POORNIMA
COLLEGE OF ENGINEERING

ISI-6, RIICO Institutional Area, Sitapura, Jaipur-302022, Rajasthan

Phone/Fax: 0141-2770790-92, www.pce.poornima.org

Compiler Design Lab Manual

(Lab Code: 5CS4-22)

5th Semester, 3rd Year

Department of Computer Engineering

Session: 2022-23

TABLE OF CONTENT

S. No.	Topic/Name of Experiment	Page Number
	GENERAL DETAILS	
1	Vision & Mission of Institute and Department	iii
2	RTU Syllabus and Marking Scheme	iv
3	Lab Outcomes and its Mapping with POs and PSOs	v-vii
4	Rubrics of Lab	viii-ix
5	Lab Conduction Plan	x
6	Lab Rotar Plan	xi
7	General Lab Instructions	xii-xiii
8	Lab Specific Safety Rules	xiii
	LIST OF EXPERIMENTS (AS PER RTU SYLLABUS)	
1	Zero Lab	1-12
2	Introduction: Objective, scope and outcome of the course.	13
3	To identify whether given string is keyword or not.	14
4	Count total no. of keywords in a file. [Taking file from user]	15-16
5	Count total no of operators in a file. [Taking file from user]	17
6	Count total occurrence of each character in a given file. [Taking file from user]	18
7	Write a C program to insert, delete and display the entries in Symbol Table.	19-20
8	Write a LEX program to identify following: 1. Valid mobile number 2. Valid url 3. Valid identifier 4. Valid date (dd/mm/yyyy) 5. Valid time (hh:mm:ss)	21-23
9	Write a lex program to count blank spaces, words, lines in a given file.	24
10	Write a lex program to count the no. of vowels and consonants in a C file.	25
11	Write a YACC program to recognize strings aaab,abbb using a^nb^n, where b>=0.	26
12	Write a YACC program to evaluate an arithmetic expression involving operators +,-,* and /.	27
13	Write a YACC program to check validity of a strings abcd,aabbcd using grammar a^nb^nc^md^m, where n , m>0	28-29
14	Write a C program to find first of any grammar.	30
15	Beyond the syllabus Exp-1	31-32
16	Beyond the syllabus Exp-2	33-36

INSTITUTE VISION & MISSION

VISION

To create knowledge-based society with scientific temper, team spirit and dignity of labor to face the global competitive challenges.

MISSION

To evolve and develop skill-based systems for effective delivery of knowledge so as to equip young professionals with dedication & commitment to excellence in all spheres of life

DEPARTMENT VISION & MISSION

VISION

Evolve as a center of excellence with wider recognition and to adapt the rapid innovation in Computer Engineering.

MISSION

- To provide a learning-centered environment that will enable students and faculty members to achieve their goals empowering them to compete globally for the most desirable careers in academia and industry.
- To contribute significantly to the research and the discovery of new arenas of knowledge and methods in the rapid developing field of Computer Engineering.
- To support society through participation and transfer of advanced technology from one sector to another.

RTU SYLLABUS AND MARKING SCHEME

5CS4-22: COMPILER DESIGN LAB	
Credit: 1	**Max. Marks: 100 (IA:60, ETE:40)**
0L+0T+2P	**End Term Exam: 2 Hours**

S. No.	**NAME OF EXPERIMENTS**
1	Introduction: Objective, scope and outcome of the course.
2	To identify whether given string is keyword or not.
3	Count total no. of keywords in a file. [Taking file from user]
4	Count total no of operators in a file. [Taking file from user]
5	Count total occurrence of each character in a given file. [Taking file from user]
6	Write a C program to insert, delete and display the entries in Symbol Table.
7	Write a LEX program to identify following: 1. Valid mobile number 2. Valid url 3. Valid identifier 4. Valid date (dd/mm/yyyy) 5. Valid time (hh:mm:ss)
8	Write a lex program to count blank spaces,words,lines in a given file.
9	Write a lex program to count the no. of vowels and consonants in a C file.
10	Write a YACC program to recognize strings aaab,abbb using a^nb^n, where b>=0.
11	Write a YACC program to evaluate an arithmetic expression involving operators +,-,* and /.
12	Write a YACC program to check validity of a strings abcd,aabbcd using grammar a^nb^nc^md^m, where n , m>0
13	Write a C program to find first of any grammar.

EVALUATION SCHEME

I+II Mid Term Examination			**Attendance and performance**			**End Term Examination**			**Total Marks**
Experiment	**Viva**	**Total**	**Attendance**	**Performance**	**Total**	**Experiment**	**Viva**	**Total**	
30	10	40	10	30	40	30	10	40	100

DISTRIBUTION OF MARKS FOR EACH EXPERIMENT

Attendance	**Record**	**Performance**	**Total**
2	3	5	10

LAB OUTCOME AND ITS MAPPING WITH PO & PSO

LAB OUTCOMES

After completion of this course, students will be able to –

5CS4-22.1	To Analysis the finite state machines, lexical analyzer, parser for the grammar.
5CS4-22.2	To Develop recognition of identifiers, constants, comments, operators, loops and keywords, and generation of parse tree and syntax tree, symbol table and non-recursive grammar-based constructs.
5CS4-22.3	To Design intermediate code generator and converted into optimized code
5CS4-22.4	To demonstrate hands on experience of working on system software.

LO-PO-PSO MAPPING MATRIX OF COURSE

LO/PO/ PSO	**PO1**	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO11**	**PO12**	**PSO1**	**PSO2**	**PSO3**
5CS4-22.1	-	-	-	-	-	-	-	-	3	-	-	-	3	-	-
5CS4-22.2	-	-	-	-	3	-	-	-	-	-	-	-	2	-	-
5CS4-22.3	-	-	-	-	-	-	-	-	3	-	-	-	2	-	-
5CS4-22.4	-	-	-	-	-	3	-	-	-	-	-	-	-	3	-

PROGRAM OUTCOMES (POs)

PO1	**Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems
PO2	**Problem analysis:** Identify, formulate, review research literature, and analyze complex engineeringproblems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4	**Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

PSO1	The ability to understand and apply knowledge of mathematics, system analysis & design, Data Modelling, Cloud Technology, and latest tools to develop computer based solutions in the areas of system software, Multimedia, Web Applications, Big data analytics, IOT, Business Intelligence and Networking systems.
PSO2	The ability to understand the evolutionary changes in computing, apply standards and ethical practices in project development using latest tools & Technologies to solve societal problems and meet the challenges of the future.
PSO3	The ability to employ modern computing tools and platforms to be an entrepreneur, lifelong learning and higher studies.

