8/04/2021

Compiler Design

*Introduction:—

→ The programming languages are divided into three categories

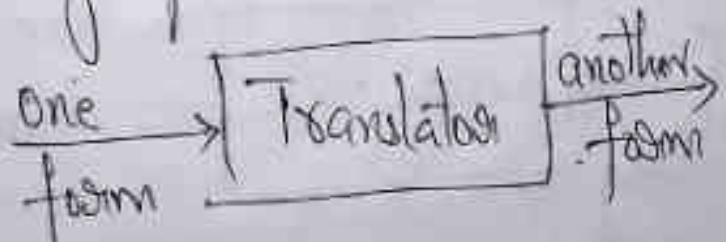
languages

- High-level languages — Eg:- java, C, C++
- middle level languages — Eg:- Assembly lang
- low level languages — Eg machine language

→ Computer understands only 0's & 1's (machine language). Therefore, translators are used to convert one language to another language.

→ Translator:—

> A Translator is a programming language processor that converts a Computer program from one language to another.



Examples:-

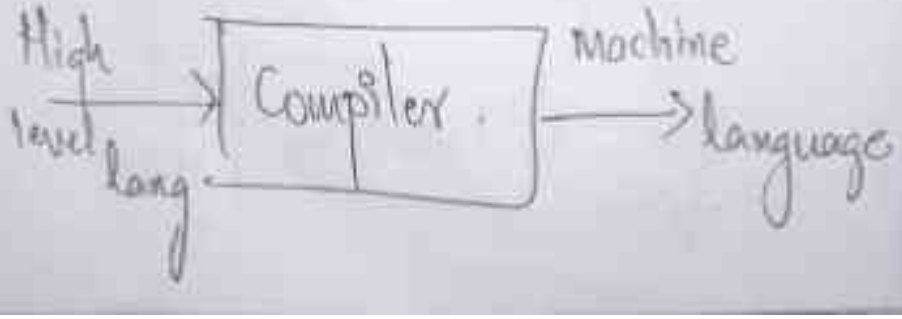
i) Compilers
ii) Interpreters
iii) Assemblers

* Compilers:-

-> Compiler is a translator / software that converts high-level languages (eg C++) into set of machine level instructions that can be understood by the CPU.

-> Compilers are very large programs, with error-checking and other abilities.

-> Some compilers generate machine language directly while some converts high-level language to assembly language which is then translated into machine code by assembler.

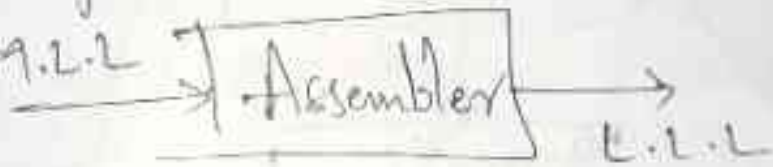


* Interpreter

-> An Interpreter is a translator that repeatedly reads instructions (one at a time) and translate them into machine lang.

* Assembler :-

-> Assembler is a translater which translates middle level languages (eg Assembly language) into machine language.



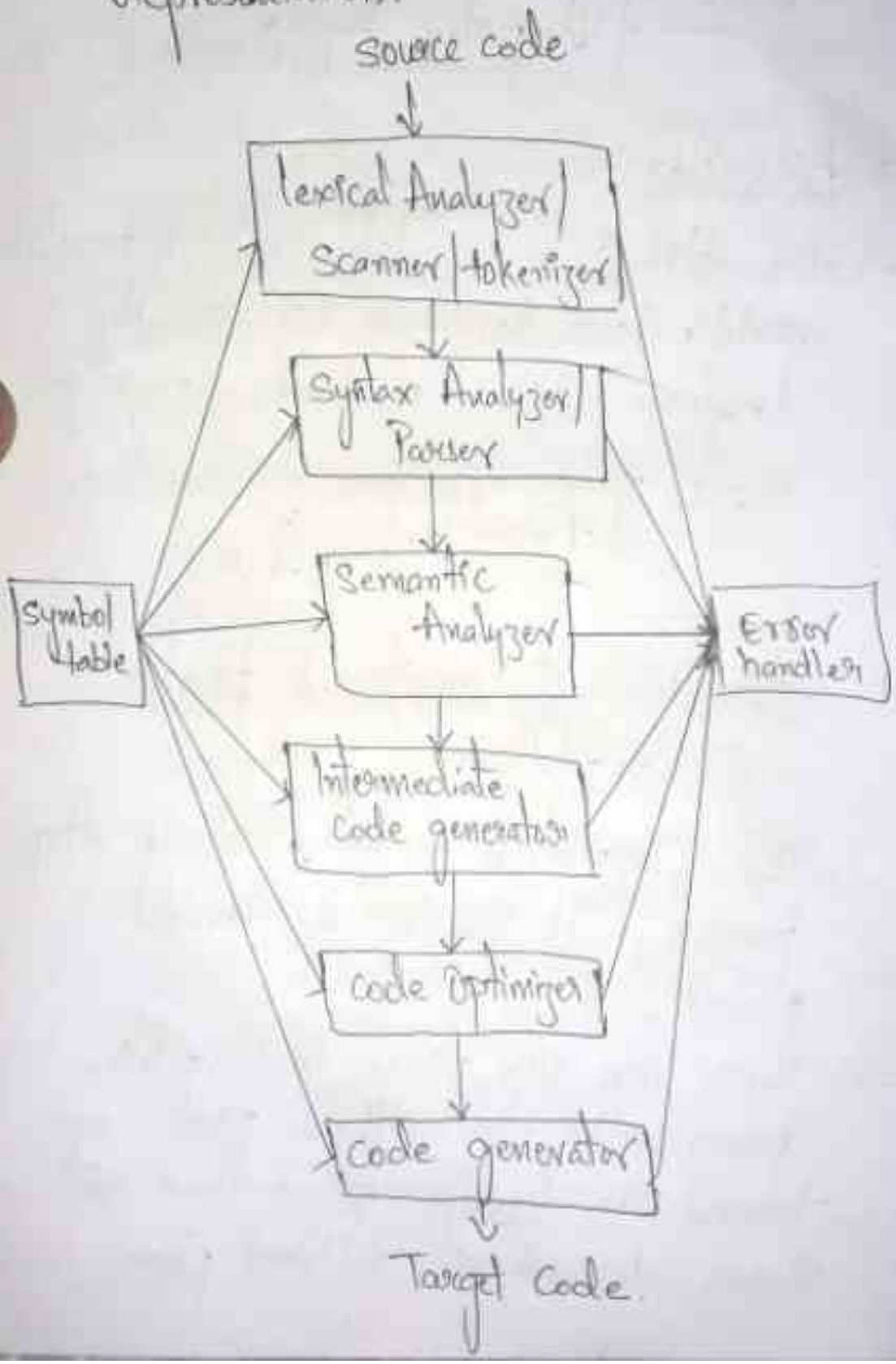
* Structure of Compiler & Phases of Compilation

-> The Compilation is not a single step process, it Consists of several phases

-> There are six phases in Compilation process, the first three phases are termed as Analysis phase and next three termed as Synthesis phase.

→ Analysis phase creates an intermediate representation from given source code

→ Synthesis phase creates an equivalent target program from intermediate representation.



-> The Compiler has two modules namely front-end and backend.

-> Front-end Constitutes of i) lexical Analyzer
ii) Syntax Analyzer
iii) Semantic Analyzer

-> Back-end Constitutes i) Intermediate code generator
ii) Code Optimizer
iii) Code Generator.

-> The first three phases are language Independent, next three phases are Machine dependent or target dependent

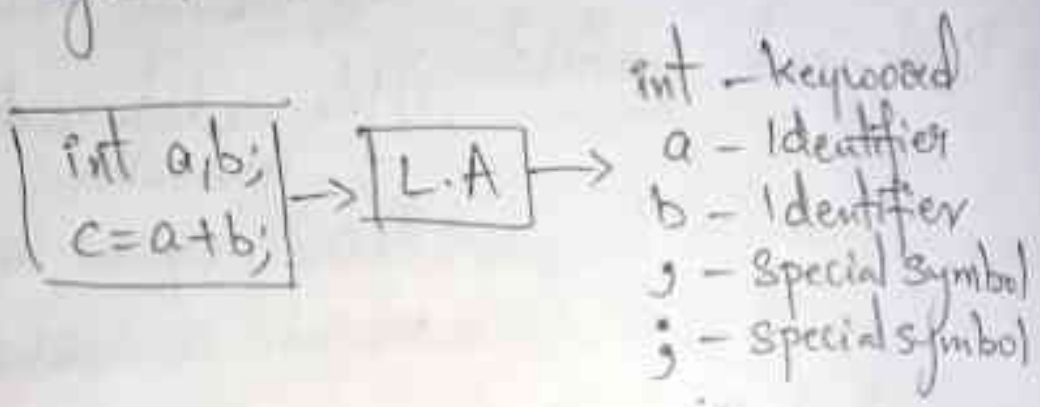
* lexical Analyzer:-

-> The lexical Analyzer Scans the whole program and breaks down into meaningful Units called tokens.

-> It is also called Scanner because it Scans the source program and also called as tokenizer because it breakes down the source program into tokens.

Input:- Source program
output:- Tokens.

→ It removes Comments and WhiteSpaces
→ The dictionary required for lexical Analyzer is Regular Expressions
→ It is a phase that makes entry into Symbol table

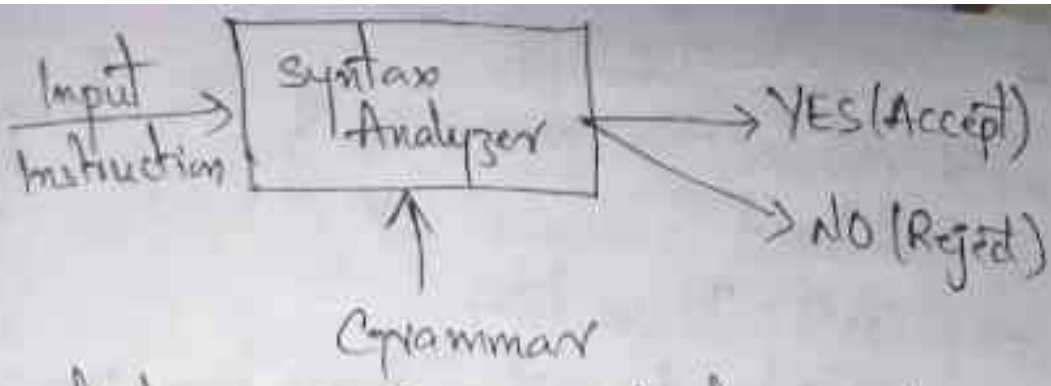


→ It deposits lexical Errors such as
i) Missing of a character
ii) Addition of a character
iii) Transposition of a character
iv) Writing outside of Comment.

* Syntax Analyzer:-

→ Every language has its own set of Rules called grammar.
→ The Syntax Analyzer will check whether the given Instruction follows the grammar or not.

Input:- Tokens
Output:- Parse Tree.



→ The dictionary required for syntax analyser is Context-free grammar.

→ Parsing :- The process of checking weather the string is accepted by the grammar or not

Eg:- $E \rightarrow E+E \mid E*E \mid id$

$G = (V, T, P, S)$ $t = \{+, *, id\}$

LMD

$E \rightarrow E+E$

$E \rightarrow E*E+E$

$E \rightarrow id*E+E$

$E \rightarrow id*id+E$

$E \rightarrow id*id+id$

Parse tree

* Semantic Analyzer:-

-> It checks the meaning of the language construct

-> Input: Parse tree
Output: Annotated parse tree / semantic tree

-> At this phase type checking and type compatibility is done.

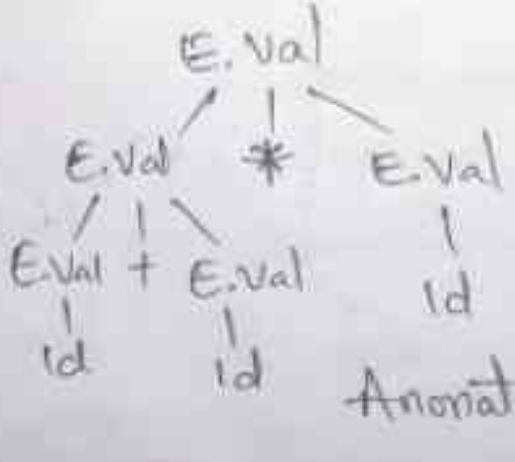
-> With each grammar symbol we associate an attribute. The grammar after attaching attribute is called as Syntax Directed Defination

-> The parse tree with attributes is called as Annotated parse tree

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

E.val -> E.val + E.val
E.val -> E.val * E.val
E.val -> id

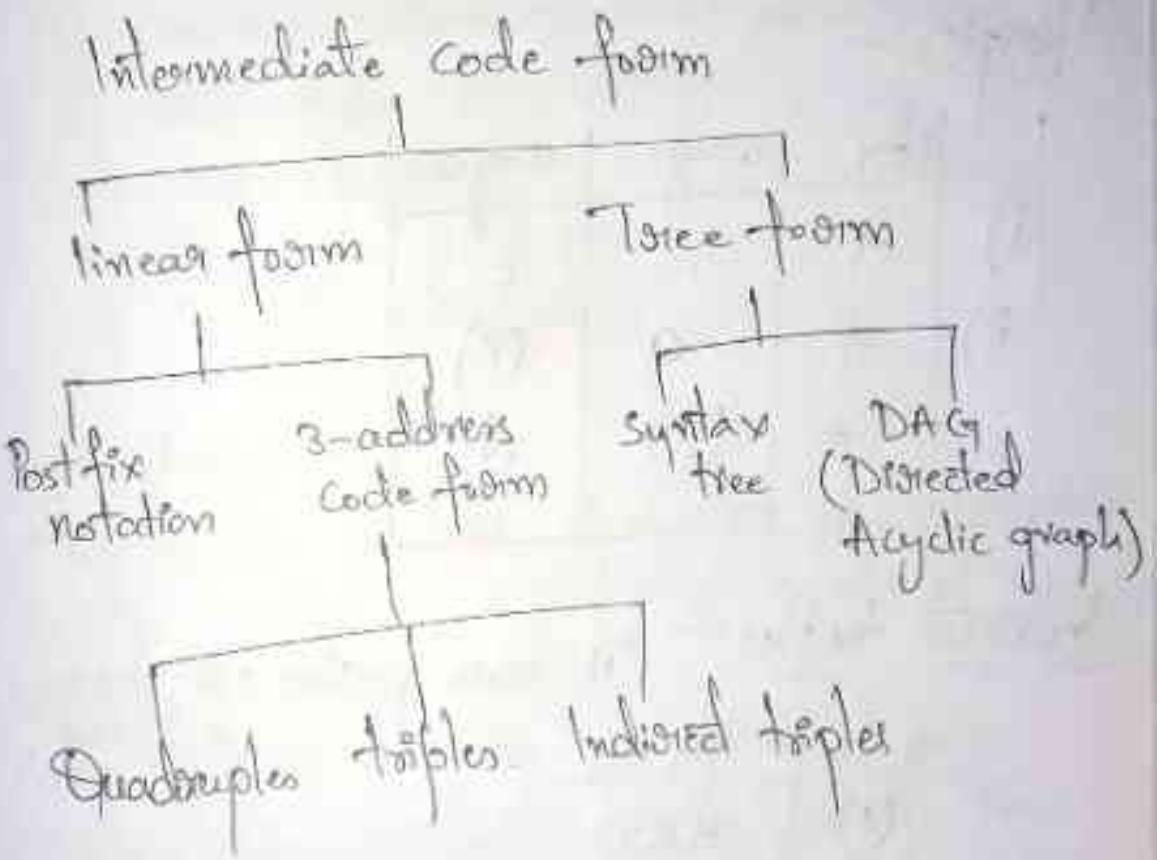
} Syntax directed defination



Annotated parse tree.

* Intermediate code generator:-

→ The machine Independent front-end part is seperated by machine dependent backend part by Intermediate code form



→ postfix notation:-

→ a + b * c

→ a + bc *

→ abc * +

Quadruples:-

d = a + b * c

Opr	arg1	arg2	Result
*	b	c	t_1
+	a	t_1	t_2
=	t_2		d

$t_1 = b * c$

$t_2 = a + t_1$

$d = t_2$

Triples:-

	Opr	arg1	arg2
i)	+	b	c
ii)	+	a	(i)
	=	(ii)	d.

Indirect triples:- It uses pointer to store address.

S.No	Ptrs to
(1)	1000
(2)	1001

Syntax tree:- d = a + b * c

* Code Optimization:-
→ It transforms the code so that it consumes fewer resources and produce more speed
→ Input: Intermediate code form
output: Optimized code
→ Various code Optimizing techniques are
i) Common sub-expression elimination
ii) Dual code elimination
iii) copy propagation
iv) Strength reduction
v) loop optimization

* Code Generator:-
→ The main purpose of target code generator is to write a code that machine can understand.
→ The Optimized code is converted into relocatable machine code which then forms the input to linker and loader
→ Input:- Optimized code
output: Target code

* Symbol table:- It is a data structure being used and maintained by the compiler, Consists of the Identifiers name along with their types. It helps the Compiler to function smoothly by finding Identifiers quickly.

* The Translation process:-

→ The translator divides the translation process into series of phases

→ Each phase manages some particular aspect of translation.

i) $a = b * c + d$. (d is float)

Sol a, b, c are integers and d is float

Symbol table :-

a	
b	
c	
d	

$a = b * c + d$

↓

lexical Analyzer

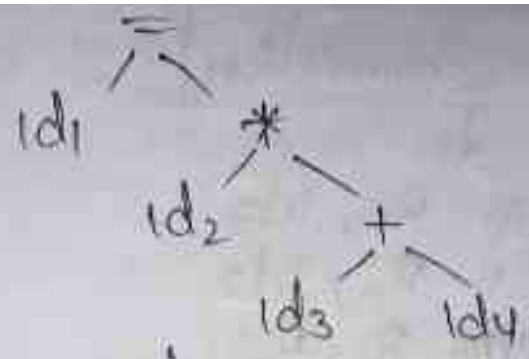
↓

$id1 = id2 * id3 + id4$

↓

Syntax Analyzer

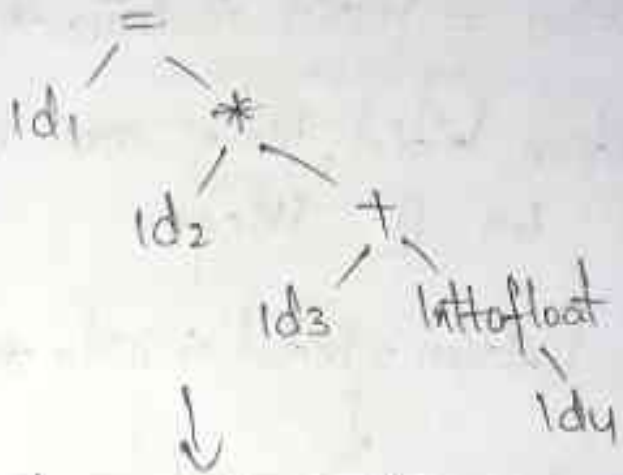
↓



↓

Semantic Analyzer

↓



↓

Intermediate code generator

↓

$t_1 = id2 * id3$

$t_2 = t_1 + id4$

$id1 = t_2$

↓

Code Optimization

↓

$t_1 = id2 * id3$

$id1 = t_1 + id4$

↓

Code Generation

↓

```
MOV R1, Id3
MOV R2, Id2
MUL R1, R2
ADD R1, Id4
```

ii) Position = initial + rate * 60

Sol Position, initial, rate are real values
& 60 is integer

Position = initial + rate * 60

↓

lexical Analyzer

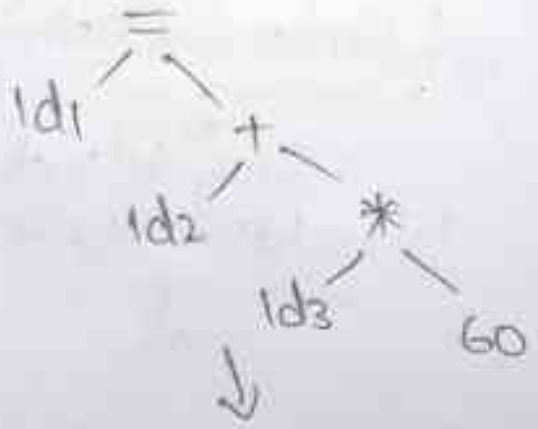
↓

Id1 = Id2 + Id3 * 60

↓

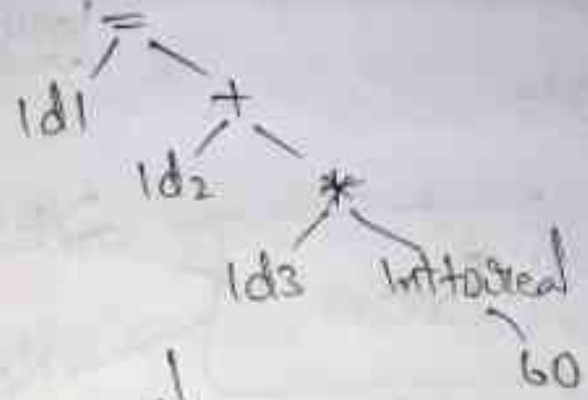
Syntax Analyzer

↓



↓

Semantic Analyzer



Intermediate code generation

$t_1 = \text{inttoreal}(60)$
$t_2 = id3 * t_1$
$t_3 = id2 + t_2$
$id1 = t_3$

Code Optimization

$t_1 = id3 * 60.0$
$id1 = id2 + t_1$

Code generation

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

* Automatic lexical Analyzer Generator (LEX).

