, we cannot ignore the fact that overhead is limited.

BUFFER PAIRS

- A buffer is divided into two N-character halves, as shown below

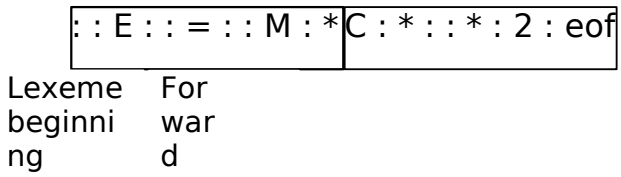


Fig. 1.9 An input buffer in two halves

- Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.
- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.
- Two pointers to the input are maintained:

1 Pointer **lexeme_beginning**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
2 Pointer **forward** scans ahead until a match for a pattern is found.

Once the next lexeme is determined, forward pointer is set to the character at its right end.

- The string of characters between the two pointers is the current lexeme. After the lexeme is recorded as an attribute value of a token returned to the parser, lexeme_beginning is set to the character immediately after the lexeme just found.

Advancing forward pointer:

Advancing forward pointer 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. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

Code to advance forward pointer:

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

Sentinels

- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.
- The sentinel arrangement is as shown below:

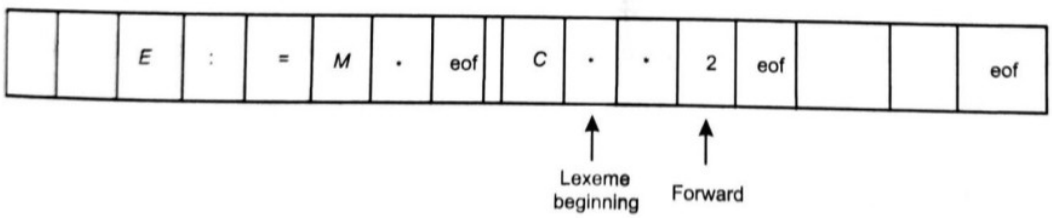


Figure 2.3 Sentinels at end of each buffer half

Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.

Code to advance forward pointer:

```
forward : = forward + 1;
if forward ↑ = 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 beginning of first half
    end
    else /* eof within a buffer signifying end of input */
        terminate lexical analysis
end
```

1.2 SPECIFICATION OF TOKENS

A token is a sequence of character that can be treated as a single local entity. There are 3 specifications of tokens:

1 Strings
2 Language
3 Regular expression

Strings and Languages

- An **alphabet** or character class is a finite set of symbols.

Example

letters and characters
The set {0,1} is the binary alphabet

- A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.
- A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six. The empty string, denoted ε, is the string of length zero.

Operations on strings

The following string-related terms are commonly used:

1 A **prefix** of string s is any string obtained by removing zero or more symbols from the end of string s. For example, ban is a prefix 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 is a suffix of banana.

3 A substring of s is obtained by deleting any prefix and any suffix from s. For example, nan is a substring 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.

Operations on languages:

The following are the operations that can be applied to languages:

1 Union
2 Concatenation
3 Kleene closure
4 Positive closure

The following example shows the operations on strings: Let L={0,1} and S={a,b,c}

1 Union : L U S={0,1,a,b,c}
2 Concatenation : L.S={0a,1a,0b,1b,0c,1c}
3 Kleene closure : L^*={ ε,0,1,00….}
4 Positive closure : L^+={0,1,00….}

Regular Expressions

- Each regular expression r denotes a language L(r).
- Here are the rules that define the regular expressions over some

alphabet Σ and the languages that those expressions denote:

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.
3 Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then, a) (r)|(s) is a regular expression denoting the language L(r) U L(s).
b) (r)(s) is a regular expression denoting the language L(r)L(s). c) (r)* is a regular expression denoting (L(r))*.
d) (r) is a regular expression denoting L(r).
4 The unary operator * has highest precedence and is left associative.
5 Concatenation has second highest precedence and is left associative.
6 | has lowest precedence and is left associative.

Regular set

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.