RUBRICS FOR LAB

Laboratory Evaluation Rubrics:

S. No.	Criteria	Sub Criteria and Marks Distribution			Outstanding (>90%)	Admirable (70-90%)	Average (40-69%)	Inadequate (<40%)
		Mid-Term	End-Team	Continues Evaluation				
A	PERFORMANCE (PO1, PO8, PO9)	Procedure Followed M.M. 50 = 3 M.M. 75 = 4 M.M. 100 = 6	Procedure Followed M.M. 50 = 3 M.M. 75 = 4 M.M. 100 = 6	Procedure Followed M.M. 50 = 1 M.M. 75 = 2 M.M. 100 = 2	• All possible system and Input/ Output variables are taken into account • Performance measures are properly defined • Experimental scenarios are very well defined	•Most of the system and Input/ Output variables are taken into account • Most of the Performance measures are properly defined • Experimental scenarios are defined correctly	• Some of the system and Input/ Output variables are taken into account • Some of the Performance measures are properly defined • Experimental scenarios are defined but not sufficient	•System and Input/ Output variables are not defined • Performance measures are not properly defined • Experimental scenarios not defined
		Individual/Team Work M.M. 50 = 3 M.M. 75 =4 M.M. 100 = 6	Individual/Team Work M.M. 50 = 3 M.M. 75 =4 M.M. 100 = 6	Individual/Team Work M.M. 50 = 1 M.M. 75 = 2 M.M. 100 = 2	•Coordination among the group members in performing the experiment was excellent	•Coordination among the group members in performing the experiment was good	•Coordination among the group members in performing the experiment was average	•Coordination among the group members in performing the experiment was very poor
		Precision in data collection M.M. 50 = 3 M.M. 75 = 4 M.M. 100 = 6	Precision in data collection M.M. 50 = 3 M.M. 75 = 4 M.M. 100 = 6	Precision in data collection M.M. 50 = 2 M.M. 75 = 2 M.M. 100 = 4	•Data collected is correct in size and from the experiment performed	•Data collected is appropriate in size and but not from proper sources.	•Data collected is not so appropriate in size and but from proper sources.	•Data collected is neither appropriate in size and norfrom proper sources
B	LAB RECORD/WRITTEN WORK (PO1, PO8, PO10)	NA	NA	Timing of Evaluation of Experiment M.M. 50 = 3 M.M. 75 = 4 M.M. 100 = 6	• On the Same Date of Performance	• On the Next Turn from Performance	• Before Dead Line	• On the Dead Line
		Data Analysis M.M. 50 = 3 M.M. 75 =5 M.M. 100 = 6	Data Analysis M.M. 50 = 3 M.M. 75 =5 M.M. 100 = 6	Data Analysis M.M. 50 = 2 M.M. 75 = 3 M.M. 100 = 4	•Data collected is exhaustively analyzed & appropriate features are selected	•Data collected is analyzed & but appropriate features are not selected	•Data collected is not analyzed properly. •Features selected are not appropriate	•Data collected is not analyzed & the features are not selected

		Results and Discussion M.M. 50 = 3 M.M. 75 =5 M.M. 100 = 6	Results and Discussion M.M. 50 = 3 M.M. 75 =5 M.M. 100 = 6	Results and Discussion M.M. 50 = 2 M.M. 75 = 3 M.M. 100 = 4	•All results are very well presented with all variables • Well prepared neat diagrams/plots/ tables for all performance measured • Discussed critically behavior of the system with reference to performance measures • Very well discussed pros n cons of outcome	•All results presented but not all variables mentioned • Prepared diagrams /plots/ tables for all performance measured but not so neat • Discussed behavior of the system with reference to performance measures but not critical • Discussed pros n cons of outcome in brief	•Partial results are included • Prepared diagrams /plots/ tables partially for the performance measures • Behavior of the system with reference to performance measures has been superficially presented • Discussed pros n cons of outcome but not so relevant	•Results are included but not as per experimental scenarios • No proper diagrams /plots/ tables are prepared • Behavior of the system with reference to performance measures has not been presented • Did not discuss pros n cons of outcome
C	**VIVA (PO1, PO10)**	Way of presentation M.M. 50 = 2.5 M.M. 75 = 4 M.M. 100 = 5	Way of presentation M.M. 50 = 2.5 M.M. 75 = 4 M.M. 100 = 5	Way of presentation M.M. 50 = 2 M.M. 75 = 3 M.M. 100 = 4	•Presentation was very good	•Presentation was good	•Presentation was satisfactory	•Presentation was poor
		Concept Explanation M.M. 50 = 2.5 M.M. 75 = 4 M.M. 100 = 5	Concept Explanation M.M. 50 = 2.5 M.M. 75 = 4 M.M. 100 = 5	Concept Explanation M.M. 50 = 2 M.M. 75 = 3 M.M. 100 = 4	•Conceptual explanation was excellent	•Conceptual explanation was good	•Conceptual explanation was somewhat good	•Conceptual explanation was Poor
D	**ATTENDANCE**	NA	NA	**A**ttendance M.M. 50 = 5 M.M. 75 =8 M.M. 100 =10	•Present more than 90% of lab sessions	•Present more than 75% of lab sessions	•Present more than 60% of lab sessions	•Present in less than 60% lab sessions

LAB CONDUCTION PLAN

Total number of Experiments - 13

Total number of turns required - 13

Number of turns required for: -

Experiment Number	**Scheduled Week**
Experiment -1	Week 1
Experiment -2	Week 2
Experiment -3	Week 3
Experiment -4	Week 4
Experiment -5	Week 5
I Mid Term	**Week 6**
Experiment -6	Week 7
Experiment -7	Week 8
Experiment -8	Week 9
Experiment-9	Week 10
Experiment-10	Week 11
Experiment-11	Week 12
Experiment-12	Week 13
Experiment-13	Week 14
II Mid Term	**Week 15**

DISTRIBUTION OF LAB HOURS

S. No.	**Activity**	**Distribution of Lab Hours**	
		Time (180 minute)	**Time (120 minute)**
1	Attendance	5	5
2	Explanation of Experiment & Logic	30	30
3	Performing the Experiment	60	30
4	File Checking	40	20
5	Viva/Quiz	30	20
6	Solving of Queries	15	15

LAB ROTAR PLAN

ROTOR-1

Ex. No.	NAME OF EXPERIMENTS
1	Introduction: Objective, scope and outcome of the course.
2	To identify whether given string is keyword or not.
3	Count total no. of keywords in a file. [Taking file from user]
4	Count total no of operators in a file. [Taking file from user]
5	Count total occurrence of each character in a given file. [Taking file from user]
6	Write a C program to insert, delete and display the entries in Symbol Table.

ROTOR-2

Ex. No.	NAME OF EXPERIMENTS
7	Write a LEX program to identify following: 1. Valid mobile number 2. Valid url 3. Valid identifier 4. Valid date (dd/mm/yyyy) 5. Valid time (hh:mm:ss)
8	Write a lex program to count blank spaces,words,lines in a given file.
9	Write a lex program to count the no. of vowels and consonants in a C file.
10	Write a YACC program to recognize strings aaab,abbb using a^nb^n, where b>=0.
11	Write a YACC program to evaluate an arithmetic expression involving operators +,-,* and /.
12	Write a YACC program to check validity of a strings abcd,aabbcd using grammar a^nb^nc^md^m, where n , m>0
13	Write a C program to find first of any grammar.

