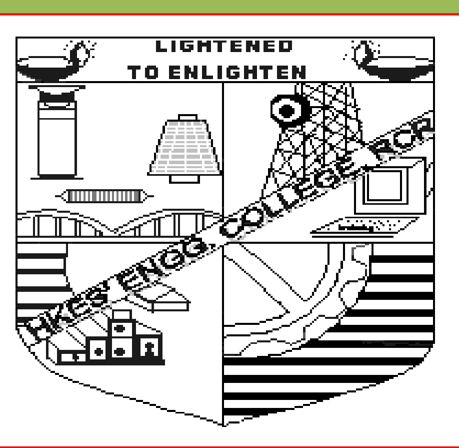
HKE'S SLN COLLEGE OF ENGINEERING

SYSTEM SOFTWARE AND OPERATING SYSTEM

LAB MANUAL

COMMON TO CSE/ISE



10CSL58

PREPARED BY:

Mr.VEERESH BALIGERI

Lecturer, C.S.E. dept.,

S.L.N. College of Engineering,Raichur-584135

SYSTEM SOFTWARE & OPERATING SYSTEM LABORATORY

Subject Code: 10CSL58 **Sem : 5th CSE & ISE**

Part A

Execution of the following programs using LEX:

1. a) Program to count the number of characters, words, spaces and lines in a given input file.

b) Program to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.

2. a) Program to recognize a valid arithmetic expression and torecognize the identifiers and operators present. Print themseparately.

b) Program to recognize whether a given sentence is simple orcompound.

3. Program to recognize and count the number of identifiers in a given input file.

Execution of the following programs using YACC:

4. a) Program to recognize a valid arithmetic expression that uses operators +, -, * and /.

b) Program to recognize a valid variable, which starts with a letter,followed by any number of letters or digits.

5. a) Program to evaluate an arithmetic expression involving operators +, -, * and /.

b) Program to recognize strings 'aaab', 'abbb', 'ab' and 'a' using the grammar (anbn, n>= 0).

6. Program to recognize the grammar (anb, n>= 10).

PART B

Unix Programming:

7. a) Non-recursive shell script that accepts any number ofarguments and prints them in the Reverse order, (For example,if the script is named rargs, then executing rargs A shouldproduce C B A on the standard output).

b) C program that creates a child process to read commands fromthe standard input and execute them (a minimal implementationof a shell – like program). You can assume that no argumentswill be passed to the commands to be executed.

8. a) Shell script that accepts two file names as arguments, checks if the permissions for these files are identical and if the permissions are identical, outputs the common permissions, otherwise outputs each file name followed by its permissions.

b) C program to create a file with 16 bytes of arbitrary data from thebeginning and another 16 bytes of arbitrary data from an offset of 48. Display the file contents todemonstrate how the hole infile is handled.

9. a) Shell script that accepts file names specified as arguments and creates a shell script that contains this file as well as the code to recreate these files. Thus if the script generated by your script is executed, it would recreate the original files(This is same as the “bundle” script described by Brain W. Kernighan and Rob Pikein “ The Unix Programming Environment”, Prentice – Hall India).

b) C program to do the following: Using fork( ) create a child process. The child process prints its own process-id and id of its parent and then exits. The parent process waits for its child tofinish (by executing the wait( )) and prints its own process-id and the id of its child process and then exits.

Operating Systems:

10. Design, develop and execute a program in C / C++ to simulate the working of Shortest Remaining Time and Round-Robin Scheduling Algorithms. Experiment with different quantum sizes for the Round-Robin algorithm. In all cases, determine the average turn-around time. The input can be read from key board or from a file.

11. Using OpenMP, Design, develop and run a multi-threaded program to generate and print Fibonacci Series. One thread has to generate the numbers up to the specified limit and another thread has to print them. Ensure proper synchronization.

12. Design, develop and run a program to implement the Banker’s Algorithm. Demonstrate its working with different data values.

Lex

Introduction:

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

The user supplies the additional code beyond expression matching needed to complete his tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the user's program fragments. Thus, a high level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often inappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called ``host languages.'' Just as general purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C, although Fortran (in the form of Ratfor [2] has been available in the past. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere the appropriate compilers exist.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

Source -> | Lex | -> yylex

Input -> | yylex | -> Output

Figure 1: An overview of Lex

Using the regular expressions, we can write LEX programs and generate various tokens. And the use of the regular expressions eases the specification of patterns. The language that we use to describe a particular pattern is called “Meta Language”. The characters that are used in this meta – language are called meta characters (Usually ASCII characters).

LIGHTENED TO ENLIGHTEN

Regular Expressions in LEX:

A regular expression is a pattern description using a meta language. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in LEX. The following tables define some of the symbols used in LEX.

<table>
<tr><th colspan="3">Characters</th></tr>
<tr><th>Character</th><th>Description</th><th>Example</th></tr>
<tr><td>Any character except [\^$.|?*+()</td><td>All characters except the listed special characters match a single instance of themselves. { and } are literal characters, unless they're part of a valid regular expression token (e.g. the {n} quantifier).</td><td>a matches a</td></tr>
<tr><td>\ (backslash) followed by any of [\^$.|?*+(){}</td><td>A backslash escapes special characters to suppress their special meaning.</td><td>\+ matches +</td></tr>
<tr><td>\Q...\E</td><td>Matches the characters between \Q and \E literally, suppressing the meaning of special characters.</td><td>\Q+-*/\E matches +-*/</td></tr>
<tr><td>\xFF where FF are 2 hexadecimal digits</td><td>Matches the character with the specified ASCII/ANSI value, which depends on the code page used. Can be used in character classes.</td><td>\xA9 matches © when using the Latin-1 code page.</td></tr>
<tr><td>\n, \r and \t</td><td>Match an LF character, CR character and a tab character respectively. Can be used in character classes.</td><td>\r\n matches a DOS/Windows CRLF line break.</td></tr>
<tr><td>\a, \e, \f and \v</td><td>Match a bell character (\x07), escape character (\x1B), form feed (\x0C) and vertical tab (\x0B) respectively. Can be used in character classes.</td><td></td></tr>
</table>

\cA through \cZ	Match an ASCII character Control+A through Control+Z, equivalent to \x01 through \x1A. Can be used in character classes.	\cM\cJ matches a DOS/Windows CRLF line break.
Dot		
Character	**Description**	**Example**
. (dot)	Matches any single character except line break characters \r and \n. Most regex flavors have an option to make the dot match line break characters too.	. matches x or (almost) any other character

Character Classes or Character Sets [abc]		
Character	**Description**	**Example**
[(opening square bracket)	Starts a character class. A character class matches a single character out of all the possibilities offered by the character class. Inside a character class, different rules apply. The rules in this section are only valid inside character classes. The rules outside this section are not valid in character classes, except \n, \r, \t and \xFF	
Any character except ^-]\ add that character to the possible matches for the character class.	All characters except the listed special characters.	[abc] matches a, b or c
\ (backslash) followed by any of ^-]\	A backslash escapes special characters to suppress their special meaning.	[\^\]] matches ^ or]
- (hyphen) except immediately after the opening [	Specifies a range of characters. (Specifies a hyphen if placed immediately after the opening [)	[a-zA-Z0-9] matches any letter or digit
^ (caret) immediately after the opening [	Negates the character class, causing it to match a single character *not* listed in the character class. (Specifies a caret if placed anywhere except after the opening [)	[^a-d] matches x (any character except a, b, c or d)
\d, \w and \s	Shorthand character classes matching digits 0-9, word characters (letters and digits) and whitespace respectively. Can be used inside and outside character classes.	[\d\s] matches a character that is a digit or whitespace
\D, \W and \S	Negated versions of the above. Should be used only outside character classes. (Can be used inside, but that is confusing.)	\D matches a character that is not a digit
[\b]	Inside a character class, \b is a backspace character.	[\b\t] matches a backspace or tab

		character
Alternation		
Character	**Description**	**Example**
| (pipe)	Causes the regex engine to match either the part on the left side, or the part on the right side. Can be strung together into a series of options.	abc|def|xyz matches abc, def or xyz
| (pipe)	The pipe has the lowest precedence of all operators. Use grouping to alternate only part of the regular expression.	abc(def|xyz) matches abcdef or abcxyz
Anchors		
Character	**Description**	**Example**
^ (caret)	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the caret match after line breaks (i.e. at the start of a line in a file) as well.	^. matches a in abc\ndef. Also matches d in "multi-line" mode.
$ (dollar)	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Most regex flavors have an option to make the dollar match before line breaks (i.e. at the end of a line in a file) as well. Also matches before the very last line break if the string ends with a line break.	.$ matches f in abc\ndef. Also matches c in "multi-line" mode.
\A	Matches at the start of the string the regex pattern is applied to. Matches a position rather than a character. Never matches after line breaks.	\A. matches a in abc
\Z	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks, except for the very last line break if the string ends with a line break.	.\Z matches f in abc\ndef
\z	Matches at the end of the string the regex pattern is applied to. Matches a position rather than a character. Never matches before line breaks.	.\z matches f in abc\ndef

Word Boundaries		
Character	**Description**	**Example**
\b	Matches at the position between a word character (anything matched by \w) and a non-word character (anything matched by [^\w] or \W) as well as at the start and/or end of the string if the first and/or last characters in the string are word characters.	.\b matches c in abc
\B	Matches at the position between two word characters (i.e the position between \w\w) as well as at the position between two non-word characters (i.e. \W\W).	\B.\B matches b in abc