Algebraic Properties of Regular Expressions

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

Table 2.4 Algebraic properties of regular expression

Axioms	Descriptions
$r\|s = s\|r$	| is commutative
$r\|(s\|t) = (r\|s)\|t$	| is associative
$(rs)t = r(st)$	Concatenation is associative
$r(s\|t) = rs\|rt$ $(s\|t)r = sr\|tr$	Concatenation distributes over |
$\in r = r$ $r\in = r$	$\in$ is the identity elements for concatenation
$r^* = (r\|\in)^*$	Relation between * and $\in$
$r^{**} = r^*$	* is idempotent

Regular Definitions

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
……
$d_n \rightarrow r_n$

1 Each d_i is a distinct name.
2 Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:
letter → A | B | …. | Z | a | b | …. | z | digit
→ 0 | 1 | …. | 9
id → letter (letter | digit) *

Notational Shorthands

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hands for them.

***1 One or more instances (+)*:**

- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L (r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator $^+$ has the same precedence and associativity as the operator *.
- Eg:

a+ = {a,aa,aaa,aaaa…….}

***2 Zero or one instance (?)*:**

- The unary postfix operator ? means “zero or one instance of”.
- The notation r? is a shorthand for r | ε.
- If ‘r’ is a regular expression, then (r)? is a regular expression that denotes the language

***3 Character Classes*:**

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A– Za–z0–9]*

4. **Zero or more instance:**

- The unary operator * means “zero or more instance”.
- Eg: a* = {ε , a,aa,aaa,aaaa…….}

5. The two algebraic identities r* = r+ | ε and r+ = rr* relate the klene and positive closure operators.

-

Non-regular Set

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

1.3 RECOGNITION OF TOKENS

Consider the following grammar fragment:

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

expr → term relop term
| term

term → id
| num

where the terminals if , then, else, relop, id and num generate sets of strings given by the following regular definitions:

if → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter(letter|digit)*
num → digit+ (.digit+)?(E(+|-)?digit+)?

Our lexical analyzer will strip out white space. It does so by comparing a string against the regular definition ws (white space) as shown:

delim →blank | tab | newline

ws → delim+

For this language fragment the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers.

If a match for ws is found, the lexical analyzer does not return a token to the parser. Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute value. The following table shows the regular expression patterns for tokens.

Table 2.5 Regular expression patterns for tokens

Regular expression	Token	Attribute-value
ws	–	–
If	if	–
then	then	–
else	else	–
id	id	Pointer to table entry
num	num	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Finite Automata

A recognizer for a language is a program that takes as input a string x and an answer 'yes' if x is a sentence of the language and no otherwise. We compile a regular expression into a recognizer called a Finite Automaton (Plural Automata). Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.
There are tow types of Finite Automata :

- Non-deterministic Finite Automata (NFA)
- Deterministic Finite Automata (DFA)

Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by

$M = \{S, \Sigma, \delta, S_0, f_n\}$

S – finite set of states

Σ – finite set of input symbols

δ – transition function that maps state-symbol pairs to set of states

S_0 – starting state

f_n – set of final or accepting state

Deterministic Finite Automata

DFA is a special case of a NFA in which
i) no state has an ε-transition.
ii there is at most one transition from each state on same input.

DFA has five tuples denoted by M = {S, Σ, δ, S_0, f_d}
S – finite set of states
Σ – finite set of input symbols
δ – transition function that maps state-symbol pairs to set of states
S_0 – starting state
f_d – set of final or accepting state

Construction of DFA from regular expression

The following steps are involved in the construction of DFA from regular expression:

ii.i Convert RE to NFA using Thomson's rules
ii.ii Convert NFA to DFA
ii.iii Construct minimized DFA

Transition Table :

Transition function(∂) is a function which maps S * ∑ into S . Here 'S' is set of states and '∑' is input of alphabets. To show this transition function we use table called transition table (State Transition Table - STT). The table takes two values a state and a symbol and returns next state.

A transition table gives the information about:

1. Rows represent different states.
2. Columns represent input symbols.
3. Entries represent the different next state.
4. The final state is represented by a star or double circle.
5. The start state is always denoted by an small arrow.

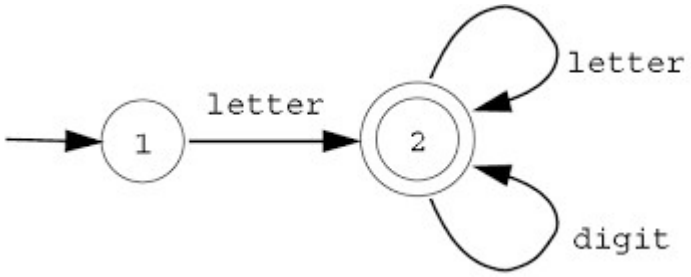
Transition table is a table in which there is a row for each state and a column for each input symbol and ε.

