KLS's VDIT HALIYAL.
DEPT OF CSE.

MODEL QUESTION PAPER-II.

SUB: SYSTEM SOFTWARE & COMPILERS. [18CS61]

MAX. MARKS: 100

NOTE: Answer any 2 full Questions, choosing ONE Full Question from each module.

MODULE-1.

1. a) Explain the SIC/XE machine architecture in detail. [10M]

b) Generate the complete Object program for the following SIC/XE assembly program. [10M]

```
WREEC   START   4050
        CLEAR   X
        LDT     LENGTH
WLOOP   TD      OUTPUT
        JEQ     WLOOP
        LDCH    BUFFER,X
        WD      OUTPUT
        TIXR    T
        JLT     WLOOP
        RSUB
OUTPUT  BYTE    X'05'
        END
```

ADDRESS OF BUFFER 4033
ADDRESS OF LENGTH 4036

OPCODES:- CLEAR-B4, JEQ-30, WD-DC, JLT-38, LDT-74, LDCH-50, TIXR-B8, RSUB-4C.

OR

2. a) List all assembler independent & dependent features & explain program relocation. [5M]

b) Give the algorithm for Pass-1 of 2-pass Assembly. [10M]

c) What is loader? Write basic functions the loader has to perform. [5M]

MODULE-2

3.a) With the help of a diagram, explain the various phases of compiler. [10M]

b) Construct the transition Diagram to recognize tokens i) identifier ii) Relational Operators iii) Unsigned no.s [6M]

c) Define token, patterns & lexems. [4M]

OR

4. a) Discuss the various application of compiler technology? [10M]

b) What is Regular Expression? Write the algebraic laws of regular expression. [6M]

c) List & explain the reasons for separating the analysis portion of compiler into lexical & Syntax analysis phase. [4M]

MODULE-3

5.a) Define left Recursion, Write an algorithm to eliminate left recursion. Eliminate left recursion from the following grammar. [10M]

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$.

b) Consider the below grammar. [10M]

$S \rightarrow (L) / a$

$L \rightarrow L,S / S$

make the necessary changes to suit for top-down parsing. Is the grammar LL(1)? & parse the i/p string (a,a).

OR

6. a) What is Handle pruning? Give Bottom-up parser for the input string aaa*a++ using the grammar
$S \rightarrow SS+ / SS* / a$. [8M]

b) What is recursive decent parser? Trace & Explain working of recursive decent parser for input "bcd" & grammar: [6M]
$A \rightarrow bCd$.
$C \rightarrow cele$.

c) Explain any 2 Error-recovery strategies. [4M]

MODULE-4

7. a) Discuss the characters used in the meta language as part of std ASCII character set used in UNIX OS. [8M]

b) Write a Lex specification? [6M]

c) Write a Lex program to identify the decimal number. [6M]

OR

8. a) What is YACC? Explain the different sections used in writing the YACC specification. [10M]

b) Write a YACC program to function as a calculator which performs addition, substraction, multiplication & division on unary operators. [10M]

MODULE-5

9. a) Consider a grammar given below: [10M]

$S \rightarrow E\$$.
$E \rightarrow E+T / E-T / T$
$T \rightarrow T*F / T/F / F$
$F \rightarrow (E) / digit$.
$N \rightarrow ;$

(i) Obtain the SDD for the above grammar.

(ii) Construct annotated parse tree for input string

b) Write 3-address code syntax tree & DAG for the expression $a+a*(b-c)+(b-c)*d$. [10M]

10. a) List & Explain the different 3-address instruction form. [6M]

b) Construct the dependency graph for the declaration float id_1, id_2, id_3. [6M]

c) Discuss the various issues in the code generation phase. [8M]

*****

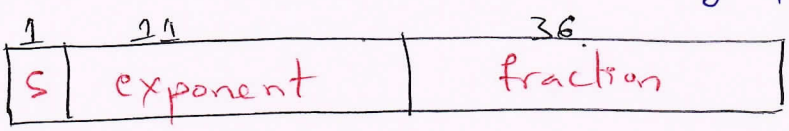
SOLUTION FOR THE MODEL QUESTION PAPER-II

1.a). SIC/XE machine architecture :- [10M]

Memory :- 2^{20} bytes — 1 MegaByte.

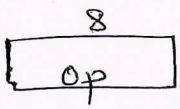
Registers :-
- B - 3 - Base Reg. used for addressing
- S - 4 - General working reg - No special use
- T - 5 - General working reg - No special use
- F - 6 - Floating-point accumulator (48 bits)

Data formats :- It provides same data formats as std version. There is a 48-bit floating-point data type :-

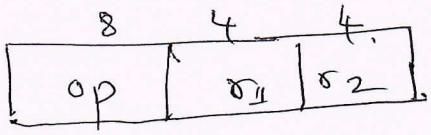


Instruction Formats :- There are 4 formats :-

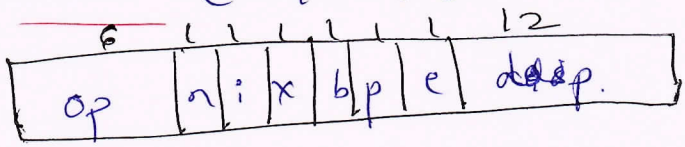
Format 1 (1byte) :