Quantifiers		
Character	**Description**	**Example**
? (question mark)	Makes the preceding item optional. Greedy, so the optional item is included in the match if possible.	abc? matches ab or abc
??	Makes the preceding item optional. Lazy, so the optional item is excluded in the match if possible. This construct is often excluded from documentation because of its limited use.	abc?? matches ab or abc
* (star)	Repeats the previous item zero or more times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is not matched at all.	".*" matches "def" "ghi" in abc "def" "ghi" jkl
*? (lazy star)	Repeats the previous item zero or more times. Lazy, so the engine first attempts to skip the previous item, before trying permutations with ever increasing matches of the preceding item.	".*?" matches "def" in abc "def" "ghi" jkl
+ (plus)	Repeats the previous item once or more. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only once.	".+" matches "def" "ghi" in abc "def" "ghi" jkl
+? (lazy plus)	Repeats the previous item once or more. Lazy, so the engine first matches the previous item only once, before trying permutations with ever increasing matches of the preceding item.	".+?" matches "def" in abc "def" "ghi" jkl

{n} where n is an integer >= 1	Repeats the previous item exactly n times.	a{3} matches aaa
{n,m} where n >= 1 and m >= n	Repeats the previous item between n and m times. Greedy, so repeating m times is tried before reducing the repetition to n times.	a{2,4} matches aa, aaa or aaaa
{n,m}? where n >= 1 and m >= n	Repeats the previous item between n and m times. Lazy, so repeating n times is tried before increasing the repetition to m times.	a{2,4}? matches aaaa, aaa or aa
{n,} where n >= 1	Repeats the previous item at least n times. Greedy, so as many items as possible will be matched before trying permutations with less matches of the preceding item, up to the point where the preceding item is matched only n times.	a{2,} matches aaaaa in aaaaa
{n,}? where n >= 1	Repeats the previous item between n and m times. Lazy, so the engine first matches the previous item n times, before trying permutations with ever increasing matches of the preceding item.	a{2,}? matches aa in aaaaa

Advanced Lex:

Lex has several functions and variables that provide different information and can be used to build programs that can perform complex functions. Some of these variables and functions, along with their uses, are listed in the following tables.

Lex variables:

yyin	Of the type FILE*. This points to the current file being parsed by the lexer.
yyout	Of the type FILE*. This points to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
yytext	The text of the matched pattern is stored in this variable (char*).
yyleng	Gives the length of the matched pattern.
yylval	An integer value associated with the token is returned by lexical analyzer.
yylineno	Provides current line number information. (May or may not be supported by the lexer.)

Lex functions:

yylex()	The function that starts the analysis. It is automatically generated by Lex.
yywrap()	This function is called when end of file (or input) is encountered. If this function returns 1, the parsing stops. So, this can be used to parse multiple files. Code can be written in the third section, which will allow multiple files to be parsed. The strategy is to make yyin file pointer (see the preceding table) point to a different

	file until all the files are parsed. At the end, yywrap() can return 1 to indicate end of parsing.
yyless(int n)	This function can be used to push back all but first 'n' characters of the read token.
yymore()	This function tells the lexer to append the next token to the current token.

Programming in Lex:

Programming in Lex can be divided into three steps:

1. Specify the pattern-associated actions in a form that Lex can understand.
2. Run Lex over this file to generate C code for the scanner.
3. Compile and link the C code to produce the executable scanner.

Note: If the scanner is part of a parser developed using Yacc, only steps 1 and 2 should be performed. Read the part B on Yacc.

Now let's look at the kind of program format that Lex understands. A Lex program is divided into three sections:

- The first section has global C and Lex declarations (regular expressions).
- The second section has the patterns (coded in C)
- The third section has supplemental C functions. main(), for example,

These sections are delimited by %%. Let us consider a word counting lex program to under stand the sections in detail.

Global C and Lex declarations:

In this section we can add C variable declarations. We will declare an integer variable here for our word-counting program that holds the number of words counted by the program. We'll also perform token declarations of Lex.

```
%{
int wordCount = 0;
%}
chars [A-za-z\_\'\.\"]
numbers ([0-9])+
delim [" "\n\t]
whitespace {delim}+
words {chars}+
%%
```

The double percent sign implies the end of this section and the beginning of the second of the three sections in Lex programming.

Lex rules for matching patterns:

Let's look at the Lex rules for describing the token that we want to match. (We'll use C to define what to do when a token is matched.) Continuing with our word-counting program, here are the rules for matching tokens.

```
{words} { wordCount++; /*increase the word count by one*/ }
{whitespace} { /* do nothing*/ }
{numbers} { /* one may want to add some processing here*/ }
%%
```

C code:

The third and final section of programming in Lex covers C function declarations (and occasionally the main function) Note that this section has to include the yywrap() function. Lex has a set of functions and variables that are available to the user. One of them is yywrap. Typically, yywrap() is defined as shown in the example below.

```
void main()
{
yylex(); /* start the analysis*/
printf(" No of words: %d\n", wordCount);
}
int yywrap()
{
return 1;
}
```

In the preceding sections we've discussed the basic elements of Lex programming, which should help you in writing simple lexical analysis programs.

Putting it all together:

This produces the lex.yy.c file, which can be compiled using a C compiler. It can also be used with a parser to produce an executable, or you can include the Lex library in the link step with the option –ll.

Here are some of Lex's flags:

- *-c* Indicates C actions and is the default.
- *-t* Causes the lex.yy.c program to be written instead to standard output.
- *-v* Provides a two-line summary of statistics.
- *-n* Will not print out the -v summary.

Yacc

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ``input language'' which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

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.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``,''

is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input
July 4, 1776
might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name  :  'J' 'a' 'n'   ;
month_name  :  'F' 'e' 'b'   ;

      ……….

month_name  :  'D' 'e' 'c'   ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month_name was seen; in this case, month_name would be a token.

Literal characters such as ``,'' must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date  :  month '/' day '/' year   ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be ``slipped in'' to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more

powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2, 3, 4] Yacc has been extensively used in numerous practical applications, including lint,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications :

Names refer to either tokens or non-terminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%'' marks. (The percent ``%'' is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also;

thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

A represents a non-terminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or non-terminal symbols.

A literal consists of a character enclosed in single quotes ``'''. As in C, the backslash ``\'' is an escape character within literals, and all the C escapes are recognized. Thus

```
'\n'    newline
'\r'    return
'\''    single quote ``'''
'\\'    backslash ``\''
'\t'    tab
'\b'    backspace
'\f'    form feed
'\xxx'  ``xxx'' in octal
```

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar ``|'' can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A   :   B  C  D  ;
A   :   E  F  ;
A   :   G  ;
```

can be given to Yacc as

```
A   :   B  C  D
    |   E  F
    |   G
    ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a non-terminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty :   ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token   name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

Of all the non-terminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. If the tokens up to, but not including, the end-marker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate; see section 3, below. Usually the end-marker represents some reasonably obvious I/O status, such as ``end-of-file" or ``end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces ``{" and ``}". For example,

```
A      :      '('  B  ')'
                  {      hello( 1, "abc" );  }
```

and

```
XXX    :      YYY  ZZZ
                  {      printf("a message\n");
                         flag = 25;   }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign" ``$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$" to some value. For example, an action that does nothing but return the value 1 is

```
{  $$ = 1;  }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A    :    B  C  D  ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr   :     '('  expr  ')'   ;
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr   :     '('  expr  ')'       {  $$ = $2 ;  }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A    :    B   ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A    :    B
              {  $$ = 1;  }
          C
              {   x = $2;   y = $3;  }
     ;
```

the effect is to set x to 1, and y to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new non-terminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT   :     /* empty */
              {  $$ = 1;  }
     ;

A    :    B  $ACT  C
              {   x = $2;   y = $3;  }
     ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function node, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr    :     expr  '+'  expr
                    {  $$ = node( '+', $1, $3 );  }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks ``%{" and ``%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called yylex. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable yylval.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the ``# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        . . .
        c = getchar();
        . . .
        switch( c ) {
                . . .
        case '0':
        case '1':
          . . .
        case '9':
                yylval = c-'0';
                return( DIGIT );
                . . .
                }
```

. . .

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names if or while will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name error is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the end-marker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the Lex program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the look-ahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no look-ahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a look-ahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the look-ahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the look-ahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a look-ahead token. For example, in state 56 there may be an action:

IF shift 34

which says, in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the look-ahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

. reduce 18

refers to grammar rule 18, while the action

IF shift 34

refers to state 34.

Suppose the rule being reduced is

A : x y z ;

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing x, y, and z, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a goto action. In particular, the look-ahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

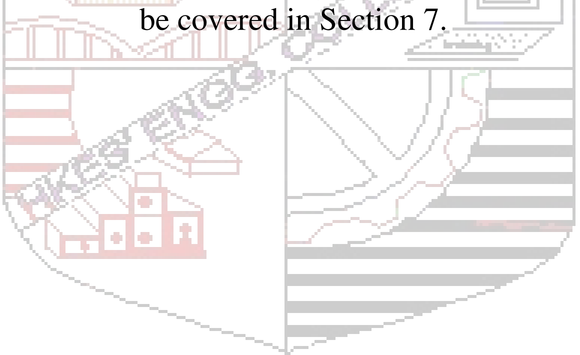
A goto 20

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action ``turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the look-ahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.



PART A

LEX

1) a. Program to count the number of characters, words, spaces and lines in a given input file.

