SYSTEM SOFTWARE & OPERATING SYSTEM LAB MANUAL (15CSL67)

FOR VI SEMESTER

CMR INSTITUTE OF TECHNOLOGY

COMPUTER SCIENCE & ENGINEERING

Table of Contents

		b. Write YACC program to evaluate arithmetic expression involving operators: +, -, *, and /.	35
	Program 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).	37

	Program 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$.	39
	Program 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.	48
	Program 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:	52

Course Details

Course Name **:** System Software and Operating System Lab

Course Code **:** 15CSL67

Course prerequisite **:** Basic Knowledge on Lex, YACC, C programming and UNIX Commands.

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.

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.

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.

10. a) Design, develop and implement a C/C++/Java program to simulate a **numerical calculator**

b) Design, develop and implement a C/C++/Java program to simulate **page replacement technique**

Note: In Examination, for question No 10: Students may be asked to execute any one of the above (10(a)

or 10(b)- Examiner choice)

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

1. LEX

The word "lexical" in the traditional sense means "pertaining to words". In terms of programming languages, words are objects like variable names, numbers, keywords etc. Such words are traditionally called tokens. A lexical analyzer, or lexer for short, will take input as a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space), i.e., lay-out characters (spaces, newlines etc.) and comments.

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. This interaction, summarized schematically in Fig-1.1, is commonly implemented by making the lexical analyzer be a subroutine or a co routine of the parser.

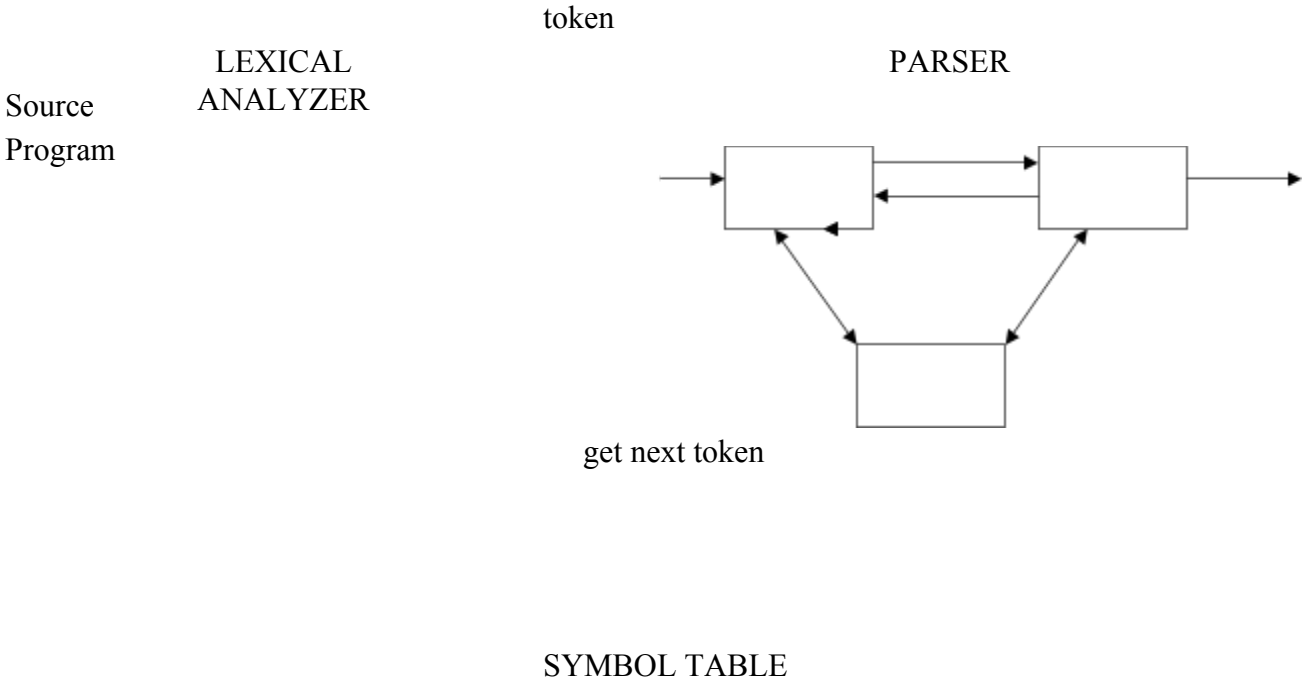


Fig.1.1 Interaction of lexical analyzer with parser

1.1 Structure of LEX source program

Several tools have been built for constructing lexical analyzers from special

purpose

notations based on regular expressions. In this section, we describe a particular tool, called Lex that has been widely used to specify lexical analyzers for a variety of languages. We refer to the tool as Lex compiler, and its input specification as the Lex language.

Lex is generally used in the manner depicted in Fig 1.2.First, a specification of a lexical analyzer is prepared by creating a program lex.l in the lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expression of lex.l, together with a standard routine that uses the table to recognize lexemes. The actions associated with regular expression in lex.l are pieces of C code and are carried over directly to lex.yy.c.Finally lex.yy.c is run through the C compiler to produce an object program a.out,which is the lexical analyzer that transforms an input stream into a sequence of tokens.

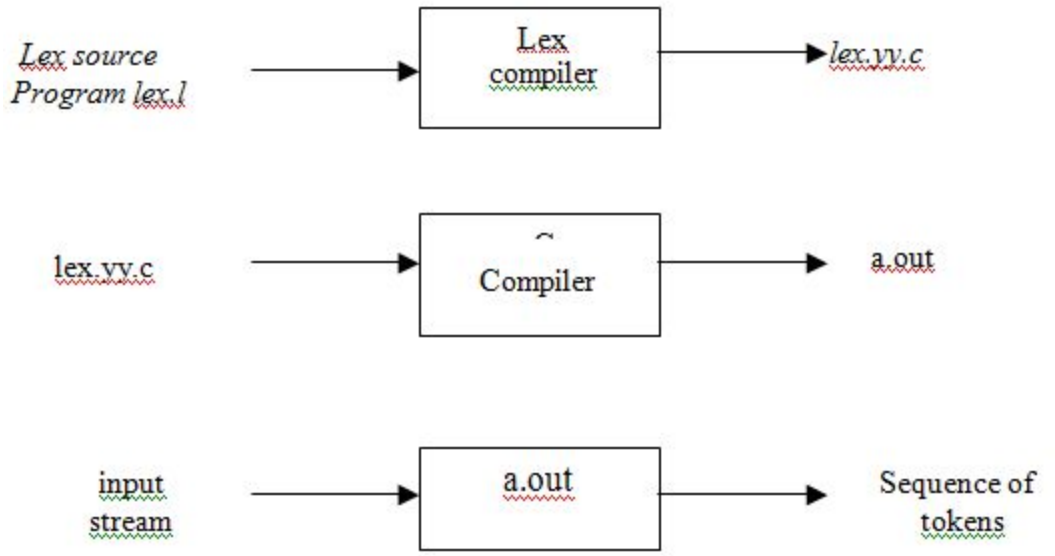


Fig 1.2 Creating a lexical analyzer with Lex

1.2 A LEX program consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

The declarations section includes declarations of variables,constants,and regular definitions.

The translation rules of a lex program are statements of the form

R1 {action1} R2
{action2}
.... …….
Rn {action n} where each Ri is regular expression and each action i, is a program fragment describing what action the lexical analyzer should take when pattern Ri matches lexeme. Typically, action i will return control to the parser. In Lex actions are written in C;in general,however,they can be in any implementation language.

The third section holds whatever auxiliary procedures are needed by the actions.

1.3 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.)

1.4Lex 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.

1.5 The Role Of Parser

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

We know that programs can contain errors at many different levels. For example, errors can be

1.Lexical ,such as misspelling an identifier,keyword,or operator.
2.Syntactic,such as arithmetic expression with unbalanced parentheses
3.Sematic,such as an operator applied to an incompatible operand.
4.Logical,such as infinitely recursive call.

Often much of the error detection and recovery in a compiler is centered around the syntax analysis phase.

1.6 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 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 |

Examples of regular expressions

Regular	**Meaning**

expression	
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. YACC

The UNIX utility yacc (Yet Another Compiler Compiler) parses a stream of token, typically generated by lex, according to a user-specified grammar.

2.1 Structure of a YACC file

A yacc file looks much like a lex file:

definitions
%%
rules
%%
code

Definition: All code between %{ and %} is copied to the beginning of the resulting C file.

Rules: A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces.

Code: This can be very elaborate, but the main ingredient is the call to **yylex**, the lexical analyzer. If the code segment is left out, a default main is used which only calls **yylex.**

Definition section
There are three things that can go in the definitions section:

C code: Any code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

Definitions: The definition section of a lex file was concerned with characters; in **yacc** this is tokens.
Example: **%token NUMBER.**
These token definitions are written to a **.h** file when yacc compiles this file.

Associativity rules These handles associativity and priority of operators.

2.2 Lex Yacc interaction

Conceptually, **lex** parses a file of characters and outputs a stream of tokens; **yacc** accepts a stream of tokens and parses it, performing actions as appropriate. In practice, they are more tightly coupled.

If your lex program is supplying a tokenizer, the yacc program will repeatedly call the yylex routine. The lex rules will probably function by calling return every time they have parsed a token.

If **lex** is to return tokens that **yacc** will process, they have to agree on what tokens there are. This is done as follows:

For Example

1.The yacc file will have token definition **%token NUMBER** in the definitions section.

2. When the yacc file is translated with **yacc –d** , a header file **y.tab.h** is created that has definitions like **#define NUMBER 258.**

3. The **lex** file can then call return **NUMBER**, and the **yacc** program can match on this token.

2.3 Rules section

The rules section contains the grammar of the language you want to parse. This looks like

```
statement : INTEGER '=' expression
| expression
;
expression : NUMBER '+' NUMBER
|      NUMBER '-' NUMBER
;
```

This is the general form of context-free grammars, with a set of actions associated with each matching right-hand side. It is a good convention to keep non-terminals (names that can be expanded further) in lower case and terminals (the symbols that are finally matched) in upper case.

