* Design of Language Processors. * 20/12/2010

Que: I. (b) Explain First three phases and error handling of a compiler. Give appropriate examples. [06]

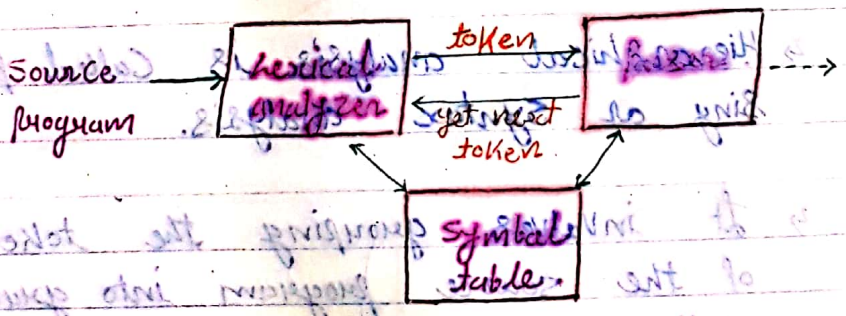
Ans:

The following are the 1st three phases of compiler.

I. Lexical Analyzer
II. Syntax Analyzer
III. Semantic Analyzer

I. Lexical Analyzer :-

- The Lexical Analyzer is the 1st phase of a Compiler., is also called scanning.
- It's main task is to read the i/p characters and produce as output a sequence of tokens that the parser uses for syntax analysis.
- The following is the interaction of lexical analyzer with parser.



2) The lexical analyzer may also perform certain secondary tasks at the user interface.

I) is stripping out from the source program comments & white space in the form of blank, tab & newline characters.

II) Another is correlating error messages from the compiler with the source program.

For example:

Position := intial + rate * 60.

$id_1 := id_2 + id_3 * 60$

2) Reads the characters in the source program & groups them into a stream of tokens.

II) Syntax Analyzer :-

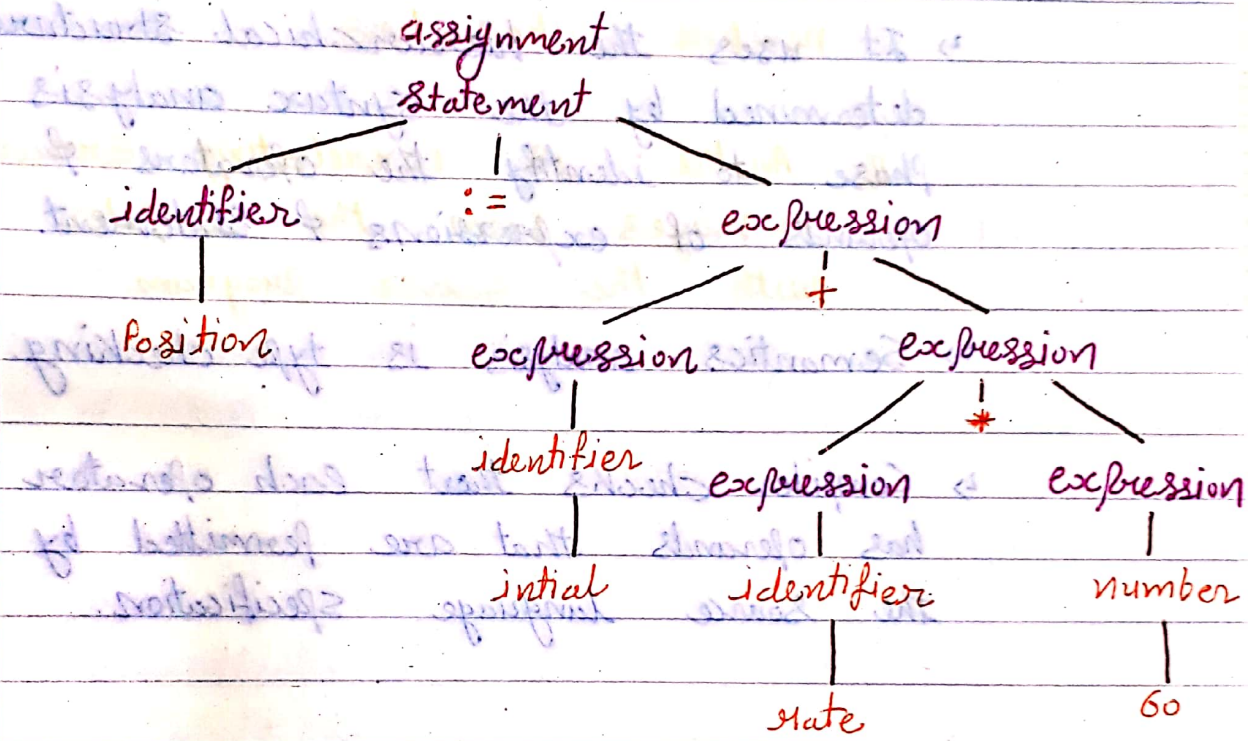
2) Hierarchical analysis is called parsing or syntax analysis.

2) It involves grouping the tokens of the source program into grammatical phrases.

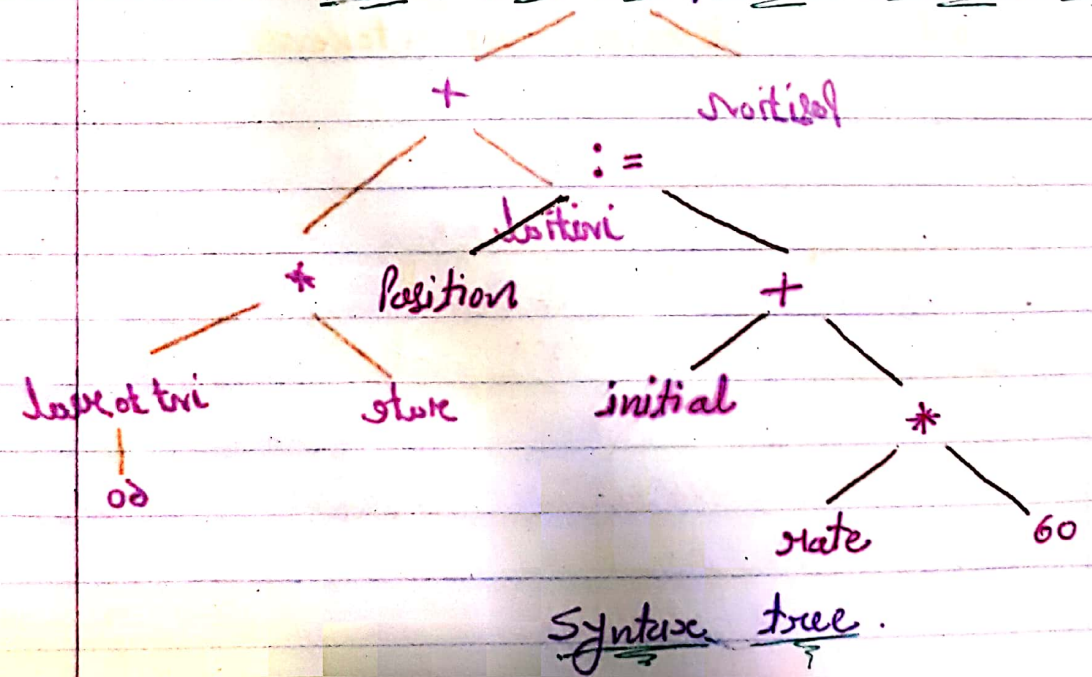
↳ Syntax analysis checks the statement is correct or not.

↳ It is also called as parsing.

↳ For example:



Parse tree for position := initial + rate * 60



Syntax tree.