Algorithm:

Step1: [Declarations and regular definition:]
Define all C global variable definition and header files to include in the first section.
integer character_count, word_count, space_count, line_count

Step2: [Transition rules]
Declare the transition rules with regular expressions

```
[^ \t\n]+ {word_count = word_count + 1
           character_count = character_count + yyleng }
\n        {line_count = line_count + 1}
" "       {space_count = space_count + 1 }
\t        {space_count = space_count + 1}
```

Step3: [Auxiliary Procedures]
Define the main() function of C program.
open a file for which action is to be performed and store the file pointer in yyin variable.
call yylex() function to perform the analysis.
print the results on the terminal

```
print word_count
print character_count
print line_count
print space_count
```

Program:

```
%{
#include<stdio.h>
int wc,cc,lc,sc;
%}
word[^\n\t]+
line[\n]
space[\t]
%%
{word} {wc++;cc+=yyleng;}
{line} {lc++;cc++;}
{space} {sc++;cc++;}
%%
int main(int argc,char * argv[])
{
if(argc!=2)
{
printf("\n\t Usage:./a.out filename\n");
return 0;
}
yyin=fopen(argv[1],"r");
yylex();
printf("\n Number of lines=%d",lc);
printf("\n Number of words=%d",wc);
printf("\n Number of character=%d",cc);
printf("\n Number of spaces=%d\n",sc);
return 0;
}
```

Output:

```
root@localhost:~
File Edit View Terminal Tabs Help
[root@localhost ~]# lex 1a.l
[root@localhost ~]# cc lex.yy.c -lfl
[root@localhost ~]# ./a.out  1a.l

 Number of lines=28
 Number of words=27
 Number of character=461
 Number of spaces=14
```

1) b. Program to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.

Algorithm:

Step1: [Declarations and regular definition:]
Define all C global variable definition and header files to include in the first section.
integer number_of_comments

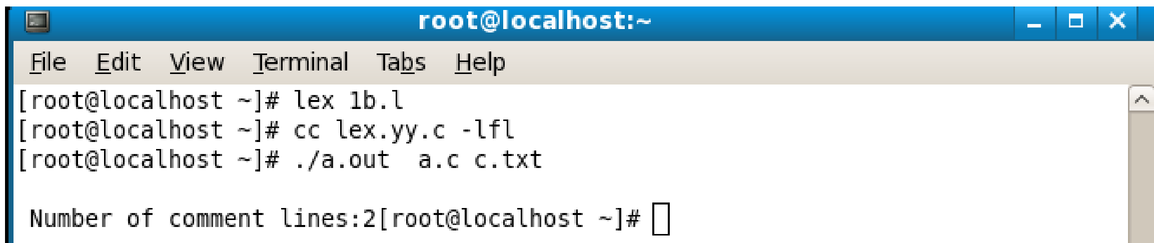
Step2: [Transition rules]
Declare the transition rules with regular expressions
a) Rule for multi – line comments
("/*"[a-zA-Z0-9\n\t]*"*/")
{number_of_comments=number_of_comments + 1 }
b) Rule for single – line comments
("//"[\n]?a-zA-Z0-9]*)
{number_of_comments=number_of_comments+ 1}
c) Any thing other than comments to be written in an output file
[a-zA-Z0-9(){}.\;#<>]* {write yytext on yyout}

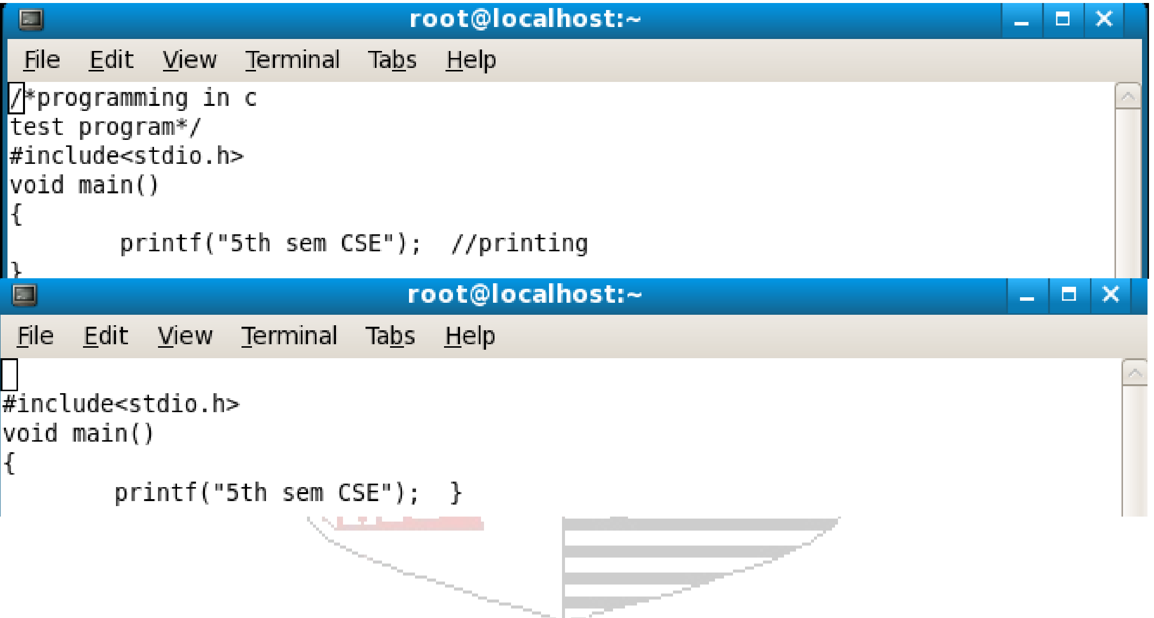
Step3: [Auxiliary Procedures]
Define the main() function of C program.
Open the input file and store file pointer in yyin variable.
Open the output file and store file pointer in yyout variable.
Call yylex() function to perform the analysis.
Print the result
print number_of_comments

Program:

```
%{
#include<stdio.h>
int flag=1,count=0;
%}
%%
"/*"[^ .]* {flag--;}
"*/" {flag++;count++;}
"//".*\n {count++;}
.|\n {if(flag)
fprintf(yyout,"%s",yytext);
}
%%
int main(int argc,char *argv[])
{
if(argc!=3)
{
printf("\n\tUsage ./a.out<file.c><file.c>\n");
exit(1);
}
yyin=fopen(argv[1],"r");
if(!yyin)
{
perror("open");
exit(1);
}
yyout=fopen(argv[2],"w");
if(!yyout)
{
perror("open");
exit(1);
}
yylex();
printf("\n Number of comment lines:%d",count);
return 0;
}
```

Output:





2) a. Program to recognize a valid arithmetic expression and to recognize the identifiers and operators present. Print them separately.

Algorithm:

Step1: [Declarations and regular definition:]
Define all C global variable definition and header files to include in the first section.
Integer number_of_plus, number_of_minus, number_of_multiplication
integer number_of_division, number_of_identifiers
integer flag_1, flag_2

Step2: [Transition rules]
Declare the transition rules with regular expressions
a) Rule for checking parenthesis
[(] {flag_1 = flag_1 + 1}
[)] {flag_1 = flag_1 - 1}
b) Rule for identifiers
[a-zA-Z0-9]+ {flag_2 = flag_2 + 1
number_of_identifiers = number_of_identifiers + 1}
c) Rule for plus symbol
[+] {flag_2 = flag_2 – 1
number_of_plus = number_of_plus + 1}
d) Rule for minus symbol
[-] {flag_2 = flag_2 – 1
number_of_minus = number_of_minus + 1}
e) Rule for multiplication symbol
[*] {flag_2 = flag_2 – 1
number_of_multiplication = number_of_multiplication + 1}
f) Rule for division symbol
[/] {flag_2 = flag_2 – 1
number_of_division = number_of_division + 1}

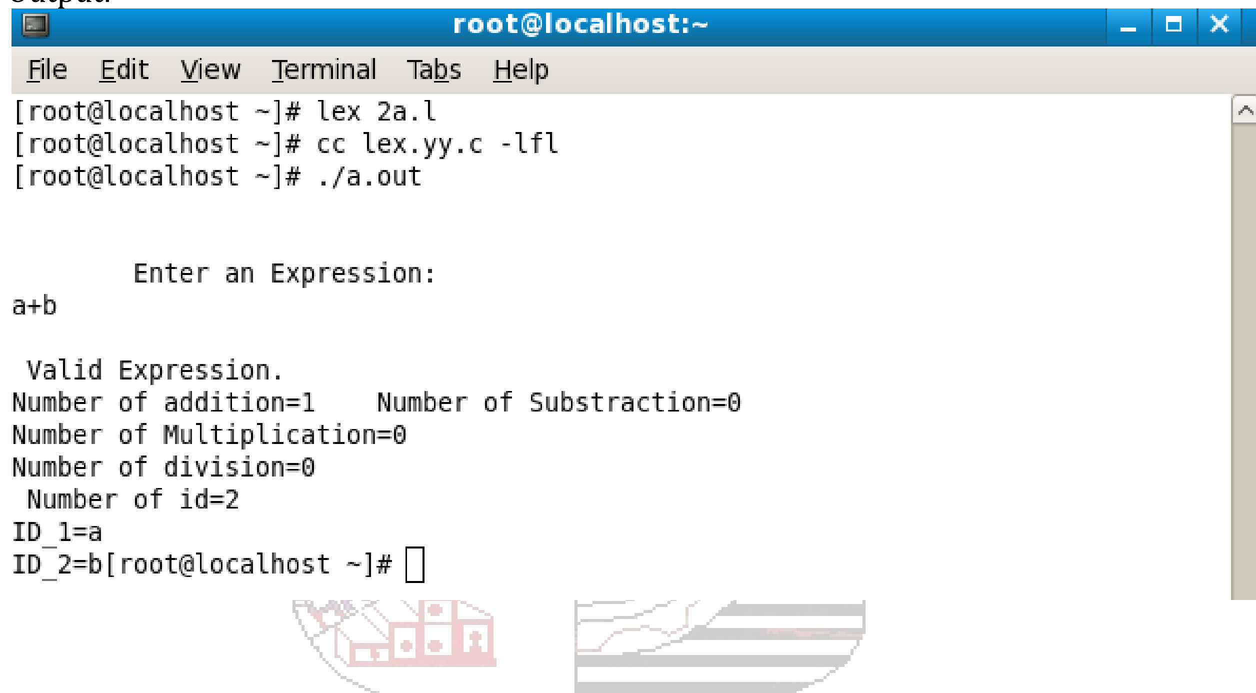
Step3: [Auxiliary Procedures]
Define the main() function of C program.
Read the input expression from standard input by calling yylex() function
Test the validity of the input string by examining the flags
flag_1 not equal to 0 or flag_2 not equal to 1
Else print the result
print number_of_plus
print number_of_minus
print number_of_multiplication
print number_of_division
print number_of_identifiers

