VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELAGAVI-590018

SYSTEM SOFTWARE AND OPERATING SYSTEMS LAB LABORATORY MANUAL (15CSL67)

DEVELOPED BY

Mr.Gururaj.T M.Tech
Asso.Professor
Department of CS & E
SJMIT, Chitradurga.

Carried out at:

SJM INSTITUTE OF TECHNOLOGY
Department of Computer Science and Engineering
P.B.No.73, NH-4, Bye pass
Chitradurga -577502

Academic Year:
2019-20

Course Details

Course Name	**:** System Software and Compiler Design / Operating Systems Lab
Course Code	**:** 15CSL67
Course prerequisite	**:** Basic Knowledge on Lex, YACC, C programming, UNIX commands and shell scripts

Course Objectives

1. To make students familiar with Lexical Analysis and Syntax Analysis phases of Compiler Design and implement programs on these phases using LEX & YACC tools and/or C/C++/Java
2. To enable students to learn different types of CPU scheduling algorithms used in operating system.
3. To make students able to implement memory management - page replacement and deadlock handling algorithms

Syllabus

Subject Code	**: 15CSL67**	**IA Marks**	**: 20**
No. of Practical Hrs/ Week	**: 01I + 02P**	**Exam Hours**	**: 03**
Total No. of Practical Hrs	**: 40**	**Exam Marks**	**: 80**

Description (If any): Exercises to be prepared with minimum three files (Where ever necessary):

i. Header file.

ii. Implementation file.

iii. Application file where main function will be present.

The idea behind using three files is to differentiate between the developer and user sides. In the developer side, all the three files could be made visible. For the user side only header file and application files could be made visible, which means that the object code of the implementation file could be given to the user along with the interface given in the header file, hiding the source file, if required. Avoid I/O operations (printf/scanf) and use ***data input file*** where ever it is possible

Laboratory Experiments:

1. a) Write a LEX program to recognize valid ***arithmetic expression***. Identifiers in the expression could be only integers and operators could be + and *. Count the identifiers & operators present and print them separately.

   b) Write YACC program to evaluate ***arithmetic expression*** involving operators: +, -, *, and /.
2. Develop, Implement and execute a program using YACC tool to recognize all strings ending with ***b*** preceded by ***n a's*** using the grammar $a^n b$ (note: input ***n*** value).
3. Design, develop and implement YACC/C program to construct ***Predictive / LL(1) Parsing Table*** for the grammar rules***: A →aBa , B →bB | ε***. Use this table to parse the sentence***: abba$.***
4. Design, develop and implement YACC/C program to demonstrate ***Shift Reduce Parsing*** technique for the grammar rules: ***E →E+T | T, T →T*F | F, F →(E) | id*** and parse the sentence***: id + id * id***.
5. Design, develop and implement a C/Java program to generate the machine code using ***Triples*** for the statement ***A = -B * (C +D)*** whose intermediate code in three-address form:

   T1 = -B

   T2 = C + D

   T3 = T1 + T2

   A = T3

6. a) Write a LEX program to eliminate ***comment lines*** in a ***C*** program and copy the resulting program into a separate file.

   b) Write YACC program to recognize valid ***identifier, operators*** and ***keywords*** in the given text (C program) file.

7. Design, develop and implement a C/C++/Java program to simulate the working of ***Shortest remaining time*** and ***Round Robin (RR)*** scheduling algorithms. Experiment with different quantum sizes for RR algorithm.

8. Design, develop and implement a C/C++/Java program to implement ***Banker's algorithm***. Assume suitable input required to demonstrate the results.

9. Design, develop and implement a C/C++/Java program to implement ***page replacement algorithms LRU*** and ***FIFO***. Assume suitable input required to demonstrate the results.

Course Outcomes

Upon successful completion of this course, students are able to:

15CSL67 System Software & Compiler Design / Operating Systems Lab	
COs	**COURSE OUTCOMES**
CO1	Implement and demonstrate Lexer's and Parser's
CO2	Evaluate different algorithms required for management, scheduling, allocation and communication used in operating system.

Conduction of Practical Examination:

- All laboratory experiments are to be included for practical examination.
- Students are allowed to pick one experiment from the lot.
- Strictly follow the instructions as printed on the cover page of answer script
- Marks distribution: Procedure + Conduction + Viva:20 + 50 +10 (80)
- Change of experiment is allowed only once and marks allotted to the procedure part to be made zero

1. Introduction to LEX

Lex and YACC helps you write programs that transforms structured input. Lex generates C code for lexical analyzer whereas YACC generates Code for Syntax analyzer. Lexical analyzer is build using a tool called LEX. Input is given to LEX and lexical analyzer is generated.

Lex is a UNIX utility. It is a program generator designed for lexical processing of character input streams. Lex generates C code for lexical analyzer. It uses the **patterns** that match **strings in the input** and converts **the strings** to tokens. Lex helps you by taking a set of descriptions of possible tokens and producing a C routine, which we call a lexical analyzer. The token descriptions that Lex uses are known as regular expressions.

1.1 Steps in writing LEX Program:

1st step: Using gedit create a file with extension l. For example: prg1.l

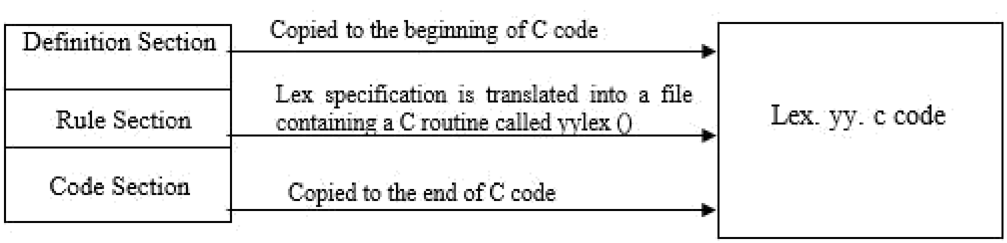
2nd Step: lex prg1.l

3rd Step: cc lex.yy.c –ll

4th Step: ./a.out

1.2 Structure of LEX source program:

1st Col	2nd Col	3rd Col	4th Col
	DEFINITION SECTION		
%%			
	RULE SECTION		
%%			
	CODE SECTION		



%% is a delimiter to the mark the beginning of the Rule section. The second %% is optional, but the first is required to mark the beginning of the rules. The definitions and the code /subroutines are often omitted

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.
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.

1.3 Regular Expressions

It is used to describe the pattern. It is widely used to in lex. It uses meta language. The character used in this meta language are part of the standard ASCII character set. 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 two tables define some of the symbols used in Lex and give a few typical examples.

Character	Meaning
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches any character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.