The terminal symbols get matched with return codes from the lex tokenizer. They are typically defines coming from **%token** definitions in the yacc program or character values.

2.4 Compiling and running a simple parser

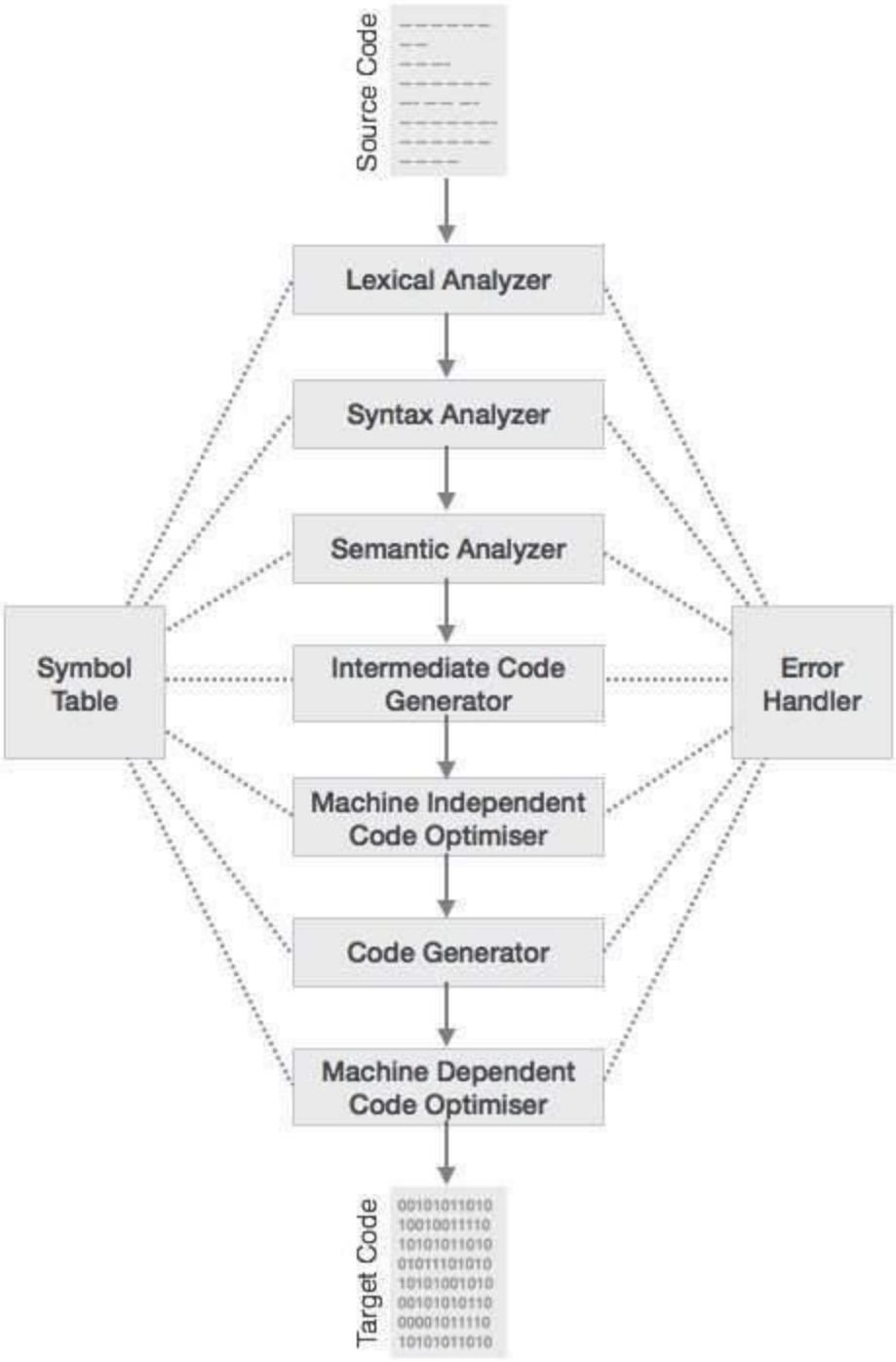
On a UNIX system ,yacc takes your grammar and creates **y.tab.c** ,the C Language parser, and **y.tab.h,** the include file with the token number definitions.

Lex creates **lex.yy.c**,the C language lexer .You need only compile them together with the yacc and lex libraries. The Libraries contain usable default versions of all of the supporting routines ,including **main( )** that calls the parser **yyparse( )** and exits.

```
[root@localhost ~]# lex filename.l                  #makes lex.yy.c [root@localhost ~]#
 yacc -d filename.y                                 #makes y.tab.c and y.tab.h
 [root@localhost ~]# cc lex.yy.c y.tab.c –ll        #compile and link C files
 [root@localhost ~]# ./a.out
```

3. COMPILER DESIGN

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

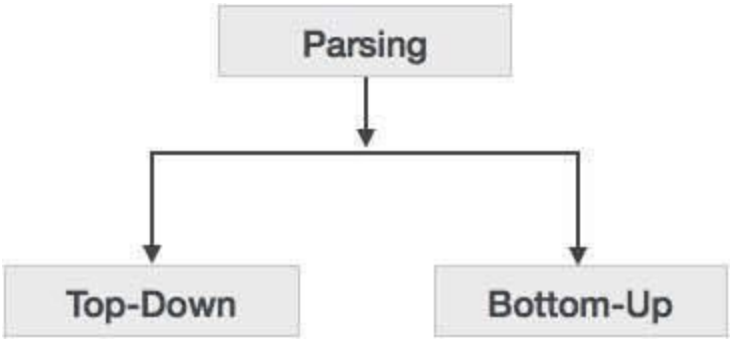
In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

3.1 Types of Parsing:

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.



3.1.1 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.

3.1.2 Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Example:

Input string : a + b * c

Production rules:

S → E

E → E + T

E → E * T

E → T

```
T → id
```

Let us start bottom-up parsing

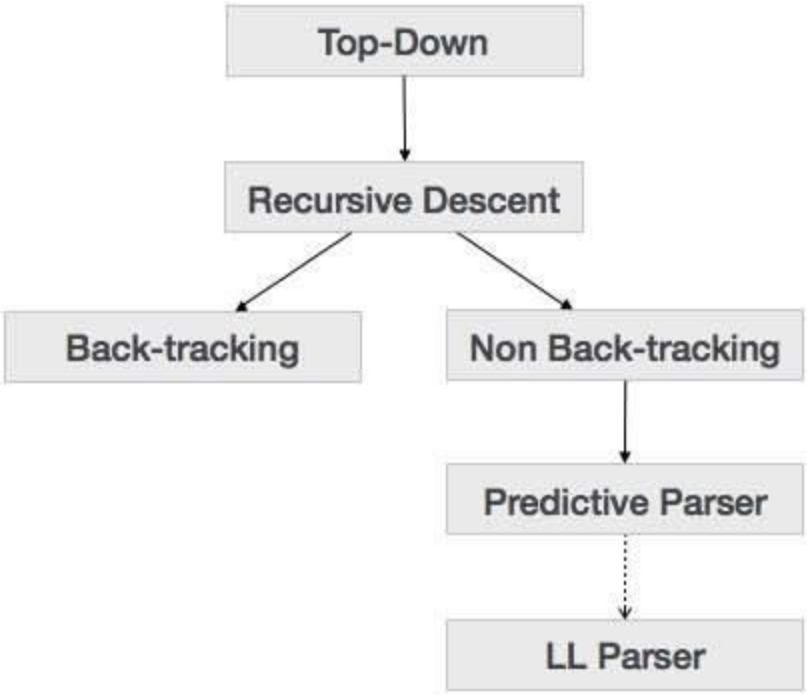
```
a + b * c
```

Read the input and check if any production matches with the input:

```
a + b * c

T + b * c

E + b * c

E + T * c

E * c

E * T

E

S
```