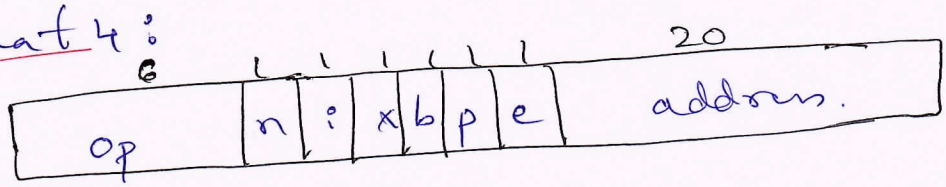
" 2 (2byte) :



Format 3 (3byte) :



Format 4 :



Addressing Modes :-

* Two relative addressing Modes are :-
  - ⇒ Base relative — b=1, p=0. TA = (B) + disp
  - ⇒ Program Counter " — b=0, p=1 TA = (PC) + disp.
* Direct addressing Mode — . Immediate ⇒ i=1, n=0.
* Indirect " " — i=0, n=1
* Indexed " " — x=1.
* Simple " " — i=n=0 or i=n=1

* Instruction Set-

⇒ load & store instruction – LDB, STB.

⇒ floating arithmetic operations – ADDF, MULF, DIVF, SUBF

⇒ Reg-Reg arithmetic " – ADDR, SUBR, DIVR, MULR

⇒ Supervisor call instn (SVC).

* Input & Output :-

⇒ To transfer a byte of info using instruction,

SIO, HIO, TIO.

b). W: [long]

405D		CLEAR	X	B410
405F		LDT	LENGTH.	774000
4062	WLOOP	TD	OUTPUT	E32011
4065		JEQ	WLOOP	332FFA
4068		LDCH	BUFFER,X	53C003
406B		WD	OUTPUT	DF2008
406E		TIXR	T.	B850
4070.		JLT	WLOOP	382FEF
4073.		RSUB		4F0000
4076	OUTPUT	BYTE	X'05	05.
4077		END.		

H^WREEC^004057^000017.

T^004057^1A^B410^774000^E32011^332FFA^53C003^DF2008^B850^382FEF^4F0000^05

E^004057.

2. a) Assembler independent machine feature:-

* Literals
* Expressions
* Control section & Program Linking.
* Symbol Defining statements
* Program Blocks.

* M/c Dependent assembler Features:-
  ⇒ Instruction Formats & Addressing Modes
  ⇒ Program Relocation.

[5M]

Program Relocation:-

⇒ Actual starting address of program is not known until load time.

⇒ An object program that contains the information neccesary to perform this kind of modification is called Relocatable Program.

⇒ No modification is needed: operand is using program-counter relative or base relative addressing.

⇒ The only parts of program that require modifications at load time are those that specified direct addresses

⇒ MODIFICATION RECORD

- COL 1 - M.
- COL 2-7 Starting location of the address field to be modified, relative to beginning of the program.
- col 8-9. length of the address field to be modified in half-bytes.

b) Pass 1 of 2-pass Assembler algorithm:-

```
begin:
    read first input line
    if OPCODE = 'START' then
    begin
        save #[OPERAND] as starting address
        initialize LOCCTR to starting address
        write line to intermediate file
        read next input line.
end.
```

