VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Belgaum-590014, Karnataka.

SYSTEM SOFTWARE & OPERATING SYSTEM LABORATORY

(18CSL66)

ACHARYA INSTITUTE OF TECHNOLOGY

Soladevanahalli, Bangalore

SYSTEM SOFTWARE AND OPERATING SYSTEM LABORATORY [As per Choice Based Credit System (CBCS) scheme] (Effective from the academic year 2016 -2017) SEMESTER – VI

Subject Code	18CSL66	IA Marks	40
Number of Lecture Hours/Week	01I + 02P	Exam Marks	60
Total Number of Lecture Hours	40	Exam Hours	03

CREDITS – 02

Course objectives: This course will enable students to

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

To enable students to learn different types of CPU scheduling algorithms used in operating system.

To make students able to implement memory management - page replacement and deadlock handling algorithms

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

Lab 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 |*** . 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: The students should be able to:

Implement and demonstrate Lexer's and Parser's

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:

```
{definitions}
%%
{rules}
%%
{user subroutines/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.
*****	Match zero or more occurrences of the preceding pattern.
+	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.
?	Matches zero or one occurrences of the preceding pattern. Ex: -?[0-9]+ matches a signed number including an optional leading minus.
$	Matches end of line as the last character of the pattern.
{ }	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}
\	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.
^	Negation.
|	Matches either the preceding regular expression or the following regular expression. Ex: cow|sheep|pig matches any of the three words.
"< symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	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 |

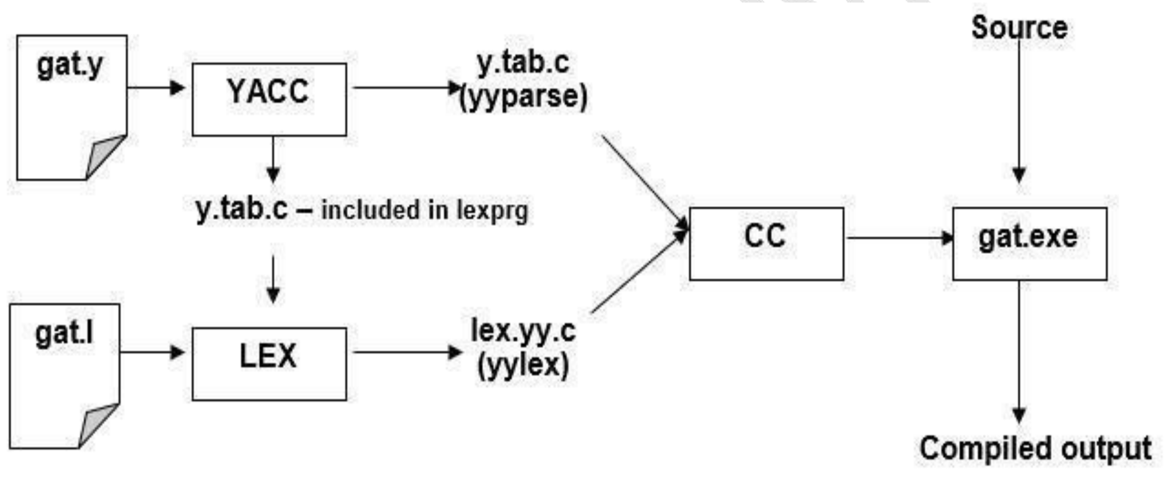
Regular expression	**Meaning**
joke[rs]	Matches either jokes or joker.
A{1,2}shis+	Matches AAshis, Ashis, AAshi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.
[0-9]	0 or 1 or 2 or………9
[0-9]+	1 or 111 or 12345 or …At least one occurrence of preceding exp
[0-9]*	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 .31417 But won"t match 0 or 2

1. 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:	Using gedit editor create a file with extension y. For example: gedit prg1.y
2nd Step:	YACC –d prg1.y
3rd Step:	lex prg1.l
4th Step:	cc y.tab.c lex.yy.c -
5th Step:	ll /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.

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

%% 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

Rules Section

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 bethe left hand side of the first grammar rule in the rules section.

2. 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 copyrecursivelythroughdirectory 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
View a file	**vi file.txt**
Concatenate files	**cat file1file2 to standard output.**
Counts-lines,-words, and- characters in a file	**wc** –l

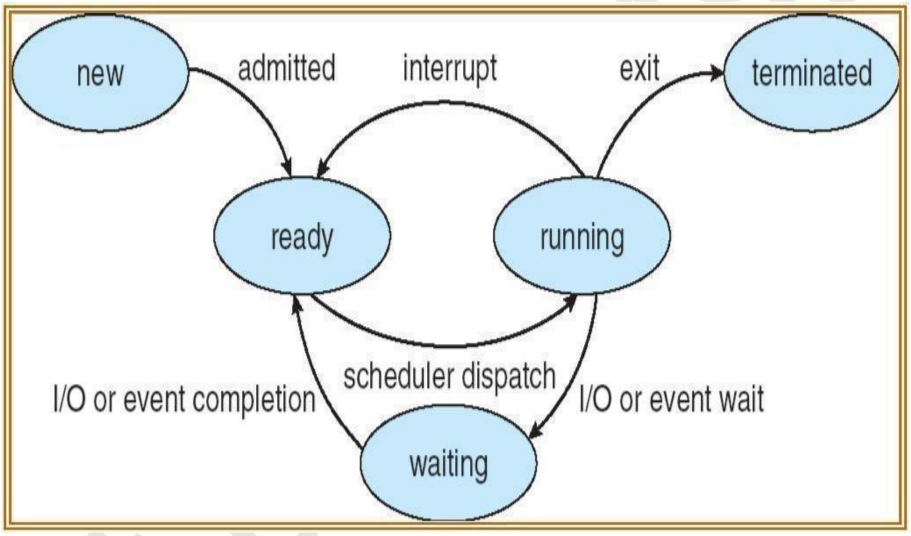
3. INTRODUCTION TO OPERATING SYSTEMS

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.

3.1 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

3.2 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