GENERAL LAB INSTRUCTIONS

DO'S

1. Enter the lab on time and leave at proper time.
2. Wait for the previous class to leave before the next class enters.
3. Keep the bag outside in the respective racks.
4. Utilize lab hours in the corresponding.
5. Turn off the machine before leaving the lab unless a member of lab staff has specifically told you not to do so.
6. Leave the labs at least as nice as you found them.
7. If you notice a problem with a piece of equipment (e.g., a computer doesn't respond) or the room in general (e.g., cooling, heating, lighting) please report it to lab staff immediately. Do not attempt to fix the problem yourself.

DON'TS

1. Don't abuse the equipment.
2. Do not adjust the heat or air conditioners. If you feel the temperature is not properly set, inform lab staff; we will attempt to maintain a balance that is healthy for people and machines.
3. Do not attempt to reboot a computer. Report problems to lab staff.
4. Do not remove or modify any software or file without permission.
5. Do not remove printers and machines from the network without being explicitly told to do so by lab staff.
6. Don't monopolize equipment. If you're going to be away from your machine for more than 10 or 15 minutes, log out before leaving. This is both for the security of your account, and to ensure that others are able to use the lab resources while you are not.
7. Don't use internet, internet chat of any kind in your regular lab schedule.
8. Do not download or upload of MP3, JPG or MPEG files.
9. No games are allowed in the lab sessions.
10. No hardware including USB drives can be connected or disconnected in the labs without

prior permission of the lab in-charge.

11. No food or drink is allowed in the lab or near any of the equipment. Aside from the fact that it leaves a mess and attracts pests, spilling anything on a keyboard or other piece of computer equipment could cause permanent, irreparable, and costly damage. (and in fact *has*) If you need to eat or drink, take a break and do so in the canteen.
12. Don't bring any external material in the lab, except your lab record, copy and books.
13. Don't bring the mobile phones in the lab. If necessary, then keep them in silence mode.
14. Please be considerate of those around you, especially in terms of noise level. While labs are a natural place for conversations of all types, kindly keep the volume turned down.
15. If you are having problems or questions, please go to either the faculty, lab in-charge or the lab supporting staff. They will help you. We need your full support and cooperation for smooth functioning of the lab.

LAB SPECIFIC SAFETY RULES

Before entering in the lab

1. All the students are supposed to prepare the theory regarding the next experiment/ Program.
2. Students are supposed to bring their lab records as per their lab schedule.
3. Previous experiment/program should be written in the lab record.
4. If applicable trace paper/graph paper must be pasted in lab record with proper labeling.
5. All the students must follow the instructions, failing which he/she may not be allowed in the lab.

While working in the lab

1. Adhere to experimental schedule as instructed by the lab in-charge/faculty.
2. Get the previously performed experiment/ program signed by the faculty/ lab in charge.
3. Get the output of current experiment/program checked by the faculty/ lab in charge in the lab copy.
4. Each student should work on his/her assigned computer at each turn of the lab.
5. Take responsibility of valuable accessories.

Zero Lab

Lexical Elements of the C Language

Like any other High-level language, C provides a collection of basic building blocks, symbolic words called lexical elements of the language. Each lexical element may be a symbol, operator, symbolic name or word, a special character, a label, expression, reserve word, etc. All these lexical elements are arranged to form the statements using the syntax rules of the C language. Following are the lexical elements of the C language.

C Character Set

Every language has its own character set. The character set of the C language consists of basic symbols of the language. A character indicates any English alphabet, digit or special symbol including arithmetic operators. The C language character set includes:

- Letter, Uppercase A ….. Z, Lower case a….z
- Digits, Decimal digits 0….9.
- Special Characters, such as comma, period. semicolon; colon: question mark?, apostrophe‘ quotation mark “ Exclamation mark ! vertical bar | slash / backslash \ tilde ~ underscore _ dollar
- $ percent % hash # ampersand & caret ^ asterisk * minus – plus + <, >, (,), [,], {, }
- White spaces such as blank space, horizontal tab, carriage return, new line and form feed.

C Tokens

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in C program, the smallest individual units are known as C tokens. C has following tokens

- Keywords or Reserve words such as float, int, etc
- Constants such 1, 15,5 etc
- Identifiers such name, amount etc
- Operators such as +, -, * etc
- Separators such as :, ;, [,] etc and special characters
- Strings

Keywords

Key words or Reserve words of the C language are the words whose meaning is already defined and explained to the C language compiler. Therefore Reserve words cannot be used as identifiers or variable names. They should only be used to carry the pre-defined meaning. For example int is a reserve word. It indicates the data type of the variable as integer. Therefore it is reserved to carry the specific meaning. Any attempt to use it other than the intended purpose will generate a compile time error. C language has 32 keywords. Following are some of them

{auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static,struct,switch,typedef, union, unsigned, void, volatile,while}

Constants

A constant can be defined as a value or a quantity which does not change during the execution of a program. Meaning and value of the constant remains unchanged throughout the execution of the program. These are also called as literals. C supports following types of constant.

Integer Constants

An integer constant refers to a sequence of digits. There three types of integer constants, namely decimal, octal and hexadecimal. Decimal integer constant consists of set of digits from 0 to 9 preceded by an optional + or – sign.

Ex: 123, -321, 0, 4567, + 78

Embedded spaces, commas and non-digit characters are not permitted between digits. An octal integer constant consists of any combination of digits from 0 to 7 with a leading 0(zero). Ex : 037, 0435, 0567. A sequence of digits preceded by 0x or 0X is considered as hexadecimal digit. They may also include alphabets A to F or a to f representing numbers from 10 to 15. The largest integer value that can be stored is machine dependent; It is 32767 for 16 bit computers. It is also possible to store larger integer constants by appending qualifiers such U, L and UL to the constants.

Floating point Constants or Real Constants

The quantities that are represented by numbers with fractional part are called floating point numbers. Ex: 0.567, -0.76, 56.78, +247.60. These numbers are shown in decimal notation , having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point or digits after the decimal point. Ex: 215., .95, -.76 or +.5 A real number or a floating point number can also expressed as in exponential notation. Ex: 2.15E2. The general form of exponential notation is mantissa e exponent or mantissa E exponent The mantissa is either a real number or an integer. The exponent is an integer with an optional plus or minus sign. Embedded white is not allowed in this notation.

Single Character constants

A single character constant contains a any valid character enclosed within a pair of single quote marks. Ex: '5', 'A', ';' ' '. The character constants have integer values associated with them known as ASCII values. For ex: A is having the ASCII value of 65.