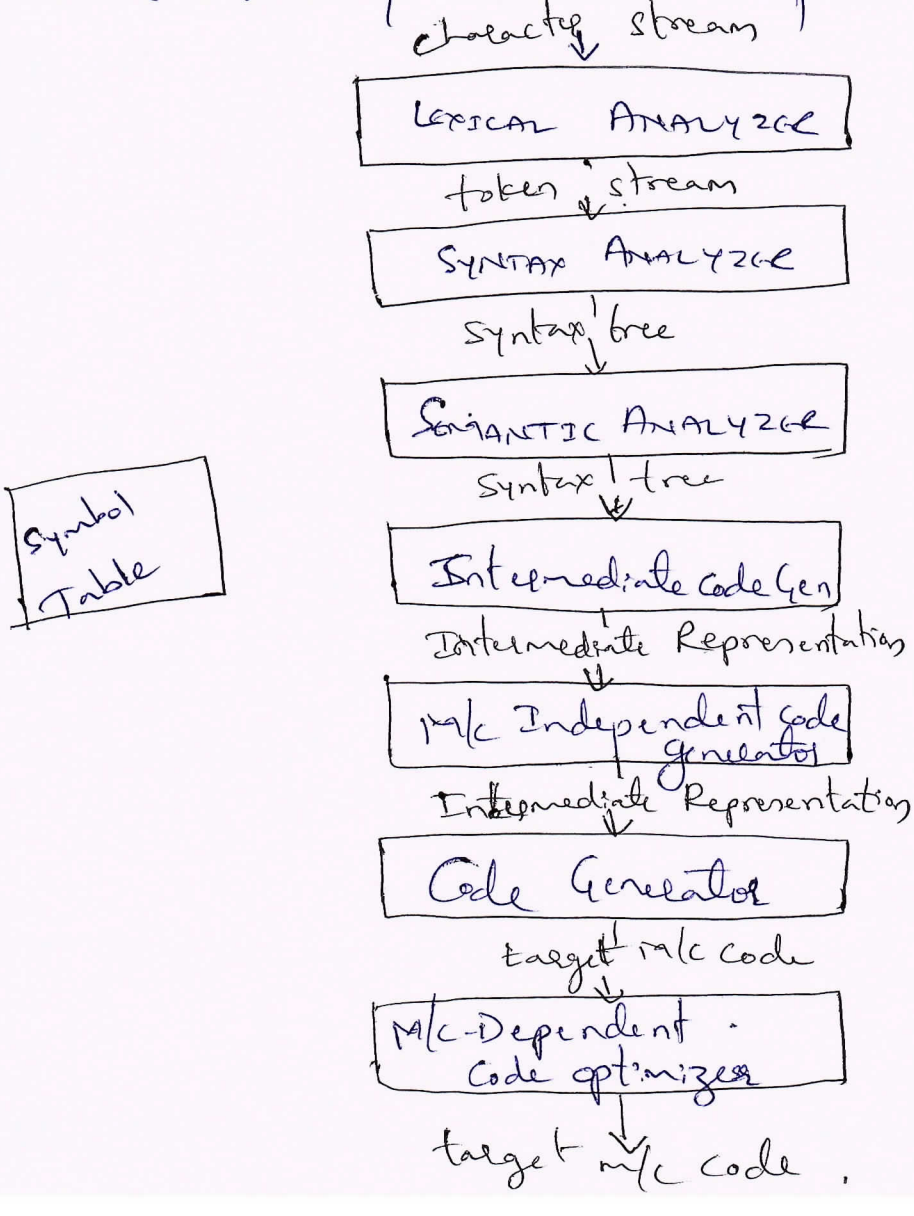
```
else
    initialize LOCCTR to 0.
while OPCODE ≠ 'END' do.
begin.
  if this is not a comment line then
  begin.
    if there is a symbol in the LABEL field then
    begin
      search SYMTAB for LABEL
      if found then
        set error flag.
      else
        insert (LABEL, LOCCTR) into SYMTAB.
    end.
  search OPTAB for OPCODE
  if found then
    add 3 to LOCCTR
  else if (OPCODE = 'WORD' then
    add 3 to LOCCTR.
  else if (OPCODE = 'RESW' then.
    add 3 * #[OPERAND] to LOCCTR
  else if OPCODE = 'RESB' then
    add #OPERAND to LOCCTR
  else if 'OPCODE = BYTE' then
  begin
    find length of constant in bytes
    add length to LOCCTR
  end.
  else
    set error flag.
  end.
  write line to intermediate file
  read next input line.
end
write last line to intermediate file
end. Save (LOCCTR - starting address) as program length.
```

[10m]

2). LOADER is system program that performs the loading function. It support relocation & Linking. [5M]

* The basic functions of loader is - to bringing an object program into memory & stating its execution.
* Relocating a program, which modifies the object program so that it can be loaded at an address different from location originally specified.
* Linking which combines two or more separate object programs. & supplies the information needed to allow references between them.

3.a) The different phases of Compiler are :- [10M]



* There are 2 parts of mapping :-
  → ANALYSIS , → SYNTHESIS.

* The different phases of Compilers are :-

⇒ LEXICAL ANALYSIS :- It reads the stream of characters making up source program & groups the characters into meaningful sequences called lexemes.

⇒ SYNTAX ANALYSIS :- Parser uses the first components of tokens produced by lexical analyzer to create a tree like intermediate representation that depicts the grammatical structure of token stream.

⇒ SEMANTIC ANALYSIS :- It uses the syntax tree & information in the symbol table to check source program for semantic consistency with the definition.

⇒ An important part of semantic analysis is type checking, where compiler checks that each operator has matching operands.

⇒ Intermediate CODE GENERATION :- In a process of translating a source program into target code compiler may construct one or more intermediate representations which have variety of forms.
* Syntax tree are a form of intermediate representation.

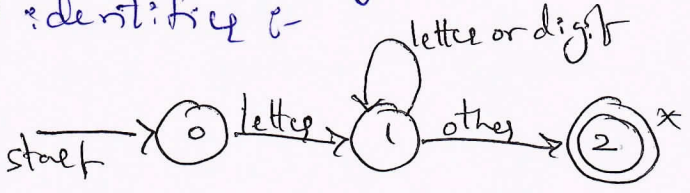
⇒ CODE OPTIMIZATION :-
There are simple optization which improve running time of target program.

⇒ CODE GENERATION :- There are ~~simple optimi~~ takes

an input an intermediate representation of source program & maps it into target language.

