Compiler Design

1) describe in detail the syntax directed translation of case statements

A) In SDT, along with the grammar we associate some informal notations and these notations are called as semantic rules.

So, we can say that,

Grammar + semantic rule = SDT

→ Along with the grammar, we are giving semantic actions also such that along with the parsing task, some other tasks can also be done in parallel like code generation, Intermediate code Generation, Expression evaluation, etc.

⇒ Steps to be followed are:-

(1) Generate parse tree from the grammar

(2) Parse the tree: from left to right and top to bottom

(3) Whenever there is a reduction, go to the production and carry out the corresponding action

Eg:-

$E \rightarrow E + T$ / $\{E.val = E.val + T.val\}$

T $\{E.val = T.val\}$

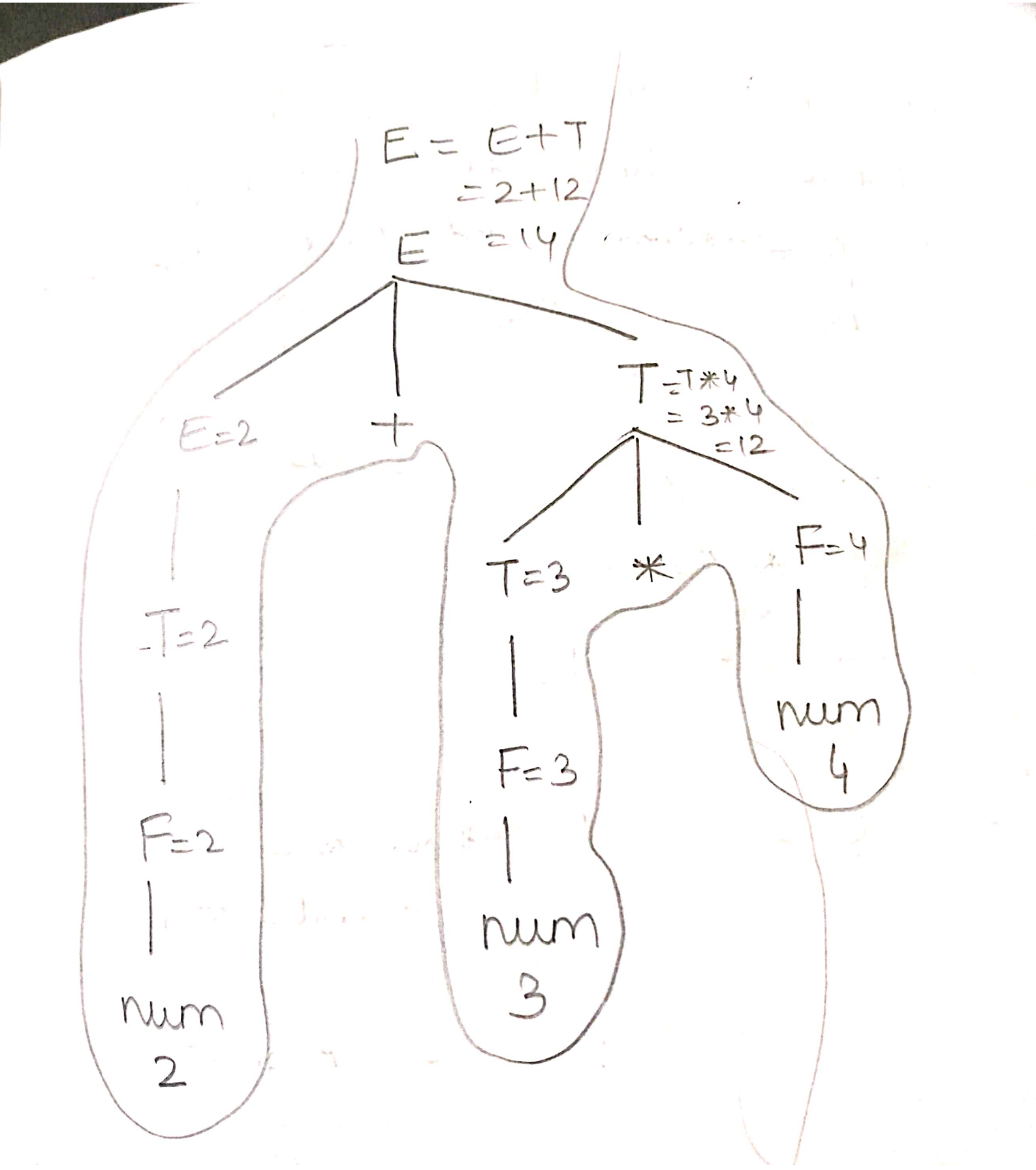
$T \rightarrow T * F$ / $\{T.val = T.val * F.val\}$

F $\{T.val = F.val\}$

$F \rightarrow num$ $\{F.val = num.val\}$

let exp

$2 + 3 * 4$


E = E+T
= 2+12
= 14
E
E=2
+
T = T*4
= 3*4
= 12
T=2
F=2
num
2
T=3
*
F=4
F=3
num
3
num
4

S

→ An SDT that uses only synthesized attribute is called S-attributed SDT

$A \rightarrow BCD$

$A = B.S$ }
$A = C.S$ } ✓
$A = D.S$ }

→ Semantic actions are always placed at the right most end of the production

$A \rightarrow BC \{ \}$

$B \rightarrow C \{ \}$

→ It supports LR parsing (LALR)

L

→ An SDT that uses both synthesized & inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called L-attributed SDT

$A = BCD$

$C = A.S$ ✓
$C = B.S$ ✓
$C = D.S$ ✗

→ Semantic actions can be placed anywhere on the RHS

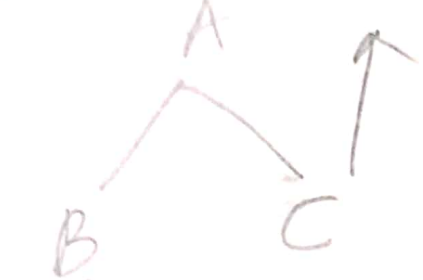
$A \rightarrow BCD \{ \}$

$B \rightarrow C \{ \} D$

$C = \{ \} D$

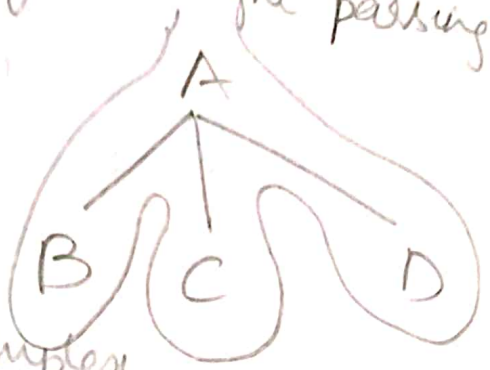
→ predictive parsing

→ attributes are evaluated using bottom up approach



→ Simple

→ attributes are evaluated using depth first, left to right passing.



→ Complex

Q3) Write an SDT to convert infix to postfix expression.

⇒ infix = An expression in which the operator is written in between the operands.

postfix = An exp in which the operator is written after the operand

For string 2 + 3 * 5.

SDT

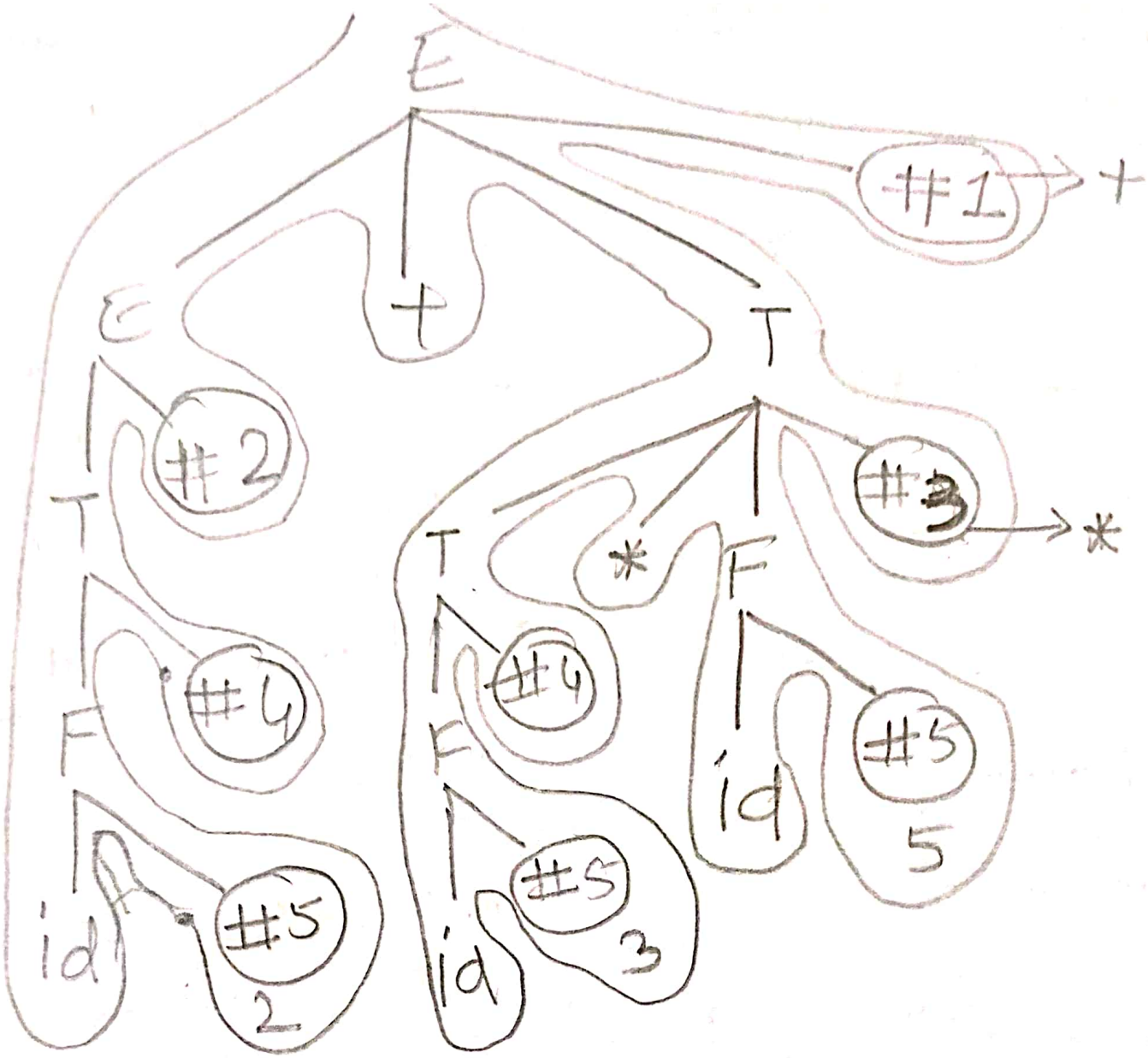
$E \rightarrow E + T$ { Printf("+") } #1

$E \rightarrow T$ #2

$T \rightarrow T * F$ { Printf("*") } #3

$T \rightarrow F \rightarrow$ #4

$F \rightarrow$ id { printf("id.lexval") } #5



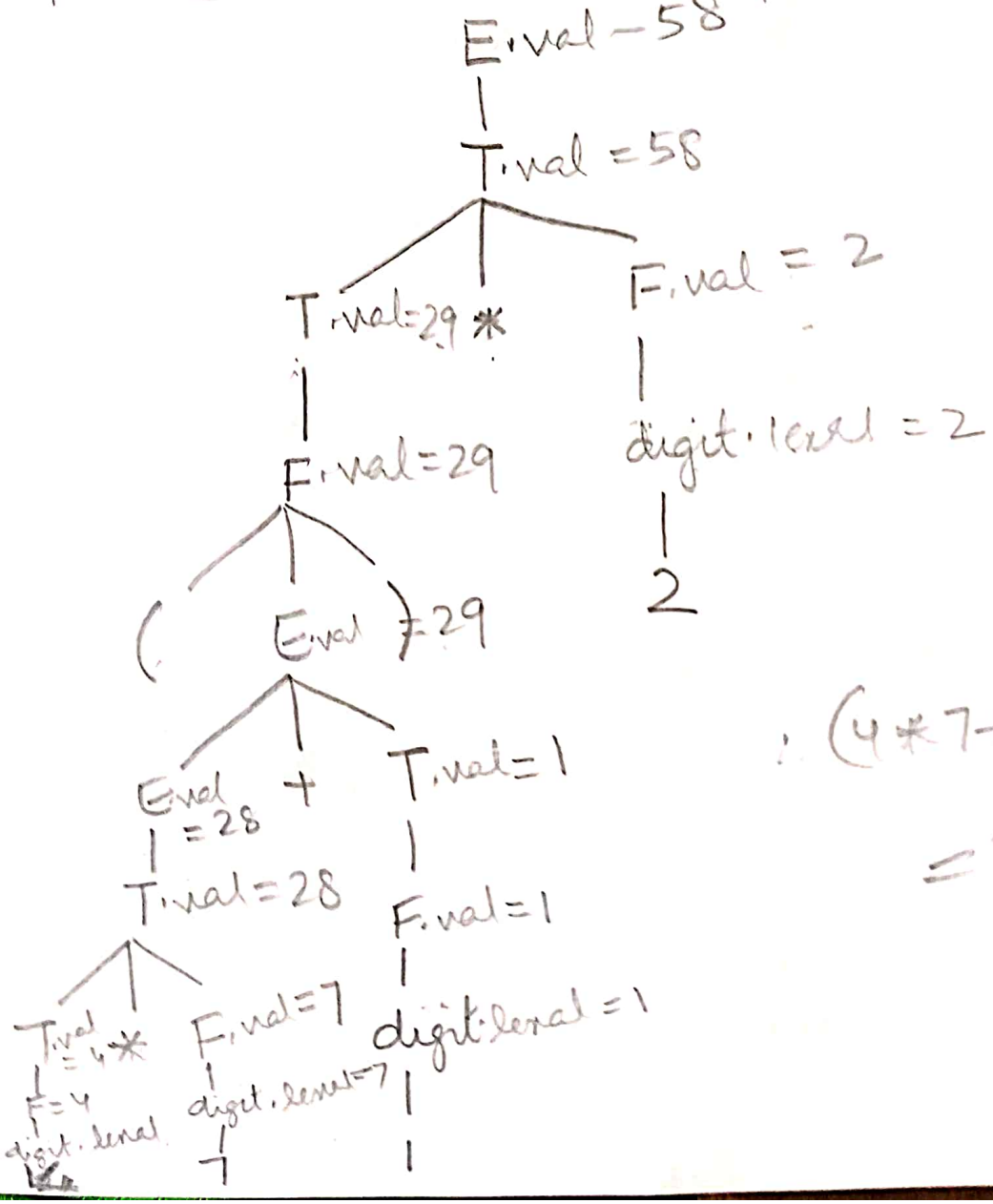
O/P:- 235*+

5) SDD of a simple desk calculator & Construct an annotated parse tree for input exp (4*7+1)* 2

Sol

$E \rightarrow E+T$	$E.val \rightarrow E_1.val + T.val$
$E \rightarrow T$	$E.val \rightarrow T.val$
$T \rightarrow T * F$	$T.val \rightarrow T.val * F.val$
$T \rightarrow F$	$T.val \rightarrow F.val$
$F \rightarrow (E)$	$F.val \rightarrow E.val$
$F \rightarrow digit$	$F.val \rightarrow digit.lexval$

Exp → (4 * 7+1) * 2



∴ (4*7+1)* 2

= 58

Compiler Design

Unit - 1

⇒ **Translator**:- Translator is a program that converts one form of language into another form without losing the functional or logical structure of the original code.
eg.
→ It converts high-level lang into machine lang or low-level language.
→ It will detect and report the errors during translation.
→ There are 3 diff types of translators:-

(1) **Assembler**:- An assembler is a type of computer program that converts program written in assembly language into machine language that can be executed by the computer.
→ It specifies the symbolic form of machine language i.e, Mnemonics and convert them into machine dependent code.

(2) **Compiler**:- Compiler is a type of computer program which takes the ~~written~~

a program written in high-level language as input and translates it into low level language as a whole.

→ It traces the errors in the source program and generates the error report.

(3) Interpreter:- An interpreter converts high level language into low-level language, just ~~as~~ like a compiler. But they are different in the way they take the input.

→ The compiler ~~scans~~ ~~the~~ translates the program as a whole while interpreter converts one statement at a time.

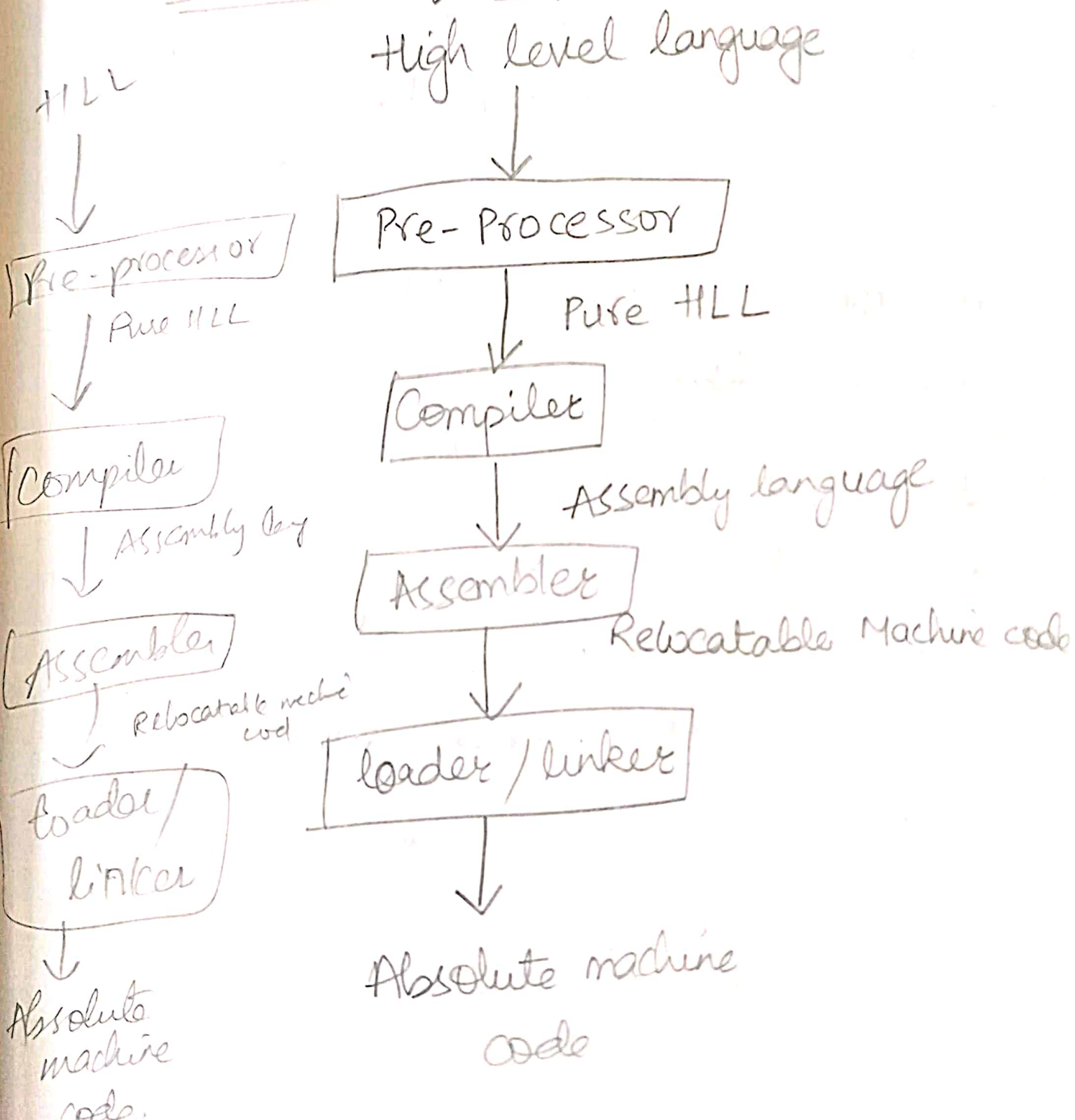
→ Errors are detected line by line.

⟹ Language Processing System:-

Any computer system is made of hardware & software. The hardware understands a language which humans cannot understand. So, we write programs in high-level lang which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get

desired code that can be used by machine. This is known as language processing system

Overview of language Processing System



(i) Pre-processor :- The pre-processor removes all the #include directives by including the file called file inclusion and all the # define directives using macro expansion.

→ It performs file inclusion, macro processing and augmentation, etc.

(ii) Compiler (from back)

(iii) Assembler (from back) converts assembly lang to relocatable machine code.

(iv) loader/linker :- It convets relocatable code into absolute code and tried to run the program resulting in running program.

→ linker :- linker links different object file into a single file/executable file.

→ loader :- loader loads that executable file in the memory and executes it

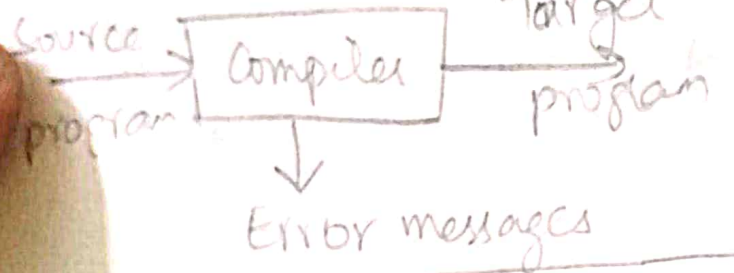
Compiler	Interpreter
1) It scans the entire program first and translates it into machine code.	1) It scans the program line by line and translates into machine code.
2) Compiler displays all the errors and warnings at a time	2) Intrepreter displays one error at a time
3) Error occures after scanning the whole program.	3) Error occurs after scanning each line.
4) Execution time is less.	4) Execution time is more.
5) It is more efficient	5) It is less efficient
6) Compilers are larger in size	6) Interpreters are ofte smaller than compiler.
7) It is not flexible	7) It is flexible
8) Both syntactic and semantic errors can be checked at the same time.	8) Only syntactic error can be checked at a time.
9) The translation carried out by ~~the~~ compiler is termed as compilation.	9) The translation carried out by interpreter is called as interpretation.

10) Execution speed is more in compiler.

11) It is also called as software translation

12) It is used by languages such as C, C++, etc

13) Compiler diagram is

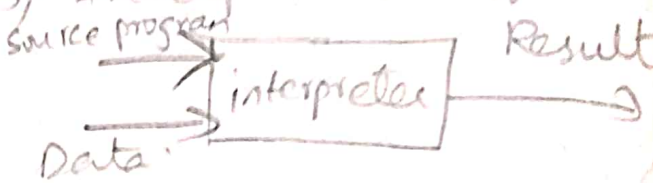


10) Execution speed is less in interpreter.

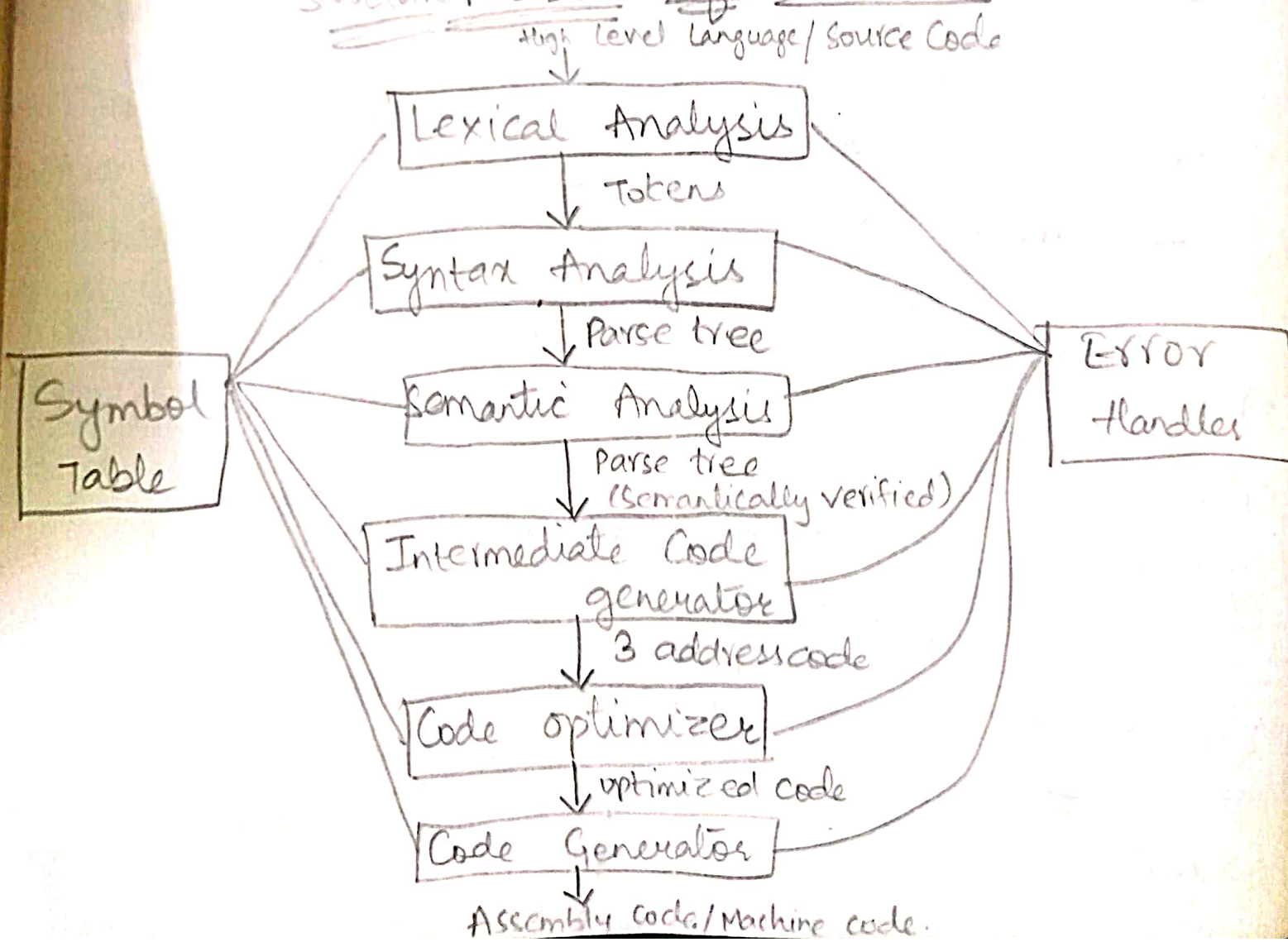
11) It is also ~~called~~ as software translation

12) It is used by lang such as Java

13) Interpreter diagram



structure Phases of Compiler



In General, compiler consists of six phases:-

(1) Lexical Analysis (2) Syntax Analysis (3) Semantic Analysis (4) Intermediate code generator (5) Code optimizer (6) Code generator.

→ The compilation process contains the sequence of various phases. Each phase takes source program in one representation & produces output in other representation.

→ Each phase takes input from its previous stage. → All phases are connected with the Symbol table & Error handler.

(1) Lexical Analysis:-

→ It is also called as Scanner.

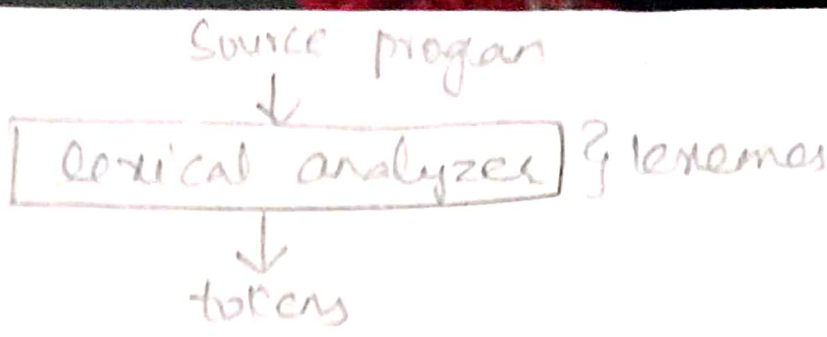
→ LA reads the same program char by char and converts the corresponding char into meaningful sequence called "lexeme"

lexeme → A sequence of char.

→ For each lexeme the LA produces "TOKEN" as o/p

→ Token may be a keyword, identifier, operator, constant, separators

→ It removes white spaces, comments, tabs.