→ LEX is a tool or utility in UNIX-OS

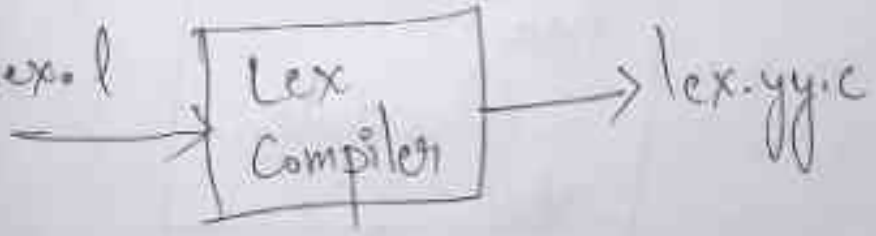
→ LEX is a program that generates lexical Analyzer. It is used with YACC parser generator.

YACC : Yet Another Compiler Compiler.

→ The lexical Analyzer is a program that transforms an Input stream into sequence of tokens.

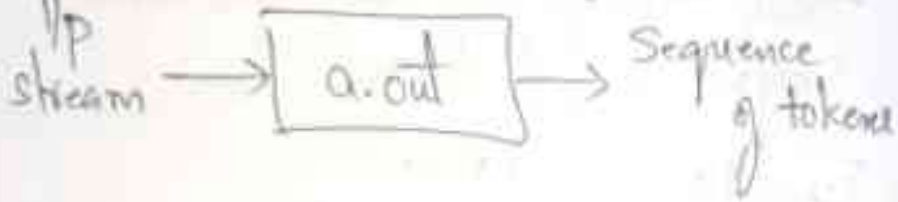
→ Functions of LEX:—

i) Firstly lexical Analyzer creates a program lex language. The lex compiler runs the lex.l program and produces a c program lex.yy.c



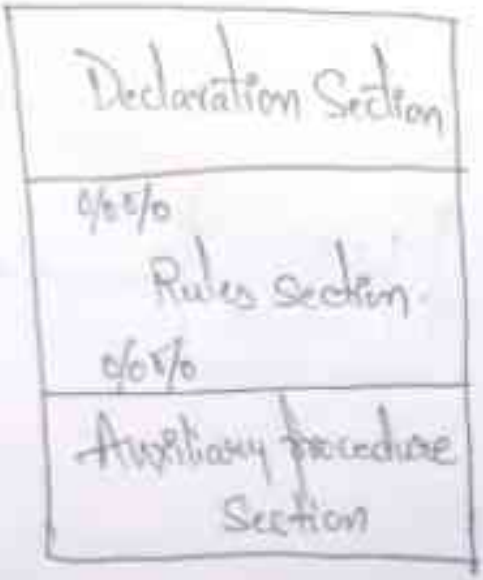
ii) C Compiler runs lex.yy.c program and produces an object program a.out

iii) a.out is lexical Analyzer that transforms an Input stream into sequence of tokens.



→ LEX file format:—

→ The lex program is seperated into three sections by %%% delimeter.



i) Declaration Section:-

→ It consists of variables, constants, manifest that are used in the program.

→ It is Optional.

ii) Rules section:-

→ The rules section gives the specification of lex file.

→ It describes the pattern or regular expression and corresponding action to be taken.

```
RE1   {action1}
RE2   {action2}
RE3   {action3}
```

→ It is compulsory.

iii) Auxiliary procedure section:-

→ It consists of all the functions and methods used in a program

Q) Write a lex program to identify keywords, numbers, identifiers, operators, special symbols.

->]$ vi lex1.l $ // Creating lex1.l file

```
%%%
[a-z A-Z][a-z A-Z 0-9]* printf("Identifier");
int|float|while|for  printf("keywords");
[+|-|*|/]  printf("Operators");
[0-9]*  printf("Numbers");
[@|$]  printf("Special Symbol");
%%%
main()
{
  yylex();
}
int yywrap()
{
}
```

->]$ lex lex1.l
]$ cc lex.yy.c
]$ ./a.out

Output:-

area	Identifier
for	keyword
+	Operator
456	Number
@	Special Symbol.

* Role of lexical Analyzer:-

→ lexical Analyzer scans volatile programs and divides into small tokens.

→ It removes volatile spaces & Comments

* lexemes, tokens, patterns:-

lexeme	token	pattern
int	keyword	→ int
circle	Identifier	→ letter(letter|digit)*
10	number	→ [0-9]*

Lexemes:- It is defined as sequence of characters that are matched with the pattern of the token and is identified by lexical Analyser as an instance of token.

Ex:- int, Circle, a etc

Token:- Tokens are the sequence of characters having collective meaning. They describe class and category of input string. Token is a pair of token name & its value (attribute)

Pattern:- Patterns are the set of rules that describe the lexeme representing a particular token

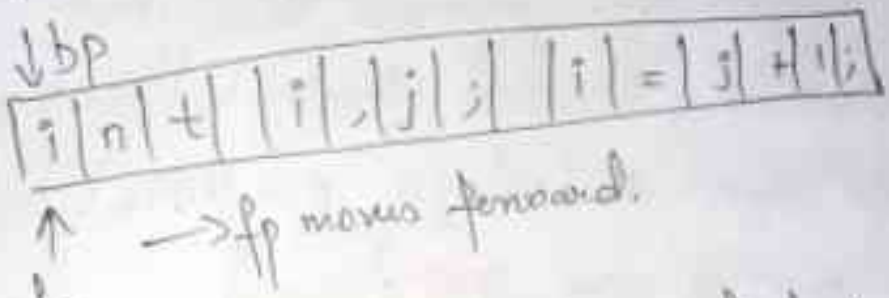
Ex:- Pattern for an Identifier -> letter (letter/digit)*

Pattern for number -> [0-9]*

* Input Buffering :-

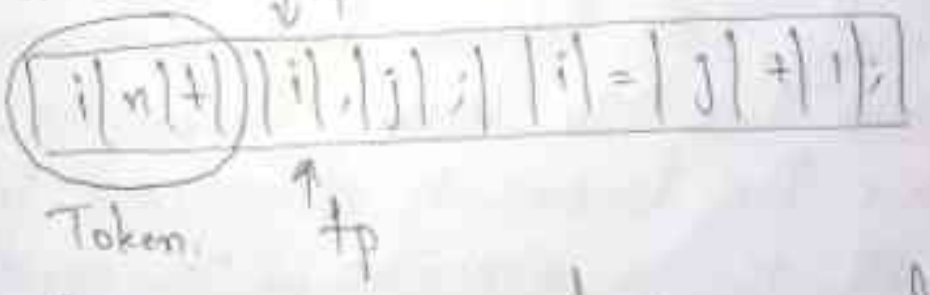
→ The lexical Analyzer Scans the input from left to right one character at a time.

→ It uses two pointers begin ptr (bp) and forward pointer (fp)



→ Initially both pointers point to first character.

→ The fp moves head to search for end of lexeme. As soon as blank space is encountered, it indicates the end of lexeme.



→ Two pass large amount of source program, specialized buffering techniques are used to reduce the amount of overhead required to process a single input character.

→ There are two techniques:-

i) Buffer pairs

ii) Sentinals

i) Buffer Pairs:-

→ Two buffers are involved that are alternatively loaded. Each buffer is of same size n (where n = 4096 bytes)

→ Two pointers are maintained.

i) lexeme begin: It marks to the beginning of current lexemes

ii) Forward pointer: It scans ahead until a pattern match is found

→ Initially, both lexeme begin & forward pointer to the same character. Once the lexeme is determined forward is set to the character at its right end.

→ Once the lexeme is recorded as token, lexeme begin is set to character immediately after the lexeme is found.

→ Advancing forward requires that the first test whether we have reached the end of one of the buffer & if so we must reload the other buffer from the inputs & move forward fp to the beginning of newly loaded buffer.

Algorithm Advance forward :-

if fp reaches the first half then
reload the second half

fp = fp + 1;

else if fp reaches the second half then reload the first half

move fp to the beginning of first half

else

fp = fp + 1;

end.

Disadvantage with buffer pairs :-

→ The pointer fp is required to perform two tests while reading the character from the i/p

a) to check the end of the half
b) whether the characters reach the end of the lexeme or not.

ii) Sentinals :-

-> The above disadvantage can be overcome by using Sentinals.

-> In Sentinals we use certain characters (not a part of source language) to mark the end of half

generally we use EOF.

Algorithm:-

```
fp = fp + 1;
if fp = EOF
    if fp reaches first half then
        reload the second half.
        fp = fp + 1
    else if fp reaches second half then
        reload the first half more
        fp to the begin of first half.

else
    terminate lexical Analysis

end;
```

* T-diagram:-

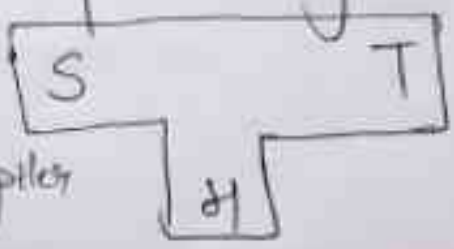
→ A Compiler can be categorized by

i) Source language

ii) Targed language

iii) Host language (Implementation language)

This is depicted by T-diagram as



S → Source lang.

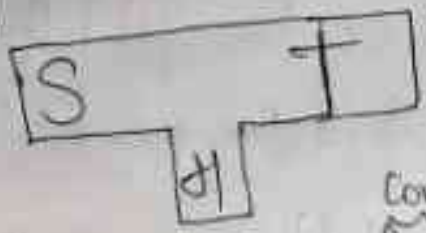
T → Target lang

H → host language

T diagram shows a Compiler $S_{H}^{T}C$

* Cross Compiler:-

→ It is defined as a Compiler running on one machine and producing the object code for another machine.



Compiler $^{S}C_{H}^{T}$

→ A Compiler written in language H (host language) that translates language S into language T

* Bootstrapping:-

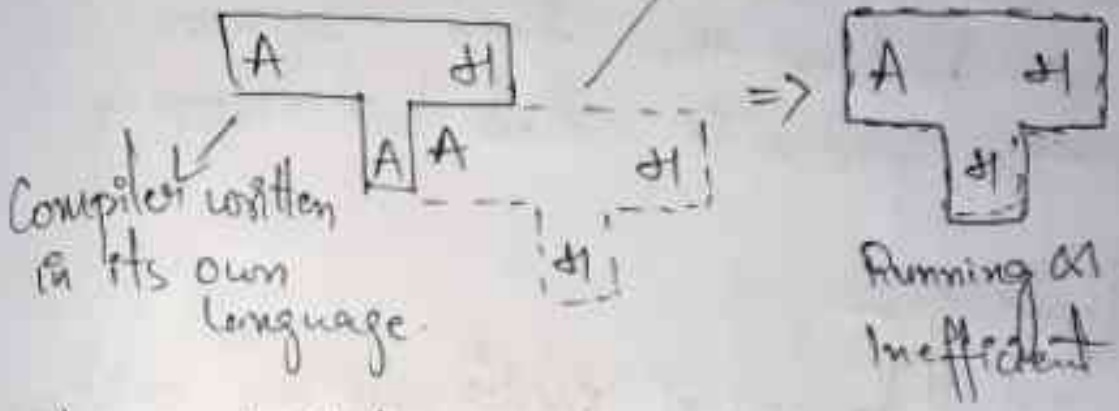
→ It is a technique for producing efficient Compilers.

→ Bootstrap Compiler is used to compile the Compiler and then you can use this Compiled Compiler to Compile everything else as well as future version of itself.

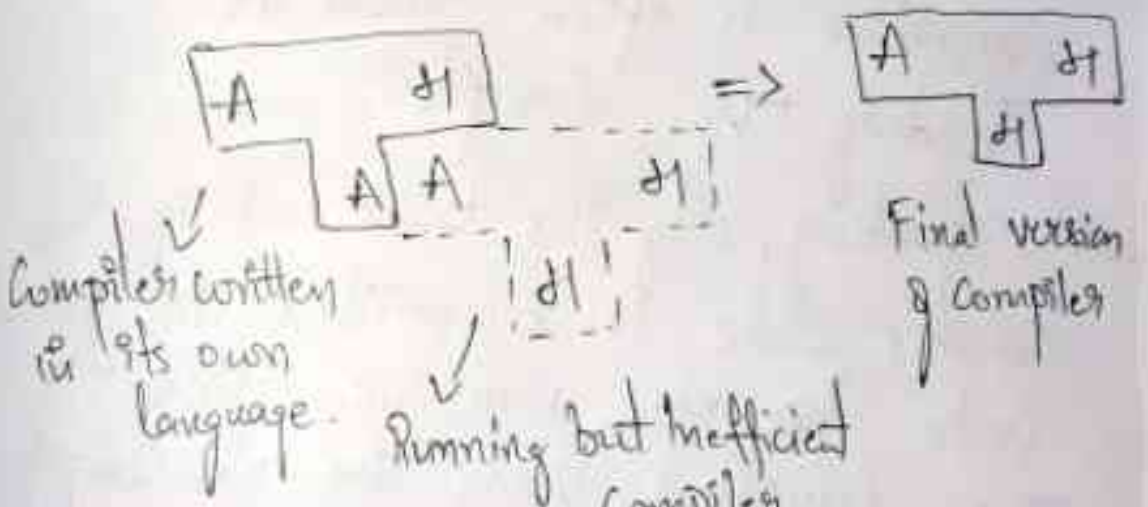
→ There are two stages in bootstrapping.

→ Stage 1 :- A Compiler for subset language is developed which is running and is Inefficient

stage 1 : subset language → Quick & dirty compiler.



stage 2 : (Full language)



→ Initially we write a quick & dirty Compiler in assembly language Translating only those features of the language that are used in the Compiler

→ The quick & dirty Compiler will produce extremely inefficient code. Once we own it we use it to compile the good Compiler.

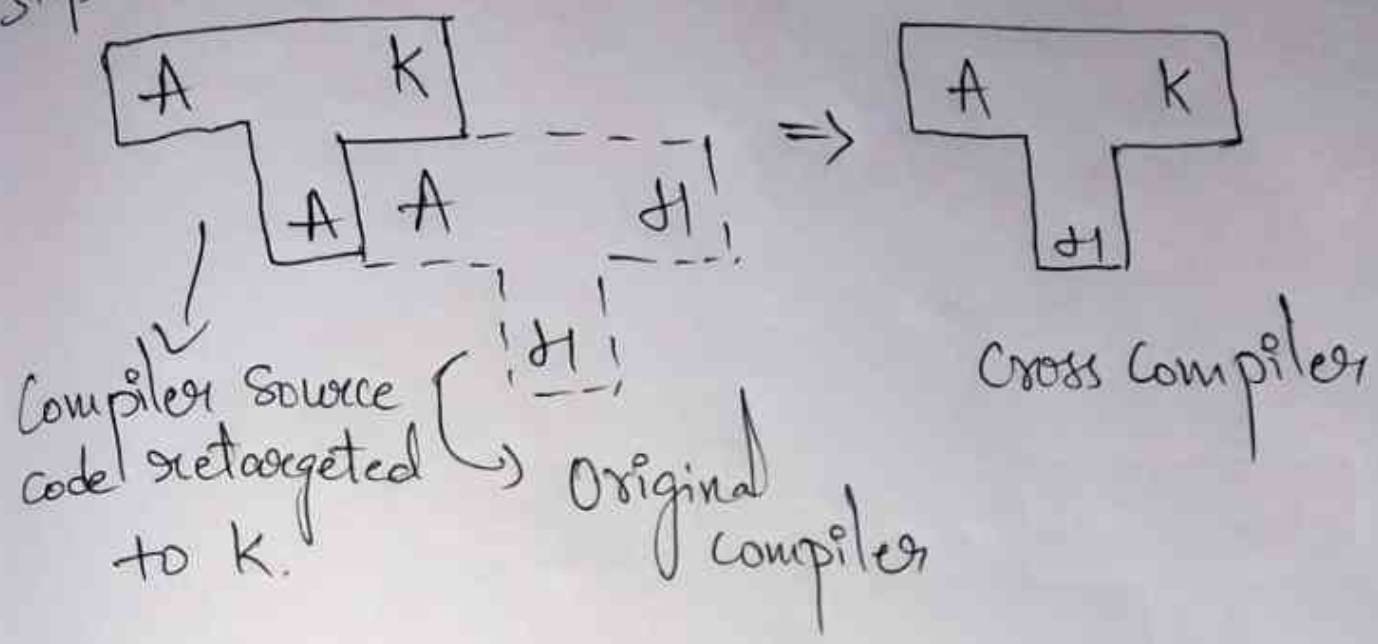
→ Then we recompile the good Compiler to produce the final version. This process is called bootstrapping.

Porting:-

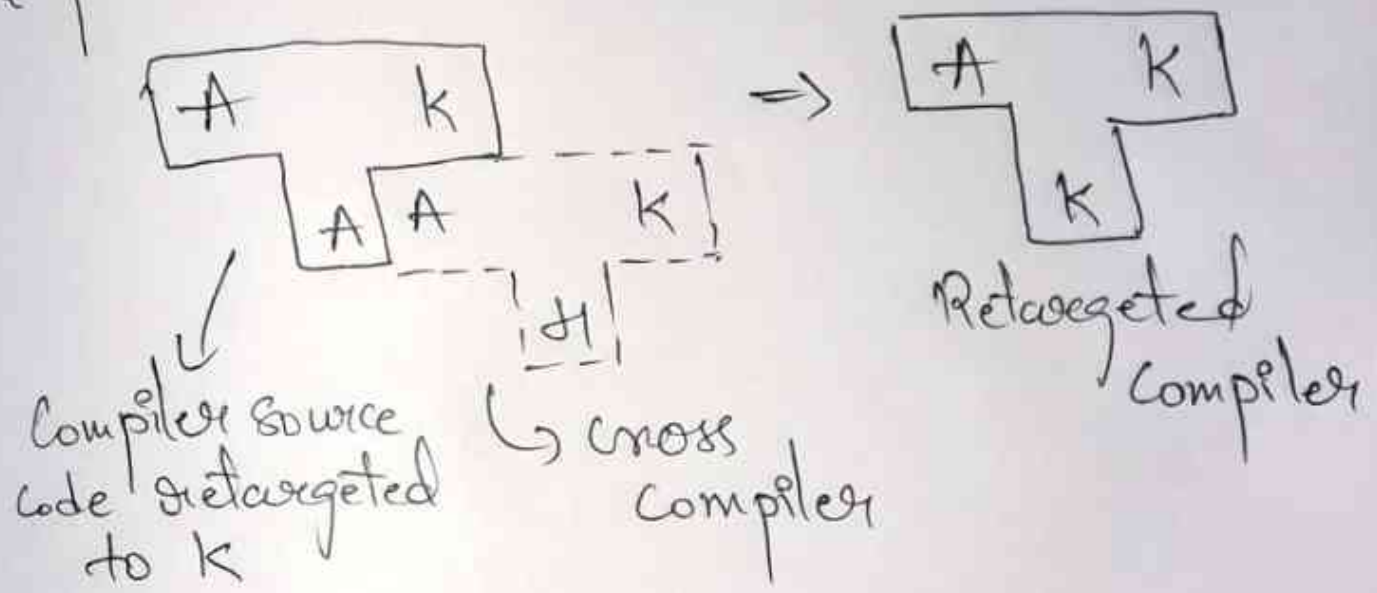
→ The process of modifying an existing Compiler to work on a new machine is often known as Porting the Compiler.

→ In porting, only the back end of the Source code is rewritten to generate code for new machine. This is then Compiled using the old Compiler to produce a cross Compiler, and the Compiler is again recompiled by the cross Compiler to produce a working version for new machine.

step1:-



step2:-



* Major Data Structures in Compiler:—

1. Symbol table
2. Literal table
3. Parse tree
4. Syntax tree

* Symbol table:—

→ It stores information about program identifiers that will be used across the phases

→ Compiler allocate memory to symbol table.

→ Symbol table has facilities to manipulate (add/delete) the elements in it

→ Identifiers are name of variables, constants, functions etc

Eg:- $x = y + z$

Symbol	Category	Type	Attribute	Memory location
x	Identifier	float	#1	1000
=	Operator		Assignment(1)	
y	Identifier	float	#2	1100
+	Operator		Arithmetic(1)	
z	Identifier	float	#3	1200

* Literal table:-

→ It is also a data structure used to maintain the details of constants & strings used in the program

→ It reduces the size of a program in memory by allowing reuse of constants & strings

→ It is needed by code generator to construct symbolic address for literals

Eg:- int x = 60;

literal	category	Attribute	memory location
60	const	int	1200

* Parse tree:-

→ It is dynamically-allocated pointer based structure.

→ It describe the syntactic structure of input.