Transition Diagram

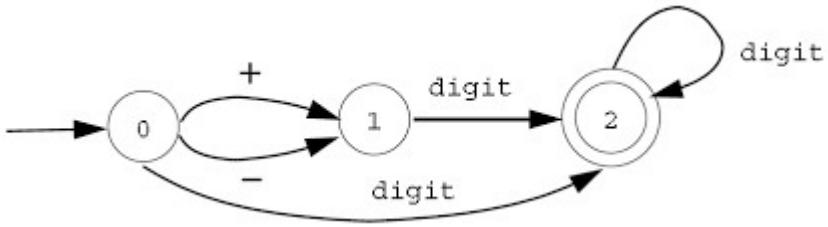
It is a diagrammatic representation to depict the action that will take place when a lexical analyzer is called by the parser to get the next token. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

Transition diagram is a special kind of flowchart for language analysis. In transition diagram the boxes of flowchart are drawn as circle and called as states. States are connected by arrows called as edges. The label or weight on edge indicates the input character that can appear after that state.

Transition diagram keep track of information about characters that are seen as the forward pointer scans the input, by moving from position to position in the diagram as characters are read.



a) Transition diagram for identifier



b. Transition diagram for numeric constant

Transition graph for NFA

An NFA can be represented diagramatically by labelled directed graph called transition graph, in which nodes are states and the labelled edges represent the transition function. This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labelled by special symbol ε as well as input symbols.

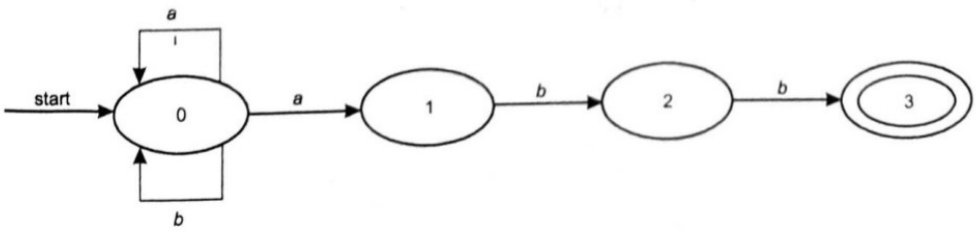


Figure 2.5 A non-deterministic finite automaton

This state transition graph recognizes the input pattern (a | b)* abb. ie. It recognizes aabb, babb, abb, ababb, baabb, etc are valid.

Here,

The set of states is {0,1,2,3}
Input symbol set is {a,b}

State 0 is distinguished as the start symbol.
Accepting state 3 is indicated as doubled circle.

Table 2.6 Transition table for the finite automaton of $(a \mid b)^* abb$

State	Input symbol	
	a	*b*
0	{0,1}	{0}
1		

C code for transition diagram

```
        Token next token(){
while (1) { Switch (state)
{       case 0: c=nextchar();
        If (c==blank || c==tab || c==newline) {
                state=0;
                Lexem_beginning ++ ;}
        Else if (c=='<') state=1;
        Else if (c=='=') state=5;
        Else if (c=='>') state=6;
        Else state=fail ();
        Break;
        ...../*case 1-8 here */
        Case 9: c=nextchar();
                If (isletter(c))
                        state=10;
                Else
                        state= fail ()
                Break;
        Case10: c=nextchar ();
                If (isletter(c))
                        state=10;
                Else if (isdigit(c))
                        state=10;
                Else
                        state=11;
                Break;
```

```
Case 11: retract(1); install_id();
         return (get_token());
...../* cases 12-24 here */
Case 25:  c=nextchar ();
          If (isdigit(c))
                    state=26
          Else
                    state=fail ()
          Break;
Case 26: c=nextchar ();
          If (isdigit(c))
                    state=26
          Else
                    state =27;
          Break;
Case 27: Retract (1);
         Install_num ();
         Return (num) ;
}}}
```

Example

The transition diagram of NFA accepting $aa^*| bb^*$ is shown in Figure 2.9.

