SSCD-Super Important questions-18CS61

By the TIE review team DSATM, BNMIT and SVIT

Answered by: Someone

Module-1

1. With suitable examples, explain the SIC/XE Machine Architecture and its I/O operation
2. What is program relocation? Explain the problem associated with it and the solutions.
3. Write the algorithm for a pass-1 of two pass assembler and pass 2 of two pass assembler - (Practice anyone)
4. Explain the following: Literals, Symbol defining statements, Expressions
5. Apply the concept of the control section to a suitable code snippet and explain how the control section helps us achieve independent assembling of the source program
6. Explain the following -15M
(a) Record operations-3M (b) Format of Header,text,end record-3M (c) Program blocks -3M (d) defn of SYMTAB,LOCCTR,OPTAB-3M (e) Diff between one pass and multipass assembler-3M

Self-solve to be done for module 1

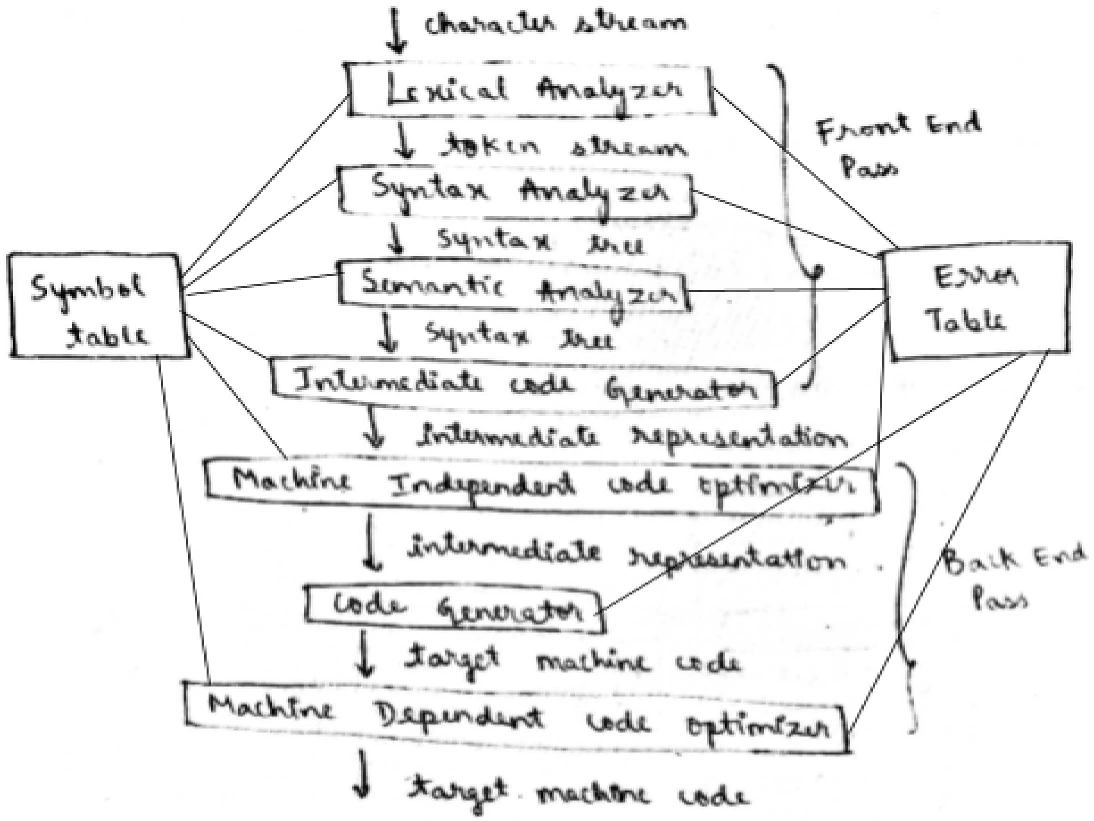
Module-2

7. Illustrate the structure of a compiler with a neat diagram, Show the output of each phase of the compiler for the assignment statement:

Profit= Sellingprice-costprice*90

Solve it.