String constants

A string constant is a sequence of characters enclosed in double quotes. The characters may be alphabets, numbers special characters and blank space. Ex: "Hello", "2002", "Wel Come", "5+3"

Backslash character constants or Escape sequence characters

C supports some special backslash character constants that are used in output functions. For ex: '\n' stands for new line characters. Each one of them represents a single character even though it consists of two characters.

\a	Audible alert or bell	\b	back space
\f	form feed	\n	new line
\r	carriage return	\t	horizontal tab
\v	vertical tab	\’	single quote
\”	double quote	\?	Question mark
\\	backslash	\0	null

Identifiers

An identifier is a sequence of letters, digits and an underscore. Identifiers are used to identify or name program elements such as variables, function names, etc. Identifiers give unique names to various elements of the program. Some identifiers are reserved as special to the C language. They are called keywords.

Variables

A variable is a data name that may be used to store data value. A value or a quantity which may vary during the program execution can be called as a variable. Each variable has a specific memory location in memory unit, where numerical values or characters can be stored. A variable is represented by a symbolic name. Thus variable name refers to the location of the memory in which a particular data can be stored. Variables names are also called as identifiers since they identify the varying quantities. For Ex : sum = a+b. In this equation sum, a and b are the identifiers or variable names representing the numbers stored in the memory locations.

Rules to be followed for constructing the Variable names(identifiers)

- They must begin with a letter and underscore is considered as a letter.
- It must consist of single letter or sequence of letters, digits or underscore character.
- Uppercase and lowercase are significant. For ex: Sum, SUM and sum are three distinct variables.
- Keywords are not allowed in variable names.
- Special characters except the underscore are not allowed.
- White space is also not allowed.
- The variable name must be 8 characters long. But some recent compilers like ANSI C supports 32 characters for the variable names and first 8 characters are significant in most compilers.

Data Types

Data refers to any information which is to be stored in a computer. For example, marks of a student, salary of a person, name of a person etc. These data may be of different types. Computer allocates memory to a variable depending on the data that is to be stored in the variable. So it becomes necessary to define the type of the data which is to be stored in a variable while declaring a variable. The different data types supported by C are as follows:

Int Data Type

Integer refers to a whole number with a range of values supported by a particular machine. Generally integers occupy one word of storage i.e 16 bits or 2 bytes. So its value can range from -32768 to

+32767.

Declaration: int variable name; Ex: int qty;

The above declaration will allocate a memory location which can store only integers, both positive and negative. If we try to store a fractional value in this location, the fractional data will be lost.

Float Data Type

A floating point number consists of sequence of one or more digits of decimal number system along with embedded decimal point and fractional part if any. Computer allocates 32 bits, i.e. 4 bytes of memory for storing float type of variables. These numbers are stored with 6 digits of precision for fractional part.

Declaration: float variableName; Ex : float amount;

Double Data Type

It is similar to the float type. It is used whenever the accuracy required to represent the number is more. In others words variables declared of type double can store floating point numbers with number of significant digits is roughly twice or double than that of float type. It uses 64 bits i.e. 8 bytes of memory giving a precision of 14 decimal digits.

Declaration : double variableName;

Char Data Type

A single character can be defined as char data type. These are stored usually as 8 bits i.e 1 byte of memory.

Declaration : char variableName ; Ex: char pass;

String refers to a series of characters. Strings are declared as array of char types. Ex: char name[20]; will reserve a memory location to store upto 20 characters.

Further, applying qualifiers to the above primary data types yield additional data types. A qualifier alters the characteristics of the data type, such as its sign or size. There are two types of qualifiers namely, sign qualifiers and size qualifiers. signed and unsigned are the sign qualifiers short and long are the size qualifiers.

Size and range of data types on a 16 bit Machine

Type	Size (bits)	Range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int	16	-32768 to 32767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 tc 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4e-38 to 3.4e+38
double	64	1.7e-308 to 1.7e+308
long double	80	3.4 e-4932 to 1.1e+4932

Declaring a variable as constant

We may want the value of the certain variable to remain constant during the execution of the program. We can achieve this by declaring the variable with const qualifier at the time of initialization.

Ex: const int tax_rate = 0.30;

The above statement tells the compiler that value of variable must not be modified during the execution of the program. Any attempt change the value will generate a compile time error.

Declaring a variable as volatile

Declaring the variable volatile qualifier tells the compiler explicitly that the variable's value may be changed at any time by some external source and the compiler has to check the value of the variable each time it is encountered.

Ex; volatile int date;

Defining Symbolic Constants

We often use certain unique constants in a program. These constants may appear repeatedly in number of places in a program. Such constants can be defined and its value can be substituted during the preprocessing stage itself.

Operators in C

An operator is a symbol which acts on operands to produce certain result as output. For example in the expression a+b; + is an operator, a and b are operands. The operators are fundamental to any mathematical computations.

Operators can be classified as follows

- Based on the number of operands the operator acts upon:
  - Unary operators: acts on a single operand. For example: unary minus(5, -20, etc), address of operator (&a)
  - Binary operators: acts on two operands. Ex: +, -, %, /, *, etc
  - Ternary operator: acts on three operands. The symbol ?: is called ternary operator in C language. Usage: big= a>b?a:b; i.e if a>b, then big=a else big=b.
  - Based on the functions
  - Arithmetic operators
  - Relational operators
  - Logical Operators
  - Increment and Decrement operators
  - Assignment operators
  - Bitwise operators
  - Conditional Operators
  - Special operators

Precedence and Associativity of operators

Each operator in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. The operator at the higher level of precedence is evaluated first. The operators of the same precedence are evaluated either from left to right or from right to left depending on the level. This is known as the associativity property of an operator.

Operator	Description	Level	Associativity
( ) [ ]	Parenthesis Array index	1	L – R
+ - ++ -- ! ~ & sizeof(type)	Unary plus Unary minus Increment Decrement Logical negation One's Complement Address of type cast conversion	2	R – L
* / %	Multiplication Division Modulus	3	L- R
+ -	Addition Subtraction	4	L – R
<< >>	Left Shift Right Shift	5	L – R
< <=	Less than Less than or equal to	6	L – R

> >=	Greater than Greater than or equal to		
= = ! =	is equal to Not equal to	7	L – R
&	Bitwise AND	8	L – R
^	Bitwise XOR	9	L – R
|	Bitwise OR	10	L – R
&&	Logical AND	11	L – R
| |	Logical OR	12	L – R
? :	Conditional Operator	13	R – L
=, +=, -=, *=, /=, %=	Assignment operator Short hand assignement	14	R – L
,	Comma operator	15	R – L

Preprocessor directives

There are different preprocessor directives. The table below shows the preprocessor directives.