3.2 Top down Parser:

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



3.2.1 Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

3.2.2 Back-tracking

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd

X → oa | ea

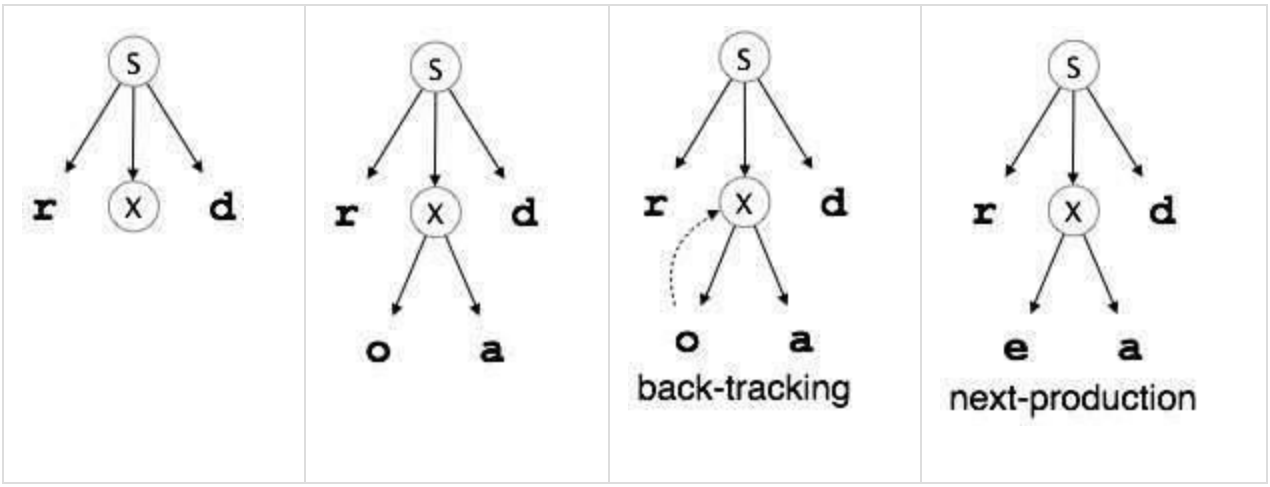
Z → ai

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the

next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

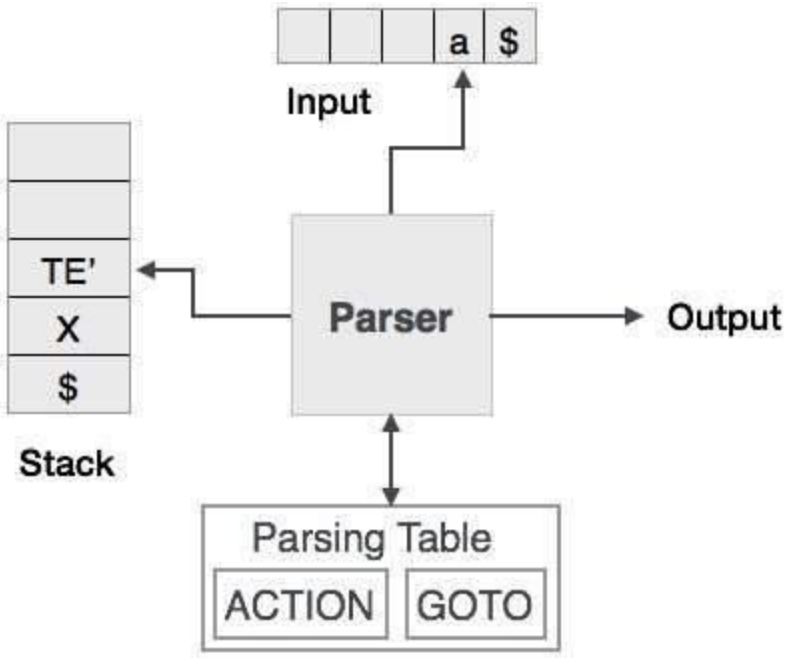
Now the parser matches all the input letters in an ordered manner. The string is accepted.



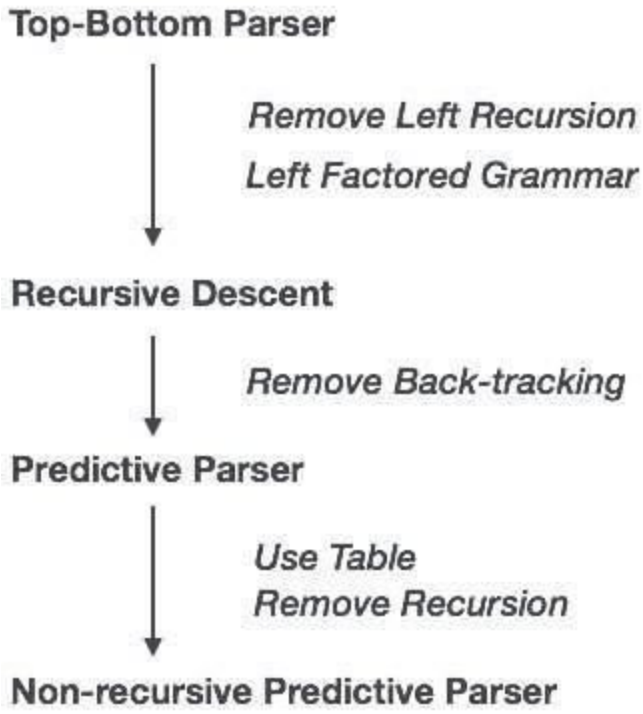
3.2.3 Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol $to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

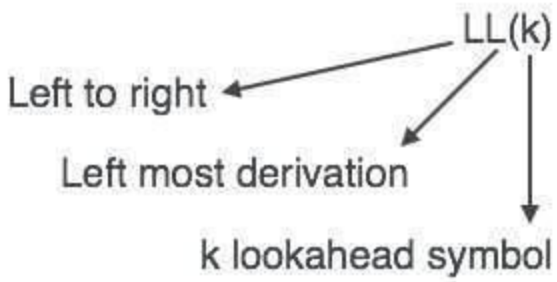


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

3.2.4 LL Parser

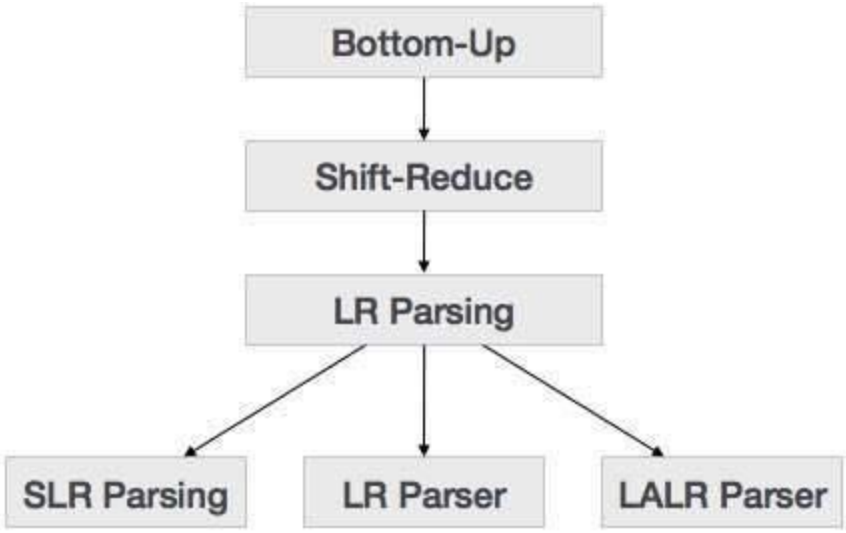
An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



3.3 Bottom Up parser:

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



3.3.1 Shift-Reduce Parsing

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

- **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

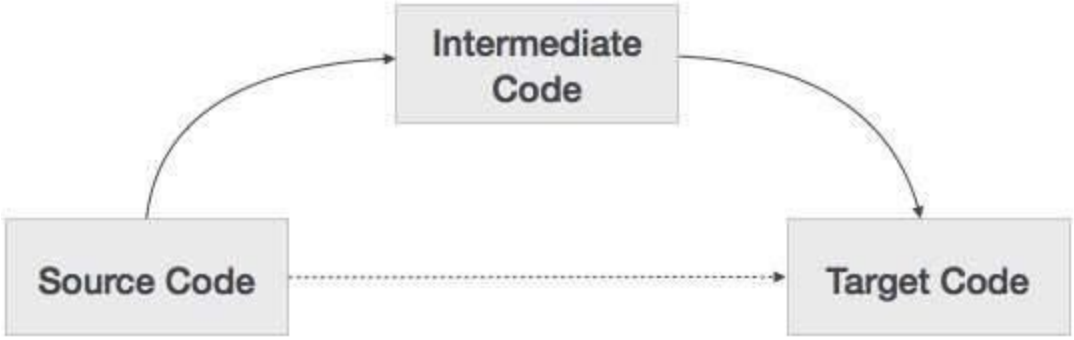
- **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

3.3.2 LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

3.4 Intermediate Code Generation

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

3.4.1 Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;

r2 = b + r1;

a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

4. OPERATING SYSTEMS

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

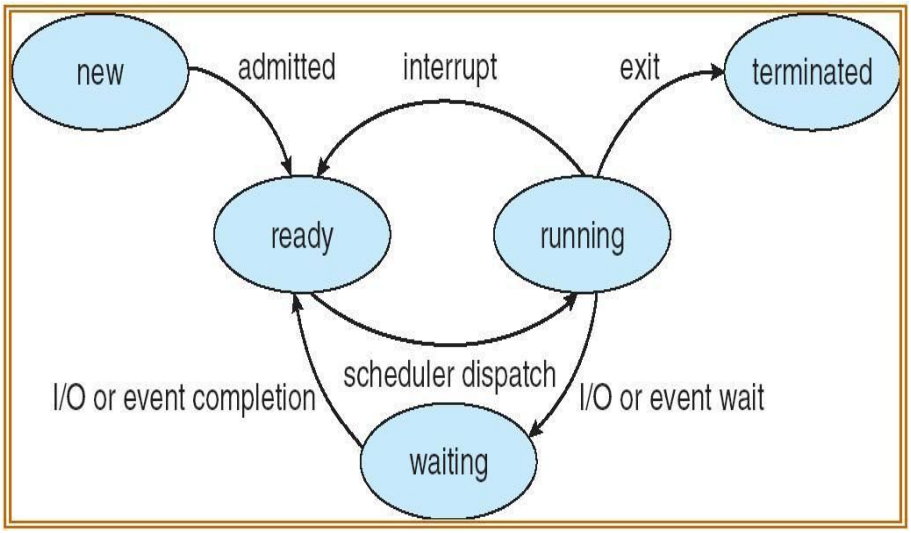
New : The process is being created

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

4.3 Page Replacement

In a operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

Page Fault – A page fault is a type of interrupt, raised by the hardware when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

4.3.1 Page Replacement Algorithms:

First In First Out (FIFO)

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example-, Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4.

Time req. / Page frames		1	2	3	4	5	6	7	8	9	10	11	12	13	14
		a	b	c	d	c	a	d	b	e	b	a	b	c	d
0	a	**a**	a	a	a	a	a	a	a	**e**	e	e	e	e	**d**
1	b		**b**	b	b	b	b	b	b	b	b	**a**	a	a	a
2	c			**c**	c	c	c	c	c	c	c	c	**b**	b	b
3	d				**d**	d	d	d	d	d	d	d	d	**c**	c
FAULTS		**x**	**x**	**x**	**x**					**x**		**x**	**x**	**x**	**x**

There are 9 page faults using FIFO algorithm.

Belady's anomaly – Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Least Recently Used

In this algorithm page will be replaced which is least recently used.

Example, Let's have a reference string: a, b, c, d, c, a, d, b, e, b, a, b, c, d and the size of the frame be 4.