<table>
<tr><th>Character</th><th>Meaning</th></tr>
<tr><td>*</td><td>Match zero or more occurrences of the preceding pattern.</td></tr>
<tr><td>+</td><td>Matches one or more occurrences of the preceding pattern.(no empty string)
Ex: [0-9]+ matches “1”,”111” or “123456” but not an empty string.</td></tr>
<tr><td>?</td><td>Matches zero or one occurrences of the preceding pattern.
Ex: -?[0-9]+ matches a signed number including an optional leading minus.</td></tr>
<tr><td>?</td><td>Matches zero or one occurrences of the preceding pattern.
Ex: -?[0-9]+ matches a signed number including an optional leading minus.</td></tr>
<tr><td>$</td><td>Matches end of line as the last character of the pattern.</td></tr>
<tr><td>{ }</td><td>1) Indicates how many times a pattern can be present. Example: A{1,3} implies one to three occurrences of A may be present.
2) If they contain name, they refer to a substitution by that name.
Ex: {digit}</td></tr>
<tr><td>\</td><td>Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
Ex: \n is a newline character, while “*” is a literal asterisk.</td></tr>
<tr><td>^</td><td>Negation.</td></tr>
<tr><td>|</td><td>Matches either the preceding regular expression or the following regular expression. Ex: cow|sheep|pig matches any of the three words.</td></tr>
<tr><td>"< symbols>"</td><td>Literal meanings of characters. Meta characters hold.</td></tr>
<tr><td>/</td><td>Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.</td></tr>
<tr><td>( )</td><td>Groups a series of regular expressions together into a new regular expression. Ex: (01) represents the character sequence 01. Parentheses are useful when building up complex patterns with *,+ and |</td></tr>
</table>

Examples of regular expressions

Regular expression	Meaning
joke[rs]	Matches either jokes or joker.
A{1,2}shis+	Matches AAshis, Ashis, AAshi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.
[0-9]	0 or 1 or 2 or………9
[0-9]+	1 or 111 or 12345 or …At least one occurrence of preceding exp
[0-9]*	Empty string (no digits at all) or one or more occurrence.
-?[0-9]+	-1 or +1 or +2 …..
[0.9]*\.[0.9]+	0.0,4.5 or .31415 But won’t match 0 or 2

Examples of token declarations

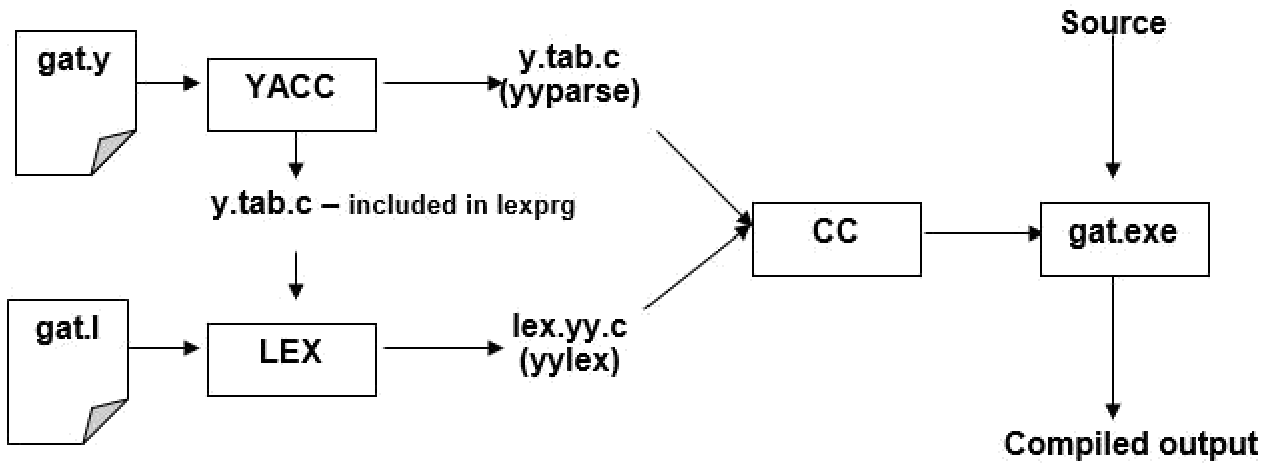
Token	**Associated expression**	**Meaning**
number	([0-9])+	1 or more occurrences of a digit
chars	[A-Za-z]	Any character
blank	" "	A blank space
word	(chars)+	1 or more occurrences of chars
variable	(chars)+(number)*(chars)*(number)*	

2. Introduction to YACC

YACC provides a general tool for imposing structure on the input to a computer program. The input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. YACC prepares a specification of the input process. YACC generates a function to control the input process. This function is called a parser.

The name is an acronym for "Yet Another Compiler Compiler". YACC generates the code for the parser in the C programming language. YACC was developed at AT& T for the Unix operating system. YACC has also been rewritten for other languages, including Java, Ada.

The function parser calls the lexical analyzer to pick up the tokens from the input stream. These tokens are organized according to the input structure rules .The input structure rule is called as grammar. When one of the rule is recognized, then user code supplied for this rule (user code is action) is invoked. Actions have the ability to return values and makes use of the values of other actions.



2.1 Steps in writing YACC Program:

1st step: Usinggedit editor create a file with extension y. For example: prg1.y

2nd Step: YACC –d prg1.y

3rd Step: lex prg1.l

4th Step: cc y.tab.c lex.yy.c -ll

5th Step: /a.out

When we run YACC, it generates a parser in file y.tab.c and also creates an include file y.tab.h. To obtain tokens, YACC calls yylex. Function yylex has a return type of int, and returns the token. Values associated with the token are returned by lex in variable yylval.

2.2 Structure of YACC source program:

Basic Specification:

Every YACC specification file consists of three sections. The declarations, Rules (of grammars), programs. The sections are separated by double percent "%%" marks. The % is generally used in YACC specification as an escape character.

The general format for the YACC file is very similar to that of the Lex file.

1st Col	2nd Col	3rd Col	4th Col
	DEFINITION SECTION		
%%			
	RULE SECTION		
%%			
	CODE SECTION		

%% is a delimiter to the mark the beginning of the Rule section.

Definition Section

%union	It defines the Stack type for the Parser. It is a union of various datas/structures/ objects
%token	These are the terminals returned by the yylex function to the YACC. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member>tokenName. Ex: %token NAME NUMBER
%type	The type of a non-terminal symbol in the Grammar rule can be specified with this.The format is %type <stack member>non-terminal.
%noassoc	Specifies that there is no associatively of a terminal symbol.
%left	Specifies the left associatively of a Terminal Symbol
%right	Specifies the right associatively of a Terminal Symbol.
%start	Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.
%prec	Changes the precedence level associated with a particular rule to that of the following token name or literal

The rules section simply consists of a list of grammar rules. A grammar rule has the form:

A: BODY

A represents a nonterminal name, the colon and the semicolon are YACC punctuation and BODY represents names and literals. The names used in the body of a grammar rule may represent tokens or nonterminal symbols. The literal consists of a character enclosed in single quotes.

Names representing tokens must be declared as follows in the declaration sections:
%token name1 name2…

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 no terminal symbols, one, called the start symbol has a particular importance. The parser is designed to recognize the start symbol. By default the start symbol is taken to be the left hand side of the first grammar rule in the rules section.

With each grammar rule, the user may associate actions to be. These actions may return values, and may obtain the values returned by the previous actions. Lexical analyzer can return values for tokens, if desired. An action is an arbitrary C statement. Actions are enclosed in curly braces.