Directive	**Function**
#define	defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included.
#ifdef	Tests for a macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Tests a compile-time condition
#else	Specifies alternatives when #if tests fails

Header files

C language offers simpler way to simplify the use of library functions to the greatest extent possible. This is done by placing the required library function declarations in special source files, called header files. Most C compilers include several header files, each of which contains declarations that are functionally related. stdio.h is a header file containing declarations for input/ouput routines; math.h contains declarations for certain mathematical functions and so on. The header files also contain other information related to the use of the library functions, such as symbolic constant definitions.

The required header files must be merged with the source program during the compilation process. This is accomplished by placing one or more #include statements at the beginning of the source program. The other header files are:

<ctype.h>	character testing and conversion functions
<stdlib.h>	utility functions such as string conversion routines , memory allocation routines, random number generator etc
<string.h>	String manipulations functions
<time.h>	Time manipulation functions

Input and Output Functions

The C language consists of input-output statements to read the data to be processed as well as output the computed results. C language provides a set of library functions or built in functions, in order to carry out input and output operations. These library functions are available in a header file called stdio.h. So for using these library functions the following preprocessor directive is essential.

The input and output functions in C language can be broadly categorized into two types:

- Unformatted Input Output functions : which provides the facility to read or output data as a characters or sequence of characters. Ex: getch(), getche(), getchar(), gets(), putch(), purchar() and puts().
- Formatted I/O functions : which allow the use format specifiers to specify the type of data to be read or printed. Ex: scanf() and printf() functions.

Introduction to Lex Programming

The unix utility lex parses a file of characters. It uses regular expression matching; typically it is used to ‘tokenize’ the contents of the file. In that context, it is often used together with the yacc utility. However, there are many other applications possible.

Structure of a lex file

A lex file looks like:

```
...definitions...
%%
...rules...
%%
...code...
```

Here is a simple example:

```
%{
int charcount=0,linecount=0;
%}

%%
. charcount++;
\n {linecount++; charcount++;}

%%

Int main( )
{
Yylex ( );
                        printf("There were %d characters in %d lines\n", charcount,linecount);
                         return 0;
                        }
```

In the example you just saw, all three sections are present:

- **definitions** 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 analyser. If the code segment is left out, a default main is used which only calls yylex.

Introduction to YACC Programming

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

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

date : month_name day ',' year ;

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

July 4, 1776

These are some points about YACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

The output file "file.output" contains the parsing tables.
The file "file.tab.h" contains declarations.
The parser called the yyparse ().
Parser expects to use a function called yylex () to get tokens.

EXPERIMENT-1

OBJECTIVE

Introduction: Objective, scope and outcome of the course.

THEORY

The laboratory course is intended to make experiments on the basic techniques of compiler construction and tools that can be used to perform syntax-directed translation of a high-level programming language into an executable code. Students will design and implement language processors in C by using tools to automate parts of the implementation process. This will provide deeper insights into the more advanced semantics aspects of programming languages, code generation, machine independent optimizations, dynamic memory allocation, and object orientation.

SCOPE

The scope of this course is to explore the principle, algorithm and data structure involved in the design and construction of compiler.

OUTCOMES

Upon the completion of Compiler Design practical course, the student will be able to:

1. Understand the working of lex and yacc compiler for debugging of programs.
2. Understand and define the role of lexical analyzer, use of regular expression and transition diagrams.
3. Understand and use Context free grammar, and parse tree construction.
4. Learn & use the new tools and technologies used for designing a compiler.
5. Develop program for solving parser problems.
6. Learn how to write programs that execute faster.

EXPERIMENT-2

OBJECTIVE

To identify whether given string is keyword or not.

PROGRAM

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


void main()
            {
char a[5][10]={"printf","scanf","if","else","break"};
char str[10];
int i,flag; clrscr();

puts("Enter the string :: "); gets(str);


for(i=0;i<strlen(str);i++)
{
if(strcmp(str,a[i])==0)
{



}
else

                                                            }
if(flag==1)
puts("Keyword");
else
puts("String");


getch();
            }
```

OUTPUT

```
            Printf
                                                            Keyword
```

```
flag=1; break;


flag=0;
```

EXPERIMENT-3

OBJECTIVE

Count total no. of keywords in a file. [Taking file from user]

PROGRAM

```
#include<stdio.h>

#include<stdlib.h> #include<string.h> #include<ctype.h> static int count=0;
int isKeyword(char buffer[]){

char keywords[32][10] =
{"auto","break","case","char","const","continue","default","do","double","else","enum","exte
rn","fl
oat","for","goto","if","int","long","register","return","short","signed","sizeof","static","struct","swi
tc h","typedef","union","unsigned","void","volatile","while"};
int i, flag = 0;

for(i = 0; i < 32; ++i){ if(strcmp(keywords[i], buffer) == 0){
flag = 1;
            count++;}}

            return flag;
}

int main(){
char ch, buffer[15] ; FILE *fp;
int i,j=0;

fp = fopen("KESHAV3.C","r"); if(fp == NULL){
printf("error while opening the file\n"); exit(0);
}

while((ch = fgetc(fp)) != EOF){ if(isalnum(ch)){
buffer[j++] = ch;
}
else if((ch == ' ' || ch == '\n') && (j != 0)){
buffer[j] = '\0'; j = 0;

if(isKeyword(buffer) == 1)
printf("%s is keyword\n", buffer);
}

}
printf("no of keywords= %d", count); fclose(fp);

return 0;
}
```

OUTPUT

```
Break case

  2
```

EXPERIMENT-4

OBJECTIVE

Count total no of operators in a file. [Taking file from user]

PROGRAM

```
#include<stdlib.h>

#include<string.h>
#include<ctype.h> static int count=0; int main(){
char ch, buffer[15], operators[] = "+-*/%="; FILE *fp;
int i; clrscr();
fp = fopen("KESHAV3.C","r"); if(fp == NULL){
printf("error while opening the file\n");
exit(0);
}

while((ch = fgetc(fp)) != EOF){ for(i = 0; i < 6; ++i){
if(ch == operators[i]) {
printf("%c is operator\n", ch); count++;
}
}
}
printf("no of operators= %d", count); fclose(fp);
return 0;
}
```

OUTPUT

```
+_*
3
```

EXPERIMENT-5

OBJECTIVE

Count total occurrence of each character in a given file. [Taking file from user]

PROGRAM

```
#include <stdio.h>
#include <string.h>
#include<conio.h>
int main ()
{
FILE * fp;
char string[100];
int c = 0, count[26] = { 0 }, x;
fp = fopen ("deepa.txt", "r"); clrscr();
while (fscanf (fp, "%s", string) != EOF)
{           c=0;
while (string[c] != '\0')
{
/** Considering characters from 'a' to 'z' only and ignoring others. */ if (string[c] >= 'a' &&
string[c] <= 'z')
{
x = string[c] - 'a'; count[x]++;
}
 c++;
}
}


for (c = 0; c <26 ; c++)
printf ("%c occurs %d times in the string.\n", c + 'a', count[c]); return 0;
}
```

OUTPUT