2) Syntax Analysis :-

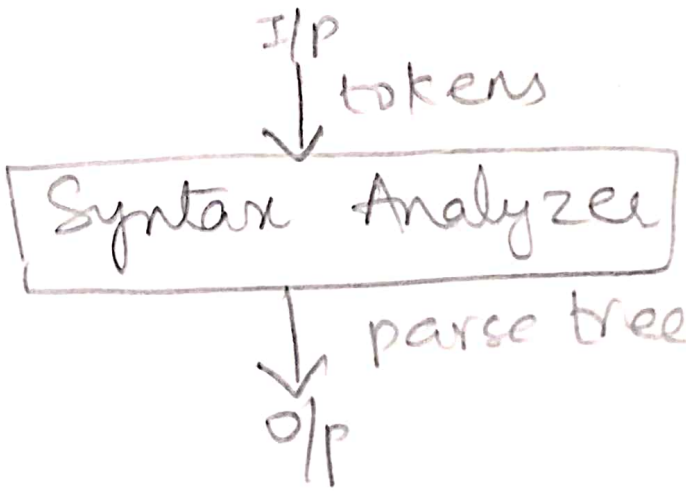
→ It is also called as parser

→ It is the second phase of compilation process

→ It will check whether the corresponding Syntax of the source program is correct or not.

→ If the syntax is not correct, Error handler reports error to the user.

→ It takes the tokens as input and generates tree like representation called as parse tree as output



3) Semantic Analysis:-

→ It is the third phase of compilation process.

→ The parse tree is given as input to the semantic analysis

→ It will check whether the meaning of the parse tree is correct or not (it semantically verifies the parse tree)

→ For verifying the parse tree, it uses a s/w called data checker which checks whether the data matches with the operands or not

→ If there is a need of type conversion it will perform type conversion.

→ output of this phase is annotated syntax tree.

4) Intermediate code generation:-

→ It is the fourth phase.

→ Semantically verified parse tree is given as input to the intermediate code generator

→ In this phase the compiler generates the source code into the intermediate code.

→ Intermediate code is generated b/w high level lang & machine lang.

→ Machine level instructions may be in different format but most common 3-address code format

~~→ 3-address code~~

eg. $x = a + b * 10$

$t_1 = b * 10$

$t_2 = a + t_1$

$x = t_2$ //

5) Code optimization-

→ It is an optional phase

→ Intermediate code is given as i/p to the code optimizer

→ It is used to improve the interm code so that o/p of the program runs & faster & take less sp

→ It removes unnecessary lines of th code & arranges the sequence of in order to speed up the program execution

→ It gives optimized intermediate code as o/

6) Target Code generator:-

→ It is the final phase of the compilation process.

→ Optimized intermediate code is given as i/p to target code generator.

→ It takes optimized intermediate code & maps it to the target machine language.

→ It converts optimized intermediate code to the target code.

→ In order to produce target code it uses some assembly languages.

⇒ Symbol table:- It is a portion of compiler which keeps track of the names used by the program & records info such as data types, precision & so on.

→ This datastruct allows us to find the record for each identifier quickly & to store (or) retrieve data quickly.

⇒ Error handler:- It interacts with all the phases of the compiler.

→ It reports the errors present in any of the ~~those~~ compiler phase to the user.

⇒ Compiler Passes:-

→ Passes means grouping of compiler phases.

→ There are two types of passes:-

1) Single pass compiler:-

→ All phases of compilation process are grouped together.

→ No intermediate representations are saved in memory.

→ These are fast compilers.

→ They are non-efficient compilers.

→ Code is not optimized in single pass compiler.

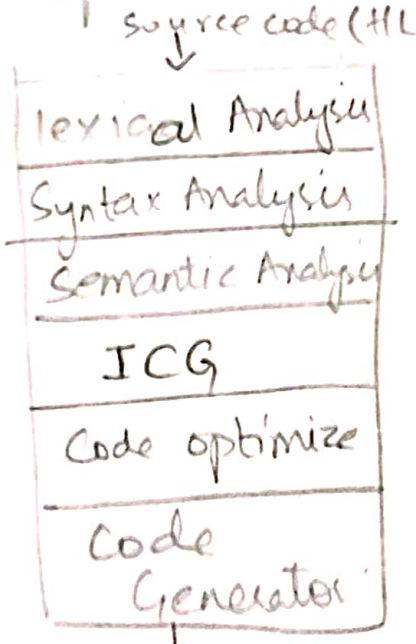


Fig:- Single pass Compiler

2) Multipass Compiler:- It is also called as two pass compiler.

→ The compilation process is divided into two pads.

(1) Analysis part or Front end of compiler:-

→ It takes source code as i/p, breaks it into meaningful pieces called tokens, constructs parse tree with these tokens and checks syntactical & semantical errors on that tree.

→ It consist of 4 phases:-
(i) lexical Analysis (ii) Syntax Analysis
(iii) Semantic Analysis (iv) Intermediate code generation

→ It produce intermediate code as o/p

→ It contains phases that are dependent on source ~~program~~ lang & independent of target program.

→ It collects info about source prog & stored it in symbol table.

(ii) Synthesis part or Back end:

→ It takes intermediate code as i/p and converts them into low level lang/ assembly language.

→ It consist of 2 phase:
i) Code Optimizer ii) Code Generator

→ It contains phases that are dependent on Target lang & independent of source program.

→ It constructs the target code from intermediate representation stored in Symbol table.

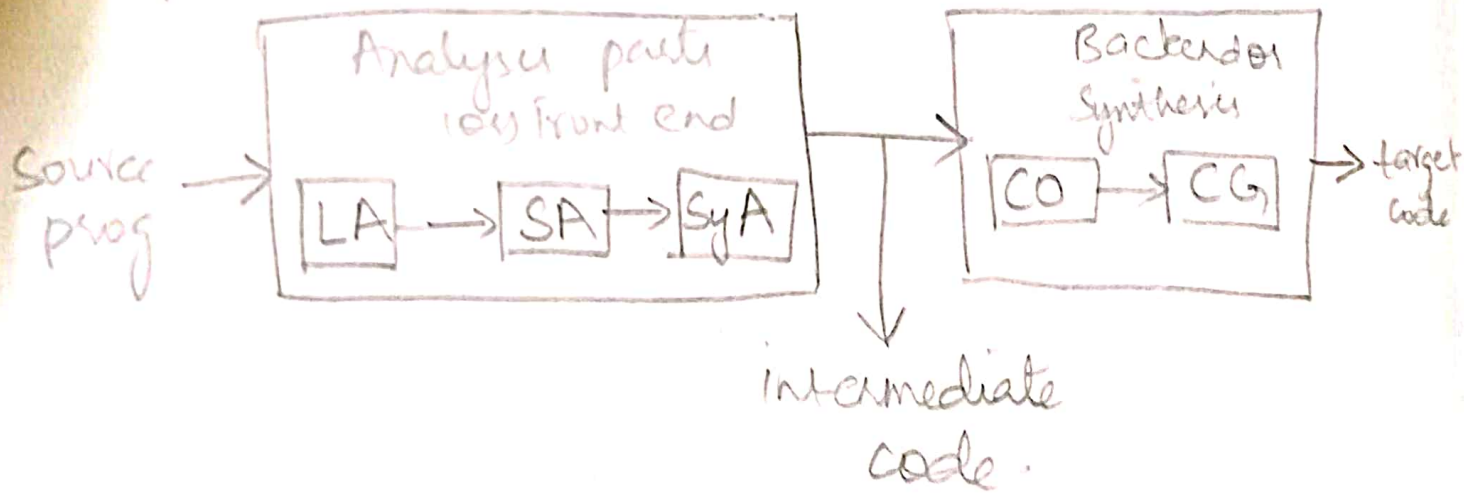


Fig- two pass compiler or multi

⇒ The Science of building Compiler :-

→ Building a compiler is a challenging task.

→ The main job of the compiler is to accept the source program of any size & convert it into suitable target program.

→ Program transformation performed by the compiler while translating source program must preserve the meaning of program being compiled.

→ The compiler writer must not only have the idea of the complete idea of the compiler they create, but all the programs that their compiler compiles.

→ The compiler study is focused mainly on study of how to design the correct mathematical model & choose correct algorithm keeping in mind to balance the need of for generality and efficiency.

→ Finite state machines & regular expressions are useful for describing the lexical units of programs (tokens) & for describing the algorithms used by computer to identify these tokens.

→ In CD, the term "code optimization"

indicates the attempts made by the compiler to produce code which is more efficient than the previous one

→ This code should be faster than any other code that performs the same task.

→ Objectives to be fulfilled by compiler optimization include

- meaning of the compiled prog must be preserved
- Optimization should improve program's performa
- Time required for compilation should be reasonable
- The engineering effort required must be manageable

⇒ Compiler Construction tools:- (Srilakshmi - Unit 1 pg

→ Application of Compiler Technology:-

Applications of compiler technology includ

(1) Implementation of high level programming language:-

Compilers are used to ~~to~~ convert high level lang into machine level lang

The major diff b/w both is that the high level lang is easy to write but less efficient. whereas the machine level lang is efficient. Moreover, low level lang is hard to write but have high level of errors & requires a lot of maintainence. So a body of compiler optimization known as the "Data Flow optimization" has been developed. They analyze the flow of data & remove redundancies in prog. They generate code which are similar to the code of a skilled programmer writes a code in LLL. The key to ideas behind object orientation are abstraction & inheritance They There are many programming lang which are capable of providing high level of abstraction these include C, C++, Java. Optimizing compiler in such lang means that, the optimized compiler must be capable of performing its operations efficiently across the procedural boundaries of the source program. New features which makes programming easier also increase the overhead. To reduce overhead many algorithm have been developed.

(2) Optimizing the Computer Architecture:-

Evolution of computer technology need evolution in compiler technology. New system takes advantage of two methods:- parallelism and memory hierarchies.

→ Parallelism may be found at two levels: at the instruction level, where many operations are performed at the same time, and at the processor level, where distinct threads of the same program are executed on different processors.

→ Memory hierarchies address the fundamental problem of being able to produce either extremely fast storage or extremely huge storage, but not both.

(3) Design of new computer architecture:-

In early days of computer architecture design, compilers were created after the machines were built. That isn't the case ~~any~~ now. Because, performance of the system isn't only determined by

its raw speed but also by how well compilers can use their capabilities. In modern days, compilers are design in the processor design stage. The following are few modern comp architectures

- RISC (Reduced Instruction Set Computer) These were designed to recognize a relatively small no. of compiler instruction so that it can perform its operation at high speed.
- CISC (Complex Instruction Set Comp):- These were used before RISC. These were used bcoz there was assembly lang compilers were not very efficient & designed over past decades includ.
- Specialized Architectures:- diff architectures
  - Data flow machines
  - Vector machines
  - Very long instr word machine (VLIW)
  - Systolic arr

→ Embedded machines are now common,

4) Program Translation :-

- Binary translation: Compiler tech can be used to translate the prog written in binary code for one machine into a form that can be used by another machine, It has been used by many companies to increase availability of software for their machines. It also provides backward compatibility mode.

- Hardware Synthesis: It perform automatic translation of Register transfer lang (RTL) into gates. These gates are mapped onto transistors followed by the physical layout.
- Interface Database Query - It consist of predicates that are interpreted into commands. This interpretation is done for searching a record in database that satisfies the predicates.

Lexical Analysis:-

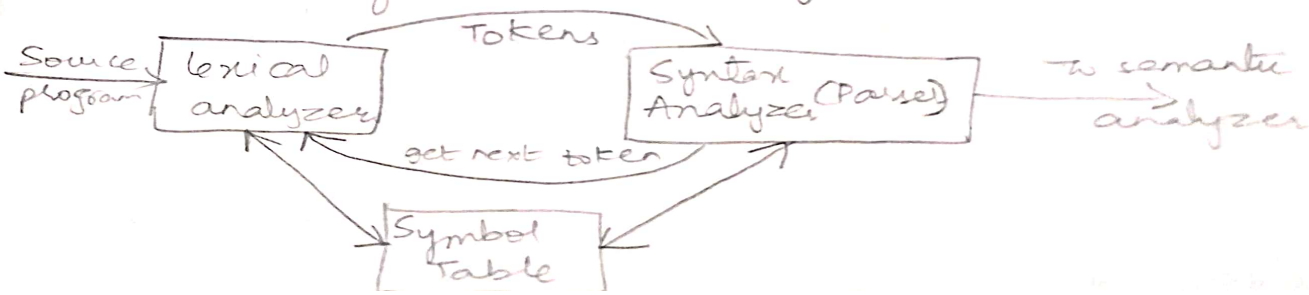
- Role of lexical Analyzer:-

→ Lexical analysis is the first phase of a compiler process.

→ In lexical analysis, lexical analyzer converts the source program into a stream of tokens.

→ LA is also called as scanning.

→ Interaction of lexical analyzer with parser is as follows



→ lexical analyzer performs the following task :-

- reads the source prog, scans the i/p characters, groups them into lexemes & produce the token as o/p
- It enters the identified tokens into the symbol tables.
- It removes white spaces & comments from the source program.
- Correlates error messages with the source program ie, it displays error message with its occurence by specifying the line no.

→ Tasks of lexical analyzer is divided into 2 parts :-

Scanning :- It performs reading of i/p characters removal of white spaces & comments.

Tokenization :- producing tokens as the o/p.

⇒ Issues in lexical analyzer (LA vs parser)

- Simplicity in design of compiler :- The removal of white spaces & comments enables the syntax analyzer for efficient syntactic constructs
- Compiler efficiency is improved :- Specialized buffering techniques for reading chars speeds up the compiler process
- Compiler probability is enhanced :- Specialized tools have been designed to help automate the const of lexical analyzers & parsers when they are separated

⇒ Token:- A token is a sequence of char that ~~can be treated as a~~ having a collective meaning

eg:- keywords, identifiers, operators, separators etc.

```
          operator
            ↑
int a = 5 ; → separator
↓   ↓   ↓
keyword  identifier  constant
```

It has 5 tokens

⇒ lexeme:- It is a sequence of char in the source prog that is mached by the pattern for a token

⇒ Pattern:- It is a rule describing the set of lexemes that can represent a particular token in the source program

Eg:- Ex:- if (a < b)

lexeme	token	Pattern
if	identifier	if
(	left paranthesis	(or)
a	identifier	letter or digit
<	relational op	< or <= or > or >= or == or <>
b	identifier	letter or digit
)	right paranthesis	(or)

⇒ Attribute :- When ~~a lexeme is~~
→ lexical analyzer provides additional info to distinguish b/w similar type of patterns that match a lexeme.
→ The lexical analyzer collects the attributes of tokens as the additional info.
→ A token has a single attribute i.e, a pointer to the the symbol table entry in which info about the token is kept.

$a = b + c$.

lexeme	token	Attribute values
a	identifier	pointer to symbol-table entry for a
=	operator	–
b	identifier	pointer to symbol-table entry for b.
+	add-op	–
c	identifier	pointer to symbol-table entry for c

⇒ Lexical Errors :-

A ~~the~~ char sequence which is not possible to be seen as any valid token is known as lexical error. ~~lexical~~ lexical analyzer detects ~~the~~ & reports the lexical errors to the error handler.

→ lexical phase error can be:-

- Spelling error
- exceeding length of identifier
- Appearance of illegal character.
- long identifier
- too long numerical literals

→ lexical errors are recovered by panic mode error recovery scheme.

Panic mode error Recovery scheme

This consist of 4 methods:-

- Deleting of extra char

  printfx → printf

  aprintf → printf

- Inserting a the correct char in place of error:-

  praintf → printf

- transposing two adjacent char

  ~~fo~~ fro → for

- Inserting the missing char

  whle → while

Input Buffering :-

→ The lexical analyzer scans the i/p from left to right one char at a time

→ lexical analyzer will take 25 to 30% of the compile time.

→ IB uses two pointers :-

- beginning ptr (bp) :- It points to the beginning of the current lexeme which is yet to be found.
- forward ptr (fp) :- It scans ahead until a match for a pattern is found.

→ It is of two types :-

(i) One buffer scheme :-

In this scheme only one buffer is used to store the input string.

Size of Buffer = N char

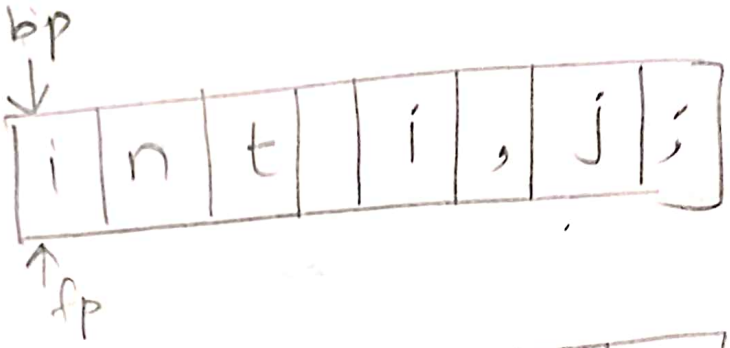
One system command read N char.

(ii) Two buffer scheme (Buffer pairs)

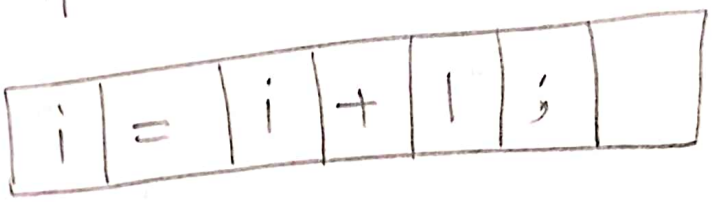
- To overcome the prob of one buffer scheme 2-buffer scheme is used.
- It means one buffer is reading & 2nd can be loading.
- both buffers are N char long.

- Size of Buffer = 2N chars
- when end of current buffer is reached the other buffer is filled.
- To identify end of buffer, eof char introduced at the end of both buffers. eof is called as sentinel.

eg:-



1st buffer



2nd buffer.

→ 1st buffer & 2nd buffer are scanned alternatively.

→ when the end of current buffer is reached the other buffer is filled.

- we have to determine whether the first buffer is completely filled or not, for that purpose "eof" is used.

- when the first buffer encounters the eof, it recognizes that the first buffer is completely filled & starts filling the second buffer

→ Sentinels:-

- A special char that suggests an end of the buffer is called sentinel.
- It is a special char that cannot be a part of the source program.
- It is represented using "eof"

⇒ Recognition of tokens:-

→ Recognition of tokens can be done by finite automata so that it can be recognized by using the transition diag or transition table.

→ variable/identifier/const/keyword - tokens can be verified/recognized with the help of transition diagram.

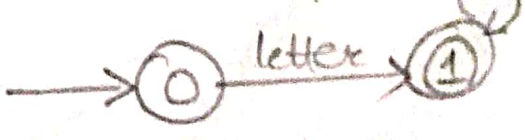
i) Recognition of tokens (identifiers):-

letters → a|b|c|....|z|A|B|....|Z|

digits → 0|1|2|....|9

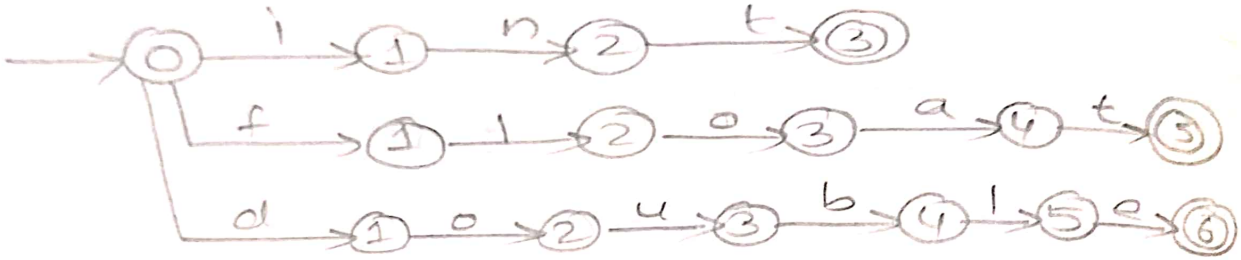
id → letter(letter/digit)*

1. Transition diag



(ii) Recognition of Keywords:-

We can use transition diag to recognize keywords like int, float, double, if, then, else, ...



(iii) Recognition of variables:-

We can use _ (underscore) at beginning a letter & later a digit or letter but cannot start with the digit.

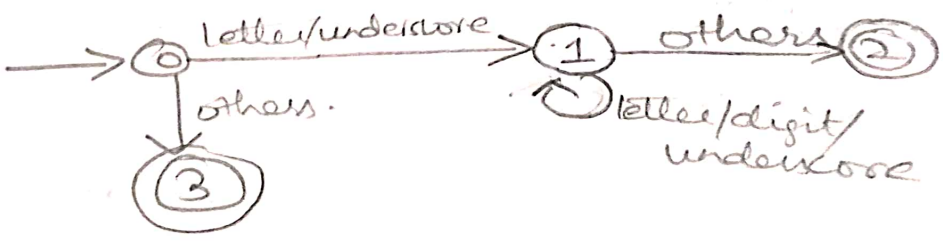
underscore → _

digit → 0|1|2|3|...|9

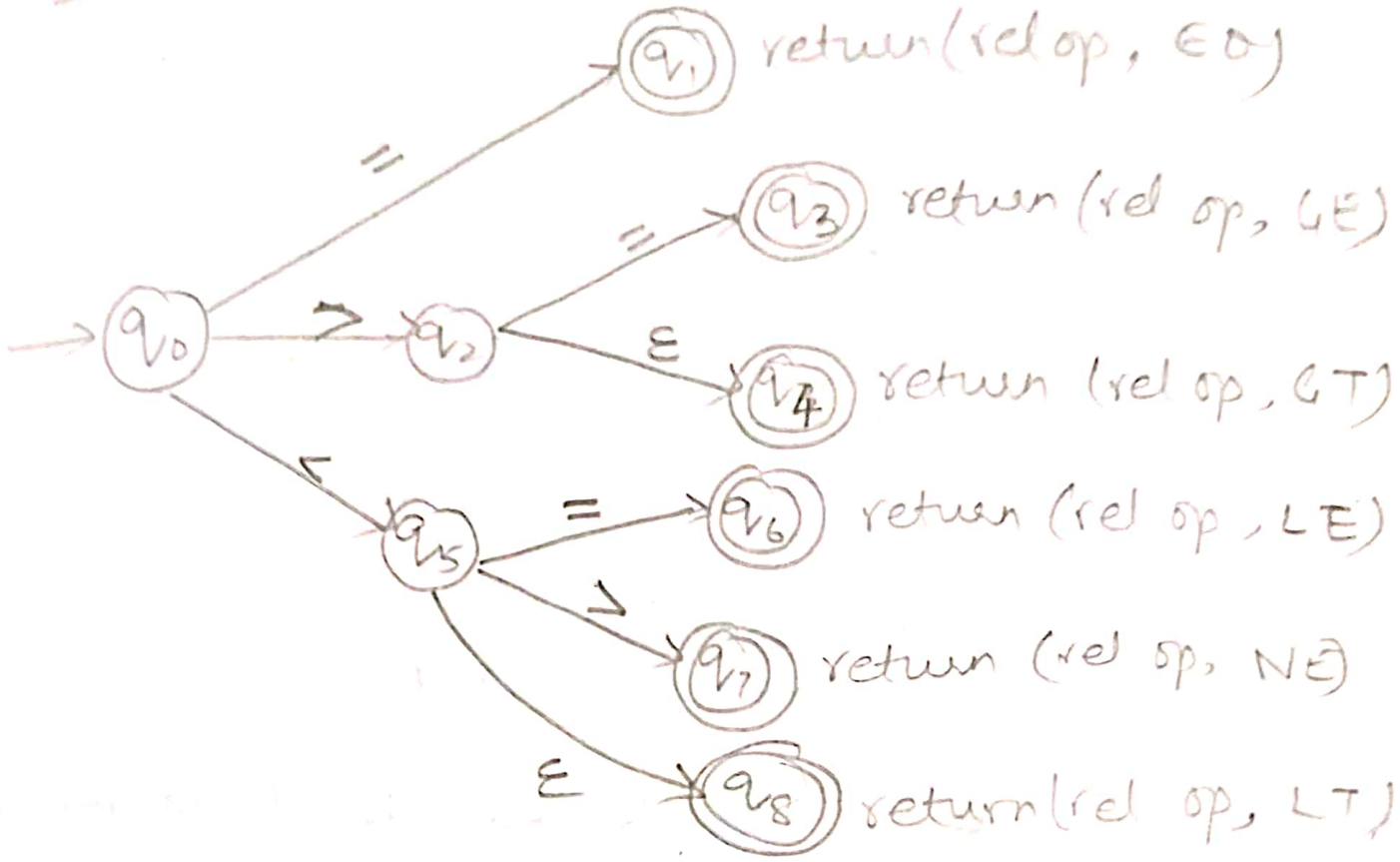
letter → a|b|...|z|A|B|...|Z

var → (letter|underscore)* (letter|digit|underscore)* others

Transition diag;



Recognition of relational operator



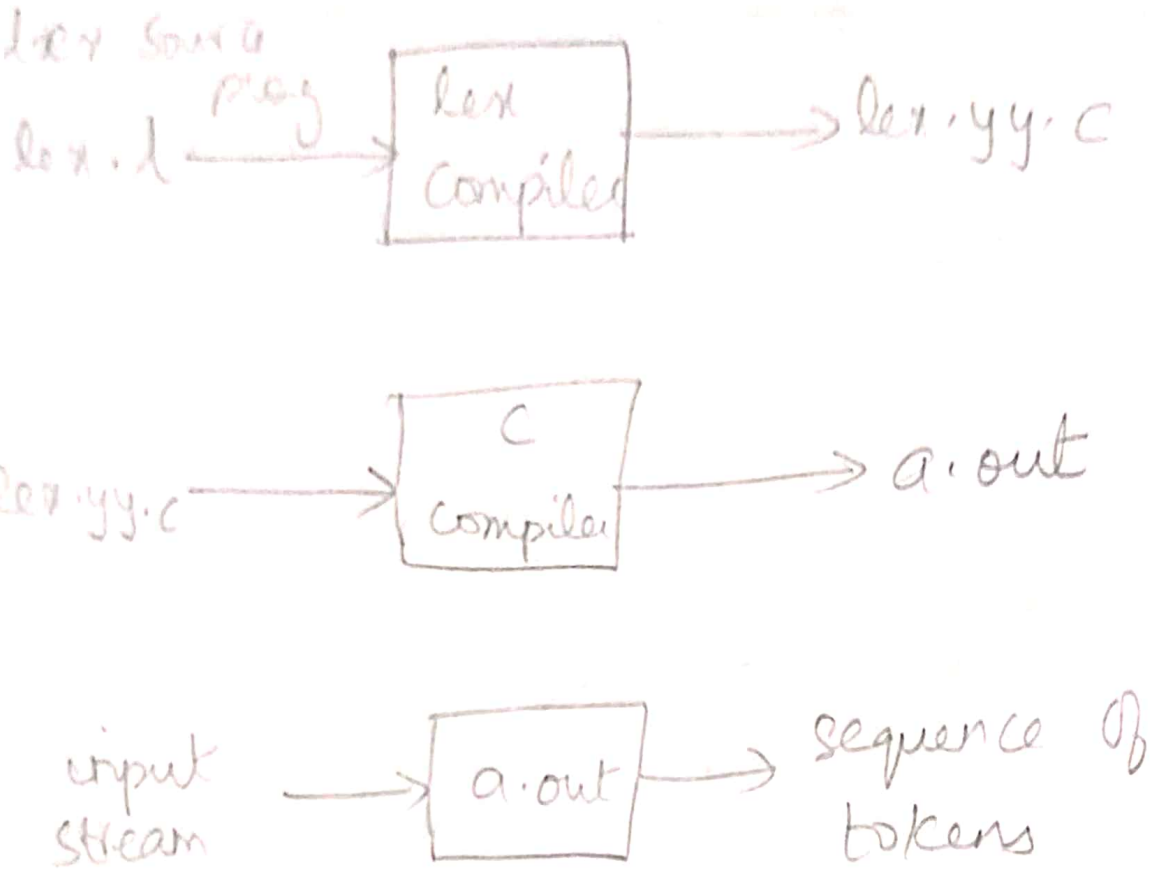
LEX TOOLS:-

→ It is a tool which generate lexical analyzer.

→ lexical analyzer is first phase of compiler which takes input as source code & generate output as tokens

→ The input notation for the lex tool is referred to as lex language and tool itself is lex compiler.

→ The lex compiler transforms the input patterns into a transition diagram & generate code, in a file called lex.yy.c



→ An input file, which we call lex.l, is written in the lex language and describes the lexical analyser to be generated.

→ The lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c

→ The latter file is compiled by C compiler into a file called a.out, as always.

→ The C-compiler output is a working lexical analyzer that can take a stream of input chars & produce a stream of tokens.