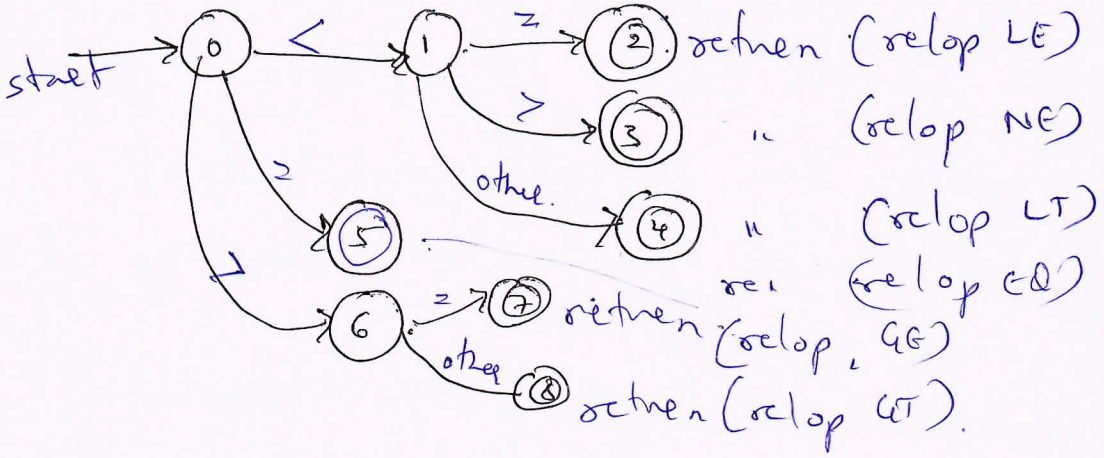
b) Transition Diagram for the following :-

i) identifier :-

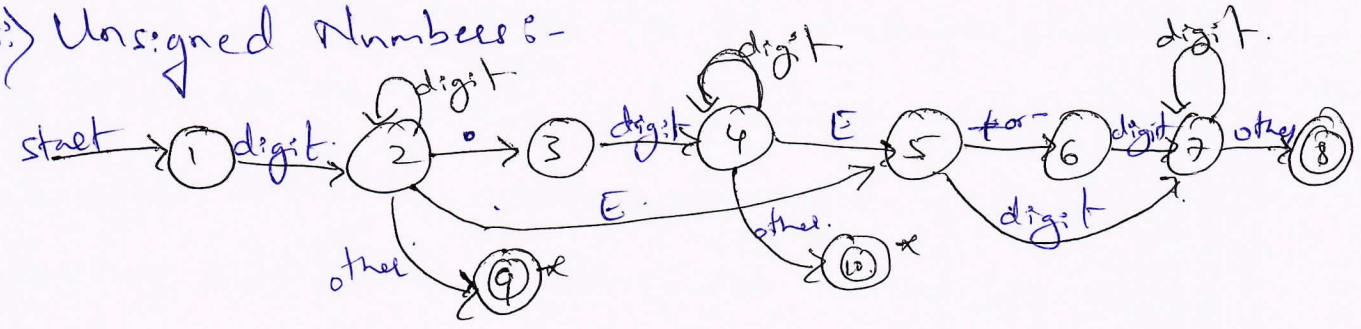


[6M]

ii) Relational operators :-



iii) Unsigned Numbers :-



c) Token is a pair consisting of token name & an optional attribute value. Token name is an abstract symbol representing a kind of lexical unit. [4M]

Pattern is a description of the form that the lexemes of a token may take.

Lexemes is a sequence of ~~char~~ characters in the source program that matches the pattern for token & it is identified by lexical analyzer.

4.a). The applications of Compiler Technology are :-

i) Implementation of High-level programming Languages :-

→ It defines a programming abstraction, the programmer expresses an algorithm using language & Compiler must translate program to target language.

⇒ There are different programming Languages like C, C++, Java, Small talk, C# etc.

⇒ The key behind object orientation are :

* Data abstraction * Inheritance of properties.

ii) Optimization for Computer architecture :-

⇒ High performance systems take advantage of same 2 basic techniques.

* Parallelism * Memory Hierarchies

⇒ Parallelism can be at several levels :- at instruction level, where multiple operations are executed simultaneously & processor level where different threads of same application are running on different processors.

⇒ Memory Hierarchies consists of several levels of storage with different speeds & sizes.

iii) Design of New Computer Architectures :

The 2 architecture are designed.

RISC :- Reduced Instr Set Computer

CISC :- Complexed Instr Set Computer.

* The processor architecture PowerPC, SPARC, MIPS Alpha & PA-RISC are based on RISC.

* x86 architecture based on CISC instr set.

* specialized architectures are :- VLIW, SIMD, systolic arrays, multiprocessors with shared memory & multiprocessors with distributed memory.

⇒ PROGRAM TRANSLATIONS :- There are some of imp. application of program translation techniques :-

* BINARY TRANSLATION :- It can be used to provide backward compatability.

* H/W SYNTHESIS :- H/W design are described in langnages like VHDL, RTL.

* DB Query Interpreters :- SQL are used to search database.

* Compiled Simulation :- It is used in many sate-of art tools that simulate designs written in VERILOG or VHDL.

* Slw Productivity TOOLS

⇒ Programs are most complicated work. ers proceed. they consist of many details every one must be correct before the programs work completely.

* Bound Checking :- 'C' doesn't have array bounds check it is upto user to ensure the arrays are not accessed out of bounds.

* Type Checking :- Operation is applied to wrong type of object, if parameters passed to procedure don't match signature of procedure.

b). Regular Expression is a pattern description using meta-language which use to describe a particular pattern of interest.