3. Introduction to UNIX

Basic UNIX commands

Folder/Directory Commands and Options

Action	UNIX options & filespec
Check current Print Working Directory	**pwd**
Return to user's home folder	**cd**
Up one folder	**cd ..**
Make directory	**mkdir** proj1
Remove empty directory	**rmdir**/usr/sam
Remove directory-recursively	**rm -r**

File Listing Commands and Options

Action	UNIX options & filespec
List directory tree- recursively	**ls -r**
List last access dates of files, with hidden files	**ls -l -a**
List files by reverse date	**ls -t -r *.***
List files verbosely by size of file	**ls -l -s *.***
List files recursively including contents of other directories	**ls -R *.***
List number of lines in folder	**wc -l *.xtumlsed -n '$='**
List files with x anywhere in the name	**ls | grep x**

File Manipulation Commands and Options

Action	UNIX options&filespec
Create new(blank)file	**touch** *afilename*
Copy old file to new file. -p preserve file attributes(e.g. ownership and edit dates)-r copy recursively through directory structure -a archive, combines the flags-p – R and-d	**cp** old.filenew.file
Move old.file(**-i** interactively flag prompts before overwriting files)	**mv** –i old.file/tmp
Remove file(-intention)	**rm** –i sam.txt
Compare two files and show differences	**diff**

File Utilities

Action	UNIX options & filespec
View a file	**vi file.txt**
Concatenate files	**cat file1file2 to standard output.**
Counts-lines,-words, and- characters in a file	**wc** -l
Displays line-by-line differences between pairs of text files.	**diff**
calculator	**bc**
calendar for September, 1752 (when leap years began)	**cal** 9 1752

Controlling program execution for C-shell

&	Run job in background
^**c**	Kill job in foreground
^**z**	Suspend job in foreground
Fg	Restart suspended job in foreground
Bg	Run suspended job in background
;	Delimit commands on same line
()	Group commands on same line
!	re-run earlier commands from history list
jobs	List current jobs

Controlling program input/output for C-shell

|	Pipe output to input
>	Redirect output to a storage file
<	Redirect input from a storage file
>>	Append redirected output to a storage file
tee	Copy input to both file and next program in pipe
script	Make file record of al terminal activity

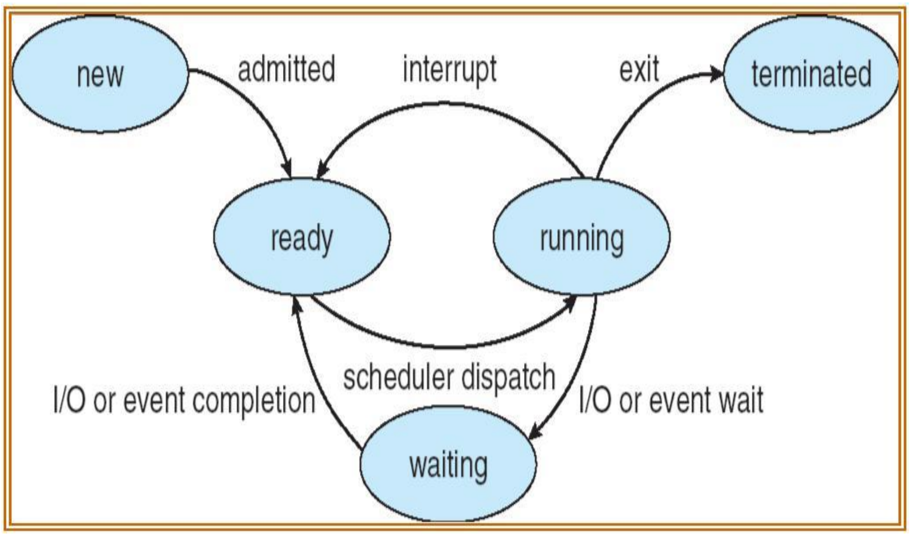
4. Introduction to Operating Systems

Introduction

An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

A Process is a program in execution. As a process executes, it changes *state*

- **New** : The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready** : The process is waiting to be assigned to a process
- **Terminated** : The process has finished execution



Apart from the program code, it includes the current activity represented by

- Program Counter,
- Contents of Processor registers,
- Process Stack which contains temporary data like function parameters, return addresses and local variables
- Data section which contains global variables
- Heap for dynamic memory allocation

A Multi-programmed system can have many processes running simultaneously with the CPU multiplexed among them. By switching the CPU between the processes, the OS can make the computer more productive. There is Process Scheduler which selects the process among many processes that are ready, for program execution on the CPU. Switching the CPU to another process requires performing a state save of the current process and a state restore of new process, this is Context Switch.

4.1 Scheduling Algorithms

CPU Scheduler can select processes from ready queue based on various scheduling algorithms. Different scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. The scheduling criteria include

- CPU utilization:
- Throughput: The number of processes that are completed per unit time.
- Waiting time: The sum of periods spent waiting in ready queue.
- Turnaround time: The interval between the time of submission of process to the time of completion.
- Response time: The time from submission of a request until the first response is produced.

The different scheduling algorithms are

- FCFS: First Come First Served Scheduling
- SJF: Shortest Job First Scheduling
- SRTF: Shortest Remaining Time First Scheduling
- Priority Scheduling
- Round Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

4.2 Deadlocks

A process requests resources; and if the resource is not available at that time, the process enters a waiting state. Sometimes, a waiting process is never able to change state, because the resource is has requested is held by another process which is also waiting. This situation is called Deadlock. Deadlock is characterized by four necessary conditions

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Deadlock can be handled in one of these ways,

- Deadlock Avoidance
- Deadlock Detection and Recover

5. Lab Syllabus Programs

1 a. Write a LEX program to recognize valid arithmetic expression. Identifiers in the expression could be only integers and operators could be + and *. Count the identifiers & operators present and print them separately.

```
1a)
%{
#include<stdio.h>
int v=0,op=0,id=0,flag=0;
%}
%%
[0-9]+ {id++;printf("\n identifier:");ECHO;}
[\+\-\*\/\=] {op++;printf("\n operator :");ECHO;}
"(" {v++;}
")" {v--;}
";" {flag=0;}
.|\n {;}
%%

main()
{
printf("enter the expression");
yylex();
if((op+1)==id && v==0)
printf("\n expression is valid\n");
else
printf("\n expression is invalid\n");
printf("no. of identifiers=%d\n",id);
printf("no. of operators=%d\n",op);
}
```