Time req. / Page frames		1	2	3	4	5	6	7	8	9	10	11	12	13	14
		a	b	c	d	c	a	d	b	e	b	a	b	c	d
0	a	**a**	a	a	a	a	a	a	a	a	e	e	e	e	e
1	b		**b**	b	b	b	b	b	b	b	b	a	a	a	a
2	c			**c**	c	c	c	c	c	**e**	c	c	b	b	**d**
3	d				**d**	d	d	d	d	d	d	d	d	**c**	c
FAULTS		**x**	**x**	**x**	**x**					**x**				**x**	**x**

There are 7 page faults using LRU algorithm

5. LESSON PLAN

Session No	Programs	Variations	Extra Programs
1	1. Introduction to LEX and YACC programming. 2. Regular Expression basics. 3. Special Fuctions.	1. Default action of LEX program 2. Count number of words starting with vowels. 3. Count the number of a's followed by b's. 4. Count the number of a's not followed by the string bc. 5. Count the number of words having length b/w 5-7.	-

2	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.	1. LEX program using states 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.	YACC Program to recognize a valid arithmetic expression that uses operators +, -, * and /.
3	1b) Write YACC program to evaluate ***arithmetic expression*** involving operators: +, -, *, and /	1. Lex program to recognize valid arithmetic expression by taking an input from file and writing the output on to another output file. Identifiers in the expression could be only integers and operators could be + and *. Count the identifiers & operators present and print them separately.	LEX Program to count the number of characters, words, spaces and lines in a given input file.
4	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)	1. YACC program to recognize strings "aaab","ab","abbb" and "a" using grammer a^nb^n where n>=0. 2. YACC program to recognize strings "aaab","ab","abbb" and "a" using grammer a^nb^n where n>=1. 3. YACC program to recognize strings "aaab","ab","abbb" and "a" using grammer ab^n where n>=0.	LEX Program to recognize whether a given sentence is simple or compound.
5	6. a) Write a LEX program to eliminate ***comment lines*** in a ***C*** program and copy the	1. LEX program using states to eliminate comment lines in a C program and display the	LEX Program to recognize and count the number of identifiers in a given input file.

	resulting program into a separate file.	resulting program on the standard output. 2. LEX program using states to eliminate comment lines in a C program and copy the resulting program on the output file.	
6	b) Write YACC program to recognize valid ***identifier, operators and keywords*** in the given text (*C program*) file.	1. YACC program to recognize a valid variable which starts with a letter followed by any number of letters or digits	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
7	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.	1. Demonstrate performance of SRTF and RR algorithms with respect to completion time, turnaround time and waiting time.	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 to finish (by executing the wait ( )) and prints its own process-id and the id of its child process and then exits.
8	8. Design, develop and implement a C/C++/Java program to implement Banker's algorithm. Assume suitable input required to demonstrate the results.	1. Implement banker's algorithm for deadlock detection.	-
9	***3.*** Design, develop and implement YACC/C program to construct ***Predictive / LL(1)*** ***Parsing Table*** for the grammar rules: $A \rightarrow aBa$, $B \rightarrow bB \mid \varepsilon$. Use this table to parse the sentence: ***abba$***	1. YACC/C program to construct Predictive / LL(1) Parsing Table for the grammar: S->aB|bA, A->a|aS|bAA, B->b|bs|aBB. Use this table to parse the sentence aabb. 2. YACC/C program to construct Predictive / LL(1) Parsing Table for the grammar:	

		S->aB|bA, A->a|aS|bAA, B->b|bs|aBB. Use this table to parse the sentence aaabbb.	-
10	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***.	1. Design, develop and implement YACC/C program to demonstrate Shift Reduce Parsing technique for the grammar: S-> E$,E ->T, E -> E + T, T -> int, T -> (E) and parse the sentence int + (int + int + int). 2. Design, develop and implement YACC/C program to demonstrate Shift Reduce Parsing technique for the grammar: S-> E$,E ->T, E -> E + T, T -> int, T -> (E) and parse the sentence int + int + (int + int)	-
11	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***	1. C/Java program to generate the machine code using Triples for the statement a = b + c * d; whose intermediate code in three-address form: r1 = c * d; r2 = b + r1; a = r2. 2. C/Java program to generate the machine code using Triples for the statement b = c + d * a; whose intermediate code in three-address form: r1 = d * a; r2 = c + r1; b = r2.	-
12	9. Design, develop and implement a C/C++/Java program to implement page	1. Demonstrate the performance of LRU and FIFO with respect to page faults and also demonstrate the FIFO's belady's anomaly.	Using Open MP, 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

	replacement algorithms LRU and FIFO. Assume suitable input required to demonstrate the results.		print them, ensure proper synchronization.
13	***10.*** a) Design, develop and implement a C/C++/Java program to simulate a ***numerical*** ***calculator***	-	-
14	b) Design, develop and implement a C/C++/Java program to simulate ***page*** ***replacement technique***	-	-

6. Lab Experiments

1a) 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.

```
%{
int count=0,ids=0,bracket=0;
%}
%%
[+] {count++;}    /*For recognizing the operators*/
[-] {count++;}
[*] {count++;}
[/] {count++;}
[a-zA-Z0-9]+ {ids++;}          /*For recognizing the identifiers*/
 [(] {bracket++;}              /*For recognizing the brackets*/
```

```
[)] {bracket--;}
%%
int main()
{
printf("Enter the Arithmetic expression:\n");
yylex();
printf("Number of Operators=%d\n",count);
printf("Number of Identifiers=%d\n",ids);
if(count>=ids||bracket!=0||ids==1)
printf("Invalid expression\n");
 else
printf("Valid expression\n");
}
```

Procedures for Execution

Save the lex code Program with .l extension

```
[root@localhost ~]# lex filename.l
[root@localhost ~]# cc lex.yy.c –ll
[root@localhost ~]# ./a.out
```

OUTPUT

Enter the Arithmetic expression:	Enter the Arithmetic expression:
2+3*4	2+3*4-
(Press Ctrl d)	**(Press ctrl d)**
Number of Operators=2	Number of Operators=3
Number of Identifiers=3	Number of Identifiers=3
Valid expression	Invalid expression

1b) Write YACC program to evaluate arithmetic expression involving operators: +, -, *, and /.

Lex Part

```
%%
[0-9]+ {yylval=atoi(yytext); return(NUM);}/*For recognizing the number and convert ASCII to integer*/
[ \t]; /*Ignore the whitespace*/
.      {return yytext[0];}
\n     {return 0;}
%%
%token NUM
%left '+"-'
%left '*"/'
```

Yacc part

```
 %%
stmt : expr { printf("Result:%d\n",$1);return 0; }        /*Production For recognizing the expression*/
;
expr :expr'+'expr         {$$=$1+$3;}
| expr'-'expr      {$$=$1-$3;}
| expr'*'expr      {$$=$1*$3;}
| expr'/'expr      {$$=$1/$3;}
| '('expr')'         {$$=-$2;}
| NUM {$$=$1;}
;
%%

int main()
{
printf("Enter the expression\n");
yyparse();
}




int yyerror()
{
printf("Invalid input\n");
exit(0);
}
```

Procedures for Execution

Save the lex code Program with .l extension

Save the yacc code Program with .y extension

```
[root@localhost ~]# lex filename.l
[root@localhost ~]# yacc -d filename.y
[root@localhost ~]# cc lex.yy.c y.tab.c -ll
```

[root@localhost ~]# ./a.out

OUTPUT

Enter the expression	Enter the expression
2+3	2*3+4
Result:5	Result:10

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 " b (note: input n value).

Lex part

```
%{
#include"y.tab.h"
%}
%%
a      {return A;}          /*For recognizing the character a*/
b      {return B;}          /*For recognizing the character b*/
```

```
.       {return yytext[0];}
\n {return yytext[0];}
%%
```

Yacc Part

```
 %token A B                  /*tokens declaration*/
%%
str: s'\n'          {return 0;}          /*Productions for recognizing the grammar*/
s : x B ;
x : x A
| ;
%%
int main()
{
printf("Type the string\n");
yyparse();
printf("Valid string");
}

int yyerror()
{
printf("Invalid string");
exit(0);
}
```

Procedures for Execution

Save the lex part code with .l extension

Save the yacc part code with .y extension

```
[root@localhost ~]# lex filename.l
[root@localhost ~]# yacc -d filename.y
[root@localhost ~]# cc lex.yy.c y.tab.c -ll
[root@localhost ~]# ./a.out
```

OUTPUT

Type the string	Type the string
aaaaab	aabb
Valid string	Invalid string

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$.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
```

```
{
char fin[10][20],st[10][20],ft[20][20],fol[20][20]; int
 a=0,e,i,t,b,c,n,k,l=0,j,s,m,p; printf("enter the no. of
 coordinates\n"); scanf("%d",&n);
printf("enter the productions in a grammar\n");
for(i=0;i<n;i++)
scanf("%s",st[i]);
for(i=0;i<n;i++)
fol[i][0]='\0';
for(s=0;s<n;s++)
{
for(i=0;i<n;i++)
{
j=3;
l=0;
a=0;

l1:if(!((st[i][j]>64)&&(st[i][j]<91)))
{
for(m=0;m<l;m++)
{

if(ft[i][m]==st[i][j])
goto s1;
}
ft[i][l]=st[i][j];
l=l+1;
```

```
s1:j=j+1;
}
else
{
if(s>0)
{
while(st[i][j]!=st[a][0])
{
a++;

}
b=0;
while(ft[a][b]!='\0')
{
for(m=0;m<l;m++)
{
if(ft[i][m]==ft[a][b])
goto s2;
}
ft[i][l]=ft[a][b];
l=l+1;
s2:b=b+1;
}
}
}
while(st[i][j]!='\0')
{
```

```
if(st[i][j]=='|')

{
j=j+1;
goto l1;
}
j=j+1;
}
ft[i][l]='\0';
}
}
printf("first pos\n");
for(i=0;i<n;i++)
printf("FIRS[%c]=%s\n",st[i][0],ft[i]);
fol[0][0]='$';
for(i=0;i<n;i++)
{
k=0;
j=3;
if(i==0)

l=1;
else
l=0;
k1:while((st[i][0]!=st[k][j])&&(k<n))
{
```

```
if(st[k][j]=='\0')
{
k++;
j=2;
}
j++;