III. Semantic Analysis :-

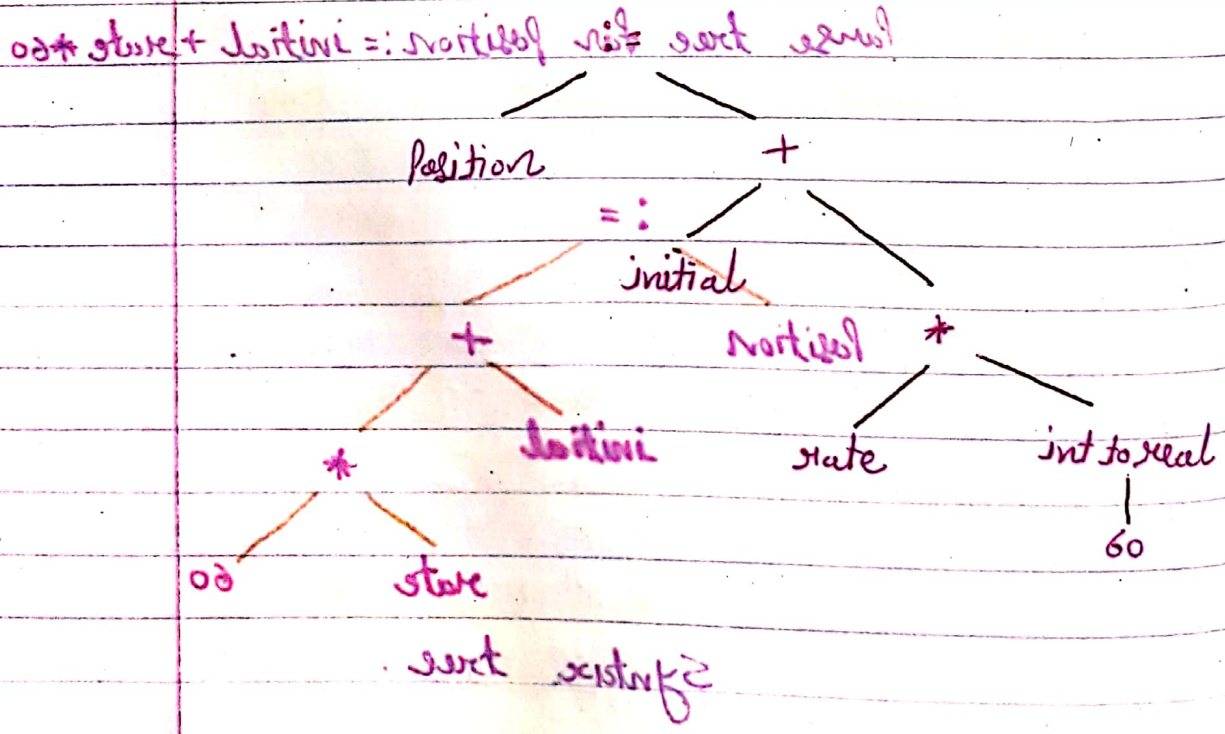
↳ Semantic analysis phase checks the source program for semantic errors & gathers type information for the subsequent code-generation phase.

↳ It uses the hierarchical structure determined by the syntax analysis phase to identify the operators & operands of expressions & statement.

↳ Semantics analysis is type checking.

↳ Compiler checks that each operator has operands that are permitted by the source language specification.

↳ For example :



* Error Handling :

⇒ Each phase can encounter errors. after detecting an error, a phase must some how deal with that error,

⇒ The syntax & semantic analysis phases usually handle a large fraction of the errors detectable by the Compiler.

⇒ The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.

⇒ For example :

⇒ If the string fi is encountered in a c program for the first time in the context

```
fi (a == f(x)) ...
```

a lexical analyzer cannot tell whether fi is a misspelling of the keyward if or an undeclared function identifier.

⇒ This type of error can be detected by Syntax analysis phase.

Que: 2. (a) State various possible implementation for a symbol table. Explain the most suitable implementation.

Ans:

⇒ A compiler uses a symbol to keep track of scope & binding information about names.

⇒ The symbol table is searched every time a name is encountered in the source text.

⇒ A symbol table mechanism must allow us to add new entries & find existing entries efficiently.

⇒ There are two types of implementation for a symbol table:

I. Linear list

II. Hash table

I. Linear list :-

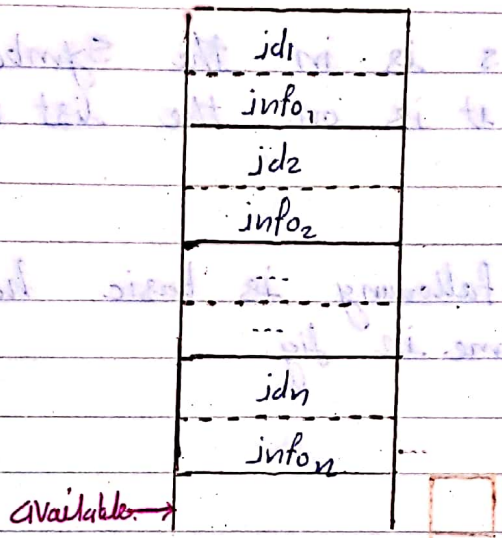
Linear list is the simplest to implement, but its performance is poor when we want to get large.

II. Hashing :-

⇒ Hashing schemes provide better performance for somewhat greater programming effort & space overhead.

* Symbol Table:

→ The simplest & easiest to implement data structure for a symbol table is a linear list of records. Shown in fig.



→ New names are added to the ~~Store names & their~~ list in the order in which they are encountered.

→ The position of the end of the array is marked by the pointer available. pointing to where the next symbol table entry will go.

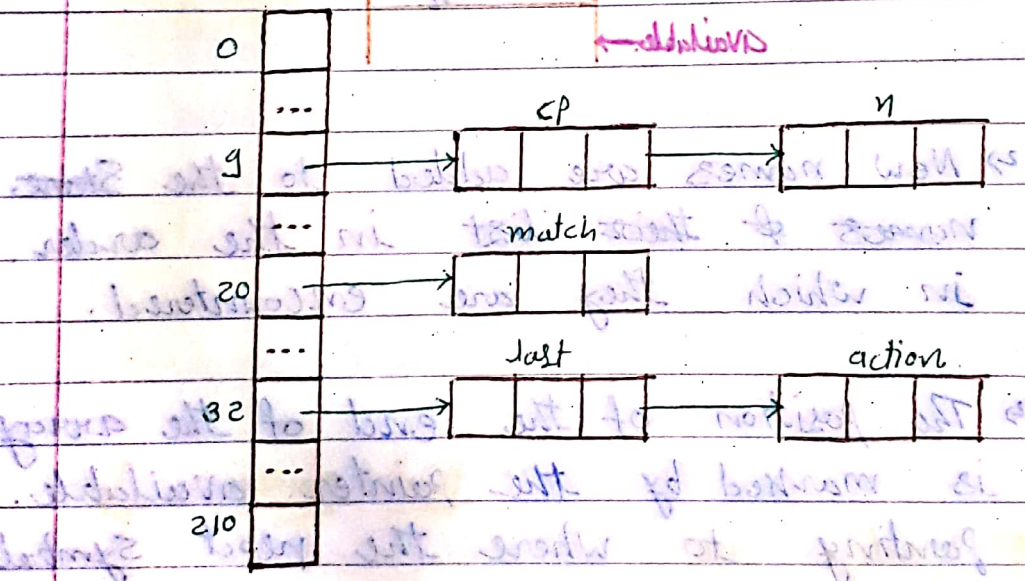
→ The search for a name proceeds backwards from the end of the array to the beginning.

Hash Tables:-

2) An entry for string s in the symbol table, we apply a hash function h to s, such that h(s) returns an integer between 0 & m-1.

2) If s is in the symbol table, then it is on the list numbered h(s).

2) The following is basic hashing scheme in fig.



Que: 3 (a) Describe of parse tree, Syntax tree and annotated + parse tree by giving ex.

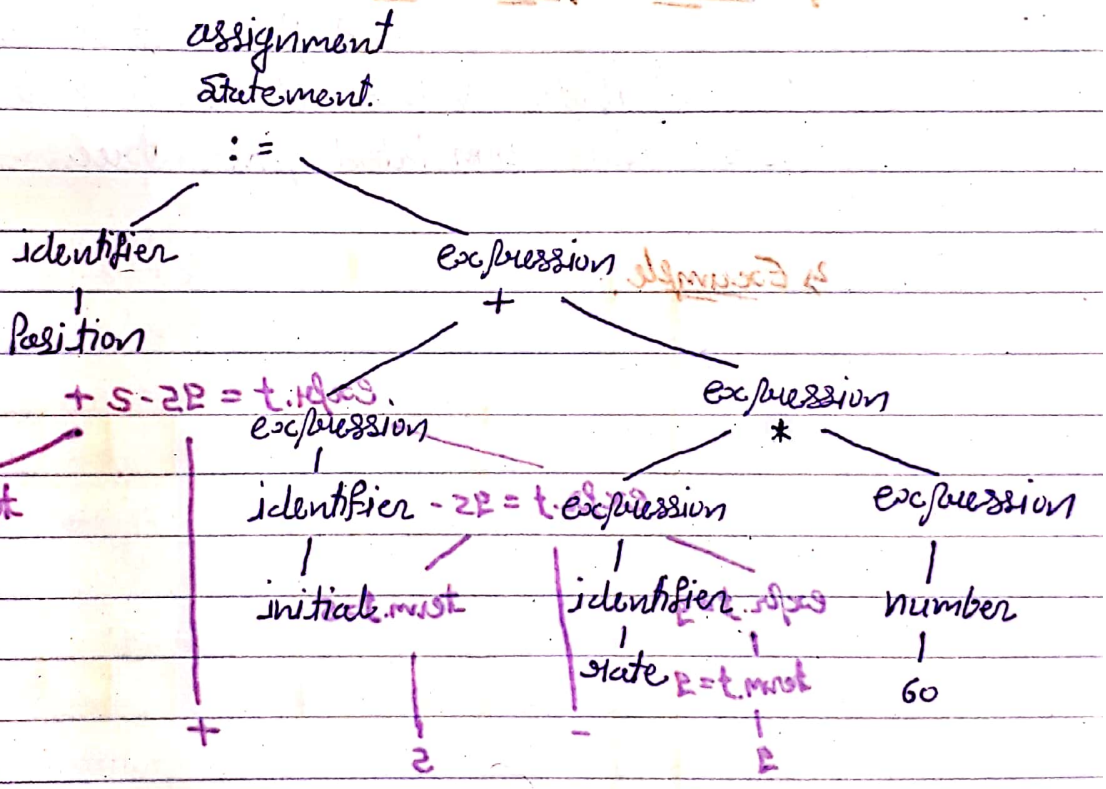
Ans:

* Parse Tree :-

A Parse tree, the start Symbol of a grammar derives a String in the language.