→ In parse tree, the terminals node represent the tokens and interior nodes represent expression

→ In parse tree nodes are variant type records storing information for different types of data

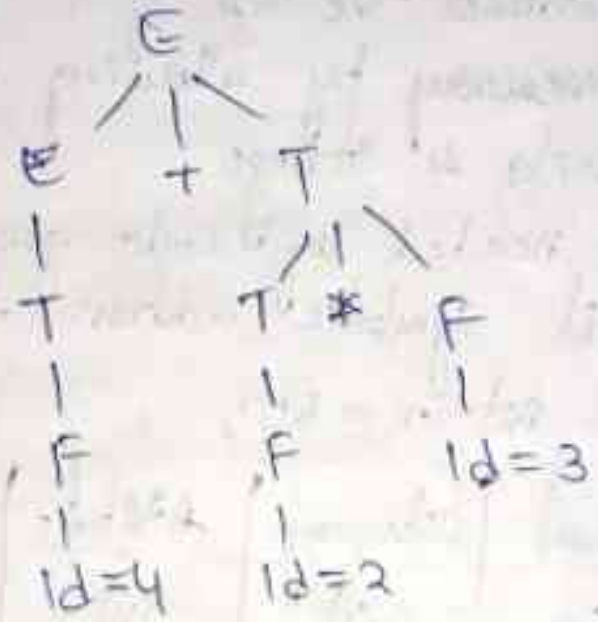
→ It is constructed from bottom-up

Eg:- $E \to E+T \mid T$

$T \to T*F \mid f$

$F \to (E) \mid id$

Expression:- 4 + 2 * 3



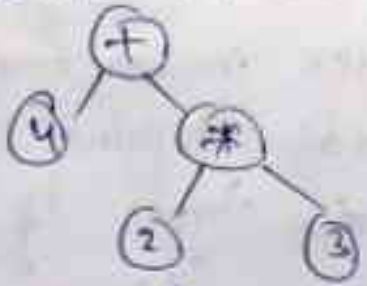
* Syntax tree:-

→ Syntactic structure is also represented using Syntax tree

→ A syntax tree is a compressed representation of parse tree.

→ In syntax tree, operators appears as interior nodes & operands as childeren.

Eg:- Expression:- 4 + 2 * 3



Syntax tree.

* Specification of tokens:-

→ There are three specification of tokens

i) String
ii) Language
iii) Regular expression

* Recognition of tokens:-

→ In this we study how to take pattern for tokens and build a piece of code that examines the input string.

Grammar: stmt → if expr then stmt
| if expr then stmt else stmt
| ∈

expr → term relop term
| term

term → id | number

Variables = { stmt, expr, term }

terminals = { if, then, else, relop, id, number }

relop : relational Operator

relop → <|>|<=|>=|=|<>

<> represent not equals

→ Pattern for tokens:-

digit → [0-9]

digits → [0-9]+ or

digits → digit+

number → digits (. digits)?

(E[+-]? digits)?

letter → [A-Za-z]

id → letter(letter|digit)*

if → if

then → then

else → else

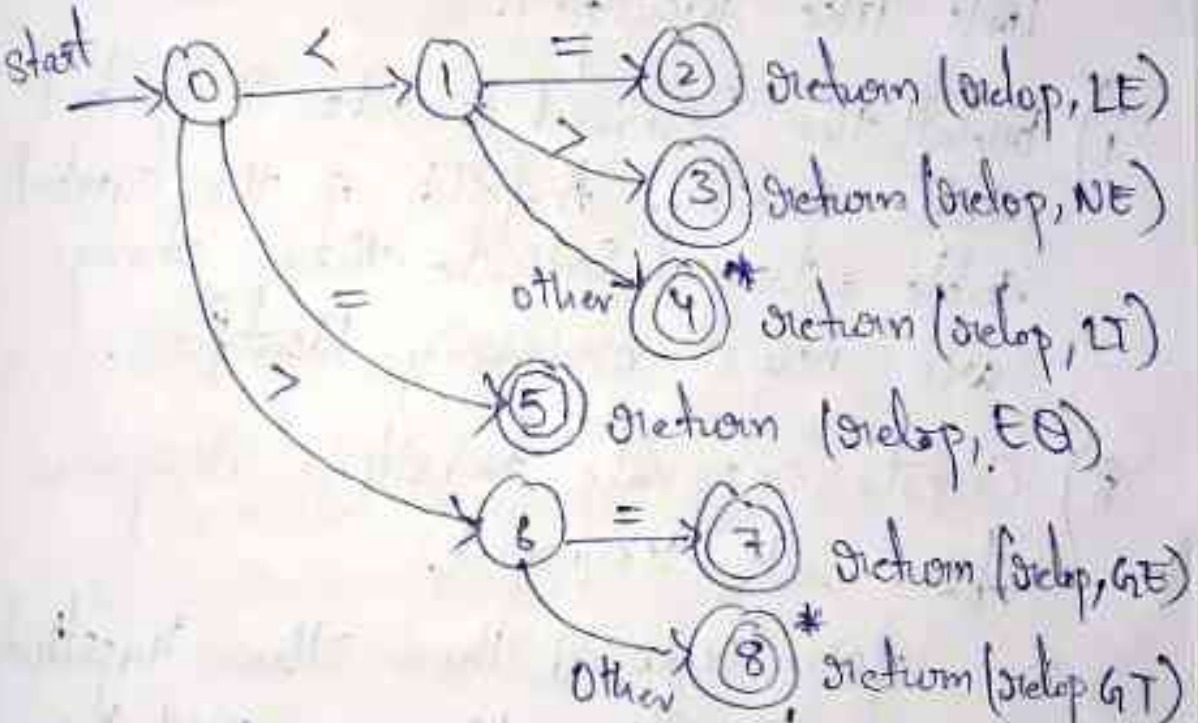
relop → <|>|<=|>=|=|<>

Transition Diagram:-

→ It is an intermediate step in Construction of lexical Analyzer.

→ It is a collection of states with edges as transitions

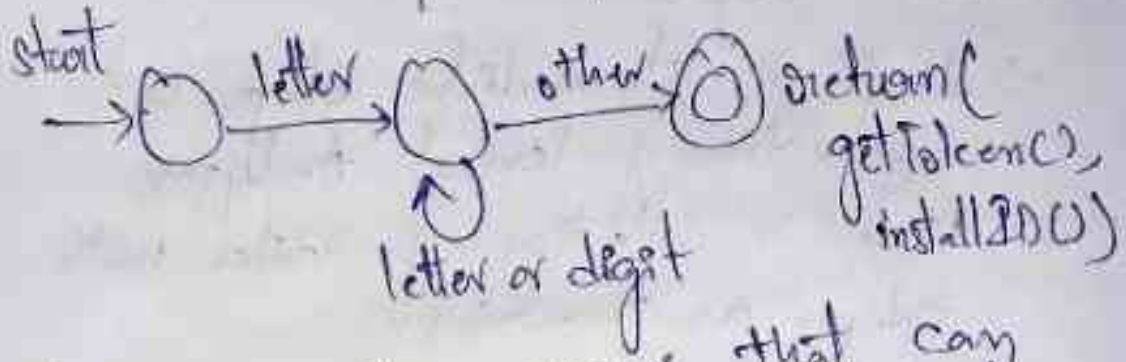
Transition diagram for relop:-



* Recognition of keyword & Identifier:-

→ Recognizing keywords and Identifier presents a problem. Usually keyword like if or then are reserved as so they are not identifiers even though they look like identifier.

Transition diagram for ids & keyword

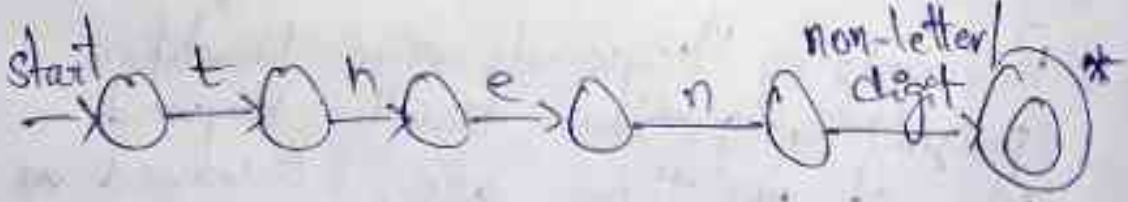


→ There are two ways that can handle reserved words that look like identifier.

i) Install the Reserved words in symbol table initially. A field of the symbol table entry indicate these strings are not ordinary identifier.

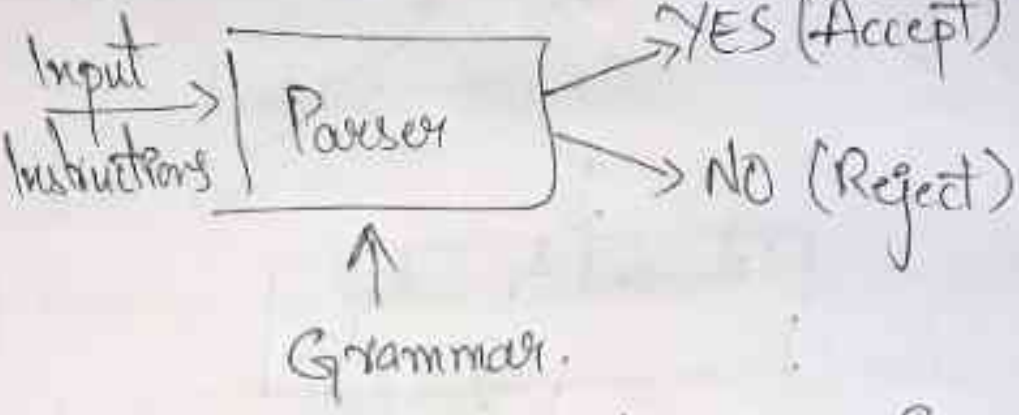
ii) Create seperate transition diagram for each keyword.

Eg:- for keyword then the hypothetical transition diagram will be



UNIT:-02 Syntax Analyzer / Parser:-

* Parser:-



→ Every programming language is described by grammar (context-free grammar)

→ A Grammar can be described using 4 tuples $G = (V, T, P, S)$

V = set of Variables / non-terminals

T = set of Terminals

P = Production Rule

S = Start Symbol

→ Terminals are lower case letters, operators and special symbols

Eg:- a, b, +, −, id

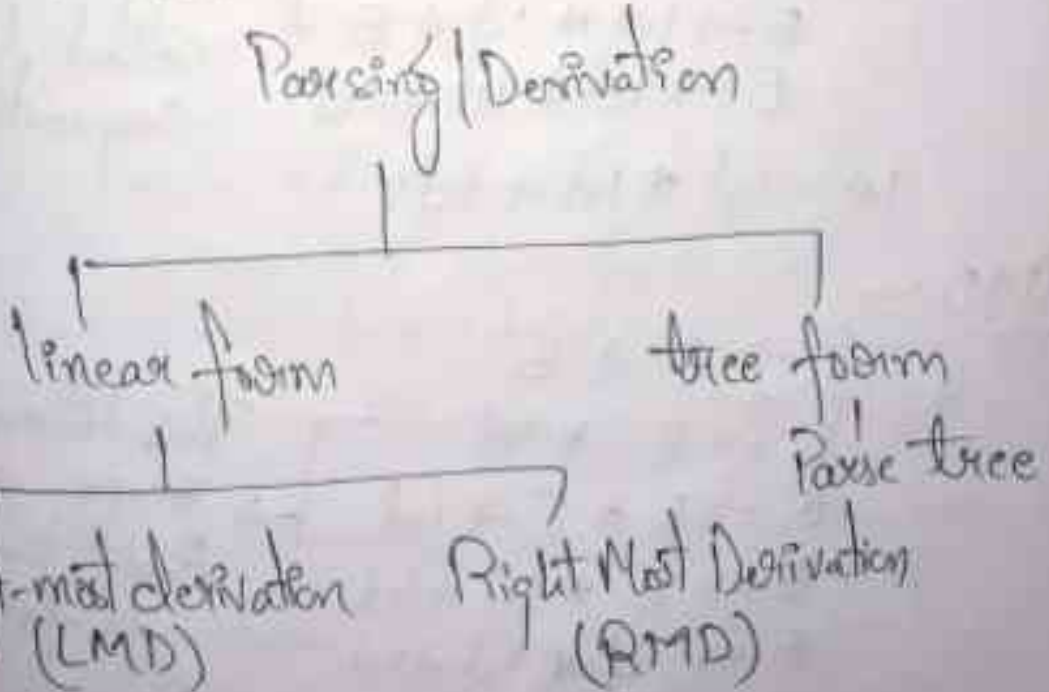
→ Variables are denoted by Upper Case letters or words.

Eg:- A, S, B

→ Variables can be reduced terminals cannot be reduced.

* Derivation:- (Parsing)

→ The process of obtaining a string from start symbol is called Derivation.



Parse tree:-

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

Parse tree for above grammar

*) Root node is start symbol

*) leaves are string.

$w = id * id + id.$

LMD:-

$E \rightarrow E+E$
$E \rightarrow E*E+E$
$E \rightarrow id*E+E$
$E \rightarrow id*id+E$
$E \rightarrow id*id+id$

→ This intermediate steps are called left sequential form

$w = id * id + id.$

RMD:-

$E \rightarrow E+E$
$E \rightarrow E+id$
$E \rightarrow E*E+id$
$E \rightarrow E*id+id$
$E \rightarrow id*id+id$

→ This intermediate steps are called Right Sequential form.

$w = id * id + id$

* Ambigous grammar :-

→ A grammar is said to be ambigous if there exists two or more LMD / Left most parse tree or two or more RMD / Right most parse tree generating the same string.

Eg :- $E \to E+E \mid E*E \mid id$.

1) L.M.D

$E \to E+E$
$E \to E*E+E$
$E \to id*E+E$
$E \to id*id+E$
$E \to id*id+id$ $\quad w_1 = id*id+id$.

2) L.M.D

$E \to E*E$
$E \to id*E$
$E \to id*E+E$
$E \to id*id+E$
$E \to id*id+id$
$w_2 = id*id+id$.

∴ w_1 is equals to w_2, hence we say the grammar is ambigous.

* Removing Ambiguity:-

→ Conversion of Ambigous grammar to Unambigous grammar

→ let $G = (V, T, P, S)$ be Ambigous grammar then its Unambigous grammar will be

$$G' = (V', T, P', S)$$

→ Rules:-

1) Number of new variables to be added = Number of Operators

2) If Operator is left associative replace the right most variable

Ex:- $E \to E+E \mid E*E \mid id$

is ambigous grammar.

No. of operators = 2 (∵ +, *)

So, 2 new variables are added to remove ambiguity.

New variables be T, P

$E \rightarrow E + E$

↳ + is left associative

So, replace its right most variable with new variable

① $E \rightarrow E + T$

② $E \rightarrow T$

Similarly, $E \rightarrow E * E$ written as

$T \rightarrow T * T$

↳ left associative

So replace right most variable with new variable

③ $T \rightarrow T * P$

④ $T \rightarrow P$

⑤ $P \rightarrow id$

So the Unambiguous grammar is

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * P$

$T \rightarrow P$

$P \rightarrow id$

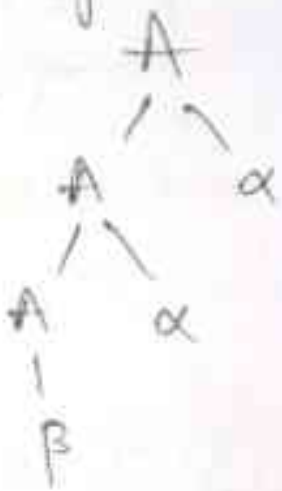
This grammar is free from ambiguity.

Note:- 1) +, -, /, * are left associative.

2) ^ & $ are right associative

* Left-Recursion:-

→ The grammar of the form $A \to A\alpha \mid \beta$ is left recursive grammar.

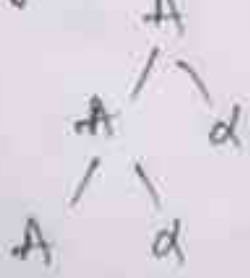


→ Eliminating left recursion we transfer the grammar into

$A \to \beta A'$

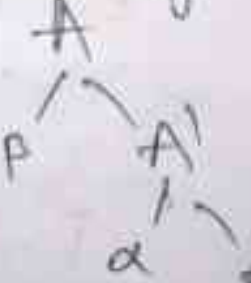
$A' \to \alpha A' \mid \epsilon$

left recursion



$w = \beta\alpha\alpha$

after eliminating left recursion



$w = \beta\alpha\alpha\alpha\epsilon$

Q) Eliminate left Recursion

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

Sol

$G = (V, T, P, S)$

$V = \{E, T, F\}$ $T = \{+, *,), (, id\}$

$E \to E + T \mid T$

left recursion, if it is in the form

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

$A = E$

$\alpha = +T$

$\beta = T$

$\therefore A \to \beta A'$

$A' \to \alpha A' \mid \epsilon$

we get,

$E \to TE'$ (1)

$E' \to +TE' \mid \epsilon$ (2)

Similarly, $T \to T * F \mid F$

$T \to FT'$ (3)

$T' \to *FT' \mid \epsilon$ (4)

Therefore, the grammar will be :

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id.$

The above grammar is "free from left recursion.

* Indirect left recursion :-

→ A grammar is said to have Indirect left recursion, on substituting any variable in other production we get direct left recursion.

Eg:- $S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \epsilon$

It is Indirect left recursion as on substituting A in S we get

$S \rightarrow Sda \mid Aca \mid a \mid b$

Left recursion

$S \rightarrow Sda \mid Aca \mid a \mid b$

left recursion

If R is the form

$A \rightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \beta_3$

we get $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A'$

$A' \rightarrow \alpha A' \mid \in$

$A \Rightarrow S$, $\alpha = da$

$\beta_1 = Aca$, $\beta_2 = a$, $\beta_3 = b$

we get

$S \rightarrow AcaA' \mid aA' \mid bA'$

$A' \rightarrow daA' \mid \in$ //

* left-factoring :-

→ left factoring is a grammar transformation technique. It consists in "factoring out" prefixes which are common to two or more production.

Eg :- $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

After doing performing left factoring on above grammar we get

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 | \beta_2$$

Eg:- $S \rightarrow iEtS | iEtSeS | a$

$E \rightarrow b$

Eliminate left factor

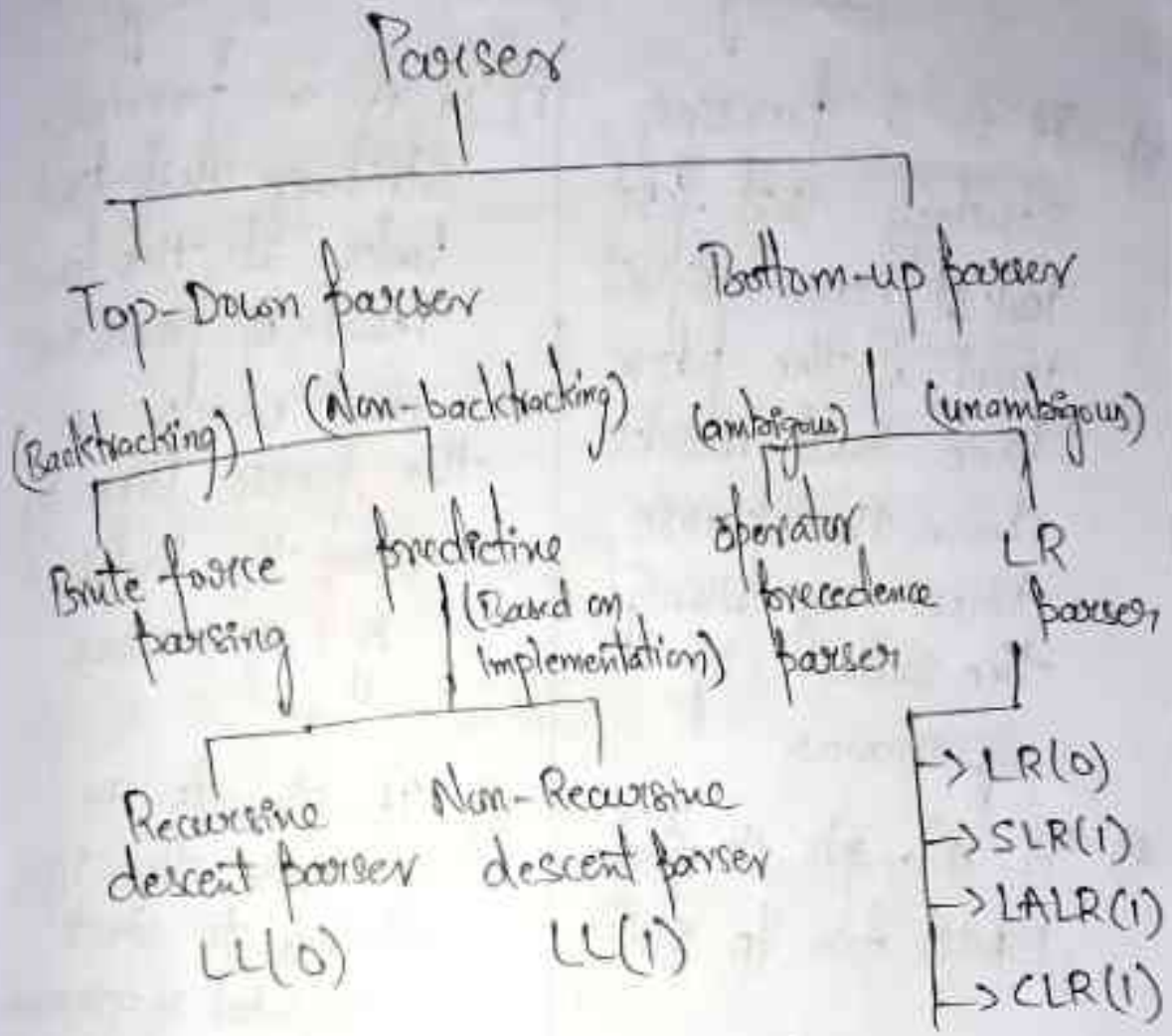
$S \rightarrow iEtSS' | a$

$S' \rightarrow eS | \in$

$E \rightarrow b$

//

* Parser (Types)



SLR(1) – Simple LR Parser
LALR(1) – Look Ahead LR Parser
CLR(1) – ~~Diagnostic~~ Canonical LR Parser

→ LL(0) or LL(1)
  - It Represent that Input is read from left to right
  - It Represent Left most Derivation

→ LR
  - It Represents Right most Derivation
  - It Represents that Input is read from left to right.