⇒ Structure of lex program

A lex prog has the following form

{ declarations }

%%

{ translation rules }

%%

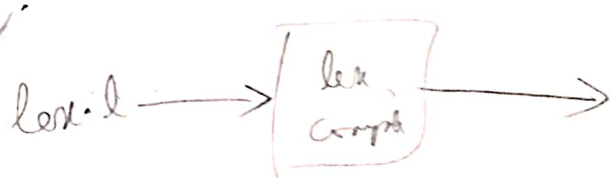
{ auxiliary functions }

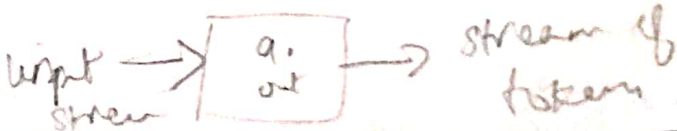
→ declarations section includes declarations of variables, constants

→ The translation rules have the form :-

Pattern { Action }

→ The third section holds whatever auxiliary functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyser.





lex program to recognize decimal no

```
%{
   #include <stdio.h>
    int i;
  %}
%% [0-9]+[.]
   {
    for (i=0; i<=yyleng; i++)
     {
      if (yytext[i] = '.')
       {printf ("%s is a decimal number", yytext);
       }
     }
  printf("%s is not a decimal number", yytext);
 }
%%
main()
{
 printf("\n enter any number \n");
  yylex();
}
int yywrap()
```

```
}
return 1;
}
```

```
# lex decimal.l
# cc lex.yy.c
# ./a.out
Enter any number 0-2(
0.25 is a decimal no
```

Lex program to convert abc to ABC

```
%{
#include <stdio.h>
#include< string.h>
 int i;
 %}
%% [a-z A-Z]*
 { for (i=0; i<=yyleng; i++)
  { if(yytext[i] == 'a') && (yytext[i+1] == 'b') &&
                  (yytext[i+2] == 'c'))
    { yytext[i] = 'A';
      yytext[i+1] = 'B';
      yytext[i+2] = 'C';
    }
  }
  printf ("%s", yytext);
 }
%%
main ()
 { printf("Enter input \n");
   yylex ();
 }
int yywrap()
 { return 1;
 }
```

output :-

```
# vi abctest.l
# lex abctest.l
# cc lex.yy.c
# ./a.out
Enter Input
abc hello world abc
ABC hello world ABC
```

⇒ Finite Automata :-

→ A FA is a mathematical model that takes string (w) as its input and check whether it belongs to the given language or not. ~~It is used mainly to design~~

→ When a regular exp is fed into FA it changes its state to each literal.

→ If input string is successfully processed the automata reaches the final state, it is accepted.

→ Mathematic model of FA consist of :-

$\{Q, \Sigma, q_0, \delta, F\}$

Q = set of finite states

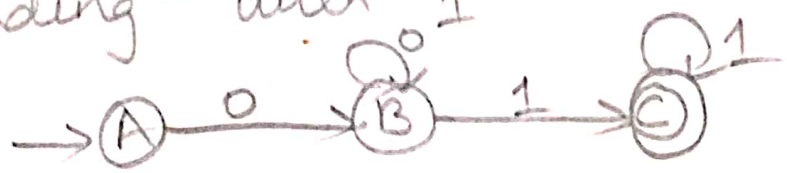
Σ = set of input symbols

q_0 = initial state

F = final state

δ = transition function $(Q \times \Sigma \to Q)$

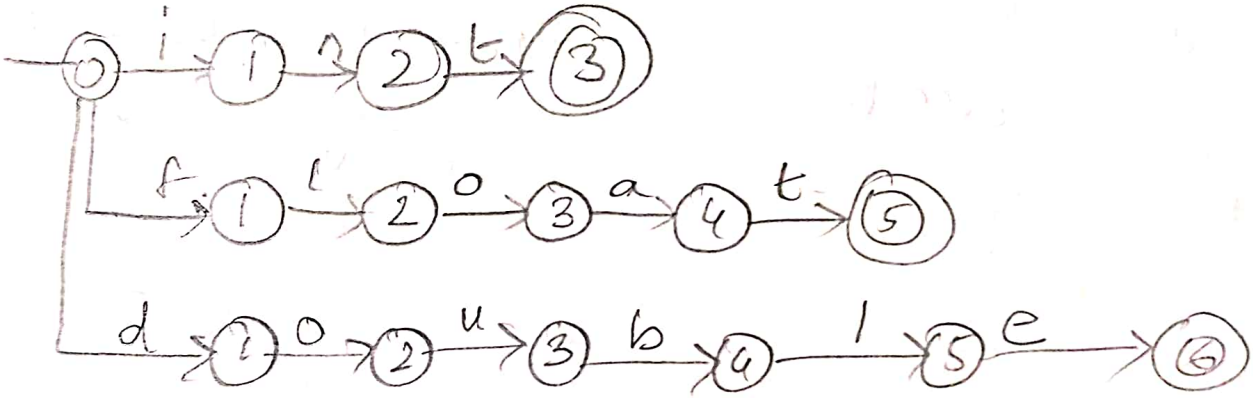
→ Transition diagram for accepting any of 01 and ending with 1



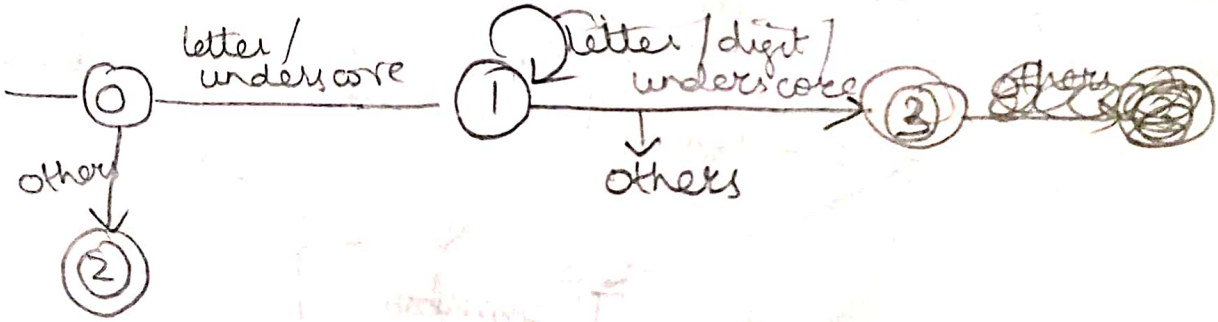
Cross Compiler — Google Drive
CD Unit1.pdf
(pg 23)

identifier

Keyword



Variable: (letter)* (letter / digit / underscore)* others



Unit - 2

Syntax Analysis

→ It is the second phase of compiler process

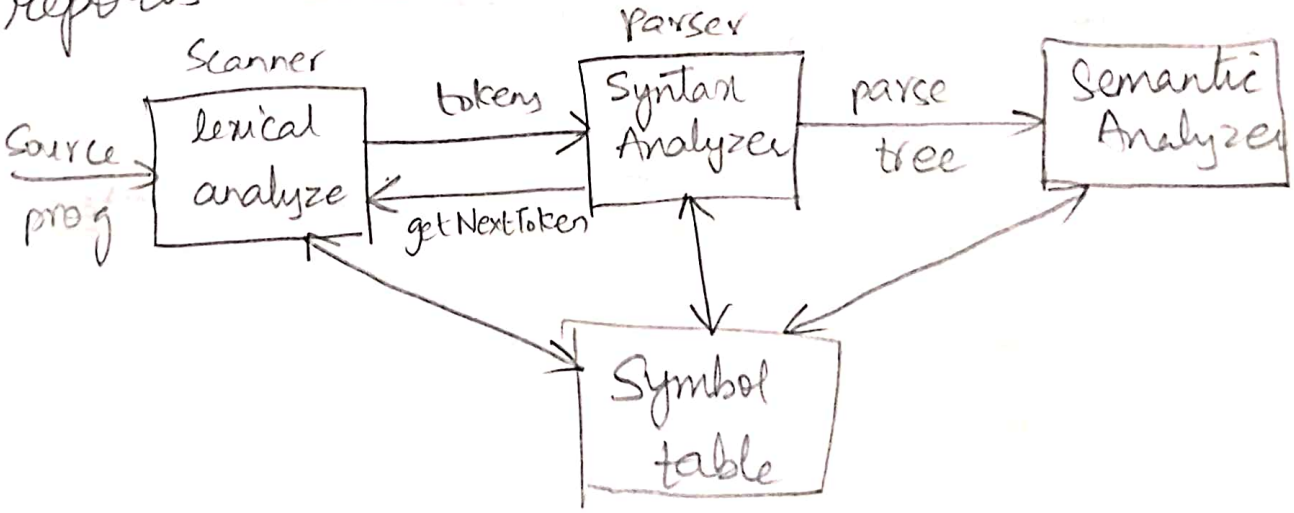
→ It is also called as parsing.

→ It ~~takes~~ receives token and sends request (get Next Token) to lexical Analyzer. The lexical analyzer responds by sending another token.

→ It takes takes tokens as input and generates tree like structure / representation called as parse tree as output.

→ It will check whether the corresponding syntax of the source program is correct or not.

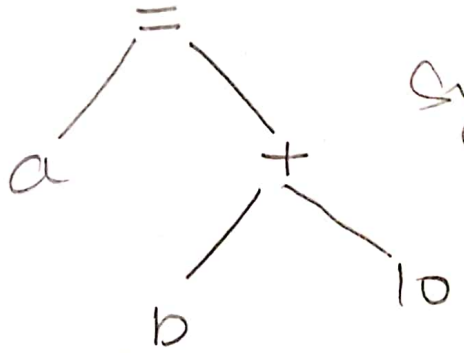
→ If the syntax is not correct, Error handler reports error to the user.



Example:-

$a = b + 10$

The above code (statement), first goes to the LA phase. It will divide it into tokens. The SA phase takes the tokens as I/P & generates parse tree.



Syntax tree or parse tree

⇒ Role of the Parser:-

→ The parser obtains a string of tokens from lexical analyzer & verifies that the string can be generated by the grammar for the source lang.

→ Parser reports any syntax errors in the program.

→ It also recovers from commonly occuring errors so that it can continue processing its input.

→ It takes tokens as input & generates parse tree as o/p.

→ There are three types of parsers for grammar:-
Universal, Top-down, bottom-up

→ As implied by the names, top-down method builds parse tree from top (root) to bottom (leaves) whereas bottom up method builds parse tree ~~leaves~~ ~~and~~ ~~vice~~ bottom (leaves) to top (root).

⇒ Content Free Grammar:-
A CFG can be defined by using 4 tuples-
$G = \{V, T, P, S\}$

where V → set of non-terminals (capital letters)
T → set of terminal (lower-case letters)
P → set of production rules.
S → start symbol.

→ Conventions:-

- Terminals:- They are symbols from which strings are formed
  - lower case letters ie; a, b, c
  - operators (+, -, *)
  - punctuation symbols (comma, parenthesis)
  - digits (0-9)
  - Bold face letters (id, if ..)

- Non-terminals - They are variables that denote a set of strings.
  - Upper case letters (A, B, C, ... Z)
- Start Symbol :- It is the head of the production stated first in the grammar.
- Production :- It is in the form LHS → RHS or head → body. where, head contains only one non terminal & body contains a collection of terminals & non-terminals.

Example- let lang $L = a^n b^n$ where $n \geq 1$, Let $G = \{V, T, P, S\}$ where $V = \{S\}$, $T = \{a, b\}$ and S is a start symbol then give product rules.

Sol Thelang $L = a^n b^n$, $n \geq 1$ $G = \{T, V, P,$

$S \rightarrow ab$ for $n = 1$

$S \rightarrow aabb$ for $n = 2$

$S \rightarrow aaabbb$ for $n = 3$

⋮

$S = a^n b^n$ for $n = n$.

⇒ Derivation :- It is a process of applying a sequence of production rules to derive a string.

→ process of derivation always starts from start symbol.

→ It is classified into two types:-

1. left most derivation
2. right most derivation

1) leftmost derivation :- In this derivation the leftmost non-terminal is replaced by its appropriate production rule.

2) right most :- In this derivation the right most non-terminal is replaced by its appropriate production rule.

Eg:- let G be a CFG for which the production rules are given below

S → aB/bA, A → a|as|bAA, B → b|bs|aBB. Derive the string "aaabbabbba" using left & right derivation

Sol) 1) leftmost derivation :-

S → aB

S → aaBB

S → aaaBBB

$S \rightarrow aaab\underline{S}BB$

$S \rightarrow aaabb\underline{A}BB$

~~$S \rightarrow aaabba\underline{S}BB$~~

~~$S \rightarrow aaabbab\underline{A}BB$~~

~~$S \rightarrow aaabbab$~~

$S \rightarrow aaabbaB\underline{B}$

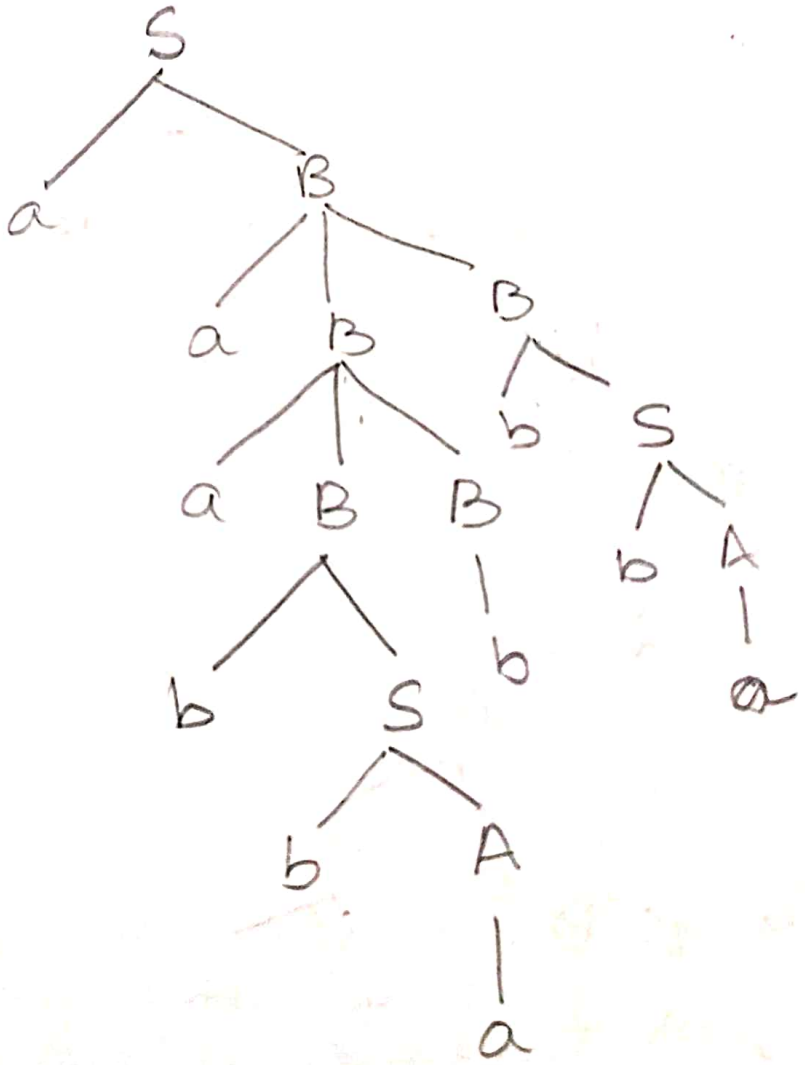
$S \rightarrow aaabbab\underline{B}$

$S \rightarrow aaabbabb\underline{S}$

$S \rightarrow aaabbabbb\underline{A}$

$S \rightarrow aaabbabbbA$

Derivation tree



→ Rightmost derivation:-

S → aB

S → aaBB

S → aaBbS

S → aaBbbA

S → aaBbba

S → aaaBBbbba

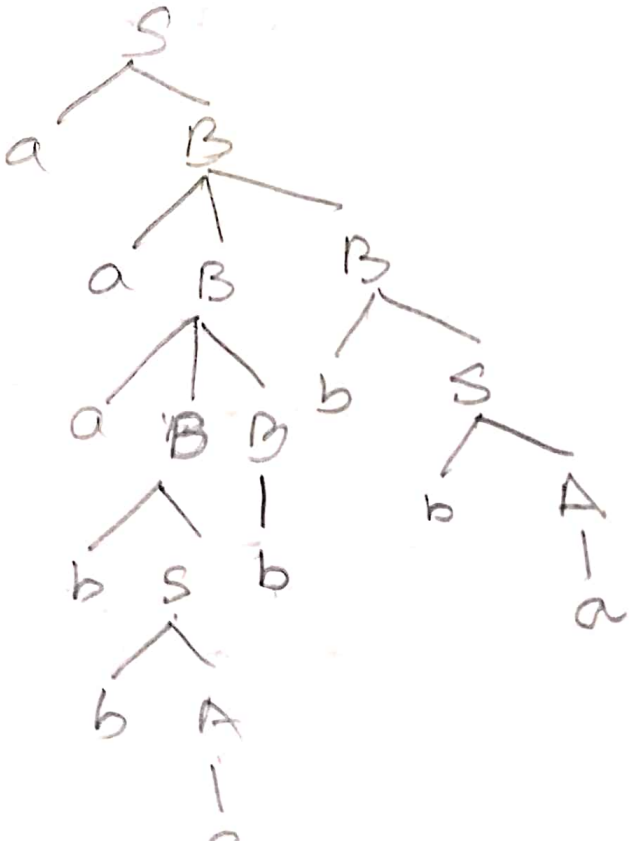
S → aaaBbbbba

S → aaabSbbbba

S → aaabbAbbbba

S → aaabbabbbba

Derivation tree



⇒ Derivation tree:- The graphical representation of derivation is called as derivation tree/ tree.

→ It is the graphical representation of symbol. The symbol can be terminal or non-terminal.

→ In parsing, the string is derived using the start symbol. The root of the parse tree is start symbol.

→ Parse tree follows three points:-
- All leaf nodes have to be terminal.
- All interior nodes have to be non-terminal.
- In order traversal gives original i/p string.

→ yield of the tree:- The yield of the tree is the collection of leaf nodes from left to right.

Example prob

1) Consider the grammar given below

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid a \mid b$. Obtain LMD and RMD for the string 'a+b*a+b'

Sol

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid a \mid b$.

LMD:-

$E \rightarrow \underline{E}*E$

$E \rightarrow \underline{E}+E*E$

$E \rightarrow a+\underline{E}*E$

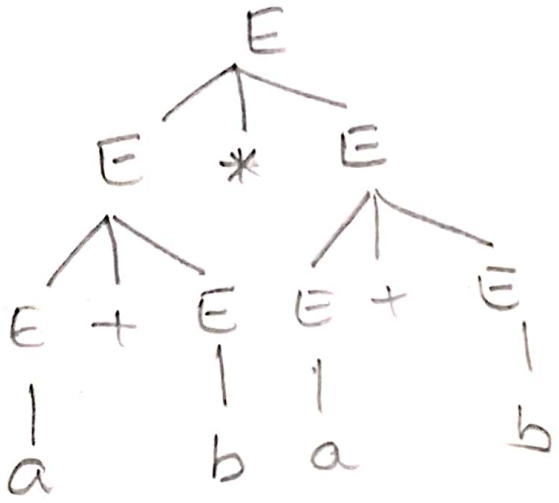
$E \rightarrow a+b*\underline{E}$

$E \rightarrow a+b*\underline{E}+E$

$E \rightarrow a+b*a+\underline{E}$

$E \rightarrow a+b*a+b$

DT



RMD:-

$E \rightarrow E*\underline{E}$

$E \rightarrow E*E+\underline{E}$

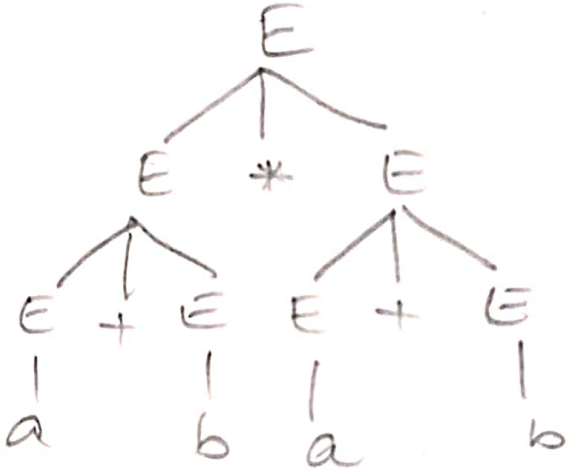
$E \rightarrow E*\underline{E}+b$

$E \rightarrow \underline{E}*a+b$

$E \rightarrow E+\underline{E}*a+b$

$E \rightarrow \underline{E}+b*a+b$

$E \rightarrow a+b*a+b$



2) Consider the following grammar

$S \rightarrow 0A \mid 1B \mid 0 \mid 1$

$A \rightarrow 0S \mid 1B \mid 1$

$B \rightarrow 0A \mid 1S$

Construct LMD & parse tree for following :-

i) 0101

ii) 1100101

Sol) i) 0101

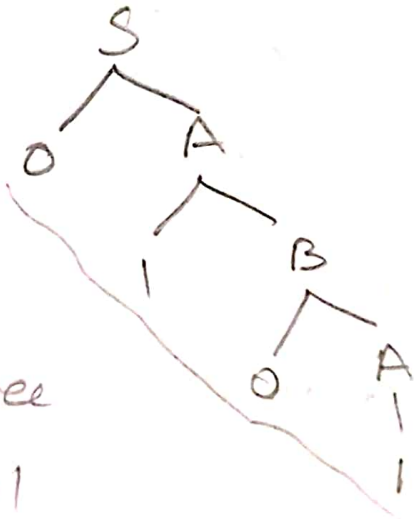
$S \rightarrow 0\underline{A}$

$S \rightarrow 01\underline{B}$

$S \rightarrow 010\underline{A}$

$S \rightarrow 0101$

parse tree



yield of tree

0101

ii) 1100101

$S \rightarrow 1\underline{B}$

$S \rightarrow 11\underline{S}$

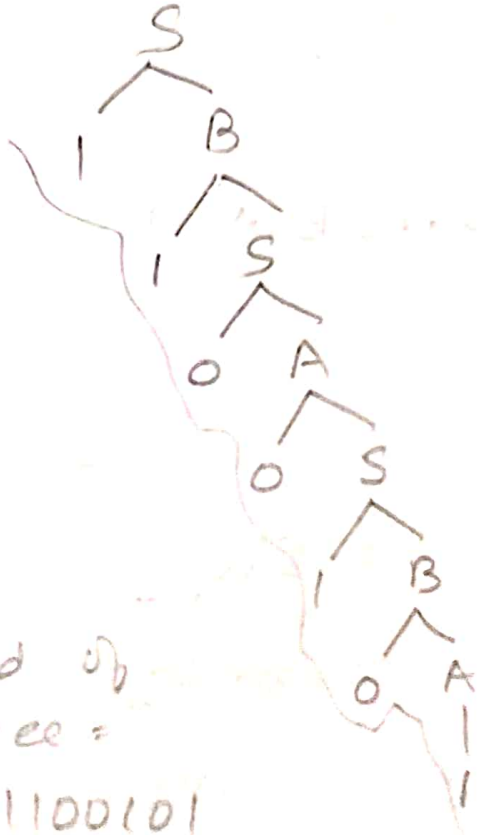
$S \rightarrow 110\underline{A}$

$S \rightarrow 1100\underline{S}$

$S \rightarrow 11001\underline{B}$

$S \rightarrow 11001\,0\underline{A}$

$S \rightarrow 1100101$



yield of tree =

1100101

⇒ Desing LMD & RMD for string aaabaab
grammar is S→as|asbs|ϵ

So LMD

S → aS

S → aaS

S → aaaSbS

~~S→aaaaSbS~~

S → aaaϵbS

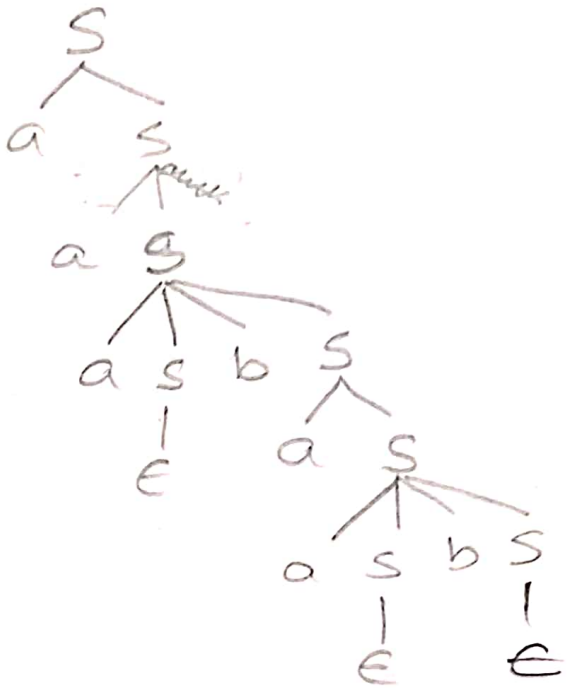
S → aaabaS

S → aaabaaSbS

S → aaabaaϵbS

S → aaabaaϵbϵ

S → aaabaaab



aaabaaab

RMD:-

S → aSbS

S → aSbϵ

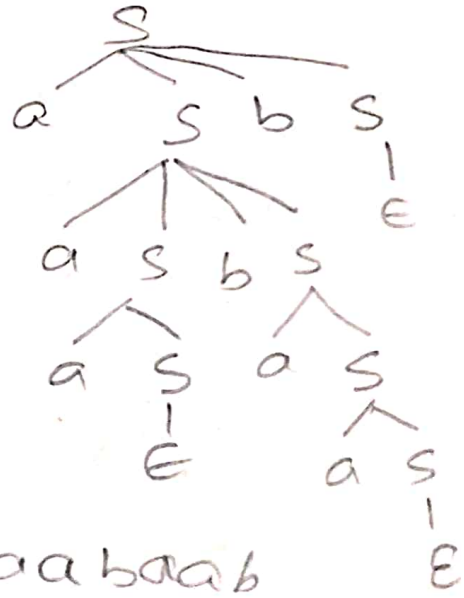
S → aaSbSbϵ

S → aaSbaSb

S → aaSbaaSb

S → aaSbaaϵb

S → aaaSbaab



aaabaaab

$S \rightarrow aaa\varepsilon baab$

$S \rightarrow aaabaab$

8) Consider the grammar. aabb is accepted or not

$S \rightarrow aAB \mid bA \mid \varepsilon$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bB \mid \varepsilon$

Sol

$S \rightarrow aAB$

$S \rightarrow aaAbB$

$S \rightarrow aa\varepsilon bB$

$S \rightarrow aabbB$

$S \rightarrow aabb\varepsilon$

$S \rightarrow aabb$

$\therefore$ It is accepted

9) Consider the grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

a) What are the non-terminals, terminals & start symbol.

b) Find LMD, RMD & parse tree for the following

(i) (a, a)

(ii) (a, (a, a))

Sol (a) non-terminal = {S, L}

terminals = {(,), a, ,}

start symbol = S

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

$L \to L, S \mid S$

(i) (a, a)

LMD

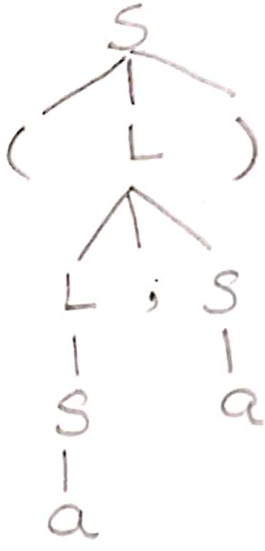
$S \to (L)$

$S \to (L, S)$

$S \to (S, S)$

$S \to (a, S)$

$S \to (a, a)$



(a, a)

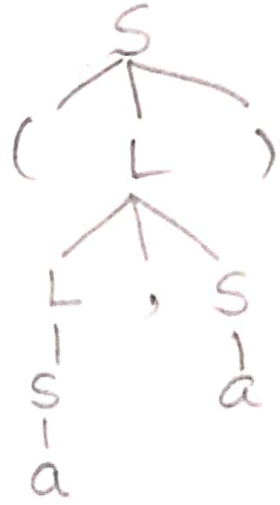
RMD

$S \to (L)$

$S \to (L, S)$

$S \to (L, a)$

$S \to (S, a)$



(a, a)

$S \rightarrow (a,a)$

(ii) $(a,(a,a))$

RMD

$S \rightarrow (\underline{L})$

$S \rightarrow (L, \underline{S})$

$S \rightarrow (L, (\underline{L}))$

$S \rightarrow (L, (L, \underline{S}))$

$S \rightarrow (L, (L, a))$

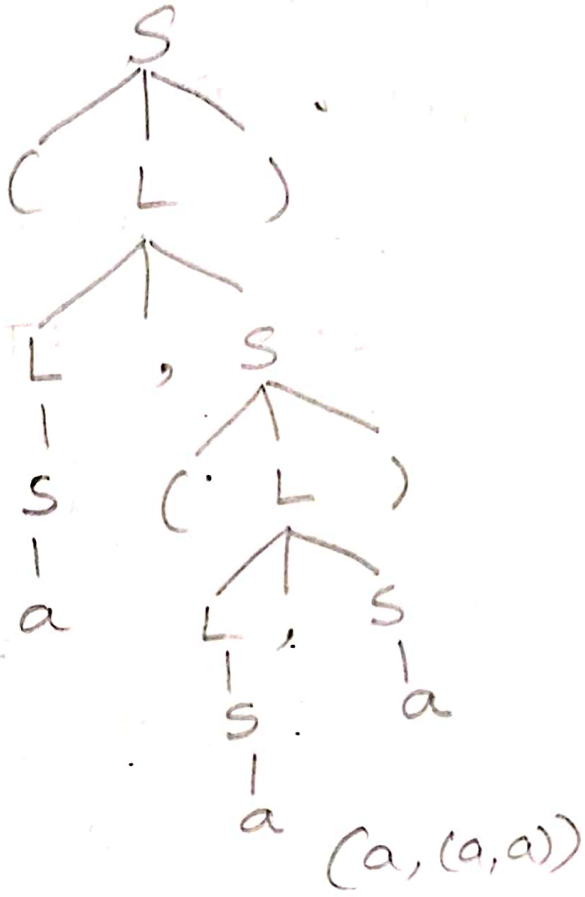
$S \rightarrow (L, (\underline{L}, a))$

$S \rightarrow (L, (\underline{S}, a))$

$S \rightarrow (\underline{L}, (a,a))$

$S \rightarrow (S, (a,a))$

$S \rightarrow (a, (a,a))$



$(a,(a,a))$

LMD

$S \rightarrow (\underline{L})$

$S \rightarrow (\underline{L}, S)$

$S \rightarrow (\underline{S}, S)$

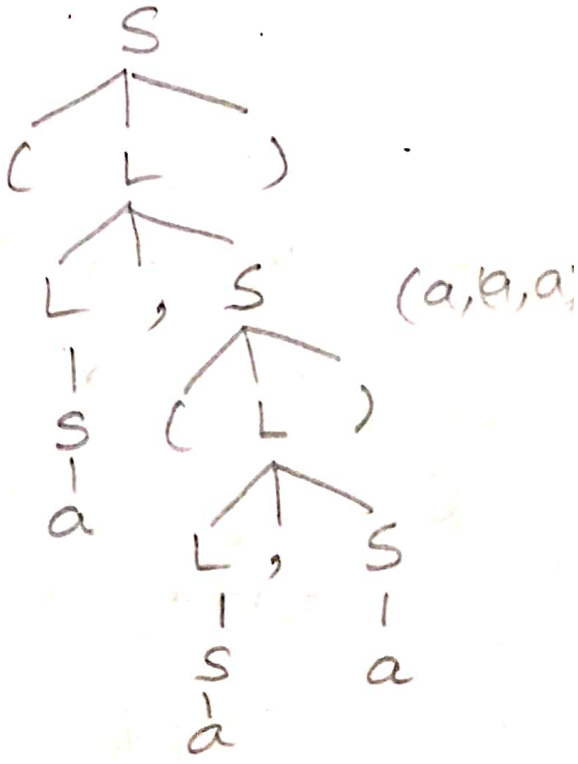
$S \rightarrow (a, \underline{S})$

$S \rightarrow (a, (\underline{L}))$

$S \rightarrow (a, (\underline{L}, S))$

$S \rightarrow (a, (\underline{S}, S))$

$S \rightarrow (a, (a, \underline{S}))$ $= (a,(a,a))$



$(a,(a,a))$

⇒ **Ambiguity**:- A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for atleast one string

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

string: id + id * id.

