10/01/2023

Compiler Design

→ human level language (source program) → machine level language (target program).

└ Both have diff. languages.

Source program → Compiler → Target.

↓

errors / messages.

Input → Target Program → Output

(executable machine programming language).

Interpreter (compilation done line by line).

Line by line Source code → Interpreter → Target program.

A **compiler** is a program that can read a program in one language that is the source language, and translate it into an equivalent program in another language (target language).

A target language may be another programming language or the machine language of any computer. One of the imp. roles of the compiler — to report any errors in the source program detected during translation process.

If the target program is an executable machine code then it processes the input and produces the output.

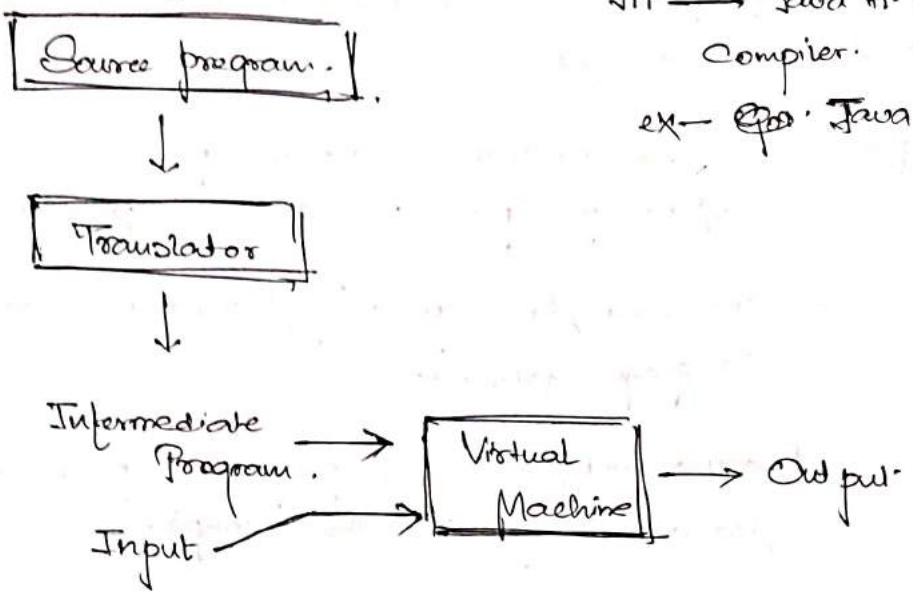
Interpreter.

An interpreter is a computer program that directly executes, i.e. performs instructions written in a programming / scripting language without previously compiling them into a machine lang. program.

Compiler	Interpreter
It takes entire program as input.	Single instruction is taken as input.
Intermediate code is generated	no intermediate code is generated.
Conditional control statements are executed faster.	Conditional control statements are executed slower.
Memory requirement is more as object code is generated.	Mem. requirement is less because no intermediate code is ~~generated~~ generated.
Program need not be compiled every time.	every time a high level program is converted to low level program.
errors are displayed after whole program is checked.	Errors are displayed for every instructions that is being interpreted.
→ C, Java.	→ Basic, Python.

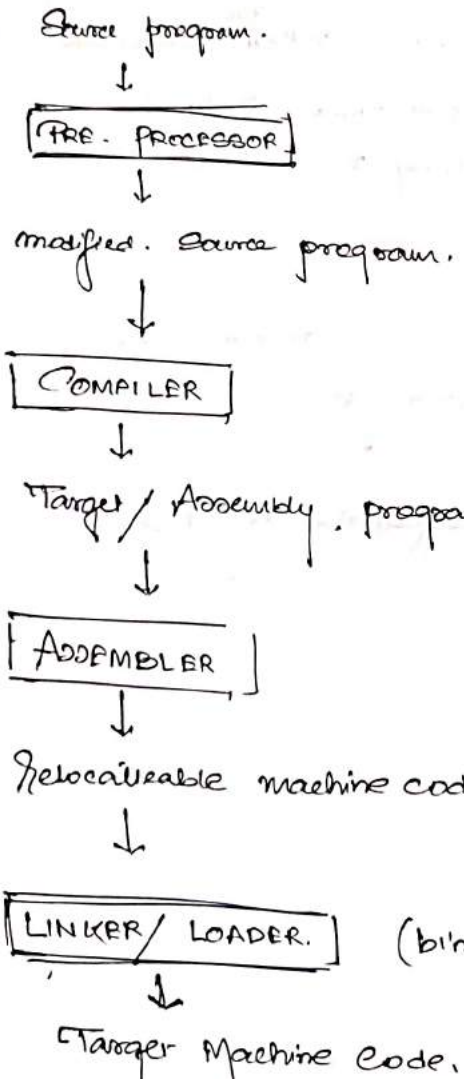
Hybrid Compiler.