```
NEHA
N=1
E=1
H=1
A=1
```

EXPERIMENT-6

OBJECTIVE

Write a C program to insert, delete and display the entries in Symbol Table.

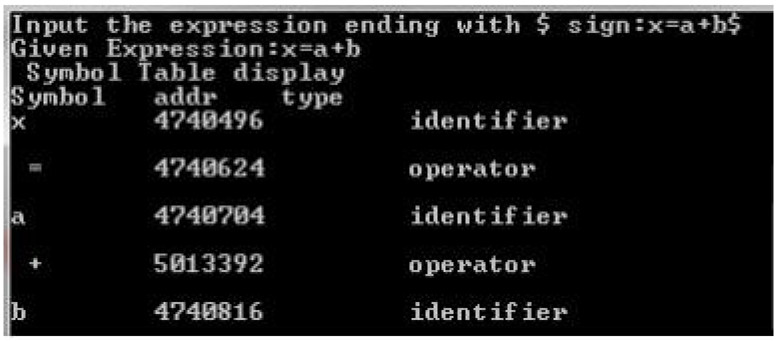
PROGRAM

//Implementation of symbol table

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
void main()
{
int i=0,j=0,x=0,n; void *p,*add[5];
char ch,srch,b[15],d[15],c; printf("Expression terminated by $:"); while((c=getchar())!='$')
{
b[i]=c; i++;
}
n=i-1;
printf("Given Expression:"); i=0;
while(i<=n)
{
printf("%c",b[i]); i++;
}
printf("\n Symbol Table\n"); printf("Symbol \t addr \t type"); while(j<=n)
{
c=b[j]; if(isalpha(toascii(c)))
{
p=malloc(c); add[x]=p;
d[x]=c;
printf("\n%c \t %d \t identifier\n",c,p); x++;
j++;
}
else
{
```

```
ch=c;
if(ch=='+'||ch=='-'||ch=='*'||ch=='=')
{
p=malloc(ch); add[x]=p;
d[x]=ch;
printf("\n %c \t %d \t operator\n",ch,p); x++;
j++;
}}}}
```

OUTPUT



```
Input the expression ending with $ sign:x=a+b$
Given Expression:x=a+b
 Symbol Table display
Symbol   addr    type
x        4740496         identifier

 =       4740624         operator

a        4740704         identifier

 +       5013392         operator

b        4740816         identifier
```

EXPERIMENT-7

OBJECTIVE

Write a LEX program to identify following:

1. Valid mobile number

PROGRAM

```
        %{
/* Definition section */
%}

/* Rule Section */
%%

[1-9][0-9]{9} {printf("\nMobile Number Valid\n");}

.+ {printf("\nMobile Number Invalid\n");}

%%

// driver code int main()
{
printf("\nEnter Mobile Number : "); yylex();
printf("\n"); return 0;
}
int yywrap()
{
}
```

OUTPUT

```
        7357733397
        Mobile Number Valid
```

2. Valid URL

PROGRAM

```
%%
((http)|(ftp))s?:\/\/[a-zA-Z0-9]{2,}(\.[a-z]{2,})+(\/[a-zA-Z0-9+=?]*)* {printf("\nURL
Valid\n");}
.+ {printf("\nURL Invalid\n");}

%%
void main() {
printf("\nEnter URL : "); yylex();
printf("\n");
}
int yywrap()
{
}
```

OUTPUT

```
            www.google.com
      URL Valid
```

3. Valid identifier

PROGRAM

```
 %%
^[a - z A - Z _][a - z A - Z 0 - 9 _] * printf("Valid Identifier");
// regex for invalid identifiers
^[^a - z A - Z _] printf("Invalid Identifier");
.;
%%
void main() {
printf("\nEnter Identifier: "); yylex();
printf("\n");
}
int yywrap()
{
}
```

OUTPUT

```
        Student1
        Valid Identifier
```

4. Valid date (dd/mm/yyyy)

PROGRAM

```
%{
#include<stdio.h> int i=0,yr=0,valid=0;
%}
%%
([0-2][0-9]|[3][0-1])\/((0(1|3|5|7|8))|(10|12))\/([1-2][0-9][0-9][-0-9]) {valid=1;}

([0-2][0-9]|30)\/((0(4|6|9))|11)\/([1-2][0-9][0-9][0-9]) {valid=1;}

([0-1][0-9]|2[0-8])\/02\/([1-2][0-9][0-9][0-9]) {valid=1;}

29\/02\/([1-2][0-9][0-9][0-9]) { while(yytext[i]!='/')i++;
i++;while(yytext[i]!='/')i++;i++;while(i<yyleng)yr=(10*yr)+(yytext[i++]-'0');
if(yr%4==0||(yr%100==0&&yr%400!=0))valid=1;}

%%
void main()
{
yyin=fopen("new","r"); yylex();
if(valid==1) printf("It is a valid date\n"); else printf("It is not a valid date\n");
}
int yywrap()
{
return 1;
}
```

OUTPUT

```
        03/06/1994
        It is a Valid date
```

5. Valid time (hh:mm:ss)

PROGRAM

```
%{
#include<stdio.h> int i=0,yr=0,valid=0;
%}
%%
([0-2][0-9]:[0-6][0-9]\:[0-6][0-9]) {printf("%s It is a valid time\n",yytext);}


%%
void main()
{
yyin=fopen("new","r"); yylex();
}
int yywrap()
{
return 1;
}
```

OUTPUT

```
01:30:05
        It is a valid time
```

EXPERIMENT-8

OBJECTIVE

Write a lex program to count blank spaces,words,lines in a given file.

PROGRAM

```
%{
#include<stdio.h>
int lines=0, words=0,s_letters=0,c_letters=0, num=0, spl_char=0,total=0;
%}
%%
\n { lines++; words++;} [\t ' '] words++;
[A-Z] c_letters++; [a-z] s_letters++; [0-9] num++;
. spl_char++;
%%
void main(void)
{
FILE *fp; char f[50];
printf("enterfile name \n"); scanf("%s",f);
yyin= fopen(f,"r"); yylex();
total=s_letters+c_letters+num+spl_char; printf(" This File contains ..."); printf("\n\t%d lines",
lines); printf("\n\t%d words",words); printf("\n\t%d small letters", s_letters); printf("\n\t%d
capital letters",c_letters); printf("\n\t%d digits", num);
printf("\n\t%d special characters",spl_char); printf("\n\tIn total %d characters.\n",total);
}
int yywrap()
{
return(1);
}
```

OUTPUT

```
 I am a V  Sem student.
Line=1
Words=6
Space=5
```

EXPERIMENT-9

OBJECTIVE

Write a lex program to count the no. of vowels and consonants in a C file.