Structure of a compiler ·



8. Explain the various applications of compiler technology

Application of compiler Technology

① Implementation of high level prog. lang using modern OOPs concept like,

→ Data Abstraction

→ Inheritance properties

② optimization for computer architectures

→ Parallelism

(i) at instruction level – multiple operations executed together

(ii) at preprocessor level – different threads run seperately

→ Memory Hierarchy

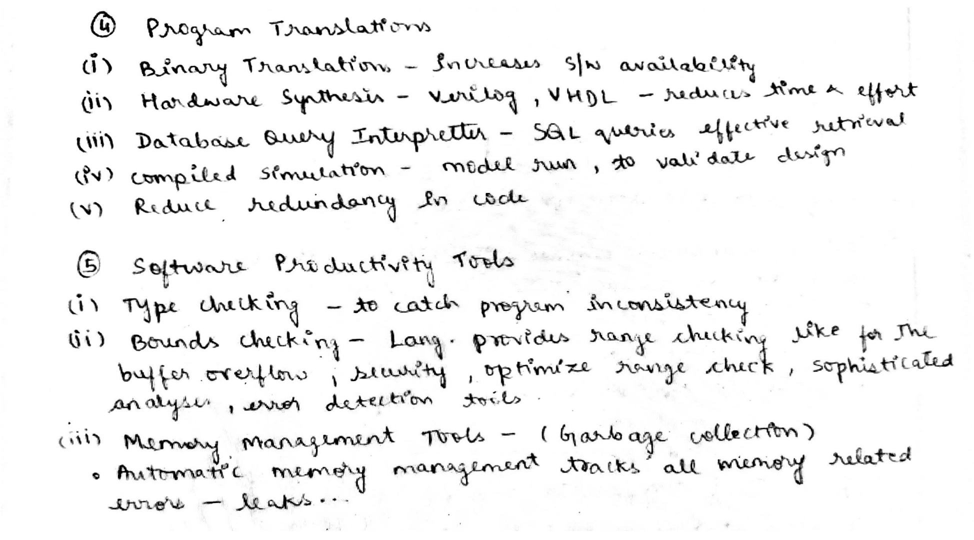
Building very Large or Fast storage, but not both

③ Design New computer Architectures

→ RISC – reduces complex memory addressing, support data structure access, procedure invocation ....

→ Specialized Architectures –

Data flow M/C, vector M/C, VLIW & SIMD M/C.



④ Program Translations
(i) Binary Translation – Increases S/W availability
(ii) Hardware Synthesis – verilog, VHDL – reduces time & effort
(iii) Database Query Interpretter – SQL queries effective retrieval
(iv) compiled simulation – model run, to validate design
(v) Reduce redundancy in code

⑤ Software Productivity Tools
(i) Type checking – to catch program inconsistency
(ii) Bounds checking – Lang. provides range checking like for the buffer overflow, security, optimize range check, sophisticated analyses, error detection tools.
(iii) Memory Management Tools – (Garbage collection)
• Automatic memory management tracks all memory related errors – leaks...

9. Apply input buffering strategy to a suitable example and explain the same, write an algorithm for lookahead code with sentinels

Input buffering is very essential for the following reasons:

Since lexical analyzer is the first phase of the compiler, it is the only phase of the compiler that reads the source program character-by-character and consumes considerable time in reading the source program. Thus, the speed of lexical analysis is a concern while designing the compiler.

Lexical analyzers may have to look one or more characters beyond the next lexeme before we have the right lexeme.

For this reason, we use the concept of input buffering where a block of 1024 or 4096 or more characters are read in one memory read operation and stored in the array to speed up the process.

Definition: The method of reading a block of characters (1K or 4K or more bytes) from the disk in one read operation and storing in memory (normally in the form of an array) for further processing and faster accessing is called input buffering.

→ Input Buffering : To speed up reading of src prog.

① Single Buffer / 1-Buffer Technique

We use only one single buffer to store processed character from large no. of characters from source prog.

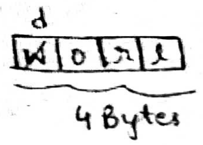
Main overhead is that if,

$\boxed{\text{lexeme size} > \text{Buffer size}}$

we loose the lexeme

World

5 Bytes



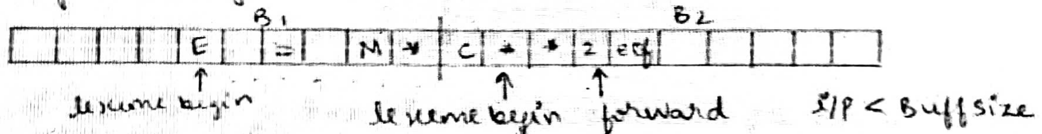
· It reloads data, removes old data.

② 2-Buffer Technique < without sentinel / with sentinel

We use two buffers that are alternately reloaded,

Each buffer of same size N, N = size of a disk block. (4096 Byte)

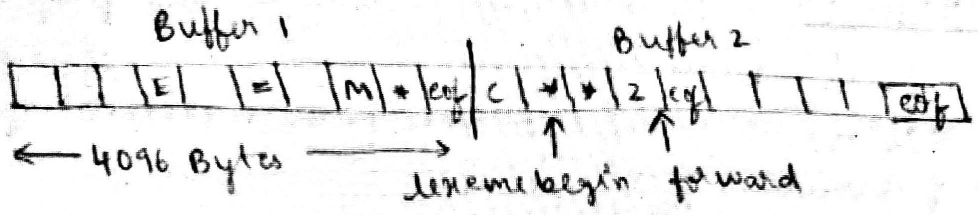
· Using read system call, N characters are read.