So

LMD:-

$E \rightarrow \underline{E}+E$

$E \rightarrow id+\underline{E}$

$E \rightarrow id+\underline{E}*E$

$E \rightarrow id+id*\underline{E}$

$E \rightarrow id+id*id$

LMD:-

$E \rightarrow \underline{E}*E$

$E \rightarrow \underline{E}+E*E$

$E \rightarrow id+\underline{E}*E$

$E \rightarrow id+id*\underline{E}$

$E \rightarrow id+id*id$

RMD:-

$E \rightarrow E+\underline{E}$

$E \rightarrow E+E*\underline{E}$

$E \rightarrow E+\underline{E}*id$

$E \rightarrow \underline{E}+id*id$

$E \rightarrow id+id*id$

RMD:-

$E \rightarrow E*\underline{E}$

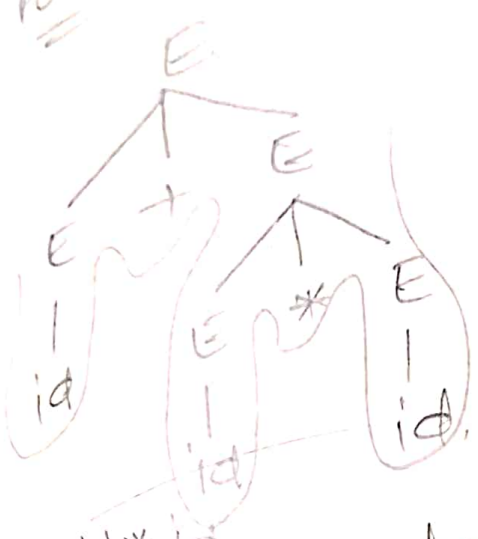
$E \rightarrow \underline{E}*id$

$E \rightarrow E+\underline{E}*id$

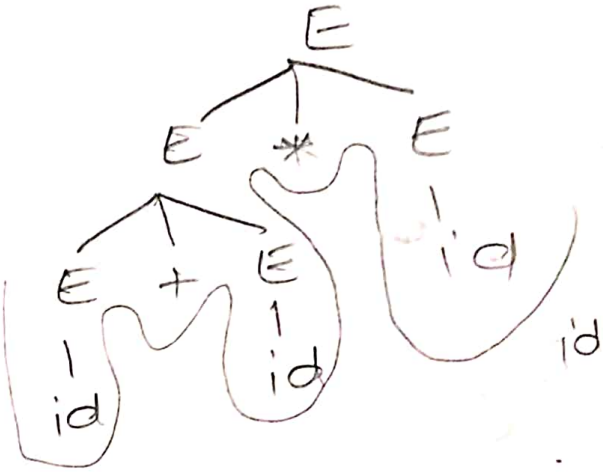
$E \rightarrow \underline{E}+id*id$

$E \rightarrow id+id*id$

Parse tree 1 =>



Parse tree 2



id+id*id

id+id*id

So, since it has more than 1 parse tree

∴ The given grammar is ambiguous.

(2) S.T the following is ambiguous.

$S \rightarrow aSbS$

$S \rightarrow bSaS$

$S \rightarrow \epsilon$

string = abab

Sol RMD

$S \rightarrow aSbS$

$S \rightarrow aSbb\epsilon$

$S \rightarrow abSabb$

$S \rightarrow abbabb$

$S \rightarrow abab$

LMD

$S \rightarrow aSbS$

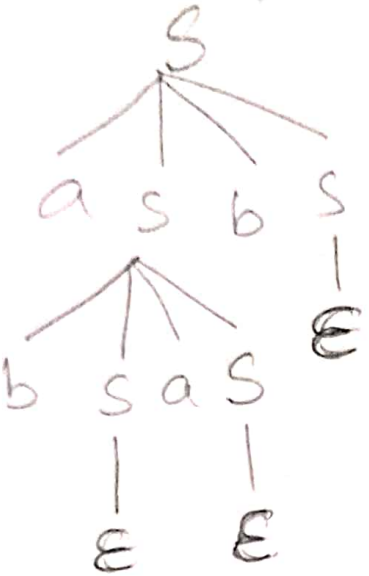
$S \rightarrow abaSbS$

$S \rightarrow ababSb\epsilon$

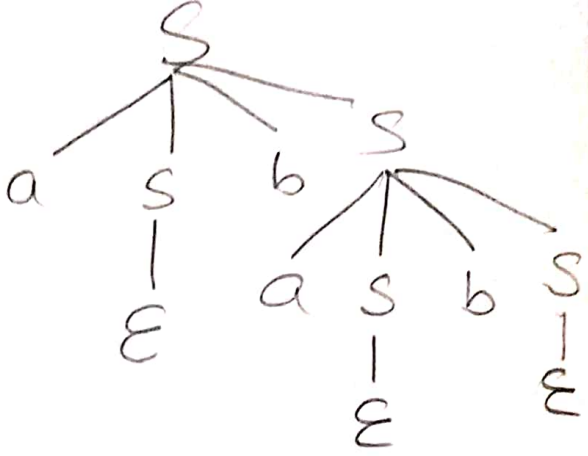
$S \rightarrow ababSb$

$S \rightarrow ababb\epsilon$

$S \rightarrow abab$



abab



abab

∵ it has more than one parse tree

∴ The given grammar is ambiguous.

Q P.T the following grammar is ambiguous

$S \to AB$

$B \to ab$

$A \to aa$

$A \to a$

$B \to b$

aab

$A \to A\alpha / \beta$

$A \to \beta A'$

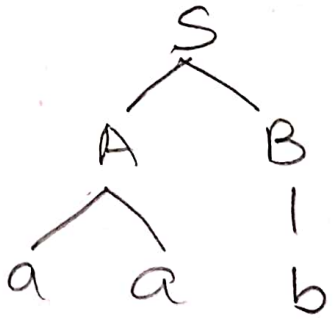
$A' \to \alpha A' / \varepsilon$

Sol LMD

$S \to \underline{A}B$

$S \to aa\underline{B}$

$S \to aab$



$A \to \alpha A / B$

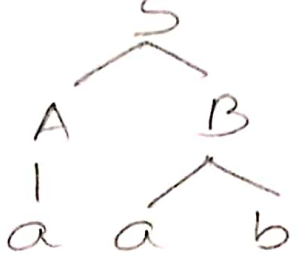
$A \to A'B$

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

$S \to AB$

$S \to aB$

$S \to aab$



∴ it has more than one parse tree

∴ The given grammar is ambiguous.

⇒ Left Recursion :- A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.

Ex: $A \to A\alpha / \beta$ $\quad A \to A\alpha / \beta$

→ There are two types of left recursion :-

(i) Direct left recursion Ex :- $B \to Ba$

(ii) Indirect left recursion Ex :- $S \to Aa$

$A \to Sm$

$A \to A\alpha / \gamma$

→ Removal of left recursion :-

→ The top down parsers cannot accept the grammar having left recursion (i.e; they do not allow left recursion grammar).

→ So, we have to remove left recursion but preserve the language generated by grammar.

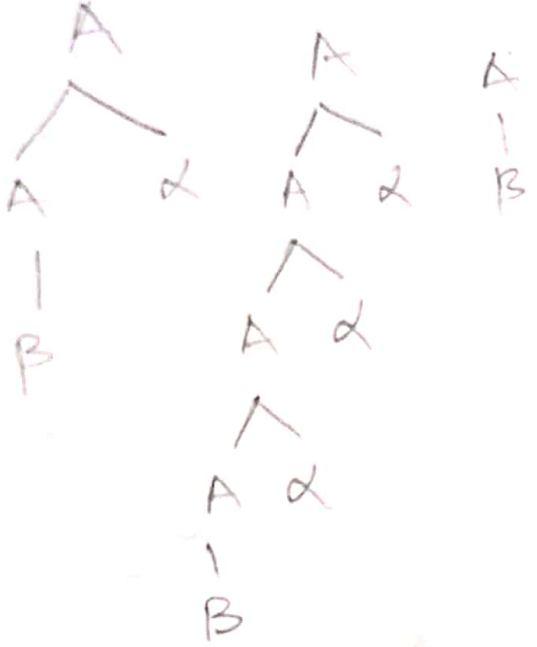
⇒ Recursion :- It can be left (LR) or right (RR)

Recursion

$(A \to A\alpha / \beta)$ LR $\qquad$ RR $(A \to \alpha A / \beta)$

LR:-

$A \rightarrow A\alpha / \beta$

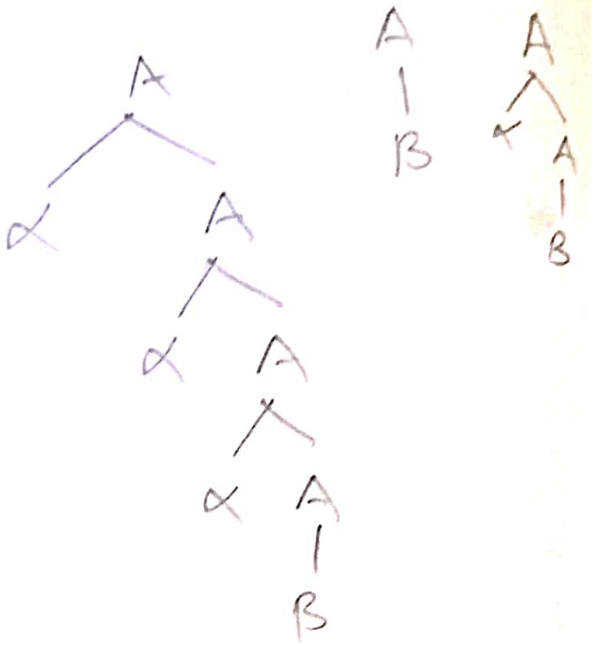


lang = $\{\beta, \beta\alpha, \beta\alpha\alpha \ldots\}$

$= \beta\alpha^*$

RR

$A \rightarrow \alpha A / \beta$



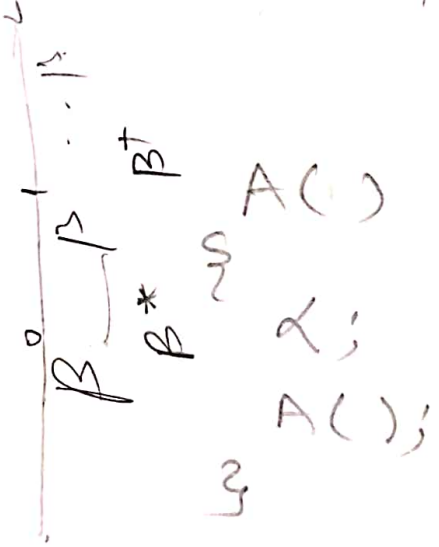
lang = $\{\beta, \alpha\beta, \alpha\alpha\beta \ldots\}$

$= \alpha^*\beta$.

Q1 we write in func

```
A()
{
    A();
    α;
}
```

infinite loop



```
A()
{
    α;
    A();
}
```

→ There is a problem in left recursion, the recursive function calls itself infinite times.

→ In RR (α -> Base condition) α is evaluated, hence it doesn't lead to infinite loop

→ Hence, we have to remove LR

LR:-

$A \rightarrow A\alpha / \beta$

Now, we want $\beta\alpha^*$ → $A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

lang:- $\beta\alpha^*$

$A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

$\therefore$ $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' / \varepsilon$ $\equiv A \rightarrow A\alpha / \beta$

→ This RR grammar is equivalent to the LR grammar.

→ So, to eliminate left recursion, we will follow these rules.

Ex:- $E \rightarrow E + T / T$ remove LR

Compare:- $A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$, $A' \rightarrow \alpha A' / \varepsilon$

$E \rightarrow \underset{\downarrow A}{E} \underset{\downarrow \alpha}{+T} / \underset{\downarrow \beta}{T}$

$E \rightarrow TE'$

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

(2) $S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

$L \rightarrow L, S \mid S$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Sol $S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow , SL' \mid \varepsilon$

(3) $A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Sol $A \rightarrow bdA' \mid A'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

(4) $\underset{A}{S} \rightarrow \underset{A}{S}\underset{\alpha}{0S1S} \mid \underset{\beta}{01}$

Sol $S \rightarrow 01S'$

$S' \rightarrow 0S1SS' \mid \varepsilon$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

Q) $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots$
$\beta_1 \mid \beta_2 \mid \beta_3$

Sol $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$

$A' \rightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots$

(iv) $E \rightarrow E+T / T$ — (1)

$T \rightarrow T*F / F$ — (2)

$F \rightarrow id$

$A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

remove LR for (1) $A \rightarrow 1A'$, $A' \rightarrow \alpha A' / \varepsilon$

(v) $A \rightarrow A\alpha / \beta$

$E \rightarrow E+T / T$

$E \rightarrow TE'$

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

Similarly for (2)

$T \rightarrow T*F / F$

$T \rightarrow FT'$

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

∴ The grammar became.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id$

(Q) $S \rightarrow Aa$ (Indirect LR)

$A \rightarrow Sb/c$

Sol Convert into direct

$A \rightarrow Aab/c$

$A \rightarrow cA'$

$A' \rightarrow abA'/\varepsilon$

So after converting it becomes.

$S \rightarrow Aa$

$A \rightarrow cA'$

$A' \rightarrow abA'/\varepsilon$

Q $S \rightarrow ABC$ —①

$A \rightarrow Aa/Ad/b$ —②

$B \rightarrow Bb/c$ —③

$C \rightarrow Cc/g$ —④

remove LR from ②

$A \rightarrow Aa/Ad/b$

$A \rightarrow bA'$

$A' \rightarrow aA'/\varepsilon/dA'$

remove LR from (3)

$B \rightarrow CB'$

$B' \rightarrow bB'/\varepsilon$

remove LR from (4)

$C \rightarrow gC'$

$C' \rightarrow cC'/\varepsilon$

∴ The grammar becomes.

$S \rightarrow ABC$

$A \rightarrow bA'$

$A' \rightarrow aA'/\varepsilon/dA'$

$B \rightarrow CB'$

$B' \rightarrow bB'/\varepsilon$

$C \rightarrow gC'$

$C' \rightarrow cC'/\varepsilon$

$A \rightarrow \alpha\beta_1/\alpha\beta_2/\alpha\beta_2$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1/\beta_2/\beta_n$

→ Left factoring :- LF is a grammar transformation that is useful for producing a grammar suitable for top-down parsing.

→ A grammar is said to have LF, if it has production in the form

$A \rightarrow \alpha\beta_1/\alpha\beta_2/\alpha\beta_2$ $\quad A \rightarrow \alpha\beta_1/\alpha\beta_2/\alpha\beta_2$

then replace it with following production

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \ldots$

$A \to \alpha\beta_1 | \alpha\beta_2 | \beta_3 | \ldots$

→ This is also called prefix problem or non-deterministic grammar.

$A \to \alpha\beta_1 | \alpha\beta_2 | \ldots$

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \ldots$

Q) $S \to iEtSeS | iEtS | a$

$E \to b$

Sol) $S \to iEtSS' | a$

$S' \to eS | \varepsilon$

$E \to b$

Q) $S \to aSSbS | aSaSb | abb | b$

Sol) $S \to aS' | b$

$S' \to SSbS | SaSb | bb$ again LF

↓ Convert.

$S' \to SS'' | bb$

$S'' \to SbS | aSb$

∴ Ans is

$S \to aS' | b$

$S' \to SS'' | bb$

$S'' \to SbS | aSb$

Q) $S \rightarrow bSSaaS/bSSasb/bSb/a$

$S \rightarrow bSS'/a$

$S' \rightarrow SaaS/Sasb/b$

again LF convert it

$S' \rightarrow SaS''/b$

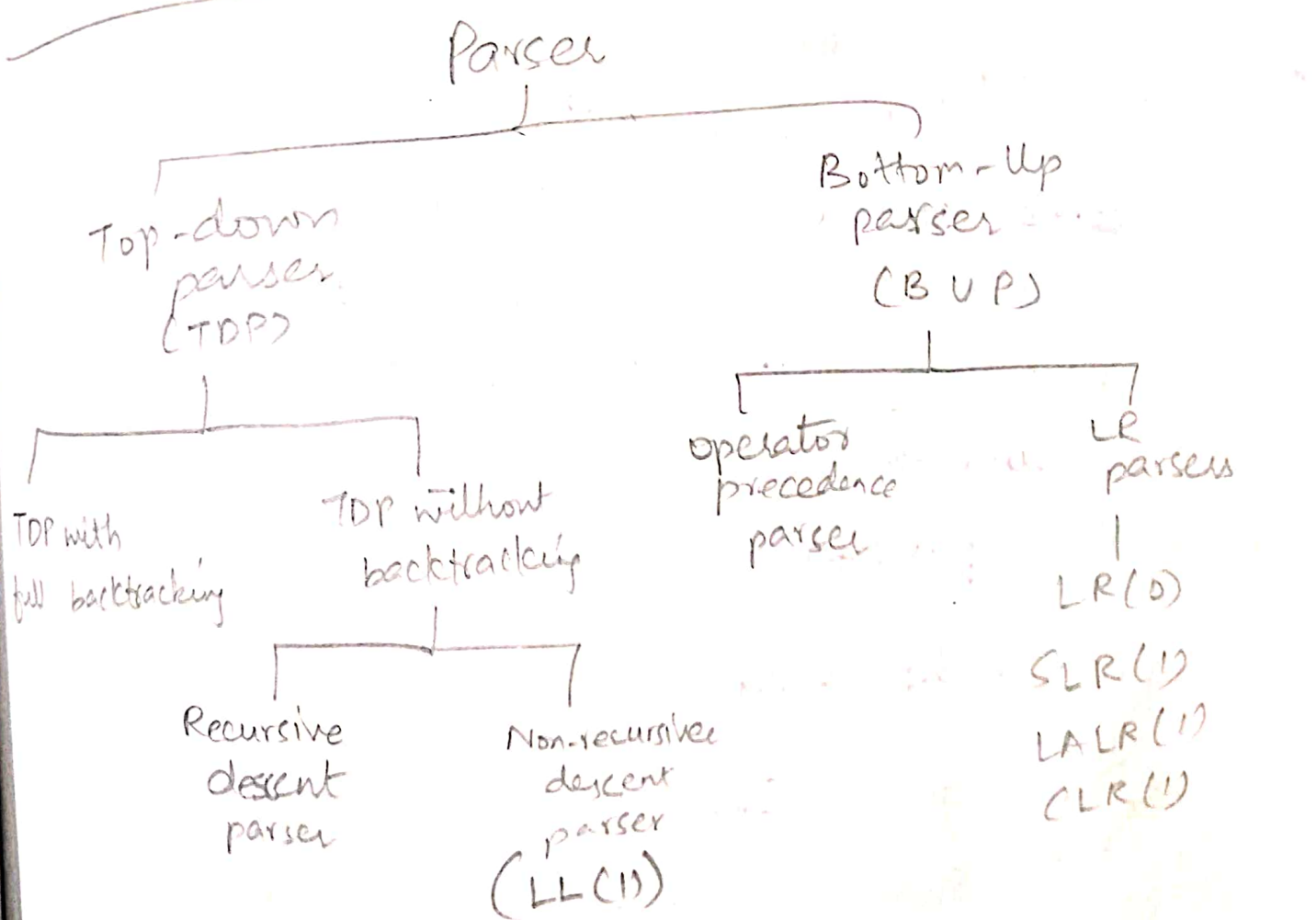
$S'' \rightarrow aS/Sb$

∴ Answer becomes

$S \rightarrow bSS'/a$

$S' \rightarrow SaS''/b$

$S'' \rightarrow aS/Sb$



⇒ Parser:-

Parser is a phase of compiler process which takes a sequence of tokens as input and produces output in the form of parse tree. It is also known as Syntax Analyzer.

→ Types:-

It is mostly classified into 2 types:-

1. Top down Parser.
2. Bottom Up parser.

→ Backtracking:- When the parser tries to pick up a production, there may exist many productions for the same non-terminals. If the parser makes the wrong choice, it will end up with incorrect derivation. It goes back and pick another production and then attempt to derive the string again. This problem is called back tracking.

Q) Apply backtracking to the given grammar.

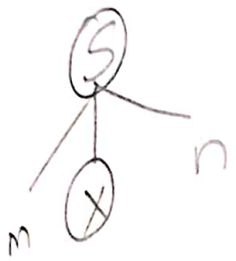
$S \longrightarrow mXn \mid mZn$

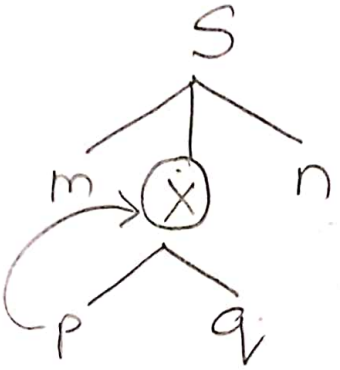
$X \longrightarrow pq \mid sq$

$Z \longrightarrow qr$

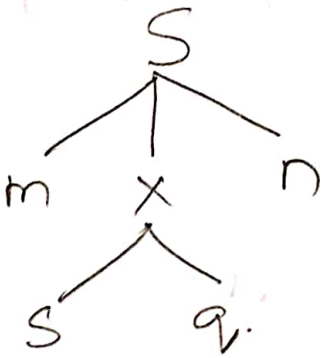
read the string "msqn".

msqn





mpqn ≠ msqn
So backtracking is applied



msqn = msqn

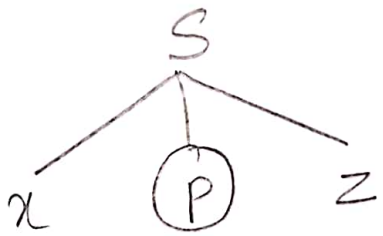
Q) $S \rightarrow xPz$

$P \rightarrow yw / y$

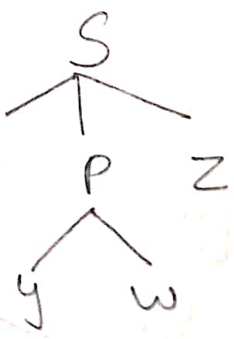
now obtain an input string xyz

Sol

$S \rightarrow xPz$



→ replace P with yw then parse tree is



$S \rightarrow xPz$

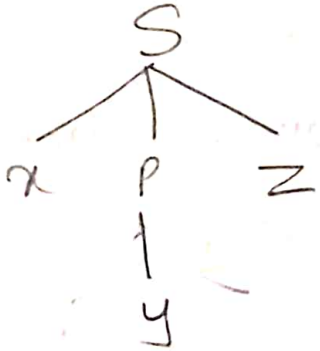
$P \rightarrow yw$

$S \rightarrow xywz$

$\neq xyz$

we obtain xywz which is not equal to xyz

backtrack i.e; move backward to P



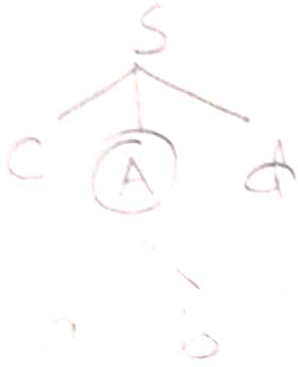
string xyz is achieved

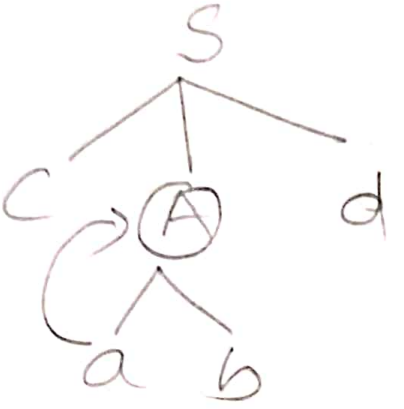
$S \rightarrow xPz$

$P \rightarrow y$

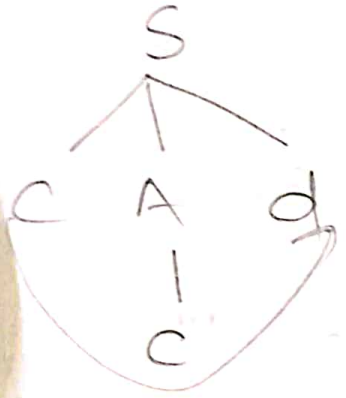
Q) S → cAd

A → ab/d derive the string cdd





cabd is achieved which is not the string we require So, we backtrack i.e, go back to



string ccd is achieved.

⇒ Top-Down Parser:-

→ In top down parsing parse tree is created from top to bottom. (i.e, we start from the root and work down the leaves)

→ TDP can be viewed as an attempt to find the leftmost derivation for an input string to check the validity of a string

→ In TDP, a string is derived using LMD from the starting symbol of the grammar.

→ TDP expands a parse tree from the start symbol to the leaves by consuming tokens generated by lexical analyzer.

→ TDP constructs from the grammar which is free from ambiguity & left recursion.

→ It allows a grammar which is free from left factoring.

→ Top down parsing is divided into two types:-

(1) TDP with backtracking : Brute Force technique

(2) TDP without " :

(i) Recursive Descent Parsing.

(ii) Non-recursive parsing (or) Predictive parsing (or) LL(1) parsing (or) Table Driven Parsing.

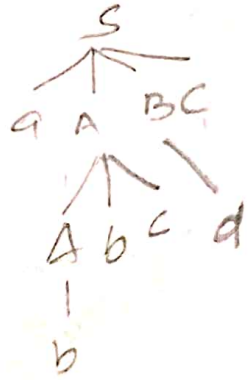
Example of TDP:-

S → a ABC

A → Abc | b

B → d

, string: abbcde

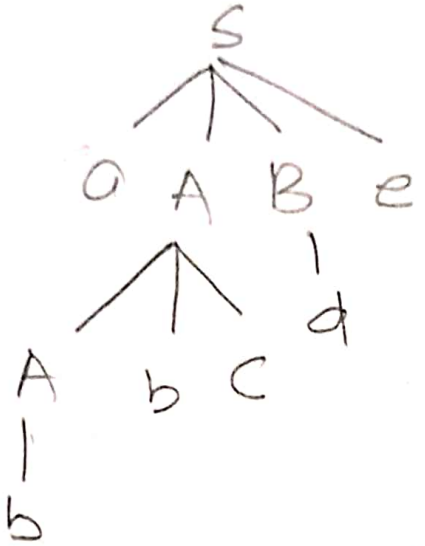


Sol string = abbcde

$S \to aABe$

$A \to Abc \mid b$

$B \to d$



yield = abbcde

LMD

$S \to aABe$

$S \to aAbcBe$

$S \to abbcBe$

$S \to abbcde$

⇒ Top Down Parser with Backtracking (Brute

→ Here, full backtracking is used to create a parse tree until the correct/given string is generated at leaves.