Top-Down parsing Vs Bottom up parsing.

Top-Down parsing	Bottom up parsing
1) It is a parsing strategy that first look at the highest level of the parse tree and works down the parse tree by using the rules of grammar	1) It is a parsing strategy that first looks at the lowest level of parse tree and works up the parse tree by using the rules of grammar
2) It attempts to find LMD for I/p string	2) It attempts to reduce the I/p string to start symbol of grammar
3) We start parsing from top to bottom	3) We start parsing from bottom to up
4) It use LMD	4) It use RMD
5) It's main decision is to select what production rule to use in order to construct the string	5) It's main decision is to select when to use a production rule to reduce the string to get starting symbol

6) It handles unambiguous grammar & grammar free from left recursion	6) It handles both ambiguous & unambiguous grammar
7) It involves production	7) It involves Reduction
8) Parsing table size is less	8) Parsing table size is more
9) It is less powerful	9) It is more powerful
10) Error detection is Easy	10) Error detection is difficult.

---

* Brute force parsing:-

-> It is a top-Down parser which involves backtracking.

-> Brute force strategy:-

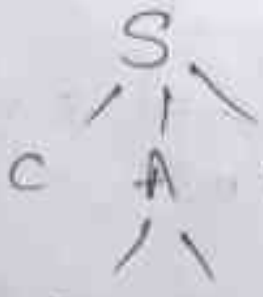
> Start with the start symbol and replace it with first Alternative.

> If there is a variable & it has many alternatives, then select the first alternative and compare leafs with given I/p string, If they are not Equal then select second alternatives. If all the alternatives are completed backtracks go back and substitute the second alternative. Continue the process until string is Accepted or Rejected.

Eg:-

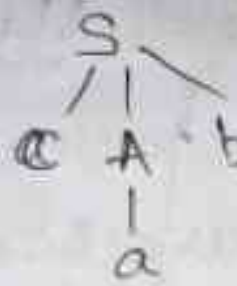
a) Show the acceptance of string cad for the grammar by brute-force method. S → cAd, A → ab|a

Sol $w = cad$



$w = cabd \neq cad$

It backtracks



$w = cab$ ✓

String is Accepted

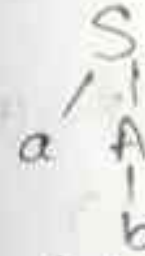
6) Show the acceptance of String $w = addc$ for

$S \to aAd \mid aB$

$A \to b \mid c$

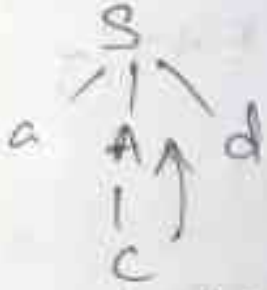
$B \to ccd \mid ddc$

Sol $w = addc$



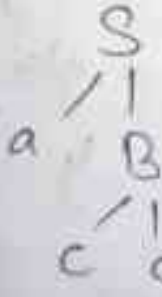
$abd \neq addc$

Backtracks →



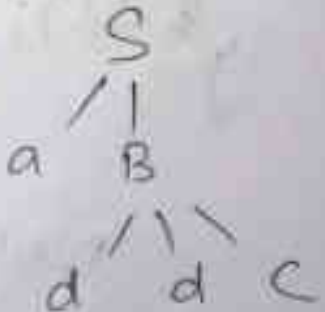
$acd \neq addc$

Backtracks



$accd \neq addc$

backtracks →



$w = addc$ ✓

String is Accepted

* Recursive descent Parser LL(0):-

-> In Recursive descent parser for each variable, we associate or write a procedure or function

Q) Construct LL(0) for following grammar

i) $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

1) $E \rightarrow E+T \mid T$ 2) $T \rightarrow T*F \mid F$ 3) $F \rightarrow (E) \mid id$

```
E()
{
  E();
  +
  T();
else
  T();
}
```

```
T()
{
  T();
  *
  F();
else
  F();
}
```

```
F()
{
  (
  E();
  )
else
  id
}
```

```
main()
{
  E(); // start symbol
}
```

ii) $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Sol LL(0) or Recursive descent parser

1) $E \rightarrow TE'$

```
E()
{
  T();
  E'();
}
```

2) $E' \rightarrow +TE' \mid \epsilon$

```
E'()
{
  +
  T();
  E'();
}
```

3) $T \rightarrow FT'$

```
T()
{
  F();
  T'();
}
```

4) $T' \rightarrow *FT' \mid \epsilon$

```
T'()
{
  *
  F();
  T'();
}
```

5) $F \rightarrow (E) \mid id$

```
F()
{
  (
  E();
  )
  else
  id
}
```

```
main()
{
  E();
}
```

* Non-Recursive Descent Parser
LL(1) parser

→ LL(1) Parser is a top-down parser without backtracking strategy.

→ LL(1)
- 1 → represent the number of look-ahead, which means how many symbols are we going to see when you want to make decision.
- 2nd L → represent that we will be using left most Derivation
- 1st L → represent that scanning of input is done from left to right.

→ LL(1) parser will have
- i) Parsing table – To check the grammar is in LL(1) or not
- ii) Driver Routine – It is used to check whether the string is accepted or not

* Algorithm to Construct LL(1) Parsing table:-

1) If grammar is left recursive then Eliminate the left recursion. The grammar must be free from left-recursion, ambiguity and left-factoring.
2) Identify null and non-null productions
3) Calculate First() for non-null production and Follow() for null production.
4) Draw the parsing table.

* First():-

→ If there is a variable, and from that variable, if we try to derive all the strings then beginning the terminal Symbol is called First.

Rules:-

i) If x is a terminal, then First(x) = {x}
ii) If $x \to \epsilon$, is a production rule then add ϵ to First(x)
iii) If $X \to Y_1 Y_2 Y_3 \ldots Y_n$ is a production,
1) First(x) = First(Y_1) If 2) First(Y_1) Contains ϵ then First(x) = {First(Y_1) − ϵ} ∪ {First(Y_2)}

3. If First(Y_i) Contains ϵ for all $i=1$ to n then add ϵ to First(X)

Eg:- 1) Production rules of Grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

FIRST Sets:-

FIRST (E) = { (, id }

FIRST (E') = { +, ϵ }

FIRST (T) = { (, id }

FIRST (T') = { *, ϵ }

FIRST (F) = { (, id }

ii) Find FIRST sets for the grammar

$S \rightarrow ACB \mid Cbb \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \epsilon$

$C \rightarrow h \mid \epsilon$

Sol

FIRST (S) = FIRST(ACB) U FIRST(Cbb) U FIRST(Ba)

= {FIRST(A) U FIRST(C) U FIRST(B)

= { d, g, h, b, a, ∈ }

FIRST (A) = { d, g, h, ∈ }

FIRST (B) = { g, ∈ }

FIRST (C) = { h, ∈ }

* FOLLOW(C):-

→ What is the terminal symbol which follows a variable in the process of derivation.

Rules:-

i) FOLLOW(S) = { $ } // S → start symbol

ii) If A → pBq is a production then Everything in FIRST(q) Except ∈ will be in FOLLOW(B)

iii) If A → pB is a production, then everything in FOLLOW(A) will be in FOLLOW(B)

iv) If A → pBq is a production and FIRST(q) Contains ∈ then FOLLOW(B) Contains

{ FIRST(q) − ∈ } U FOLLOW(A)

Eg:-

E → TE'
E' → +TE' | ∈
T → FT'
T' → *FT' | ∈
F → (E) | id.

FOLLOW(E) = { $,) }

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

FOLLOW(T) = { FIRST(E') − ϵ } ∪
{ FOLLOW(E') } ∪
{ FOLLOW(E) }
= { +, $,) }

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

FOLLOW(F) = { FIRST(T') − ϵ } ∪
{ FOLLOW(T') } ∪
{ FOLLOW(T) }
= { *, +, $,) }

Therefore:

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = { *, +, $,) }

8) Construct LL(1) Parser or check weather the given grammar is in LL(1) or not

i) $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id.$

Sol Variables = $V = \{E, T, F\}$

$T = \{+, *, (,), id\}$

The grammar is left recursive

Step1: Eliminate left recursion from grammar we get,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

$V = \{E, E', T, T', F\}$

$T = \{+, *, (,), id\}$

step 2:-

Non-Null productions

$E \rightarrow TE'$

$E' \rightarrow +TE'$

$T \rightarrow FT'$

$T' \rightarrow *FT'$

$F \rightarrow (E) | id$

Null productions

$E' \rightarrow \epsilon$

$T' \rightarrow \epsilon$

step 3:- Find FIRST() for non-null production and FOLLOW for null production

FIRST(E) = { (, id }

FIRST(E') = { + }

FIRST(T) = { (, id }

FIRST(T') = { * }

FIRST(F) = { (, id }

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

FOLLOW(T') = FOLLOW(T) = FIRST(E')

= { FIRST(E') - ε } ∪ FOLLOW(E')

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

Step 4:- Using FIRST() & FOLLOW() computed above fill the parsing table.

Parsing table:-

	+	*	(	)	id	$
E	-	-	$E \to TE'$	-	$E \to TE'$	-
E'	$E' \to +TE'$	-	-	$E' \to \epsilon$	-	$E' \to \epsilon$
T	-	-	$T \to FT'$	-	$T \to FT'$	-
T'	$T' \to \epsilon$	$T' \to *FT'$	-	$T' \to \epsilon$	-	$T' \to \epsilon$
F	-	-	$F \to (E)$	-	$F \to id$	-

→ The above parsing table does not contain multiple entries in single cell therefore the grammar given is in "LL(1)".

Ex) $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

Sol)
$V = \{ S, E \}$
$T = \{ i, t, e, a, b \}$

Step 1: For Constructing LL(1) the grammar must be free from ambiguity, left recursion and left factor

The above grammar contains left factor
Eliminating left factor.

$S \rightarrow iEtSS' \mid a$
$S' \rightarrow eS \mid \epsilon$
$E \rightarrow b$

$V = \{ S, S', E \}$
$T = \{ i, t, e, a, b \}$

Step 2:- Identify null & non-null productions

Non-Null
$S \rightarrow iEtSS' \mid a$
$E \rightarrow b$
$S' \rightarrow eS$

Null
$S' \rightarrow \epsilon$

Step 3:- Calculate FIRST() for non-null production & FOLLOW() for null productions.

FIRST(S) = { i, a }
FIRST(E) = { b }
FIRST(S') = { e }

FOLLOW(S') = FOLLOW(S)
= { FIRST(S) - ε } ∪ FOLLOW(S)
FOLLOW(S') = { e, $ }

Step 4:- Construct Parse table using above FIRST() & FOLLOW

Parse table:-

	i	e	a	t	b	$
S	S→iEtSS'	-	S→a	-	-	-
S'	-	S'→eS S'→ε	-	-	-	S'→ε
E	-	-	-	-	E→b	

The parse table contains multiple entries [S', e]
Therefore it is not in LL(1).

* Driver Routine :-

→ It is used to check wheather the string is accepted by the grammar or not.

→ It Contains i) Stack
ii) Input tape

Stack

Input tape

→ Rules :-

i) If x is a top of stack & a is an Input symbol if $x = a = \$$ then string is accepted.

ii) If $x = a \neq \$$, then pop x and advancing Input pointer

iii) If $x \neq a$, then look into parse table and write RHS of the production in reverse order.

Q) Check whether the string id+id*id is accepted by

$E \to TE'$

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

$T \to FT'$

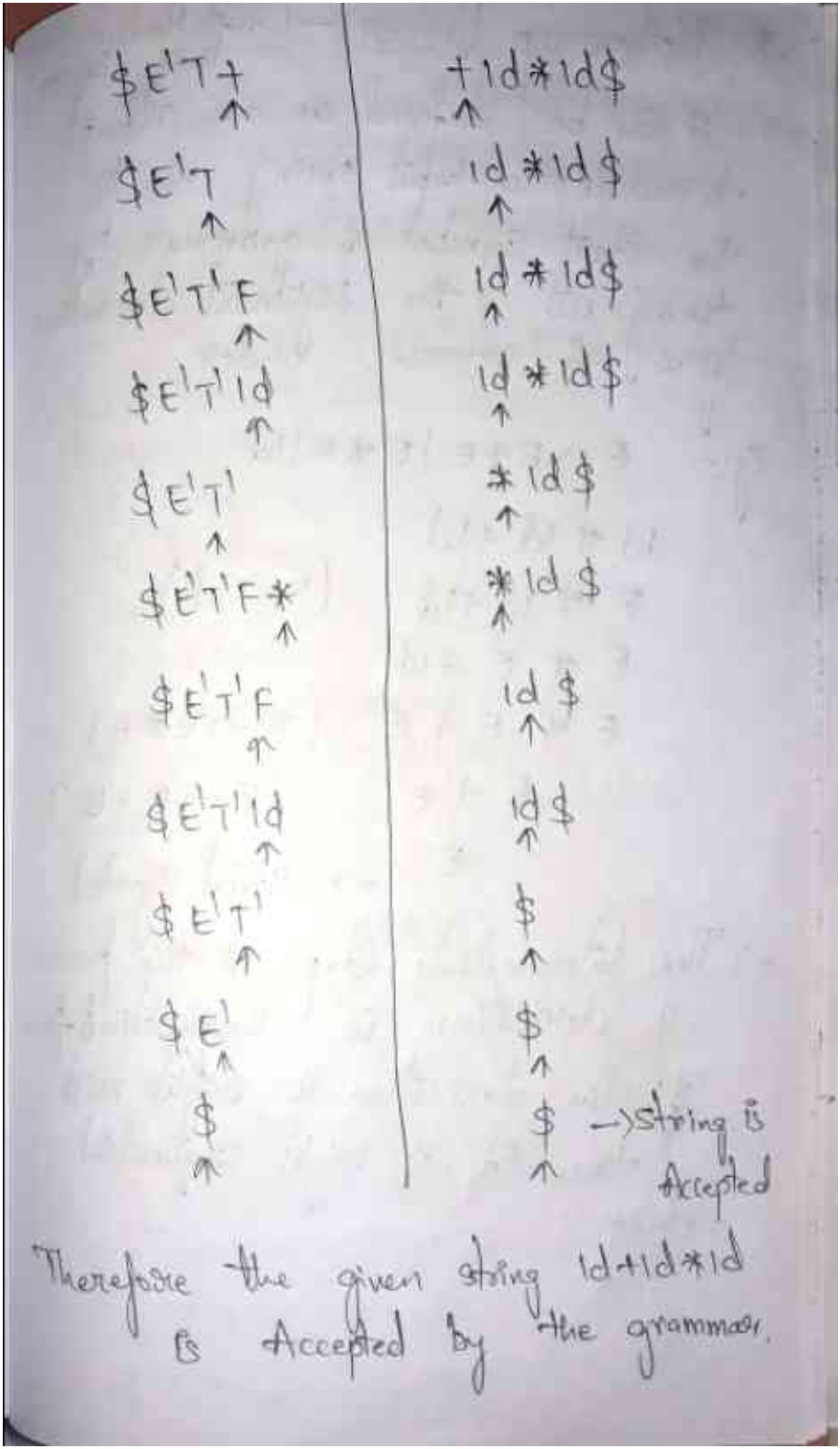
$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

Using LL(1) parse table.

Sol Note:- The LL(1) parse table is constructed above in LL(1) parser topic

Stack	I/p string
$E	id+id*id$
$E'T	id+id*id$
$E'T'F	id+id*id$
$E'T'id	id+id*id$
$E'T'	+id*id$
$E'	+id*id$



Therefore the given string id+id*id is Accepted by the grammar.

* Bottom-up Parser:- (Shift-Reduce Parser)

→ It can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse order

Eg:- $E \to E+E \mid E*E \mid id$

id * id + id
E * id + id (E → id)
E * E + id
E * E + E (E → E*E)
E + E (E → E+E)
E → start symbol

→ The intermediate steps in the process of derivation is sequential form.
If the derivation is right most then it is right sequential form

→ The task in bottom up parser is to find the substring that could be reduced by appropriate variable such a substring is called as handle.

→ The Operations involved in bottom-up parser are

i) Shift
ii) Reduce
iii) Accept
iv) Error

Eg:- E → E+E|E*E|id
w = id + id * id.

Stack	Input	Action
$	id+id*id$	
$id	+id*id$	Shift
$E	+id*id$	Reduce
$E+	id*id$	Shift
$E+id	*id$	Shift
$E+E	*id$	Reduce
$E	*id$	Reduce
$E*	id$	shift
$E*id	$	shift

$E*E $ Reduce

$E $ Reduce

↳ Start symbol.

∴ The string is Accepted.

→ Bottom up parser is used for both ambiguous and unambiguous grammar.

→ Bottom-up parser

↓

LR Parsers

LR(0) SLR(1) CLR(1) LALR(1)

* LR parser

→ LR parsers are bottom up parser which handles unambiguous grammar

L → left to right Scanning i/p

R → RMD in reverse order

Types:- SLR – Simple LR
CLR – Canonical LR
LALR – Look Ahead LR

→ In LR parsing, the rows corresponds to items and Columns are divided into two parts: 1) Terminal part called as Action part
2) Variable part called goto part.

→ Items: – Introducing a • (dot) on RHS forms an item. It signifies what is parsed and what is yet to parse.

Eg:- $A \rightarrow .XYZ$ → X is parsed

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$ → XY is parsed & Z is yet to parse

$A \rightarrow XYZ.$ → Complete item

* Steps in Construction of LR parser:-

1) Construct Augmented grammar
2) Canonical Collection of LR(0) items
3) Construction of DFA
4) Construction of Parse table
5) Driver Routine.

Augmented Grammar:-

→ Including an additional start symbol forms an Augmented grammar.

Rules for Canonical Collection of LR:-

i) Write all production in Augmented grammar in I(0) & place dot at beginning of RHS

ii) If there is a Variable after dot place all productions of that Variable in that item & place a Dot at beginning of RHS

Note:- IN SLR(1) of parsing table, we use FOLLOW() to place Reduce Operations.

Parse table

a) Construct SLR(1) for below grammar

i) $E \to E+T \mid T$.

$T \to T*F \mid F$

$F \to (E) \mid id$

Sol Step 1:- Augmented grammar

$E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

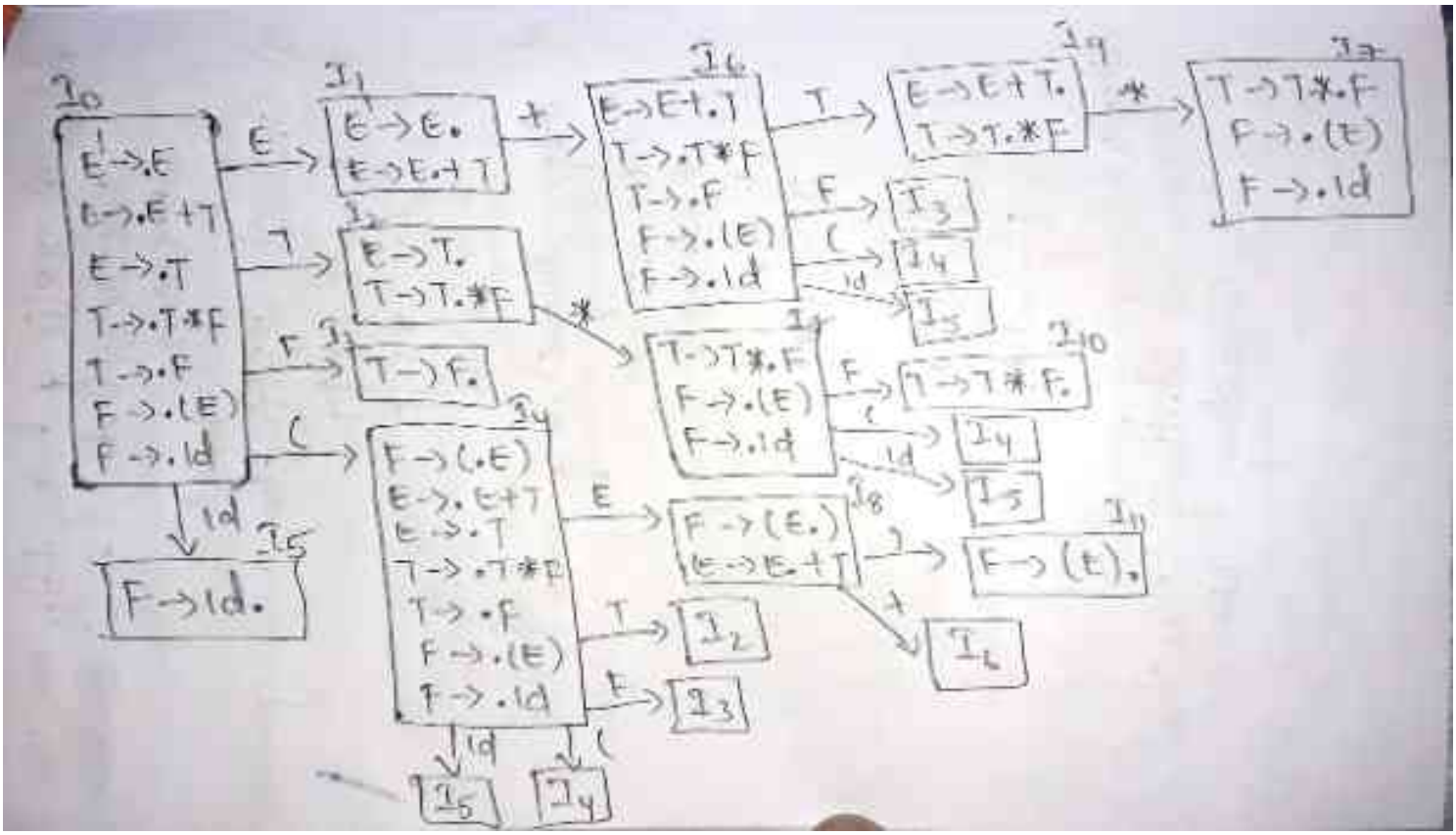
$T \to .F$

$F \to .(E)$

$F \to .id$.