→ Special char eof marks end of src file and this char is different from any other char of src prog.

→ Sentinels : (2 Buffer technique with Sentinels)

Using sentinel character at the end which is a special char that is not part of src prog (usually eof)



Here check if reached end of Buffer or not

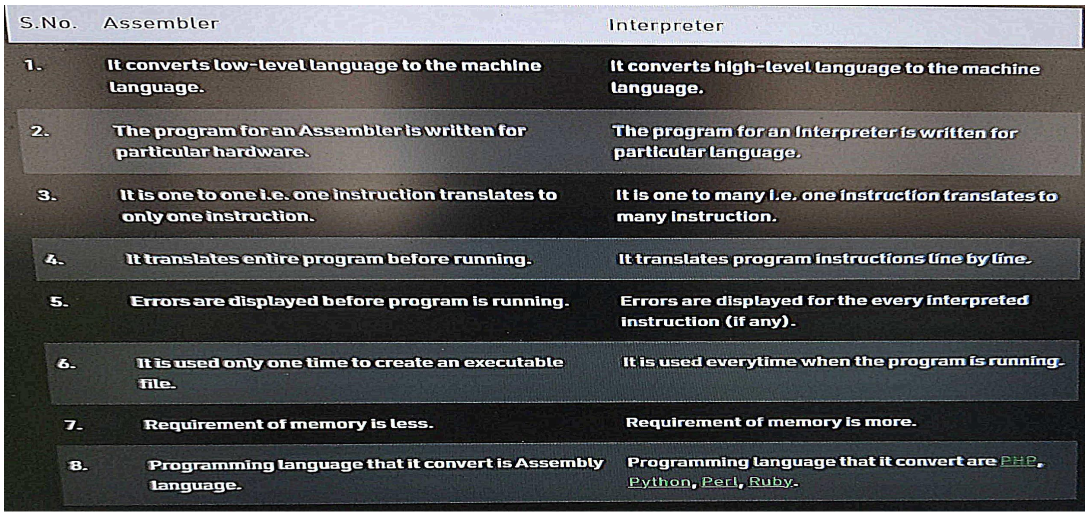
Look Ahead is atmost 1 char, make previous char as returned valid token.

The algorithm consisting of lookahead code with sentinels is shown below:

```
switch (*inputPointer++)
{
case eof: If (inputPointer is at end of first buffer)
{
reload the second buffer;
inputPointer = beginning of second buffer;
break;
}
E = M *
eof C * * 2 eof
lexemeBeginning
Buffer 1 Buffer 2
Input pointer
if (inputPointer is at the end of second buffer)
{
reload the first buffer;
inputPointer = beginning of first buffer;
break;
}
/* eof within a buffer indicates the end of the input */
/* So, terminate lexical analysis */
break;
/* Cases for other characters */
}
```

10. Explain the following -3M each

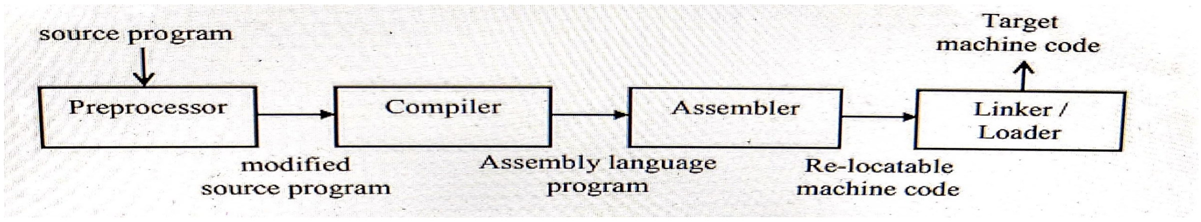
(i)Interpreter v/s assembler



S.No.	Assembler	Interpreter
1.	It converts low-level language to the machine language.	It converts high-level language to the machine language.
2.	The program for an Assembler is written for particular hardware.	The program for an Interpreter is written for particular language.
3.	It is one to one i.e. one instruction translates to only one instruction.	It is one to many i.e. one instruction translates to many instruction.
4.	It translates entire program before running.	It translates program instructions line by line.
5.	Errors are displayed before program is running.	Errors are displayed for the every interpreted instruction (if any).
6.	It is used only one time to create an executable file.	It is used everytime when the program is running.
7.	Requirement of memory is less.	Requirement of memory is more.
8.	Programming language that it convert is Assembly language.	Programming language that it convert are PHP, Python, Perl, Ruby.