⇒ Algebraic Laws of Regular Expression :- [6M]

i) Commutative Law → $r|s = s|r$

ii) Associative Law → $r|(s|t) = (r|s)|t$

iii) Concatenation is associative → $r(st) = (rs)t$.

iv) " is Distributive → $r(s|t) = rs|rt$.

v) Identity for concatenation → $r\epsilon = \epsilon r = r$.

vi) Closure Law → $r^* = (r|\epsilon)^*$

vii) Idempotent Law → $r^{**} \Rightarrow r^*$

c) The analysis of compiler is separated into LEXICAL & SYNTAX ANALYSIS is :- [4M]

* Simplicity of design - The separation of lexical & syntactic analysis allows us to simplify atleast one of the tasks.

* Compiler efficiency is improved ; where Lexical analyzer use to apply specialized techniques which serve only lexical task not the job of parsing.

* Compiler portability is enhanced.

5. a) A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α.

Algorithm for Eliminating Left Recursion :-

arrange the nonterminals in some order $A_1, A_2 \ldots A_n$.

for(each i from 1 to n-1){

for(each j from 1 to i-1){

replace each production of the form $A_i \to A_j\gamma$

by the productions $A_i \to \delta_1\gamma / \delta_2\gamma / \cdots / \delta_k\gamma$.

where $A_j \to \delta_1 / \delta_2 / \cdots / \delta_k$ are all current

A_j-productions.

}

eliminate the immediate left recursion among

the A-production.

}.

[10M]

* To Eliminate the left recursion : apply

$A \to A\alpha / \beta \Rightarrow$ $A \to \beta A'$, $A' \to \alpha A' / \varepsilon$.

$E \to E+T / T$.

$T \to T*F / F$

$F \to (E) / id$

i) $E \to E+T / T$ $\quad A = E \quad \alpha = +T \quad \beta = T$

$A \to A\alpha$.

$E \to TE'$

$E' \to +TE' / \varepsilon$

ii) $T \to T*F / F \Rightarrow$ $T \to FT'$

$T' \to *FT' / \varepsilon$

iii) $F \to (E) / id \Rightarrow$ The Grammar is

$E \to TE'$

$E' \to +TE' / \varepsilon$

$T \to FT'$.

$T' \to *FT' / \varepsilon$

$F \to (E) / id$.

b) $S \to (L) / a$

$L \to L, S / S$.

$L \to L, S / S \Rightarrow$ $L' \to ,SL' / \varepsilon$, $L \to SL'$.

$\Rightarrow$ After Eliminating Left Recursion :-

$\therefore$ $S \to (L)/a$

$L \to SL'$

$L' \to ,SL'/\varepsilon$

Symbol	First	Follow
S	{(, a}	{$, , ,)}
L	{(, a}	{)}
L'	{, , ε}	{)}

Predictive Parsing Table:-

I/p Symbol	(	a	,	)	$
S	$S \to (L)$	$S \to a$			
L	$L \to SL'$	$L \to SL'$			
L'		$L' \to \varepsilon$	$L' \to ,SL$		

It is LL(1) grammar.

[10M]

*Parsing story. (a, a);

Stack	Input	Action
S $	(a,a)$	$S \to (L)$
(L) $	(a,a)$	match (
L) $	a,a)$	$L \to SL'$
SL') $	a,a)$	$S \to a$
aL') $	a,a)$	match a
L') $	,a)$	$L' \to ,SL'$
,SL') $	,a)$	match ,
SL') $	a)$	$S \to a$
aL') $	a)$	match a
L') $	)$	$L' \to \varepsilon$
') $	)$	match)
$	$	Accepted.
$	$	

6.a) Handle is substring that matches the body of a production & whose reduction represents one step along the reverse of rightmost derivation. [8M]

* Grammar $S \to SS+ / SS* / a$. ⇒ i/p : aaa*a++.

STACK	Input	Action
$	aaa*a++$	Shift a
$a	aa*a++$	Reduce S→a
$S	aa*a++$	Shift a
$Sa	a*a++$	Reduce S→a
$SS	a*a++$	Shift a
$SSa	*a++$	Reduce S→a
$SSS	*a++$	Shift *
$SSS*	a++$	Reduce S→SS*
$SS	a++$	Shift a
$SSa	++$	Reduce S→a
$SSS	++$	Shift +
$SSS+	+$	Reduce S→SS+
$SS	+$	Shift +
$SS+	$	Reduced S→SS+
$S	$	Accept.

b) Recursive Descent parsing which consists of a set of procedures, one for each non-terminal.