2) Example: Parse tree for

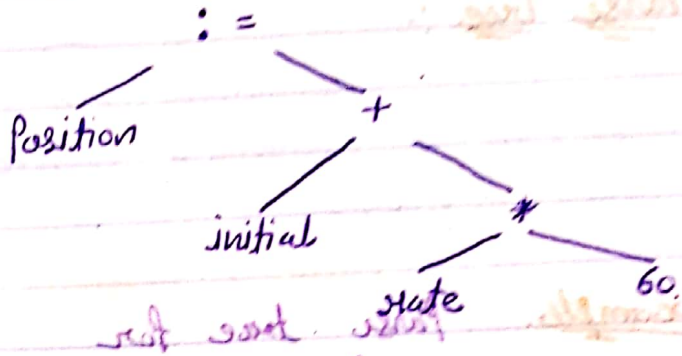
position := intial + rate * 60



* Syntax Tree :-

A Syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, & the operands of an operator are the children of the node for that operator.

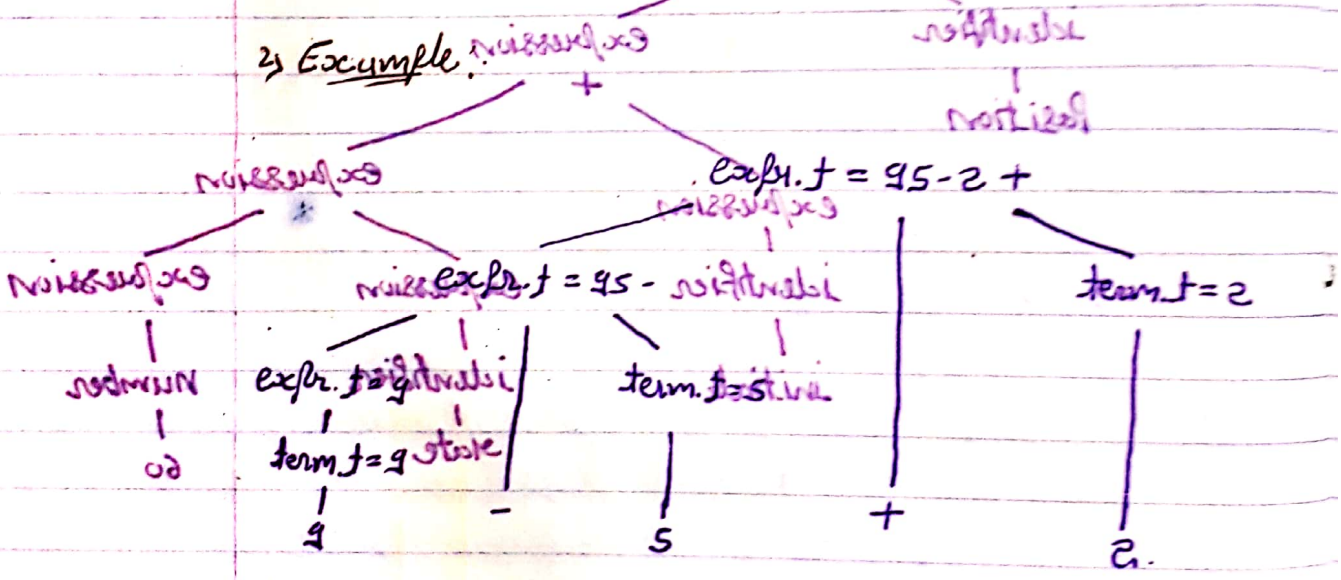
↳ Example: Syntax tree for
position := initial + rate * 60



Annotated Parse Tree:

A parse tree showing the attribute values at each node is called an annotated parse tree.

↳ Example:



OR.

Que:-3 (a) Explain Simple LR Parser.

Ans:-

2) The 1st method is "Simple LR" or SLR for short in LR Parser.

2) An SLR Parser can be constructed is said to be an SLR grammar.

2) The other two methods augment the SLR method with lookahead inf^n,

2) An LR(0) item of a grammar G is a production G with a dot at some position of the right side.

2) A → ·XYZ (no. of the prod^n)
A → X·YZ (position of the dot)
A → XY·Z
A → XYZ·
A → G generates A → ·

2) The Central idea in the SLR method is 1st to construct from the grammar a deteremistic finite auto-maton to recognize Variable prefixe

2) If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' & production S'→S.

2) The purpose of this new starting production is to indicate to the

parser when it should stop parsing & announce acceptance of the i/p.

⇒ Acceptance occurs when & only when the parser is about to reduce by $S' \to S$.

* The closure operation:-

⇒ If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

I. Initially, every items in I is added to closure(I).

II. If $A \to \alpha . B\beta$ is in closure(I) & $B \to \gamma$ is a production, then add the item $B \to .\gamma$ to I, if it is not already there. We apply this rule until no more new items can be added to closure(I).

* The Goto operation:-

⇒ 2nd function is goto(I, X) where I is a set of items & X is a grammar symbol.

⇒ goto(I, X) is defined to be the closure of the set of all items $[A \to \alpha X . \beta]$

* Example:- $S \to aSb$
$S \to ab$
perform SLR parsing table.

Ans:-

Step-I:- Augmented grammar.

0 $S' \to S$
1 $S \to aSb$
2 $S \to ab$

Step-II:- Closure operation.

I_0 = Closure ({[$S' \to .S$]})

$= \left\{ \begin{array}{l} S' \to .S \\ S \to .aSb \\ S \to .ab \end{array} \right.$

goto (I_0, S) = Closure [$S' \to S.$]
$= S' \to S.$
$= I_1$

goto (I_0, a) = Closure ({[$S \to a.Sb$]}, ({[$S \to a.b$]})

$= S \to a.Sb$
$S \to a.b$
$S \to .aSb$
$S \to ab$
$= I_2$

goto (I_2, S) = Closure ({[S → aS · b]})

= S → aS · b

= I_3

goto (I_2, b) = Closure ({[S → a · b]})

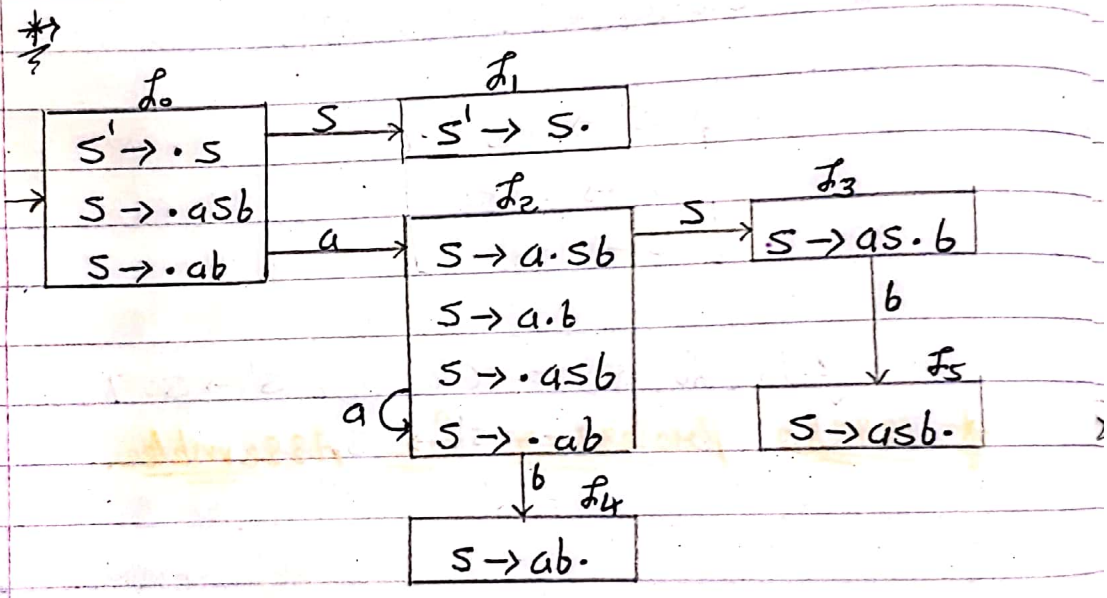
= S → ab ·

= I_4

goto (I_3, b) = Closure ({[S → asb ·]})

= S → asb ·

= I_5



* follow (S) = { $, b }

	Action			goto
	a	b	$	S
0	S_2			1
1			acc.	
2	S_2	S_4		3
3		S_5		
4		r_2	r_2	
5		r_1	r_2.	