Program:

```
%{
#include<stdio.h>
#include<stdlib.h>
int na,ns,nm,nd,id,pc,oc;
char s[25][10];
%}
%%
[(] {if(oc!=0)
yyerror("System Error");
pc++;
}
[)] {if(pc==0 || oc!=1)
yyerror("Syntax error");
pc--;
}
[a-z A-Z 0-9]+ {if(oc!=0)
yyerror("Syntax Error");
oc++;
strcpy(s[id++],yytext);
}
[+] {oc--;na++;}
[-] {oc--;ns++;}
[/] {oc--;nd++;}
[*] {oc--;nm++;}
[ \n\t]+ ;
. { yyerror("Invalid"); }
%%
yyerror(char *s)
{
puts(s);
printf("\n\t Invalid Expression.\n");
exit(1);
}
int main()
{
int i;
printf("\n\n\tEnter an Expression:\n");
yylex();
if(pc!=0 || oc!=1)
printf("\n\t Invalid Expression.\n");
else
{
printf("\n Valid Expression.");
```

```
printf("\nNumber of addition=%d\tNumber of Substraction=%d",na,ns);
printf("\nNumber of Multiplication=%d\nNumber of division=%d",nm,nd);
printf("\n Number of id=%d",id);
for(i=0;i<id;i++)
printf("\nID_%d=%s",i+1,s[i]);
}
return 0;
}
```

Output:



```
[root@localhost ~]# lex 2a.l
[root@localhost ~]# cc lex.yy.c -lfl
[root@localhost ~]# ./a.out


        Enter an Expression:
a+b

 Valid Expression.
Number of addition=1    Number of Substraction=0
Number of Multiplication=0
Number of division=0
 Number of id=2
ID_1=a
ID_2=b[root@localhost ~]#
```

2) b. Program to recognize whether a given sentence is simple or compound.

Algorithm:

Step1: [Declarations and regular definition:]
Define all C global variable definition and header files to include in the first section.

integer flag = 0

Step2: [Transition rules]
Declare the transition rules with regular expressions
a) Rules for testing the sentence
([aA][nN][dD]) { flag = 1}
("or") { flag = 1}
("nevertheless") { flag = 1}
("inspite") { flag = 1}

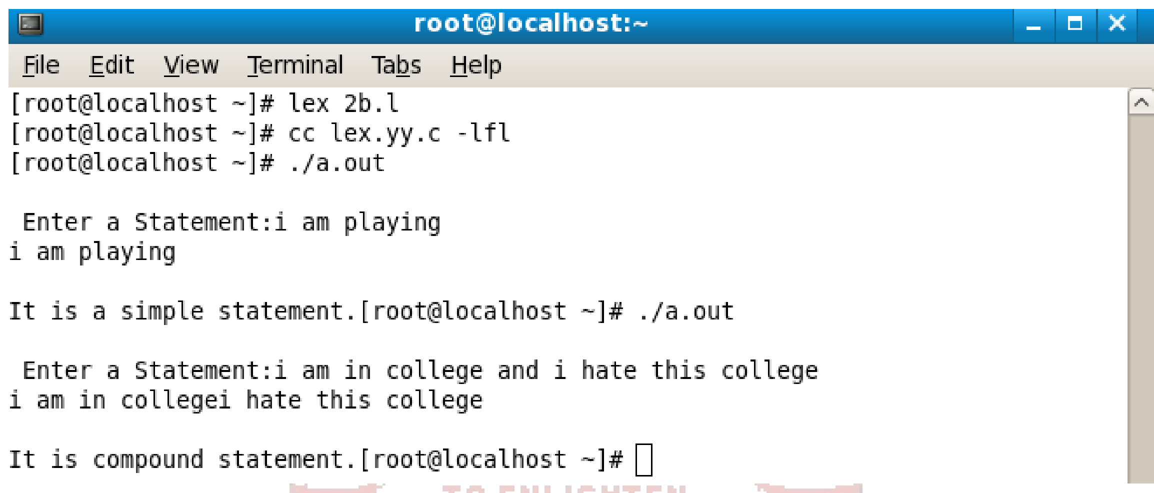
Step3: [Auxiliary Procedures]
Define the main() function of C program.
Read the input sentence from the standard input by calling yylex() function

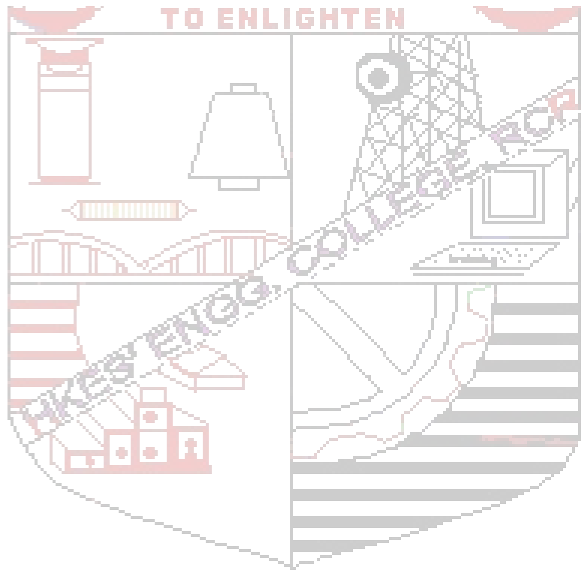
Test the flag variable to check the sentence
if flag is equal to zero then
print the sentence is simple
else
print the sentence is compound

Program:

```
%{
int flag=0;
%}
%%
(""[aA][nN][dD]"")|(""[oO][rR]"")|(""[bB][uU][tT]"") {flag=1;}
%%
int main()
{
printf("Enter the sentence\n");
yylex();
if(flag==1)
printf("\nCompound sentence\n");
else
printf("\nSimple sentence\n");
return 0;}
```

Output:





3) Program to recognize and count the number of identifiers in a given input file.

Algorithm:

Step1: [Declarations and regular definition:]
Define all C global variable definition and header files to include in the first section.
integer count

Step2: [Transition rules]
Declare the transition rules with regular expressions
Rules for identifiers in a source C program
"int" |
"float" |
"double" |
"char" { read a character by calling input() function and store in a variable say *ch*
repeat for ever
test *ch* if it is ";" the count = count + 1
test *ch* if it is "\n" then break the loop
read the next character and store in *ch*}

Step3: [Auxiliary Procedures]
Define the main() function of C program.
Open the input file and store file pointer in yyin variable.
Call yylex() function to perform the analysis.
Print the result
print count

Program:

```
%{
#include<stdio.h>
int count=0;
%}
op [+-*/]
letter [a-zA-Z]
digitt [0-9]
id {letter}*|({letter}{digitt})+
notid ({digitt}{letter})+
%%
[\t\n]+
("int")|("float")|("char")|("case")|("default")| ("if")|("for")|("printf")|("scanf") {printf("%s
is a keyword\n", yytext);}
```

```
{id} {printf("%s is an identifier\n", yytext); count++;}
{notid} {printf("%s is not an identifier\n", yytext);}
%%
int main()
{
FILE *fp;
char file[10];
printf("\nEnter the filename: ");
scanf("%s", file);
fp=fopen(file,"r");
yyin=fp;
yylex();
printf("Total identifiers are: %d\n", count);
return 0;
}
```

LIGHTENED TO ENLIGHTEN
COLLEGE RCR

Output:

```
root@localhost:~
File Edit View Terminal Tabs Help
[root@localhost ~]# lex 3.l
[root@localhost ~]# cc lex.yy.c -lfl
[root@localhost ~]# ./a.out   p1.c
Number of ID=2
[root@localhost ~]#
```

YACC:

4) a. Program to recognize a valid arithmetic expression that uses operators +, -, * and /.

Algorithm: Lex

Step1: [Declarations and regular definition:]
Define head files to include in the first section.
Step2: [Transition rules]
Tokens generated are used in yacc file
2. [a-zA-Z] Alphabets are returned.
Step3 : 0-9 one or more combination of Integers

Algorithm: Yacc

Step1:Define head file to include in the first section.
step2.Accept token generated in lex part as input.
step3.Specify the order of procedure.
Step4.Transition rules
Define the rules with end points.
Step5.Parse input string form standard input by calling yyparse() in main function.
Print the result of any of the rules defined matches.
Arithmetic expression is valid.
Step6.If none of the rules defined matches. Print Arithmetic expression is invalid.