(ii)Language processing System

Now, let us see "What are the cousins of the compiler?" or let us "Explain language processing system" During software development, in addition to a compiler several other programs may be required to create an executable target program. All the programs that are helpful in generating the required executable target program are called cousins of the compiler. All these cousins of the compiler are together represent language processing system. The language processing system or the various cousins of the compiler and their interaction are shown in block diagram below:



Thus, the various programs that constitute language processing system (or cousins of the compiler) are:

Preprocessor: The preprocessor is executed before the actual compilation of code begins. A source program may be divided into modules and can be stored in separate files. The preprocessor will collect all source files and may expand macros into source language statements. Thus the output of the preprocessor is also a source program but expanded and modified. This expanded source program is input to the compiler. The main activities performed by the preprocessor are:

1) Macro processing which is identified using #define directive

2) File inclusion which is identified using #include directive

3) It also helps in conditional compilation with the help of the directives such as #if, #else etc.

Compiler: The compiler is a translator that accepts the expanded and modified source program as the input from preprocessor and converts it into assembly language program. If any errors are present, they are displayed along with appropriate error messages and line numbers so that the user can correct the program. The various activities performed by the compiler are:

1) The syntax errors are identified and appropriate error messages are displayed along with line numbers

2) An optional list file can be generated by the compiler

3) The compiler replaces each executable statement in high level language into one or more machine language instructions

4) Converts HLL into assembly language or object program if the program is syntactically correct.

Assembler: An assembler is a translator which converts assembly language program into equivalent machine code. The machine code thus produced is written into a file called object program. The various functions of the assembler are shown below:

1) Convert assembly language instructions to their machine language equivalent

2) Convert the data constants into equivalent machine representation. For example, if decimal data 4095 is used as operand, it is converted into hexadecimal value (machine representation) FFF.

3) Build the machine instruction using the appropriate instruction format.

4) Process the instructions given to the assembler directives. These directives help the assembler while converting from assembly language to machine language

5) Create the list file. The list file consists of source code along with address for each instruction and corresponding object code.

6) Display appropriate errors on the screen so that the programmer can correct the program and re-assemble it.

7) Create the object program. The object program consists of machine instructions and instructions to the loader.

Linker: Large programs are often compiled in pieces and generate object programs and stored in files. The linker or linkage editor accepts one or more object programs generated by a compiler and links the library files and creates an executable file. The linker resolves external reference symbols where code in one file may refer to a location in another file.

Loader: A loader is the part of an operating system that is responsible for loading programs into main memory for execution. This is one of the important stages in the process of executing a program. Loading a program involves reading the contents of executable file, loading the file containing the program text into memory, and then carrying out other tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code and execution begins.

(iii)Phase v/s pass

First, let us see “What is a phase?”

Definition: Since compiler is a very complex program, to understand the design of the compiler, it is logically divided into various sub-programs called modules. These sub- programs or modules are also called phases. Each phase or module accepts one representation of the program as the input and generates another representation of the program as the output. Thus, a phase is nothing but a module of the compiler that accepts one representation of the program as the input and produces another representation of the program as the output.

Now, let us see “What is a pass?”

Definition: A group of one or more phases of a compiler that perform analysis or synthesis of the source program is called a pass of the compiler. Even though the compiler is divided into various phases, in the actual implementation of the compiler, one or more phase may be grouped together into a pass. Each pass reads an input file and writes an output file. The compilers based on the number of passes are classified as shown below:

One pass compiler

Two pass compiler

Three pass compiler

Multipass compiler

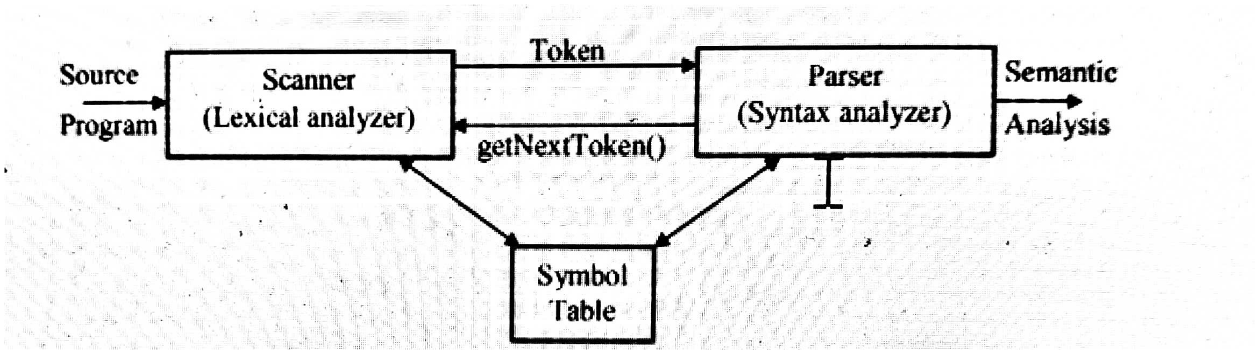
(iv)Lexical analysis (role)

Now, let us see "What is the role of lexical analyzer?" The lexical analyzer is the first phase of the compiler. The various tasks that are performed by the lexical analyzer are:

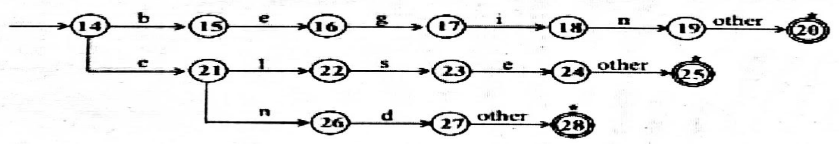
Read a sequence of characters from the source program and produce the tokens.