OR.

Que:3 (b) Explain macros & macro processors for the assembler

Ans:

Macro :-

A macro is a unit of specification for program generation through expansion.

2) A macro consist of a name, a set of formal parameters & a body of code.

3) Two kinds of expansions can be readily identified:

I. lexical expansion

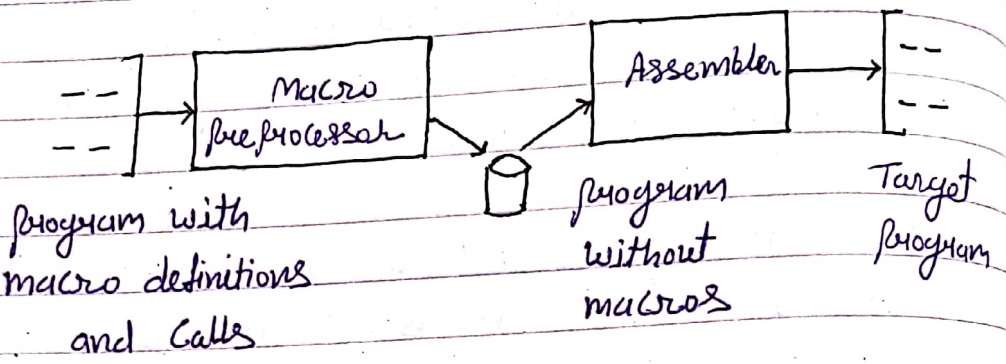
II. Semantic expansion.

MACRO processor for Assembler:-

2) The macro preprocessor accepts an assembly program containing definitions of & calls & translates it into an assembly program which does not contain any macro def^n or calls.

2) The macro preprocessor segregates macro expansion from the process of program assembly.

2) Design overview of a macro pre-processor by listing all tasks involved in macro expansion.



I. Identify macro Calls in the program.

2) A table called the macro name table (MNT) is designed to hold the name of all macros defined in a program.

2) A macro name is entered in this table when a macro definition is processed.

II. Determine the value of formal parameters:

2) A table called the actual parameter table (APT) is designed to hold the values of formal parameters during the expansion of a macro call.

(<formal parameter>, <value>)

↳ Two items of information are needed to construct this table, names of formal parameters, & default values of keyword parameters, a table called parameter default table (PDT)

↳ This table accessible from the MNT entry of a macro & would contain pairs of the form

(<formal parameter name>,
<default value>).

III. Maintain expansion time Variables:-

↳ An expansion time Variables' table (EVT) is maintained for this purpose. The table contains pairs of the form

(<EV_name>, <value>)

↳ The value field of a pair is accessed when a preprocessor stmt or a model stmt under expansion refers to an EV.

IV. Organize expansion time control flow:-

↳ The body of macro, i.e. the set of preprocessor statements & model stmts in it, is stored in a table called the macro definition table (MDT) for use during macro expansion.

2. MEC is initialized to the 1st stmt of the macro body in the MDT.

V. Determine Values of sequencing Symbols:

1. A sequencing Sybol Table (SST) is maintained to hold this information.

2. The table Contains pairs of the form

(<sequencing symbol name>, <MDT entry #>)

Where <MDT entry #> is the no. of the MDT entry which contains the model stmt defining the sequencing symbol.

VI. Perform expansion of model stmt.

1. MEC points to the MDT entry containing the model stmt.

2. Values of formal parameters & EV's

3. The model statement defining a sequencing symbol can be identified from SST.

Que:- 4: (b) Write unambiguous production rules for if then else construct & parse following statement using shift-reduce parser:

if E1 then if E2 then S2 else if E3 then S3 else S4.

Ans:-

Unambiguous Production Rule's

Stmt → matched_Stmt
| unmatched_Stmt

matched_Stmt → if expr then matched_Stmt else matched_Stmt
| other

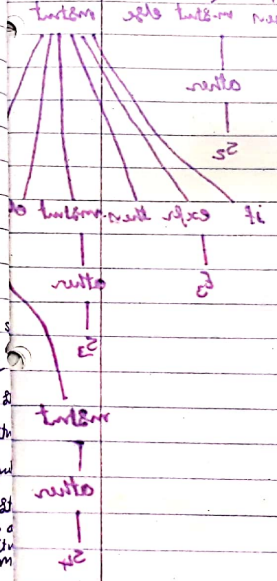
unmatched_Stmt → if expr then Stmt
| if expr then matched_Stmt else unmatched_Stmt