Step 2 & 3:-

Canonical Collection of LR(0) items & Constructing DFA



		Action: Item in which it is Completely parsed	Goto: FOLLOW(LHS)
$E \to E+T$	ϑ_1	I_9	FOLLOW(E) = {$, +,)}
$E \to T$	ϑ_2	I_2	
$T \to T*F$	ϑ_3	I_{10}	FOLLOW(T) = {$, +, *,)}
$T \to F$	ϑ_4	I_3	
$F \to (E)$	ϑ_5	I_{11}	FOLLOW(F) = {$, +, *,)}
$F \to id$	ϑ_6	I_5	

Parsing table (Parse table)

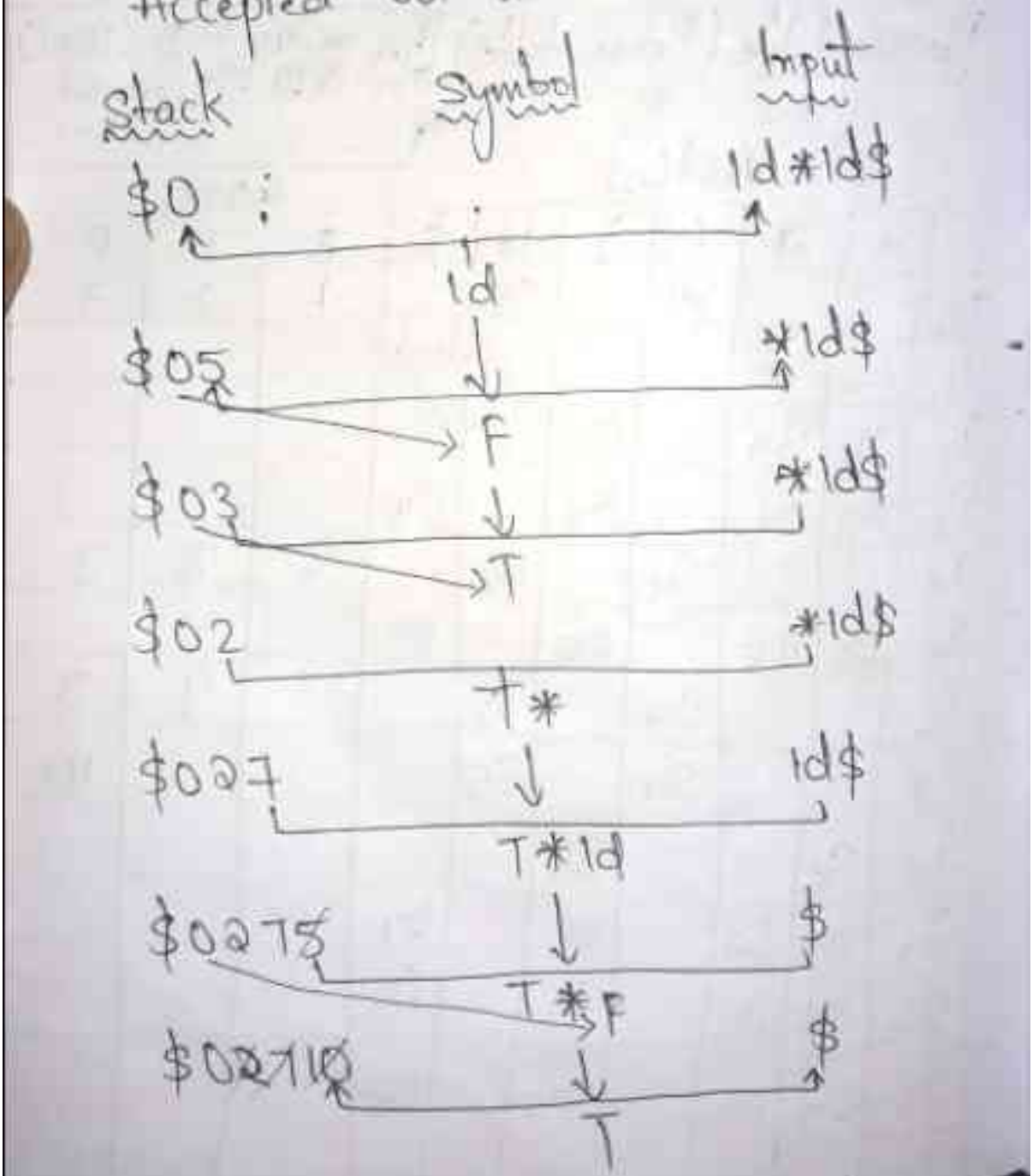
For reduce use → FOLLOW()
For shift use → Canonical Items

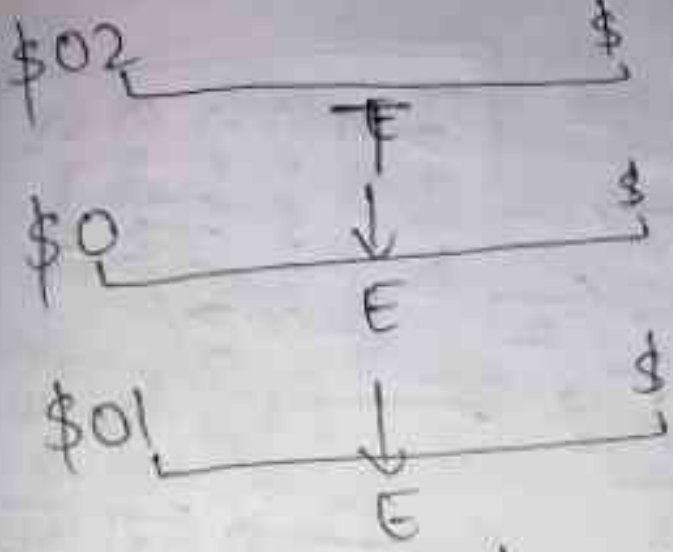
	Action						GOTO		
	+	*	(	)	id	$	E	T	F
I_0			S_4		S_5		1	2	3
I_1	S_6					Accepted			
I_2	ϑ_2	S_7		ϑ_2		ϑ_2			
I_3	ϑ_4	ϑ_4		ϑ_4		ϑ_4			
I_4			S_4		S_5		8	2	3
I_5	ϑ_6	ϑ_6		ϑ_6		ϑ_6			
I_6			S_4		S_5			9	3
I_7			S_4		S_5				10
I_8	S_6			S_{11}					
I_9	ϑ_1	S_7		ϑ_1		ϑ_1			
I_{10}	ϑ_3	ϑ_3		ϑ_3		ϑ_3			
I_{11}	ϑ_5	ϑ_5		ϑ_5		ϑ_5			

→ The above grammar is not SLR(1), since no cell contains multiple entries.

* Driver Routine:-

→ Using the above parse table we check whether the string is Accepted or not.





∴ String is Accepted
(since, I1 on $ is Accepted)

ii) S -> A|bAC|dC|bda
A -> d. Check whether the grammar is in SLR(1) or not

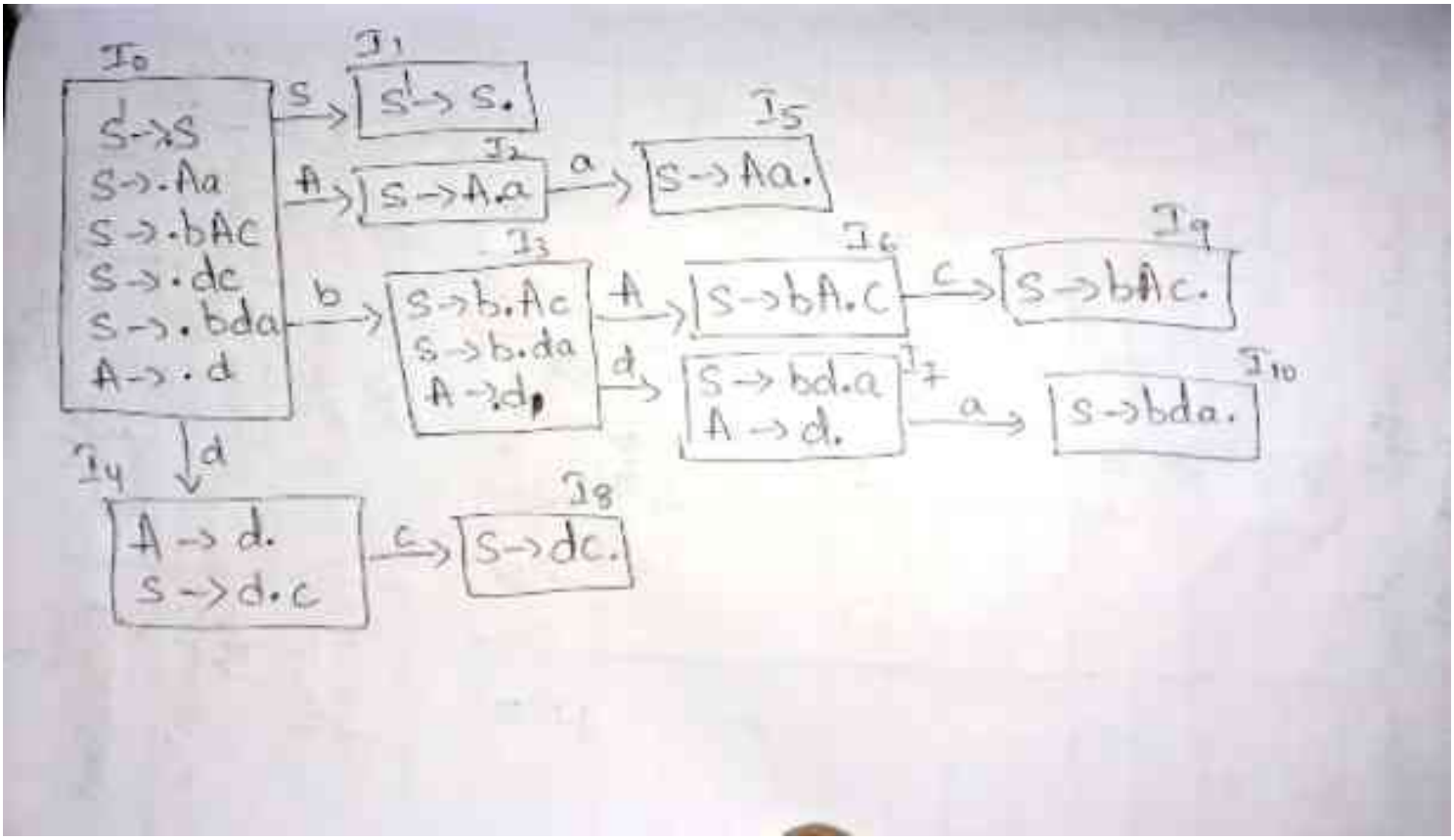
Sol

S -> A|a|bAc|dc|bda
A -> d

Augumented Grammar:-

S' -> .S
S -> .Aa
S -> .bAc
S -> .dc
S -> .bda
A -> .d

Canonical Collection of LR(0) items.



		Item in which it is completely placed	FOLLOW(LHS)
S → Aa	σ_1	I_5	FOLLOW(S) = {$, $y}
S → bAc	σ_2	I_9	
S → dc	σ_3	I_8	FOLLOW(A) = {a, c}
S → bda	σ_4	I_{10}	
A → d	σ_5	I_4 & I_7	

Parse table :-

	a	b	c	d	$	S	A
I_0		S_3		S_4		1	2
I_1					Accept		
I_2	S_5						
I_3				S_7			6
I_4			S_8				
I_5					σ_1		
I_6			S_9				
I_7	σ_5, S_{10}		σ_5				
I_8					σ_3		
I_9					σ_2		
I_{10}					σ_4		

I_7, a} contains both Shift & Reduce operation. Therefore the given grammar is not in SLR(1).

* Canonical LR parser:- CLR(1)

→ In CLR(1) parser we use symbols called look aheads to place 'reduce' Operation in parse table.

→ With each production rule we associate a look ahead.

→ The look ahead for start symbol is $

→ If $A \to \alpha . B\beta , a$ then all
(a = look ahead)

productions starting with B will have look ahead as FIRST(βa)

→ In SLR(1) we used LR(0) canonical items, but in CLR(1) and LALR(1) we make use of LR(1) canonical Items.

{ LR(1) item = LR(0) item + lookahead }

→ The look ahead is used to determine where we place the final item.

→ The look ahead always add $ symbol for augumented production.

Conflicts in LR parser:-

i) Shift Reduce Conflict
ii) Reduce Reduce Conflict

Q) Construct CLR(1) parse table for the following grammar.

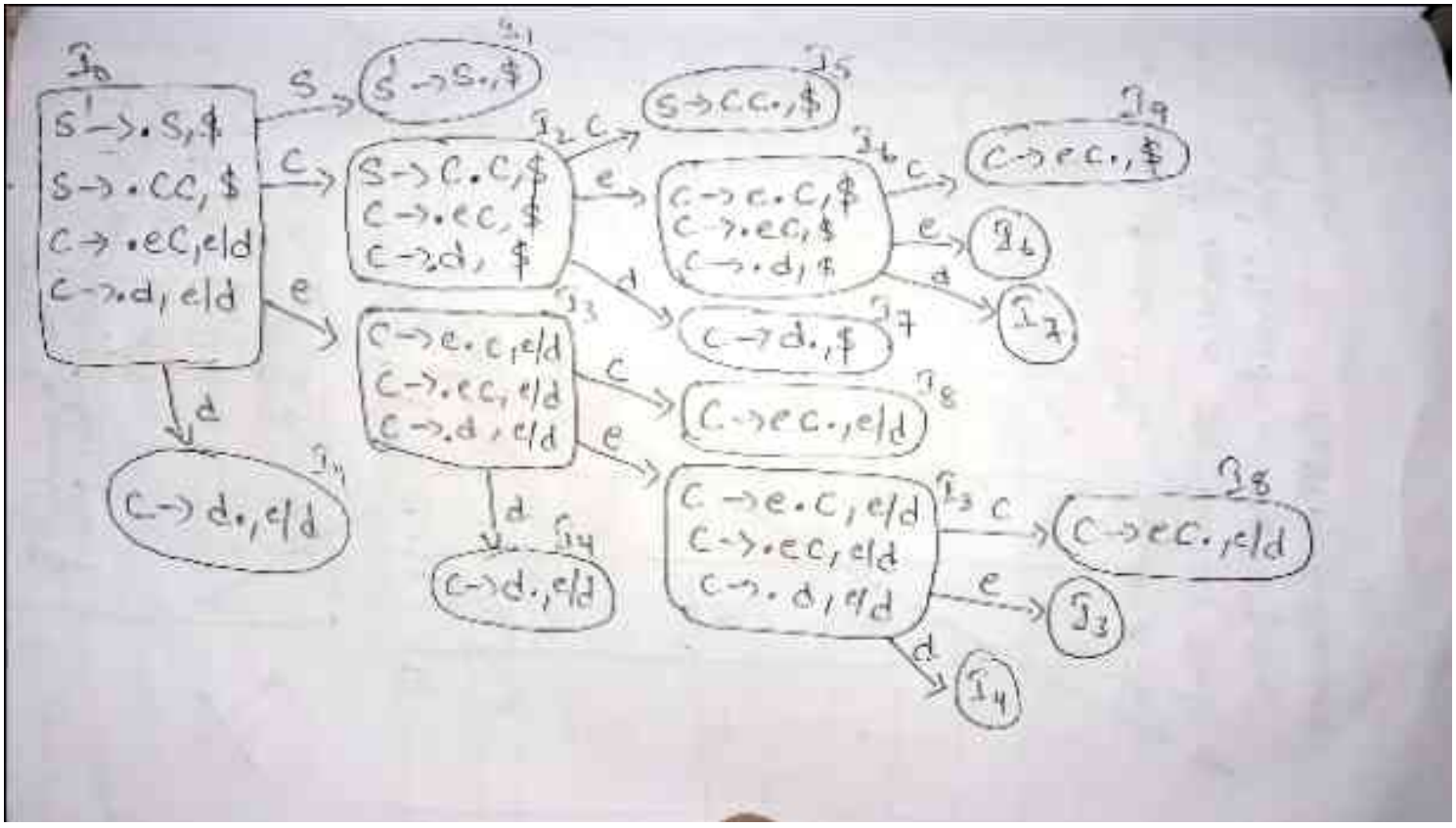
1) S → CC
   C → eC
   C → d

Sol I) Augmented Grammar:-

S' → . S, $
S → . CC, $
C → . eC, e/d
C → . d , e/d

FIRST(CC)
C → FIRST(C,$)
FIRST(e,$)
or
FIRST(d,$)
→ e/d

II) Construct LR(1) Canonical Items
III) for the above grammar



		Item in which completely parsed
$S \to CC$	θ_1	I_5
$C \to eC$	θ_2	I_8, I_9
$C \to d$	θ_3	I_4, I_7

Parse table:-

	ACTION			GOTO	
	e	d	$	S	C
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	θ_3	θ_3			
I_5			θ_1		
I_6	S_6	S_7			9
I_7			θ_3		
I_8	θ_2	θ_2			
I_9			θ_2		

The above grammar is CLR(1) grammar since Parse table does not contain Shift Reduce Conflict.

ii) $S \rightarrow L = R \mid R$

$L \rightarrow * R \mid id$

$R \rightarrow L$

Sol Augumented Grammar

$S \rightarrow . L = R, \$$

$S \rightarrow . R, \$$

$L \rightarrow . * R$

$L \rightarrow . id$

$R \rightarrow . L$

Look-Ahead for L :-

$S \rightarrow . L = R, \$$

It is in the form $A \rightarrow \alpha . B\beta \mid a$

$B = L \quad \beta = = R \quad a = \$$

Look Ahead of $L = FIRST(\beta a)$

$= = \mid \$$

Look Ahead for R :-

$S \rightarrow . R, \$$

Look Ahead for $R = \$$

Now,

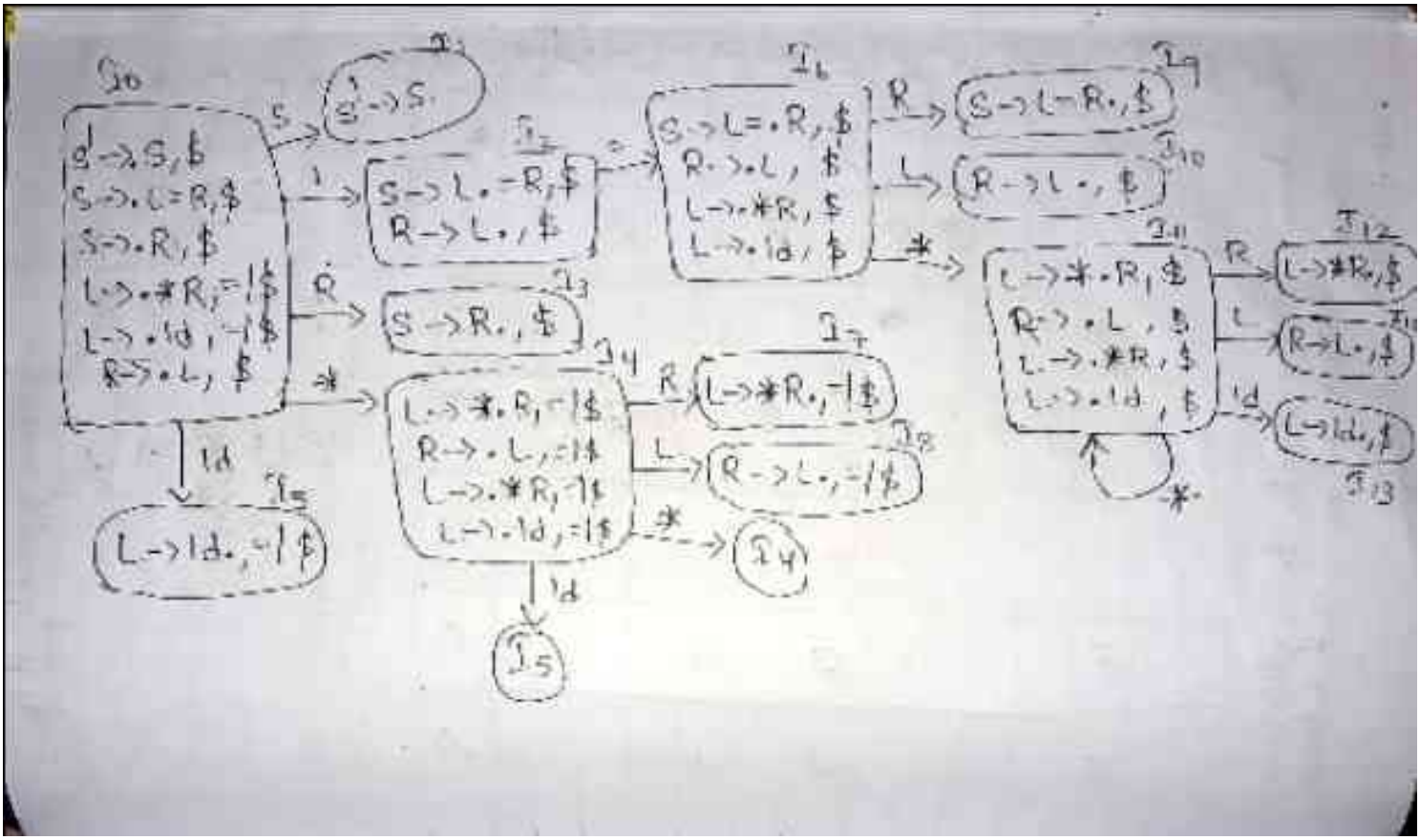
$S \rightarrow . L = R, \$$

$S \rightarrow . R, \$$

$L \rightarrow . * R, = \mid \$$

$L \rightarrow . id, = \mid \$$

$R \rightarrow . L, \$$



		Item in which it is completely parsed
$S \to L = R$	g_1	I_9
$S \to R$	g_2	I_3
$L \to *R$	g_3	I_4, I_{12}
$L \to id$	g_4	I_5, I_{13}
$R \to L$	g_5	I_2, I_8, I_{10}

Parse Table:-

	*	=	id	$	S	L	R
I_0	S_4		S_5		1	2	3
I_1				Accept			
I_2		S_6		g_5			
I_3				g_2			
I_4	S_4		S_5			8	7
I_5		g_4		g_4			
I_6	S_{11}					10	9
I_7		g_3		g_3			
I_8		g_5		g_5			
I_9				g_1			
I_{10}				g_5			
I_{11}	S_{11}		S_{13}			10	12
I_{12}				g_3			
I_{13}				g_4			

The above grammar is an CLR(1) since there is no Shift reduce conflict.

