Model Question Paper

Sub : System Software & Compilers

Subcode : 18CS61

Max. Marks : 100

Module - 1

1. a) Explain the instruction formats & addressing modes of SIC/XE m/c architecture. [10M]

b) Write SIC/XE program to read 100 byte record from Device 'F5' into Buffer. use immediate & register to register instructions. [6M]

c) What are the fundamental functions of assembler [4M]

OR

2. a) With an algorithm explain Pass-1 of two pass assembler. [10M]

b) Define loader ? Write an algorithm for absolute loader ? [6M]

c) Compare two pass-assembler with single pass assembler How forward references are handled in one pass assembler. [4M]

Module-2

3a) Explain the different phases of compiler & translate the assignment "init = val + rate * 30;" by clearly indicate the output of each phase. [10M]

b) Explain the applications of compiler technology ? [8M]

c) Enlist the algebraic Laws for regular expression. [2M]

OR

4. a) Why the analysis of a compiler is separated into lexical & Syntax analysis? [4M]

b) How is input buffering of lexical analyzer implemented? Write an algorithm to lookahead code with sentinels. [8M]

c). Construct the transition diagrams for the following:
i) Relop ii) identifier iii) unsigned number. [8M]

Module-3

5. a) Write an algorithm to eliminate left factoring from the given grammar. [4M]

b) Construct Predictive parsing table by making neccesary changes to grammar given below & show parsing of string "id+id*id". [12M]

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id.$

c) Eliminate the Left Recursion from the following Grammar. [4M]

$A \rightarrow BC/a.$

$B \rightarrow CA/Ab$

$C \rightarrow AB/CC/a.$

OR

6. a) Write a Recursive Descent parser for the grammar [4M]

$S \rightarrow cAd$

$A \rightarrow ab/a.$

for the input "cad" trace the parser.

b) Construct First & Follow functions for the grammar $S \rightarrow (L) / a$, $L \rightarrow L, S / S$ [8M]

c) What is meant by handle pruning? Explain the shift Reduce parsing with example. [8M]

MODULE-4

7.a) Explain the communication between Parser & lexer with neat block diagram? [5M]

b) What is regular expression? Explain various regular expression with example? [10M]

c) Write a LEX program to count the number of vowels & consonants in a given string. [5M]

OR

8.a) Explain the structure of YACC program. [6M]

b) Write a YACC program to recognize an arithmetic expression involving operators. +, *, -, /. [8M]

c) What is shift / Reduce parsing? Explain with an example? [6M]

MODULE-5

9.a) Give SDD to process a simple desk calculator & show annotated parse tree for the expression 3*5+4n; [10M]

b) Give SDD for simple variable declaration in 'C' & construct a dependency graph for

the following expression. int a,b,c; [10M]

OR

10. a) Discuss the issues in the design of a code generator. [10M]

b) Obtain the directed Acyclic graph for the expression "a+a*(b-c)+(b-c)*d". [10M]
Also give sequence of steps for constructing the same.

SOLUTIONS

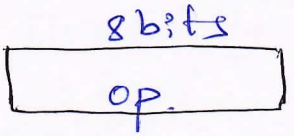
By : PROF. FARZANA. N. NADAF

1.a) The SIC/XE machine architecture :-

Instruction Formats :-

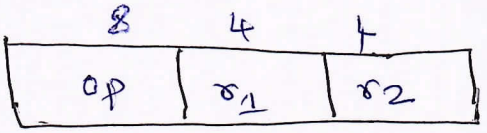
There are 4 types of formats :-

(i) Format 1 (1byte) :



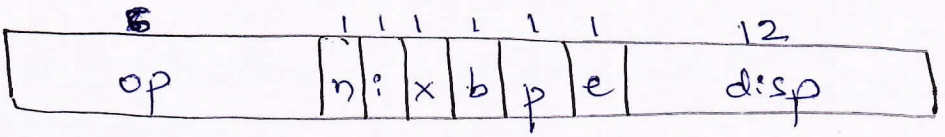
eg:- RSUB

(ii) Format 2 (2byte) :



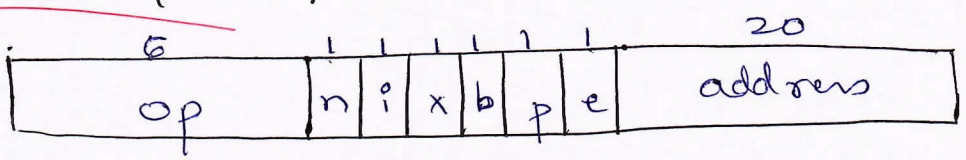
eg:- COMPR A, S.

(iii) Format 3 (3byte) :



eg:- LDA RETADR.

(iv) Format 4 (4byte) :



eg:- +JSUB RDREC.

ADDRESSING MODES :-

There are 2 new Relative addressing modes.

MODE	INDICATION	T.A calculation
BASE RELATIVE	b=1, p=0	TA = (B)+disp
PC RELATIVE	b=0, p=1	TA=(PC)+disp

* Base relative addressing mode the displacement

is 12 bit unsigned integer.

* When P = 1, B = 0 it is program relative addressing mode.
* When flag e = 1 then it is called as Format 4 addressing mode & b = p = 0 bits are set to zero.
* When flag bit i = 1 & n = 0 then it is immediate addressing mode.
* i = 0 & n = 1 it is indirect addressing mode.
* When i = n = 0 or 1, both the bits are set 0 or one then it is called Simple addressing mode.
* When bit X = 1, it is indexed addressing mode.

b) Read 100Byte Data using immediate & Reg-Reg instructions

```
RDATA   LDX   #0.
        LDT   #100.
RLOOP   TD    INDEV.
        JEQ   RLOOP
        RD    INDEV
        STCH  BUFFER,X.
        TIXR  T.
        JLT   RLOOP
        RSUB.
        :
INDEV   BYTE  X'F5'
BUFFER  RESB  100.
```

c) The fundamental functions of assembler are :-

* Convert mnemonic operation codes to their machine language equivalents - eg:- Translate STL to 14.
* Convert symbolic operands to their equivalent machine addresses.
  eg:- Translate RETADR to 1033.
* Build the machine instructions in the proper format.
* Convert the data constants specified in src program into their internal machine representation.
  eg:- Translate EOF to 454F46.
* Write the object program & assembly listing.