--OUTPUT--

```
Commands:
$lex 1a.l
$cc lex.yy.c -ll
$./a.out

1)Enter the expression 1+2
  Identifier:1
  Operator:+
  Identifier:2
 Press:ctrl+d
  Expression is vaild
  No.of identifier:2
  No.of operator:1
2) 1)Enter the expression (1+2
  Identifier:1
  Operator:(
  Identifier:2
  Operator:+
 Press:ctrl+d
  Expression is invaild
  No.of identifier:2
  No.of operator:2
```

b. Write YACC program to evaluate arithmetic expression involving operators: +, -, *, and / Lex Part

```
1b)
%{
#include<stdio.h>
%}
%token NUM UMIN
%left '+''-'
%left '*''/'
%left UMIN UPLUS
%%
line:
|line exp'\n'{printf("\n\n valid expresstion and value=%d\n\n",$2);
return 0;
}
exp:NUM{$$=$1;}
|exp'+'exp{$$=$1+$3;}
|exp'-'exp{$$=$1-$3;}
|exp'*'exp{$$=$1*$3;}
|exp'/'exp{if($3==0)
{printf("\n\n invaild\n\n");
return 0;
}
$$=$1/$3;
}
|'('exp')'{$$=$2;}
|'-'exp%prec UMIN{$$=-$2;}
|'+'exp%prec UPLUS{$$=+$2;}
%%
main()
{
printf("\n\n enter the expressions:\n\n");
yyparse();
return 0;
}
yyerror()
{
printf("\n\n invaild \n\n");
}
yylex()
{
char c=getchar();
if(isdigit(c))
{
ungetc(c,stdin);
scanf("%d",&yylval);
return NUM;
}
return c;
}
------------------------------Output------------------------------------
Commands:
Yacc -d 1b.y
Cc y.tab.c
./a.out
1)Enter the expression:3+6
  Vaild expression and value=9
2)enter the expression:
  2+*
  invalid
```

2. Develop, Implement and execute a program using YACC tool to recognize all strings ending with *b* preceded by *n a's* using the grammar $a^n b$ (note: input *n* value).

```
2)
%{
#include<stdio.h>
#include<string.h>
#include<ctype.h>
int valid=0,n=0;
%}
%%
line:
|line an'b''\n'{
if(n>=0)valid=1;
return;
}
an:
|'a'an{n++;}
;
%%
main()
{
printf("\n enter the string:\n");
yyparse();
if(valid)
printf("the string is valid and number of a's=%d",n);
else
printf("\n the string is invalid and number of a's=%d",n);
}
yylex()
{
int c=getchar();
return;
}
yyerror()
{
printf("invalid input\n");
return;
}
--------------------------Output-----------------------------------------
Commands:
Yacc -d 2.y
Cc y.tab.c
./a.out

1)enter the string:aab
  The string is valid and number of a's=2

2)enter the string:bba
  The string is invalid and number of a's=1
```

3. Design, develop and implement YACC/C program to construct *Predictive / LL(1) Parsing Table* for the grammar rules*: A →aBa* , *B →bB* | *ε*. Use this table to parse the sentence*: abba$*.

```
3)
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char ip[20],stack[20];
int main()
{
char m[2][3][10]={{"aBa","E","E"},{"n","bB","n"}};
int size[2][3]={3,1,1,1,2,1};
int i,j,k,n,row,col;
printf("\n enter the input string:");
scanf("%s",ip);
strcat(ip,"$");
n=strlen(ip);
stack[0]='$';
stack[1]='A';
i=1;
j=0;
printf("\n stack\t\t input\n");
printf("______________\t\t_______________________\n");
while((stack[i]!='$') && (ip[j]!='$'))
{
if(stack[i]==ip[j])
{
i--;
j++;
}
switch(stack[i])
{
case 'A':row=0;
break;
case 'B':row=1;
break;
}
switch(ip[j])
{
case'a':col=0;
if(stack[i]=='$')
{
printf("error");
exit(0);
}
break;
case'b':col=1;
if(stack[i]=='$')
{
printf("error");
exit(0);
}
break;
case'$':col=2;
if(stack[i]!='$')
{
printf("error");
exit(0);
}
break;
```

```
}
if(m[row][col][0]=='E')
{
printf("\nERROR");
exit(0);
}
else if(m[row][col][0]=='n')
i--;
else if(m[row][col][0]==ip[j])
{
for(k=size[row][col]-1;k>=0;k--)
{
stack[i]=m[row][col][k];
i++;
}
i--;
}
for(k=0;k<=i;k++)
printf("%c",stack[k]);
printf("\t\t");
for(k=j;k<=n;k++)
printf("%c",ip[k]);
printf("\n");
}
k--;
if(i<0)
printf("\nSUCCESS");
else
printf("\nerror");
return 0;
}
```

```
------------------------Output------------------------------------------

[root@localhost ~]# gedit 3.c
[root@localhost ~]# cc 3.c
[root@localhost ~]# ./a.out

 1)enter the input string:abba

 stack           input
______________         _______________________
$aBa            abba$
$aBb            bba$
$aBb            ba$
$a              a$
                $

SUCCESS

 2)enter the input string:abab

 stack           input
______________         _______________________
$aBa            abab$
$aBb            bab$
$a              ab$
error
```

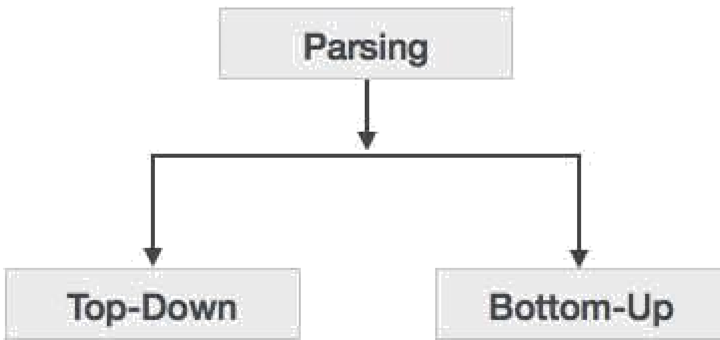
---

4. Design, develop and implement YACC/C program to demonstrate ***Shift Reduce Parsing*** technique for the grammar rules: ***E →E+T | T, T →T*F | F, F → (E) | id*** and parse the sentence***: id + id * id***.

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

A parser's main purpose is to determine if input data may be derived from the start symbol of the grammar.

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.



Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- **Recursive descent parsing**: It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- **Backtracking**: It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

Bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Shift-reduce Parsing (Bottom-up Parsing)

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of “reducing” (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar. At every (reduction) step, a particular substring matching the RHS of a production rule is replaced by the symbol on the LHS of the production.

A general form of shift-reduce parsing is **LR** (scanning from **L**eft to right and using **R**ight-most derivation in reverse) parsing, which is used in a number of automatic parser generators like Yacc, Bison, etc.

```
4)
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
{
printf("GRAMMER is E->E+T|T\n T->T*F|F\n F->(E)\n F->id");
printf("enter input string");
scanf("%s",a);
c=strlen(a);
strcpy(act,"SHIFT->");
puts("stack\t input\t action");
for(k=0,i=0;j<c;k++,i++,j++)
{
if(a[j]=='i' && a[j+1]=='d')
{
stk[i]=a[j];
stk[i+1]=a[j+1];
stk[i+2]='\0';
a[j]=' ';
a[j+1]=' ';
printf("\n$%s\t%s$\t%sid",stk,a,act);
check();
}
else
{
stk[i]=a[j];
stk[i+1]='\0';
a[j]=' ';
printf("\n$%s\t%s$\t%s symbols",stk,a,act);
check();
}
}
return 0;
}
void check()
```

```
{
strcpy(ac,"REDUCE");
for(z=0;z<c;z++)
if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
{
stk[z]='F';
stk[z+1]='\0';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}
for(z=0;z<c;z++)
if(stk[z]=='i' && stk[z+1]=='d')
{
stk[z]='F';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
j++;
}
for(z=0;z<c;z++)
{
if(stk[z]=='T' && stk[z+1]=='*' && stk[z+2]=='F')
{
stk[z]='T';
stk[z+1]='\0';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}
else if(stk[z]=='F')
{
stk[z]='T';
printf("\n$%s\t%s$\t%s",stk,a,ac);
}
}
for(z=0;z<c;z++)
{
if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='T' && stk[z+3]=='*')
break;
if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='T')
if(a[j+1]=='*')
break;
else
{
stk[z]='E';
stk[z+1]='\0';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}
else if(stk[z]=='T')
{
stk[z]='E';
printf("\n$%s\t%s$\t%s",stk,a,ac);
}
}
}
```

