St. Joseph's College of Engineering
Department of CSE
Assignment-II
CS6660-Compiler Design

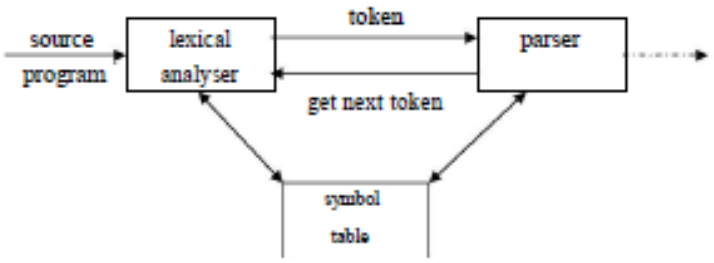
Part-B

1) Explain in detail about the role of Lexical analyzer with the possible error recovery actions. (16)

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lexer often exists as a single function which is called by a parser or another function.

THE ROLE OF THE LEXICAL ANALYZER

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

ISSUES OF LEXICAL ANALYZER

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

TOKENS A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**. A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language: sum=3+2

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
	End of statement

LEXEME: Collection or group of characters forming tokens is called Lexeme.

PATTERN: 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.

Attributes for Tokens Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant : value of the constant

ii) Identifiers: pointer to the corresponding symbol table entry.

ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:

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.

2) Prove that the following two regular expressions are equivalent by showing that minimum state DFA's are same.(16)

i) (a|b)* #

ii) (a*|b*)*#

Refer Text Book

3) a) Describe the specification of tokens and how to recognize the tokens(8)

SPECIFICATION OF TOKENS

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.

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 strings.

Forexample, 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….}

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+)?

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.

Transition diagrams 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.

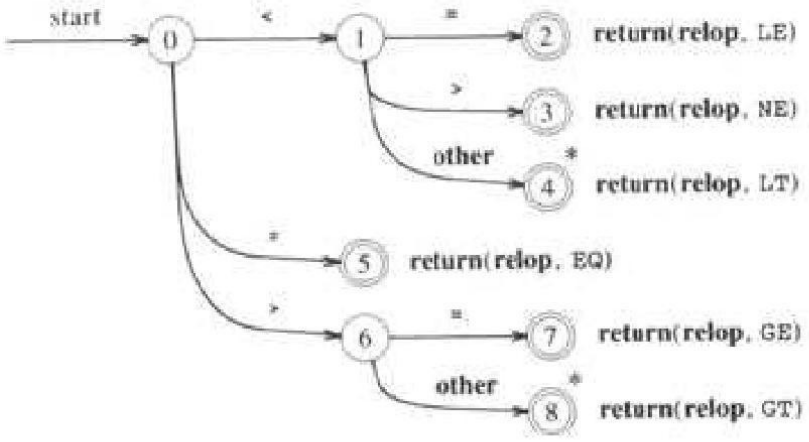
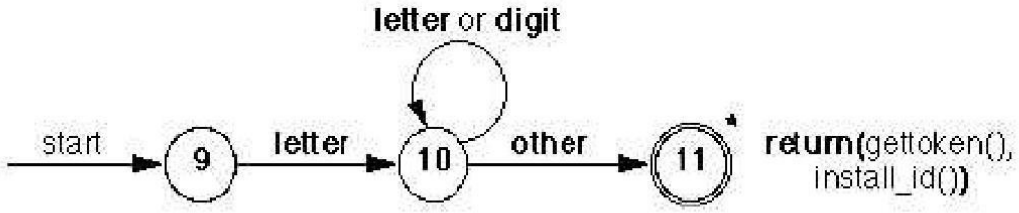


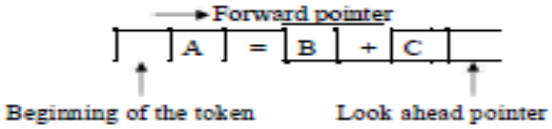
Fig. 3.12. Transition diagram for relational operators.

Transition diagram for identifiers and keywords



b) Explain in detail about input buffering.(8)

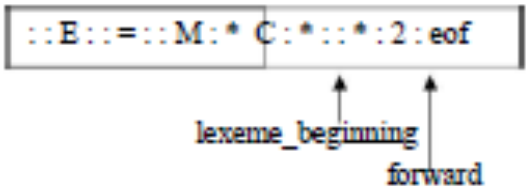
INPUT BUFFERING We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:



We introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

BUFFER PAIRS

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



- • 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 pattern match is found.

Once the next lexeme is determined, forward 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.

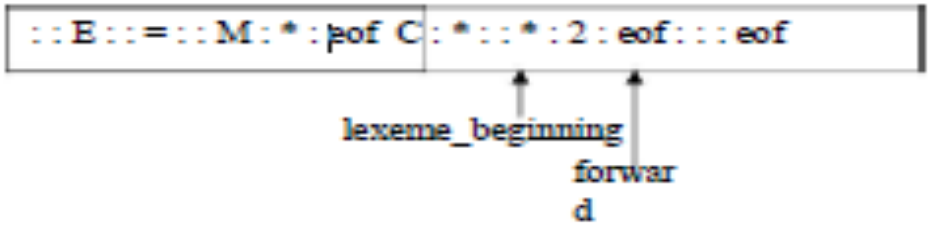
Code to advance forward pointer:

```
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:



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

4) a) Write LEX specifications and necessary C code that reads English words from a text file and response every occurrence of the sub string 'abc' with 'ABC'. The program should also compute number of characters, words and lines read. It should not consider and count any lines(s) that begin with a symbol '#' (12)

Sol:

```
%{
#include<stdio.h>
#include<string.h>
int c=0,lc=0,wc=0;
char *s1="abc",*s2;
char *ptr;
%}
word[^ \t\n]+
%%
(#)(.)*(\n)    {}
{wors}     {s2=strdup(yytext);
           do
                {
                ptr=strstr(s2,s1);
                if(ptr1=null)
                {
                        printf("abc");
                        s2=ptr+strlen(s1)+1;
                }
                }while(ptr!=null);
                cc+=yyleng;
                wc++;
                }
                {cc++;}
\n              {cc++;lc++}
%%
extern file *yyin;
int main (int argc,char**argv)
{
        file *file;
```

```
        void display();
if(argc>1)
{
file=fopen(argv[1],"r")
if(!file)
{
printf("error in opening the file");
exit(0);
}
yyin=file;
}
printf("\n the substring is replaced");
yylex();
display();
return 0;
}
void display()
{
printf("\n character count=%d",cc);
printf("\n line count-%d",lc);
printf("\nword count=%d",wc);
printf("\n");
}
int yywrap()
{
return 1;
}
```

b) Write a short note on LEX.(4)

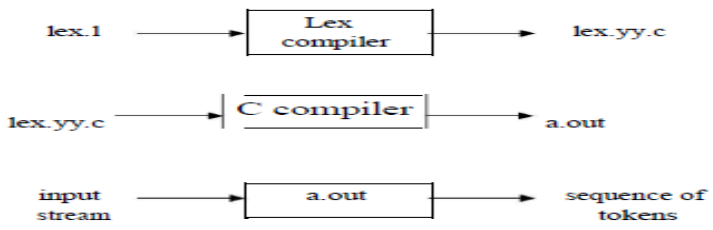
LEX

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 program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



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.