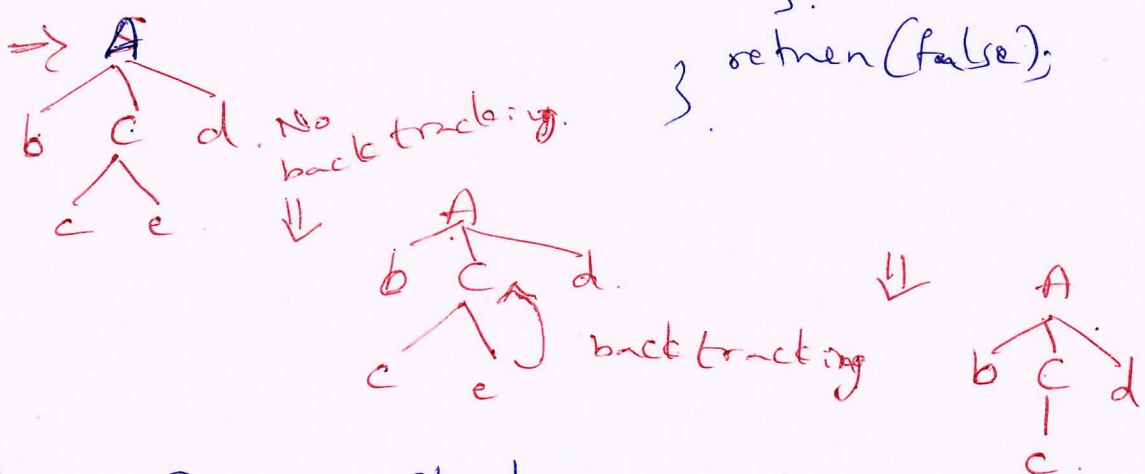
∴ Grammar :- A → bCd.
C → ce|e

```
procedure A()
{ if (input == 'b')
  { Advance();
    C();
    if(input == 'd')
    { Advance();
      return (true);
    }
  else
    { return (false); }
  }
else return (false);
}.
```

```
procedure C()
{ isave = in_ptr;
  if (input == 'c')
  { Advance();
    if(input == 'e')
    { Advance();
      return (true);
    }
  else { inptr = isave;
    if (input == 'e')
    { Advance();
    } return (true);
  } return (false);
}.
```

[6M]

bcd.



c). Error Recovery Strategies are :-

[4M]

⇒ Panic Mode Recovery :-

Parser discards input symbols one at a time until one of a designated set of synchronizing tokens are found.

* It often skips a some amount of input without checking it for additional errors. It has advantage simplicity & guaranteed not to go into an infinite loop.

* Phrase-level Recovery :- The parser may perform local corrections on remaining input, it may replace a prefix of remaining input by some string. that parser allows to continue.

7.a) The characters used in the meta-language are :-

. : Matches any single character except the newline characters.

* : Matches zero or more copies of preceding expression.
eg :- [0-9]*\.[0-9]+.

[ ] : Character class which matches any character within brackets.

^ : Matches beginning of line as first character of Regular Expression. eg :- [^abc].

$: Matches end of line as the last character of Regular Expression. eg : [abc]$.

{ } : Indicates how many times the previous pattern is allowed to match when containing one or 2 numbers. eg :- A{1,3}

\ : Used to escape meta character & 'c' escape sequence
eg :- [\n] , [*].

? : Matches zero or one occurence of preceding Regular expressions. eg :- -?[0-9]+.

| :- Matches either the preceding RE or following RE.
eg :- cow | sheep | dog.

b) LEX specification is a structure of writing a LEX program., which consists of 3 parts: [6M]

* DEFINITION SECTION :- It has literal code block which enclosed in %{ & %}. The declaration of 'C' can be embedded here.. & also header files can be added.

* Followed by this section the Rule Section begins. It is begin with %%.

* RULE SECTION :- contains 2 parts.

* pattern and action :- pattern is a LEX specification which can be matched & execute the given action.

* Pattern & action will be separated by one space. action is nothing but the actual 'C' code. which is embedded between the { & }.

* Subroutine Section :- It begins after the second %%. It actually contains any 'C' code & copied into the resulting parser.

* We can call yylex() to accept the input & divides the input into no. of tokens. Once tokens are matched then it executes the action.

eg:-

```
%{ #include <stdio.h>
%}
%%
. {printf ("Hello world"); }
%%
main()
{
   yylex();
}
```

c) Lex program to identify the decimal numbers:- [6M]

```
%{ #include <stdio.h>
   int a=0;
%}
%%
[0-9]*\.[0-9]+.   { printf("%s is decimal number", yytext); }
.   ;
%%
main()
{
 printf("Enter input:");
 yylex();
}
```

8. a) YACC is yet another Compiler Compiler, which takes a grammar to specify & writes a parser for recognizes valid sentences in the grammar.

* YACC has the same 3-part structure as a LEX specification. First section.
* DEFINITION SECTION: which handles control info for yacc generated parser & sets up the

the execution environment in which parser will operate. It begins with %{ & %}. [10M]

⇒ RULE SECTION :- Second section which is begin with %%, it contains the rules for the parser. Each rule made up of 2 parts: pattern & action.

* It describes the actual grammar as a set of production rules or simply rules. Each rule consists of a single name on LHS of ":" operator, a list of symbols & action code on RHS. & semicolon indicates the end of rule

* We use special character "|" which introduces a rule with same LHS as previous one.