--------------------------------OUTPUT-----------------------------------

```
[root@localhost ~]# cc 4.c
[root@localhost ~]# ./a.out
GRAMMER is E->E+T|T
 T->T*F|F
 F->(E)
 F->identer input string id+id*id
stack    input   action

$id       +id*id$       SHIFT->id
$F        +id*id$       REDUCE
$T        +id*id$       REDUCE
$E        +id*id$       REDUCE
$E+        id*id$       SHIFT-> symbols
$E+id        *id$       SHIFT->id
$E+F         *id$       REDUCE
$E+T         *id$       REDUCE
$E+T*         id$       SHIFT-> symbols
$E+T*id         $       SHIFT->id
$E+T*F          $       REDUCE
$E+T            $       REDUCE
$E+T            $       REDUCE
$E              $       REDUCE
$E              $       REDUCE
$E              $       REDUCE
```

---

5. Design, develop and implement a C/Java program to generate the machine code using ***Triples*** for the statement ***A = -B * (C +D)*** whose intermediate code in three-address form:

T1 = -B

T2 = C + D

T3 = T1 *

T2 A = T3

```
5)
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
char op[2],arg1[5],arg2[5],result[5];
int main()
{
FILE*fp1,*fp2;
fp1=fopen("input.txt","r");
fp2=fopen("output.txt","w");
while(!feof(fp1))
{
fscanf(fp1,"%s%s%s%s",result,arg1,op,arg2);
if(strcmp(op,"+")==0)
{
fprintf(fp2,"\n MOV R0,%s",arg1);
fprintf(fp2,"\n ADD R0,%s",arg2);
fprintf(fp2,"\n MOV %s,R0",result);
}
if(strcmp(op,"*")==0)
{
fprintf(fp2,"\n MOV R0,%s",arg1);
fprintf(fp2,"\n MUL R0,%s",arg2);
fprintf(fp2,"\n MOV %s,R0",result);
}
if(strcmp(op,"-")==0)
{
fprintf(fp2,"\n MOV R0,%s",arg1);
fprintf(fp2,"\n SUB R0,%s",arg2);
fprintf(fp2,"\n MOV %s,R0",result);
}
if(strcmp(op,"/")==0)
{
fprintf(fp2,"\n MOV R0,%s",arg1);
fprintf(fp2,"\n DIV R0,%s",arg2);
fprintf(fp2,"\n MOV %s,R0",result);
}
if(strcmp(op,"=")==0)
{
fprintf(fp2,"\n MOV R0,%s",arg1);
fprintf(fp2,"\n MOV %s,R0",result);
}
}
fclose(fp1);
fclose(fp2);
return 0;
}
```

```
---------------------------Output---------------------------------------
```

```
[root@localhost ~]# gedit 5.c
[root@localhost ~]# cc 5.c
[root@localhost ~]# gedit input.txt
------------------------------------------
T1 -B = ?
T2 C + D
T3 T1 * T2
A T3 = ?
------------------------------------------
[root@localhost ~]# cc 5.c
[root@localhost ~]# ./a.out input.txt
[root@localhost ~]# cat output.txt
---------------------------------------------
 MOV R0,-B
 MOV T1,R0
 MOV R0,C
 ADD R0,D
 MOV T2,R0
 MOV R0,T1
 MUL R0,T2
 MOV T3,R0
 MOV R0,T3
 MOV A,R0
 MOV R0,T3
 MOV A,R0
```

Output:

input.txt	**output.txt**

T1 -B = ?
T2 C + D
T3 T1 * T2
A T3 = ?
	MOV R0,-B
	MOV T1,R0
	MOV R0,C
	ADD R0,D
	MOV T2,R0
	MOV R0,T1
	MUL R0,T2
	MOV T3,R0
	MOV R0,T3
	MOV A,R0

6. a) Write a LEX program to eliminate *comment lines* in a *C* program and copy the resulting program into a separate file.

```
6a)
%{
#include<stdio.h>
int c_count=0;
%}
%%
"/*"[^*/]*"*/" {c_count++;}
"//".* {c_count++;}
%%
int main(int argc,char**argv)
{
FILE*f1,*f2;
if(argc>1)
{
f1=fopen(argv[1],"r");
if(!f1)
{
printf("file error\n");
exit(1);
}
yyin=f1;
f2=fopen(argv[2],"w");
if(!f2)
{
printf("error");
exit(1);
}
yyout=f2;
yylex();
printf("number of comment lines:%d\n",c_count);
}
return 0;
}
------------------------------output-------------------------------------
Commands:
Lex 6a.l
cc lex.yy.c -ll
gedit input file1.c
----------------------------------------------------------------
#include<stdio.h>
//header line
void main()
{
/* main starts here
and program continues*/
/*ffutftufu*/
/*ggffgnhnj*/
printf("hi");
}
```

```
./a.out file.c file2.c
Number of comments lines:4
----------------------------------------------------------------------
```

```
Gedit file2.c
-------------------------
#include<stdio.h>

void main()
{



printf("hi");
}
```

b) Write YACC program to recognize valid *identifier*, *operators* and *keywords* in the given text (C program) file.

```
6b)
lex part:

%{
#include<stdio.h>
#include"y.tab.h"
extern yylval;
%}
%%
[\t];
[+|-|*\/|=\<\>]  {printf("operator is %s\n",yytext);return OP;}
[0-9]+ {yylval=atoi(yytext); printf("number is %d\n",yylval); return DIGIT;}
int|char|bool|float|void|for|do|while|if|else|return|void {printf("keyword is
%s\n",yytext);return
KEY;}
[a-zA-Z0-9]+ {printf("identifier is %s\n",yytext);return ID;}
.;
%%




yacc part:

%{
#include<stdio.h>
#include<stdlib.h>
int id=0,dig=0,key=0,op=0;
%}
%token DIGIT ID KEY OP
%%
input:
DIGIT input { dig++; }
|ID input { id++; }
|KEY input { key++; }
|OP input { op++; }
|DIGIT { dig++; }
|ID { id++; }
|KEY { key++; }
|OP { op++; }
;
%%
#include<stdio.h>
extern int yylex();
```

```
extern int yyparse();
extern FILE *yyin;
main() {
FILE *myfile=fopen("sam_input.c","r");
if(!myfile) {
printf("i cant open sam_input.c!");
return -1;
}
yyin=myfile;
do {
yyparse();
} while(!feof(yyin));
printf("numbers=%d\n keywords=%d\n identifiers=%d\n
operators=%d\n",dig,key,id,op);
}
void yyerror() {
printf("EEK,parse error!Message:");
exit(-1);
}
```

```
------------------output:------------------------------------
type and save a file with .c extention:

void main()
{
float a123;
char  a ;
char   b123 ;
char b ;
if ( sum == 10 )
printf (" pass ");
else
printf (" fail ");
}
-------------------------------------------------------
[root@localhost ~]# lex 6b.l
[root@localhost ~]# yacc -d 6b.y
[root@localhost ~]# cc y.tab.c lex.yy.c -ll
[root@localhost ~]# ./a.out
```

```
keyword is void
 identifier is main
()
{
keyword is float
 identifier is a123
;
keyword is char
  identifier is a

keyword is char
   identifier is b123

keyword is char
```

```
 identifier is b

keyword is if
 ( identifier is sum
 operator is =
operator is =
 number is 10
 )
identifier is printf
 (" identifier is pass
 "
keyword is else

identifier is printf
 (" identifier is fail
 "
}

 numbers=1
 keywords=7
 identifiers=10
 operators=2
```