* Look-Ahead LR parser:— LALR(1)

→ In LALR we combine the items having common core but different look aheads.

→ LALR(1) parsing is same as CLR(1) parsing, only difference in the parsing table.

8) Construct LALR(1) parse table for the following grammar.

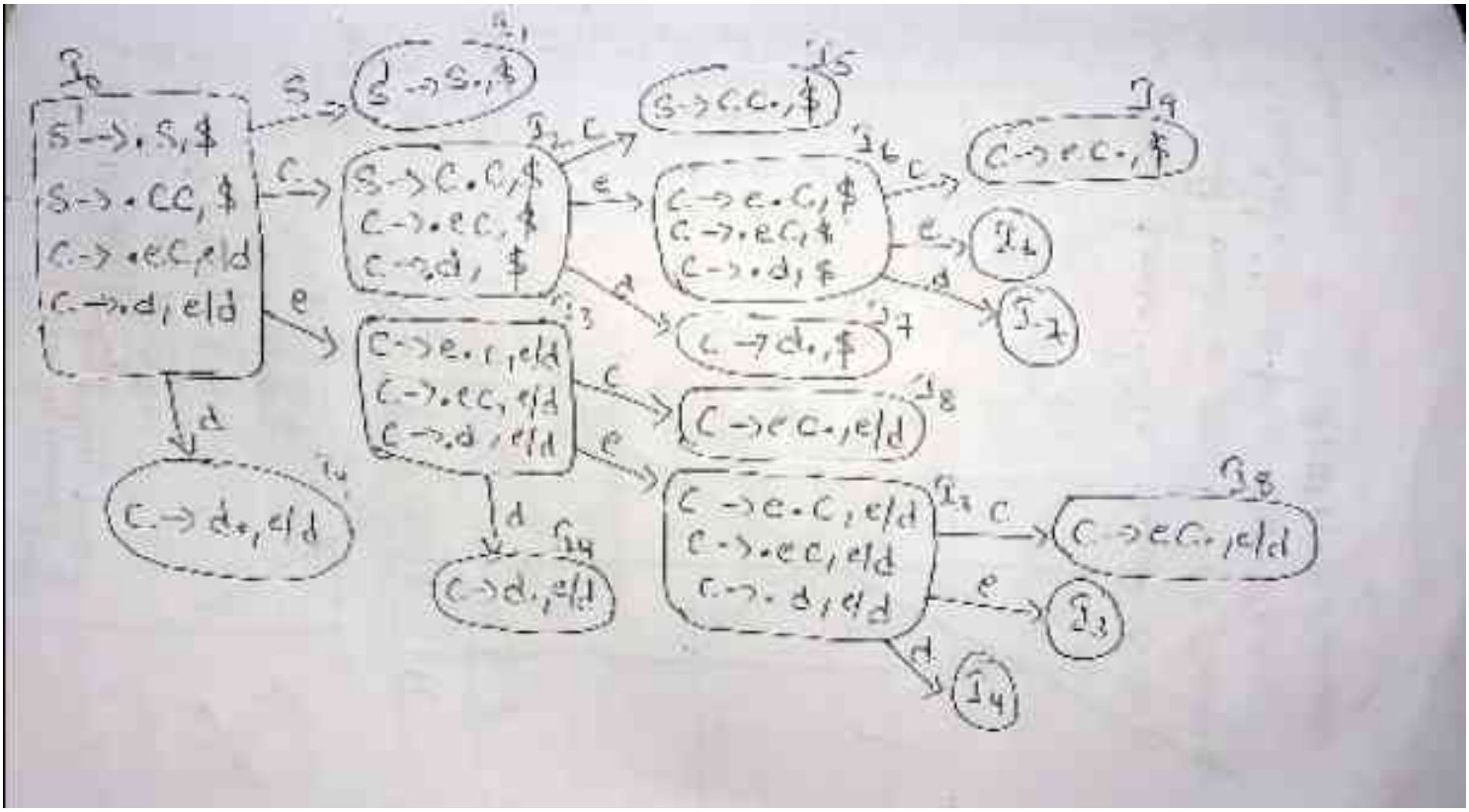
i) S → CC
C → eC
C → d.

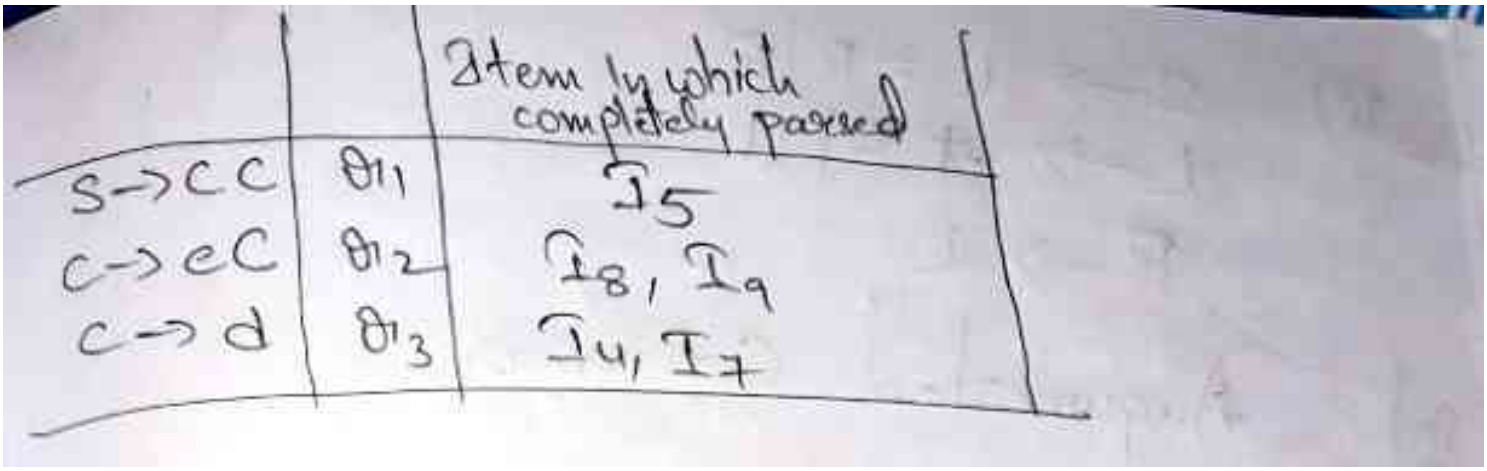
Sol Augmented Grammar:—

S' → . S, $
S → . CC, $
C → . eC
C → d

FIRST(C)
C → yFIRST(C$)
→ e/d.

Canonical Collection of LR(1) items





→ The above canonical LR(1) items have common core part but different look aheads. Therefore, Combine common core part and then Construct parse table.

→ $I_3 \cup I_6 \cong I_{36} = I_3$

$I_4 \cup I_7 \cong I_{47} = I_4$

$I_8 \cup I_9 \cong I_{89} = I_8$

Parse table

	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			
I_4	γ_3	γ_3	γ_3		
I_5			γ_1		
I_8	γ_2	γ_2	γ_2		

The above parse table doesn't contain shift reduce conflict. Therefore The grammar is in LALR(1).

ii) $S \rightarrow Aa|bAc|Bc|bBa$

$A \rightarrow d$

$B \rightarrow d$

Augmented grammar.

$S' \rightarrow .S, \$$

$S \rightarrow .Aa, \$$

$S \rightarrow .bAc, \$$

$S \rightarrow .Bc, \$$

$S \rightarrow .bBa, \$$

$A \rightarrow d, a$

$B \rightarrow d, c$

Look Ahead for A :-

$S \rightarrow .Aa, \$$ ($A \rightarrow \alpha . B\beta, a$)

FIRST(βa)

Look Ahead for A : FIRST($a\$$)

$= a$

Look Ahead for B :-

$S \rightarrow .Bc, \$$

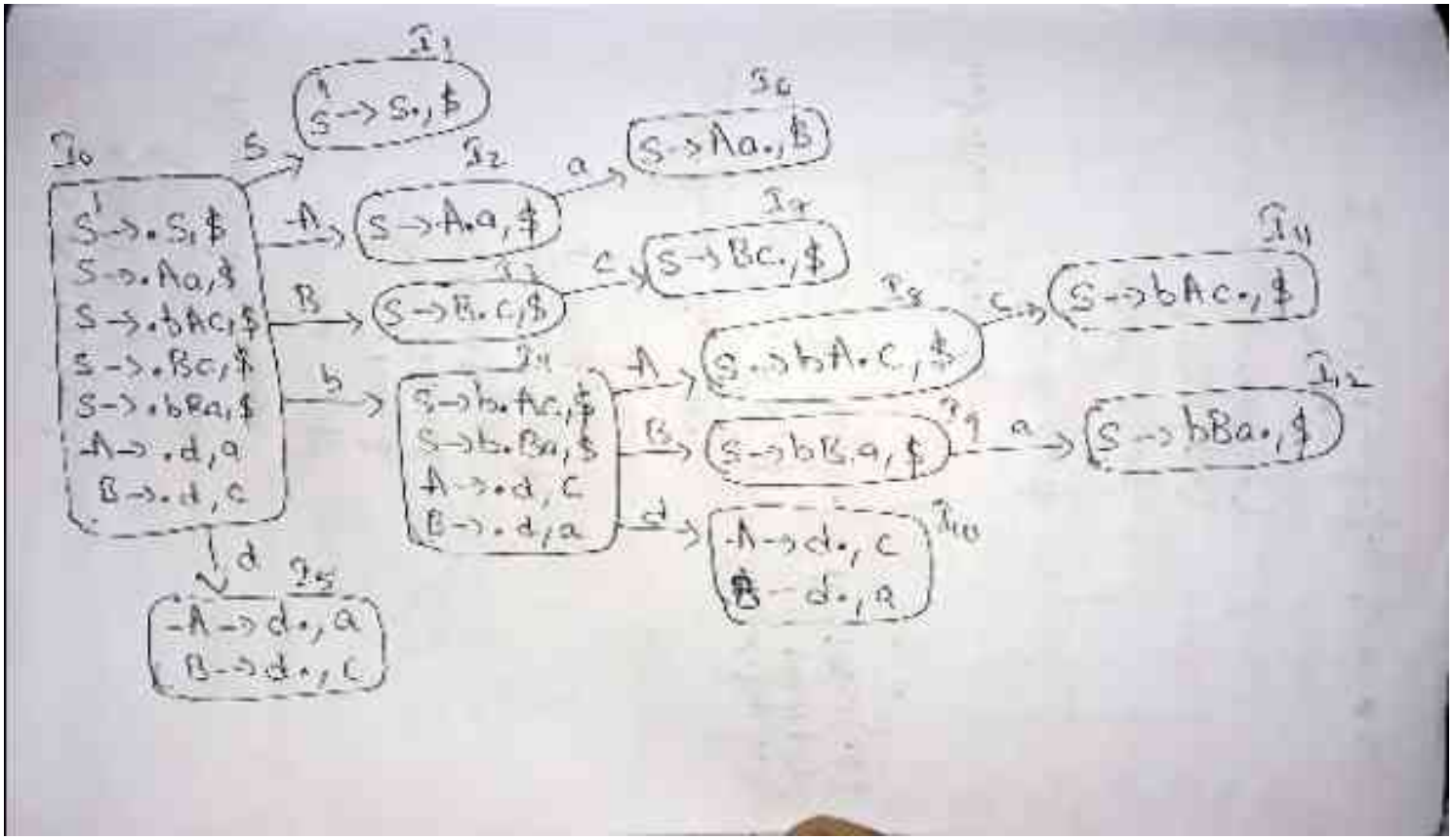
$A \rightarrow \alpha . B\beta, a$

$\rightarrow$ FIRST(βa)

$\rightarrow$ FIRST($c\$$)

$\rightarrow c$

$\$ \Leftrightarrow$ Look Ahead for $B = c$



		State in which it is completely parsed.
S → Aa	ϑ_1	I_6
S → bAc	ϑ_2	I_{11}
S → Bc	ϑ_3	I_7
S → bBa	ϑ_4	I_{12}
A → d	ϑ_5	I_5 & I_{10}
B → d	ϑ_6	I_5 & I_{10}

I_5 & I_{10} has same core part & different look aheads so we combine I_5 & I_{10}

$I_5 \cup I_{10} \cong I_{510} = I_5$

Parse table:-

	a	b	c	d	$	S	A	B
I_0		S_4		S_5		1	2	3
I_1					Accept			
I_2	S_6							
I_3			S_7					
I_4				S_5			8	9
I_5	ϑ_5, ϑ_6		ϑ_5, ϑ_6					
I_6					ϑ_1			
I_7					ϑ_3			
I_8			S_{11}					
I_9	S_{12}							
I_{11}					ϑ_2			
I_{12}					ϑ_4			

The above parse table contains multiple entries i.e Reduce Reduce conflict. So the grammar is not in LALR(1).

Note:- The above grammar is not in LALR(1) but it is in CLR(1).

* Parser Generator YACC :-

-> YACC stands for Yet Another Compiler Compiler

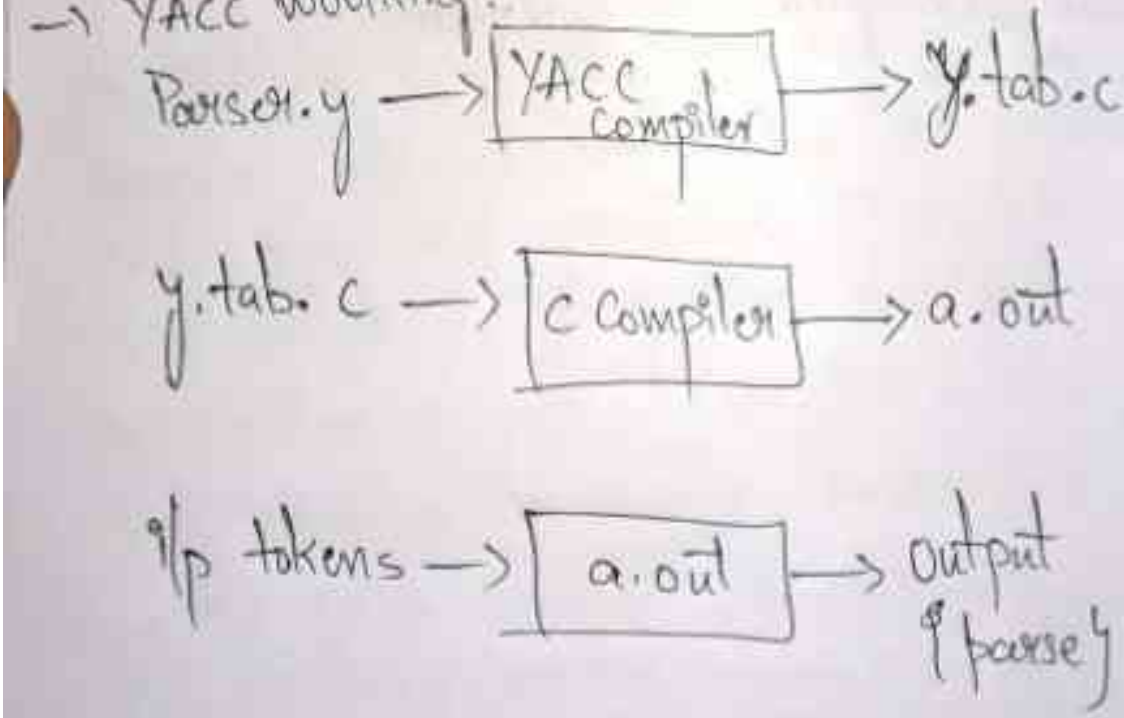
-> YACC is an automated tool to produce a parser for given grammar

-> YACC is a program designed to Compile a LALR(1) grammar.

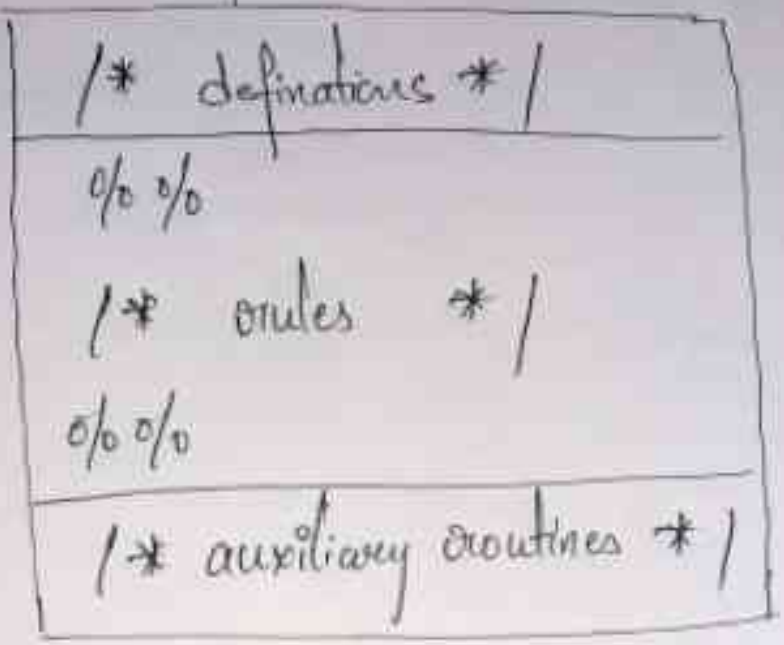
-> The input of YACC is the rule or grammar and output is C program.

-> YACC file has .y extension.

-> YACC working :-



-> YACC Input file is divided into three parts.



-> The defination part includes information about tokens used in syntax defination

-> The rules part contains grammar defination in modified BNF Form. Actions in C code in { } and can be embedded inside

-> The Auxiliary routines part is only C code. It includes definations for every function needed in rules part.

-> It contains the main() function and main function must call yyparse().

* Syntax Error handling & Recovery:-

→ In this phase of compilation, all possible errors made by the user are detected and reported to the user in the form of error message.

→ This process of locating errors are reported it to user is called Error handling process.

→ Functions of Error handler:-

1) Detection
2) Reporting
3) Recovery

→ Common Errors that can occur

i) lexical errors include misspellings of identifiers, keyword or Operators.

ii) syntactic errors include misplaced semicolons or extra or missing braces.

iii) Semantic errors include type mismatches between Operators & Operands.

iv) logical errors can be anything from incorrect reasoning.

→ The precision of parsing methods allows syntactic errors to be detected very efficiently.

→ Several parsing methods, such as LL & LR methods, detect an error as soon as possible.

→ The error handler in a parser has goals that are simple to state but challenging to realize:

i) Report the presence of errors clearly & accurately.

ii) Recover from error quickly enough to detect subsequent errors.

iii) Add minimal overhead to the processing of correct programs.

* Error Recovery Strategies:—

→ Once the error is detected following strategies are used to recover from that errors.

1) Panic Mode Recovery:—

→ In this method, successive characters from input are removed one at

a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimeters such as ; or }

→ Advantage of this method is that its easy to implement and guarantees not to go in infinite loop.

→ Disadvantage is that a considerable amount of input is skipped without checking it for additional errors.

2) Phrase-level Recovery:-

→ In this method, when a parser encounters an error, it performs necessary corrections on remaining input so that the rest of the input statements allow the parser to parse ahead.

→ The correction can be deletion of extra semicolons, replacing comma, by semicolon or inserting missing semicolon.

→ While performing correction, almost care should be taken for not going in infinite loop

→ Disadvantage is that it finds difficult to handle situation where actual errors occured before point of detection.

3) Error Productions :-

→ If User has knowledge for common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.

→ If this is used then, during parsing appropriate error message can be generated and parsing can be continued.