Lex part :

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z] {return ALPHA;}
[0-9]+ {return NUMBER;}
[\t\n]+ ;
. {return yytext[0];}
%%
```

Yacc part :

```
%{
#include<stdio.h>
%}
%token NUMBER ALPHA
```

```
%left '+''-'
%left '*''/'
%left '('')'
%%
expr:'+'expr
    |'-'expr
    |expr'+'expr
    |expr'-'expr
    |expr'*'expr
    |expr'/'expr
    |'('expr')'
    |NUMBER
    |ALPHA
    ;
%%
int main()
{
  printf("enter an arithematic expression\n");
  yyparse();
  printf("arithematic expression is valid\n");
  return 0;
}
int yyerror(char *msg)
{
  printf("\n%s",msg);
  printf("\narithematic expression is invalid");
  exit(0);
}
```

Output :

```
root@localhost:~
File Edit View Terminal Tabs Help
[root@localhost ~]# lex 4a.l
[root@localhost ~]# yacc -d 4a.y
[root@localhost ~]# cc lex.yy.c y.tab.c -lfl
[root@localhost ~]# ./a.out
Enter an Expression:2*8-5
Valid Expression
[root@localhost ~]#
```

4) b. Program to to recognize a valid variable, which starts with a letter, followed by any number of letters or digits .

Algorithm: Lex

step1.Define head file to include in the first section.
step2.Accept token generated in lex part as input.
step3.Specify the order of procedure.
Step4:Define header file to include in the first section.
Step5:Transition rules
a.[a-z] letters are returned
b.[0-9] digits are returned

Algorithm:Yacc

step1.Step1:Define head file to include in the first section.
step2.Accept token generated in lex part as input.
Step3.Transition rules
a.Define the rules with end points.
Step4.Parse input string form standard input by calling yyparse(); in main() function.
Print the result of any of the rules defined matches Valid variable
Step5.If none of the rules defined matches. Print Invalid variable

Lex Part :

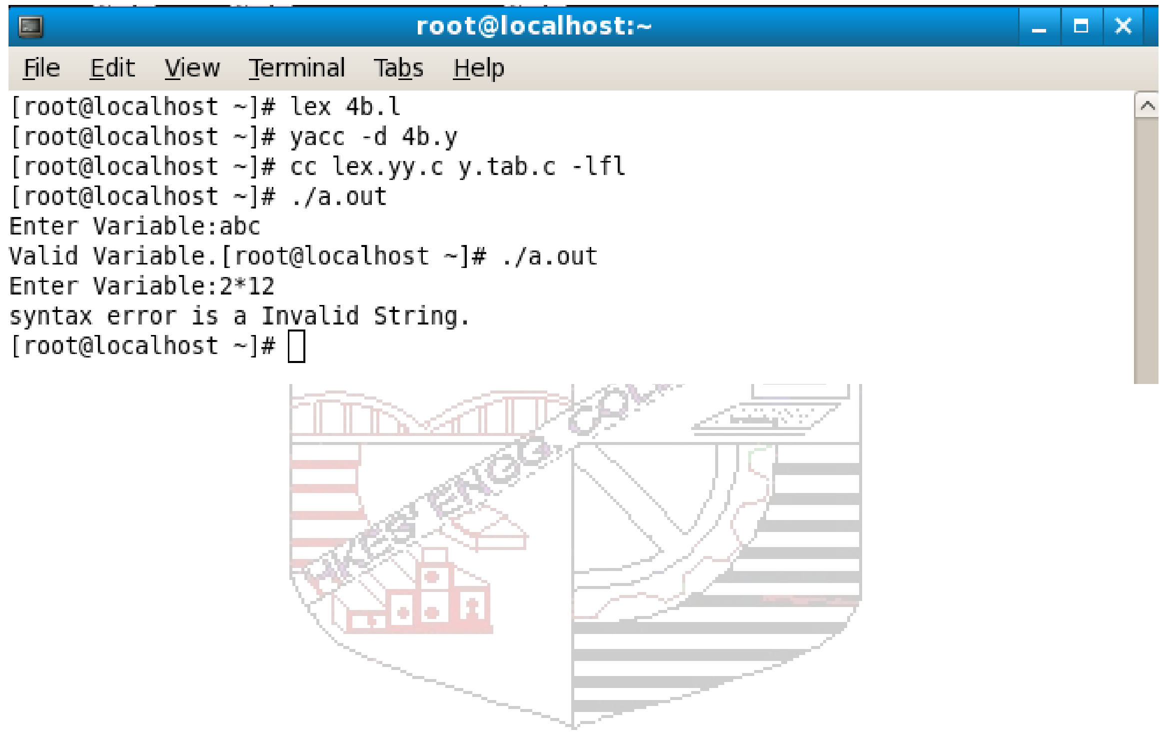
```
%{
 #include "y.tab.h"
%}
%%
[a-zA-Z]        return L;
[a-zA-Z0-9]     return D;
.               return P;
%%
```

Yacc Part :

```
%{#include<stdio.h>
%}
%token L D P
%%
var:L X
X:X D
| {printf("\nvalid variable"); return 0;}
P  { ; }
%%
```

```
main()
{
        printf("\nEnter variable");
        yyparse();
}
yyerror()
{
        printf("\nInvalid variable");
}
```

Output :



HKES' ENGG. COL.

5) a. Program to evaluate an arithmetic expression involving operators +, -, *,/ .

Algorithm:Lex

Step1: Declaration section
Define header file and c global variable definition in the first section.
Step2: Transition rules
[0-9] one or more combination of integers.

Algorithm: Yacc

Step1: Declaration section.
Define header file to include in the first section.
Step2: Accept the token generated in lex part as input.
Step3: Specify the order of procedure.
Step4: Transition rules.
a. Define the rules with end point
b. Parse input string from standard input by calling yyparse(); by in main function.
Step5: Print the result if any of the rules defined matches.
Step6: If none of the rules defined matches. Print Invalid expression.

Lex part :

```
%{
#include<stdio.h>
#include"y.tab.h"
extern int yylval;
%}

%%
[0-9]+  { yylval=atoi(yytext);
         return NUM;
        }
[\t] ;
\n return 0;
. return yytext[0];
%%
```

Yacc Part

```
%{
#include<stdio.h>
%}
%token NUM
```

```
%left '+' '-'
%left '*' '/'
%left '(' ')'
%%
expr: e
        { printf("result:%d\n",$$);
         return 0;
        }
e:e'+'e {$$=$1+$3;}
 |e'-'e {$$=$1-$3;}
 |e'*'e {$$=$1*$3;}
 |e'/'e {$$=$1/$3;}
 |'('e')' {$$=$2;}
 | NUM {$$=$1;}
;
%%

main()
{
printf("\n enter the arithematic expression:\n");
yyparse();
printf("\nvalid expression\n");
}

yyerror()
{
printf("\n invalid expression\n");
exit(0);
}
```

Output :

```
root@localhost:~
File  Edit  View  Terminal  Tabs  Help
[root@localhost ~]# lex 5a.l
[root@localhost ~]# yacc -d 5a.y
[root@localhost ~]# cc lex.yy.c y.tab.c -lfl
[root@localhost ~]# ./a.out
Enter the Expression:5*8-3
Result = 37
Valid Expression.[root@localhost ~]#
```

5 b) Program to recognize strings 'aaab', 'abbb', 'ab', 'a' using the grammer. ($a^n b^n$,n>=0)

Algorithm: Lex

Step1:Define head file to include in the first section.
Step2:Transition rules.

Example
I. a A is returned
II. b B is returned

Algorithm:Yacc

Step1:Include global c declaration and assign it to one.
Step2:Accept token generated in lex part as input.
Step3:Define header file to include in the first section.
Step4: Transition rules.
a.Define the rules with end point
b.Parse input string from standard input by calling yyparse(); by in main function.
Step5:Print the result Valid string if any of the rules defined matches.
Step6: If none of the rules defined matches print Invalid string.

Lex part :

```
%{
#include "y.tab.h"
%}
%%
a return A;
b return B;
.|\n return yytext[0];
%%
```

Yacc Part :

```
%{
#include<stdio.h>
int valid=1;
%}
%token A B
%%
str:S'\n' {return 0;}
S:A S B
 |
 ;
```

```
%%
main()
{
printf("Enter the string:\n");
yyparse();
if(valid==1)
  printf("\nvalid string");
}

yyerror()
{
valid=0;
printf("\ninvalid string");
return 1;
}
```

Output :

```
root@localhost:~
File Edit View Terminal Tabs Help
[root@localhost ~]# yacc -d 5b.y
[root@localhost ~]# lex 5b.l
[root@localhost ~]# cc lex.yy.c y.tab.c -lfl
[root@localhost ~]# ./a.out
Enter the string m:aabb
It is a valid string
[root@localhost ~]# ./a.out
Enter the string m:abb
Invalid String.[root@localhost ~]#
```

6) Program to recognize the grammar (an b ,n>=10)

Algorithm: Lex

Step1:Define head file to include in the first section.
Step2:Transition rules.
a A is returned
b B is returned

Algorithm:Yacc

Step1:Include global c declaration and assign it to one.
Step2:Accept token generated in lex part as input.
Step3:Define header file to include in the first section.
Step4: Transition rules.
a.Define the rules with end point
b.Parse input string from standard input by calling yyparse(); by in main function.
Step5:Print the result Valid string if any of the rules result defined matches.
Step6: If none of the rules defined matches print Invalid string.

Lex Part :