Shift Reduce Parser's

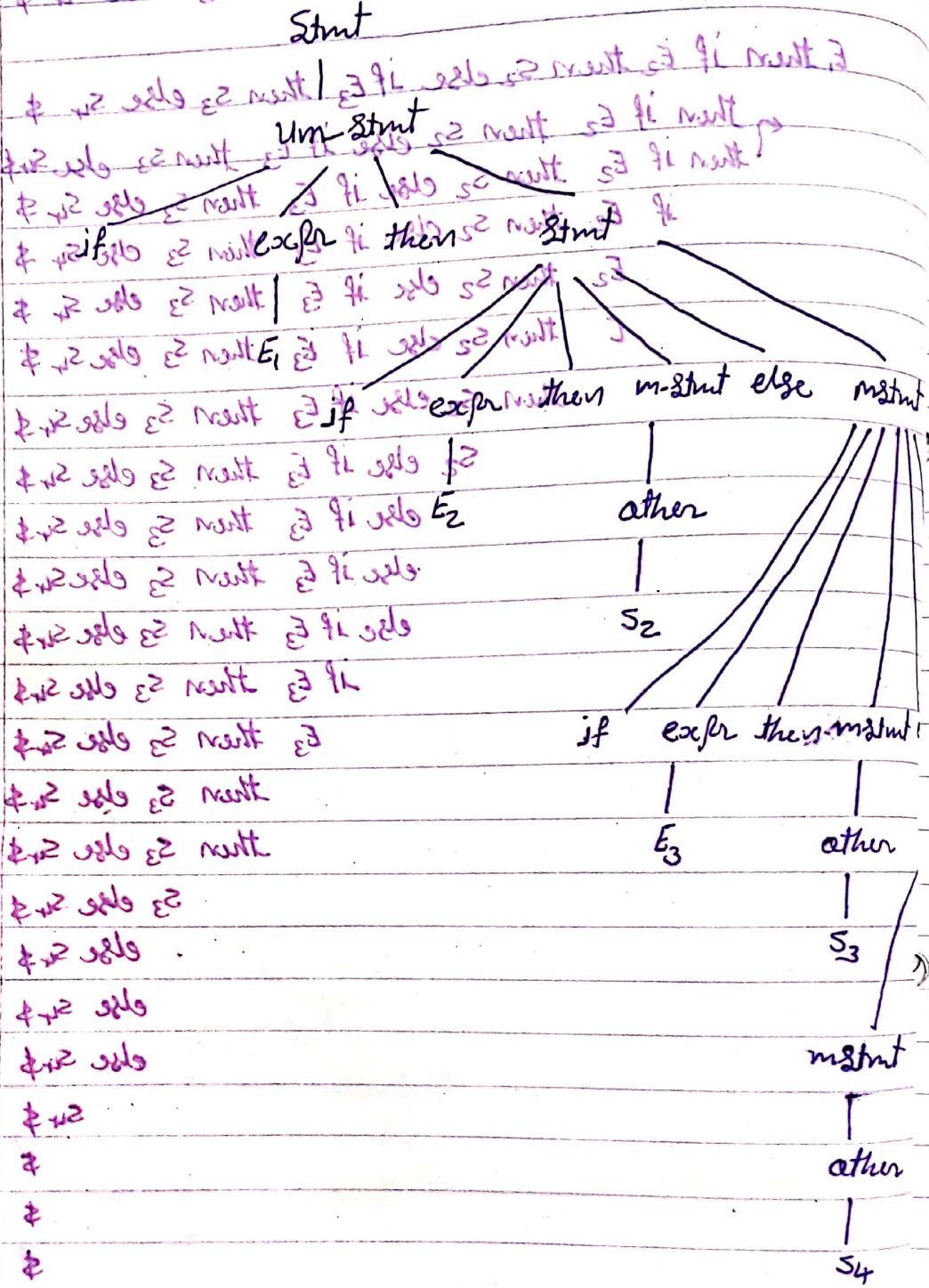
Given String is

if E_1 then S_1 if E_2 then S_2 else if E_3 then S_3 else S_4

Stack	String	Action
$	$ if E_1 then if E_2 then S_2 else if E_3 then S_3 else S_4 $	Shift
$ if	E_1 then if E_2 then S_2 else if E_3 then S_3 else S_4 $	Shift
$ if E_1	then if E_2 then S_2 else if E_3 then S_3 else S_4 $	Reduce
$ if E_1 then	then if E_2 then S_2 else if E_3 then S_3 else S_4 $	Shift
$ if E_1 then if	if E_2 then S_2 else if E_3 then S_3 else S_4 $	Shift
$ if E_1 then if E_2	E_2 then S_2 else if E_3 then S_3 else S_4 $	Shift
$ if E_1 then if exp	C then S_2 else if E_3 then S_3 else S_4 $	Reduce
$ if E_1 then if exp then	then S_2 else if E_3 then S_3 else S_4 $	Shift
$ if E_1 then if exp then S_2	S_2 else if E_3 then S_3 else S_4 $	Shift
$ if E_1 then if exp then other	else if E_3 then S_3 else S_4 $	Reduce
$ if E_1 then if exp then m-stmt	else if E_3 then S_3 else S_4 $	Reduce
$ if E_1 then if exp then m-stmt else	else if E_3 then S_3 else S_4 $	Shift
$ if E_1 then if exp then m-stmt else if	if E_3 then S_3 else S_4 $	Shift
$ if E_1 then if exp then m-stmt else if E_3	E_3 then S_3 else S_4 $	Shift
$ if E_1 then if exp then m-stmt else if exp	then S_3 else S_4 $	Reduce
$ if E_1 then if exp then m-stmt else if exp then	then S_3 else S_4 $	Shift
$ if E_1 then if exp then m-stmt else if exp then S_3	S_3 else S_4 $	Shift
$ if E_1 then if exp then m-stmt else if exp then other	else S_4 $	Reduce
$ if exp then if exp then m-stmt else if exp then m-stmt	else S_4 $	Reduce
$ if exp then if exp then m-stmt else if exp then m-stmt	else S_4 $	Shift
$ if exp then if exp then m-stmt else if exp then m-stmt else	S_4 $	Shift
$ if exp then if exp then m-stmt else if exp then m-stmt else m	$	Reduce
$ if exp then if exp then m-stmt else m-stmt	$	"
$ if exp then m-stmt	$	"
$ if exp then stmt	$	"
$ m-stmt	$	"
$ stmt	$	Accept.



Parse tree:



Que:- 4. b. Explain & Justify whether following types of files are relocatable & executable:
.obj, .com & .exe.

Ans:-

* .obj :-

1) The Microsoft translator produces object modules; these object modules are stored in a file having extension .obj.

2) Each object module contains binary coding of the translated instruction & data plus information about all the segments.

* .Exe :- (Contain relocatable program)

1) The MS-DOS LINK is a linkage editor which binds all these object modules together to form an executable program. This executable program has an extension .exe

* .COM :- (Contain Non-relocatable program)

2) MS Dos Contains a program EXE2BIN which Converts a .EXE program into a .COM program.

3) When the user types a filename with the .COM extension, the absolute loader is involked to simply load the obj* prog. into m

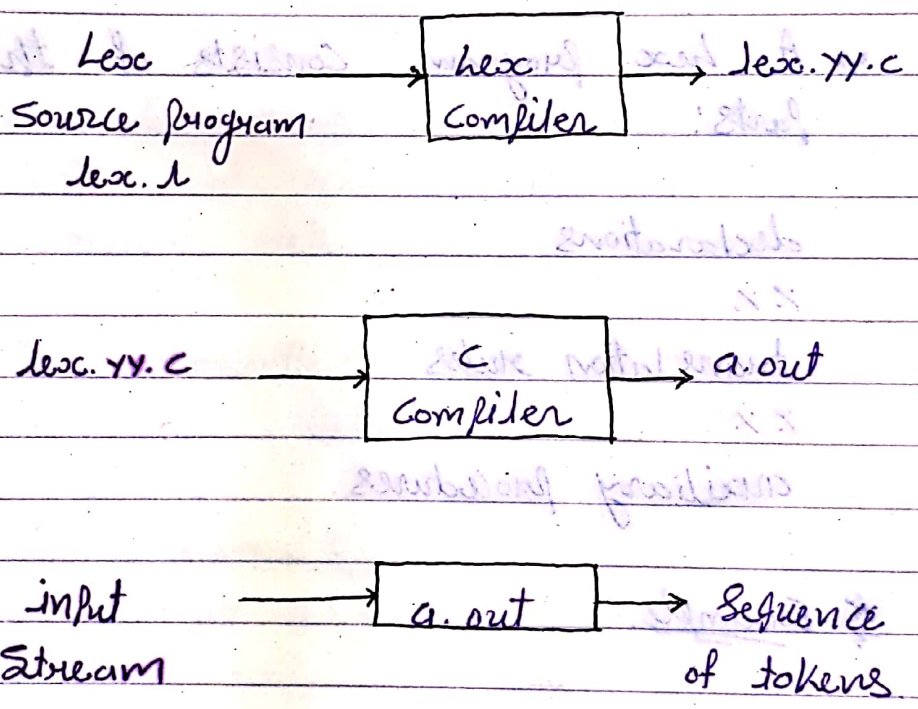
Que:- 5. (a) Explain tools - lex & YACC

Ans:-

LEX:-

1) LEX, that has been widely used to specify lexical analyzers for a variety of Language.

2) Lex is generally used in the manner depicted in following fig.



Creating a lexical analyzer with lex

1) First, a specification of a lexical analyzer is prepared by creating a program lex.l in the lex language.

2) Lex.l is run through the lex Compiler to produce a C program lex.yy.c

2) The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expressions of lex.l

3) lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer

★ Lex specification.

1) A lex program consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures.

★ Example:

```
%{
letter        [A-Za-z]
digit         [0-9]
%}
%%
begin         {return(BEGIN);}
end           {return(END);}
":="          {return(ASSGOP);}
{letter}({letter}|{digit})*   {yylval = enter_id();
                               return(ID);}
```

{digit}+ {yylval = enter_num();
return (NUM); }

%%

enter_id()
{ /* enters the id in the symbol
table & return entry number */
}

enter_num()
{
/* enters the num. in the Constants
table & return entry number */
}

2) The declarations section includes declarations of variables, manifest constants, & regular definitions.

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

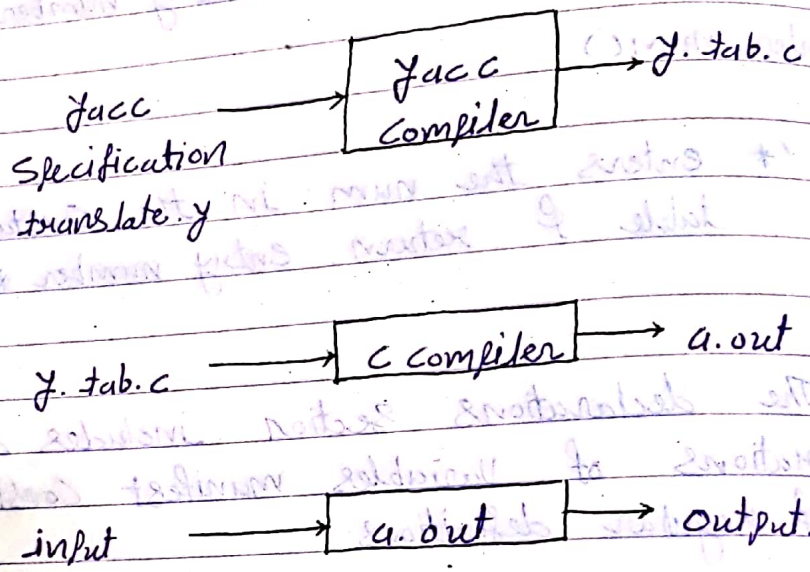
P1 {action 1}
P2 {action 2}
. . . . . .
Pn {action n}

Where each Pi is a RE & each action is a program fragement.