→ It is difficult to maintain

4) Global Correction :-

→ The parser examines the whole program and tries to find out the closet match for it which is error free.

→ The closet match program has less number of insertions, deletions and changes of tokens to recover from erroneous input

→ Due to high time and space complexity, this method is not implemented practically.

* Parsing Ambiguous grammar:-

→ LR parser can be used to parse ambiguous grammar. LR parser resolves the conflicts (shift/reduce or reduce/reduce) in parsing table of ambiguous grammar based on certain rules (precedence or associativity) of grammar.

NYAAC:-

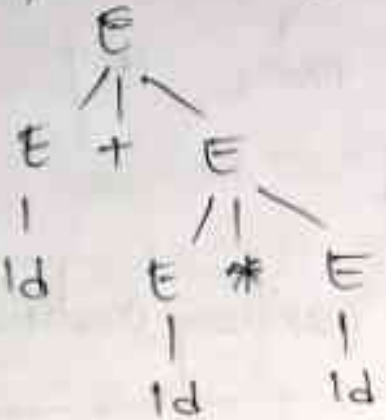
→ When shift-reduce conflict occurs, it is resolved by giving priority to shift move over reduce move. If the string is accepted for shift move, then reduced move is removed, otherwise shift move is removed.

→ When reduce-reduce conflict occurs, it is resolved by giving priority to first reduce move over second reduce move. If the string is accepted for first reduce move, then second reduce move is removed, otherwise first reduce move is removed.

UNIT:-03 Syntax-Directed Translation

* CFG:- $E \rightarrow E+E \mid E*E \mid id$

Parse tree for above grammar



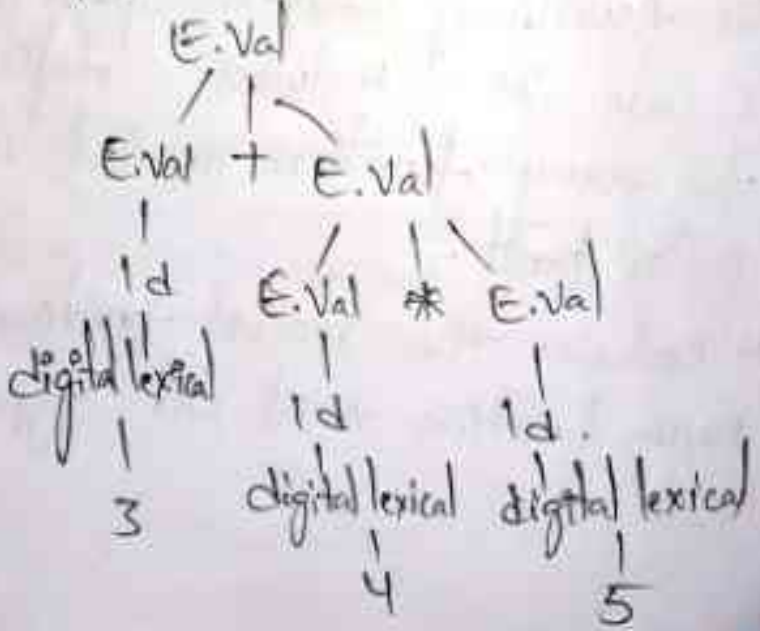
Syntax directed defination

E.Val → E.Val + E.Val

E.Val → E.Val * E.Val

E.Val → id

Annotated parse tree



* Syntax Directed Translation:-

→ Syntax Directed Translation are augmented rules to the grammar that facilitates semantic analysis. SDT involves passing information bottom-up / top-down the parse tree in the form of attributes attached to nodes.

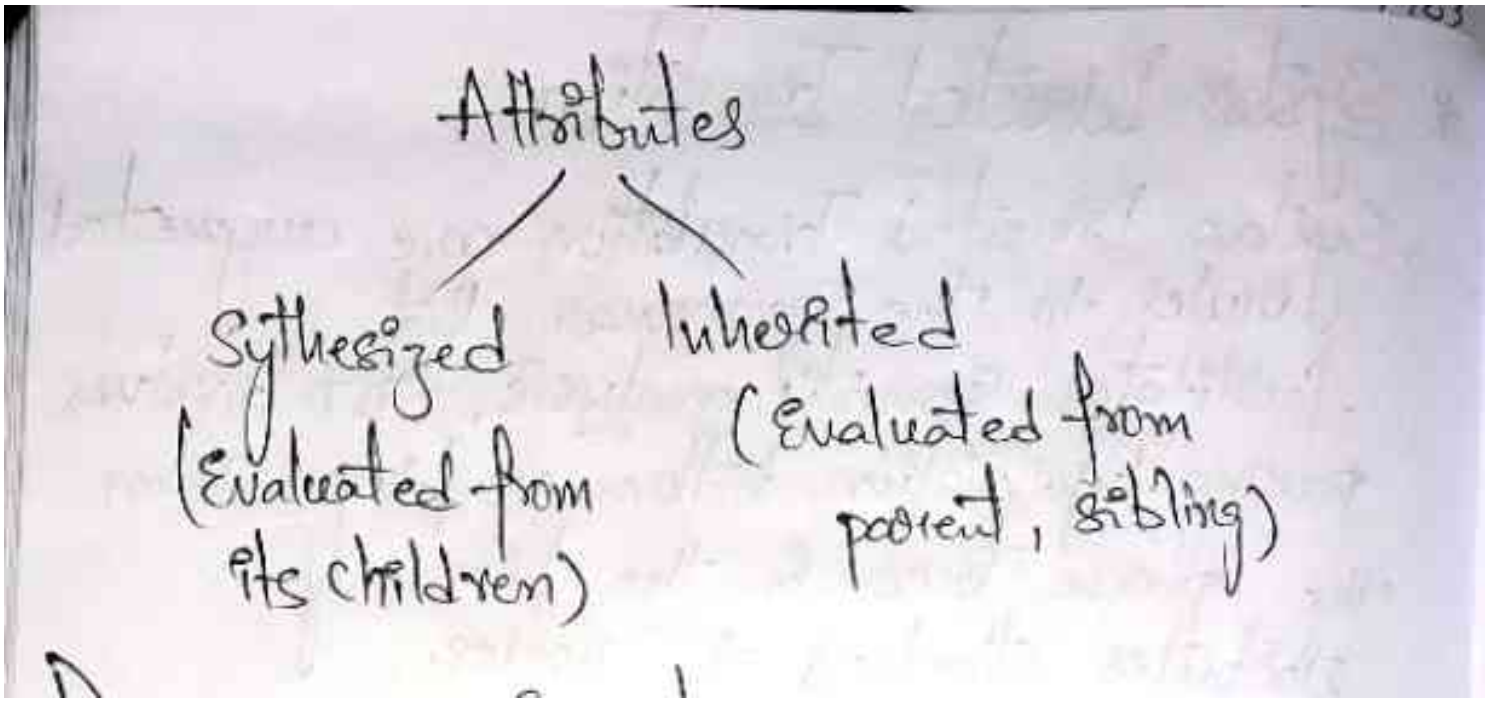
→ In SDT along with grammar we associate some informal notation and these notations are called Semantic Rules.

Grammar + Semantic Rule = SDT

→ In SDT, every non-terminal can get one or more than one attribute or sometime 0 attribute depending upon the type of attribute. The value of these attribute is evaluated by semantic rules associated with production rule.

Syntax directed Translation / Defination

- L-attributed
  - (higher class)
  - (left to right)
  - (post order traversal)
  - (Uses Synthesized & Inherited attribute with restrictions)
- S-attributed
  - (Inner class/subclass)
  - (Bottom up parser)
  - (only Synthesized attribute)



* Example of Synthesize Attribute

$E \rightarrow E_1 + E_2 \{ E.val = E_1.val + E_2.val \}$

* Example of Inherited Attribute

$A \rightarrow XYZ \{ Y.val = 2 * A.val \}$

* Annotated parse tree :-

→ A parse tree showing the values of attribute at each node is called as Annotated parse tree

→ The process of Computing the attributes value at nodes is called annotating of the parse tree

Dependency Graph:-

→ A Graph representing dependency b/w the attributes is the dependency graph.

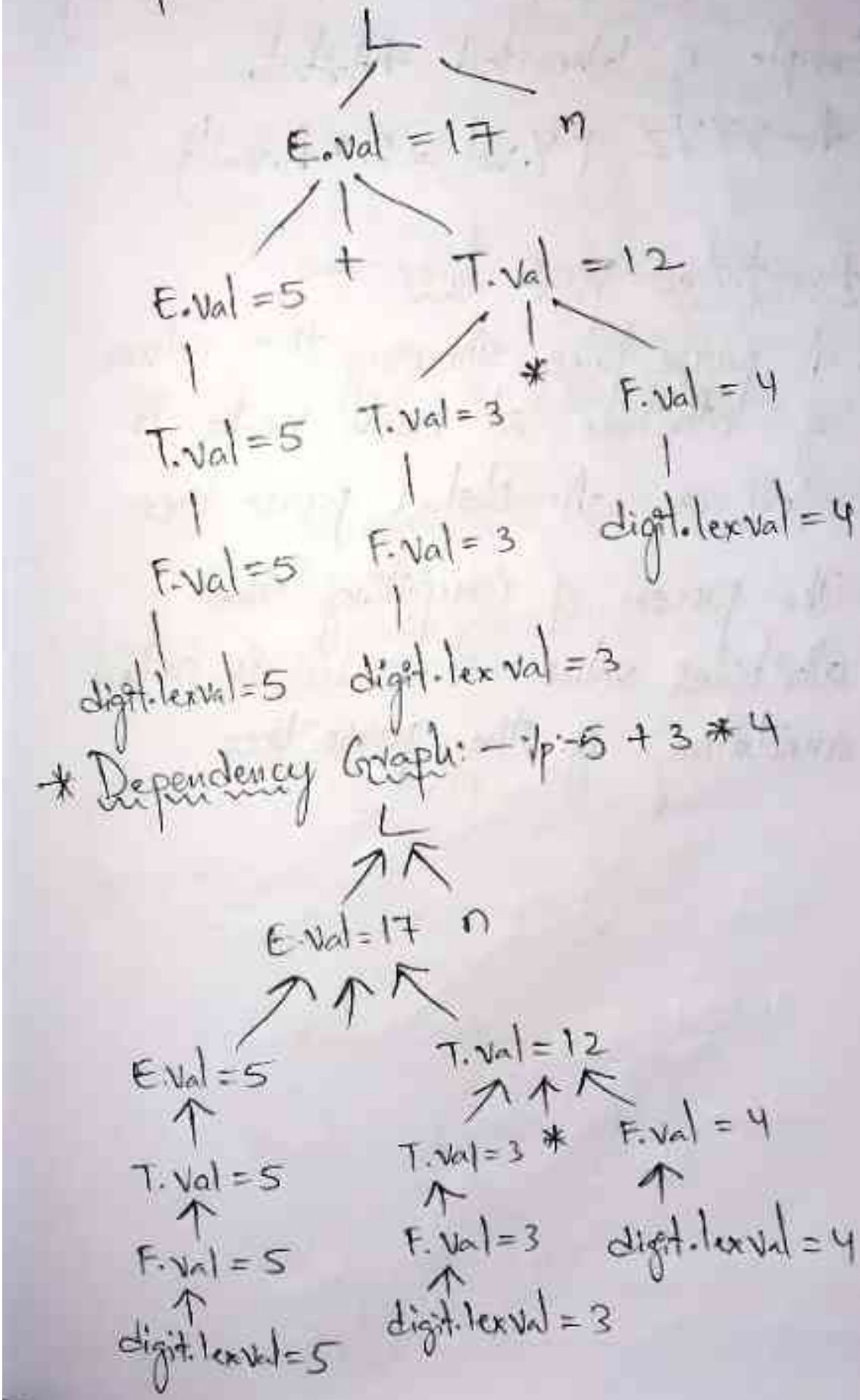
* Syntax Directed Defination :- Example (Synthesized)

Production	Semantic Rule
$L \rightarrow En$	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow digit$	$F.val = digit.lexval$

Note :- Val & lexval are Synthesized Attribute
Therefore, above SDD is S-Attributed.

* Annotated parse tree

Input:- 5 + 3 * 4



* Dependency Graph:- I/p:- 5 + 3 * 4

* Syntax Directed Definition :- (Inherited Attribute)

(i)

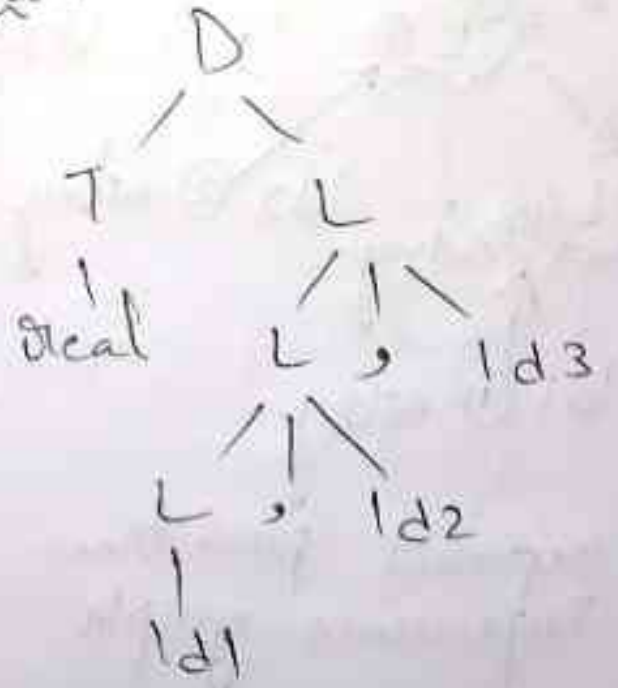
Production	Semantic Rule
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L, id$	$L.in = L.in$ $addtype(id.entry, L.in)$
$L \rightarrow id$	$addtype(id.entry, L.in)$

→ Symbol T is associated with synthesized attribute type

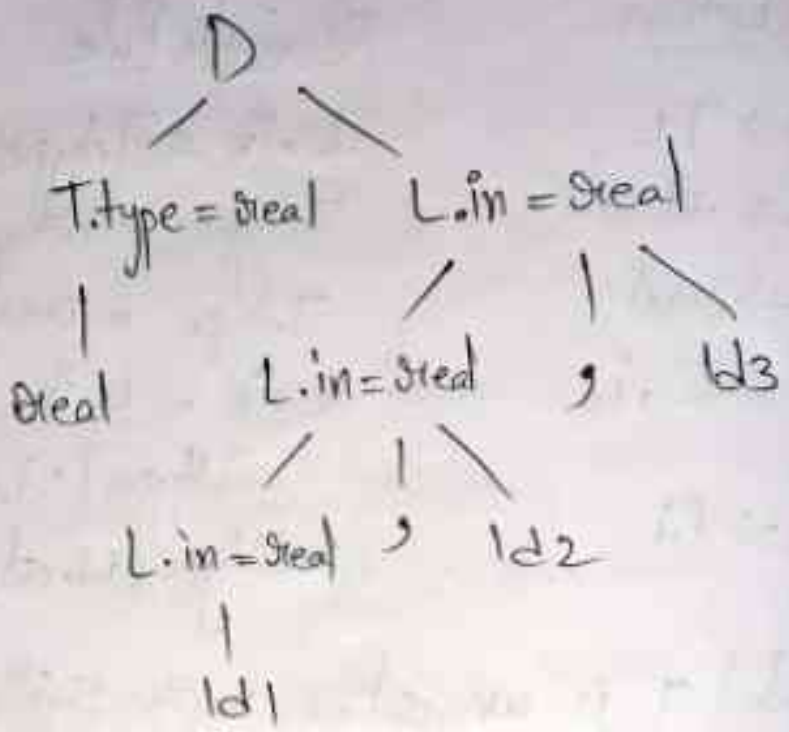
→ Symbol L is associated with an inherited attribute in