7. Design, develop and implement a C/C++/Java program to simulate the working of ***Shortest remaining time*** and ***Round Robin (RR)*** scheduling algorithms. Experiment with different quantum sizes for RR algorithm.

Round-robin (RR) is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices (also known as time quanta) are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks. It is an operating system concept.

The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

```
#include<stdio.h>
struct proc
{
	int id;
	int arrival;
	int burst;
	int rem;
	int wait;
	int finish;
	int turnaround;
	float ratio;
}process[10]; //structure to hold the process information
struct proc temp;
int no;

int chkprocess(int);
int nextprocess();
 void roundrobin(int, int, int[], int[]);
 void srtf(int);
main()
{
	int n,tq,choice;
	int bt[10],st[10],i,j,k;
for(; ;)
{
printf("Enter the choice \n");
printf(" 1. Round Robin\n 2. SRT\n 3. Exit \n");
```

```
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Round Robin scheduling algorithm\n");
printf("Enter number of processes:\n");
  scanf("%d",&n);
  printf("Enter burst time for sequences:");
  for(i=0;i<n;i++)
   {
     scanf("%d",&bt[i]);
     st[i]=bt[i];          //service time
   }
   printf("Enter time quantum:");
   scanf("%d",&tq);
   roundrobin(n,tq,st,bt);
   break;

case 2:
          printf("\n \n ---SHORTEST REMAINING TIME NEXT---\n \n ");
          printf("\n \n Enter the number of processes: "); scanf("%d", &n);

          srtf(n);
break;

case 3: exit(0);
}// end of switch
}// end of for
}//end of main()

void roundrobin(int n,int tq,int st[],int bt[])
 {
          int time=0;
          int tat[10],wt[10],i,count=0,swt=0,stat=0,temp1,sq=0,j,k;
          float awt=0.0,atat=0.0;
          while(1)
          {
                    for(i=0,count=0;i<n;i++)
                    {
      temp1=tq;
      if(st[i]==0) // when service time of a process equals zero then
            //count value is incremented
      {
          count++;
```

```
            continue;
        }
        if(st[i]>tq) // when service time of a process greater than
              time //quantum then time
                    st[i]=st[i]-tq; //quantum value subtracted from service time
        else
          if(st[i]>=0)
          {
                temp1=st[i];      // temp1 stores the service time of a process
                st[i]=0;           // making service time equals 0
          }
          sq=sq+temp1;      // utilizing temp1 value to calculate turnaround time
          tat[i]=sq;            // turn around time
      } //end of for
      if(n==count)          // it indicates all processes have completed their task
                              because the count value
      break;                  // incremented when service time equals 0
    } //end of while

    for(i=0;i<n;i++)      // to calculate the wait time and turnaround time of each
process
    {
     wt[i]=tat[i]-bt[i];    // waiting time calculated from the turnaround time - burst
                             time
     swt=swt+wt[i];      // summation of wait time
     stat=stat+tat[i];     // summation of turnaround time
    }
    awt=(float)swt/n;        // average wait time
    atat=(float)stat/n;       // average turnaround time
    printf("Process_no         Burst time    Wait time     Turn around time\n");
    for(i=0;i<n;i++)
     printf("%d\t\t%d\t\t%d\t\t%d\n",i+1,bt[i],wt[i],tat[i]);
     printf("Avg wait time is %f\n Avg turn around time is %f\n",awt,atat);
 }// end of Round Robin
int chkprocess(int s)       // function to check process remaining time is zero or not
{
          int i;
          for(i = 1; i <= s; i++)
          {
                    if(process[i].rem != 0)
                              return 1;
          }
          return 0;
} // end of chkprocess
```

```
int nextprocess()       // function to identify the next process to be executed
{
        int min, l, i;
        min = 32000;  //any limit assumed
        for(i = 1; i <= no; i++)
        {
                if( process[i].rem!=0 && process[i].rem < min)
                {
                        min = process[i].rem;
                        l = i;
                }
        }
        return l;
} // end of nextprocess

void srtf(int n)
{
int i,j,k,time=0;
float tavg,wavg;
for(i = 1; i <= n; i++)
        {
                process[i].id = i;
                printf("\n\nEnter the arrival time for process %d: ", i);
                scanf("%d", &(process[i].arrival));
                printf("Enter the burst time for process %d: ",
                i); scanf("%d", &(process[i].burst));
                process[i].rem = process[i].burst;
        }
        for(i = 1; i <= n; i++)
        {
                for(j = i + 1; j <= n; j++)
                {
                        if(process[i].arrival > process[j].arrival)   // sort arrival time of a
process
                        {
                                temp = process[i];
                                process[i] = process[j];
                                process[j] = temp;
                        }
                }
        }

        no = 0;
        j = 1;
```

```
                              while(chkprocess(n) == 1)
    {
        if(process[no + 1].arrival == time)
    {
        while(process[no+1].arrival==time)
                    no++;
            if(process[j].rem==0)
                    process[j].finish=time;
            j = nextprocess();
        }
        if(process[j].rem != 0)         // to calculate the waiting time of a process
        {
            process[j].rem--;
            for(i = 1; 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 FIRST---");
    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 = 1; i <= n; i++)
       {
         process[i].turnaround = process[i].wait + process[i].burst; // calc of turnaround
        process[i].ratio = (float)process[i].turnaround / (float)process[i].burst;

         printf("%5d %8d %7d %8d %10d %9d %10.1f ", process[i].id, process[i].arrival,
        process[i].burst, process[i].wait, process[i].finish, process[i].turnaround,
        process[i].ratio);
        tavg=tavg+ process[i].turnaround;    //summation of turnaround time
        wavg=wavg+process[i].wait;           // summation of waiting time
         printf("\n\n");
```

```
    }

    tavg=tavg/n;       // average turnaround time
    wavg=wavg/n;     // average wait time
     printf("tavg=%f\t wavg=%f\n",tavg,wavg);
}// end of srtf
```