2. a) Algorithm for Pass-I of 2-pass Assembler.

Pass-I :-

```
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 zero (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 (duplicate symbol)
          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
save [LOCCTR - starting address] as program length
end.
```

Pass 1: (define symbol)

* Assign addresses to all instruction in program
* Save the values assigned to all labels.
* Perform some procesing of assembler directives
* store the instruction after assembly in a intermediate file.

b) LOADER is a system program that performs the loading function. also support for linking & relocation.

* Algorithm for an ABSOLUTE LOADER.

```
begin.
    read HEADER RECORD
    verify program name & length.
    read first TEXT RECORD
    while record type ≠ 'E' do
    begin
        move object code to specified location
                                    in memory
        read next object program record.
    end
    jump to address specified in END RECORD.
end.
```

c) * The main problem in trying to assemble a program in one pass involves forward references.

→ It is easy to eliminate forward references to data items by simply defining symbols

in the source program before they are referenced.

* In multi-pass assembler that can make as many passes as are needed to process the definitions of symbols.

* The one-pass assembler handles forward references in two ways :-
  * Defining all symbols before they referenced in a source program.
  * To prohibit forward references to data items. so that we can reduce the size of problem. (forward jumping in loop conditions).

MODULE - 2

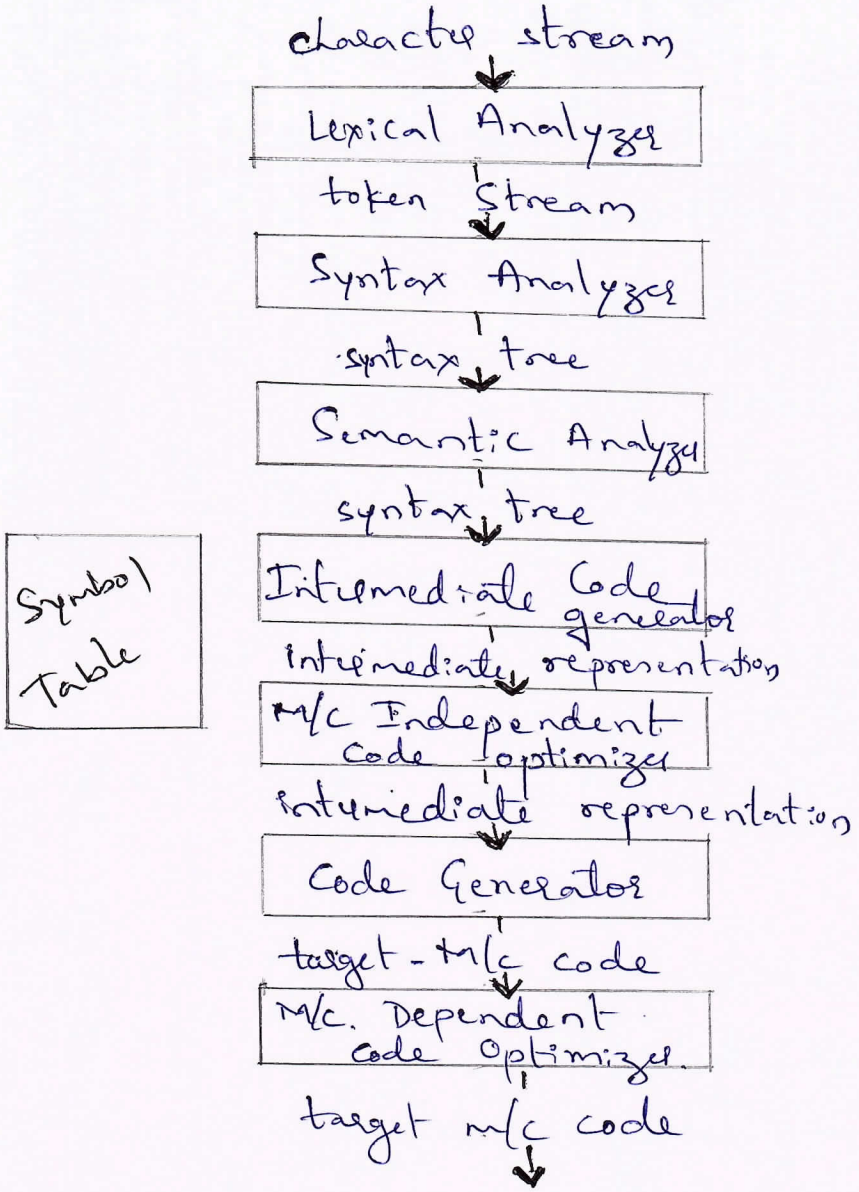
3.a) Explain the different phases of compiler & translate the assignment "init = val rate * 30;" by clearly indicate the o/p of each phase?

* There are 2 parts of mapping :-
  - ⇒ Analysis
  - ⇒ Synthesis.

⇒ Analysis part breaks up the source program into constituent pieces & imposes a gramatical structure on them.

⇒ Synthesis part constructs the desired target program from the intermediate representation & the information from the symbol table.

→ The block diagram indicates the different phases of compiler.