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

YACC: (Yet another Compiler Compiler)

2) A translator can be constructed using YACC in the manner illustrated in fig.



Creating an i/p/o/p translator with yacc

2) First, a file, say translate.y containing a Yacc specification of the translator is prepared.

2) The UNIX System Command yacc translate.y transforms the file translate.y into a C program called y.tab.c using the LALR method.

2) The object program a.out that performs the translation specified by the original YACC grammar.

Syntax:-

→ A yacc source program has three parts:-

declarations
%%
translation rules
%%
supporting C-routines.

Example:-

```
%{
#include <ctype.c>
%}
%token DIGIT
%%
<line> ::= <expr> | <integer> {printf("%d\n", $1);}
        ;
expr : expr '+' term { $$ = $1 + $3; }
     | term
     ;
term : term '*' factor { $$ = $1 * $3; }
     | factor
     ;
factor : '(' expr ')'  { $$ = $2; }
       | DIGIT
       ;
%%
yy lex()
{
        int c;
```

```
c = getchar();
if (isdigit(c))
{
    yylval = c - '0';
    return DIGIT;
}
return c;
}
```

2) The declarations part of a yacc program contains two section.

2) The translation rules part: Each rule consists of a grammar prod^n & the associated semantic action.

2) Set of production:

<left side> → <alt 1> | <alt 2> | ... | <alt n>

2) YACC semantic action is a sequence of C statements.

Que: 5. (b) Explain Various parameter passing technique.

Ans:-

=> There are two types of parameter
   i. Formal parameters
   ii. Actual parameters.

=> Based on these parameters there are various parameter passing methods, the most common methods are:

1. Call by value:-

=> This is the simplest method of parameter passing.

=> The actual parameters are evaluated & their r-values are passed to called procedure.

=> The operations on formal parameters do not changes the values of actual parameter.

* Example:-

- Languages like C, C++ use actual parameter passing method.
- In PASCAL the non-Var parameters

II. Call-by-reference:

2) This method is also called as call by address or call by location.

2) The L-value, the address of actual parameter is passed to the called routines activation record.

2) The values of actual parameters can be changed.

2) The actual parameters should have an L-value.

★ Example:

→ Reference parameters in C++, PASCAL's Var parameter.

III. Copy restore:

2) This method is a hybrid between call by value & call by reference.

★ Example:

2) This method is also known as copy-in copy-out or values result.

2) The calling procedure calculates the value of actual parameters & it is then copied to activation record for the called procedure.

2) During execution of called procedure, the actual parameters value is not affected.

2) If the actual parameter is L-value then at return the value of formal parameter is copied to actual parameter.

★ Example:

- In Ada this parameter passing is used.

III. Call by name:

2) This is less popular method of parameter passing.

2) Procedure is treated like macro. The procedure body is substituted for formals.

2) The actual parameters can be surrounded by parenthesis to preserve their integrity.

2) The local names of called procedure & name of calling procedure are distinct.

★ Example:

- ALGOL uses call by name method.

* Example:

```
procedure exchange (m, n : integer);
var t : integer;
begin
    t := m;
    m := n;
    n := t;
end;
.....
j := 1
a[j] := 50 { a: array [1...10] of integer}
print (j, a[j]);
exchange { j, a[j] };
print (j, a[j]);
```

* output:

Call by Value		Call by reference		Copy restore		Call by name	
1	50	1	50	1	50	1	50
1	50	50	1	50	1	Error	

* Example:

OR

Que:-5. (a) Explain peep-hole optimization.

Ans:-

★ Definition :-

→ Peephole optimization is a simple & effective technique for locally improving target code.

→ This technique is applied to improve the performance of the target program by examining the short sequence of target instructions & replacing these instructions by shorter or faster sequence.

★ Characteristics of peephole optimization.

I. Redundant instruction elimination.

II. Flow of Control optimization.

III. Algebraic Simplification.

IV. use of machine idioms.

I. Redundant instruction elimination

→ Especially the redundant loads & stores can be eliminated in this type of transformations.

→ For example.

```
MOV R0, x
MOV x, R0
```

2> We can eliminate the second instruction since x is already in R0. But if (mov x, R0) is a label statement then we cannot remove it.

2> We can eliminate the unreachable instructions. For example, following is a piece of C code.

```
Sum = 0
if (sum)  /* eliminate - unreachable code */
  printf("%d", sum);
```

2> Similarly

```
int fun(int a, int b)
{
  c = a + b;
  return c;
  printf("%d", c); // unreachable code
}
```

hence eliminated.

II Flow of Control optimization :-

2> Using peephole optimization unnecessary jumps on jumps can be eliminated.

* Example:

```
goto test
...
test: goto done
...
done;
```

⇒

```
test: goto done
...
done;
```

2) Thus, we reduce one jump by this transformation

2) Another example is,

```
if a < b                    if a < b
    goto L1      ⟹             goto L2
L1: goto L2                 L1: goto L2
```

III. Algebraic Simplification:

2) Peephole optimization is an effective technique for algebraic simplification.

2) Example:

```
x := x + 0
x := x * 1
```

can be eliminated by peephole optimization.

IV. Reduction in Strength:

2) Certain m/c instructions are cheaper than the other. In order to improve the performance of the intermediate code we can replace these instructions by equivalent cheaper inst^{n}.

2) For example, x^2 is cheaper than $x*x$. Similarly '+' & '-' is cheaper than '*' & '/'.

So, we can effectively use equivalent addition & substraction for multiplication & division.

IV. M/c idioms:

2) The target instructions have equivalent m/c instructions for performing some operations.

2) Hence, we can replace these target instructions by equivalent m/c instructions in order to improve the efficiency.

2) For example, some m/cs have auto-increment or auto-decrement addressing modes that are used to perform add or subtract operations.

Que:- 1. (b) Explain Symbol table in detail.

Ans:-

2> Symbol table is a data structure used by Compiler to keep track of semantics of Variable.

2> That means symbol table stores the information about scope & binding inf^n about names.

2> Symbol table is built in lexical and syntax analysis phases.

2> The symbol table is used by various phases as follows

- Semantic analysis phase refers symbol table for type conflict issue.

- Code generation refers symbol table knowing how much run-time space is allocated &

★ l-value & r-value:-

2> The l & r prefixes come from left & right side assignment.

★ Example:-

a := j + 1

l-value ↑ a

r-value ↑ j + 1

* Symbol - Table Entries:

2) To achieve compile time efficiency compiler makes use of symbol table.

2) It associates lexical names with their attributes.

2) The items to be stored in symbol to table are:

i) Variable names
ii) Constants
iii) Procedure names
iv) Function names
v) Literal Constants & Strings
vi) Compiler generated temporaries
vii) Labels in Source Language.

2) Compilers uses following types of information from symbol table.

i) Data type
ii) Name
iii) Declaring procedure.
iv) offset in storage.
v) If structure or record then pointer to structure table.
vi) For parameter.
vii) Number & type of arguments passed to the fun^n
viii) Base address.

Example:

✱ Attribute of Symbol Table :-

1) Variable names
2) Constants
3) Data type
4) Compiler generated temporaries
5) Function names
6) Parameters names
7) Scope information.

Que: 2. (a) Define token, Pattern & lexeme. Differentiate between them using proper examples.

Ans:-

* **Tokens:** It describes the class or category of i/p string. For example, identifiers, keywords, constants are called tokens.

* **Patterns:** set of rules that describe the token.

* **Lexemes:** Sequence of characters in the source program that are matched with the pattern of the token. For example, int, i, num ans, choice.

* **Example:-**

```
Void swap(int i, int j)
{
        int t;
        t = i;
        i = j;
        j = t;
}
```

→ In the above program,

lexeme	Token
Void	Keyword
swap	identifier
(	operator
int	Keyword
i	identifier
,	operator
int	Keyword
j	identifier
)	operator
{	operator
int	Keyword
*t	identifier
*t	identifier
=	operator
*j	identifier
j	identifier
=	operator
i	identifier
i	identifier
=	operator
+	operator
}	operator.

* Patterns:

1. Identifier:

- It is Collection of letters.
- It is a Collection of alphanumeric characters.

- The first character of identifier must be a letter.

II- Operator:-

i. Operator can be arithmetic, logical, relational operators.
ii. The parenthesis are considered as operators.
iii. Comma is treated as separation operator.
iv. Assignment is denoted by operator.

III- Keyword:-

1) Keyword are special words to which some meaning is associated with.
2) int, void, are keywords for denoting data types.

Que:- (a) What is called ambiguous grammar? Explain your answer with proper examples.