Output:

```
Enter the choice
 1) Round Robin 2) SRT
 3) Exit
1
Round Robin scheduling algorithm
**********************************
Enter number of processes:3
Enter burst time for sequences:24
3
3

Enter time quantum:4
 Process_no     Burst time        Wait time    Turnaround time
        1              24                6             30
        2              3                 4             7
        3              3                  7           10
Avg wait time is 5.666667
Avg turnaround time is 15.666667
Enter the choice
 1) Round Robin 2) SRT
 3) Exit
2
 ---SHORTEST REMAINING TIME NEXT---
 Enter the number of processes: 4
Enter the arrival time for process 1: 0
Enter the burst time for process 1: 8
Enter the arrival time for process 2: 1
Enter the burst time for process 2: 4
Enter the arrival time for process 3: 2
Enter the burst time for process 3: 9
Enter the arrival time for process 4: 3
Enter the burst time for process 4: 5
  1          24           6           30
  2            3            4          7
  3            3            7          10
 ---SHORTEST REMAINING TIME FIRST---
 Enter the number of processes: 4
```

Enter the arrival time for process 1: 0
Enter the burst time for process 1: 8
Enter the arrival time for process 2: 1
Enter the burst time for process 2: 4
Enter the arrival time for process 3: 2
Enter the burst time for process 3: 9
Enter the arrival time for process 4: 3
Enter the burst time for process 4: 5

---SHORTEST REMAINING TIME NEXT---

Process id	Arrival time	Burst time	Waiting time	Finishing time	turnaround time	Tr/Tb
1	0	8	9	17	17	2.1
2	1	4	0	5	4	1.0
3	2	9	15	26	24	2.7
4	3	5	2	10	7	1.4

tavg=13.000000
wavg=6.500000
Using OpenMP

8. Design, develop and implement a C/C++/Java program to implement ***Banker's algorithm***. Assume suitable input required to demonstrate the results.

The **Banker's algorithm**, sometimes referred to as the **detection algorithm**, is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The algorithm was developed in the design process for the operating system and originally described (in Dutch) in EWD108. When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

```
8.
#include<stdio.h>
#include<stdlib.h>
int main()
{
int
Max[10][10],need[10][10],alloc[10][10],avail[10],completed[10],safeSequence[10];
int p,i,j,r,process,count;
count=0;
printf("enter the no of processes:");
scanf("%d",&p);
for(i=0;i<p;i++)
completed[i]=0;
printf("\n\n enter the no of resources:");
scanf("%d",&r);
printf("\n\n enter the MaxMatrix for each process:");
for(i=0;i<p;i++)
{
printf("\n For process %d:",i+1);
for(j=0;j<r;j++)
scanf("%d",&Max[i][j]);
}
printf("\n\n enter the allocation for each process:");
for(i=0;i<p;i++)
{
printf("\n for process %d:",i+1);
for(j=0;j<r;j++)
scanf("%d",&alloc[i][j]);
}
printf("\n\n enter the available resources:");
for(i=0;i<r;i++)
scanf("%d",&avail[i]);
for(i=0;i<p;i++)
for(j=0;j<r;j++)
need[i][j]=Max[i][j]-alloc[i][j];
do
{
printf("\n Maxmatrix:\t allocation matrix:\n");
for(i=0;i<p;i++)
```

```
{
for(j=0;j<r;j++)
printf("%d",Max[i][j]);
printf("\t\t");
for(j=0;j<r;j++)
printf("%d",alloc[i][j]);
printf("\n");
}
process=-1;
for(i=0;i<p;i++)
{
if(completed[i]==0)
{
process=i;
for(j=0;j<r;j++)
{
if(avail[j]<need[i][j])
{
process=-1;
break;
}
}
}
if(process!=-1)
break;
}
if(process!=-1)
{
printf("\n process %d runs to completion ",process+1);
safeSequence[count]=process+1;count++;
for(j=0;j<r;j++)
{
avail[j]+=alloc[process][j];
alloc[process][j]=0;
Max[process][j]=0;
completed[process]=1;
}
}
}
while(count!=p && process!=-1);
if(count==p)
{
printf("\n the system is in a safe state!!\n");
printf("Safe Sequence:<");
for(i=0;i<p;i++)
printf("%d",safeSequence[i]);
printf(">\n");
}
else
printf("\n the system is in an unsafe state!!");
}
output:
[root@localhost ~]# cc uma8.c
[root@localhost ~]# ./a.out
enter the no of processes:5
 enter the no of resources:3
 enter the MaxMatrix for each process:
 For process 1:7
5
3
 For process 2:3
```

```
2
2
 For process 3:7
0
2
 For process 4:2
2
2
 For process 5:4
3
3
 enter the allocation for each process:
 for process 1:0
1
0
 for process 2:2
0
0
 for process 3:3
0
2
 for process 4:2
1
1
 for process 5:0
0
2
 enter the available resources:3
3
2
 Maxmatrix:      allocation matrix:
753             010
322             200
702             302
222             211
433             002
 process 2 runs to completion
 Maxmatrix:      allocation matrix:
753             010
000             000
702             302
222             211
433             002
 process 3 runs to completion
 Maxmatrix:      allocation matrix:
753             010
000             000
000             000
222             211
433             002
 process 4 runs to completion
 Maxmatrix:      allocation matrix:
753             010
000             000
000             000
000             000
433             002
 process 1 runs to completion
 Maxmatrix:      allocation matrix:
000             000
000             000
```

```
000             000
000             000
433             002
 process 5 runs to completion
 the system is in a safe state!!
Safe Sequence:<23415>
```

____

9. Design, develop and implement a C/C++/Java program to implement ***page replacement algorithms LRU*** and ***FIFO***. Assume suitable input required to demonstrate the results.

In a computer operating system that uses paging for virtual memory management, **page replacement algorithms** decide which memory pages to page out, sometimes called swap out, or write to disk, when a page of memory needs to be allocated. Page replacement happens when a requested page is not in memory (page fault) and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the *quality* of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself.