* 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 the 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 src prog 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

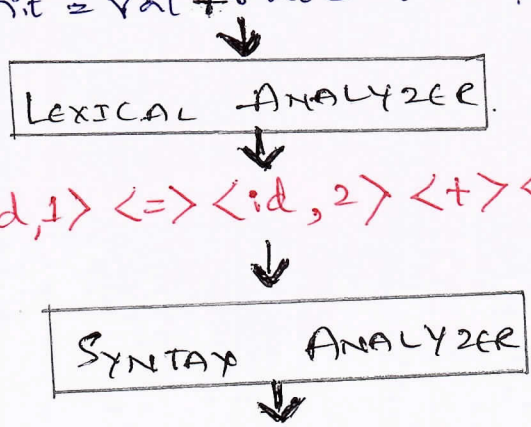
⇒ They are simple optimizations which improve running time of target program.

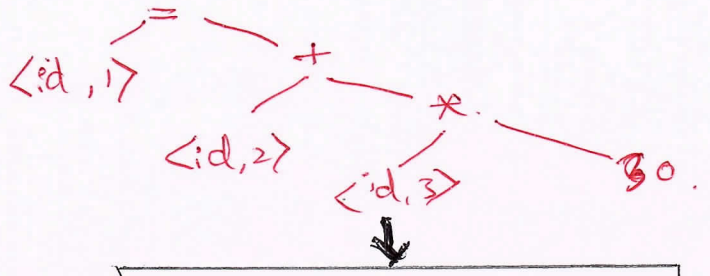
* Code Generation.

⇒ The code generator takes an i/p an intermediate representation of source programs & maps it into target language.

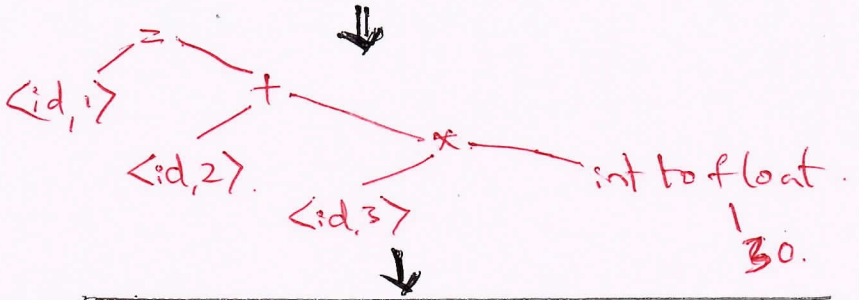
* Translation of an assignment statement.

init = val + rate * 30.





SEMANTIC ANALYZER



INTERMEDIATE CODE GENERATOR

$t_1 = \text{int to float}(30)$
$t_2 = id_3 * t_1$
$t_3 = id_2 * t_2$
$id_1 = t_3$

CODE OPTIMIZER

$t_1 = id_3 * 30.0$
$id_1 = id_2 + t_1$

CODE GENERATOR

```
LDF   R2, id3
MULF  R2, R2, #30.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

3.b) Explain the applications of compiler technology.

* Applications of Compiler Technology are :-

i) Implementation of High-level programming languages :-

* It defines a programming abstraction : the programer 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.

* OPTIMIZATIONS for COMPUTER ARCHITETURES

→ 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 applications are run on different processors.

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

* DESIGN OF NEW COMPUTER ARCHITECTURES.

→ The two architecture are designed :-
RISC :- Reduced Instn Set Computer.
CISC : Complexed Instn set Computer.

* The processor architecture PowerPC, SPARC, MIPS, Alpha & PA-RISC are based on RISC.
* x86 architecture based on CISC instn set.

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

* PROGRAM TRANSLATIONS :-

There are some of important applications of program Translation techniques :-

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

→ H/w Synthesis :- H/w design are described in languages like VHDL, RTL.

→ DB Query Interpreters :- SQL are used to search databases.

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

* S/w PRODUCTIVITY TOOLS.

→ Programs are most complicated work ever produced, they consist of many details, every one must be correct before the program 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 :- Eg:- operation is applied to wrong type of object, if parameters passed to procedure donot match signature of procedure

c) Enlist the algebraic laws for regular expression.

* Commutative Law
* Associative Law
* Closure
* Identity Law.

4.a) Why analysis of a compiler is separated into lexical & syntax analysis?

* The analysis of a compiler is seperated into lexical & synatax analysis is :-

i) Simplicity of design. The separation of lexical & syntactic analysis allows us to simplify at least one of the tasks.

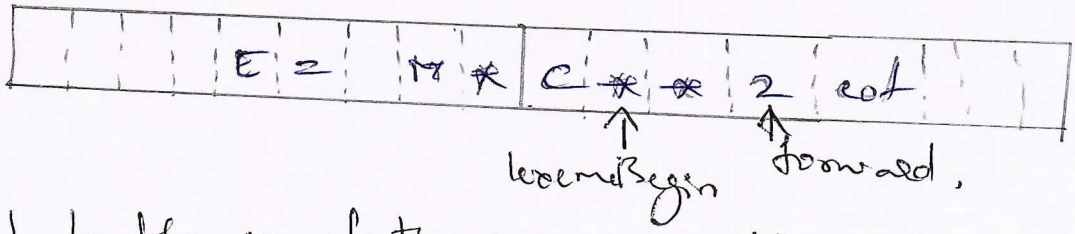
ii) Compiler efficiency is improved. when lexical analyzer use to apply specialized techniques. which serve only lexical task not the job of parsing.

iii) Compiler portability is enhanced.

b) How is i/p buffering of lexical analyzer implemented? Write algorithm to lookahead code with sentinels.

* The i/p buffering of lexical analyzer is implemented with buffer pairs & sentinels.

*



→ Each buffer is of the same size N, N is usually size of disk block eg. 4096.

⇒ The 2 pointers are maintained to the input are:-

* Pointer lexemeBegin, marks the beginning of the current lexeme.
* Pointer forward scans ahead until a pattern match is found.

⇒ Once next lexeme is forward set to the character at it's right end. After lexeme is recorded as an attribute value of token returned to the parser, lexemeBegin is set to character immediately after lexeme just found.

* Algorithm to lookahead code with sentinels