The tokens thus generated are sent to the parser for syntax analysis. The parser is also called syntax analyzer. During this process, lexical analyzer interacts with symbol table to insert various identifiers and constants. Sometimes, information of identifier is read from symbol table to assist in determining the proper token to send to the parser.

The interaction between the lexical analyzer and the parser is pictorially represented as shown below:



11. Design a lexical analyzer to identify the keywords begin, else, and end. Sketch the program segment to implement it showing the first two states and one final state



Similar to the previous two problems, we can write various cases to identify the tokens BEGIN, ELSE and END. The lexical analyzer is implemented by writing the code for first two states and one final state as shown below:

```
        state = 0;
        for (;;)
        {
                switch (state)
                {
                        /* Case 0-13: Identify other tokens */
                        case 14:
                                ch = getchar()
                                if (ch == 'b')
                                        state = 15;
                                else if (ch == 'e')
                                        state = 21;
                                else
                                        state = 29;        /* other token */
                        case 15:
                                ch = getchar();
                                if (ch == 'e')
                                        state = 16;
                                else
                                        state = 29;        /* other token */
```

```
                                break;
                        .......
                        .......
                        case 28: retract();
                                return END;
                }
        }
```

12. What is the meaning of patterns, lexemes and tokens? Identify lexemes and tokens in the following statement: a= b*d;

lexemes are input to lexical analyzer and lexical analyzer produces tokens. The description of a lexeme is a pattern.

A token is a pair consisting of token name and an optional attribute value. The token names are basically integer codes represented using symbolic names written in capital letters such as INT, FLOAT, SEMI_COLON etc. The attribute values are optional and will not be present for keywords, operators and symbols. The attribute values are present for all identifiers and constants.

A sequence of characters in the source program that matches the patterns such as identifiers, numbers, relational operators, arithmetic operators, symbols such as #, [,], (,) and so on are called lexemes. In other words, a lexeme is a string of patterns read from the source file that corresponds to a token.

The description of a lexeme is called pattern. More formally, a pattern is described as a rule describing set of lexemes.

a= b*d;

The lexemes, patterns and tokens for the above expression are shown below:

a is a lexeme matching the pattern identifier and returning the token <ID, 1> where
ID is the token name and 1 is the position of identifier a in the symbol table

= is a lexeme matching the pattern symbol „=" and returning the token ASSIGN

b is a lexeme matching the pattern identifier and returning the token <ID, 2> where
ID is the token name and 2 is the position of identifier b in the symbol table

* is a lexeme matching the pattern symbol „*" and returning the token STAR

d is a lexeme matching the pattern identifier and returning the token <ID, 3> where
ID is the token name and 3 is the position of identifier d in the symbol table

; is a lexeme matching the pattern semicolon and returning the token SEMICOLON

Module-3

13. Build an algorithm to eliminate left recursion. Eliminate left recursion from the following grammar

(a)E → E +T | T

T → T * F | F

F→ (E) | id Solve it.

(b) S → Aa | b

A → Ac | Sd | ε Solve it.

```
Input : Grammar G without ε-productions and with no cycles
Output: Grammar without left recursion. It may have ε-productions
Arrange the non-terminals in the order A1 , A2 , A3,........An
S → Aa | b
A → bd A'| A'
A'→ cA'| adA'|ε
for i = 1 to n do
    for j = 1 to i-1 do
            Let Aj → β1 | β2 | β3 |..... Bk
            Replace Ai → Ajα by Ai → β1α | β2α | β3α |..... Bkα
    end for
Eliminate immediate left recursion among Ai productions
end for
```

14. What are the different sentential forms? What is the left sentential form? What is the right sentential form?

Let G = (V, T, P, S) be a CFG. Any string w (V T)* which is derivable from the start symbol S such that S w is called a sentence or sentential form of G.

The two sentential forms are:

Left sentential form

Right sentential form

Left sentential form : If there is a derivation of the form S α, where at each step in the derivation process only a left most variable is replaced, then α is called left-sentential form of G.

Right sentential form : If there is a derivation of the form S α, where at each step in the derivation process only a right most non-terminal is replaced, then α is called right-sentential form of G.

Refer Lmd and Rmd for example.