```
%{
#include "y.tab.h"
%}
%%
a return A;
b return B;
.|\n return yytext[0];
%%
```

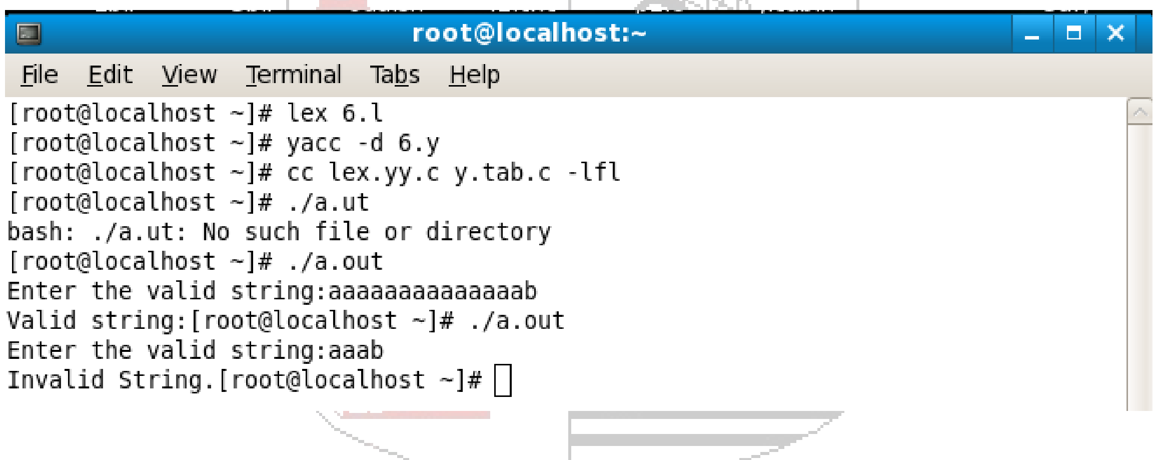
Yacc Part :

```
%{
#include<stdio.h>
int valid=1;
%}
%token A B
%%
str:S'\n' {return 0;}
S:A S B
 |
 ;
```

```
%%
main()
{
printf("Enter the string:\n");
yyparse();
if(valid==1)
  printf("\nValid string");
}

yyerror()
{
valid=0;
printf("\nInvalid string");
return 1;
}
```

Output :


```
[root@localhost ~]# lex 6.l
[root@localhost ~]# yacc -d 6.y
[root@localhost ~]# cc lex.yy.c y.tab.c -lfl
[root@localhost ~]# ./a.ut
bash: ./a.ut: No such file or directory
[root@localhost ~]# ./a.out
Enter the valid string:aaaaaaaaaaaaaab
Valid string:[root@localhost ~]# ./a.out
Enter the valid string:aaab
Invalid String.[root@localhost ~]#
```

PART B

Unix Programming:

Shell Programming :

The Shell is one of the major components of the Unix system. As a command interpreter, it provides an interface between the user and the operating system. The shell is also a programming language that executes shell scripts in the interpretive mode – one line at a time.

The Shell programs or **shell scripts** are executable text files that contain UNIX commands.

The Unix Systems offers a variety of shells like **Bourne shell , C shell, Korn shell and bash(born again shell)** shell for you to choose.

Shell scripts are typically written when:

- there is a command or string of commands that you will use more than once.
- you want access to command line arguments
- you need looping and testing.

Writing and running a Shell Script

1. Use your **vi** editor to create the shell script
2. To run the script, use sh followed by the name of the shell script.
   For example,
   $ sh filename.sh
   When used in this way, the script doesn't need to have executable permission.

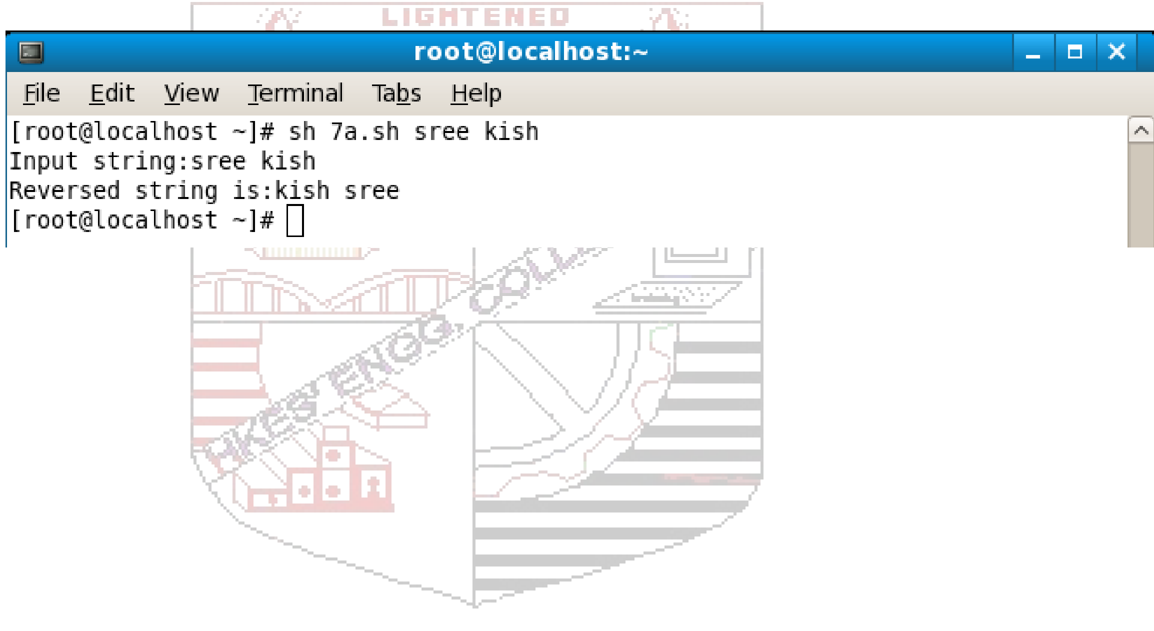
   OR

   To run the script, make it executable and invoke the script name.
   For example,
   $ chmod +x filename.sh
   $ filename.sh

7) a. Non-recursive shell script that accepts any number of arguments and prints them in the Reverse order, (For example, if the script is named ranges, then executing ranges A B C should produce C B A on the standard output).

```
echo "Input string is:$*"
for x in "$@"
do
y=$x" "$y
done
echo "Reversed string is:$y"
```

Output:



7) b. C program that creates a child process to read commands from the standard input and execute them (a minimal implementation of a shell – like program). You can assume that no arguments will be passed to the commands to be executed.

```
#include<stdio.h>
int main()
{
  char str[10];
  int pid;
  pid=fork();
  if(!pid)
  {
       printf("Child process...");
       printf("\nEnter a command:");
       scanf("%s",str);
       system(str);
       printf("Finished with child");
  }
 else
{
         wait();
         printf("\nParent Process");
}
return 0; }
```

Sample input/output :

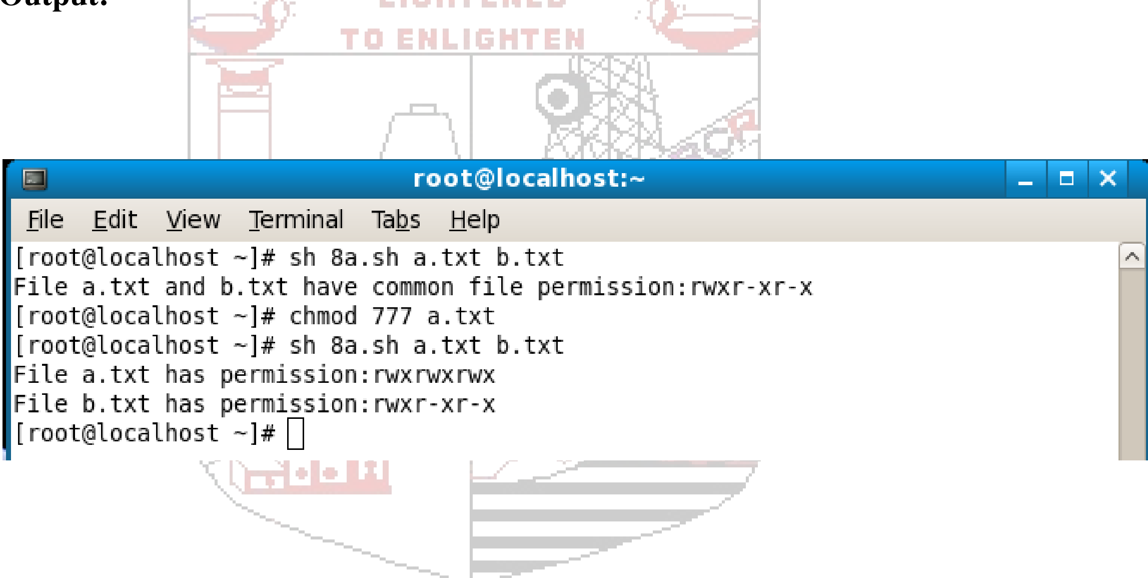
```
root@localhost:~
File  Edit  View  Terminal  Tabs  Help
[root@localhost ~]# vi 7b.c
[root@localhost ~]# cc 7b.c
[root@localhost ~]# ./a.out
child process
enter th ecommand
ls
2a.l    7a.sh  b.txt    final1                  p1          prog1.c   unix8a.sh
2b.l    7b.c   child.c  gnome-terminal.desktop  p1.c        prog3.l   unix8b.c
3a.l    9a.sh  dist.c   lex.yy.c                pg8.c       t.txt     unix.sh
3a.sh~  a.out  f1.txt   m1.c                    praveen.c   unix      unzip.sh
5b.y~   a.txt  f2.txt   nautilus-debug-log.txt  praveen.c~  unix7b.c  xyz
finished with child

 parent process
[root@localhost ~]#
```

8) a. Write a shell script that accepts two filenames as arguments, checks if the permissions for these files are identical & if the permissions are identical, outputs the common permissions, otherwise outputs each file names followed by its permissions