```
switch (*forward++)
{ case eof :
    if (forward is at end of first buffer)
    { reload second buffer;
      forward = beginning of second buffer;
    }
  else if (forward is at end of second buffer)
    { reload first buffer;
      forward = beginning of first buffer;
    }
  else { terminate lexical analysis; }
  break;
  cases for other characters
}
```

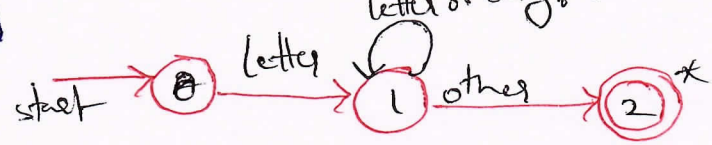
c. Construct the transition diagrams for the following:

i) Relop

ii) Identifier

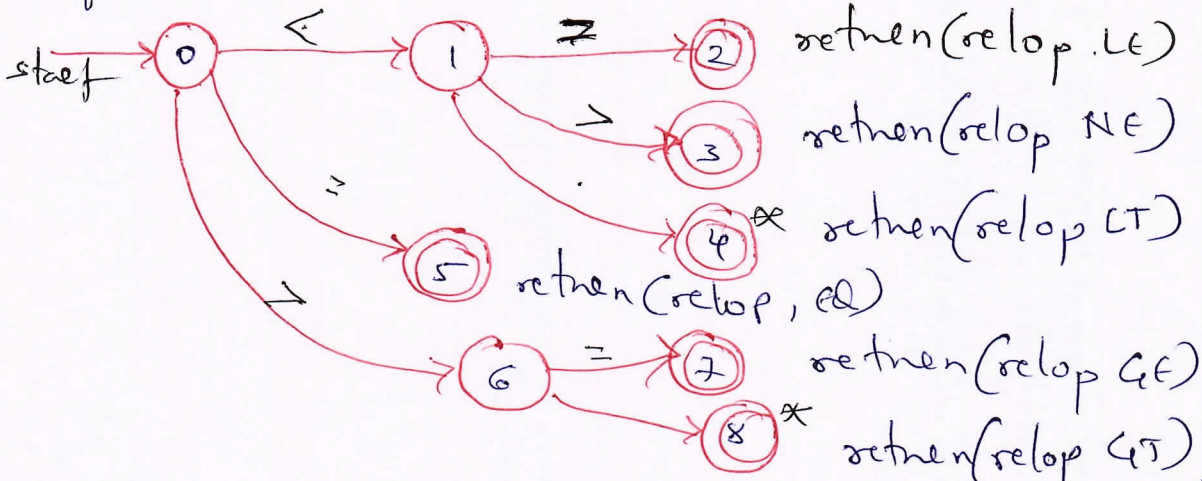
iii) Unsigned Nos

i)

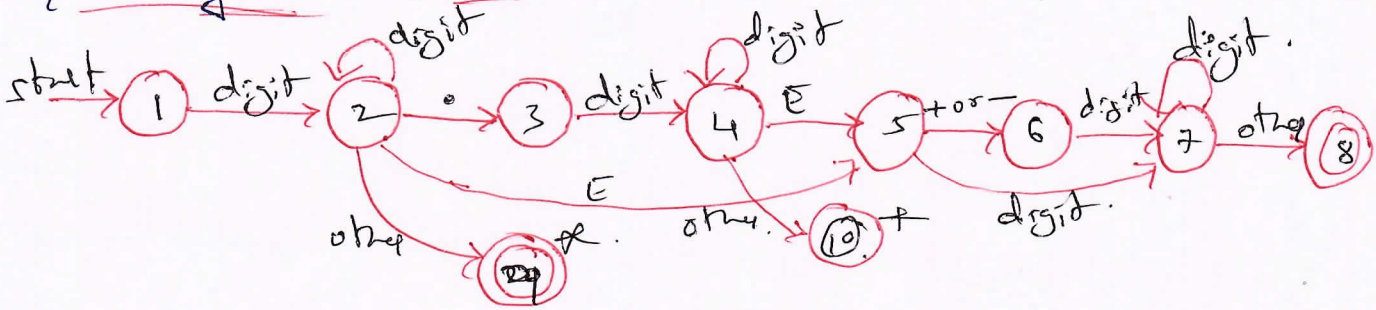


return (getToken(), installID()).

ii) Relops :-



iii) Unsigned Numbers :-



MODULE - 3

5. a) Write an algorithm to eliminate left factoring from given grammar.

Algorithm to Eliminate LEFT FACTORING G-

INPUT : Grammar G

OUTPUT : An equivalent left-factored Grammar.

METHOD : For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$ - i.e there is a nontrivial common prefix - replace all of the A-productions.

$A \to \alpha\beta_1 | \alpha\beta_2 | \text{----} | \alpha\beta_n | \gamma$ where γ represents all alternatives that do not begin with α by

$A \to \alpha A' | \gamma$

$A' \to \beta_1 | \beta_2 | \text{----} | \beta_n$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

b) Construct predictive parsing table by making neccessary changes to grammar given below & show parsing of string "id + id * id".

- $E \to E+T \mid T$
- $T \to T*F \mid F$
- $F \to (E) \mid id$

→ Grammar has left recursion;

:) $E \to E+T \mid T$

G.F $A \to A\alpha \mid \beta$.

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

$A = E \quad \alpha = +T \quad \beta = T$

$E \to TE'$

$E' \to +TE' \mid \varepsilon$.

:) $T \to T*F \mid F$ $\quad A = T \quad \alpha = *F \quad \beta = F$

$A \to A\alpha \mid \beta$.

$T \to FT'$

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

→ After eliminating the Grammar looks like

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

⇒Construction of FIRST & FOLLOW Functions:

I/P Symbol	FIRST	FOLLOW
E	{ (, id }	{), $ }
E'	{ +, ε }	{), $ }
T	{ (, id }	{ +,), $ }
T'	{ *, ε }	{ +,), $ }
F	{ (, id }	{ +, *,), $ }

⇒Construction of predictive parsing table:

i) $E \to TE'$

$A \to \alpha$. $A = E$ $\alpha = TE'$

$FIRST(\alpha) = FIRST(T) = \{ (, id \}$

$M[E, (] = E \to TE'$. $M[E, id] = E \to TE'$

ii) $E' \to +TE'$

$A \to \alpha$. $A = E'$ $\alpha = +TE'$.

$FIRST(\alpha) = \{ + \}$

$M[E', +] = E' \to +TE'$.

iii) $E' \to \varepsilon$

$A \to \alpha$ $A = E'$ $\alpha = \varepsilon$