The page replacing problem is a typical online problem from the competitive analysis perspective in the sense that the optimal deterministic algorithm is known.

```
9.
#include<stdio.h>
#include<stdlib.h>
void FIFO(char[],char[],int,int);
void lru(char[],char[],int,int);
void opt(char[],char[],int,int);
int main()
{
int ch,YN=1,i,l,f;
char F[10],s[25];
printf("\n\n enter the no of empty frames:");
scanf("%d",&f);
printf("\n\n enter the length of string:");
scanf("%d",&l);
printf("\n\n\t enter the string:");
scanf("%s",s);
for(i=0;i<f;i++)
F[i]=-1;
do
{
printf("\n\n\t***********MENU**********\n");
printf("\n\n\t 1:FIFO\n\n\t 2:LRU\n\n\t 4:EXIT");
printf("\n\n\t enter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
for(i=0;i<f;i++)
{
F[i]=-1;
}
```

```
FIFO(s,F,l,f);
break;
case 2:
for(i=0;i<f;i++)
{
F[i]=-1;
}
lru(s,F,l,f);
break;
case 4:
exit(0);
}
printf("\n\n\t Do you want to continue IF YES PRESS 1\n\n\t IF NO PRESS 0:");
scanf("%d",&YN);
}while(YN==1);return(0);
}
void FIFO(char s[],char F[],int l,int f)
{
int i,j=0,k,flag=0,cnt=0;
printf("\n\tPAGE\t FRAMES\t FAULTS");
for(i=0;i<l;i++)
{
for(k=0;k<f;k++)
{
if(F[k]==s[i])
flag=1;
}
if(flag==0)
{
printf("\n\t %c\t",s[i]);
F[j]=s[i];
j++;
for(k=0;k<f;k++)
{
printf("%c",F[k]);
}
printf("\t page-fault %d",cnt);
cnt++;
}
else
{
flag=0;
printf("\n\t %c\t",s[i]);
for(k=0;k<f;k++)
{
printf("%c",F[k]);
}
printf("\t no page-fault");
}
if(j==f)
j=0;
}
}
void lru(char s[],char F[],int l,int f)
{
int i,j=0,k,m,flag=0,cnt=0,top=0;
printf("\n\t PAGE \t FRAMES \t FAULTS");
for(i=0;i<l;i++)
{
for(k=0;k<f;k++)
{
```

```
if(F[k]==s[i])
{
flag=1;
break;
}
}
printf("\n\t %c\t",s[i]);
if(j!=f && flag!=1)
{
F[top]=s[i];
j++;
if(j!=f)
top++;
}
else
{
if(flag!=1)
{
for(k=0;k<top;k++)
{
F[k]=F[k+1];
}
F[top]=s[i];
}
if(flag==1)
{
for(m=k;m<top;m++)
{
F[m]=F[m+1];
}
F[top]=s[i];
}
}
for(k=0;k<f;k++)
{
printf("%c",F[k]);
}
if(flag==0)
{
printf("\t page-fault %d",cnt);
cnt++;
}
else
printf("\t no page fault");
flag=0;
}
}
output:
[root@localhost ~]# gedit uma9.c
[root@localhost ~]# cc uma9.c
[root@localhost ~]# ./a.out
 enter the no of empty frames:3
 enter the length of string:5
         enter the string:hello
        ***********MENU***********
         1:FIFO
         2:LRU
         4:EXIT
         enter your choice:1
        PAGE     FRAMES  FAULTS
         h      hï¿½ï¿½      page-fault 0
```

```
 e      heï¿½      page-fault 1
 l      hel      page-fault 2
 l      hel      no page-fault
 o      oel      page-fault 3
 Do you want to continue IF YES PRESS 1
 IF NO PRESS 0:1
************MENU***********
 1:FIFO
 2:LRU
 4:EXIT
 enter your choice:2
 PAGE    FRAMES          FAULTS
 h      hï¿½ï¿½      page-fault 0
 e      heï¿½      page-fault 1
 l      hel      page-fault 2
 l      hel      no page fault
 o      elo      page-fault 3
 Do you want to continue IF YES PRESS 1
 IF NO PRESS 0:4
```

6. Viva Questions

- **Define system software**.
  System software is computer software designed to operate the computer hardware and to provide a platform for running application software. Eg: operating system, assembler, and loader.
- **What is an Assembler?**
  Assembler for an assembly language, a computer program to translate between lower-level representations of computer programs.
- **Explain lex and yacc tools**
  - Lex: - scanner that can identify those tokens
  - Yacc: - parser.yacc takes a concise description of a grammar and produces a C routine that can parse that grammar.
- **Explain yyleng?**
  Yyleng-contains the length of the string our lexer recognizes.
- **What is a Parser?**
  A Parser for a Grammar is a program which takes in the Language string as it's input and produces either a corresponding Parse tree or an Error.
- **What is the Syntax of a Language?**
  The Rules which tells whether a string is a valid Program or not are called the Syntax.
- **What is the Semantics of a Language?**
  The Rules which gives meaning to programs are called the Semantics of a Language.
- **What are tokens?**
  When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the Language.
- **What is the Lexical Analysis?**
  The Function of a lexical Analyzer is to read the input stream representing the Source program, one character at a time and to translate it into valid tokens.
- **How can we represent a token in a language?**

The Tokens in a Language are represented by a set of Regular Expressions. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters. The Advantage of using regular expression is that a recognizer can be automatically generated.

- **How are the tokens recognized?**

  The tokens which are represented by an Regular Expressions are recognized in an input string by means of a state transition Diagram and Finite Automata.

- **Are Lexical Analysis and Parsing two different Passes?**

  These two can form two different passes of a Parser. The Lexical analysis can store all the recognized tokens in an intermediate file and give it to the Parser as an input. However it is more convenient to have the lexical Analyzer as a co routine or a subroutine which the Parser calls whenever it requires a token.

- **How do we write the Regular Expressions?**
  The following are the most general notations used for expressing a R.E.

Symbol	Description
|	OR (alternation)
()	Group of Subexpression
*	0 or more Occurrences
?	0 or 1 Occurrence
+	1 or more Occurrences
{n,m}	n-m Occurrences

- **What are the Advantages of using Context-Free grammars?**
  ⇒ It is precise and easy to understand.
  ⇒ It is easier to determine syntactic ambiguities and conflicts in the grammar.
- **If Context-free grammars can represent every regular expression, why do one needs R.E at all?**
  ⇒ Regular Expression are Simpler than Context-free grammars.
  ⇒ It is easier to construct a recognizer for R.E than Context-Free grammar.
  ⇒ Breaking the Syntactic structure into Lexical & non-Lexical parts provide better front end for the Parser.
  ⇒ R.E are most powerful in describing the lexical constructs like identifiers, keywords etc while Context-free grammars in representing the nested or block structures of the Language.
- **What are the Parse Trees?**
  Parse trees are the Graphical representation of the grammar which filters out the choice for replacement order of the Production rules.
- **What are Terminals and non-Terminals in a grammar?**
  Terminals:- All the basic symbols or tokens of which the language is composed of are called Terminals. In a Parse Tree the Leafs represents the Terminal Symbol.
  Non-Terminals:- These are syntactic variables in the grammar which represents a set of strings the grammar is composed of. In a Parse tree all the inner nodes represents the Non-Terminal symbols.
- **What are Ambiguous Grammars?**
  A Grammar that produces more than one Parse Tree for the same sentences or the Production rules in a grammar is said to be ambiguous.
- **What is bottom up Parsing?**
  The Parsing method is which the Parse tree is constructed from the input language string beginning from the leaves and going up to the root node.
  Bottom-Up parsing is also called shift-reduce parsing due to its implementation. The YACC supports shift-reduce parsing.
- **What is the need of Operator precedence?**
  The shift reduce Parsing has a basic limitation. Grammars which can represent a left-sentential parse tree as well as right-sentential parse tree cannot be handled by shift reduce parsing. Such a grammar ought to have two non-terminals in the production rule. So the Terminal sandwiched between these

two non-terminals must have some associability and precedence. This will help the parser to understand which non-terminal would be expanded first.

- **What is exit status command?**
  Exit 0- return success, command executed successfully. Exit 1 – return failure.
- **Define API's**
  An application programming interface (API) is a source code based specification intended to be used as an interface by software components to communicate with each other.