```
f1=`ls -l $1|cut -c2-10`
f2=`ls -l $2|cut -c2-10`
if  [ $f1  ==  $f2 ]
then
echo "files $1 & $2 have common file permissions: $f1"
else
echo "file $1 has file permissions : $f1"
echo "file $2 has file permissions : $f2"
fi
```

Output:



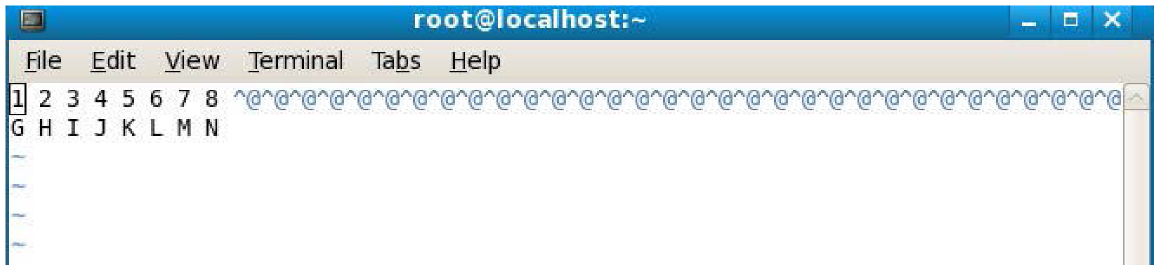
8) b. Write a C program to create a file with 16 bytes of arbitrary data fro the beginning and another 16 bytes of arbitrary data from an offset of 48.Display the file contents to demonstrate how the hole in file is handled.

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>

int main()
{
        char buf1[]="123456789ABCDEFG";
        char buf2[]="HIJKLMNOPQRSTUVW";
        int fd;
        fd=creat("t.txt",O_WRONLY|777);
        write(fd,buf1,16);
        lseek(fd,48,SEEK_SET);
        write(fd,buf2,16);
        return 0;
}
```

Output:

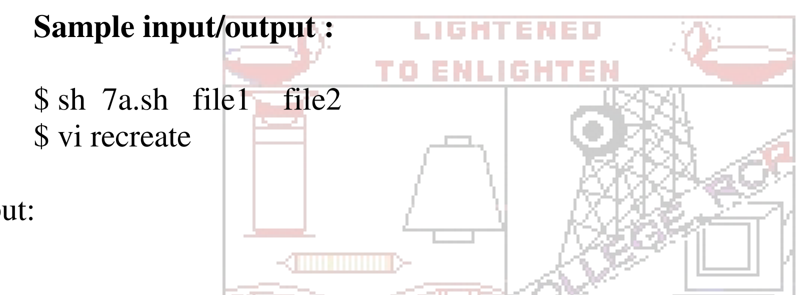
```
root@localhost:~
File Edit View Terminal Tabs Help
[root@localhost ~]# cc 8b.c
[root@localhost ~]# ./a.out
[root@localhost ~]# vi t.txt
[root@localhost ~]#
```



9) a. Shell script that accepts file names specified as arguments and creates a shell script that contains this file as well as the code to recreate these files. Thus if the script generated by your script is executed, it would recreate the original files (This is same as the “bundle” script described by Brain W. Kernighan and Rob Pike in “The Unix Programming Environment”, Prentice – Hall India).

```
for x in $*
do
echo "cat > $x << here
abc
def
ghi
here"
done > recreate
```

Sample input/output :




```
$ sh  7a.sh   file1    file2
$ vi recreate
```

Output:

```
root@localhost:~/xyz
File  Edit  View  Terminal  Tabs  Help
[root@localhost ~]# vi c.txt
[root@localhost ~]# vi d.txt
[root@localhost ~]# sh 9a.sh c.txt d.txt >unzip.sh
[root@localhost ~]# mkdir xyz
mkdir: cannot create directory `xyz': File exists
[root@localhost ~]# cd xyz
[root@localhost xyz]# sh ../unzip.sh
c.txt
d.txt
[root@localhost xyz]#
```

9) b. write a C program to create child process. the child process prints its own process-id and id of its parent and then exits. The parent process waits for its child to finish & prints its own process id & the id of its child process and then exits.

```
int main()
{
        char str[10];
        int pid;
        pid=fork();
        if(!pid)
        {
                printf("Child process...");
                printf("\n\nChild PID : %d",getpid());
                printf("\nParent PID : %d",getppid());
                printf("\n\nFinished with child\n");
        }
        else
        {
                wait();
                printf("\nParent process");
                printf("\nPARENT PID : %d",getpid());
                printf("\nChild PID : %d",pid);
        }
        return 0;
}
```

Output:

```
root@localhost:~
File Edit View Terminal Tabs Help
[root@localhost ~]# cc 9b.c
[root@localhost ~]# ./a.out

The child process
The child PID:4142
The parent PID:4141

The parent process
The child PID:4141
The parent PID:4142

[root@localhost ~]#
```

VIVA Questions

Part-B

1. Explain the feature of Unix OS.
2. What is process? Explain the parent child relationship.
3. What are the main functions of shell?
4. What are the steps involved in creating child process?
5. What are positional parameters?
6. What are environmental variables?
7. What is $*,$#,$$,$_,$!
8. Where do you use expr in shell scripts?
9. What is shell script? How it is different form c program?
10. What do you mean by exit status of a command?
11. Where do you use cut command?
12. Differentiate head and tail?
13. What is set command?
14. What is the function of lseek ( ) ?
15. What are types of files in Unix?
16. How can you change the file permissions?
17. What is fork ( ) ?
18. What is the purpose of wait ( ) ?
19. What is the purpose pf getenv ( ) ?
20. Who is the parent of all processes?
21. Explain the mechanism of process creation?
22. List the file attributes?
23. What are shell variables? Give an example.
24. What does sed command do?

Operating Systems:

10. Design, develop and execute a program in C / C++ to simulate the working of Shortest Remaining Time and Round-Robin Scheduling Algorithms. Experiment with different quantum sizes for the Round- Robin algorithm. In all cases, determine the average turn-around time. The input can be read from key board or from a file.

```
#include<stdio.h>
#include<stdlib.h>
struct proc
{
int id;
int arrival;
int burst;
int rem;
int wait;
int finish;
int turnaround;
float ratio;
}process[10];
struct proc temp;
int no;
int chkprocess(int);
int nextprocess();
void roundrobin(int,int,int[],int[]);
void srtf(int);
void main()
{
int n,tq,choice,bt[10],st[10],j,i;
for(;;)
{
printf("Enter your choice\n");
printf("1.Round Robin\n2.Shortest Remaining Time First\n3.Exit\n");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Round Robin scheduling\n\n");
printf("Enter number of processes:");
scanf("%d",&n);
printf("\nEnter burst time for sequences:");
for(i=0;i<n;i++)
{
scanf("%d",&bt[i]);
st[i]=bt[i];
```

```
}
printf("\nEnter time quantum:");
scanf("%d",&tq);
roundrobin(n,tq,st,bt);
break;
case 2:
printf("Shorest Remaining Time First\n");
printf("Enter the number of process:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Input Arrival time of process %d\n",i+1);
scanf("%d",&process[i].arrival);
printf("Input Burst time of process %d\n",i+1);
scanf("%d",&process[i].burst);
process[i].rem=process[i].burst;
}
srtf(n);
break;
case 3: exit(0);
}
}
}
void roundrobin(int n,int tq,int st[],int bt[])
{
int tat[10],wt[10],i,count=0,swt=0,stat=0,temp1,sq=0;
float awt=0.0,atat=0.0;
while(1)
{
for(i=0,count=0;i<n;i++)
{
temp1=tq;
if(st[i]==0)
{
count++;
continue;
}
if(st[i]>tq)
st[i]-=tq;
else
if(st[i]>=0)
{
temp1=st[i];
st[i]=0;
}
```

```
sq+=temp1;
tat[i]=sq;
}
if(n==count)
break;
}
for(i=0;i<n;i++)
{
wt[i]=tat[i]-bt[i];
swt+=wt[i];
stat+=tat[i];
}
awt=(float)swt/n;
atat=(float)stat/n;
printf("Process_No\tBurst time\tWait time\tTurn around time\n");
for(i=0;i<n;i++)
printf("%d\t\t%d\t\t%d\t\t%d\t\t\n",i+1,bt[i],wt[i],tat[i]);
printf("Avg wait time is %f\nAvgTurn Around time is %f\n",awt,atat);
}
//SHORTEST REMAINING TIME FIRST
int chkprocess(int s)
{
int i;
for(i = 0; i < s; i++)
{
if(process[i].rem != 0)
return 1;
}
return 0;
}
int nextprocess()
{
int min,l,i;
min = 32000;
for(i=0;i<no;i++)
{
if(process[i].rem!=0&&process[i].rem<min)
{
min = process[i].rem;
l = i;
}
}
return l;
}
void srtf(int n)
```

```
{
int i,j,k,time=0;
float tavg=0,wavg=0;
no = 0;
j = 1;
while(chkprocess(n) == 1)
{
if(process[no].arrival == time)
{
no++;
if(process[j].rem==0)
process[j].finish=time;
j = nextprocess();
}
if(process[j].rem != 0)
{
process[j].rem--;
for(i = 0; i < no; i++)
{
if(i != j && process[i].rem != 0)
process[i].wait++;
}
}
else
{
process[j].finish = time;
j=nextprocess();
time--;
k=j;
}
time++;
}
process[k].finish = time;
printf("\n\n\t\t\t---SHORTEST REMAINING TIME NEXT---");
printf("\n\n Process Arrival Burst Waiting Finishing turnaround Tr/Tb \n");
printf("%5s %9s %7s %10s %8s %9s\n\n", "id", "time", "time", "time", "time",
"time");
for(i = 0; i < n; i++)
{
process[i].turnaround = process[i].wait + process[i].burst;
process[i].ratio = (float)process[i].turnaround / (float)process[i].burst;
printf("%5d %8d %7d %8d %10d %9d %10.1f ", process[i].id+1,
process[i].arrival,process[i].burst,process[i].wait,
process[i].finish,process[i].turnaround, process[i].ratio);
tavg+=process[i].turnaround;
```

```
wavg+=process[i].wait;
printf("\n\n");
}
tavg/=n;
wavg/=n;
printf("Turn Around average time is %f\nWaiting average time is %f",tavg,wavg);
}
```