$FOLLOW(A) = FOLLOW(E') = \{), \$ \}$

$M[E',)] = E' \to +TE'$

$M[E', \$] = E' \to +TE'$

iv) $T \to FT'$

$A \to \alpha$. $A = T$, $\alpha = FT'$.

$FIRST(F) = \{ (, id \}$

$M[T, (] = T \to FT'$

$M[T, id] = T \to FT'$.

v) $T' \to *FT'$

$A \to \alpha$ $A = T'$ $\alpha = *FT'$.

$FIRST(\alpha) = \{ * \}$

$M[T', *] = *FT'$

vi) $T' \to \varepsilon$

$A \to \alpha$ $A = T'$, $\alpha = \varepsilon$

$FOLLOW(T') = \{ +,), \$ \}$

$M[T', +] = T' \to \varepsilon$.

$M[T',)] = T' \to \varepsilon$

$M[T', \$] = T' \to \varepsilon$

vi) $F \to (E)$

$A \to \alpha$ $\quad A = F \;\; \alpha = (E)$

$FIRST((E)) = \{(\}$

$M[F, (] = F \to (E)$

vii) $F \to id$

$A \to \alpha$ $\quad A = F \;\; \alpha = id$

$FIRST(\alpha) = \{id\}$

$M[F, id] = F \to id$

NON TERMINAL	I/P SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

* Parsing of input string "id+id*id"

Matched	STACK	I/P	ACTION
	$E\$$	$id+id*id\$$	
	$TE'\$$	$id+id*id\$$	Pushed $E' \to TE'$
	$FT'E'\$$	$id+id*id\$$	" $T \to FT'$
	$id\,T'E'\$$	$id+id*id\$$	Match id
id	$T'E'\$$	$+id*id\$$	$F \to id$
id	$E'\$$	$+id*id\$$	$T' \to \varepsilon$
id	$+TE'\$$	$+id*id\$$	$E' \to +TE'$
id	$TE'\$$	$id*id\$$	Match +
id+	$FT'E'\$$	$id*id\$$	$T \to FT'$
id+	$id\,T'E'\$$	$id*id\$$	$F \to id$
id+id	$T'E'\$$	$*id\$$	Match id
id+id	$*FT'E'\$$	$*id\$$	$T' \to *FT'$
id+id*	$FT'E'\$$	$id\$$	Match *
id+id*	$id\,T'E'\$$	$id\$$	$F \to id$
id+id*id	$T'E'\$$	$\$$	Match id
id+id*id	$E'\$$	$\$$	$T' \to \varepsilon$
id+id*id	$\$$	$\$$	$E' \to \varepsilon$

∴ When there is $ in i/p & STACK remains $.
in annances Successful parsing.

c) Eliminate the Left Recursion from the following grammar:-

$A \to BC / a$
$B \to CA / Ab$.
$C \to AB / CC / a$.

⇒ Substituting C production in A production.

i) $A \to ABC / a$.
$A \to A\alpha / \beta$. $A \to \beta A' / \cdot A' \to \alpha A' / \varepsilon$.

$A \to aA'$.
$A' \to bCA' / \varepsilon$.

ii) $C \to CC / AB / a$.
$A \to A\alpha / \beta$ $A \to \beta A' \cdot A' \to \alpha A' / \varepsilon$.
$A = C$ $\alpha = \varepsilon$, $\beta = AB, a$.

$C \to ABC' / aC'$
$C' \to C' / \varepsilon$ After eliminating LR.

∴

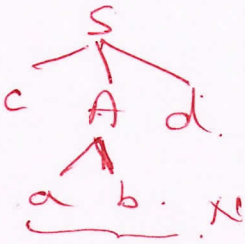
$A \to aA'$.
$A' \to bCA' / \varepsilon$
$B \to CA / AB$
$C \to ABC' / aC'$
$C' \to C' / \varepsilon$

6.a) Write a Recursive Descent parser for the grammar
$S \to cAd$ for i/p "cad" trace the parser.
$A \to ab / a$

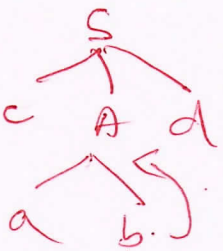
⇒.
```
Procedure S()
{ if(input == 'c')
  { Advance();
    A();
    if(input == 'd')
    { Advance();
      } return(true);
  else { return(false); }
```