15. Explain the process of identifying ambiguous grammar, check whether the grammar below is ambiguous and if yes eliminate the ambiguity. S → iCtS | iCtSeS | a
C → b

Let G = (V, T, P, S) be a context free grammar. A grammar G is ambiguous if and only if there exists at least one string w T* for which two or more left derivations exist or two or more right derivations exist. That is, the ambiguous grammar has two or more meanings or interpretations.Since, for every derivation a parse tree exist, the ambiguous grammar can also be defined as the one which has two or more different parse trees for the string w derived from start symbol S.

Solve it.

16. Given the following grammar:
Z → d | XYZ
Y→ ε |c
X → Y | a
a. Compute FIRST and FOLLOW sets for each non-terminal
b. Without constructing the parsing table, check whether the grammar is LL(1).
c. By constructing the parsing table, check whether the grammar is LL(1).

Solve it.

17. Consider the following grammar:

$$\begin{array}{lll} E & \rightarrow & TE^{1} \\ E^{1} & \rightarrow & +TE^{1} \mid \in \\ T & \rightarrow & FT^{1} \\ T^{1} & \rightarrow & *FT^{1} \mid \in \\ F & \rightarrow & (E) \mid id \end{array}$$

a) Identify FIRST and FOLLOW
b) Construct the predictive parsing table
c) Show the sequence of moves made by the parser for the string id+id*id
d) Add the synchronizing tokens for the above parsing table and show the sequence of moves made by the parser for the string “) id * + id”

Solve it.

18. Consider S -> SS+ |SS* |a, and input string aa+a*, and do the following
a)Identify LMD and RMD
b)Check whether the grammar is ambiguous
(iii)Do Left factoring and solve for left recursion if any

Solve it.

19. Given the following grammar:
S → a | (L)
L → L, S | S
a. Is the grammar suitable for predictive parser?
b. Do the necessary changes to make it suitable for LL(1) parser c.
Compute FIRST and FOLLOW sets for each non-terminal
d. Obtain the parsing table and check whether the resulting grammar is LL(1) or not.
e. Show the moves made by the predictive parser on the input “(a, (a , a))”

Solve it

Module-4

20. (a) Write a LEX program to count the number of scanf and printf statements and replace them with readf and writef respectively.
(b)Write a LEX program to identify vowels and constants

Solve it.

21. What is ambiguous grammar? Explain the problem of arithmetic ambiguity by considering the expression 2+3*4, Make use of precedence and associativity rules to resolve the ambiguity

Solve it.

Definition: Let G = (V, T, P, S) be a context free grammar. A grammar G is ambiguous if and only if there exists at least one string w T* for which two or more left derivations exist or two or more right derivations exist. That is, the ambiguous grammar has two or more meanings or interpretations. Since, for every derivation a parse tree exist, the ambiguous grammar can also be defined as the one which has two or more different parse trees for the string w derived from start symbol S.

22. Explain/Write the following wrt YAAC-12M

(i)Types of conflicts with ex

If your grammar has shift-reduce or reduce-reduce conflicts, there is also a table of conflicts in the statistics section of the parser description. For example, if you change the rules section of the sample grammar to:

```
stmt : IF stmt ELSE stmt
     | IF stmt
     | stmt stmt
     | A ;
```

you get the following conflict report:

```
Conflicts:
  State  Token    Action
    5  IF       shift 2
    5  IF       reduce (3)
    5  A        shift 1
    5  A        reduce (3)
```

This shows that state 5 has two shift-reduce conflicts. If the parser is in state 5 and encounters an **IF** token, it can shift to state 2 or reduce using rule 3. If the parser encounters an **A** token, it can shift to state 1 or reduce using rule 3. This is summarized in the final statistics with the line:

```
2 shift-reduce conflicts
```

Reading the conflict report shows you what action the parser takes in case of a conflict: The parser always takes the *first* action shown in the report. This action is chosen in accordance with the two rules for removing ambiguities.

(ii)Errors handled by YACC

When the parser reads an input stream, that input stream might not match the rules in the grammar file. The parser detects the problem as early as possible. If there is an error-handling subroutine in the grammar file, the parser can allow for entering the data again, ignoring the bad data, or initiating a cleanup and recovery action. When the parser finds an error, for example, it may need to reclaim parse tree storage, delete or alter symbol table entries, and set switches to avoid generating further output. When an error occurs, the parser stops unless you provide error-handling subroutines. To continue processing the input to find more errors, restart the parser at a point in the input stream where the parser can try to recognize more input. One way to restart the parser when an error occurs is to discard some of the tokens following the error. Then try to restart the parser at that point in the input stream. The yacc command uses a special token name, error, for error handling. Put this token in the rules file at places that an input error might occur so that you can provide a recovery subroutine. If an input error occurs in this position, the parser executes the action for the error token, rather than the normal action. The following macros can be placed in yacc actions to assist in error handling:

Macros	Description
YYERROR	Causes the parser to initiate error handling
YYABORT	Causes the parser to return with a value of 1
YYACCEPT	Causes the parser to return with a value of 0
YYRECOVERING()	Returns a value of 1 if a syntax error has been detected and the parser has not yet fully recovered

To prevent a single error from producing many error messages, the parser remains in error state until it processes three tokens following an error. If another error occurs while the parser is in the error state, the parser discards the input token and does not produce a message.