→ In worst case, when string is not ~~gene~~ in given language all possible combinations are checked before the failure to construct a parse tree for given string is recognized.

Example:

Consider grammar $S \to \varepsilon cAd$

$A \to ab \mid a$

input string $w = cad$

⇒ Step1:- Initially create a tree with single node S. An input ptr points to 'c', the first symbol of w.

⇒ Step2:- The leftmost leaf 'c' matches the first symbol of w, so advance the input ptr to the second symbol of w 'a' and consider the next leaf 'A'.

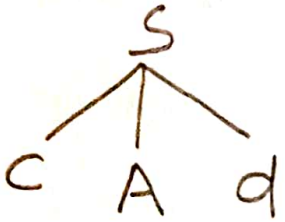
→ Expand A using the first alternative.

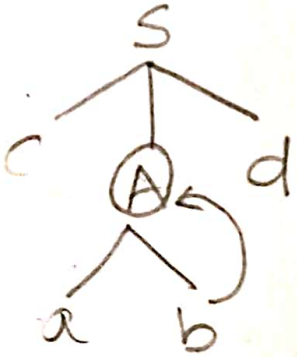
⇒ Step3:- The second symbol 'a' of w also matches with the second leaf of tree. So advance the input ptr to third symbol of w 'd'.

→ But the third leaf of tree is 'b' which does not match with the input symbol 'd' Hence move back the ptr to second position (i.e, backtrack)

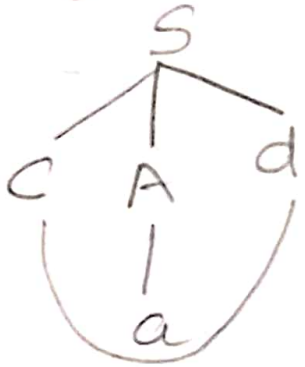
Step4:- Now try the second alternative for A. If matching doesn't occur then match with alternative

→ If it match atleast one alternative then parsing is successful else fail





cabd ≠ cad
backtrack to A



cad = cad
∴ parsing is successful

8) $S \rightarrow aAb$

$A \rightarrow d/c$

string $w = acb$

LMD

$S \rightarrow aAb$ $\quad w = acb$

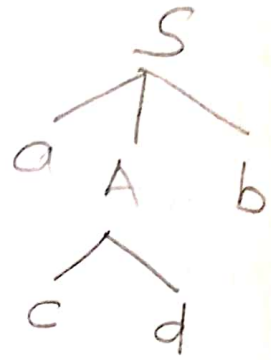
a matches with first char move ptr to second char

$S \rightarrow acdb$ $\quad w = acb$

second also matches move input ptr to third.

The third char $d \neq b$

∴ backtrack the ptr to second position i.e, back to 1 and try the other alternative

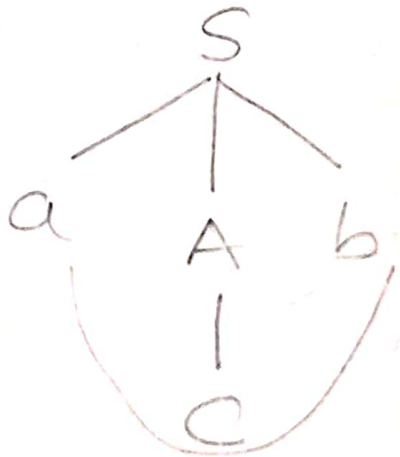


$S \rightarrow aAb$ $\quad w = acb$ matches

$S \rightarrow acb$ $\quad w = acb$

cb, matches with

∴ parsing is successful



acb string matches with

Recursive Descent Parsing:-

→ A TDP that uses collection of recursive procedures for passing the given i/p string is called as Recursive Descent Parser.

→ In RDP the CFG is used to build the recursive routines.

→ A procedure is associated with each non-terminal of a grammar.

→ The R.H.S of the production rule is directly converted to a program which is the body of the corresponding non-terminal of LHS

⇒ **Basic steps for construction of RDP:-**

1) Write procedure for start variable production of given grammar. The R.H.S of the production is directly converted into program code symbol by symbol.

2) If the input symbol is non-terminal then call procedure of that NT

Example:-

$$E \rightarrow num\ T$$

$$T \rightarrow * num\ T / \varepsilon$$

```
procedure E ( )
{
    if (lookahead = num) then
```

```
    {
      match (num)
       T()
     }
else
     error();
    if (lookahead = $)
    {
     declare success;
    }
else
     error();
}

Procedure T()
{
    if (lookahead = '*')
    {
     match ('*')
     if (lookahead = 'num') then
      {
       match ('num')
        T();
      }
}
```

```
else
  error();

}
else NULL;
}
procedure match(Token t)
{ if (lookahead = t)
    lookahead = next token;

  else
    error();
  }
```

Q) $E \rightarrow T + E / T$

$T \rightarrow V * T / V$

$V \rightarrow id.$

Sol

```
procedure E()
{
  T();
  if (lookahead = '+')
  {
    match('+');
    E();
  }
else
```

```
    error;
  if (lookahead = $)
    declare success;
  }
procedure T()
{
  V();
  if (lookahead = '*')
  {
    match('*');
    T();
  }
  else
    error();
}
procedure V()
{
  if (lookahead='id')
  {
    match('id')
  }
  else
    error();
}
Procedure match(Token t)
{
```

```
if (lookahead = t)
   lookahead = next token;

else
   error;
}
```

2) $E \rightarrow E+T/T$

$T \rightarrow TF/F$

$F \rightarrow F*/a/b$

$A \rightarrow A\alpha/\beta$

$A \rightarrow \beta A'$

$A' = \alpha A'/\varepsilon$

Sol Convert first from LR

$E \rightarrow TE'$

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

convert second

$T \rightarrow FT'$

$T' \rightarrow FT'/\varepsilon$

Illy, $F \rightarrow aF'/bF'$

$F' \rightarrow *F'/\varepsilon$

After eliminating LR, then grammar is

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow FT'/\varepsilon$

$F \rightarrow aF'/bF'$

$F' \rightarrow *F'/\varepsilon$

Now the RDP is

```
procedure E()
{
   T();
   E'();
}
procedure E'()
{
   if(lookahead='+')
   {
    match('+');
    T();
    E'();
   }
else
   NULL;
}
procedure T()
{
   F();
   T'();
}
procedure T'()
{
   if (true)
   {
      F();
      T'();
   }
else
   NULL;
}
procedure F()
{
 if(lookahead='a'||lookahead='b')
 {
  match(a||b)
 }
else
  error;
}
procedure F'()
 {
  if(lookahead='*')
  {
   match('*')
    F'();
  }
  else
    NULL;
  }
```

⇒ Advantages

→ It's easy to construct

→ Overhead associated with backtracking is eliminated.

⇒ Disadvantages-

- We cannot write Recursive descent parser for all types of CFG.
- It can be implemented only for those languages which supports recursive procedure calls.

⇒ LL(1) parser:

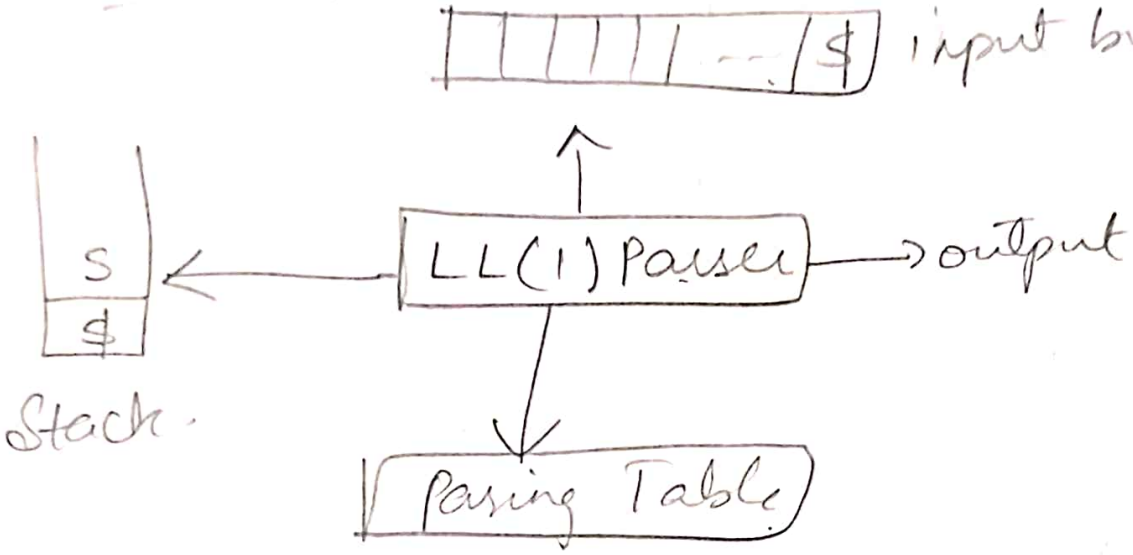
→ It is also known as non-recursive descent parser (or) predictive parse.

→ This top-down parser/parsing algorithm is of non-recursion type.

→ In this type of parsing a table is built for LL(1).

→ The first L means the i/p is read from left to right. The second L means it uses left most derivation for i/p string. The number 1 in the i/p symbol means it takes only one i/p symbol (lookahead) to predict the parsing process.

→ Block diagram for LL(1):



→ Datastructures used by LL(1) parser are
(1) Input buffer.
(2) Stack
(3) parsing table

→ No left recursive or ambigious grammar can be LL(1)

→ LL(1) parser uses i/p buffer to store the i/p tokens

→ predictive parser has the capability to predict which production is to be used to replace the i/p string

→ It generates a parse tree from top to bottom

→ Both the stack & i/p contains an end symbol $ where $ in stack represent empty stack
$ in i/p. represents i/p is consumed.

→ The parser consults the table M[A, a] (A → top of stack, a → i/p) each time while taking the actions. Hence, this type of parsing method is called table driven parsing algorithm

⇒ Steps to be followed:-

1) for given grammar remove left recursion if required.
2) Do left factoring if required.
3) Calculate first & follow sets for each & every variable of the grammar
4) Construct the table
5) Parse the string using the table

⇒ First(A) Contains all the terminals present in first place of every string derived by A.

eg:- S → abc | def | ghi

first(S) = {a, d, g}

(2) S → ABC | ghi | jkl ⇒ f(S) = {a, b, c, g, j}

A → a | b | c ⇒ f(A) = {a, b, c}

B → b ⇒ f(B) = b

D → d ⇒ f(D) = d

Q $S \to ABC$ $f(S) = \{a, b, \varepsilon, d, e, f\}$

$A \to a \mid b \mid \varepsilon$ $f(A) = \{a, b, \varepsilon\}$

$B \to c \mid d \mid \varepsilon$ $f(B) = \{c, d, \varepsilon\}$

$C \to e \mid f \mid \varepsilon$ $f(C) = \{e, f, \varepsilon\}$

Sol $f(S) = \{a, b, c, d, e, f, \varepsilon\}$.

∴ its ε substitute it place of A it become

$S \to \varepsilon BC$

first of S becomes first of B

again ε sub in place of B

$S \to \varepsilon\varepsilon C$

first of S is first of C

$E \to TE'$ $F(E) = \{id, (\}$

$E' \to *TE' \mid \varepsilon$ $F(E') = \{*, \varepsilon\}$

$T \to FT'$ $F(T) = \{id, (\}$

$T' \to \varepsilon \mid +FT'$ $F(T') = \{\varepsilon, +\}$

$F \to id \mid (E$ $F(T) = \{id, (\}$

→ Follow (A) :- It contains set of all terminals present immediately to the right of A.

→ follow of start symbol is $

→ follow never contains ε.

Example:

S → AaAb | BbBa.

A → ε

B → ε.

FO(A) = {a, b}

fo(B) = {b, a}

Q S → ABC C → ε
A → DEF D → ε
B → ε E → ε
F → ε

Sol Fo(A) = F(B) but its F(B) ε then put ε
= F(C) = ε
= fo(S) = $

S → ABCD

A → b

B → C

C → d

D → e

fo (A) = {c}

fo (S) = {$}

fo (C) = {c}

fo (B) = {d}

fo (D) = fo (S)
= $

S → ABCD, A → b/ε, B → c/ε,
C → d, D → c/ε

Sol fo (S) = {$}

fo (A) = {c, d}

fo (B) = {d}

fo (C) = {c, $}

fo (D) = fo (S) = {$}

Q) Calculate FIRST( ) & follow of given

$S \to A, A \to aB/Ad, B \to b, C \to g$

So remove left recursion.

$A \to \beta A'$

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

~~$A \to Aa/\beta$~~

$A \to aBA'$

$A' \to dA'/\varepsilon$

So grammar becomes

$S \to A$

$A \to aBA'$

$A' \to dA'/\varepsilon$

$B \to b$

$C \to g$

$f(S) = \{a\}$

$f(A) = \{a\}$

$f(A') = \{d, \varepsilon\}$

$f(B) = \{b\}$

$f(C) = \{g\}$

$fo(S) = \{\$\}$

$fo(A) = fo(S) = \{\$\}$

$fo(A') = \{\$\}$

$fo(B) = \{d, \$\}$

$fo(C) = \{$ not accepted $\}$

First & follow rules

First

1) If $A \rightarrow a\alpha$,
then $F(A) = a$

2) If $A \rightarrow \varepsilon$,
then $F(A) = \varepsilon$

3) If $A \rightarrow BC$
then $F(A) = f(B)$ if $F(B)$ doesn't contain epsilon
If $F(B)$ contains ε then $F(A) = f(B) \cup F(C)$.

Follow:

1) follow of start symbol is \$, if A is start
$fo(A) = \$$

2) $S \rightarrow ACD$
$C \rightarrow a/b$
$Fo(A) = Fo(C) = \{a, b\}$
$Fo(D) =$ there is nothing we can consider it as ε. then in that case
$Fo(D) = Fo(S) = \{\$\}$

3) $S \rightarrow aSbS / bSaS / \varepsilon$
$fo(S) = \{\$, b, a\}$
we aren't considering the other two S bcoz there is nothing beside them then $Fo(S) = fo(S)$; both are same
∴ We do not write anything related to them

Q) $E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

$A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

Sol: eliminate left recursion from (1) & (2)

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

After elimination of LR it becomes

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id.$

Step 2: Remove left factoring

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \ldots$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2 / \beta_3$

There is no left factoring

Step 3 Calculation of first & follow

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

$F(E') = \{+, \varepsilon\}$

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

$F(T') = \{*, \varepsilon\}$

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

$Fo(E) = \{), \$\}$

$Fo(E') = \{), \$\}$

$Fo(T) = \{), +, \$\}$

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

$Fo(F) = \{*,), +, \$\}$

Step 4:- Construction of parsing table

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

Step 5:- Check whether input string is accepted or not let us consider i/p = id + id $

stack	input	Action
$	id + id $	-
$E	id + id $	$E \to TE'$ (push RHS in stack in reverse order)
$E'T	id + id $	$T \to FT'$ push
$E'T'F	id + id $	$F \to id$
$E'T'id	id + id $	pop id from stack & remove id from i/p string

$E'T'	+id$	$T' \to \varepsilon$ pop
$E'	+id$	$E' \to +TE'$ pop
$E'T+	+id$	pop + from stack
$E'T	id$	$T \to FT'$ push
$E'T'E	id$	$F \to id$ push
$E'T'id	id$	pop
$E'T'	$	$T' \to \varepsilon$ push
$E'	$	$E' \to \varepsilon$ push
$	$	accept

After performing entire string, if the stack & i/p string contains $,
The string is accepted

Q) $S \to A$, $A \to aB/Ad$ & $B \to b$, $C \to g$

Sol Step 1:- elimination of LR from A

$A \to aB/Ad$

$A \to aBA'$

$A' \to dA'/\varepsilon$

$S \to A$

$A \to aBA'$

$A' \to dA'/\varepsilon$

$B \to b$

$C \to g$

Step 2:- elimination of left factoring

There is no left factoring

Step 3:- find first & follow

$F(S) = \{a\}$ $Fo(S) = \{\$\}$

$F(A) = a$ $Fo(A) = \{\$\}$

$F(A') = \{d, \varepsilon\}$ $Fo(A') = \{\$\}$

$F(B) = \{b\}$ $Fo(B) = \{d, \$$

$F(C) = \{g\}$ $Fo(C) =$ ~~$\{g\}$~~ not accepted

Step 4:- Parse table

	a	b	d	g	$
S	S→A				
A	A→aBA'				
A'			A'→dA'		A'→ε
B		B→b			
C				C→g	

Step 5:-

Check whether the i/p string can be accepted or not.

Consider string abd$

Stack	Input String	Action
$	abd$	
$S	abd$	S → A push
$BA	abd$	A → aBA' push
$A'Ba	abd$	pop
$A'B	bd$	B → b
$A'b	bd$	pop
$A'	d$	A' → dA'
$A'd	d$	pop
$A'	$	A' → ε
$	$	

After performing entire string if the stack string constitutes $

∴ the string is accepted

Q Consider $E \to TE'$, $E' \to +TE' / \varepsilon$, $T \to FT'$, $T' \to *FT' / \varepsilon$, $F \to (E) / id$

Sol

$F(E) = \{(, id\}$ $\quad$ $Fo(E) = \{), \$\}$

$F(E') = \{+, \varepsilon\}$ $\quad$ $Fo(E') = \{\$,)\}$

$F(T) = \{(, id\}$ $\quad$ $Fo(T) = \{+, \$,)\}$

$F(T') = \{*, \varepsilon\}$ $\quad$ $Fo(T') = \{+, \$,)\}$

$F(F) = \{(, id\}$ $\quad$ $Fo(F) = \{*, +, \$,)\}$

Parse table

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

The given grammar is LL(1) since each cell consist of only one production rule

Stack implementation

Let us consider string id+id*id

stack	input	production
$	id+id*id$	
$E	id+id*id$	E→TE'
$E'T	id+id*id$	T→FT'
$E'T'F	id+id*id$	F→id
$E'T'id	id+id*id$	pop
$E'T'	+id*id$	T'→ε
$E'	+id*id$	E'→+TE'
$E'T+	+id*id$	pop
$E'T	id*id$	T→FT'
$E'T'F	id*id$	F→id
$E'T'id	id*id$	pop
$E'T'	*id$	T'→*FT'
$E'T'F*	*id$	pop
$E'T'F	id$	F→id
$E'T'id	id$	pop

$+E'T'$	$	$T' \to \varepsilon$
E'	$	$E' \to \varepsilon$
$	$	

After performing entire string if i/p string & stack constitutes $
∴ String is accepted

⇒ Bottom - Up Parsing:-

→ In Bottom-Up Parsing, the input string is taken first & we try to reduce this string with the help of grammar & try to obtain the start of grammar.

→ The process of bottom-up parsing stops successful as soon as we reach the start symbol.

→ It uses reverse right-most derivation

→ Parse tree is constructed from bottom to top i.e, from leaves to root

→ In BUP, parser tries to identify the RHS of production rule & replace it by corresponding LHS. This activity is called reduction

→ So, the prime task in B.U.P is to find the production that can be used for reduction

Ex:-

$E \rightarrow E+E$

$E \rightarrow id$

i/p string :- id+id+id

RMD:-

$E = E+E$

$E = E+E+E$

$E = E+E+id$

$E = E+id+id$

$E = id+id+id$

Bottom up

id+id+id

id+id+E (E→id)

id+E+E (E→id)

id+E (E→E+E)

E+E (E→id)

E (E→E+E)

Handle & Handle pruning:-

Handle is a substring that maches with the right side of production, then it is replaced with the LHS non-terminal.

Ex: $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

i/p string: abbcde

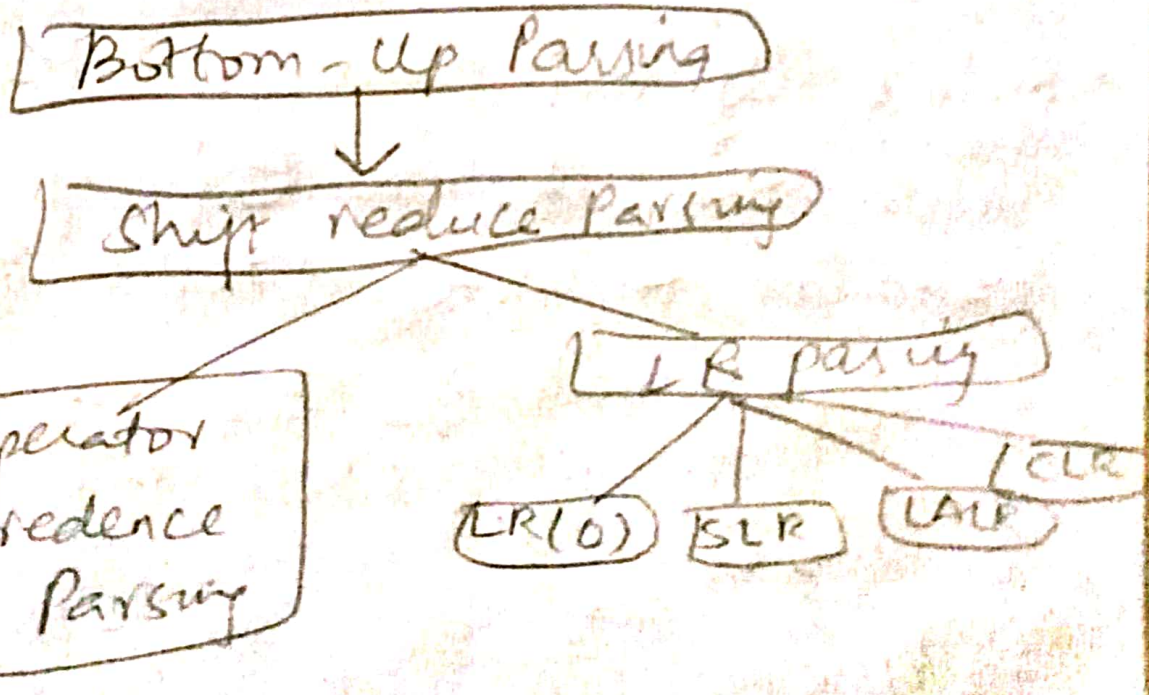
Sentential form	Handle	Reducing prod
abbcde	b	$A \rightarrow b$
aAbcde	Abc	$A \rightarrow Abc$
aAde	d	$B \rightarrow d$
aABe	aABe	$S \rightarrow aABe$
S		

→ Bottom-up parsing uses right-most derivation in reverse order is called handle pruning.

Q $E \rightarrow E + T / T$
$T \rightarrow T * F / F$
$F \rightarrow (E) / id$

i/p string = id * id

Sentential form	Handle	Reducing production
id * id	id	F → id
F * id	F	T → F
T * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T
E		



⇒ Shift reduce parsing

→ It attempts for construction of parse tree in a similar manner as done in bottom up parsing ie, the parse tree is constructed from leaves (bottom) to the root (up). A more general form of shift reduce parser is LR parser.

→ This parser requires some data structure i.e.,

- An input buffer for storing the i/p string
- A stack for storing and accessing the production rules.

⇒ Basic Operations:-

- **Shift**:- This involves moving of symbols from i/p buffer onto the stack.
- **Reduce**:- If the handle appears on the top of the stack then it is reduced by using appropriate production i.e, RHS of the production is popped out of stack & LHS of production is pushed onto the stack.
- **Accept**:- If only start symbol is present in the stack & the i/p buffer is empty then, the parsing action

is called accept. When accept action is obtained it means successful parsing is done.

- Error: This is the situation in which the parser can neither perform shift nor reduce actions & not even accept action.

Q $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

String : id * id

stack	i/p buffer	Action
$	id * id $	shift
$id	* id $	Reduce F→id
$F	* id $	Reduce T→F
$T	* id $	shift
$T*	id $	shift
$T*id	$	reduce F→id
$T*F	$	Reduce T→T*F
$T	$	Reduce E→T
$E	$	Accepted

1. Shift-reduce conflict: It occurs if parser has choice to select both shift action and reduce action. But only one can be selected.

2. Reduce-reduce conflict: It occurs if more than one reduction is possible.

Q) $E \to E-E,\ E \to E*E,\ E \to id$

I/p string- id - id * id

Stack	I/p buffer	Action
$	id - id * id $	shift
$id	- id * id $	reduce E→id
$E	- id * id $	shift
$E-	id * id $	shift
$E-id	* id $	reduce E→id
$E-E	* id $	shift
$E-E*	id $	shift
$E-E*id	$	reduce E→id
$E-E*E	$	reduce E→E*E
$E-E	$	reduce E→E-E

4E	$	Accept

Q) S → TL; T → int / float, L → L, id / id

string = int id, id;

Stack	I/P buffer	Action
$	int id, id; $	shift
$int	id, id; $	reduce T → int
$T	id, id; $	shift
$T id	, id; $	reduce L → id
$TL	, id; $	shift
$TL,	id; $	shift
$TL, id	; $	reduce L → L, id
$TL	; $	shift
$TL;	$	reduce S → TL;
$S	$	accept

⇒ Operator Precedence Parser:-

→ It is mainly used to define mathematical operator for the compiler.

→ We use operator grammar:-

(i) No RHS of any production has a ε.

(ii) No two "Non-terminals" are adjacent at RHS

→ It can only be established b/w the terminals of the grammar. It ignores the non-terminals.

→ It is a kind of "Shift-Reduce" Parsing method.

Eg. i) $E \rightarrow E+T/T$
$T \rightarrow T+F/F$
$F \rightarrow id$

is a operator precedence grammar

ii) $E \rightarrow ET/T$
$T \rightarrow TF/\varepsilon$
$F \rightarrow id$

It is not bcoz first & second production has two non-terminals are adjacent & ε is also there in RHS.

Note:

(i) id is having highest priority compared to any operator

(ii) $ has the least

Steps:

1) Check the given grammar is operator precedence or not
2) Construct the relational table
3) Parse the i/p string
4) Generate the parse tree

Q) $E \rightarrow EAE/id$

$A \rightarrow +/*$ then parse string: id+id*id

Sol) the given grammar is not operator precedence let us convert it into OP Parser.

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

Operator Precedence table

	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

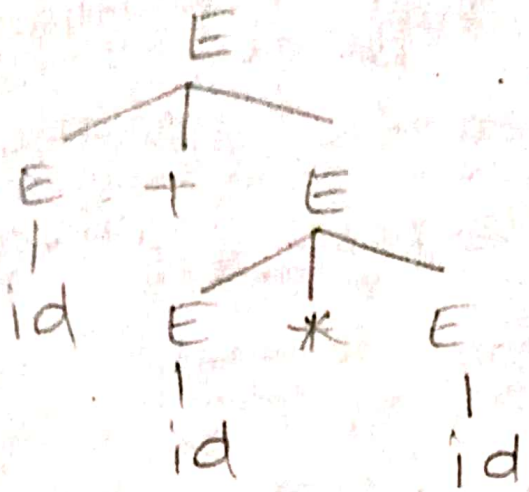
Step 3:-

Parse the String

given String = id+id*id

Stack	Relation	i/p string	Action
$	<	id + id * id $	shift
$ id	>	+ id * id $	reduce E→id
$ E	<	+ id * id $	shift
$ E +	<	id * id $	shift
$ E + id	>	* id $	reduce E→id
$ E + E	<	* id $	shift
$ E + E *	<	id $	shift
$ E + E * id	>	$	reduce E→id
$ E + E * E	>	$	reduce E→E
$ E + E	>	$	reduce E→E
$ E		$	Accepted

→ Step 4:- Now construct the parse tree (Using Bottom Up parser)



string is id + id * id

→ LR - Parsing :-

→ It is one type of bottom-up parsing. It is used to parse the large class of grammar.