```
else {
    return (false); }
}
```

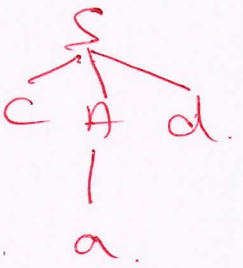
```
procedure A()
{
    isave = in_ptr;
    if (input == 'a')
    {
        Advance();
        if (input == 'b')
        {
            Advance();
        } return(true);
    else{
            in_ptr = isave;
        if (input == 'a')
        {
            Advance();
    } } return(true);
return (false);
}
```



No backtrack.



Backtracking.



b) Construction of FIRST & FOLLOW functions grammar:-

$S \to (L) / a$

$L \to L, S / S$

⇒

To eliminate the LR.

$L \to L, S / S$

$A \to A\alpha / \beta$

$A \to \beta A'$

$A' \to \alpha A' / \varepsilon$

$A = L \quad \alpha = , S$

$\beta = S$

$L \to S L'$

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

S/R Symbol	FIRST	FOLLOW
S	{ (, a }	{ $, , ,) }
L	{ (, a }	{) }
L'	{ , , ε }	{) }

Grammar :-

$S \to (L) / a$

$L \to S L'$

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

i) Follow(S) :- { $ }

S → (L)

A → αBβ

A = S α = (B = L β =)

First(β) = {) }

ii) Follow(S) :- { $, , ,) }

L → , SL'

A → αBβ

A = L α = , B = S, β = L'

First(β) = (L') = { , , ε }

apply rule (3)

Follow(L) = {) }

iii) Follow(S) = { , , $,) }

L → SL' A = L

A → αBβ α = ε B = S β = L'

First(β) = (L') = { , , ε }

iv) Follow(L') = {), $ }

L → SL' A = L B = L'

A → αBβ β = ε α = S

First(β) = { ε } rule (2)

Follow(L) = {) }

v) L' → , SL'

A → αBβ A = L' α = , S B = L' β = ε

rule (3)

Follow(L') = {) }

c) What is meant by handle pruning? Explain the shift Reduce parsing with example?

⇒ "Handle" is substring that matches the body of a production & whose reduction represents one step along the reverse of rightmost derivation.

⇒ Shift/Reduce parsing has 4 different actions:

* Shift : shift next i/p symbol onto top of stack.

* Reduce :- The right end of the string to be reduced must be at top of stack.

* Accept :- Announces successful completion of parsing.

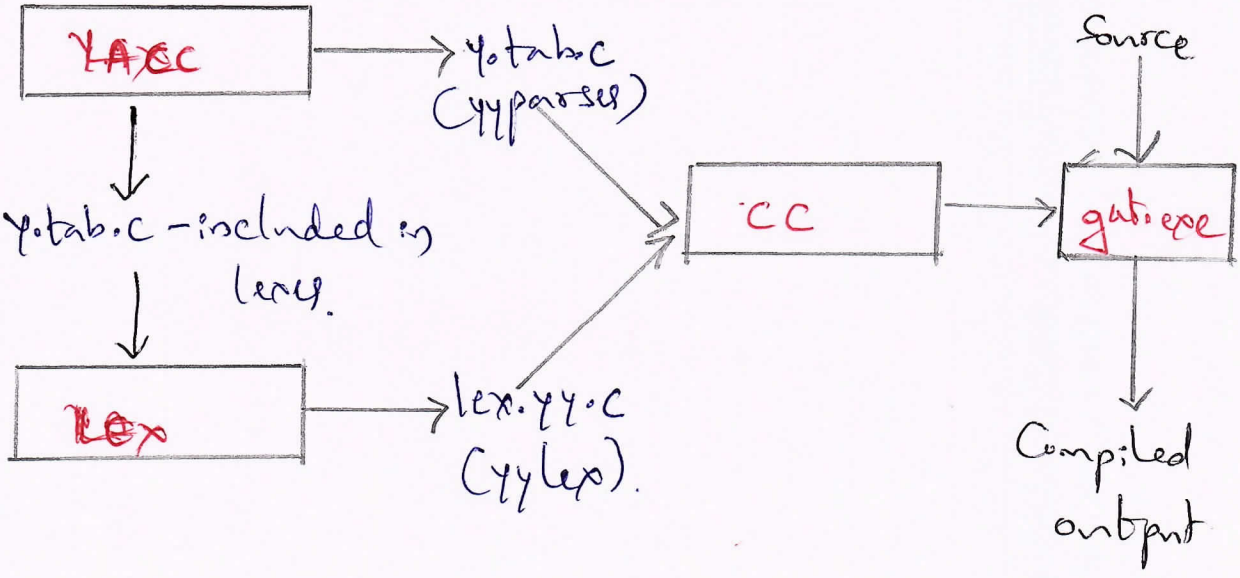
* Error :- Discover a syntax error & call an error recovery routine.

Eg :- id * id

STACK	I/P	ACTION
$	id * id $	shift
$ id	* id $	reduce F → id
$ F	* id $	Red T → F
$ T	* id $	shift
$ T *	id $	shift
$ T * id	$	red F → id
$ T * F	$	red T → T * F
$ T	$	red E → T
$ E	$	Accept

MODULE - 4

7. a) Explain the communication between parser & lexer with neat block diagram?



⇒ When we use a lex scanner & YACC parser together, Parser is higher level routine.

⇒ It calls the lexer yylex() whenever it needs a token from input.

⇒ The lexer then scans through the input recognizing tokens.

⇒ As soon as it finds a token of interest to parser returning the tokens code as value of yylex().

⇒ YACC can optionally write a C header file containing all of the token definitions.

⇒ We include the file y.tab.h in Lexer & use preprocessor symbols in lexes action code.

b) What is Regular Expression? Explain various RE with example?

⇒ Regular expression is a pattern description using meta-language which use to describe a particular pattern of interest.

⇒ The characters that form Regular Expressions are :-

. : Matches any single character except the newline character.

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

[ ] : A character class which matches any char with brackets. If the first char is circumflex ("^") it changes the meaning to.

match any characters except ones within the brackets. [0-9] , [a-zA-Z].

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

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

{} : 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 characters & 'c' escape sequences. eg:- [\n] , [\c].

+ : Matches one or more occurences of preceding Regular expression. eg:- [0-9]+

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

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

"..." : Interprets everything within the quotation masks literally. eg:- "abc".

/ : Matches preceding RE but only if followed by the following RE. eg:- 0/1.

() : Groups a series of RE together into a new RE.

Eg:- (01)

c) Write a Lex program to count number of vowels & consonants in a given string.