Ans:-

* Ambiguous Grammar:-

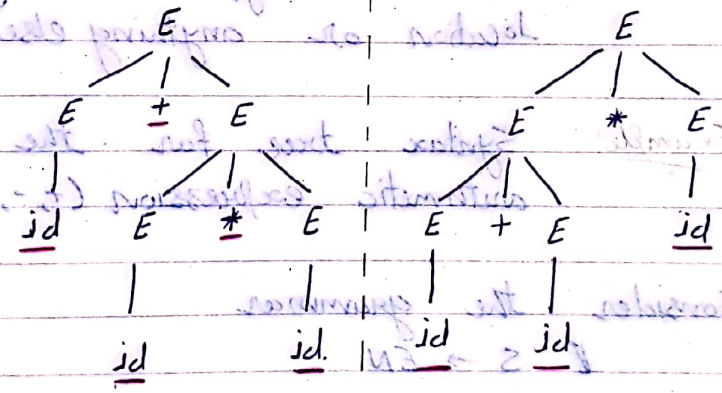
A grammar that produce more than one parse tree for the same sentence is said to be ambiguous.

* Example:-

1) Consider the following grammar.

$E \rightarrow E+E \mid E*E \mid (E) \mid id$

generate the parse tree for id + id * id.



2) We can get the more than one parse tree for this grammar. So we can say that this grammar is ambiguous grammar.

OR

Que:- 4. (a) Write Syntax directed definition for constructing a syntax tree for the arithmetic expression including +, -, *, / operators.

Ans:-

* Syntax Directed Definition:-

A syntax directed def^{n} is a generalization of grammar in which each grammar syntax has an associated set of attributes, partitioned into two subsets called the synthesized & inherited attributes of that grammar symbol.

2) Attribute:- It can be a string, a number, a type, a memory location or anything else.

* Example:- Syntax tree for the arithmetic expression (+, -, /)

2) Consider the grammar

S → EN

E → E + T

E → E - T

E → T

T → T * F

T → T / F

T → F

F → (E)

F → digit

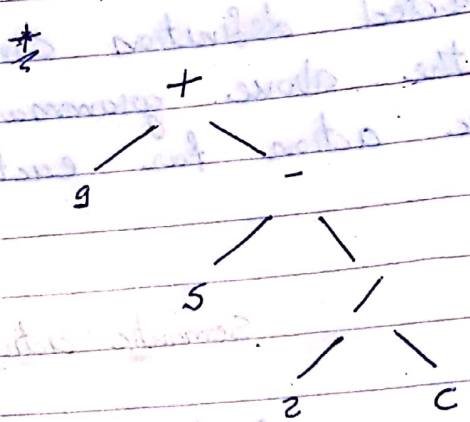
N → ;

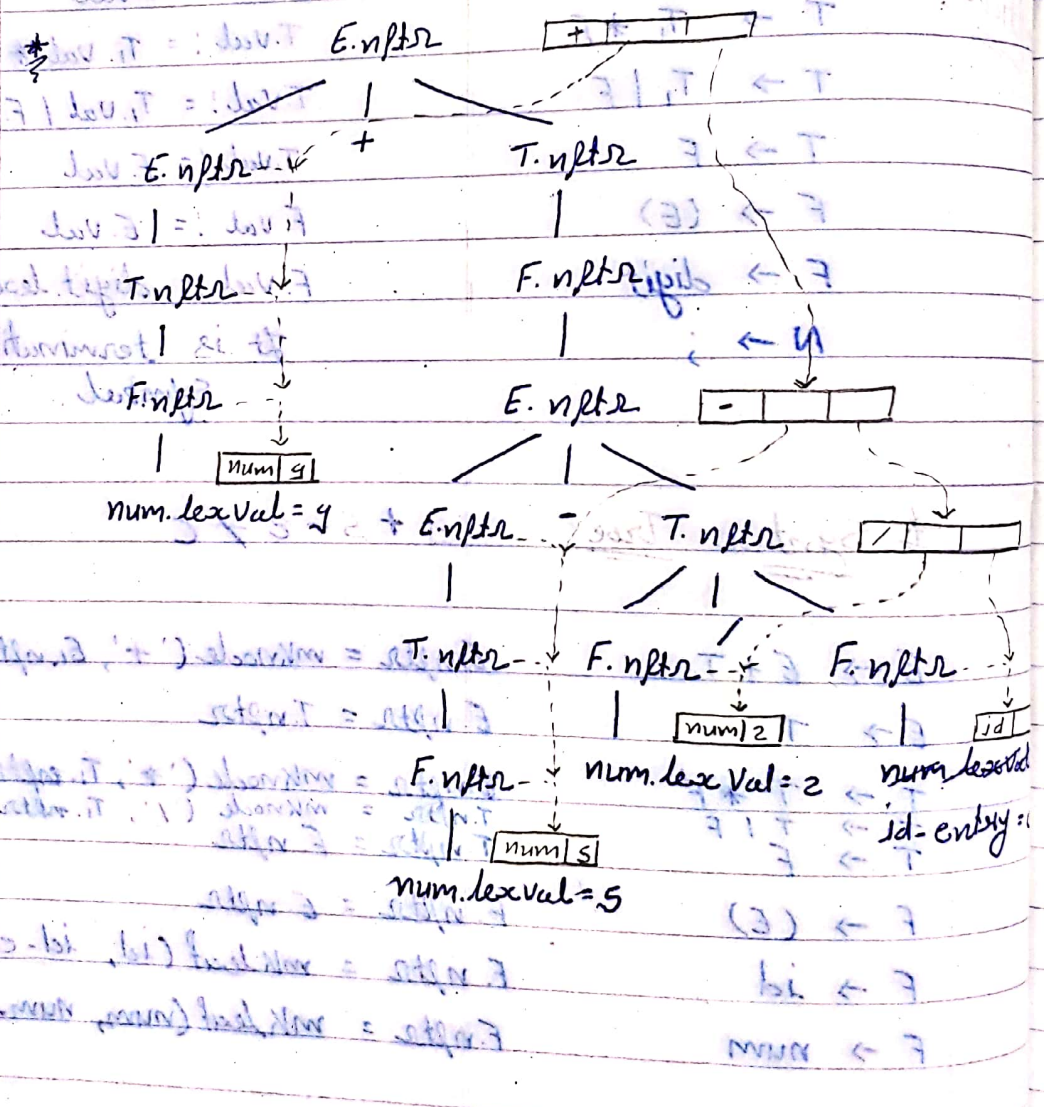
→ The Syntax-directed definition can be written for the above grammar by using semantic action for each production.

Production Rule.	Semantic actions.
$S \rightarrow EN$	print (E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow E_1 - T$	E.val := E_1.val + T.val
optional ⇒ $E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow T_1 / F$	T.val := T_1.val / F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ digit	F.val := digit.lexval
$N \rightarrow$;	It is terminating symbol.

* Syntax Tree: 9 + 5 - 2 / C

$E \rightarrow E + T$	E.nptr = mknode ('+', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr = T.nptr
$T \rightarrow T * F$	T.nptr = mknode ('*', T_1.nptr, F.nptr)
$T \rightarrow T / F$	T.nptr = mknode ('/', T_1.nptr, F.nptr)
$T \rightarrow F$	T.nptr = F.nptr
$F \rightarrow (E)$	F.nptr = E.nptr
$F \rightarrow id$	F.nptr = mkleaf (id, id-entry)
$F \rightarrow num$	F.nptr = mkleaf (num, num.lexval)





OR.

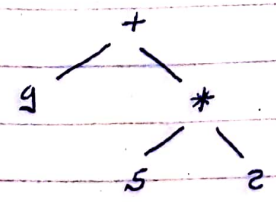
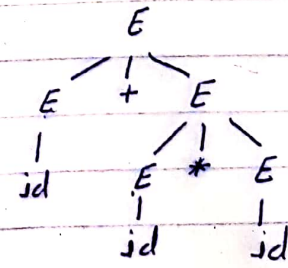
Que:-4- (b) What is peephole optimization & which are the characteristics of it? Discuss in detail.

Ans:-

Note:- Read answer from Dec-2010 Question Paper (Q.5(a) OR)

Que:-5 (a) Differentiate the following:
Syntax tree & Parse tree.
Synthesized attribute & Inherited att.

Ans:-

Syntax Tree	Parse Tree
⇒ A tree in which each leaf represents an operand & each interior node an operator is called syntax tree.	⇒ A graphical representation for derivation that filter out the choice regarding replacement order. This representation is called parse tree.
⇒ Starting node of syntax tree is always operator (Terminal)	⇒ Starting node of Parse tree is alway starting symbol of the gram^{r}(NT)
⇒ It require less memory.	⇒ It require more memory.
⇒ Ex:- 9 + 5 * 2. 	⇒ Ex:- E → E+E/E*E/ id / (E) id + id * id 

* Chapter :- 7 Linkers *

Paper [Jan-2011]

* What is program Relocation & Explain the use of EXTRN and ENTRY statements in linking. [07]

Ans:

* Program Relocation :-

Program relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.