Input:- real p, q, r

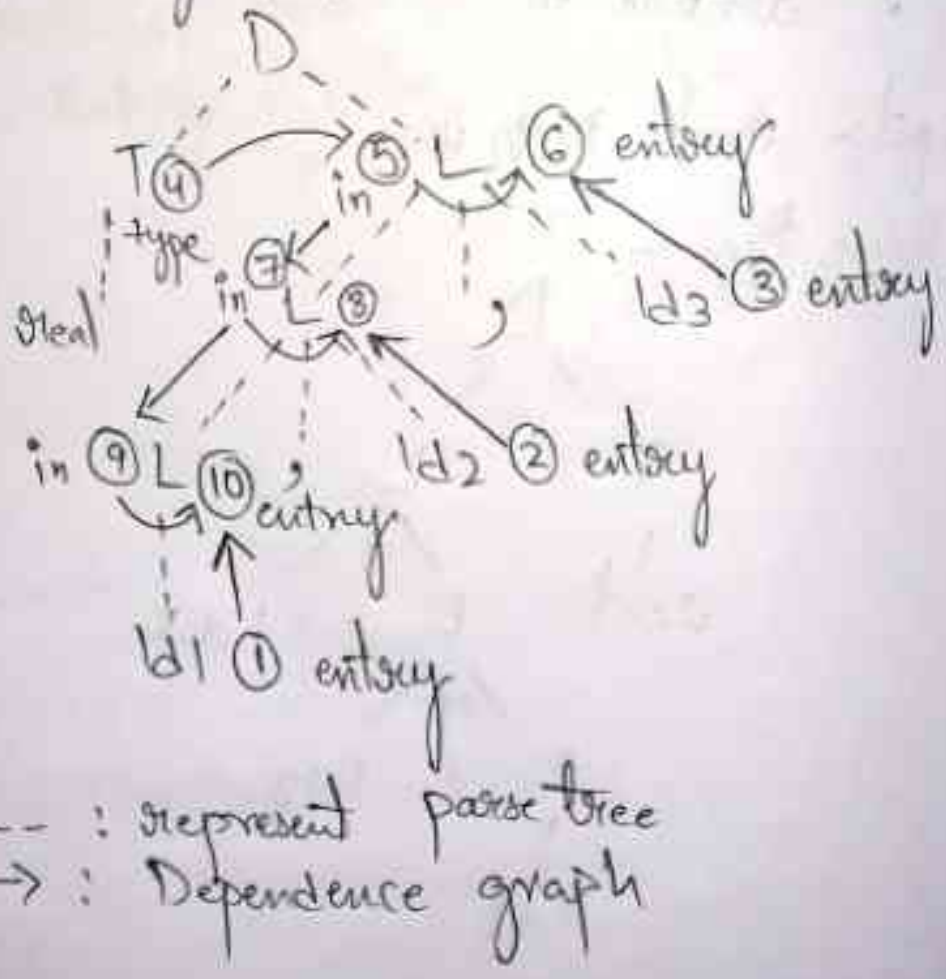
Parse tree :-



Annotated parse tree:-



Dependency graph:-



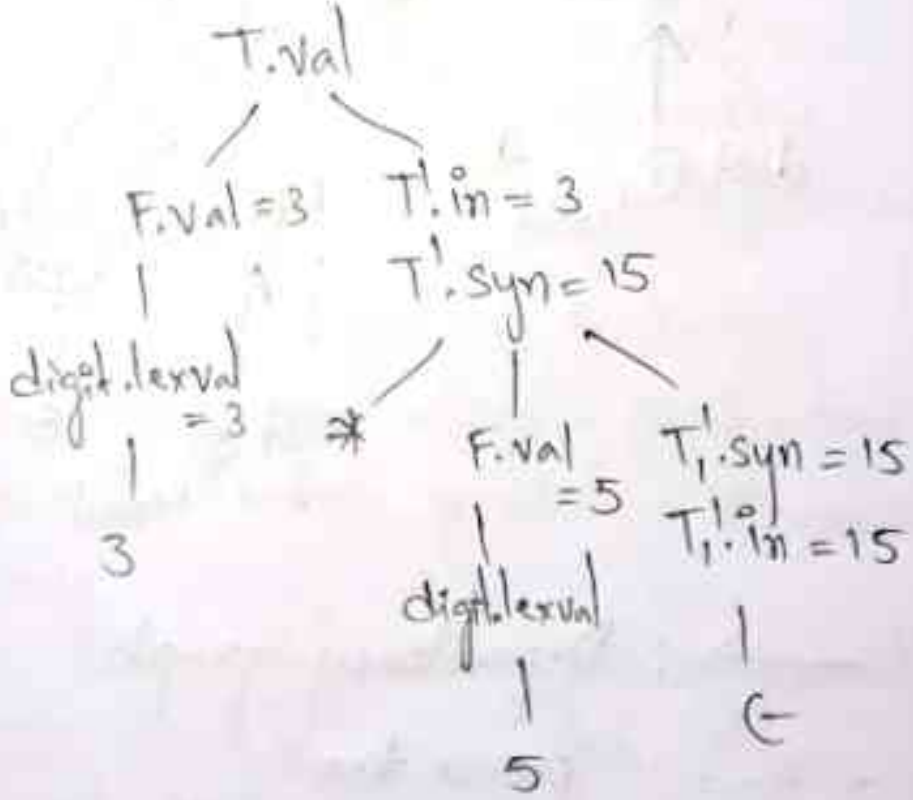
---- : represent parse tree

⟶ : Dependence graph

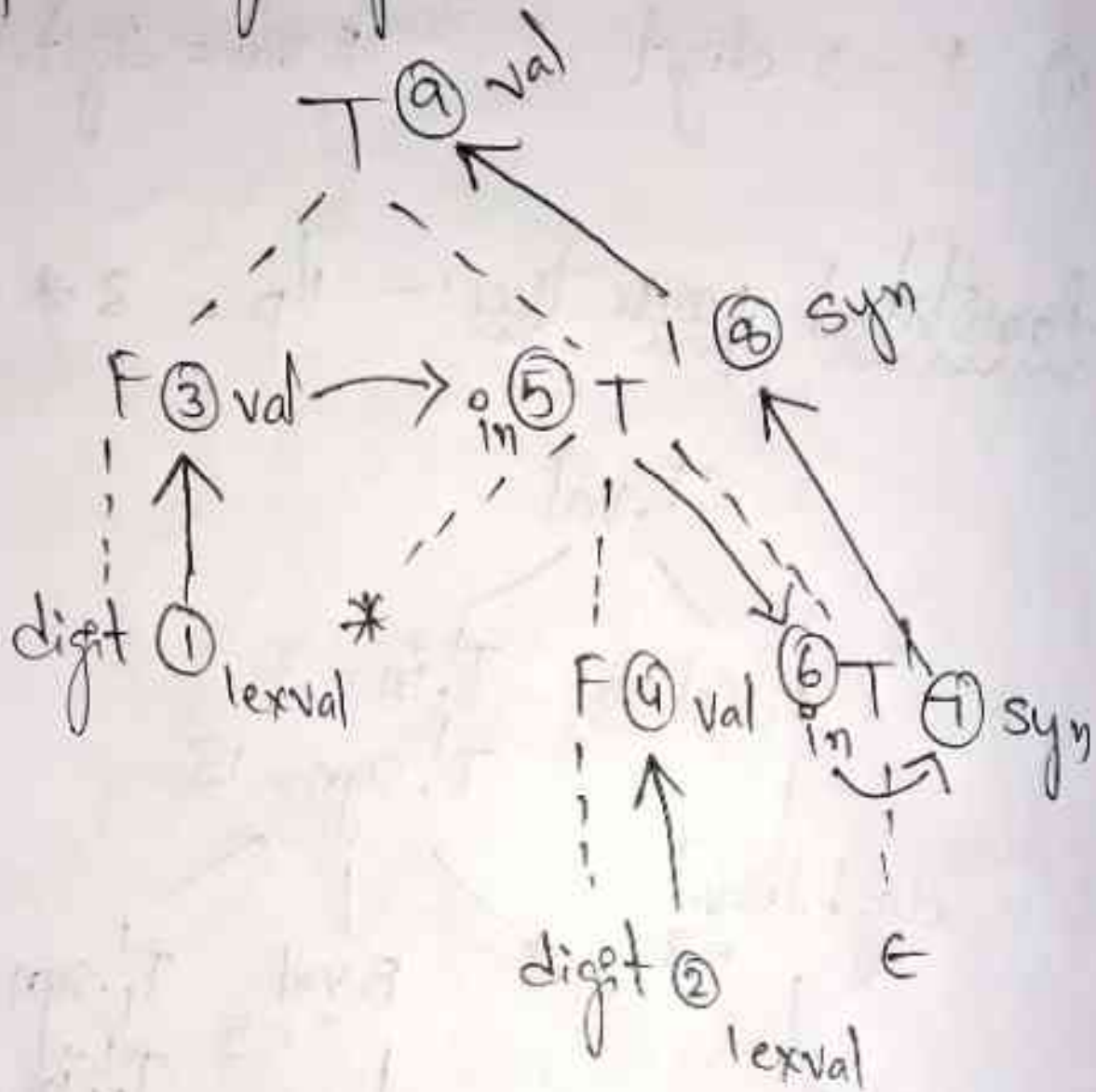
(ii)

Production	Semantic Rule
1) $T \rightarrow F T'$	$T'.in = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T_1'$	$T_1'.in = T'.in * F.val$ $T'.syn = T_1'.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.in$
4) $F \rightarrow$ digit	$F.val =$ digit.lexval

* Annotated parse tree:- I/p = 3 * 5



Dependency graph:-



⟶ : Dependency graph

- - - - : parse tree.

*Applications of Syntax Directed Translation

→ It is used for semantic analysis and SDT is basically used to construct parse tree with grammar and semantic action.

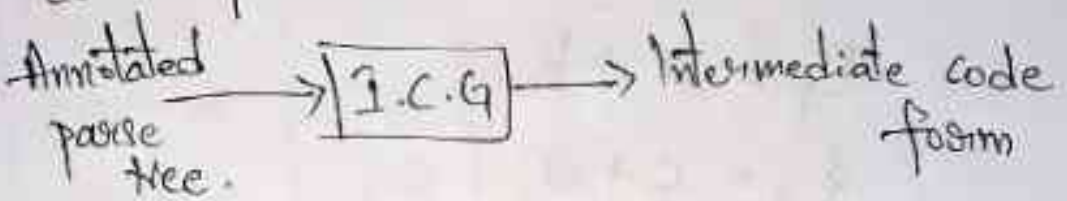
→ In grammar, need to decide who has the highest priority will be done first and in Semantic Action, will decide type of action done by grammar.

→ Applications:—

i) SDT are used for Executing Arithematic Expressions
ii) Conversion from Infix to postfix
iii) Conversion from Infix to prefix
iv) It is also used in Binary to decimal conversion
v) In Counting number of Reduction
vi) It is used in Creating a syntax tree
vii) SDT is used to generate Intermediate Code
viii) In Storing information into symbol table
ix) SDT is also used for type Checking.

UNIT:-04 Intermediate Code generation

-> Input to Intermediate code generation phase is Annotated parse tree and output is any intermediate code form.



-> Syntax tree:-

It is a tree representation of an expression. In syntax tree nodes are operators and leaves are operands.

-> 3-Address Code:-

The instructions expressed by three addresses. The various 3-address code representation are

i) Quadruple
ii) Triple
iii) Indirect triple

6) Write 3-address code for the expressions.

i) $y = (a+b) * (c+d)$

→ Three address code

$$t_1 = a + b$$
$$t_2 = c + d$$
$$t_3 = t_1 * t_2$$
$$y = t_3$$

Quadruple :-

Opr	arg1	arg2	Result
+	a	b	t_1
+	c	d	t_2
*	t_1	t_2	t_3
=	t_3	-	y

Triple:-

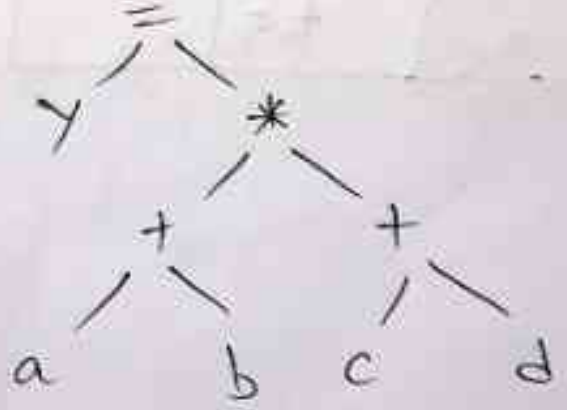
→ In triple representation, we don't use temporary variable

	op1	arg1	arg2
(0)	+	a	b
(1)	+	c	d
(2)	*	(0)	(1)
(3)	=	2	Y

Indirect triple:-

Address	Pointer to Triple
(0)	100
(1)	101
(2)	102
[3]	103

Syntax tree:-



OR
11) $a = b * -c + b * -c$

↳ unary minus

→ 3-address code:-

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

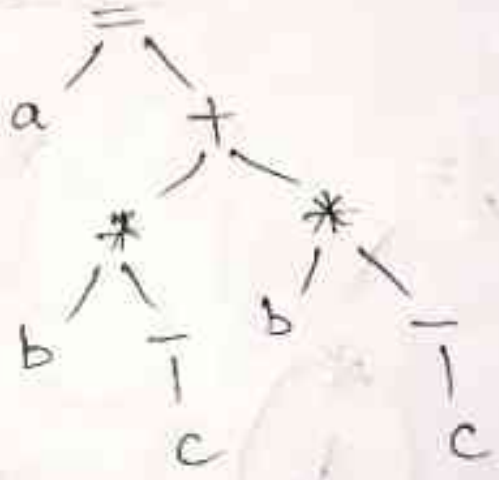
Quadruple:-

Oppr	arg1	arg2	Result
-	c		t_1
*	b	t_1	t_2
-	c		t_3
*	b	t_3	t_4
+	t_2	t_4	t_5
=	t_5		a

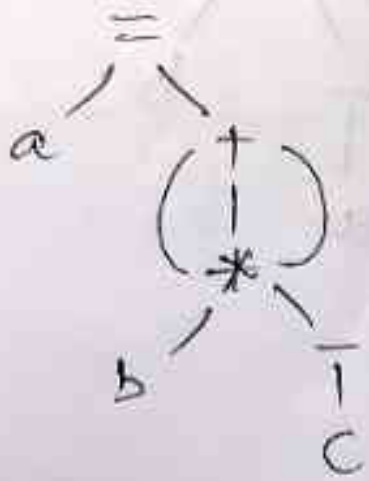
Triple:-

	op	arg1	arg2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Syntax Tree:-



Directed Acyclic Graph [DAG]:-



In DAG we combine common sub expression

iii) DAG for i) $x = -a+b*-a+b$

→ 3-address code

$t_1 = -a$

$t_2 = t_1 + b$

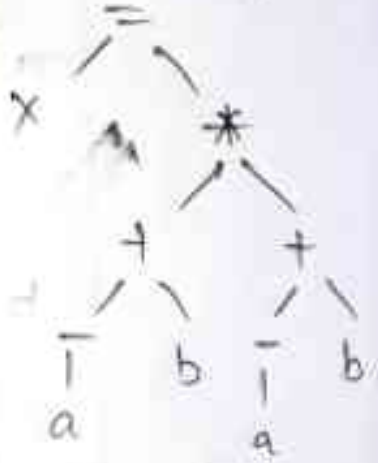
$t_3 = -a$

$t_4 = t_3 + b$

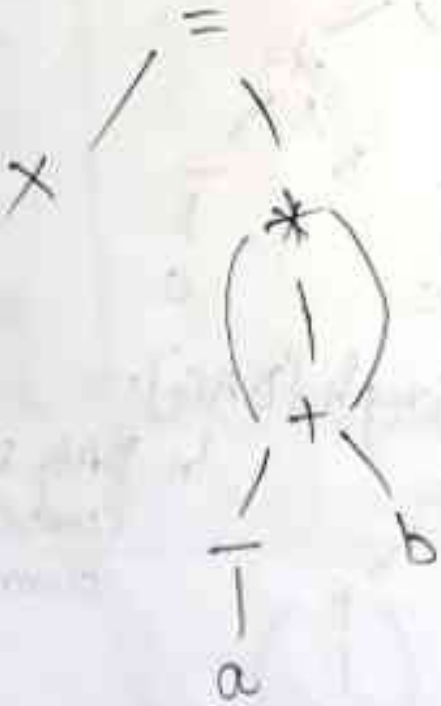
$t_5 = t_2 * t_4$

$x = t_5$

Syntax tree

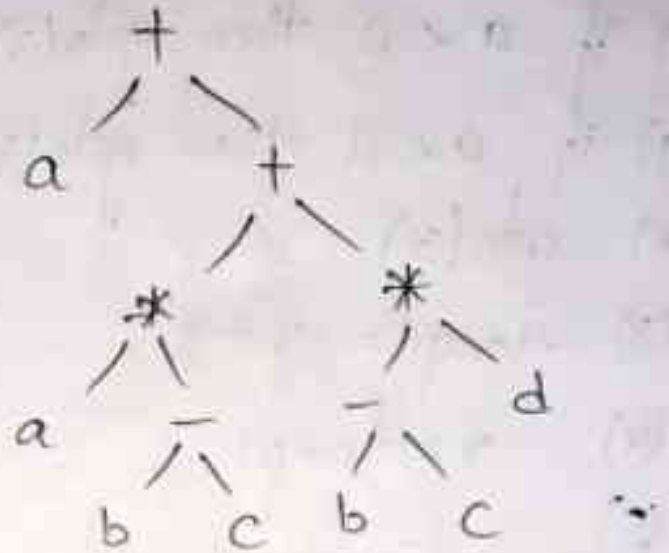


DAG:-

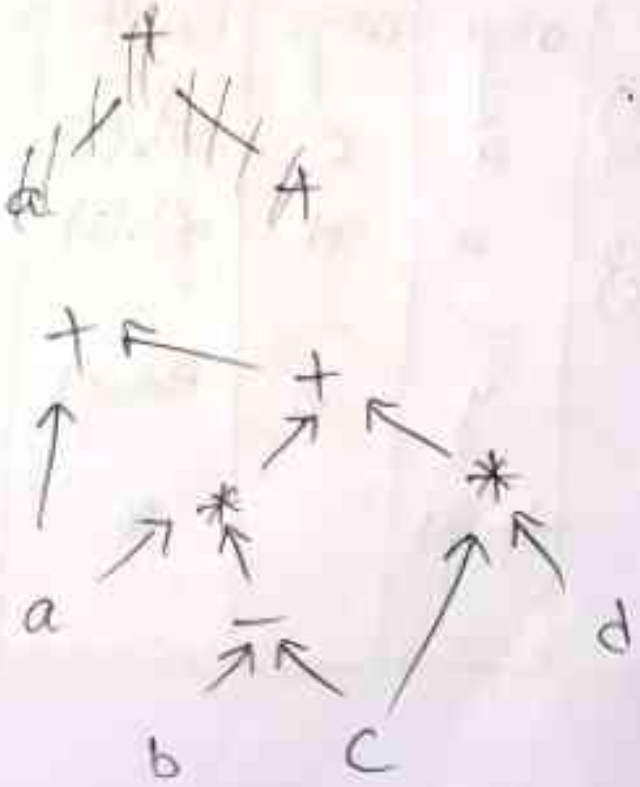


#) DAG for a + a * (b - c) + (b - c) * d

-> syntax tree :-



DAG :-



Q) If $a < c \,||\, a < d$ then $x = y + 3$.
Give quadruple, triple, indirect triple

→ (0) if $a < c$ then goto (3)
(1) if $a < d$ then goto (3)
(2) goto (5)
(3) temp1 = $y + 3$
(4) x = temp1
(5) —

Quadruple:-

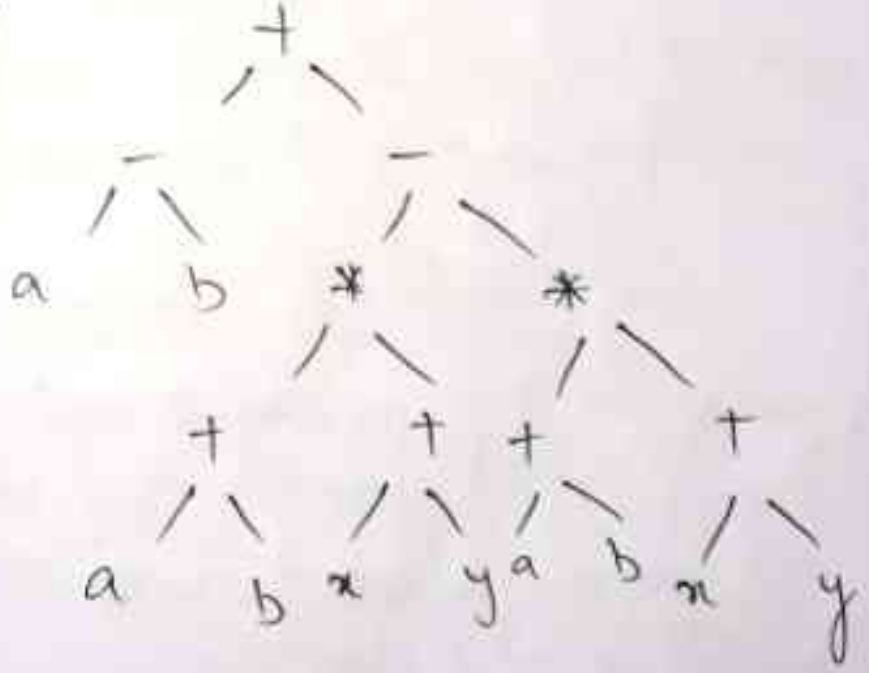
	op1	arg1	arg2	Result
(0)	lessthan (<)	a	c	goto (3)
(1)	lessthan (<)	a	d	goto (3)
(2)	-	-	-	-
(3)	+	y	3	temp1
(4)	=	temp1		x
(5)	-	-	-	-

Teaple :-

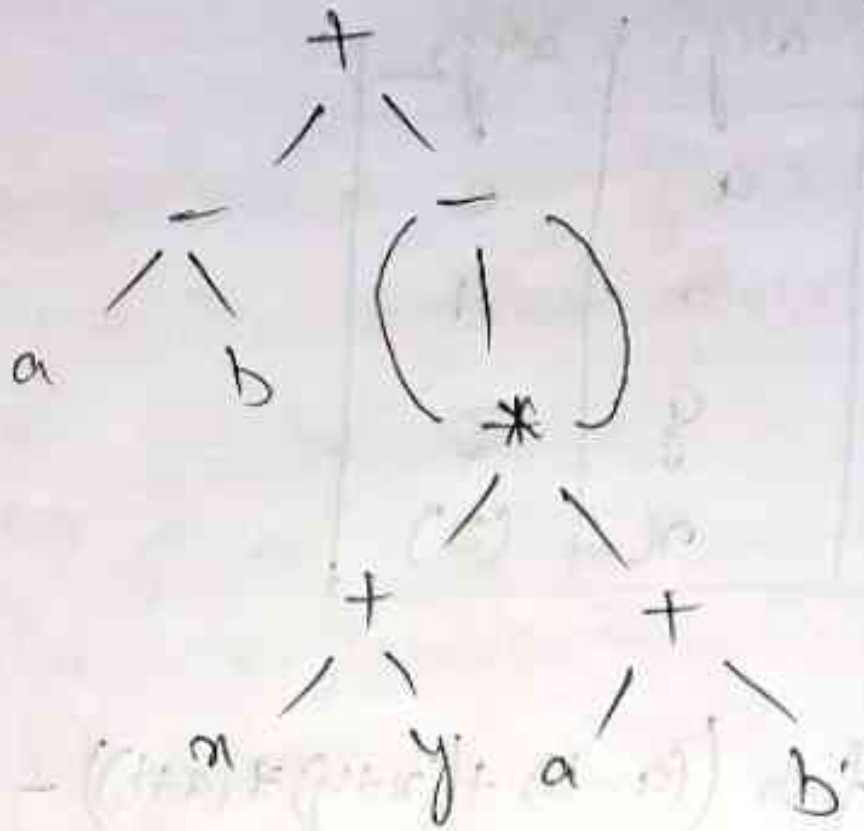
	opor	aseg1	aseg2
10)	<	a	c
(1)	<	a	d
(2)	+	y	3
(3)	=	x	(2)

a) DAG for $((a-b)+((x+y)*(a+b))-((a+b)*(x+y)))$

-> Syntax tree



DAG:-



* Type Expression :-

-> The idea is to associate each language construct with an expression describing its type and so called type expressions.

-> Type expressions are defined inductively from basic types and constants using type constructors.

> A basic type is a type expression
Ex:- int, float, char

> Typename is a type expression
int rollno.

> A type constructor applied to type expression is a type expression.

-> Type expression for Array :-
The type expression for an array is given as Array [I, T]

I -> Index of array
T -> data type of array

Ex:- int a[20];

Type expression -> Array [0-19, int]

→ Type Expression for Cartesian product :-

Type Expression for Cartesian product is given by $T_1 \times T_2$ where T_1 & T_2 are data type

→ Struct :- (Record).

The struct given by keyword struct is also a type Expression.

The type expression is given as product of types of field or member of structure.

```
Struct student
{  int arr[20];
   float marks;
}
```

The type Expression is given as

struct(arr * array[0-19, int] * marks * float)

array(1-5, array(1-5, int))

→ Pointer:-

The type expression for pointer is given as pointer(T) where T is the data type

→ Function:-

The type expression for function is given as Domain → Range

```
int sum(int, int) { }
```

Range ↓ Domain

Type Expression: int × int → int

a) Write type expression for an array of pointers to real where array index ranges from 10 to 100

→ Type Expression → Array[0-99, Pointer(real)]

a) Write type Expression of function whose domain are function from integer to pointer of integer & whose ranges are records consisting of integers & characters

→ Typ Exp: int → pointer(int) → int × char

* Intermediate code for procedure:-

Q) Suppose that a is an array of integers and that f is a function from integer to integer. Then the assignment

$$n = f(a[i])$$

Three-address code:-

1) $t_1 = i * 4$
2) $t_2 = a[t_1]$
3) Param t_2
4) $t_3 = \text{call } f, 1$
5) $n = t_3$

→ line ① & ② compute the value of expression $a[i]$ in t_2.

→ line ③ makes t_2 an actual parameter for the call on line ④ of f with one parameter.

→ line ⑤ assign the value returned by function call.

Unit 3:

Symbol table
https://www.geeksforgeeks.org/symbol-table-compiler/#:~:text=Symbol%20Table%20is%20an%20important,%2C%20classes%2C%20objects%2C%20etc.

Unit 4:

1. Storage Organization
   https://www.javatpoint.com/storage-organization
2. Activate Record
   https://www.javatpoint.com/activation-record
3. Storage Allocation
   https://www.javatpoint.com/storage-allocation
4. Access to non local data on stack
   https://www.brainkart.com/article/Access-to-Nonlocal-Data-on-the-Stack_8172/
5. Parameter passing
   https://www.geeksforgeeks.org/runtime-environments-in-compiler-design/