}
j=j+1;
if(st[i][0]==st[k][j-1])
{
if((st[k][j]!='|')&&(st[k][j]!='\0'))

{
a=0;
if(!((st[k][j]>64)&&(st[k][j]<91)))
{
for(m=0;m<l;m++)
{
if(fol[i][m]==st[k][j])
goto q3;
}
q3:
fol[i][l]=st[k][j];
l++;
}
else
```

```
{
while(st[k][j]!=st[a][0])
{
a++;
}
p=0;
while(ft[a][p]!='\0')
{
if(ft[a][p]!='@')
{
for(m=0;m<l;m++)
{
if(fol[i][m]==ft[a][p])
goto q2;
}
fol[i][l]=ft[a][p];
l=l+1;
}
else
e=1;
q2:p++;
}
if(e==1)
{
e=0;
goto a1;
}
```

```
}
}
else
{
a1:c=0;
a=0;
while(st[k][0]!=st[a][0])
{
a++;
}
while((fol[a][c]!='\0')&&(st[a][0]!=st[i][0]))
{
for(m=0;m<l;m++)
{
if(fol[i][m]==fol[a][c])
goto q1;
}
fol[i][l]=fol[a][c];

l++;
q1:c++;
}
}
goto k1;

}
fol[i][l]='\0';
```

```
}
printf("follow pos\n");


for(i=0;i<n;i++)
printf("FOLLOW[%c]=%s\n",st[i][0],fol[i]);
printf("\n");
s=0;
for(i=0;i<n;i++)
{
j=3;
while(st[i][j]!='\0')
{
if((st[i][j-1]=='|')||(j==3))
{
for(p=0;p<=2;p++)
{
fin[s][p]=st[i][p];
}
t=j;
for(p=3;((st[i][j]!='|')&&(st[i][j]!='\0'));p++)
{
fin[s][p]=st[i][j];
j++;
}
fin[s][p]='\0';
if(st[i][t]=='@')
{
```

```
b=0;
a=0;
while(st[a][0]!=st[i][0])
{
a++;
}
while(fol[a][b]!='\0')

{
printf("M[%c,%c]=%s\n",st[i][0],fol[a][b],fin[s]);
b++;
}
}
else if(!((st[i][t]>64)&&(st[i][t]<91)))
printf("M[%c,%c]=%s\n",st[i][0],st[i][t],fin[s]);
else
{
b=0;
a=0;
while(st[a][0]!=st[i][3])
{
a++;
}
while(ft[a][b]!='\0')
{

printf("M[%c,%c]=%s\n",st[i][0],ft[a][b],fin[s]);
```

```
b++;
}
}
s++;
}
if(st[i][j]=='|')
j++;
}}
getch();}
```

Procedures for Execution

Save the code with .c extension

```
[root@localhost ~]# cc filename.cpp
[root@localhost ~]# ./a.out
```

OUTPUT

```
Enter the no. of coordinates
2
Enter the productions in a grammer
A->aBa
B->bB|@
First pos
FIRS[A]=a
FIRS[B]=b@
Follow pos
FOLLOW[A]=$
FOLLOW[B]=a

M[A,a]=a->aBa
M[B,b]=B->bB
```

M[B,a]=B->@

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.

```
#include<stdio.h>
#include<conio.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();
void main()
{
puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
puts("enter input string ");

gets(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%ssymbols",stk,a,act);
check();
}
}
```

```
getch();
}
void check()
{

strcpy(ac,"REDUCE TO E");
for(z=0; z<c; z++)
if(stk[z]=='i' && stk[z+1]=='d')
{
stk[z]='E';

stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
j++;
}
for(z=0; z<c; z++)

if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
{
stk[z]='E';
stk[z+1]='\0';

stk[z+2]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}
for(z=0; z<c; z++)
```

```
if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E')
{
stk[z]='E';
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]=='(' && stk[z+1]=='E' && stk[z+2]==')')
{
stk[z]='E';
stk[z+1]='\0';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}}
```

Procedures for Execution

Save the code with .c extension

```
[root@localhost ~]# cc filename.cpp
[root@localhost ~]# ./a.out
```

OUTPUT

GRAMMAR is E->E+E

E->E*E

E->(E)

E->id

Enter input string

id+id*id

stack	**input**	**action**
$id	+id*id$	SHIFT->id
$E	+id*id$	REDUCE TO E
$E+	id*id$	SHIFT->symbols
$E+id	*id$	SHIFT->id
$E+E	*id$	REDUCE TO E
$E	*id$	REDUCE TO E
$E*	id$	SHIFT->symbols
$E*id	$	SHIFT->id
$E*E	$	REDUCE TO E
$E	$	REDUCE TO E

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 = 1 + T2

A = T3

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
char op[2],arg1[5],arg2[5],result[5];
void 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,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nADD R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"*")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nMUL R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
```

```
}
if(strcmp(op,"-")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nSUB R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"/")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nDIV R0,%s",arg2);
fprintf(fp2,"\nMOV %s,R0",result);
}
if(strcmp(op,"=")==0)
{
fprintf(fp2,"\nMOV R0,%s",arg1);
fprintf(fp2,"\nMOV %s,R0",result);
}
}
fclose(fp1);
fclose(fp2);
getch();
}
```

Procedures for Execution

Save the code with .c extension

[root@localhost ~]# cc filename.c

[root@localhost ~]# ./a.out

Input.txt

T1 -B = ?

T2 C + D

T3 T1 * T2

A T3 = ?

OUTPUT

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

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

```
%{
int comment=0;
%}
%%
"/*"([^*]|\*[^/])*"*/"  {comment++;}          /*Multiple line comment comment*/
"//".*   { comment++;}          /*Single line comment */
%%
int main()
{
char infile[256],outfile[256];
printf("Enter the input filename:\n");
 scanf("%s",infile);
printf("Enter the output filename:\n");
scanf("%s",outfile);
yyin=fopen(infile,"r");
yyout=fopen(outfile,"w");
yylex();
printf("Number of comment lines in the given file: %d\n",comment);
}
```

Procedures for Excecution

Save the lex code Program with .l extension.

Create a file using the editor command and write the content with comment line

```
[root@localhost ~]# vi sum.c
/*To find the sum of two numbers*/
#include<stdio.h>
void main()
{
int a=1,b=2;
printf("%d",a+b);       /*To display the sum*/
```

}

[root@localhost ~]# lex filename.l

[root@localhost ~]# cc lex.yy.c -ll

[root@localhost ~]# ./a.out

OUTPUT

Enter the input filename:

sum.c

Enter the output filename:

add.c

Number of comment lines in the given file: 2

Atlast see the content of add file using the command cat add.c

[root@localhost ~]# cat add.c

#include<stdio.h>

void main()

{

int a=1,b=2;

printf("%d",a+b);

}

6b. Write YACC program to recognize valid identifier, operators and keywords in the given text (C program) file.

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("numbers 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 File

```
%{

#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 can't open sam_input.c!");


return -1;
}
yyin = myfile;
do {
yyparse();


} while (!feof(yyin));
printf("numbers = %d\nKeywords = %d\nIdentifiers = %d\noperators = %d\n",
dig, key,id, op);
}


void yyerror() {
printf("EEK, parse error! Message: ");
exit(-1);


}
```

Procedures for Execution

Save the lex part code with .l extension

Save the yacc part code with .y extension

```
[root@localhost ~]# lex filename.l
[root@localhost ~]# yacc -d filename.y
[root@localhost ~]# cc lex.yy.c y.tab.c -ll
[root@localhost ~]# ./a.out
```

sam_input.c

```
void main()
```

```
{
Float a123;
Char a;
Char b123;
If(sum==10)
Printf(“pass”);
Else
Printf(“fail”);
}
```

OUTPUT

```
Keyword is void
Identifier is main
Keyword is float
Identifier is a123
Keyword is char
Identifier is b123
Keyword is char
Identifier is c
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.

```
#include<iostream>
#define MAX 20
#define INF 999
using namespace std;
struct time
{
float at,bt,tat;     //structure variable
int p;
};

void sjf(struct time a[],int n,int sum)
{
        float ts=0,diff; //ts=time spent
        int i,j=1,k=1;
        while(ts<sum)
```

```
        {
          if(j<=n-1)
          {
                    j++;
                    diff=a[j].at-a[j-1].at; //difference of arrival time
                    a[k].bt-=diff;
                    ts+=diff; //adding the difference to time spent

                    if(a[k].bt==0)
                              a[k].tat=ts-a[k].at; //if Burst time becomes zero calculate T.A.T
          }
          else
          {
                    j=n;  //if its last process
                    ts+=a[k].bt;
                    a[k].bt=0;
                    a[k].tat=ts-a[k].at;
          }
          int small=INF;
          for(i=1;i<=j;i++)
                    if(a[i].bt<small && a[i].bt!=0)
                    {
                              small=a[i].bt;  //small holds the smallest burst time
                              k=i;                         //k holds the index of shortest job
                    }
        }
}

void rr(struct time a[],int n,int sum,int q)
{
        float ts=0;
        int i,count=0;
        while(count<=n && ts<sum)
```

```
    {
      for(i=1;i<=n;i++)
      {
            if(a[i].bt!=0 && a[i].at<=ts)
            {
                  if(a[i].bt<q)  //if burst time is less than q
                  {
                        ts+=a[i].bt;
                        a[i].bt=0;
                  }
                  else
                  {
                        a[i].bt-=q;  //if burst time is greater than q
                        ts+=q;
                  }
                  if(a[i].bt==0)
                  {
                        a[i].tat=ts-a[i].at;   //calculate TAT once burst time becomes zero
                        count++;
                  }
            }
            if(count==n-1 && a[i].at>ts)
                  {
                        ts++;
                        sum++;
                  }
      }