```
%{ #include<stdio.h>
   int v=0, c=0;
%}
%%
[aeionAEIOU]    {v++;}
[bcdfghjklmnpqrstvwxyz]   {c++;}
.   ;
%%
main()
{
   printf("Enter the i/p string: ");
   yylex();
   printf("No. of vowels are : %d &
          Consonants are : %d ", v, c);
}
```

8. a) Explain the structure of YACC Program.

⇒ There are 3 sections in YACC program.

*DEFINITION SECTION:-

⇒ It has a literal code block enclosed in %{ & %}. The header files must be included in this section.

⇒ Later LEX copies the material bet^n %{ & %} directly to the generated 'C' file, so we may can write any valid 'C' code here.

⇒ The %% marks the end of this section.

*Rules Section :- Each rule is made up two parts : a pattern & action separated by whitespace.

⇒ 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 indicating the end of rule.

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

*Subroutine section :- It begins after second %%. It can contain any 'C' code & copied into the resulting parser.

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

⇒ Eg :-

```
%{ #include<stdio.h>
   - - -
%}
%%
  st : A s B
     ;
  S : A
    | B
    ;
%%
Main()
{ pf("Enter string: ");
    yyparse();
}
```

```
yyerror()
{
  - -
}  - -
```

8.b) Write a YACC program to recognize an arithmetic expression involving operators +, *, -, /.

⇒ YACC PART :-

```
%{ #include<stdio.h>
   #include<stdlib.h>
%}
%token ID
%left '+' '-'
%left '*' '/'
%%
in : exp  {printf("Valid exp");
           exit(0); }
   ;
exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '(' exp ')'
    | ID
    ;
%%
main()
{ printf("Enter expression");
  yyparse();
}
yyerror()
{ printf("Invalid Expression"); exit(0); }
```

LEX PART :-