OUTPUT:

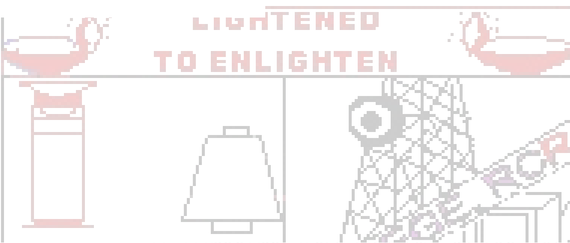
```
Enter your choice
1.Round Robin
2.Shortest Remaining Time First
3.Exit
1
Round  Robin scheduling

Enter number of processes:3

Enter burst time for sequences:24
3
3

Enter time quantum:4
Process_No      Burst time      Wait time       Turn around time
1               24              6               30
2               3               4               7
3               3               7               10
Avg wait time is 5.666667
AvgTurn Around time is 15.666667
Enter your choice
1.Round Robin
2.Shortest Remaining Time First
3.Exit
```

```
2
Shorest Remaining Time First
Enter the number of process:4
Input Arrival time of process 1
0
Input Burst time of process 1
8
Input Arrival time of process 2
1
Input Burst time of process 2
4
Input Arrival time of process 3
2
Input Burst time of process 3
9
Input Arrival time of process 4
3
Input Burst time of process 4
5
```




```
                    ---SHORTEST REMAINING TIME NEXT---

 Process  Arrival  Burst   Waiting  Finishing turnaround  Tr/Tb
   id      time    time      time     time       time

    1        0       8         9        17         17       2.1

    1        1       4         0         5          4       1.0

    1        2       9        15        26         24       2.7

    1        3       5         2        10          7       1.4

Turn Around average time is 13.000000
Waiting average time is 6.500000Enter your choice
1.Round Robin
2.Shortest Remaining Time First
3.Exit
```

11. Using OpenMP, Design, develop and run a multi-threaded program to generate and print Fibonacci Series. One thread has to generate the numbers up to the

specified limit and another thread has to print them. Ensure proper synchronization.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
void Usage(char prog_name[]);
int main(int argc, char* argv[]) {
int thread_count, n, i;
long long* fibo;
if (argc != 3) Usage(argv[0]);
thread_count = strtol(argv[1], NULL, 10);
n = strtol(argv[2], NULL, 10);
fibo = malloc(n*sizeof(long long));
fibo[0] = fibo[1] = 1;
# pragma omp parallel for num_threads(thread_count) \
schedule(static,1)
for (i = 2; i < n; i++)
fibo[i] = fibo[i-1] + fibo[i-2];
printf("The first n Fibonacci numbers:\n");
for (i = 0; i < n; i++)
printf("%d\t%lld\n", i, fibo[i]);
free(fibo);
return 0;
}
void Usage(char prog_name[]) {
fprintf(stderr, "usage: %s <thread count> <number of Fibonacci numbers>\n",
prog_name);
exit(0);
}
```

Output:

```
root@localhost:~
File  Edit  View  Terminal  Tabs  Help
[root@localhost ~]# gcc -g -Wall -fopenmp fs.c
[root@localhost ~]# ./a.out 1 5
The first n Fibonacci numbers:
0       1
1       1
2       2
3       3
4       5
[root@localhost ~]#
```

12) Design, develop and run a program to implement the Banker's Algorithm. Demonstrate its working with different data values.

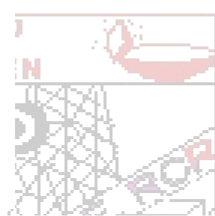
```
#include<stdio.h>
#include<conio.h>
struct da
{
int max[10],a1[10],need[10],before[10],after[10];
}p[10];
void main()
{
int i,j,k,l,r,n,tot[10],av[10],cn=0,cz=0,temp=0,c=0;
clrscr();
printf("\n ENTER THE NO. OF PROCESSES:");
scanf("%d",&n);
printf("\n ENTER THE NO. OF RESOURCES:");
scanf("%d",&r);
for(i=0;i<n;i++)
{
printf("PROCESS %d \n",i+1);
for(j=0;j<r;j++)
{
printf("MAXIMUM VALUE FOR RESOURCE %d:",j+1);
scanf("%d",&p[i].max[j]);
}
for(j=0;j<r;j++)
{
printf("ALLOCATED FROM RESOURCE %d:",j+1);
scanf("%d",&p[i].a1[j]);
p[i].need[j]=p[i].max[j]-p[i].a1[j];
}
}
for(i=0;i<r;i++)
{
printf("ENTER TOTAL VALUE OF RESOURCE %d:",i+1);
scanf("%d",&tot[i]);
}
for(i=0;i<r;i++)
{
for(j=0;j<n;j++)
temp=temp+p[j].a1[i];
av[i]=tot[i]-temp;
temp=0;
}
printf("\n\tRESOURCES\tALLOCATED\tNEEDED\tTOTAL\tAVAIL");
for(i=0;i<n;i++)
```

```
{
printf("\nP%d\t\t",i+1);
for(j=0;j<r;j++)
printf("%d",p[i].max[j]);
printf("\t\t");
for(j=0;j<r;j++)
printf("%d",p[i].a1[j]);
printf("\t");
for(j=0;j<r;j++)
printf("%d",p[i].need[j]);
printf("\t");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",tot[j]);
}
printf("\t");
for(j=0;j<r;j++)
{
if(i==0)
printf("%d",av[j]);
}
}
printf("\n\n\t AVAIL BEFORE\T AVAIL AFTER ");
for(l=0;l<n;l++)
{
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
if(p[i].need[j] >av[j])
cn++;
if(p[i].max[j]==0)
cz++;
}
if(cn==0 && cz!=r)
{
for(j=0;j<r;j++)
{
p[i].before[j]=av[j]-p[i].need[j];
p[i].after[j]=p[i].before[j]+p[i].max[j];
av[j]=p[i].after[j];
p[i].max[j]=0;
}
printf("\nP%d\t\t",i+1);
for(j=0;j<r;j++)
```

```
printf("%d",p[i].before[j]);
printf("\t\t");
for(j=0;j<r;j++)
printf("%d",p[i].after[j]);
cn=0;
cz=0;
c++;
break;
}
else
{
cn=0;cz=0;
}
}
}
if(c==n)
printf("\n THE ABOVE SEQUENCE IS A SAFE SEQUENCE");
else
printf("\n DEADLOCK OCCURED");
getch();
}
```

```
 ENTER THE NO. OF PROCESSES:4

 ENTER THE NO. OF RESOURCES:3
PROCESS 1
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:1
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:0
PROCESS 2
MAXIMUM VALUE FOR RESOURCE 1:6
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:3
ALLOCATED FROM RESOURCE 1:5
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1
PROCESS 3
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:4
ALLOCATED FROM RESOURCE 1:2
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1_
```



Output1:

```
PROCESS 4
MAXIMUM VALUE FOR RESOURCE 1:4
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:0
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:2
ENTER TOTAL VALUE OF RESOURCE 1:9
ENTER TOTAL VALUE OF RESOURCE 2:3
ENTER TOTAL VALUE OF RESOURCE 3:6

        RESOURCES       ALLOCATED       NEEDED  TOTAL   AVAIL
P1              322             100     222     936     112
P2              613             511     102
P3              314             211     103
P4              422             002     420

         AVAIL  BEFORET AVAIL AFTER
P2              010             623
P1              401             723
P3              620             934
P4              514             936
 THE ABOVE SEQUENCE IS A SAFE SEQUENCE_
```

```
 ENTER THE NO. OF PROCESSES:4

 ENTER THE NO. OF RESOURCES:3
PROCESS 1
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:1
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:1
PROCESS 2
MAXIMUM VALUE FOR RESOURCE 1:6
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:3
ALLOCATED FROM RESOURCE 1:5
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:1
PROCESS 3
MAXIMUM VALUE FOR RESOURCE 1:3
MAXIMUM VALUE FOR RESOURCE 2:1
MAXIMUM VALUE FOR RESOURCE 3:4
ALLOCATED FROM RESOURCE 1:2
ALLOCATED FROM RESOURCE 2:1
ALLOCATED FROM RESOURCE 3:2_
```

Output2:

```
PROCESS 4
MAXIMUM VALUE FOR RESOURCE 1:4
MAXIMUM VALUE FOR RESOURCE 2:2
MAXIMUM VALUE FOR RESOURCE 3:2
ALLOCATED FROM RESOURCE 1:0
ALLOCATED FROM RESOURCE 2:0
ALLOCATED FROM RESOURCE 3:2
ENTER TOTAL VALUE OF RESOURCE 1:9
ENTER TOTAL VALUE OF RESOURCE 2:3
ENTER TOTAL VALUE OF RESOURCE 3:6

        RESOURCES       ALLOCATED       NEEDED  TOTAL   AVAIL
P1              322             101     221     936     110
P2              613             511     102
P3              314             212     102
P4              422             002     420

         AVAIL  BEFORET AVAIL AFTER
 DEADLOCK OCCURED_
```

Compilation

Lex

If Program name is p1.l
$ lex p1.l
$ cc lex.yy.c –ll
$ ./a.out

Yacc with Lex program

If Program name is p1.l, p1.y
$ lex p1.l
$ yacc p1.y
$ cc lex.yy.c y.tab.c –ll
$ ./a.out