    }
}
int main()
{
    int i,j,ch,k=1;
```

```
        int n,sum,q;
        struct time a[MAX],temp;

while(k)  //while k remains 1 more p...
{

        cout<<"\n enter the number of processes\n";
        cin>>n;
        sum=0; //
        for(i=1;i<=n;i++)
        {
          cout<<"\n enter arrival time for process"<<i<<":";  //if arrival time is s not given in round robin, give all arrival time as zeros.
          cin>>a[i].at;
          cout<<"\n enter burst time for process"<<i<<":";
          cin>>a[i].bt;
          a[i].p=i;
          sum=sum+a[i].bt;
        }

        for(i=1;i<=n;i++)      //to sort according to the arrival time;
          for(j=1;j<=n-i;j++)
                  if(a[j].at>a[j+1].at)
                  {
                          temp=a[j];
                          a[j]=a[j+1];
                          a[j+1]=temp;
                  }
        cout<<"Process\tArrival time\tBurst time\n";
        for(i=1;i<=n;i++)
        cout<<a[i].p<<"\t"<<a[i].at<<"\t\t"<<a[i].bt<<"\n";
        cout<<"1.SJF 2.RR";
        cin>>ch;
```

```
        switch(ch)
        {
        case 1:sjf(a,n,sum);
          cout<<"\n \n SHORTEST REMAINING JOB \n \n"; break;
        case 2:cout<<"enter q";
              cin>>q;
             rr(a,n,sum,q);
             cout<<"\n \n ROUND ROBIN \n \n";
        }
        cout<<"\n \n \n \n"<<"P.NO"<<"\t"<<"T.A.T"<<endl;
        for(i=1;i<=n;i++)
        cout<<a[i].p<<"\t"<<a[i].tat<<"\n";
        float avg=0;
        for(i=1;i<=n;i++)
        {
          avg+=a[i].tat;
        }
        cout<<"\n average T.A.T is"<<avg/n<<endl;
cout<<endl<<"If you want to continue press 1 or else 0"<<endl;
cin>>k;
}
return 0;
}
```

Procedures for Execution

Save the code with .cpp extension

```
[root@localhost ~]# g++ filename.cpp
[root@localhost ~]# ./a.out
```

OUTPUT

Process Arrival time Burst time

p1	1	5
p2	2	7
p3	0	5
p4	1	2

SJF - preemptive

P3	P4	P3	P1	p2
0 1	3	7	12	19

TAT:

p1=11

p2=17

p3=7

p4=2

Avg TAT=9.25

RR

Time Quantum=2

P3	P1	P4	P2	P3	P1	P2	P3	P1	p2
0	2	4	6	8	10	12	14	15	16 19

TAT:

p1=15

p2=17

p3=15

p4=5

Avg TAT=13

enter the number of processes

4

enter arrival time for process1:1

enter burst time for process1:5

enter arrival time for process2:2

enter burst time for process2:7

enter arrival time for process3:0

enter burst time for process3:5

enter arrival time for process4:1

enter burst time for process4:2

1.SJF 2.RR1

SHORTEST REMAINING JOB

P.NO T.A.T

3 7

1 11

4 2

2 17

Average T.A.T is9.25

If you want to continue press 1 or else 0

1

enter the number of processes

4

enter arrival time for process1:1

enter burst time for process1:5

enter arrival time for process2:2

enter burst time for process2:7

enter arrival time for process3:0

enter burst time for process3:5

enter arrival time for process4:1

enter burst time for process4:2

1.SJF 2.RR2

enter q2

ROUND ROBIN

P.NO T.A.T

3 15

1 15

4 5

2 17

Average T.A.T is13

If you want to continue press 1 or else 0

0

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

```
#include<iostream>
#define MAX 20
using namespace std;
void safe(int n,int r,int a[MAX][MAX],int need[MAX][MAX],int avail[MAX])
{
      int count=0,looping=0,finish[20]={0},work[20]={0},i=1,j,flag=0,p[20];
      for(i=0;i<r;i++)
        work[i]=avail[i]; //assigning available to work
      i=0;
```

```
while(count<n && looping<5*n) //looping is used to loop for some defenite number of times
{
  flag=0;
  looping++;
  if(finish[i]==0)
  {
        for(j=0;j<r;j++)
              if(need[i][j]>work[j])  //if need is greater than work
              {
                    flag=1;
                    break;
              }
        if(!flag)                   //if need is less than work
        {
              for(j=0;j<r;j++)
              work[j]=work[j]+a[i][j];
              finish[i]=1;
              count++;
              p[count]=i;     //p[] will store the process number which has finished execution

        }
  }
  i=(i+1)%n;                  //increment i for next process

}
flag=0;
for(i=0;i<n;i++)
  if(finish[i]==0)       //if some of the processes have not completed
  {
        flag=1;
        break;
  }
if(flag==0)
```

```
    {
      cout<<"\n system is in safe state and the sequece is\n";
      for(i=1;i<=n;i++)
            cout<<"P"<<p[i]<<"\t";      //print safe sequence
    }
    else
      cout<<"\n system is not in safe state\n";
}

void newreq(int n,int r,int a[MAX][MAX],int need[MAX][MAX],int avail[MAX])
{
    int p,nr[20],flag=0,j,i;
    cout<<"\n enter requesting process"<<endl;
    cin>>p;
    cout<<"\n enter request in order of resources \n";
    for(i=0;i<r;i++)
      cin>>nr[i];                          //input request array
    for(j=0;j<r;j++)
      if(nr[j]>need[p][j] && nr[j]>avail[j])   //if reequest is greater than need and if request is greater
than available
            flag=1;
    if(!flag)
    {
      for(i=0;i<r;i++)
      {
            avail[i]-=nr[i];           //update available, need and allocation matrix of process p
            a[p][i]+=nr[i];
            need[p][i]-=nr[i];
      }
      safe(n,r,a,need,avail);            //check if allocation leads to safe state
    }
    else
    {
```

```
        cout<<"\n the process exceeds its maximum"<<endl;
    }
}
int main()
{
    int i,j,ch;
    int n,r,a[MAX][MAX],m[MAX][MAX],need[MAX][MAX],avail[MAX];
    cout<<"\n enter the number processes"<<endl;
    cin>>n;
    cout<<"\n enter the number of resources"<<endl;
    cin>>r;
    cout<<"\n enter allocation matrix"<<endl;
    for(i=0;i<n;i++)
      for(j=0;j<r;j++)
            cin>>a[i][j];
    cout<<"\n enter max matrix"<<endl;
    for(i=0;i<n;i++)
      for(j=0;j<r;j++)
            cin>>m[i][j];
    for(i=0;i<n;i++)
      for(j=0;j<r;j++)
            need[i][j]=m[i][j]-a[i][j]; //calculate need matrix
    cout<<"\n need matrix\n";
    for(i=0;i<n;i++)
    {
      for(j=0;j<r;j++)
      {
            cout<<need[i][j]<<" ";
      }
      cout<<"\n";
    }

    cout<<"\n enter available resources\n";
```

```
        for(i=0;i<r;i++)
          cin>>avail[i];
        safe(n,r,a,need,avail);              //check if the current state is safe or not
        while(true)
        {
        cout<<"\n 1.new request.\n 2.exit\n enter your choice:";
        cin>>ch;
        switch(ch)
        {
        case 1:newreq(n,r,a,need,avail);        //resource request implementation
          continue;
        case 2:    return 0;

        default:cout<<"re-enter";
          continue;
        }
        }
return 0;

}
```

Procedures for Execution

Save the code with .cpp extension

```
[root@localhost ~]# g++ filename.cpp
[root@localhost ~]# ./a.out
```

OUTPUT

```
enter the number processes
4
 enter the number of resources
3
 enter allocation matrix
1 2 3
4 5 6
7 1 2
3 4 5

 enter max matrix
9 8 7
6 5 7
8 9 7
6 5 4

 need matrix
8 6 4
2 0 1
1 8 5
3 1 -1


enter available resources
5 9 8

 system is in safe state and the sequece is
1 2 3 0
 1.new request.
 2.exit
```

enter your choice:1

enter requesting process

1

enter request in order of resources

6 9 10

the process exceeds its maximum

1.new request.

2.exit

enter your choice:1

enter requesting process

1

enter request in order of resources

1 2 3

system is in safe state and the sequece is

1 2 3 0

1.new request.

2.exit

enter your choice:2

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.

```
#include<stdio.h>
#include<stdlib.h>
```

```
void FIFO(char [ ],char [ ],int,int);

void lru(char [ ],char [ ],int,int);


int main()

{

int ch,YN=1,i,l,f;

char F[10],s[25];

printf("\n\n\tEnter the no of empty frames: ");
   scanf("%d",&f);


printf("\n\n\tEnter the length of the string: ");

scanf("%d",&l);

printf("\n\n\tEnter the string: ");

scanf("%s",s);

for(i=0;i<f;i++)


F[i]=-1;

do

{

printf("\n\n\t***********          MENU          ***********");
     printf("\n\n\t1:FIFO\n\n\t2:LRU                \n\n\t4:EXIT");
     printf("\n\n\tEnter 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\tDo u want to continue IF YES PRESS 1\n\n\tIF NO PRESS 0 : ");

scanf("%d",&YN);
}while(YN==1);
return(0);
}
//FIFO
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
for(i=0;i<l;i++)
{
FAULTS");
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("\tPage-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("\tNo page-fault");
}
if(j==f)
j=0;
}

}
//LRU
void lru(char s[],char F[],int l,int f)
{

int i,j=0,k,m,flag=0,cnt=0,top=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;

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("\tPage-fault%d",cnt);
cnt++;
}
else
```

```
printf("\tNo page fault");
flag=0;
}
}
```

Procedure for Execution:

Save the code with .c extension

```
[root@localhost ~]# cc filename.c
[root@localhost ~]# ./a.out
```

Output:

```
Enter the no of empty frames: 3
Enter the length of the string: 5
Enter the string: hello