* EXTRN & ENTRY statements :-

→ The ENTRY statement lists the public definitions of a program unit, i.e. it lists those symbols defined in the program unit which may be referenced in other program units.

→ The EXTRN statement lists the symbols to which external references are made in the program unit.

→ Ex :-

```
        START   500
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A                500) +09 0 540
LOOP    ;                        501)
        :
        MOVER   AREG, ALPHA      518) +04 1 000
        :
        STOP                     539) +00 0 000
A       DS      1                540)
TOTAL   DS      1                541)
        END
```

Madhuri Bera

2) In the above program the ENTRY statement indicates that a public definition of TOTAL exists in the program.

2) Note that LOOP & A are not public definitions even though they are defined in the program.

2) The EXTRN statement indicates that the program contains external references to MAX & ALPHA.

2) The assembler does not know the add. of an external symbol.

[Jan-2011]

* What is the advantage of overlay? How it works? Explain in detail. [07]

Ans: * Advantage:

2) overlays are used to reduce the main memory requirement of a program.

* overlays Work:

2) program Consists of

1: A permanently resident portion, called the root.

2: A set of overlays.

2) For linking & execution of an overlay structured program in MS Dos the linker produces a single executable file at the output, which contains two provisions to support overlays.

2) First, an overlay manager module is included in the executable file.

2) This module is responsible for loading the overlays when needed.

2) Second, all calls that cross overlay boundaries are replaced by an interrupt producing instruction.

2) To start with, the overlay manager receives control & loads the root.

2) A procedure call which crosses overlay boundaries leads to an interrupt.

2) This interrupt is processed by the overlay manager & the appropriate overlay is loaded into memory.

2) When each overlay is structured in to a separate binary program, as in IBM mainframe systems, a call which crosses overlay boundaries leads to an interrupt which is attended by the OS kernel. Control is now transferred to the OS loader to load the appropriate binary program.

Madhvi Bera

[June-2011]

Q.2 A Compiler typically generates a .OBJ file, which is later converted into .EXE or a .COM File. Clearly describe the difference between the three files. [03]

Ans:

1) A file with .COM extension contains a non relocatable object program whereas a file with .EXE extension contains a relocatable program.

2) MS Dos contains a program EXE2BIN which convert a .EXE program into a .COM program.

3) When the user types a filename with the .COM extension, the absolute loader ~~involving~~ is invoked to simply load the object program into the memory.

4) When a filename with the .EXE extension is given, the relocating loader relocates the program to the ~~designed~~ designated load area before passing control to it for execution.

Madhuri Rao

June-2011]

Q. Explain an algorithm for Pass-I of linker. [07]

Ans: Note:- Read answer from Book
Page No:- 240.

June-2011]

Q. Write the advantages of overlay technique. [04]

Ans:

1) Overlays are used to reduce the main memory requirement of a program.

2) It also makes it possible to execute programs whose size exceeds the amount of memory which can be allocated to them.

June-2011]

Q. Write a short note on program relocation with suitable example. [03]

Ans:

1) Program relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.

2) If linked origin ≠ translated origin, relocation must be performed by the linker.

Madhu Bera.

⇒ If load origin ≠ linked origin, relocation must be performed by the loader.

⇒ In general, a linker always performs relocation, whereas some loaders do not perform relocation that is.

load origin = linked origin.

⇒ Example:

```
        START   500
        ENTRY   TOTAL
        EXTRN   MAX, ALPHA
        READ    A                   500) + 09 0 540
                                    504)
LOOP
        :
        MOVER   AREG, ALPHA         518) + 04 1 000
        BC      ANY, MAX.           519) + 06 6 000
        :
        BC      LT, LOOP            538) + 06 1 501
                                    539) + 00 0 000
        STOP
A       DS      1                   540)
TOTAL   DS      1                   541)
        END.
```

⇒ Address of A = 540

If linked origin = 900

∴ A - link time address = 940.

Dec. 2011

* Explain object module and its components. [03]

Ans: => The object module of a program contains all information necessary to relocate and link the program with other programs.

=> The object module of a program P consists of 4 components:

1. Header: The header contains translated origin, size and execution start address of P.

2. Program: This component contains the machine language program corresponding to P.

3. Relocation table: (RELOCTAB) This table describes IRRp. Each RELOCTAB entry contains a single field:

Translated address: Translated address of an address sensitive instruction.

4. Linking table (LINKTAB):-

This table contains information concerning the public definitions & external references in P.

Madhuri Bera

⇒ Each LINKTAB entry contains three fields:

Symbol : Symbolic name.

Type : PD/EXT indicating whether public definition / external reference.

Translated Add. : - For a public definition, this is the address of the first memory word allocated to the symbol.
- For an external reference, it is the add. of the memory word which is required to contain the address of the symbol.

⇒ Example: Consider the assembly program of previous question-answer example (Page No:- 6). The object module of the program contains the following information:

⇒
1. translated origin = 500, size = 42, execution start add = 500.
2. M/c lang. instructions shown in Page No:- 6. program.
3. Relocation table

500
538

Madhuri Bora

4. linking table.

ALPHA	EXT	518
MAX	EXT	514
A	PD	540

Dec-2011

* Write short note on Absolute Loader and Relocatable loader. [04]

Ans:

1) An absolute loader can only load programs with load origin = linked origin.

2) This can be inconvenient if the load address of a program is likely to be different for different executions of a program.

3) A relocating loader performs relocation while loading a program for execution.

4) This permits a program to be executed in different parts of the memory.

5) The system contains two loaders an absolute loader & a relocating loader.

6) When the user types a filename with the .COM extension, the absolute loader is invoked to simply load the object program into the memory.

Madhu Bera.

⇒ When a filename with the .EXE extension is given, the relocating loader relocates the program to the designated load area before passing control to it for execution.

[Dec-2011]

* Define: Translated origin, Linked origin & Load origin. [CO3]

Ans:

* Translated origin:

Address of the origin assumed by the translator. This is the address specified by the programmer in an ORIGIN statement.

* Linked origin:

Address of the origin assigned by the linker while producing a binary program.

* Load origin:

Address of the origin assigned by the loader while loading the program for execution.

Madhvi Bera.

Dec-2011

* Write short note on self-relocatable programs. [04]

Ans:

2) Program classify into the following:

1: Non relocatable programs,
2: Relocatable programs,
3: self-Relocating programs.

I: Non Relocatable Programs.

2) A non relocatable program is a program which cannot be executed in any memory area other than the area starting on its translated origin.

2) Non relocatability is the result of address sensitivity of a program & lack of information concerning the address sensitive instructions in the program.

2) The difference between a relation relocatable program and a non relocatable program is the availability of information concerning the address sensitive instructions in it.

II Relocatable Programs:

2) A relocatable program can be processed to relocate it to a desired area of memory.

2) Representative examples of non relocatable & relocatable programs are a hand coded machine language program & an object module, respectively.

III) Self-Relocatable programs.

2) A self-relocating program is a program which can perform the relocation of its own address sensitive instructions.

2) It contains the following two provisions for this purpose:

1. A table of information concerning the address sensitive instructions exists as a part of the program.

2. Code to perform the relocation of address sensitive instructions also exists as a part of the program. This is called the relocating logic.

2) A self-relocating program can execute in any area of the memory. This is very important in time sharing ~~program~~ operating systems where the load add. of a program is likely to be different for different executions.

Mabhui Bera.

-2012]
* What is the job of linker?
Ans:
→ Linking is the process of binding an external reference to the correct link time address.

-2012]
* What is linked address?
Ans:
It contains the linked origin of the object module.

program_linked_origin = <link origin>

-2012]
* Define overlay.
An overlay is a part of program (or software package) which has the same load origin as some other parts of the program.

May-2012]
* What is object module?
The object module of a program contains all information necessary to relocate & link the program with other programs. It is contain four components.

i. Header.
ii. Program
iii. Relocation table.
iv. Linking table.

Madhu Babu.

[May-2012]

* Explain non-relocatable, relocatable & self relocating program. [07]

Ans:-

Note:- Read the answer from Page No:- 11. (Dec-2011. Que. Paper)

[May-2012]

* What is the function of loader? Define an absolute loader. [03]

Ans:-

⇒ Loading of the program in the memory for the purpose of execution.

⇒ An absolute loader can only load programs with load origin = linked origin.

⇒ The loader system contains two loaders - an absolute loader & a relocating loader.

⇒ When the user types a filename with the .COM extension, the absolute loader is invoked to simply load the object program into the memory.

Madhvi Bera