⇒ SUBROUTINE SECTION :- It begins after %% & contains any 'C' code is copied into the resulting parser

* We call yyparse() to parse given grammar. yyerror() to check the errors of grammar.

eg:- %{ #include<stdio.h>
— — —
%}
%%
St : A
| B
;
%%
main()
{
printf("Enter the string: ");
yyparse();
pf("valid");
}

Lex
%{ #include "y.tab.h"
%}
%%
(aA) return A;
[bB] return B;
-/-)

yyerror()
{
pf("Invalid");
exit(0);
}

6) YACC program to function as calculator as +, /, -, *

YACC Part :-

```
%{ #include<stdio.h>
   #include<stdlib.h>
%}
%token NUM
%left '+' '-'
%left '*' '/'
%%
st : exp {printf("Valid Expression Result = %d", $$);
          exit(0); }
   ;
exp : exp '+' exp   { $$ = $1 + $3; }
    | exp '-' exp   { $$ = $1 - $3; }
    | exp '*' exp   { $$ = $1 * $3; }
    | exp '/' exp   { $$ = $1 / $3; }
    | '(' exp ')'   { $$ = $2; }
    | NUM           { $$ = $1; }
    ;
%%
main()
{ printf("Enter expression ");
  yyparse();
}
yyerror()
{ printf("Invalid Expression");
  exit(0);
}
```

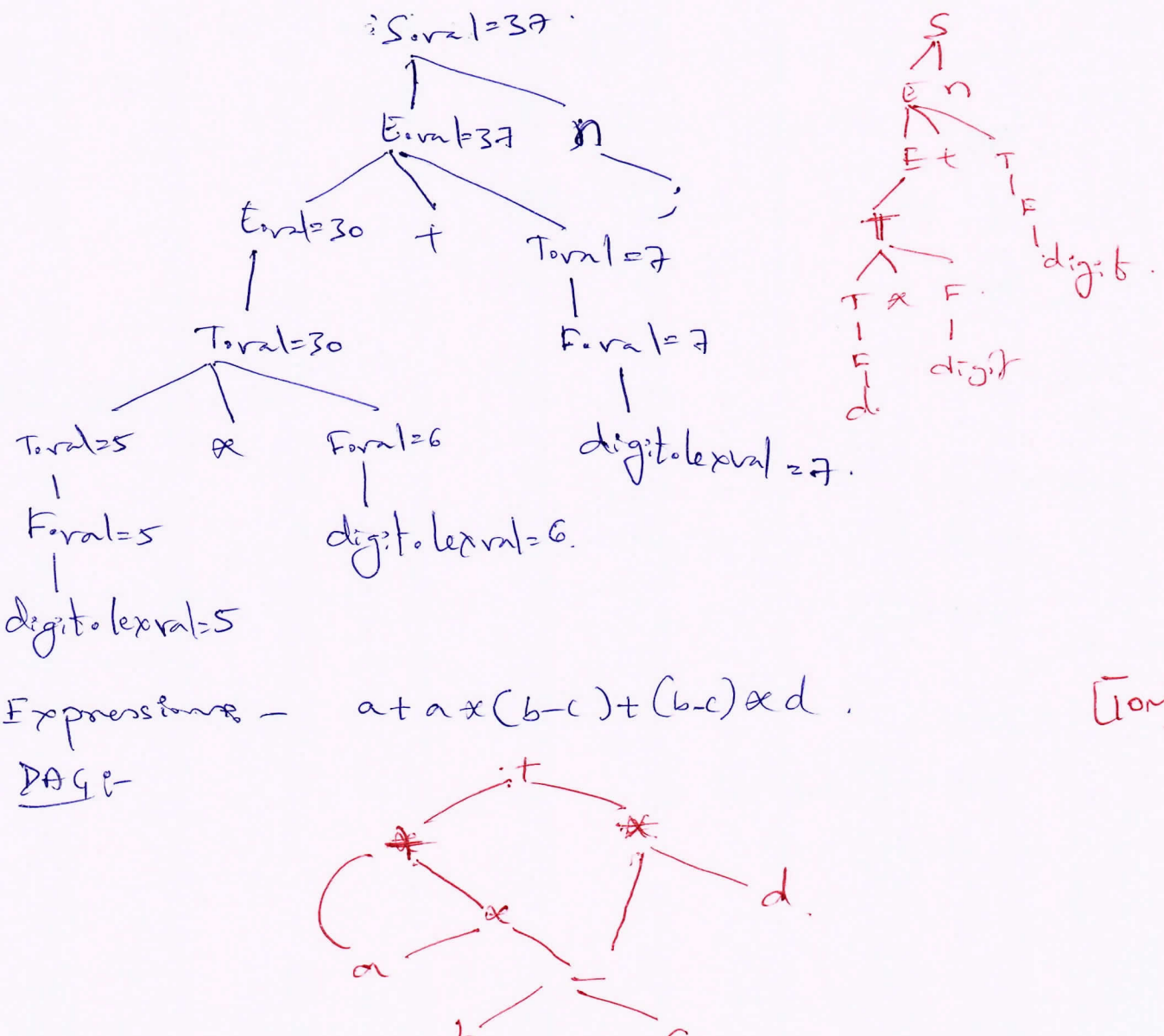
Lex part :-

```
%{ #include"y.tab.h"
%}
%%
[0-9]+ { yylval = atoi(yytext);
         return NUM; }
.      return yytext[0];
%%
```

9. a) SDD for the grammar :- [10M]

(i) Productions	Semantic Rules
1) S → E$	S.val = E.val
2) E → E + T	E.val = E.val + T.val
E → E − T	E.val = E.val − T.val
E → T	E.val = T.val
3) T → T * F	T.val = T.val * F.val
T → T / F	T.val = T.val / F.val
T → F	T.val = F.val
4) F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

(ii) Annotated Parse tree for 5*6+7;



b). Expression – a + a * (b − c) + (b − c) * d. [10M]

DAG :-

*3 address code :- $t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1$ *d

$t_5 = t_3 + t_4$.