********** MENU **********
1:FIFO
2:LRU
4:EXIT
   Enter your choice: 1
   PAGE   FRAMES      FAULTS
   h      h           Page-fault 0
   e      h   e       Page-fault 1
   l      h   e   l   Page-fault 2
   l      h   e   l   No page-fault
```

o o e l Page-fault 3

Do u 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 h Page-fault 0

e h e Page-fault 1

l h e l Page-fault 2

l h e l No page fault

o e l o Page-fault 3

Do u want to continue IF YES PRESS 1

IF NO PRESS 0 : 1

********** MENU **********

1:FIFO

2:LRU

4:EXIT

Enter your choice: 4

10a. Design, develop and implement a C/C++/Java program to simulate a numerical calculator

```
#include <stdio.h>
void main()
{
char operator;
```

```
float num1, num2, result;
printf("Simulation of a Simple Calculator\n");
printf("**********************************\n");
printf("Enter two numbers \n");
scanf("%f %f", &num1, &num2);
fflush(stdin);
printf("Enter the operator [+,-,*,/] \n");
scanf("%s", &operator);
switch(operator)
{
case '+': result = num1 + num2;
break;
case '-': result = num1 - num2;
break;
case '*': result = num1 * num2;
break;
case '/': result = num1 / num2;
break;
default : printf("Error in operationn");
break;
}
printf("\n %5.2f %c %5.2f = %5.2f\n", num1, operator, num2, result);
}
```

Procedure for Execution:

Save the code with .c extension

```
[root@localhost ~]# cc filename.c
[root@localhost ~]# ./a.out
```

Output:

```
Simulation of a Simple Calculator
********************************
Enter two numbers
2
3
Enter the operator [+,-,*,/]
+
2.00 + 3.00 = 5.00
```

10b.Design, develop and implement a C/C++/Java program to simulate page replacement technique

```
#include<stdio.h>
```

```
int n,nf;

int in[100];

int p[50];

int hit=0;

int i,j,k;

int pgfaultcnt=0;

void getData()

{

printf("\nEnter length of page reference sequence:");
   scanf("%d",&n);


printf("\nEnter the page reference sequence:"); for(i=0;
   i<n; i++)

scanf("%d",&in[i]);


printf("\nEnter no of frames:");

scanf("%d",&nf);

}

void initialize()

{

pgfaultcnt=0;

for(i=0; i<nf; i++)

p[i]=9999;


}


int isHit(int data)

{
```

```
hit=0;

for(j=0; j<nf; j++)
{
if(p[j]==data)
{

hit=1;
break;
}
}
return hit;
}

int getHitIndex(int data)
{
int hitind;
for(k=0; k<nf; k++)
{
if(p[k]==data)

{
hitind=k;
break;
}
}
```

```
return hitind;
}

void dispPages()
{
for (k=0; k<nf; k++)
{
if(p[k]!=9999)
printf(" %d",p[k]);

}
}

void dispPgFaultCnt()
{
printf("\nTotal no of page faults:%d",pgfaultcnt);
}

void fifo()
{
initialize();
for(i=0; i<n; i++)
{
printf("\nFor %d :",in[i]);
if(isHit(in[i])==0)
{
for(k=0; k<nf-1; k++)
```

```
p[k]=p[k+1];
p[k]=in[i];
pgfaultcnt++;
dispPages();
}
else
printf("No page fault");
}
dispPgFaultCnt();
}
void optimal()
{
initialize();
int near[50];
for(i=0; i<n; i++)
{
    printf("\nFor %d :",in[i]);
if(isHit(in[i])==0)
{
for(j=0; j<nf; j++)

{
int pg=p[j];
int found=0;
for(k=i; k<n; k++)
{
```

```
if(pg==in[k])
{
near[j]=k;
found=1;
break;
}
else
found=0;
}

if(!found)
near[j]=9999;
}
int max=-9999;
int repindex;

for(j=0; j<nf; j++)
{
if(near[j]>max)
{
max=near[j];
repindex=j;
}
}
p[repindex]=in[i];

pgfaultcnt++;
```

```
dispPages();
}
else
found=0;
}

if(!found)
near[j]=9999;
}
int max=-9999;
int repindex;

for(j=0; j<nf; j++)
{
if(near[j]>max)
{

max=near[j];
repindex=j;
}
}
p[repindex]=in[i];

pgfaultcnt++;

dispPages();
}
```

```
else
printf("No page fault");

}
dispPgFaultCnt();

}
void lru()
{
initialize();

int least[50];
for(i=0; i<n; i++)
{
printf("\nFor %d :",in[i]);

if(isHit(in[i])==0)
{
for(j=0; j<nf; j++)
{ int pg=p[j]; int found=0;
for(k=i-1; k>=0; k--) {
if(pg==in[k])
{

least[j]=k;
found=1;
break;
```

```
}

else
found=0;
}
if(!found)
least[j]=-9999;
}
int min=9999;
int repindex;
for(j=0; j<nf; j++)

{
if(least[j]<min)
{
min=least[j];
repindex=j;
}
}
p[repindex]=in[i];
pgfaultcnt++;
dispPages();
}
else
printf("No page fault!");
}
```

```
dispPgFaultCnt();
}

void lfu()
{
int usedcnt[100];
int least,repin,sofarcnt=0,bn;
initialize();
for(i=0; i<nf; i++)
usedcnt[i]=0;
for(i=0; i<n; i++)
{

printf("\n For %d :",in[i]);
if(isHit(in[i]))
{
int hitind=getHitIndex(in[i]);
usedcnt[hitind]++;
printf("No page fault!");
}
else
{
pgfaultcnt++;
if(bn<nf)
{
p[bn]=in[i];
```

```
usedcnt[bn]=usedcnt[bn]+1;
bn++;
}
else
{
least=9999;
for(k=0; k<nf; k++)
if(usedcnt[k]<least)
{
least=usedcnt[k];
repin=k;
}
p[repin]=in[i];
sofarcnt=0;

for(k=0; k<=i; k++)
if(in[i]==in[k])
sofarcnt=sofarcnt+1;
usedcnt[repin]=sofarcnt;
}
dispPages();
}
}
dispPgFaultCnt();
}
```

```
void secondchance()
{
int usedbit[50];
int victimptr=0;
initialize();
for(i=0; i<nf; i++)
usedbit[i]=0;
for(i=0; i<n; i++)
{
printf("\nFor %d:",in[i]);
if(isHit(in[i]))
{
printf("No page fault!");
int hitindex=getHitIndex(in[i]);
if(usedbit[hitindex]==0)

usedbit[hitindex]=1;
}
else
{
pgfaultcnt++;
if(usedbit[victimptr]==1)
{
do
{
usedbit[victimptr]=0;
```

```
victimptr++;
if(victimptr==nf)
victimptr=0;
}
while(usedbit[victimptr]!=0);
}
if(usedbit[victimptr]==0)
{
p[victimptr]=in[i];
usedbit[victimptr]=1;
victimptr++;
}
dispPages();
}
if(victimptr==nf)
victimptr=0;
}
dispPgFaultCnt();
}
int main()
{
int choice;
while(1)
{
printf("\nPage Replacement Algorithms\n1.Enter
data\n2.FIFO\n3.Optimal\n4.LRU\n5.LFU\n6.Second Chance\n7.Exit\nEnter your
```

```
choice:");
scanf("%d",&choice);
switch(choice)
{
case 1: getData();
break;
case 2: fifo();
break;
case 3:optimal();
break;
case 4:lru();
break;
case 5:lfu();
break;
case 6:secondchance();
break;
default:return 0;
break;
}
}
}
```

Procedure for Execution:

Save the code with .c extension

```
[root@localhost ~]# cc filename.c
[root@localhost ~]# ./a.out
```

Output:

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.LFU

6.Second Chance

7.Exit

Enter your choice:1

Enter length of page reference sequence:8

Enter the page reference sequence:2

3

4

2

3

5

6

2

Enter no of frames:3

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.LFU

6.Second Chance

7.Exit

Enter your choice:2

For 2 : 2

For 3 : 2 3

For 4 : 2 3 4

For 2 :No page fault

For 3 :No page fault

For 5 : 3 4 5

For 6 : 4 5 6

For 2 : 5 6 2

Total no of page faults:6

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.LFU

6.Second Chance

7.Exit

Enter your choice:3

For 2 : 2

For 3 : 2 3

For 4 : 2 3 4

For 2 :No page fault

For 3 :No page fault

For 5 : 2 5 4

For 6 : 2 6 4

For 2 :No page fault

Total no of page faults:5

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.LFU

6.Second Chance

7.Exit

Enter your choice:4

For 2 : 2

For 3 : 2 3

For 4 : 2 3 4

For 2 :No page fault!

For 3 :No page fault!

For 5 : 2 3 5

For 6 : 6 3 5

For 2 : 6 2 5

Total no of page faults:6

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.LFU

6.Second Chance

7.Exit

Enter your choice:5

For 2 : 2

For 3 : 2 3

For 4 : 2 3 4

For 2 :No page fault!

For 3 :No page fault!

For 5 : 2 3 5

For 6 : 2 3 6

For 2 :No page fault!

Total no of page faults:5

Page Replacement Algorithms

1.Enter data

2.FIFO

3.Optimal

4.LRU

5.LFU

6.Second Chance

7.Exit

Enter your choice:7

7. Extra Lab experiments:

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

```
%{
int ch=0,sp=0,wd=0,ln=0;
%}
%%
[\n] {ln++;}                /*For counting the line*/ [^\t\n ]+ {ch+=yyleng,wd++;}      /*For the character and words*/ " " {sp++;}              /*For counting the space*/
%%
int main(int argc,char *argv[])
{
++argv,--argc;
if(argc>0)
yyin=fopen(argv[0],"r");
else yyin=stdin; yylex();
printf("Number of characters:%d\n",ch);
printf("Number of spaces:%d\n",sp);
printf("Number of words:%d\n",wd);
printf("Number of lines:%d\n",ln);
}
```

Procedures for Excecution

Save the lex code program with .l extension.

Create a file using the editor command and write the content of the file

[root@localhost ~]# vi data.txt

December

August

[root@localhost ~]# lex filename.l

[root@localhost ~]# cc lex.yy.c -ll

[root@localhost ~]# ./a.out data.txt

OUTPUT

Number of characters:14

Number of spaces:0

Number of words:2

Number of lines:2

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