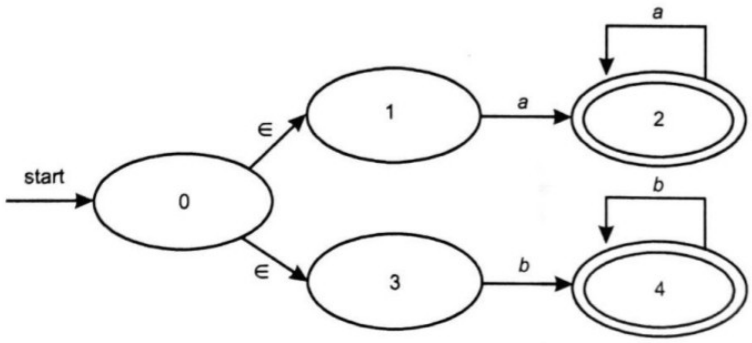


Figure 2.9 NFA accepting $aa^* \mid bb^*$

String *aaa* is accepted by moving through states 0, 1, 2, 2 and 2. The labels of these edges e ∈, *a*, *a* and *a* whose concatenation is *aaa*. Note that ∈'s disappear in a concatenation.

Transition diagram for DFA

The following state transition diagram recognizes the input pattern (a | b)* abb.

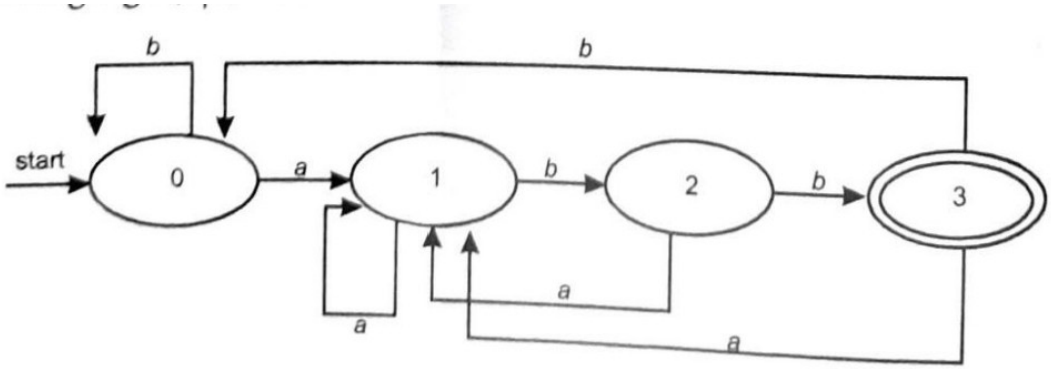


Figure 2.10 DFA accepting $(a/b)^* abb$

It does not contain ε transition from any state. Also it has no two state change from a state for same next input symbol.

1.4 A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

There is a wide range of tools for constructing lexical analyzers.

- ❖ Lex
- ❖ YACC

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

Creating a lexical analyzer

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
- Finally, lex.yy.c is run through the C compiler to produce an object progra m a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

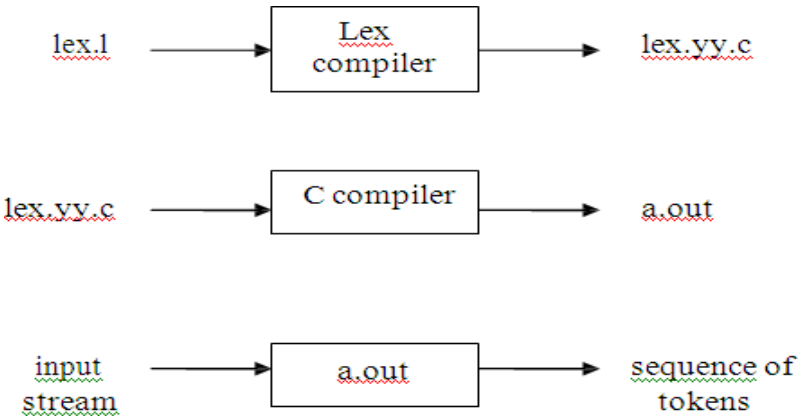


Fig1.11 Creating a lexical analyzer with lex

Lex Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

- **Definitions** include declarations of variables, constants, and regular definitions

- **Rules** are statements of the form

p_1 $\{action_1\}$
p_2 $\{action_2\}$
…
p_n $\{action_n\}$

where p_i is regular expression and $action_i$ describes what action the lexical analyzer should take when pattern p_i matches a lexeme. Actions are written in C code.

- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

YACC- YET ANOTHER COMPILER-COMPILER

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized.

Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

A parser generator is a program that takes as input a specification of a syntax and produces as output a procedure for recognizing that language. Historically, they are also called compiler compilers. YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File: YACC input file is divided into three parts.

```
        /* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```