Steps for constructing the DAG :-

1) P_1 = Leaf (id, entry-a)
2) P_2 = Leaf (id, entry-a) = P_1
3) P_3 = Leaf (id, entry-b)
4) P_4 = Leaf (id, entry-c)
5) P_5 = Node ('-', P_3, P_4);
6) P_6 = Node ('*', P_1, P_5);
7) P_7 = Node ('+', P_1, P_6).
8) P_8 = Leaf (id, entry-b) = P_3
9) P_9 = Leaf (id, entry-c) = P_4
10) P_{10} = Node ('-', P_3, P_4) = P_5
11) P_{11} = Leaf (id, entry-d)
12) P_{12} = Node ('*', P_5, P_{11})
13) P_{13} = Node ('+', P_7, P_{12});

10.a) Three address instruction forms are :- [6M]

*Assignment Instruction is a form as $x = y$ op z.

eg:- x = op y (unary operation.)

*Copy Instruction.

eg:- $x = y$ where x is assigned the value of y.

*Unconditional Jump Instruction.

eg:- jump goto L

* Conditional Jump Instruction. (<, =, ?, >=, ==).

eg:- If x goto L.
iFalse x goto 4.

* Procedure Call & return.

It is used to call any procedure & return [illegible] value from procedure.

*Indexed Copy instruction :-

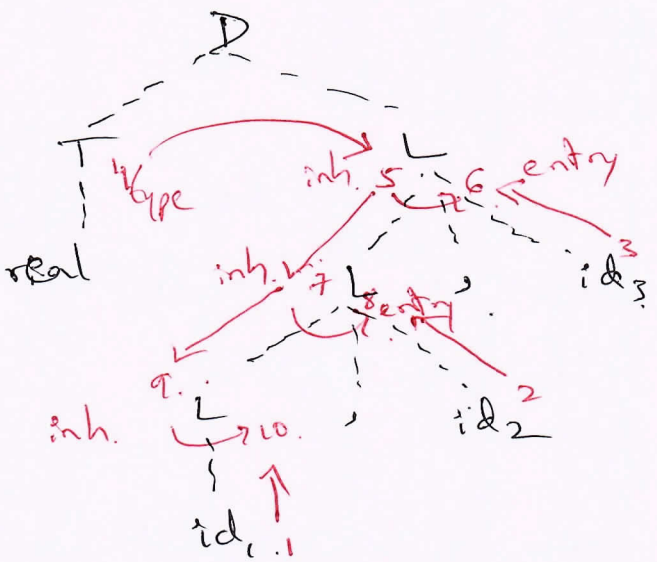
eg:- $z = y[i]$. , $z[i] = y$.

*Address & Pointer Assignment Instruction.

eg:- $x = \&y$. , $x = *y$. , $*x = y$.

b) Dependency graph for a declaration float id_1, id_2, id_3. [6m]

Production	Semantic Rules
i) $D \rightarrow TL$	$L.inh = T.type$
ii) $T \rightarrow int$	$T.type = int$
iii) $T \rightarrow float$	$T.type = float$.
iv) $L \rightarrow L_1, id$	$L_1.inh = L.inh$. addType(id.entry, L.inh)
v) $L \rightarrow id$	AddType(id.entry, L.inh).



c) The various issues in the Code Generator are:-

*Input to code Generator :-

It is intermediate representation of source program produced by frontend in the symbol table used to determine runtime address of data objects by the IR.

*There are many choices for IR include 3 address representations. such as Quadruples, triples & Indirect triples

*TARGET PROGRAM :-

The most common target m/c architecture are RISC & CISC, stack based architecture.

⇒ RISC m/c typically has many registers; 3-addrss instruction, simple addressing modes & simple instruction set architecture.

[8M]

⇒ CISC has few registers, 2 addrss instruction. variety of addressing modes, several register classes, variable length instruction.

⇒ Stack based machine operations are done by pushing operations on operands at top of stack.

*INSTRUCTION SELECTION :-

There are 4 factors of mapping IR into target m/c

⇒ level of IR.

⇒ Nature of Instn set Architecture

⇒ Desired Quality of generated code.

*If IR is high level, translate each IR stmt into sequence of m/c instn.

⇒ The nature of instn set of target m/c has a set of strong effect on difficulty of instn selection.

eg:- $x = y + z$

```
LD  R0, y
ADD R0, R0, z
ST  x, R0
```

⇒ Quality of generated code is determined by its speed & size.

*REGISTER ALLOCATION :-

Registers are fastest computational unit on target m/c.

⇒ The use of registers is divided into 2 subproblems :

1) Register Allocation :- select set of variables which resides in register at each point of program.

ii) Register Assignment :- we pick the specific register that a variable resides in.

* EVALUATION ORDER :-

The order in which computations are performed can affect the efficiency of target code.

→ Some computation order require fewer registers to hold intermediate results.

→ Picking a best order in the general case is difficult.

* * * * *

Verified

PREPARED BY :-

PROF. FARZANA. N. NAWAZ

Dean, Academics.