(iii) A YACC program to check whether the given arithmetic expression is valid or not. (6M)

Solve it.

23. Explain the metacharacters used in a regular expression with an example.

Metacharacters are the building blocks of regular expressions. Characters in RegEx are understood to be either a metacharacter with a special meaning or a regular character with a literal meaning.
The following are some common RegEx metacharacters and examples of what they would match or not match in RegEx.

Metacharacter	Description	Examples
\d	Whole Number 0 - 9	\d\d\d = 327 \d\d = 81 \d = 4 -- \d\d\d ≠ 24631 \d\d\d doesn't return 24631 because 24631 contains 5 digits. \d\d\d only matches for a 3-digit string.
\w	Alphanumeric Character	\w\w\w = dog \w\w\w\w = mule \w\w = to -- \w\w\w = 467 \w\w\w\w = 4673 -- \w\w\w ≠ boat \w\w\w doesn't return boat because boat contains 4 characters. -- \w ≠ ! \w doesn't return the exclamation point ! because it is a non-alphanumeric character.

\W	Symbols	\W = % \W = # \W\W\W = @#% -- \W\W\W\W ≠ dog8 \W\W\W\W doesn't return dog8 because d, o, g, and 8 are alphanumeric characters.
[a-z] [0-9]	Character set, at least one of which must be a match, but no more than one unless otherwise specified. The order of the characters does not matter.	pand[ora] = panda pand[ora] = pando -- pand[ora] ≠ pandora pand[ora] doesn't bring back pandora because it is implied in pand[ora] that only 1 character in [ora] can return. (Quantifiers that allow pand[ora] to match for pandora is discussed below.)
(abc) (123)	Character group, matches the characters abc or 123 in that exact order.	pand(ora) = pandora pand(123) = pand123 -- pand(oar) ≠ pandora pand(oar) does not match for pandora because it's looking for the exact phrase pandoar.
|	Alternation - allows for alternate matches. | operates like the Boolean OR.	pand(abc|123) = pandora OR pand123
?	Question mark matches when the character preceding ? occurs 0 or 1 time only, making the character match optional.	colou?r = colour (u is found 1 time) colou?r = color (u is found 0 times)

*	Asterisk matches when the character preceding * matches 0 or more times. Note: * in RegEx is different from * in dtSearch. RegEx * is asking to find where the character (or grouping) preceding * is found ZERO or more times. dtSearch * is asking to find where the string of characters preceding * or following * is found 1 or more times.	tre*= tree (e is found 2 times) tre* = tre (e is found 1 time) tre* = tr (e is found 0 times) -- tre* ≠ trees tre* doesn't match the term trees because although "e" is found 2 times, it is followed by "s" , which is not accounted for in the RegEx.
+	Plus sign matches when the character preceding + matches 1 or more times. The + sign makes the character match mandatory.	tre+ = tree (e is found 2 times) tre+ = tre (e is found 1 time) -- tre+ ≠ tr (e is found 0 times) tre+ doesn't match for tr because e is found zero times in tr.

. (period)	The period matches any alphanumeric character or symbol.	ton. = tone
		ton. = ton#
		ton. = ton4
		-- 	
		ton. ≠ tones
		ton. doesn't match for the term tones because . by itself will only match for a single character, here, in the 4th position of the term.Â In tones, s is the 5th character and is not accounted for in the RegEx.

.*	Combine the metacharacters . and *, in that order .* to match for any character 0 or more times. NOTE: .* in RegEx is equivalent to dtSearch wildcard * operator.	tr.* = tr tr.* = tre tr.* = tree tr.* = trees tr.* = trough tr.* = treadmill

Module-5

24.Obtain SDD for the following grammar using a top-down approach:

S → En
E → E + T | T
T → T * F | F
F → (E) | digit
and Obtain annotated parse tree for the expression (3 + 4) * (5 + 6)n

Solve it.