```
%{
int flag=0;
%}
%%
" and " | " or " |
" but " |          /*Words which leads to the compound statement*/ " because " |
" than " |
" nevertheless " {flag=1;}
%%
int main()
{
printf("Enter the sentence:\n");
yylex();
if(flag==1)
printf("Given sentence is compound statement\n");
else
printf("Given sentence is simple statement\n");
}
```

Procedures for Excecution

Save the lex code Program with .l extension

[root@localhost ~]# lex filename.l

[root@localhost ~]# cc lex.yy.c –ll

[root@localhost ~]# ./a.out

OUTPUT

Enter the sentence:

abc is alphabet

abc is alphabet **(Press Ctrl d)**

Given sentence is simple statement

[root@localhost ~]# ./a.out

Enter the sentence:

abc or 123 are not equal

abc123 are not equal **(Press Ctrl d)**

Given sentence is compound statement

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

```
%{
int ids=0;
%}
%%
(([0-9]+[a-zA-Z]*)|([_][0-9]*)|([`!@#$%^&*-+=][a-zA-Z]+)) {;}  //Recognize invalid identifiers
([a-zA-Z]|[_])(([0-9]|[a-zA-Z]|[_])*) {printf("%s\t",yytext);ids++;} // Recognize the identifiers
%%
int main(int argc,char *argv[])
{
--argc,++argv;
if(argc>0)
yyin=fopen(argv[0],"r");
printf("Identifiers in the given file are\n");
yylex();
```

```
printf("Number of identifiers are %d",ids);
}
```

Procedures for Excecution

Save the lex code Program with .l extension

Create a file using editor command and write the content

```
[root@localhost ~]#vi data.txt
bangalore
_bangalore
12bangalore
+bangalore
banga_lore

[root@localhost root]# lex filename.l
[root@localhost root]# cc lex.yy.c -ll
[root@localhost root]# ./a.out data.txt
```

OUTPUT

```
Identifiers in the given file are bangalore
_bangalore
banga_lore
Number of identifiers are 3
```

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

Lex Part

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z][a-zA-Z0-9]* {return ID;}
[0-9]+  {return NUMBER;}
.       {return yytext[0];}
\n      {return 0;}     /*Logical EOF*/

%%
```

Yacc part

```
%token NUMBER ID/*token definition*/
%left '+"-'       /*Operator precedence's*/
%left '*"/'       /*Operator precedence's*/
%%
expr:expr '+' expr;
|expr '-' expr;   /*Grammar*/
|expr '*' expr;
|expr '/' expr;
|'('expr')'
|NUMBER
|ID
;
%%
int main()
{
printf("Enter the Expression\n");
yyparse();
printf("Valid Expression\n");
}
int yyerror()
{
printf("Expression is invalid\n");
exit(0);
}
```

Procedures for Excecution

Save the lex code Program with .l extension

Save the yacc code Program with .y extension

[root@localhost ~]# lex filename.l

[root@localhost ~]# yacc -d filename.y

[root@localhost ~]# cc lex.yy.c y.tab.c –ll

[root@localhost ~]# ./a.out

OUTPUT

Enter the Expression	Enter the Expression
+23	2+3-4
Expression is invalid	Valid Expression

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

Lex Part

```
%{
#include"y.tab.h"
%}
%%
[0-9]    {return DIG;} [a-z]    {return LET;}
.        {return yytext[0];}
\n       {return 0;}      /*Logical EOF*/
```

```
%%
```

Yacc part

```
%token LET
%token DIG
%%
stmt:id {printf("Valid identifier \n");}
;
id: letter next
| letter {;}
;
next: letter next
| digit next
| letter
| digit {;}
;
letter: LET {;}
;
digit: DIG {;}
;
%%
int main()
{
printf("Enter an identifier:");
yyparse();
}
int yyerror()
```

```
{
printf("Not a valid identifier\n");
exit(0);
}
```

Procedures for Excecution

Save the lex code Program with .l extension

Save the yacc code Program with .y extension

```
[root@localhost ~]# lex filename.l
[root@localhost ~]# yacc -d filename.y
[root@localhost ~]# cc lex.yy.c y.tab.c -ll
[root@localhost ~]# ./a.out
```

OUTPUT

Enter an identifier:ab12	Enter an identifier:12dc
Valid identifier	Not a valid identifier

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

Lex part

```
%{
#include"y.tab.h"
%}
```

```
%%
a       {return A;}
b       {return B;}
.       {return yytext[0];}
\n {return yytext[0];}
%%
```

Yacc part

```
%token A B
%%
str : s'\n'         {return 0;}
s : A s B ;
| ;
%%
int main()
{
printf("Type the string\n");
yyparse();
printf("Valid string");
}
int yyerror()
{
printf("Invalid string");
exit(0);
}
```

<u>Procedures for Excecution</u>

Save the lex code Program with .l extension

Save the yacc code Program with .y extension

[root@localhost ~]# lex filename.l

[root@localhost ~]# yacc -d filename.y

[root@localhost ~]# cc lex.yy.c y.tab.c -ll

[root@localhost ~]# ./a.out

OUTPUT

Type the string	Type the string
aaabbb	aaab
Valid string	Invalid string

7.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

to finish (by executing the wait ( )) and prints its own process-id and the id of its child process and then exits.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
int status;
int pid,ppid,mpid; pid=fork(); if(pid<0)
{
printf("Error in forking child");
}
if(pid==0)
{
ppid=getppid();
printf("I am the child and my parent id %d\n",ppid);
mpid=getpid(); printf("Child process\n"); printf("My own id is %d\n",mpid);
}
wait();
mpid=getpid();
printf("My id is %d and my child id is %d\n",mpid,pid);
}
```

Procedures for Execution

Save the file with .c extension

[root@localhost ~]# cc filename.c

[root@localhost ~]# ./a.out

OUTPUT

I am the child and my parent id 16599

Child process

My own id is 16600

My id is 16600 and my child id is 0

My id is 16599 and my child id is 16600s

8.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>

#include<stdlib.h>

#include<unistd.h>

main()

{

int x,i;

char cmd[10];

x=fork ();        /*To create a child process*/

if(x==0)

do

{

printf("Child process has been created\n"); printf("Enter the command to be executed\n"); scanf("%s",cmd);

system(cmd);

printf("Enter 1 to continue and 0 to exit\n");

scanf("%d",&i);

} while(i!=0);

wait();

}
```

Procedures for Execution

Save the file with .c extension

[root@localhost ~]# cc filename.c

[root@localhost ~]# ./a.out

OUTPUT

Child process has been created Enter the command to be executed date

Tue Jan 20 16:17:01 IST 2009

Enter 1 to continue and 0 to exit

1

Child process has been created

Enter the command to be executed

cal

January 2009

Su	Mo	Tu	We	Th	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Enter 1 to continue and 0 to exit

0

9.Using Open MP, 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<iostream>
#include<omp.h>
#define MAX 1000
using namespace std;

int main()
{
      long limit,lock=1,fibnumbers[MAX];
      omp_set_num_threads(2);
      cout << "Enter the limit\n";
      cin >> limit;

#pragma omp parallel
      {

#pragma omp critical
           {
                 cout << "\n\n*****INSIDE CRITICAL SECTION*****\n\n";
                 if(lock == 1 )
                 {
                       cout << "Thread number " << omp_get_thread_num();
                       cout << "\nGenerating fib series\n";
                       fibnumbers[0]=0;
                       fibnumbers[1]=1;
```

```
                        for(int i=2;i<limit;i++)
                        {
                              fibnumbers[i]=fibnumbers[i-1]+fibnumbers[i-2];
                        }
                        lock=0;
                  }
                  else
                  {
                        cout << "Thread number " << omp_get_thread_num();
                        cout << "\nPrinting fib series\n";
                        for(int i=0;i<limit;i++)
                        {
                              cout << fibnumbers[i] << "  ";
                        }
                        cout << endl;
                  }
            }
      }
                  return 0;
}
```

Procedures for Execution

Save the file with .cpp extension

[root@localhost ~]# g++ filename.cpp

[root@localhost ~]# ./a.out

OUTPUT

1.
Enter the limit
40
*****INSIDE CRITICAL SECTION*****
Thread number 0
Generating fib series
*****INSIDE CRITICAL SECTION*****
Thread number 1
Printing fib series
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986

2.
Enter the limit
45

*****INSIDE CRITICAL SECTION*****

Thread number 0
Generating fib series

*****INSIDE CRITICAL SECTION*****

Thread number 1
Printing fib series
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733

8. 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 Sub expression
*	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.

- Give the structure of the lex program:-

Definition section- any initial 'c' program code

% %

Rules section- pattern and action separated by white space

%%

User subroutines section-concsit of any legal code.

- The lexer produced by lex in a 'c' routine is called yylex()
- Explain yytext:- contains the text that matched the pattern.
- The yacc produced by parser is called yyparse().
- Why we have to include 'y.tab.h' in lex?

y.tab.h contains token definitions eg:- #define letter 258.

- Explain the structure of a yacc program?

Defn section- declarations of the tokens used in the grammar

%%

The rules section-pattern action

%%

User's subroutines section

- Explain yyleng?

Yyleng-contains the length of the string our lexer recognizes.

- What are tokens or terminal symbols?

Symbols that appear in the input are returned by the lexer are terminal symbols.

- What type of data structures is used by shift/reduce parsing?

Stack.

- Shell- it is a command interpreter
- Kernel- is the core of the operating system.
- What is lexical analyzer?

Lex taking a set of descriptions of possible tokens and producing a ‘C” routine is called a lexical analyzer (or) lexer (or) scanner.

- Define grammar?

The list of rules that define the relationship that the program understands is a grammar.

- What is symbol table?

The table of words is a simple symbol table, a common structure in lex and yacc applications.

- What is pseudo token?

A pseudo token standing for unary minus, has no associatively is at the highest precedence.

- What does $$ represents?

$$ represents the value of the left hand side.