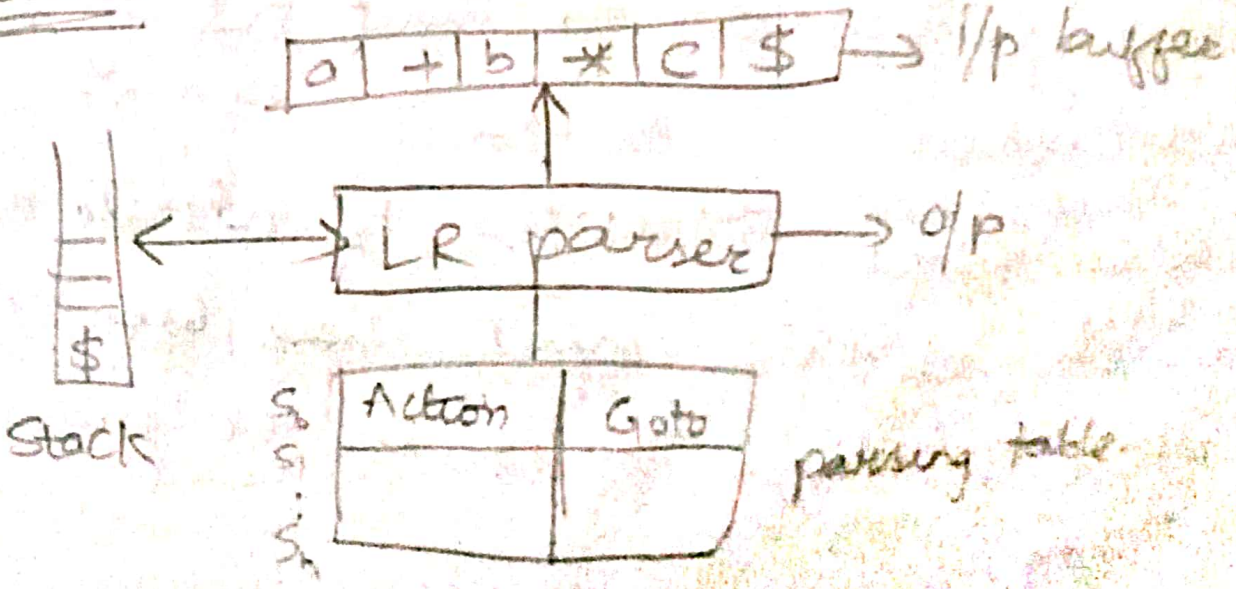
→ It is also called LR(K) parsing where

L → stands for left to right scanning.

R → " " right most derivation

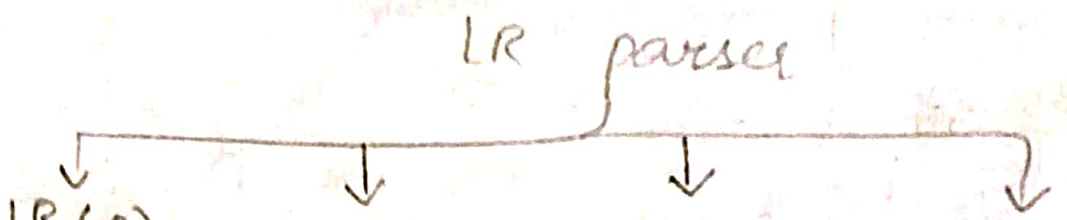
K → no of i/p symbols.

→ Structure :-



→ It consist of i/p buffer for storing the i/p st
a stack for storing the grammar symbols, o/p
a parsing table comprised of two parts mainly
"Action" & "go to".

⇒ Types of LR parsers:- Types of LR parsers are



The relation of these parsers is LR(0)<SLR<LALR<CLR
That means CLR parser is the powerful than LALR and LALR powerful than SLR

→ LR(0) parser:-

→ An LR(0) is a production G with dot at some position on the right side of the production.

→ LR(0) items is useful to indicate that how much of the input has been scanned upto a given point in the process of parsing.

In LR(0) ... we place the reduce node in the entire row.

Steps to solve -

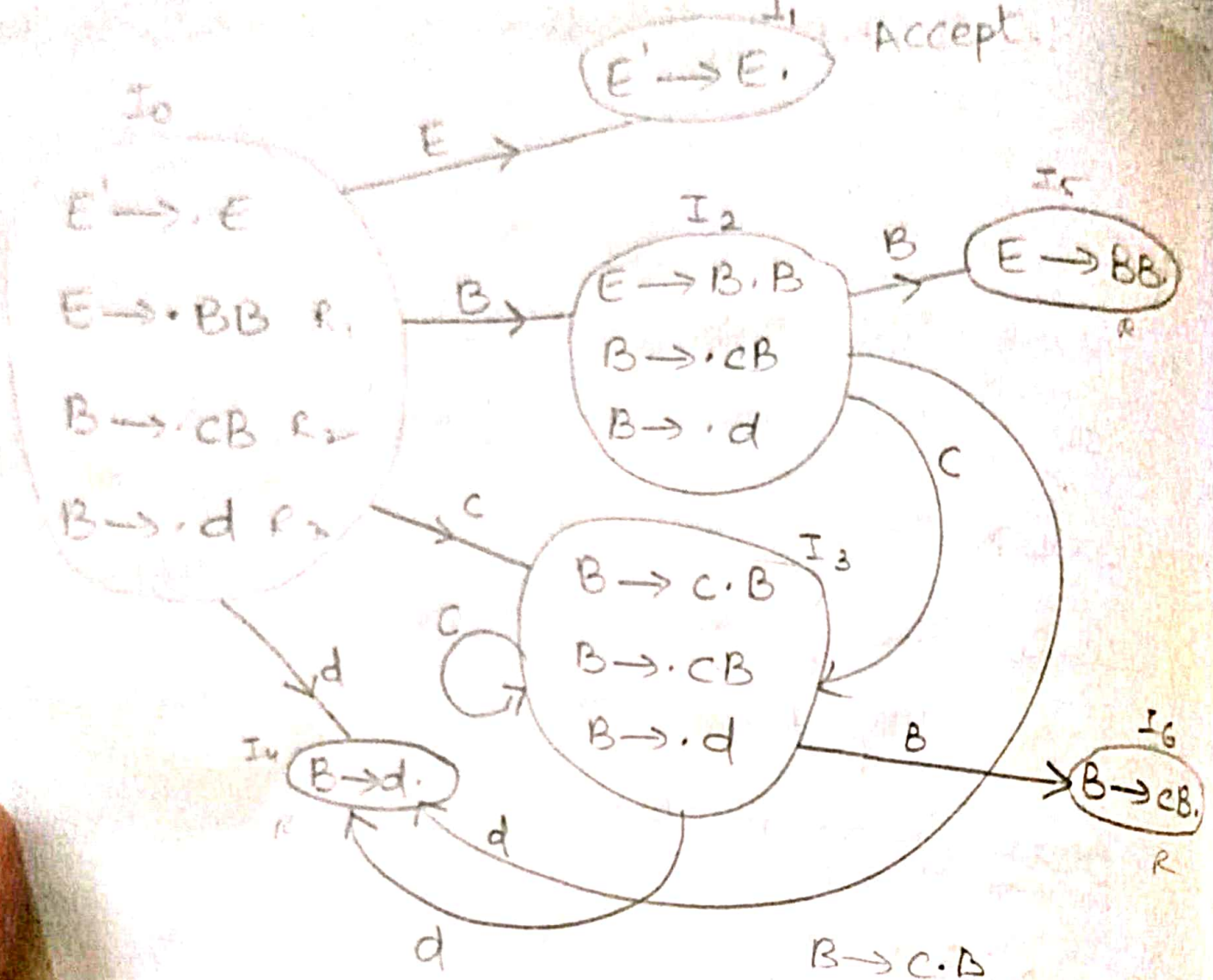
1) Augment the given grammar.
2) Draw the canonical collection of LR(0)
3) Number the production
4) Create the parsing table
5) Code implementation
6) Draw parse tree.

Q) E → BB
B → cB / d

Sol) 1) Augment the grammar

E' → .E
E → .BB
B → .cB
B → .d

2) draw canonical collection of LR(0)



I_0 $E' \to .E$, $E \to .BB$ $B \to .CB$, $B \to .d$

goto(I_0, E): $E' \to E.$ I_1

goto (I_0, B): $E \to B.B$
$B \to .CB$ I_2
$B \to .d$

goto (I_0, C): $B \to C.B$
$B \to .CB$ I_3
$B \to .d$

$B \to C.B$
$B \to .CB$
$B \to .d$

goto(I_0, d): $B \to d.$ I_4

goto(I_2, B): $E \to BB.$ I_5

goto(I_2, C): $B \to C.B$
$B \to .CB$ I_3
$B \to .d$

goto(I_2, d): $B \to d.$ I_4

goto(I_3, B): $B \to CB.$ I_6

goto(I_3, C): $B \to C.B$ I_3
$B \to .CB$

(I_3, d) : $B \to d \cdot$ I_4

Step 4:- number the production

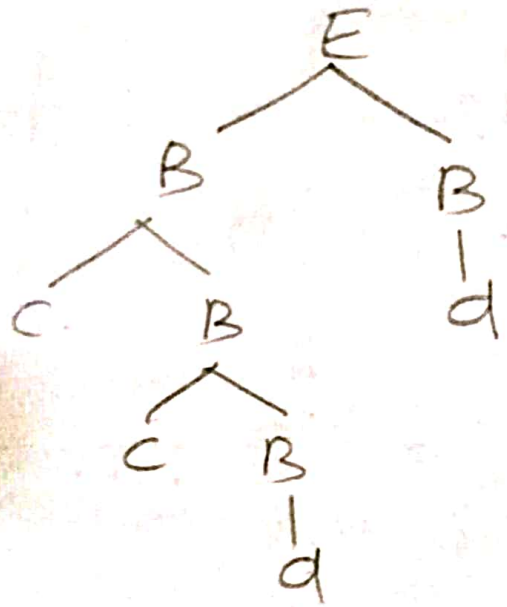
State	Action			GOTO	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	R_3	R_3	R_3		
I_5	R_1	R_1	R_1		
I_6	R_2	R_2	R_2		

Step 5 Stack implement- i/p : ccdd $

Stack	Input	Action
$0	ccdd$	shift C → s3
$0c3	cdd$	shift C → s3
$0c3c3	dd$	shift d → s4
$0c3c3d4	d$	reduce B → d
$0c3c3B6	d$	reduce B → cB
$0c3B6	d$	reduce B → cB
$0B2	d$	shift d → s4

$0B2d4	$	reduce B→d
$0B2B5	$	reduce E→BB
$0E1	$	Accept

Step6:- Parse tree



Q) E → TT

T → aT/b

Sol Augment the given grammar

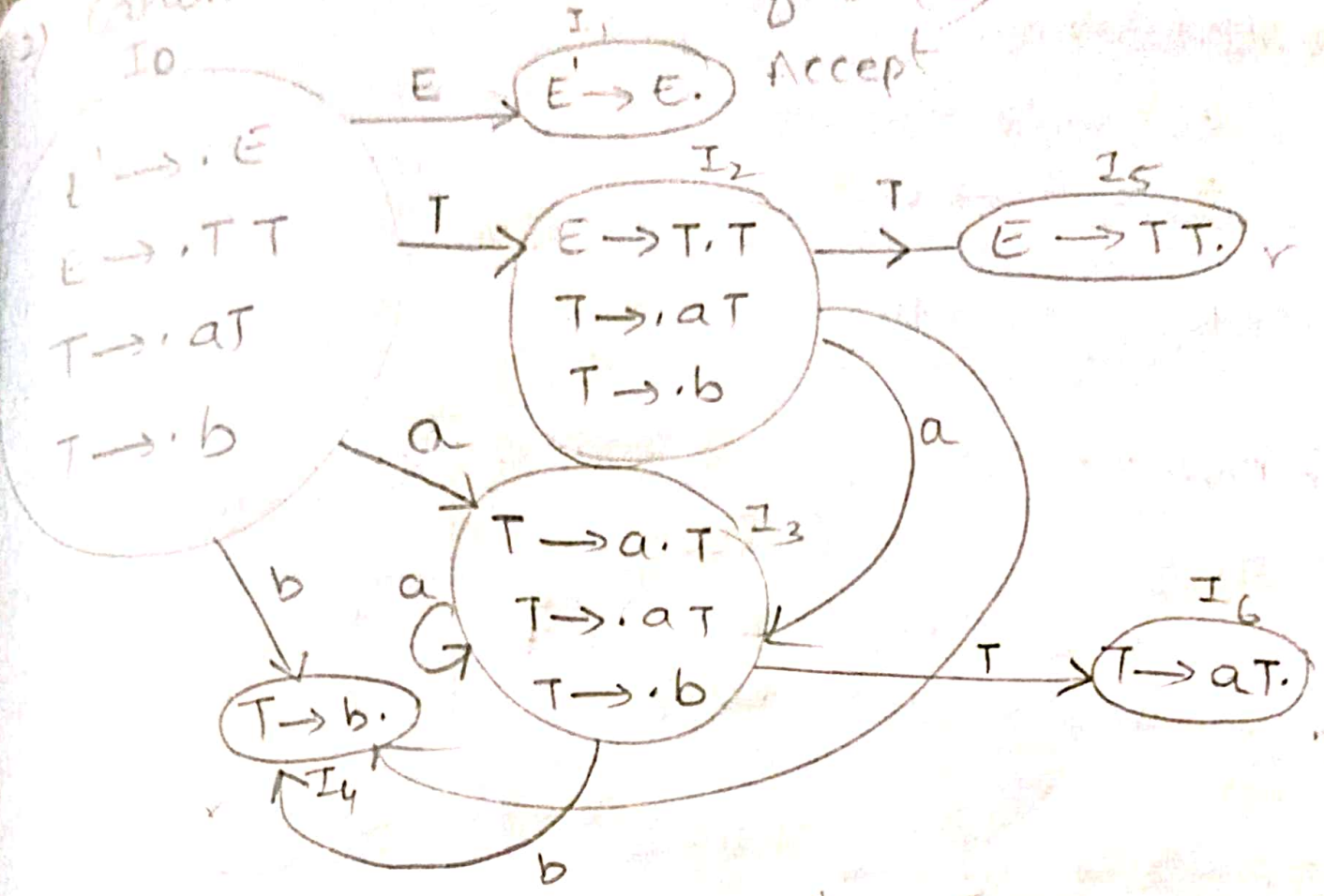
E' → .E

E → .TT R1

T → .aT R2

T → .b R3

Canonical collection of LR(0)



goto (I_0, E): $E' \to E$. I_1

goto (I_0, T): $E \to T \cdot T$
$T \to \cdot aT$ I_2
$T \to \cdot b$

goto (I_0, a): $T \to a \cdot T$
$T \to \cdot aT$ I_3
$T \to \cdot b$

goto (I_0, b): $T \to b$. I_4

goto (I_2, T): $E \to TT$. I_5

goto (I_2, a): $T \to a \cdot T$
$T \to \cdot aT$ I_3
$T \to \cdot b$

goto (I_2, b): $T \to b$. I_4

goto (I_3, T): $T \to aT$. I_6

goto (I_3, a): $T \to a \cdot T$
$T \to \cdot aT$ I_3
$T \to \cdot b$

goto (I_3, b): $T \to b$. I_4

3) number the productions

R_1 : $E \rightarrow TT$

R_2 : $T \rightarrow aT$

R_3 : $T \rightarrow b$

4) number the production action

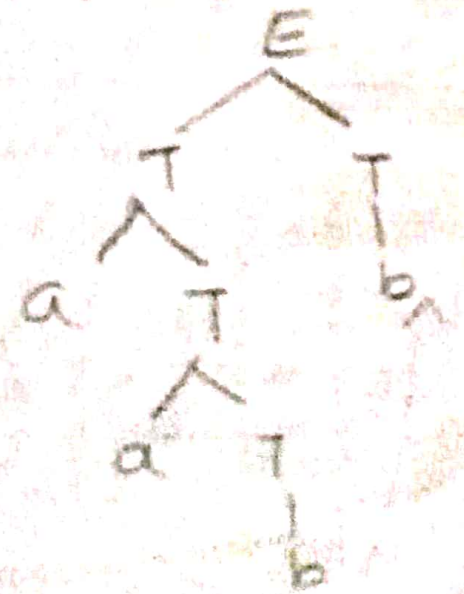
State	Action			Go to	
	a	b	$	E	T
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	R_3	R_3	R_3		
I_5	R_1	R_1	R_1		
I_6	R_2	R_2	R_2		

5) Stack implementation

String : aabb

Stack	Input	Action
$0	aabb$	shift a → S_3
$0a3	aabb$	shift a → S_3
$0a3a3	bb$	shift b → S_4
$0a3a3b4	b$	reduce T → b
$0a3a3T6	b$	reduce T → aT
$0a3T6	b$	reduce T → aT
$0T2	b$	shift b → S_4
$0T2b4	$	reduce T → b
$0T2T5	$	reduce E → TT
$0E1	$	Accept

6) Parse tree



string aabb is accepted

9) $S \rightarrow SS/a$. draw goto graph. Find the conflicts if any. LR(0)

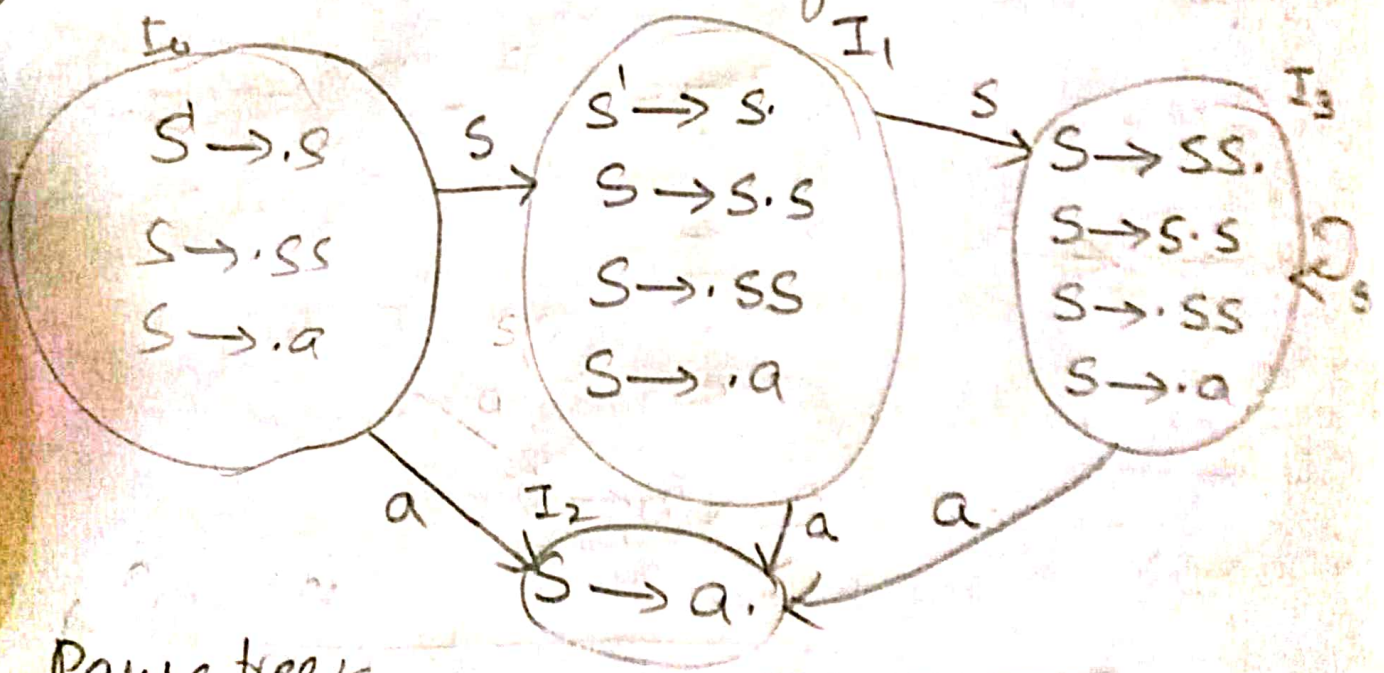
Sol) Augment

$S' \rightarrow S$

$S \rightarrow .SS$ R_1

$S \rightarrow .a$ R_2

2) Canonical collection of LR(0) items



Parse tree:-

	Action		Goto
	a	$	S
I_0	S_2		1
I_1	S_2	Accept	3
I_2	R_2	R_2	
I_3	S_2, R_1	R_1	3

↓ SR conflict

→ conflict occurs. (Shift - Reduce Conflict)

Conflict: It has multiple entries in action [3,a] = s_2 & r_3. That means shift & reduce occurs at i/p symbol a.

⇒ Augmented Grammar:- If a grammar has starting symbol S then augment grammar is S' → .S. The purpose of it is to show acceptance of i/p.

⇒ Closure:- It means place a dot (.) on immediate left of RHS production.

⇒ Goto:- It means move dot to RHS of the variable.

SLR (Simple LR):-

Simple left to right and

→ Goto :- It means move dot to on RHS of the variable.

SLR (Simple LR) :-

→ SLR stands for "Simple left to right and right most derivation" parser.

→ It works on smallest class of grammar.

→ It requires few no. of states

→ Simple & fast to construct.

→ The only diff in LR(0) & SLR is in SLR parsing table, whenever any state is having final state dont place reduce move in entire row, just place on "FOLLOW" of production rule.

Q) $E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow id$

Sol Augment

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E+T \quad R_1$

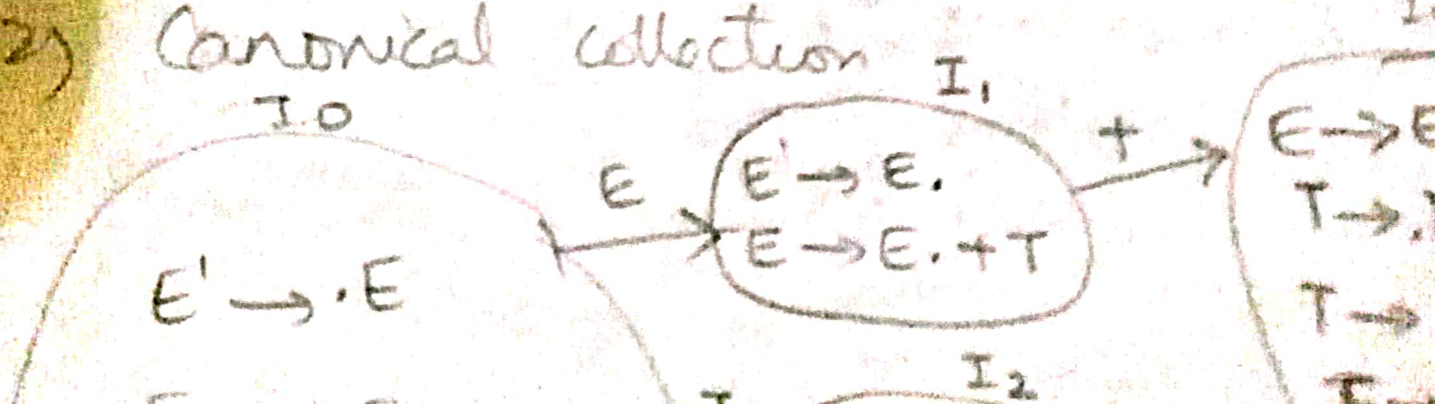
$E \rightarrow \cdot T \quad R_2$

$T \rightarrow \cdot T*F \quad R_3$

$T \rightarrow \cdot F \quad R_4$

$F \rightarrow \cdot id \quad R_5$

2) Canonical collection



	Action				Goto		
	id	+	*	$	E	E	T
I_0	S_4				1	3	2
I_1		S_5		Accept			
I_2		R_2	S_6	R_2			
I_3		R_4	R_4	R_4			
I_4		R_5	R_5	R_5			
I_5						3	7
I_6	S_4					8	
I_7	S_4						
I_8		R_1	S_6	R_1			
		R_3	R_3	R_3			

$T \to T * F.$ R_3

$E \to E + T.$ R_1

$E \to T.$ R_2

$Fo(E) \{\$, +\}$

$T \to F.$ R_4

$F \to id.$ R_5

$Fo(E') = \{\$, +\}$

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

$Fo(F) = \{*, \$, +\}$

5) Stack implementation

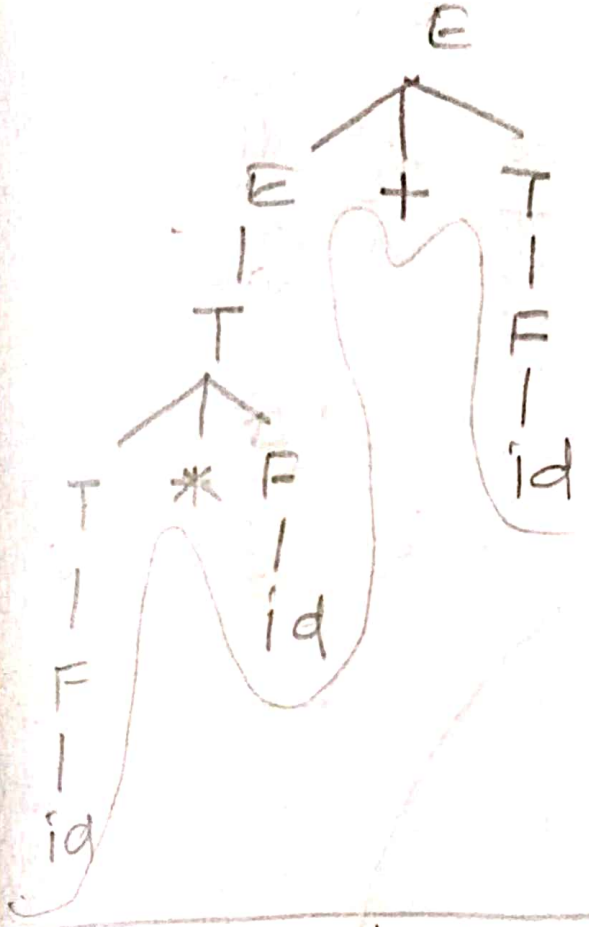
String id * id + id,

I_8

$T \to T * F.$

Stack	input	Action
$0	id * id + id $	shift id → s_4
$0 id 4	* id + id $	reduce F → id
$0 F 3	* id + id $	reduce T → F
$0 T 2	* id + id $	shift * → s_6
$0 T 2 * 6	id + id $	shift id → s_4
$0 T 2 * 6 id 4	+ id $	reduce F → id
$0 T 2 * 6 F 8	+ id $	reduce T → T * F
$0 T 2	+ id $	reduce E → T
$0 E 1	+ id $	Shift + → s_5
$0 E 1 + 5	id $	shift id → s_4
$0 E 1 + 5 id 4	$	reduce F → id
$0 E 1 + 5 F 3	$	reduce T → F
$0 E 1 + 5 T 7	$	reduce E → E + T
$0 E 1	$	Accept

6) Parse tree



id * id + id //

Q) Check whether it is SLR(1) or not

$S \to L=R$, $S \to R$, $L \to *R$, $L \to id$, $R \to L$

$S' \to S$

SOL

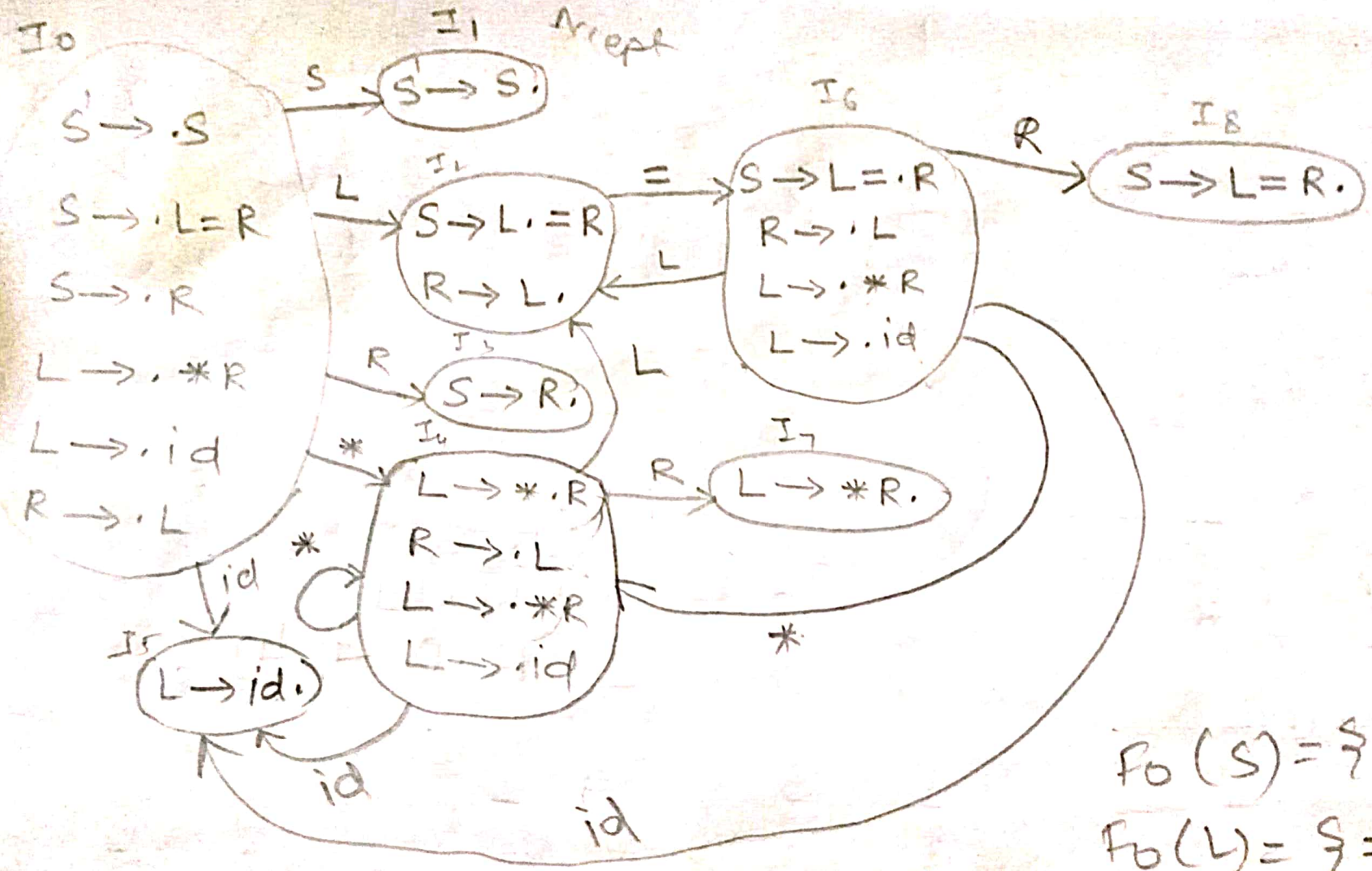
$S \to .L=R$ 1

$S \to .R$ 2

$L \to .*R$ 3

$L \to .id$ 4

$R \to .L$ 5


I0
S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L
I1
S → S.
Accept
I2
S → L.=R
R → L.
I3
S → R.
I4
L → *.R
R → .L
L → .*R
L → .id
I5
L → id.
I6
S → L=.R
R → .L
L → .*R
L → .id
I7
L → *R.
I8
S → L=R.
S
L
R
=
*
id
id
id
*
*

$f_0(S) = \{\$\}$

$f_0(L) = \{=, \$\}$

$f_0(R) = \{\$, =\}$

Parsing table

	Action				Goto		
	=	id	*	$	S	L	R
I_0		S_5	S_4		1	2	3
I_1				Accept.			
I_2	R_5, S_6			R_5			
I_3				R_2			
I_4		S_5	S_4			2	7
I_5	R_4			R_4			
I_6		S_5	S_4			2	8
I_7	R_3			R_3			
I_8				R_1			

It is not SLR bcoz we have multiple entries in Action [2, =] = S_6 and R_5) that means shift and reduce conflict occurs on symbol '='.