25. Define SDD, Synthesized and inherited attributes with examples, Analyse and bring out the distinct differences between S attributes and L attributes

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. For example,

Production	Semantic Rules
E → E1 + T	E.val = E1.val+T.val
E → T	E.val = T.val

Synthesized Attributes: The attribute of node that are derived from its children nodes are called synthesized attributes. Assume the following production:

S → ABC

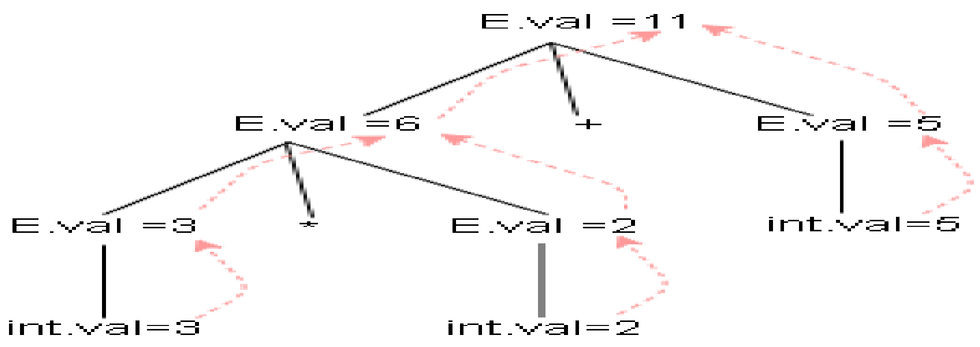
If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

Example for synthesized attribute:

Production	Semantic Rules
E ⟶ E1 * E2	{E.val = E1.val * E2.val}
E ⟶ E1 + E2	{E.val = E1.val +E2.val}
E ⟶ int	{E.val = int.val}

The Syntax Directed Definition associates to each non terminal a synthesized attribute called val.

In synthesized attributes, value owns from child to parent in the parse tree. For example, for string 3*2+5



<u>Inherited Attributes</u>

The attributes of node that are derived from its parent or sibling nodes are called inherited attributes. If the value of the attribute depends upon its parent or siblings then it is inherited attribute. As in the following production,

S ⟶ ABC

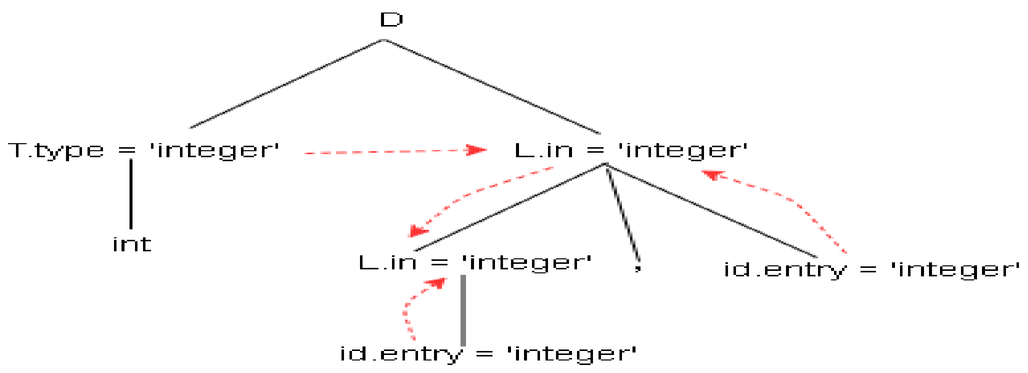
"A‟ can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Example for Inherited Attributes:

Production	Semantic Rules
D ® TL	L.in = T.type
T ® int	T.type = integer
T ® real	T.type = real
L ® L1, id	L1.in = L.in; addtype(id.entry, L.in)
L ® id	addtype(id.entry, L.in)

Symbol T is associated with a synthesized attribute type and symbol L is associated with an inherited attribute in.

In inherited attributes, values flows into a node in the parse tree from parents and/or siblings. For example, for input: int id, id



In an *L-attributed* SDD attributes may be inherited or synthesized, this is referred to as an *L-attribute* definition.
In an *S-attributed* SDD, attributes all attributes are synthesized - *S-attribute definition*.
An *L-attributed* grammar is a grammar that node dependency graph of any of its production rules has data-flow arrow pointing from an attribute to an attribute to its left. Such grammars allow attributes to be evaluated in a *left-to-right* traversal.
An *S-attribute* grammars don't have inherited attributes at all, here attributes need to be retained only for non-terminal nodes that haven't yet been reduced to other non-terminals.

26. Write Semantic rules and a dependency graph for the Input string, based on the SDD given below: float id1,id2,id3
D → T L
T → int
T → float
L → L1 , id
L → id

Solve it.

27. (a) Analyse the expression given and write the three address form and DAG for a + a * (b – c) + (b - c) * d

Solve it.

(b)explain the three address form and apply this to form quadruple representation and triples and indirect triples for a= c *-d+ c *-d

Solve it.

(c)Write the SDT for converting an infix to prefix expression. Show the actions for translating the expression 2*3+4 into its equivalent prefix expression

Solve it.

28. Explain the following with an example
(a)Semantic rule (b)Synthesized attribute (iii)inherited attribute (iv)Annotated parse tree