```
%{ #include"y.tab.h"
%}
%%
[a-zA-Z]+  {return ID;}
.          {return yytext[0];}
%%
```

8c) What is Shift/Reduce parsing ? Explain with an example ?

⇒ When YACC processes a parser, it creates a set of states each of which reflects a possible position in one or more partially parsed rules.

⇒ The parser reads a token, each time it reads a token that doesn't complete a rule it pushes the token on an internal stack & switches to a new state. This is called SHIFT.

⇒ When it has found all symbols that RHS, it pops the RHS symbols off the stack, pushes the LHS symbol onto the stack & switches to a new state. This action is called REDUCTION.

⇒ Eg:- "fred = 12 + 13".

parser starts by shifting token on to the internal stack one at a time.

fred.
fred =
fred = 12
fred = 12 +
fred = 12 + 13

⇒ It can reduce the rule "exp → NUM + NUM"
So it pops 12, the plus & the 13 from stack & replaces them with expression & fred = exp.

⇒ Now, it reduces the rule "stmt → NAME = exp",

it pops fred, =, & expr & replaces them with ~~expr~~ start.

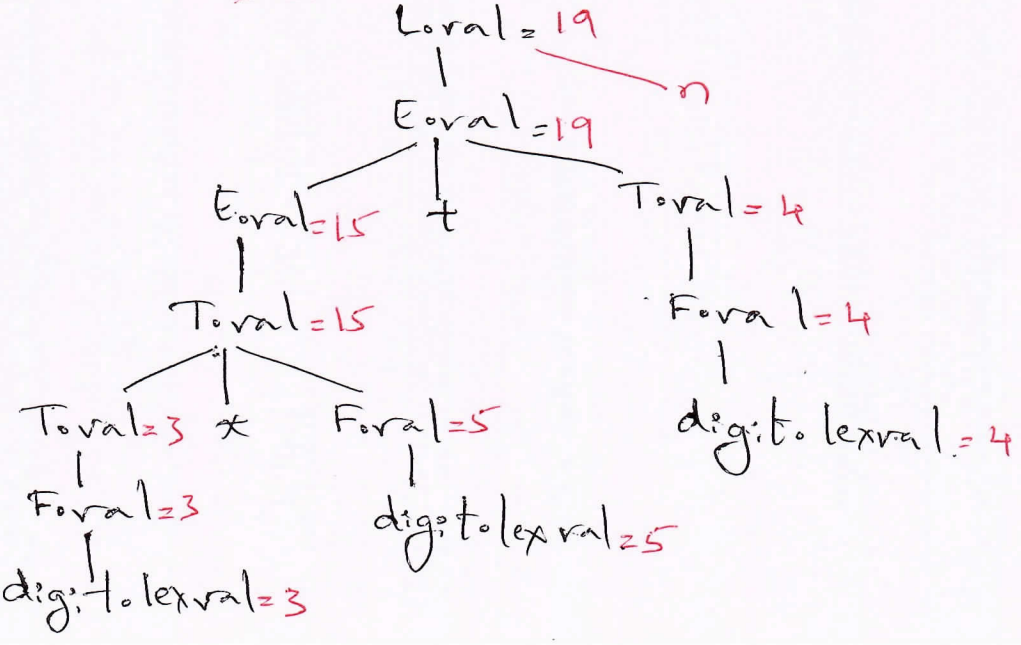
⇒ We have reached the end of the input & stack has been reduced to the start symbol. ∴ It is a valid string.

MODULE-5

9.a) Give a SDD to procces a simple desk calculator & show annotated parse tree for expression. 3*5+4n;

	PRODUCTION	SEMANTIC RULE
1)	L → E.n	L.val = E.val
2)	E → E.+T	E.val = E.val + T.val
3)	E → T	E.val = T.val
4)	T → T*F	T.val = T.val * F.val
5)	T → F	T.val = F.val
6)	F → (E)	F.val = E.val
7)	F → digit	F.val = digit.lexval

Annotated Parse Tree :-

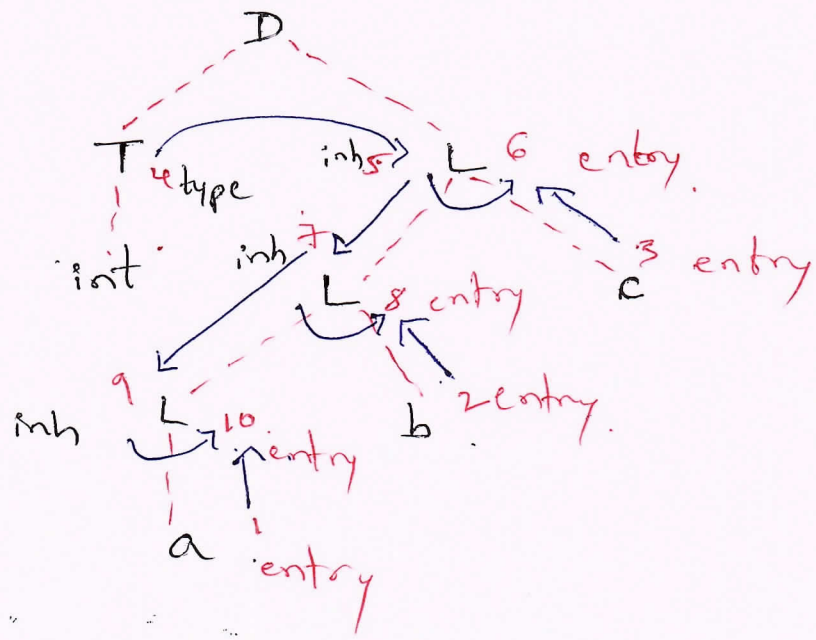


b) Give SDD for simple variable declaration in 'C' & construct a dependency graph for the following expression. int a,b,c ;

⇒ Grammar :- D → TL
T → int / float
L → L1 , id
L → id

Production	Semantic Rules
1) D → TL	L.inh = T.type
2) T → int	T.type = integer
3) T → float	T.type = float
4) L → L1 , id	L1.inh = L.inh addType(id.entry , L.inh)
5) L → id	addType(id.entry , L.inh)

Dependency graph for "int a,b,c"



10. a) Discuss the issues in the design of a code generator.

⇒ The issues in the design of code generator are :-

* Input to code Generator :-

⇒ It is a intermediate representation of source program produced by front end : in the symbol table, used to determine run-time addresses of data objects. by the IR.

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

* TARGET PROGRAM :-

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

⇒ RISC m/c typically has many registers, 3-address inst^n, simple addressing modes & simple inst^n. set architecture.

⇒ CISC has few registers, 2-address inst^n, a variety of addressing modes, several reg classes, variable length inst^n.

⇒ Stack-based machine, operations are done by pushing operands onto a stack & then performing the operations on operands at top of stack.

⇒ It is revived with JVM.

*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:- $x = y + 2$.

```
LD   R0, y
ADD  R0, R0, 2
ST   x, R0
```

⇒ The quality of generated code is determined by its speed & size. On most m/c, a given IR program can be implemented by different code sequences.

*REGISTER ALLOCATION :-

⇒ Registers are fastest computational unit on target machine.

⇒ The use of registers is divided into 2 subproblems:

i) Register allocation : we select set of variables will reside in register at each point of program.

ii) Register assignment : we pick the specific reg that a variable resides in.

* EVALUATION ORDER :-

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

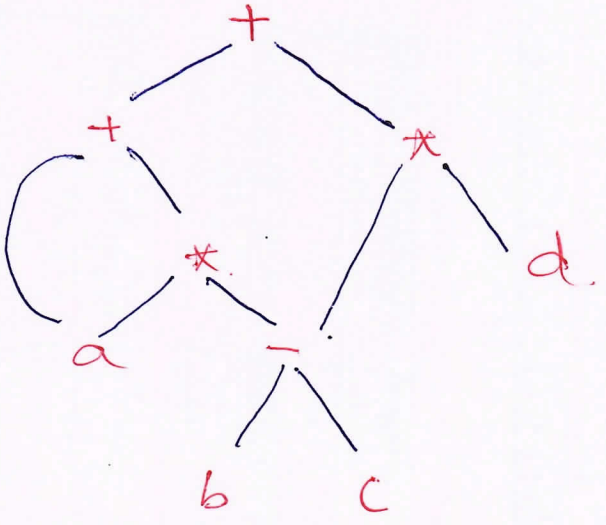
⇒ Some computation orders require fewer registers to hold intermediate results.

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

b) Obtain the DAG for expression : "a+a*(b-c)+(b-c)*d.
Also give sequence of steps for constructing same.

* The steps for constructing the DAG are :-

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}).



* * * * *