Hence it is not SLR grammar.

Q) $S \rightarrow AS$
$S \rightarrow b$
$A \rightarrow SA$
$A \rightarrow a$

(exams).

⇒LALR:-

→ It stands for lookahead LR. To construct LALR(1) parsing table, we use the canonical collection of LR(1) items.

→ In LALR parsing, the LR(1) items which have same productions but different lookahead are combined to form a single set of items.

→ LALR(1) parsing is same as CLR(1) parsing only diff is the parsing table.

LR(1) item = LR(0) item + lookahead.

LALR(1) = put reduce only on lookahead.

⇒Procedure to find lookahead:-

$E \to BB$

$B \to cB \mid d$

Aug

$E' \to .E$

$E \to .BB$

$B \to .cB$

$B \to .d$

$E' \to .E, \$$

$E \to .BB, \$$

$B \to .cB, c/d$

$B \to .d, c/d$

for 1st prod, lookahead is $

First(B,$) = First(B)

Q) Construct LALR parse for

$S \to CC$

$C \to aC/b$ string = bb

Augment grammar

Sol)

$S' \to S$.

$S \to CC$ 1

$C \to aC$ 2

$C \to b$ 3

Step 1 calculation of LR(1) items

I_0: $S' \to .S, \$$

$S \to .CC, \$$

$C \to .aC, a/b$

$C \to .b, a/b$

goto(I_0, S)

I_1: $S' \to S., \$$

goto(I_0, C)

I_2: $S \to C.C, \$$

$C \to .aC, \$$

$C \to .b, \$$

goto(I_0, a)

I_3: $C \to a.C, a/b$

$C \to .aC, a/b$

$C \to .b, a/b$

goto(I_0, b)

I_4: $C \to b., a/b$

goto(I_2, C)

I_5: $S \to CC., \$$

goto(I_2, a)

I_6: $C \to a.C, \$$

$C \to .aC, \$$

$C \to .b, \$$

goto(I_2, b)

I_7: $C \to b., \$$

goto(I_3, C)

I_8: $C \to aC., a/b$

goto(I_6, C)

I_9: $C \to aC., \$$

We can combine I_3 & $I_6 \rightarrow I_{36}$ (same productions diff lookahead)

I_{36} : $C \rightarrow a.C, a/b/\$$

$C \rightarrow .aC, a/b/\$$

$C \rightarrow .b, a/b/\$$

Illy I_4 & $I_7 = I_{47}$

I_{47} : $C \rightarrow b., a/b/\$$

Illy I_8 & $I_9 = I_{89}$

I_{89} : $C \rightarrow aC., a/b/\$$

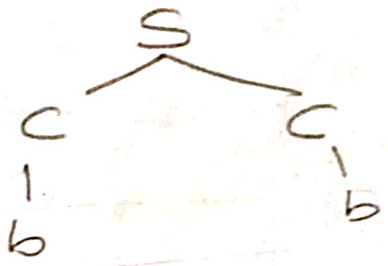
⇒ Parsing table

	Action			S	C
	a	b	$		
I_0	S_{36}	S_{47}		1	2
I_1			Accepted		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

stack implementation

bb

stack	input	action
10	bb$	shift b→s4
$0b4	b$	reduce c→b
$0c2	b$	shift b→s4
$0c2b4	$	reduce c→b
$0c2c5	$	reduce s→cc
$0s1	$	Accepted.



→ CLR(1) Parsing :-

→ It refers to canonical lookahead. CLR parsing uses the collection of LR(1) items to build the CLR(1) parsing table

→ CLR(1) parsing table produces the more no. of states as compared to SLR(1) parsing.

→ In CLR(1), we place the reduce node only in the lookahead symbols.

→ steps involved:-

(1) for given input string, write a CFG
(2) Check the ambiguity of the grammar.
(3) Add augment production in the given grammar
(4) Create canonical collection of LR(0) items
(5) Create a data flow diagram (DFA)
(6) Construct a CLR(1) parsing table.

LR(1) item = LR(0) item + lookahead.

Q Construct CLR

$S \to CC$ 1
$C \to aC$ 2
$C \to a$ 3

parse dd $

Sol Augmented grammar.

$S' \to S$
$S \to CC$
$C \to aC$
$C \to d$

LR(1) items :-

I0: $S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a/d$
$C \to .d, a/d$

goto(I_0, S)

I_1: $S' \to S.$, \$

goto(I_0, C)

I_2: $S \to C.C$, \$

$C \to .aC$, \$

$C \to .d$, \$

goto(I_0, a)

I_3: $C \to a.C$, a/d

$C \to .aC$, a/d

$C \to .d$, a/d

goto(I_0, d)

I_4: $C \to d.$, a/d

goto(I_2, C)

I_5: $S \to CC.$, \$

goto(I_2, a)

I_6: $C \to a.C$, \$

$C \to .aC$, \$

$C \to .d$, \$

go(I_2, d)

I_7: $C \to d.$, \$

goto(I_3, C)

I_8: $C \to aC.$, a/d

goto(I_6, C)

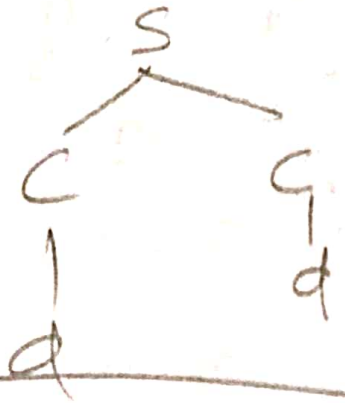
I_9: $C \to aC.$, \$

State	Action			goto	
	a	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	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

⇒ Stack implementation

String dd $

Stack	Input	Action
	dd$	
$0	dd$	shift d → $_4$
$0d4	d$	reduce C → d
$0C2	d$	shift d → $_7$
$0C2d7	$	reduce C → d
$0C2C5	$	reduce S → CC
$0S1	$	Accept



⇒ Parser Generator:-

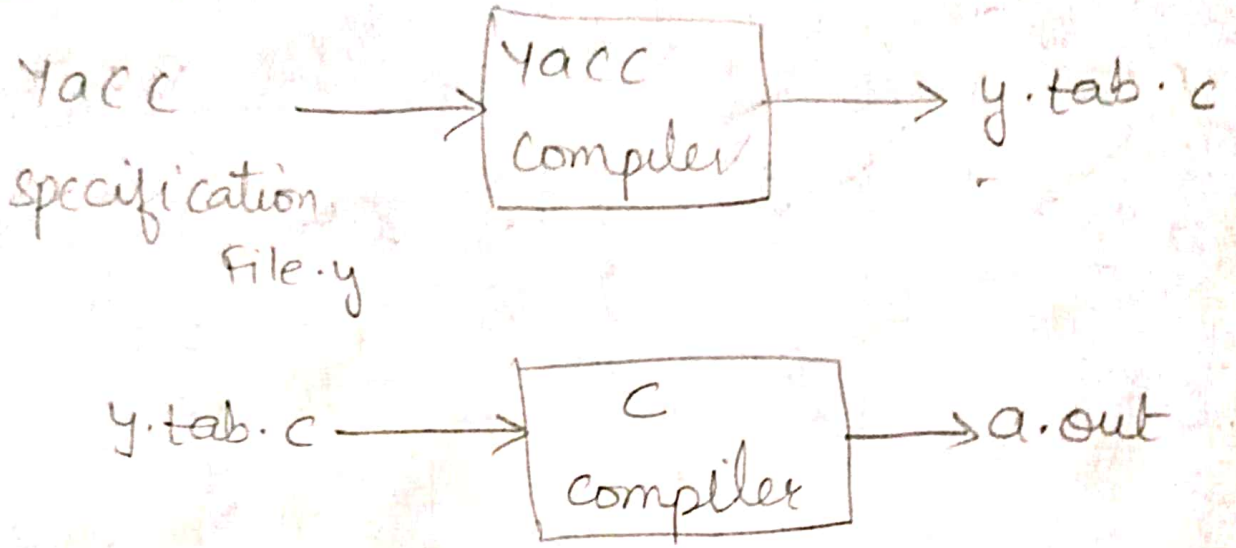
→ Various tools are used for parser generators to describe syntax of given expression.

→ A tool called yacc is used for construction of parser generators. yacc stands for Yet Another Compiler - Compiler

→ It is developed by Stephen C. Johnson

→ It accepts tokens as i/p and produces parse tree as o/p.

⇒ Working :-



Syntax :-

definitions

%%

translation rules

%%

Auxiliary functions.

Q.114) pg 5.5

i) Elimination of Global Common Sub expression

Common exp is an exp which appears repeatedly in the program, which is computed previously but the values of variables in expression haven't changed. this technique replaces the redundant expression each time it is encounted.

unoptimized	optimized
a = b+c	a = b+c
b = a-d	b = a-d
c = b+c	c = b+c
d = a-d	d = b

b) Constant Folding - def 5.6

i) int maximum = 10;
r = maximum / 2

ii) Area = 22/7 * r * r

Here 22/7 is evaluated & 3.14 is replaced

Area = 3.14 * r * r

c) Dead-Code Elimination :- It includes eliminating those statements which are never executed or if executed then its output is never used.

eg:

Unoptimized

c = a * b
r = b ; dead state
d = a * b + 4

Optimized

c = a * b
d = a * b + 4

d) Copy / Variable Propogation :-

x := y then

Unoptimized

a := b + c
d := c
⋮
e : b + d - 3

Optimized

a : b + c
d := c
⋮
e : b + c - 3

⇒ **Loop :-** (p. 5.6 Q15)

1) **Code motion :-** It is used to decrease the amount of code in loop. This transformation takes a statement or expression which can be moved outside the loop without affecting the semantics of the program.

Unoptimized

```
a=100;
while (a>0)
{
    x=y+z;
    if (a%x==0)
    printf("%d", a);
}
```

Optimized

```
a=100;
x=y+z;
while (a>0)
{
    if (a%x==0)
    printf("%d", a);
}
```

2) **Elimination of induction variable :-**

40C

```
int a[10], b[10];
void fun(void)
{ int i, j, k;
  for(i=0; j=0; k=0; i<10; i++)
  a[j++] = b[k++]
  return;
}
```

OC

```
int a[10], b[10];
  void fun(void)
  { int i;
for(i=0; i<10; i++)
  a[i] = b[i]
   return;
  }
```

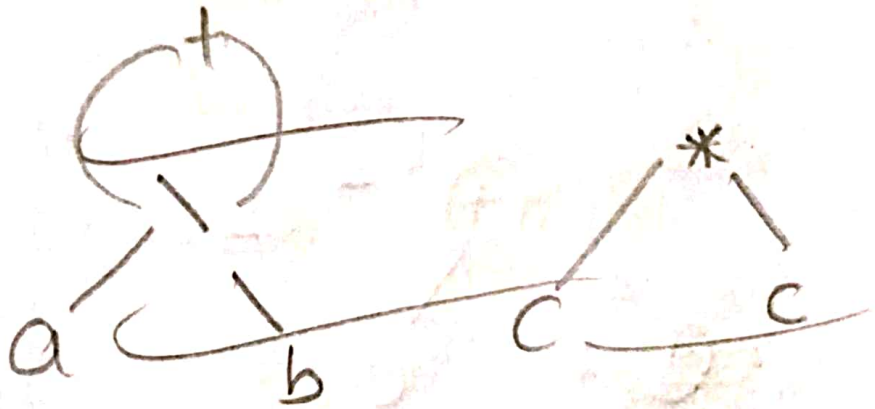
Strength Reduction

$a^2 = a * a$

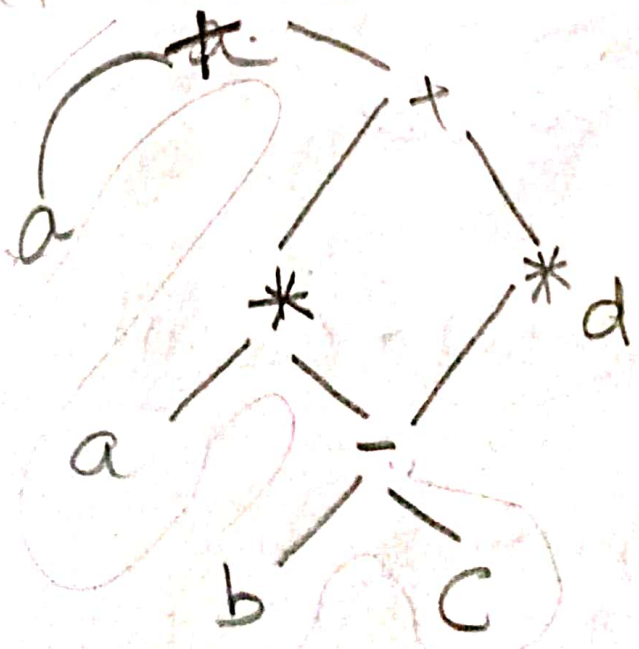
$x * 2 =$ left shift

$x / 2 =$ right shift

$(a/b) + (a/b) * (c * cd)$



$a + a * (b - c) + (b - c) * d.$



1) $(a+b) * (a+b+c)$

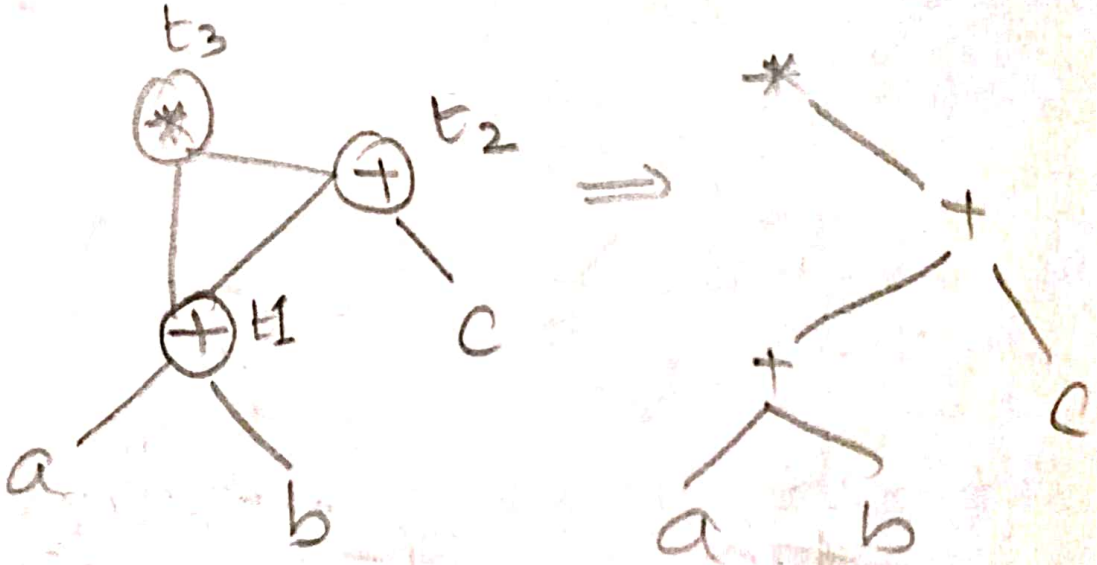
Three address code

$t_1 = a+b$

$t_2 = t_1 + c$

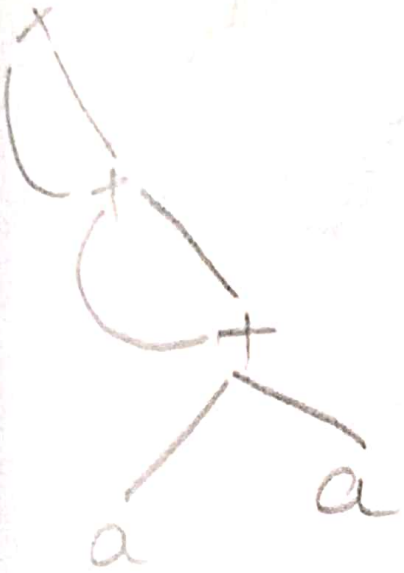
$t_3 = t_1 * t_2$

DAG



2) $((a+a)+(a+a))+((a+a)+(a+a))$

DAG for this expression is -

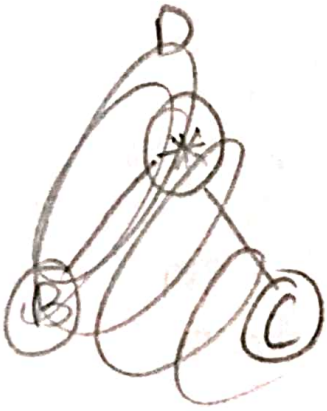


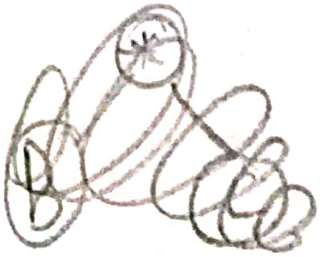
D := B * C

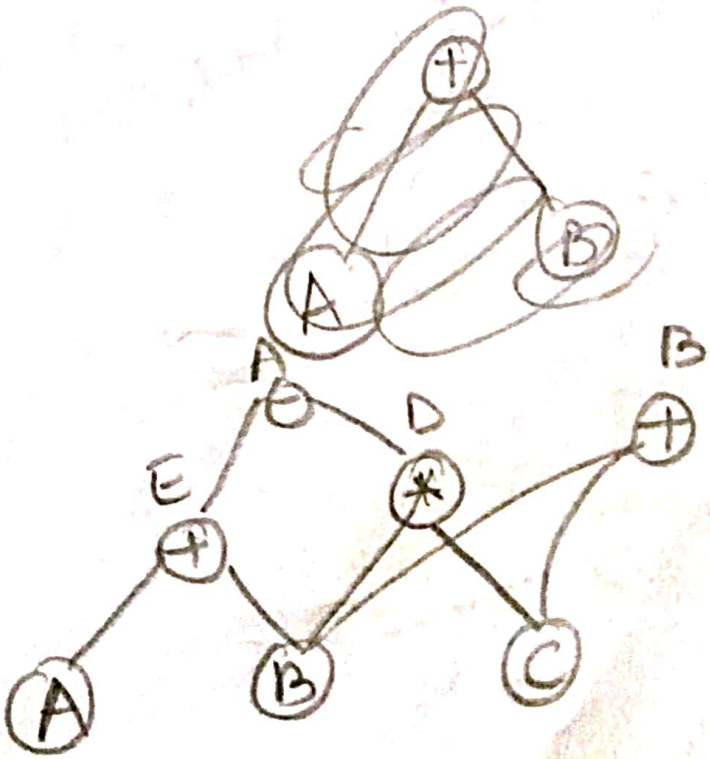
E := A + B

B := B + C

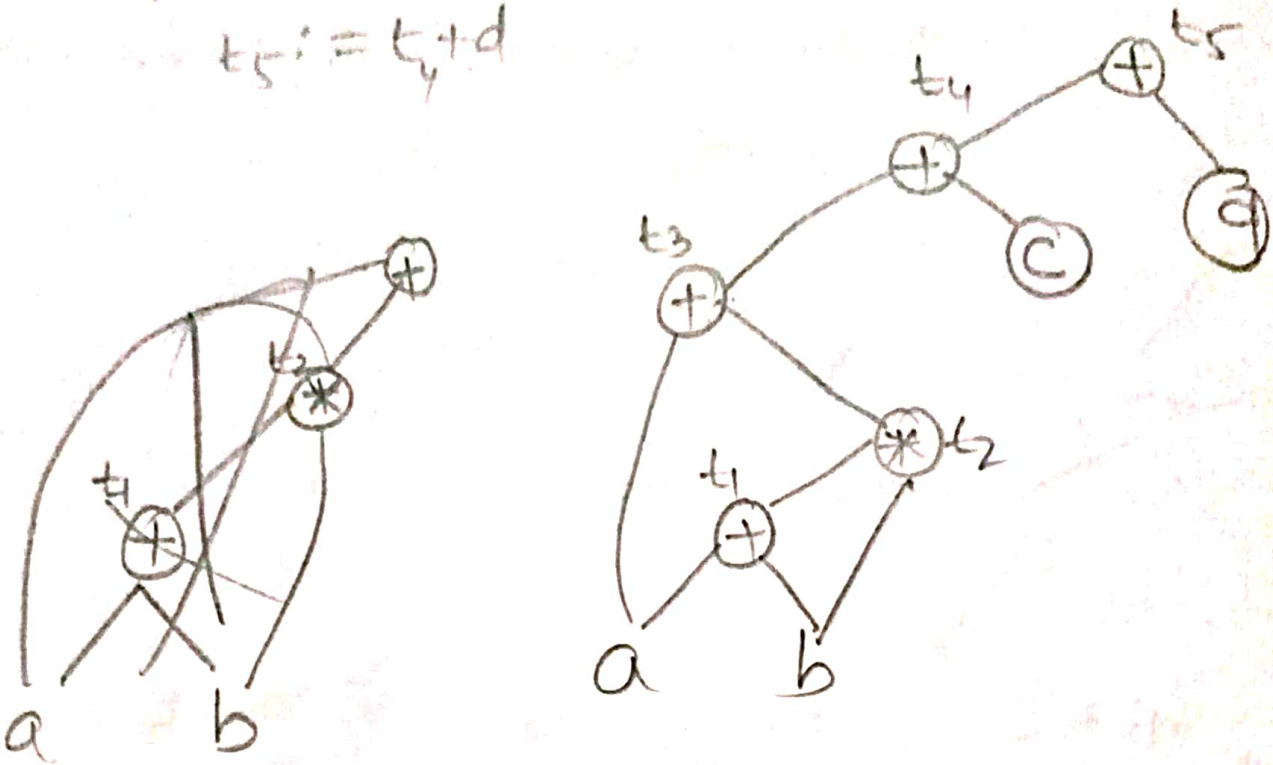
A := E - D



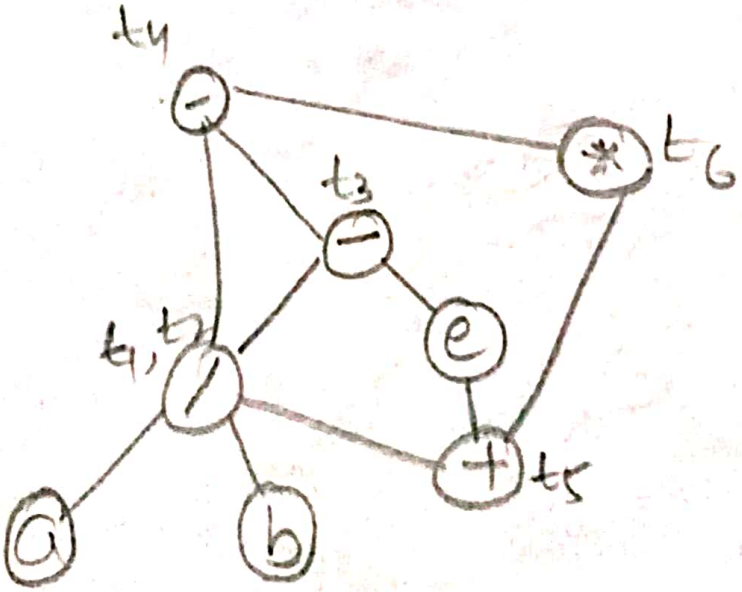




$t_1 := a+b$, $t_2 := b * t_1$, $t_3 := a+t_2$, $t_4 := t_3+c$,
$t_5 := t_4+d$

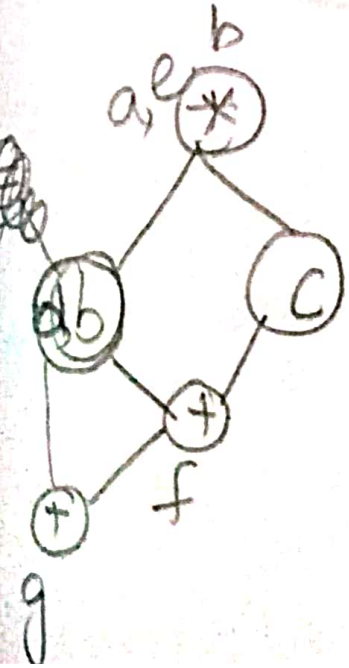


$t_1 := a/b$, $t_2 := a/b$, $t_3 := e-t_2$, $t_4 := t_1-t_3$
$t_5 := e+t_2$ $t_6 := t_4 * t_5$



t_1: c+a, t_2: x[t_1], t_3: i+a, t_4=y[t_3],
t_5 = b_1+t_4, t_6 = b* t_5 , b= t_6, t_7 = a*3, a=t_7,
10) if a<10 goto (1)

a=b* c, d=b, e=d* c, b=e, f=b+c, g=f+d



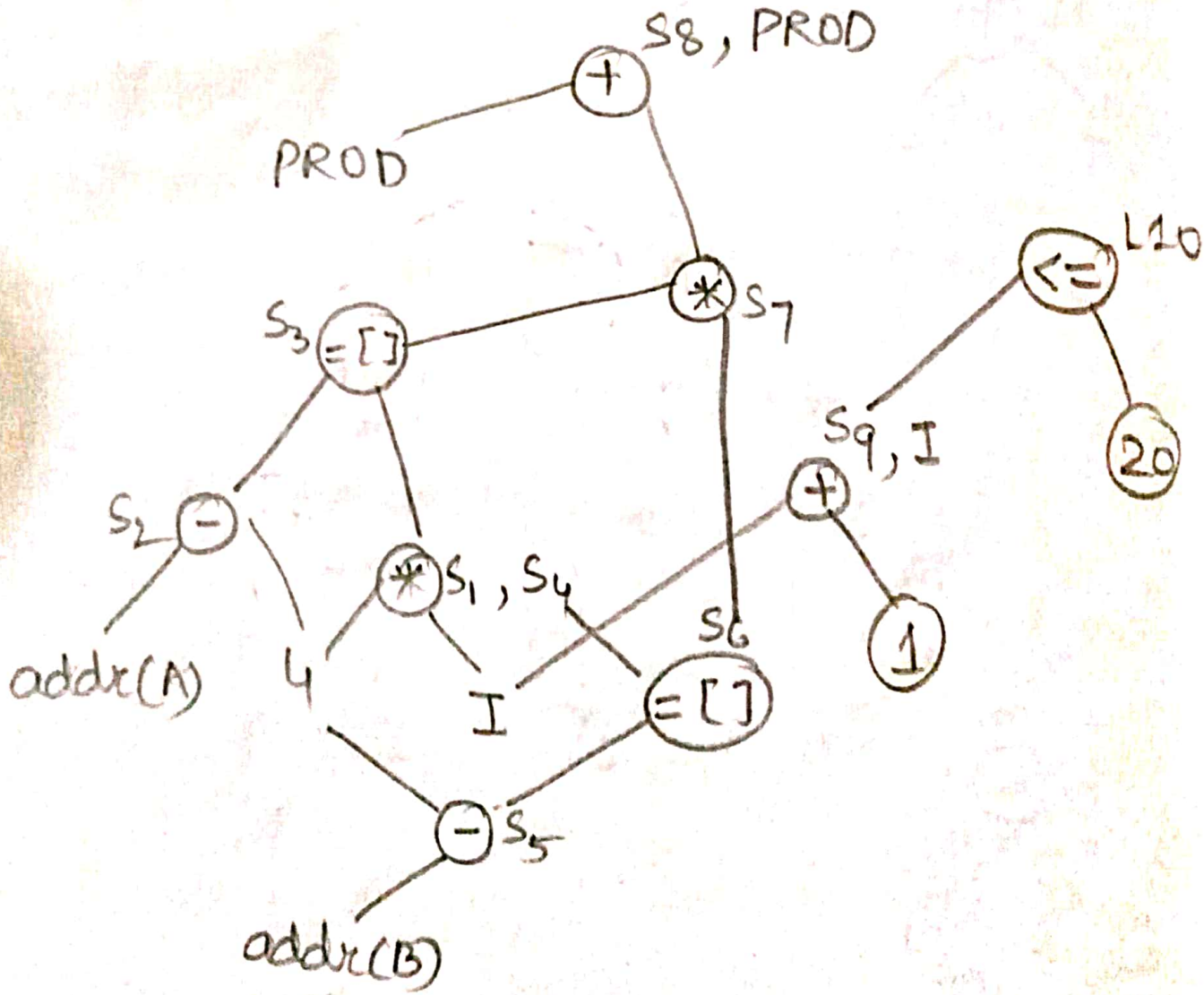
$S_1 = 4 * I$, $S_2 = addr(A) - 4$, $S_3 = S_2[S_1]$,