PROGRAM

```
 %{

#include<stdio.h>
int vcount=0,ccount=0;
%}
%%
[a|i|e|o|u|E|A|I|O|U] {vcount++;}
[a-z A-Z (^a|i|e|o|u|E|A|I|O|U) ] {ccount++;}
%%
int main()
{
FILE *fp; char f[50];
printf("enterfile name \n"); scanf("%s",f);
yyin= fopen(f,"r"); yylex();
printf("No. of Vowels :%d\n",vcount); printf("No. of Consonants :%d\n",ccount);
return 0;
}
int yywrap()
{
}
```

OUTPUT

```
NEHA
 No. of Vowels=2
 No. of Consonants=2
```

EXPERIMENT-10

OBJECTIVE

Write a YACC program to recognize strings aaab,abbb using a^nb^n, where b>=0.

PROGRAM

```
 Gm.l
%{
#include "y.tab.h"
%}
%%
"a"|"A" {return A;}
"b"|"B" {return B;}
[ \t] {;}
\n {return 0;}
. {return yytext[0];}
%%
int yywrap()
{
return 1;
}
Gm.y
%{
#include<stdio.h>
%}
%token A B
%%
stmt: S
;
S: A S B
|
;
%%
void main()
{
printf("enter \n"); yyparse(); printf("valid");

exit(0);
}
void yyerror()
{
printf("invalid "); exit(0);
}
```

OUTPUT

```
aabb
Valid
```

EXPERIMENT-11

OBJECTIVE

Write a YACC program to evaluate an arithmetic expression involving operators +,-,* and /.

PROGRAM

```
Expr.l
%{
#include "y.tab.h" extern int yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext); return number;}
[\t] {;}
[\n] {return 0;}
. {return yytext[0];}
%%
int yywrap()
{
return 1;
}
Expr.y
%{
#include<stdio.h> int res=0;
%}
%token number
%left '+' '-'
%left '*' '/'
%%
stmt:expr {res=$$;}
;
expr:expr '+' expr {$$=$1+$3;}
|expr '-' expr {$$=$1-$3;}
|expr '*' expr {$$=$1*$3;}
|expr '/' expr {if($3==0) exit(0);
else $$=$1/$3;}
 |number
;%%
void main()
{
printf(" enter expr\n"); yyparse(); printf("valid=%d",res); exit(0);
}
void yyerror()
{
printf("invalid\n"); exit(0);}
```

OUTPUT

```
2+3
5
```

EXPERIMENT-12

OBJECTIVE

Write a YACC program to check validity of a strings abcd,aabbcd using grammar a^nb^nc^md^m, where n , m>0

PROGRAM

```
Grammer.y
%{
#include<stdio.h> #include<stdlib.h> int yyerror(char*); int yylex();

%}
%token A B C D NEWLINE
%%
stmt: S NEWLINE { printf("valid\n"); return 1;
}
;
S: X Y
;

X: A X B
|
;
Y: C Y D
|
;
%%
extern FILE *yyin; void main()
{
printf("enter \n"); do
{
yyparse();
}
while(!feof(yyin));

}
int yyerror(char* str)
{

printf("invalid "); return 1;
}
Grammer.l

%{
#include"y.tab.h"
%}
%%
a |
A {return A;} c |
C {return C;} b |
B {return B;} d |
D {return D;} [ \t] {;}
"\n" {return NEWLINE;}
```

```
. {return yytext[0];}
%%
int yywrap()
{
return 1;
}
```

OUTPUT

```
aabbcccddd
Valid
```

EXPERIMENT-13

OBJECTIVE

Write a C program to find first of any grammar

PROGRAM

```
#include<stdio.h>

#include<ctype.h> void FIRST(char ); int count,n=0;
char prodn[10][10], first[10];

void main()
{
int i,choice; char c,ch;
printf("How many productions ? :"); scanf("%d",&count);
printf("Enter %d productions epsilon= $ :\n\n",count); for(i=0;i<count;i++)
scanf("%s%c",prodn[i],&ch); do
{ n=0;
printf("Element :");
scanf("%c",&c); FIRST(c);
printf("\n FIRST(%c)= { ",c); for(i=0;i<n;i++)
printf("%c ",first[i]);
printf("}\n");

printf("press 1 to continue : "); scanf("%d%c",&choice,&ch);
}
while(choice==1);
}



void FIRST(char c)
{
int j;
if(!(isupper(c)))first[n++]=c; for(j=0;j<count;j++)
{
if(prodn[j][0]==c)
{
if(prodn[j][2]=='$') first[n++]='$';
else if(islower(prodn[j][2]))first[n++]=prodn[j][2]; else FIRST(prodn[j][2]);
}
}
}
```

OUTPUT

```
S-> {abb}
First(s)= {a}
```

BEYOND THE SYLLABUS EXPERIMENT-1

OBJECTIVE

Write a C program to find Follow of any grammar

PROGRAM

```
#include<stdio.h>
#include<string.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);
int main()
{
 int i,z;
 char c,ch;
 printf("Enter the no.of productions:");
 scanf("%d",&n);
 printf("Enter the productions(epsilon=$):\n");
 for(i=0;i<n;i++)
  scanf("%s%c",a[i],&ch);

 do
 {
  m=0;
  printf("Enter the element whose FOLLOW is to be found:");

  scanf("%c",&c);
  follow(c);
  printf("FOLLOW(%c) = { ",c);
  for(i=0;i<m;i++)
   printf("%c ",f[i]);
  printf(" }\n");
  printf("Do you want to continue(0/1)?");
  scanf("%d%c",&z,&ch);
 }
 while(z==1);
}
void follow(char c)
{

 if(a[0][0]==c)f[m++]='$';
 for(i=0;i<n;i++)
 {
  for(j=2;j<strlen(a[i]);j++)
  {
   if(a[i][j]==c)
   {
    if(a[i][j+1]!='\0')first(a[i][j+1]);

    if(a[i][j+1]=='\0'&&c!=a[i][0])
     follow(a[i][0]);
   }
  }
 }
}
void first(char c)
{
    int k;
            if(!(isupper(c)))f[m++]=c;
            for(k=0;k<n;k++)
            {
            if(a[k][0]==c)
            {
            if(a[k][2]=='$') follow(a[i][0]);
            else if(islower(a[k][2]))f[m++]=a[k][2];
            else first(a[k][2]);
            }
            }

}
```

OUTPUT

```
productions:
E=TD
D=+TD
D=$
T=FS
S=*FS
S=$
F=(E)
F=a

FOLLOW(E)=FOLLOW(D)={),$}
FOLLOW(T)=FOLLOW(S)={+,),$}
FOLLOW(F)={+,*,),$}
```