$S_4 = 4 * I$, $S_5 = addr(B) - 4$, $S_6 = S_5[S_4]$,

$S_7 = S_3 * S_6$, $S_8 = Prod + S_7$, $Prod = S_8$, $S_9 = I + 1$,

$I = S_9$

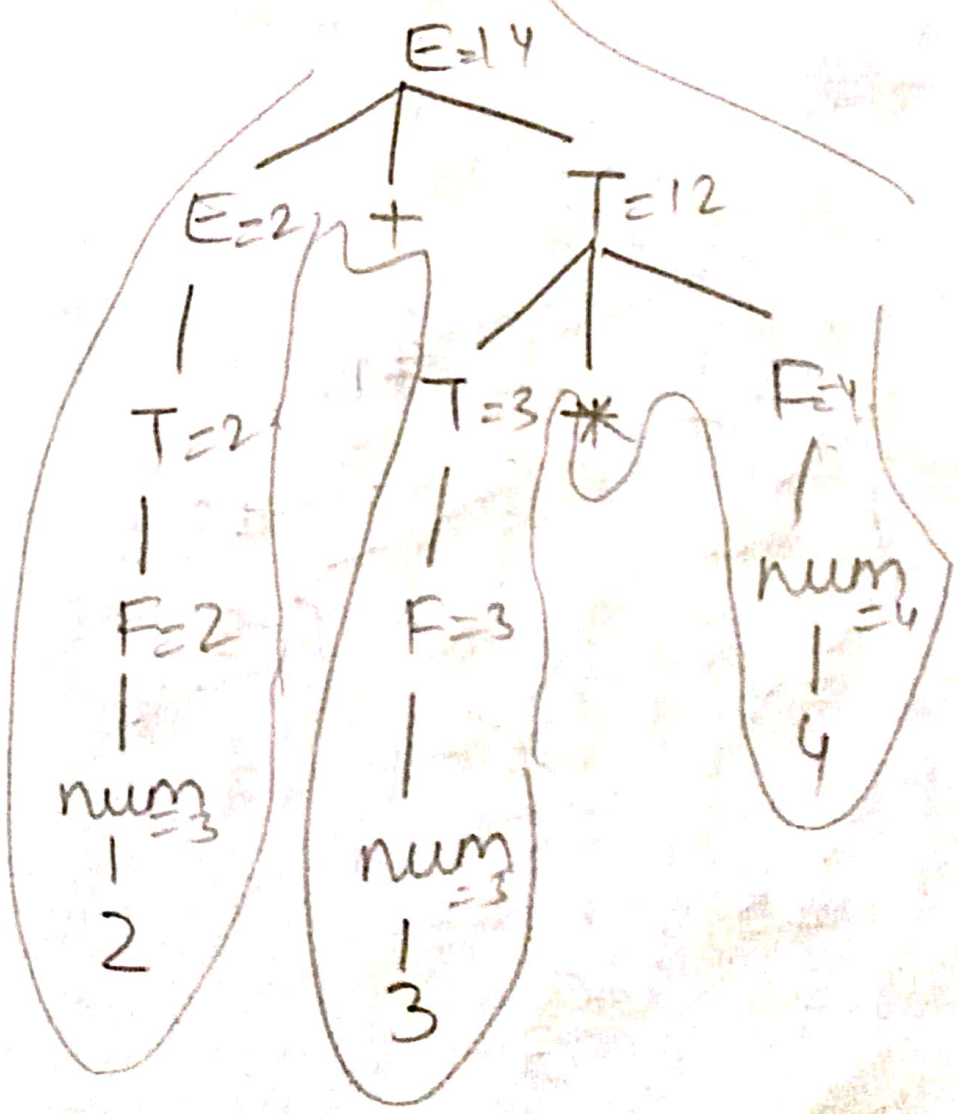
if $I = 20$ goto L10



SDT

$E \to E+T$ / ~~E~~	$E.val = E.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T*F$ / ~~E~~	$T.val = T.val * F.v$
$T \to F$	$T.val = F.$
$F \to num.$	$F.val = lex.val$

Let exp be $2+3*4$



infix to postfix

E → E+T (print '+') #1

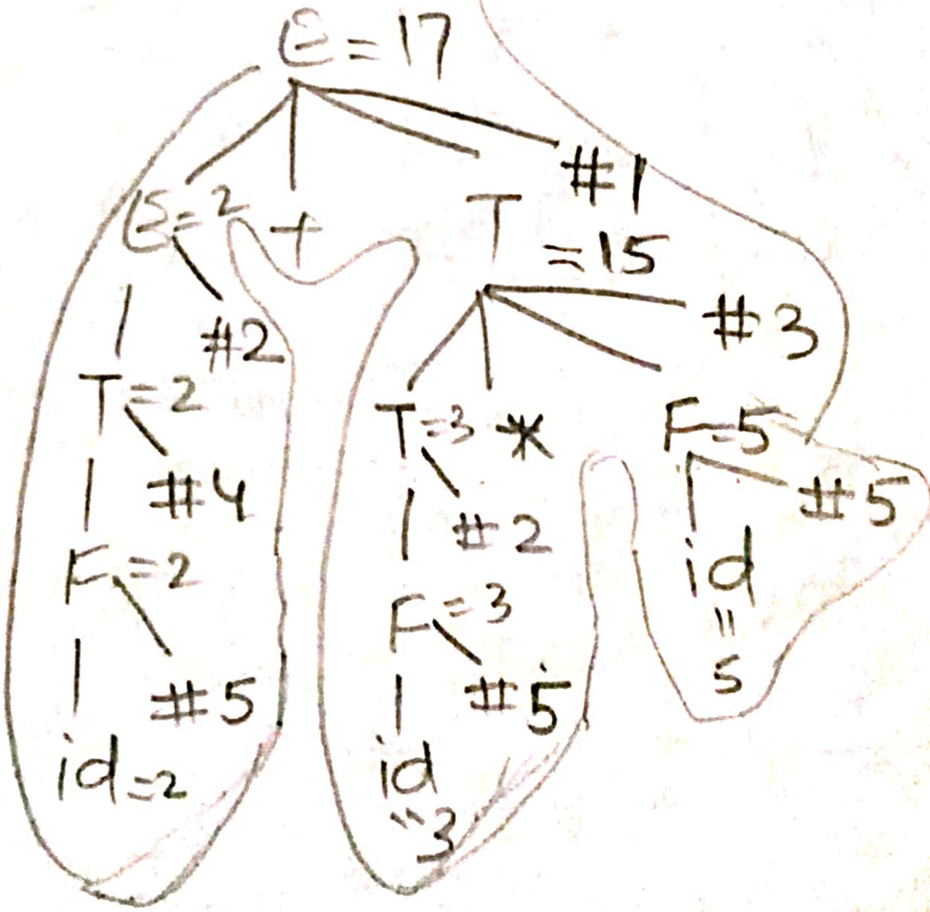
E → T #2

T → T*F (print '*') #3

T → F #4

F → id (print 'id') #5

Sol 2+3*5



o/p:- 235*+

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid a \mid b$

$a+b*a+b$.

LMD

$E \rightarrow \underline{E}+E$

$E \rightarrow \underline{E}+E+E$

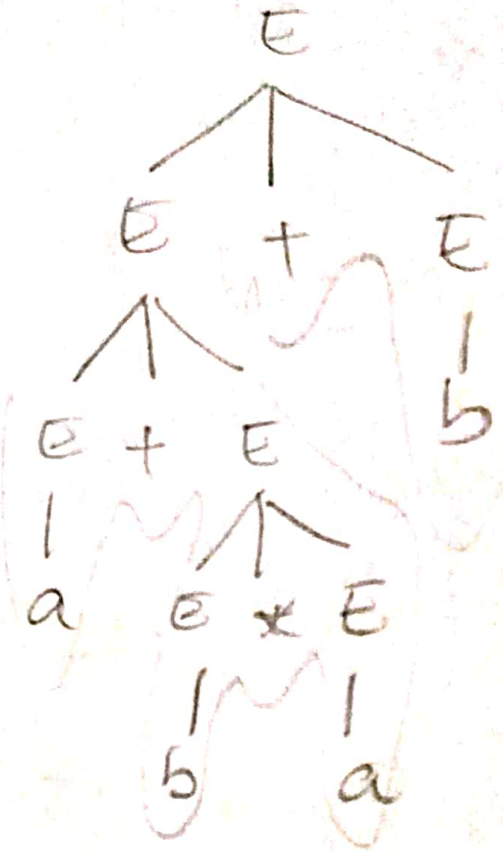
$E \rightarrow a+\underline{E}+E$

$E \rightarrow a+\underline{E}*E+E$

$E \rightarrow a+b*\underline{E}+E$

$E \rightarrow a+b*a+\underline{E}$

$E \rightarrow a+b*a+b$



yield = $a+b*a+b$

RMD

$E \rightarrow E*\underline{E}$

$E \rightarrow E*E+\underline{E}$

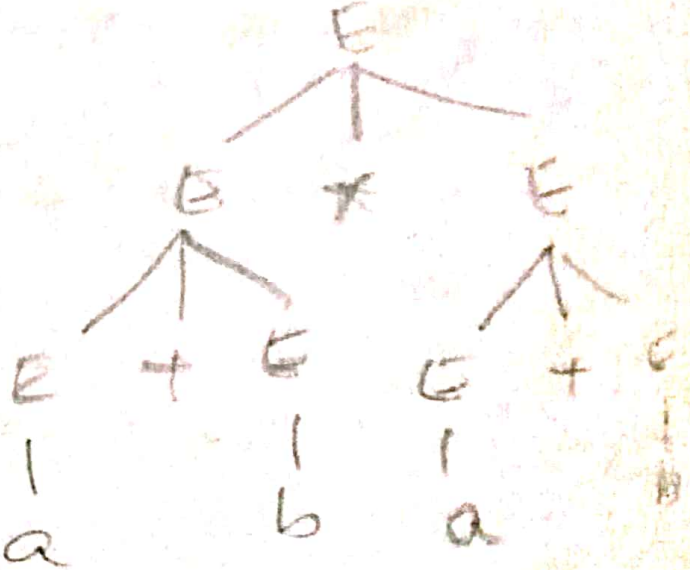
$E \rightarrow E*\underline{E}+b$

$E \rightarrow \underline{E}*a+b$

$E \rightarrow E+\underline{E}*a+b$

$E \rightarrow \underline{E}+b*a+b$

$E \rightarrow a+b*a+b$



$S \to aSbS$

$S \to bSaS$

$S \to \varepsilon$

string = abab

LMD

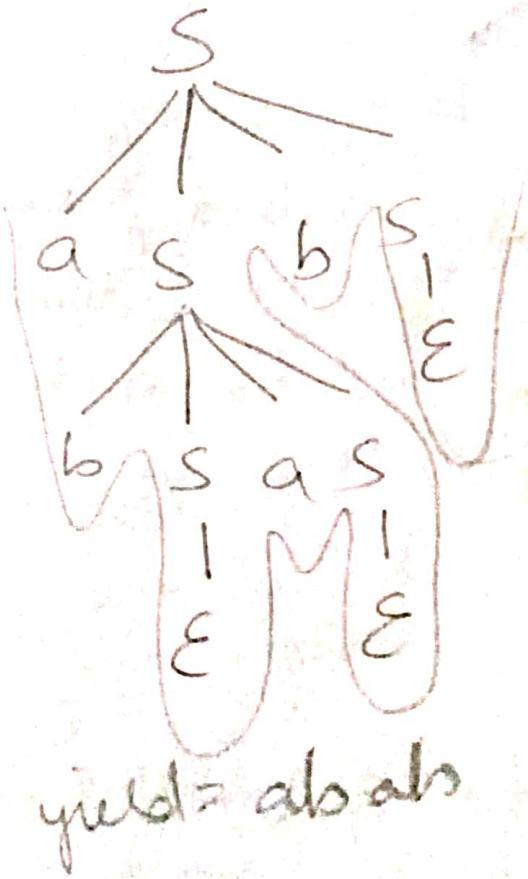
$S \to a\underline{S}bS$

$S \to ab\underline{S}aSbS$

$S \to abaSbS$

$S \to ababS$

$S \to abab$



yield = abab

RMD

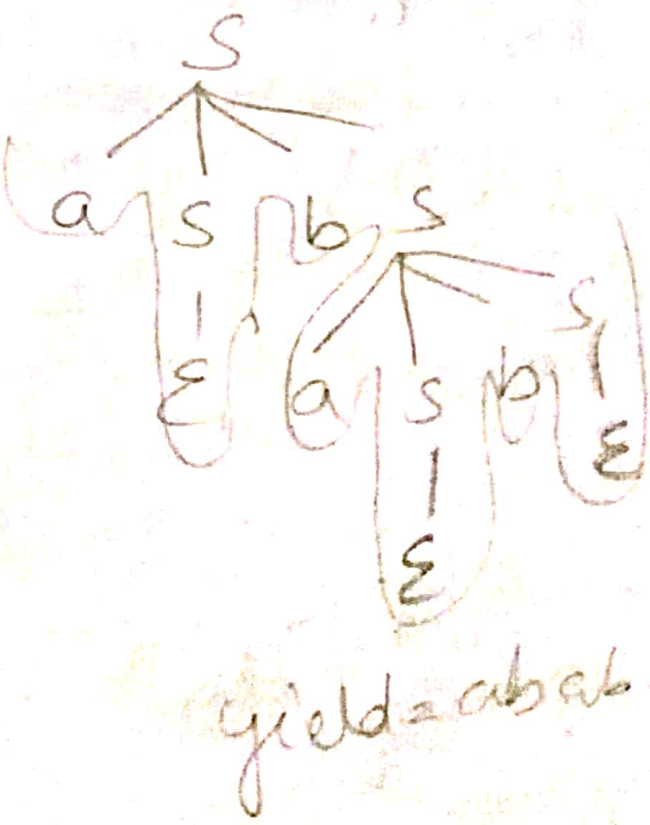
$S \to a\underline{S}bS$

$S \to a\varepsilon bS$

$S \to abaSbS$

$S \to ababS$

$S \to ababa$



yield = abab

$E \rightarrow E + T / T$

$E \rightarrow \underset{A}{E} \underset{\alpha}{+} \underset{\beta}{T}$

$E \rightarrow TE'$

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

$A \rightarrow A\alpha / \beta$

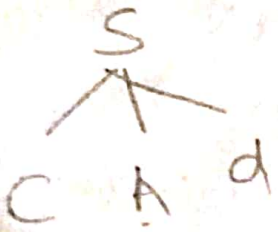
$A \rightarrow \beta A'$

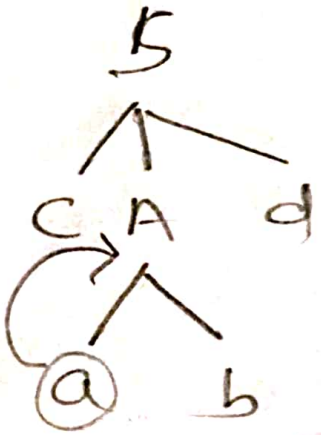
$A' \rightarrow \alpha A' / \varepsilon$

$S \rightarrow cAd$

$A \rightarrow ab / ad$

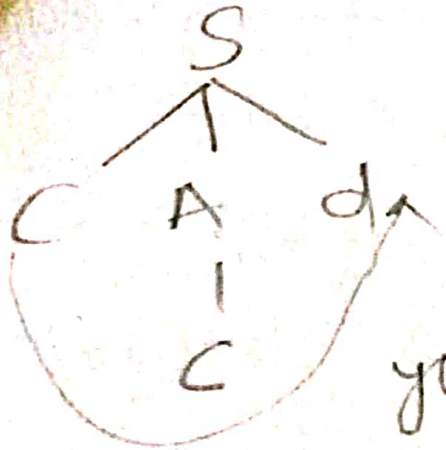
string cdd





yield = cabd ≠ required string

So, we backtrack to A



yield = ccd = required string

E → T + E / T

T → V * T / V

V → id

```
procedure (E)
{
 T()
 if (lookahead = '+')
 { match ('+');
   E();
 }
}
else
error;
if (lookahead = $)
{ declare success;
}
procedure T()
{
  V()
  if (lookahead = '*')
  { match ('*');
    T();
else
  error;
Procedure V()
{ if (lookahead = 'id')
  { match('id');
  }
else
  error;
procedure match(token t);
if( lookahead = t)
{lookahead = next token
}
else error;
}
```

E → E+T/T

T → T*F/F

F → (E)/id.

Sol) remove left recursion.

E → TE'

E' → +TE'/ε

T → FT'

T' → *FT'/ε

F → (E)/id

2) remove left factoring

there is no LF

3) First(E) = {(, id}

E' = {+, ε}

T = {(, id}

T' = {*, ε}

F = {(, id}

Fo(E) = {$,)}

E' = {$,)}

T = {$,), +}

T' = {$,), +}

F = {$,), *, +}

Parsing table

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

Ex: check if string is accepted

id+id

Stack	Input	Action
$	id+id$	
$E	id+id$	$E \to TE'$ push
$E'T	id+id$	$T \to FT'$ push
$E'T'F	id+id$	$F \to id$ push
$E'T'id	id+id$	pop
$E'T'	+id$	$T' \to \varepsilon$ push
$E'	+id$	$E' \to +TE'$
$E'T+	+id$	pop
$E'T	id$	$T \to FT'$
$E'T'F	id$	$F \to id$ push
$E'T'id	id$	pop
$E'T'	$	$T' \to \varepsilon$ push
$E'	$	$E' \to \varepsilon$ push
$	$	accepted

$E' + id$ | id $
$E' T$ | $
E' | $
$ | $

pop
$T' \to \varepsilon$ push
$E' \to \varepsilon$ push
Accepted

Shift reduce Parsing

$E \to E+T/T$
$T \to T*F/F$
$F \to (E)/id$

id * id

Stack	input buffer	Action
$	id * id $	shift id
$id	* id $	reduce F→id
$F	* id $	reduce T→F
$T	* id $	shift
$T*	id $	shift
$T* id	$	reduce F→id
$T*F	$	reduce T→T*F
$T	$	reduce E→T
$E	$	Accepted

LR(0)

$E \to TT$

$T \to aT / b$

Augmeat

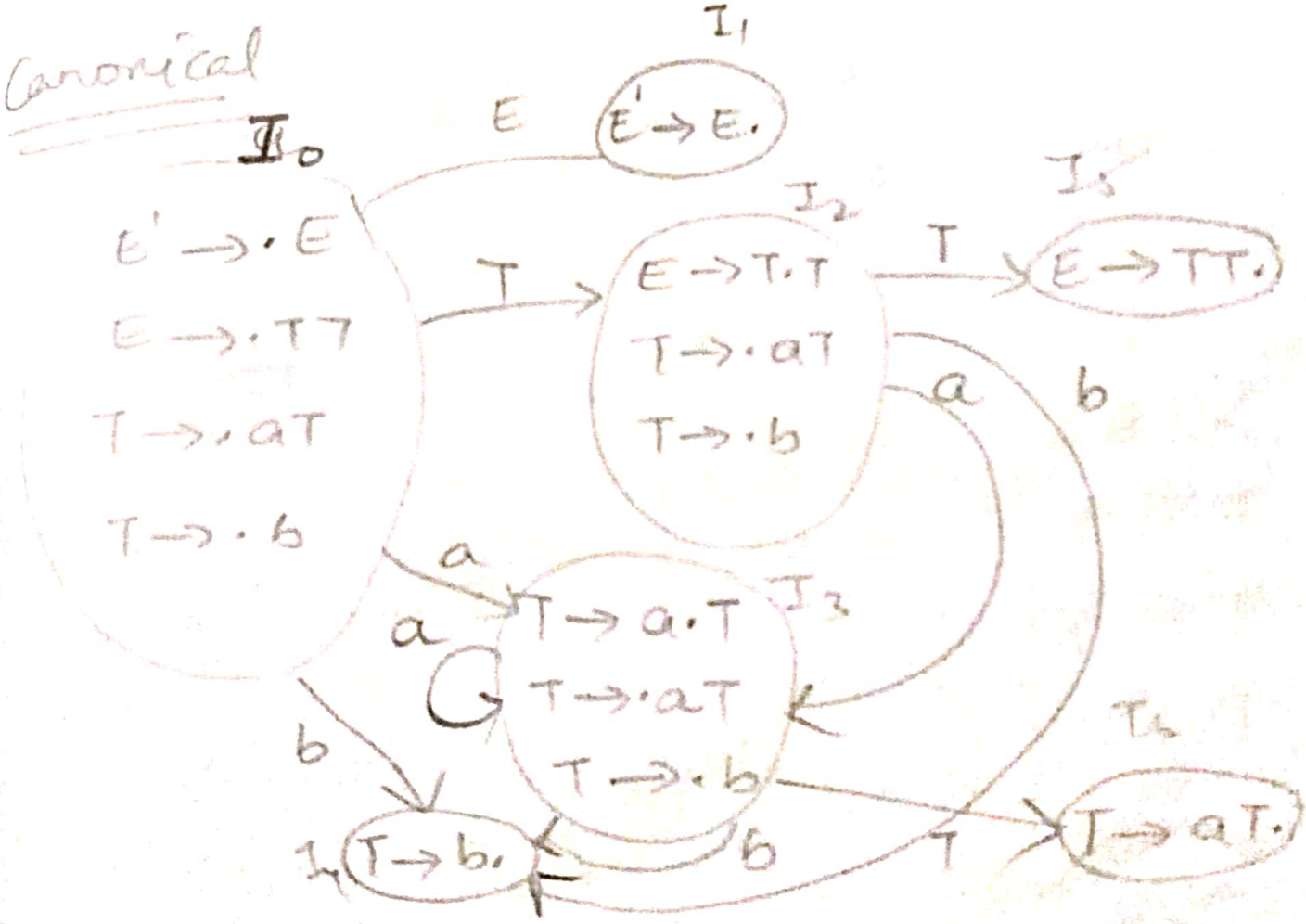
$E' \to .E$ R_1

$E \to .TT$ R_2

$T \to .aT$

$T \to .b$ R_3

Canonical



Parsing table

	action			goto	
	a	b	$	E	T
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	R_3	R_3			
I_5	R_1	R_1			
I_6	R_2	R_2			

$S \to AS$

$S \to b$

$A \to SA$

$A \to a$

Augment

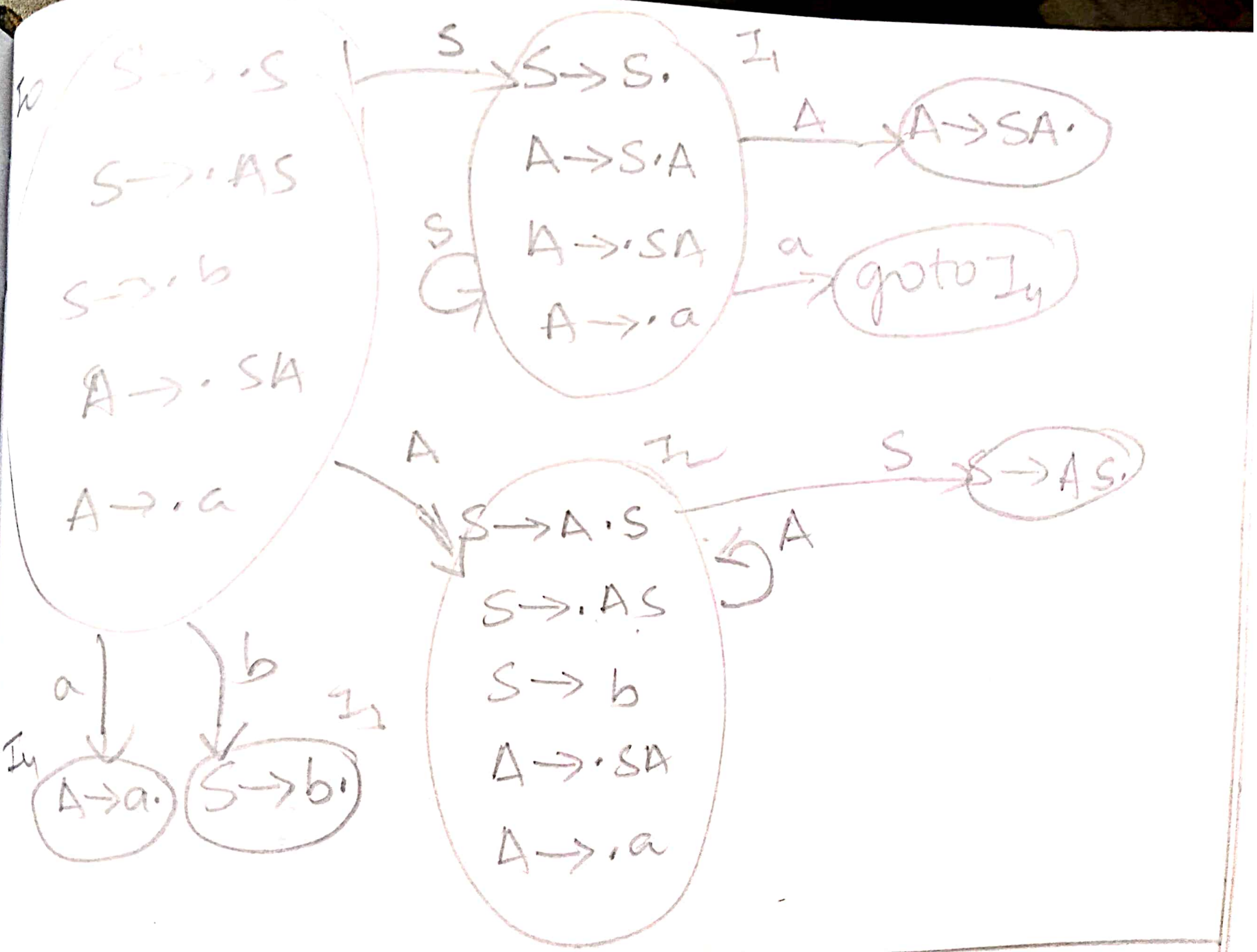
$S' \to .S$

$S \to .AS$ R_1

$S \to .b$ R_2

$A \to .SA$ R_3

$A \to .a$ R_4


I0
S' → .S
S → .AS
S → .b
A → .SA
A → .a
S
I1
S' → S.
A → S.A
A → .SA
A → .a
A
I2
S → A.S
S → .AS
S → b
A → .SA
A → .a
S
S → AS.
A
a
goto I4
b
I3
S → b.
a
I4
A → a.

LALR

$S' \to S$ $S \to CC$, $C \to aC$, $C \to b$

Augment

I_0 :- $S' \to .S, \$$

$S \to .CC, \$$

$C \to .aC, a/b$

$C \to .b, a/b$

Goto (I_0, S)

I_1: $S' \to S., \$$

Goto (I_0, C)

I_2: $S \to C.C, \$$ $C \to .b, \$$

$C \to .aC, \$$

Goto (I_0, a)

$C \to a.C, a/b$

$C \to aa.ac, a/b$ I_3

$C \to .b, a/b$

Goto (I_0, b)

$C \to b., a/b$ - I_4

Goto (I_2, C)

I_5: $CC., \$$

Goto (I_2, a)

I_6: $a.C, \$$

$C \to .aC, \$$

$C \to .b, \$$

I_7: Goto (I_2, b)

I_7: $b., \$$

Goto (I_3, C)

$C \to aC., a/b$

Goto (I_3, a) $\to I_3$

Goto (I_3, b)