BEYOND THE SYLLABUS EXPERIMENT-2

OBJECTIVE

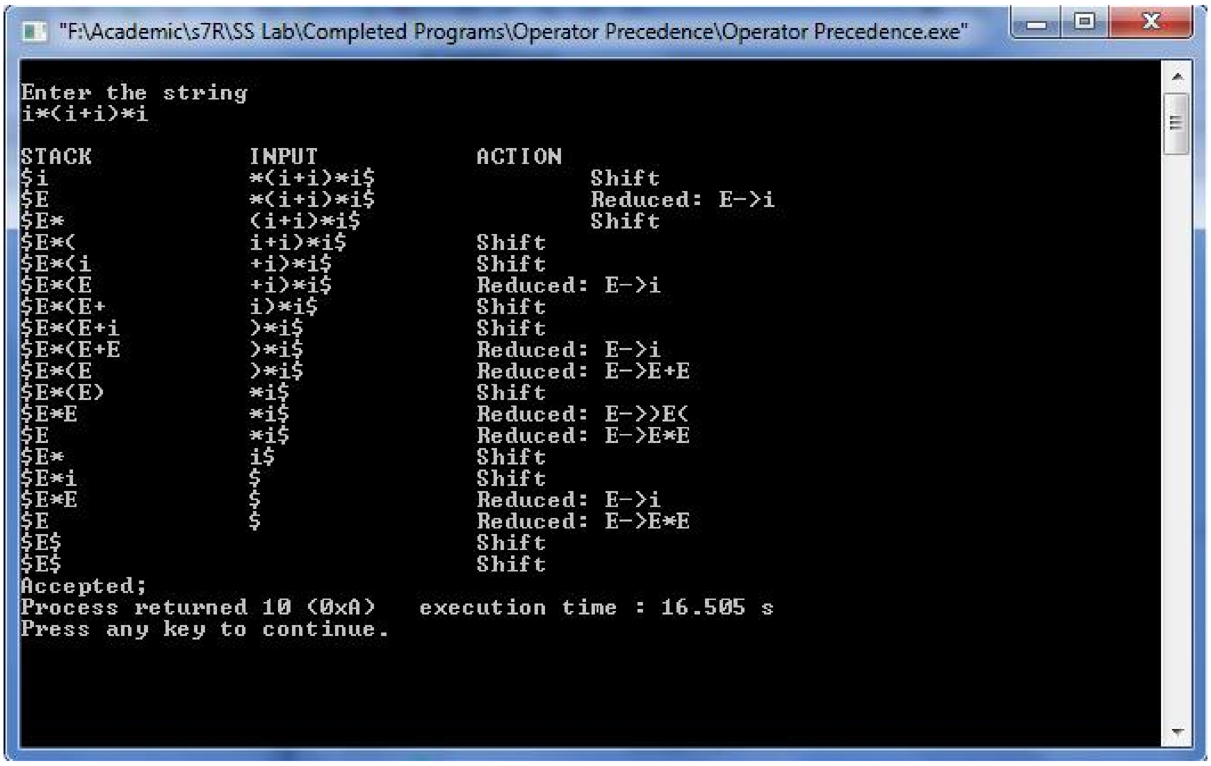
Write a C program for operator precedence parsing

PROGRAM

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

char *input;
int i=0;
char lasthandle[6],stack[50],handles[][5]={")E(","E*E","E+E","i","E^E"};
//(E) becomes )E( when pushed to stack

int top=0,l;
char prec[9][9]={

                    /*input*/

        /*stack   +   -   *   /   ^   i   (   )   $  */

        /*  + */  '>', '>','<','<','<','<','<','>','>',

        /*  - */  '>', '>','<','<','<','<','<','>','>',

        /*  * */  '>', '>','>','>','<','<','<','>','>',

        /*  / */  '>', '>','>','>','<','<','<','>','>',

        /*  ^ */  '>', '>','>','>','<','<','<','>','>',

        /*  i */  '>', '>','>','>','>','e','e','>','>',

        /*  ( */  '<', '<','<','<','<','<','<','>','e',

        /*  ) */  '>', '>','>','>','>','e','e','>','>',

        /*  $ */  '<', '<','<','<','<','<','<','<','>',

           };

int getindex(char c)
{
switch(c)
   {
   case '+':return 0;
   case '-':return 1;
   case '*':return 2;
   case '/':return 3;
   case '^':return 4;
   case 'i':return 5;
   case '(':return 6;
    case ')':return 7;
    case '$':return 8;
    }
}


int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}


int reduce()
{
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
    {
    len=strlen(handles[i]);
    if(stack[top]==handles[i][0]&&top+1>=len)
        {
        found=1;
        for(t=0;t<len;t++)
            {
            if(stack[top-t]!=handles[i][t])
                {
                found=0;
                break;
                }
            }
        if(found==1)
            {
            stack[top-t+1]='E';
            top=top-t+1;
            strcpy(lasthandle,handles[i]);
            stack[top+1]='\0';
            return 1;//successful reduction
            }
        }
    }
return 0;
}



void dispstack()
{
int j;
for(j=0;j<=top;j++)
    printf("%c",stack[j]);
```

```
}



void dispinput()
{
int j;
for(j=i;j<l;j++)
   printf("%c",*(input+j));
}



void main()
{
int j;

input=(char*)malloc(50*sizeof(char));
printf("\nEnter the string\n");
scanf("%s",input);
input=strcat(input,"$");
l=strlen(input);
strcpy(stack,"$");
printf("\nSTACK\tINPUT\tACTION");
while(i<=l)
            {
            shift();
            printf("\n");
            dispstack();
            printf("\t");
            dispinput();
            printf("\tShift");
            if(prec[getindex(stack[top])][getindex(input[i])]=='>')
                        {
                        while(reduce())
                                    {
                                    printf("\n");
                                    dispstack();
                                    printf("\t");
                                    dispinput();
                                    printf("\tReduced: E->%s",lasthandle);
                                    }
                        }
            }

if(strcmp(stack,"$E$")==0)
   printf("\nAccepted;");
else
   printf("\nNot Accepted;");
}
```

OUTPUT:



"F:\Academic\s7R\SS Lab\Completed Programs\Operator Precedence\Operator Precedence.exe"

```
Enter the string
i*(i+i)*i

STACK           INPUT           ACTION
$i              *(i+i)*i$               Shift
$E              *(i+i)*i$               Reduced: E->i
$E*             (i+i)*i$                Shift
$E*(            i+i)*i$         Shift
$E*(i           +i)*i$          Shift
$E*(E           +i)*i$          Reduced: E->i
$E*(E+          i)*i$           Shift
$E*(E+i         )*i$            Shift
$E*(E+E         )*i$            Reduced: E->i
$E*(E           )*i$            Reduced: E->E+E
$E*(E)          *i$             Shift
$E*E            *i$             Reduced: E->)E(
$E              *i$             Reduced: E->E*E
$E*             i$              Shift
$E*i            $               Shift
$E*E            $               Reduced: E->i
$E              $               Reduced: E->E*E
$E$                             Shift
$E$                             Shift
Accepted;
Process returned 10 (0xA)   execution time : 16.505 s
Press any key to continue.
```