JIT ⟶ Java In-time Compiler.
ex— Eg: Java



→ Java source program is compiled into an intermediate form called byte codes, Then these byte codes are interpreted by a virtual machine to get the outputs.

Cousins of Compiler.



Cousins of Compiler :—

11/01/2023

A source program is divided into seperate modules. Which are stored in seperate files.

Preprocessor — task of collecting source program is entrusted by a seperate program, called a preprocessor.

— It is used for expanding shorthands (macros) into source language statements.

Finally a modified source program is produced which is fed as an input to the compiler.

COMPILER : This translates modified source program into a target assembly program.

Assembler : Assembler processes the target assembly lang. program, and produces a relocatable machine code.

— Large programs are combined in pieces so ~~that~~ the ~~relocatable~~ Relocatable machine code may be linked with other. Relocatable object files and library files that actually runs on the machine.

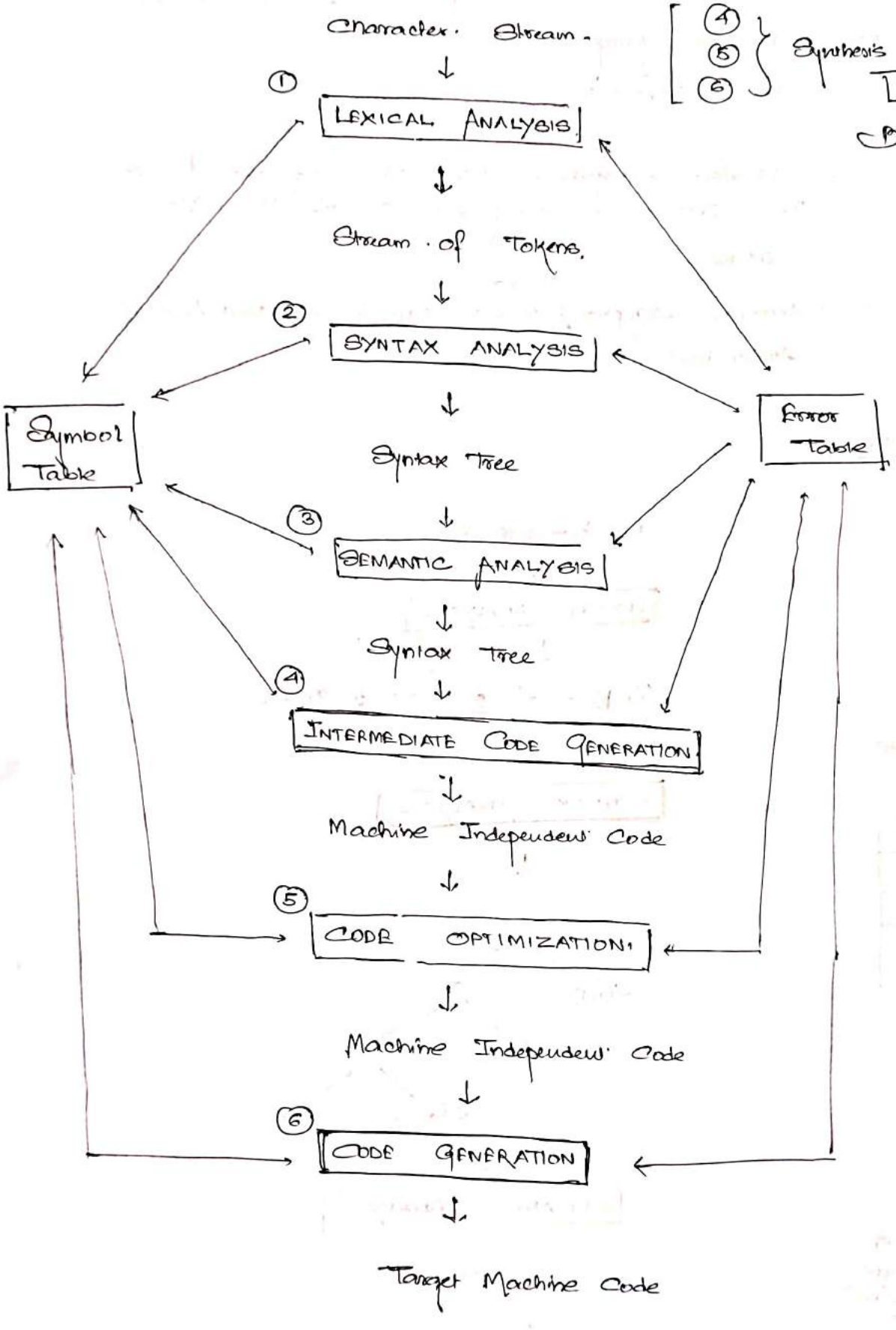
Linker :— Linker resolves external memory addresses where the code in one file may refer to a location in another file.

Loader :— Loader puts together all the executable object files into memory for execution.

Phases of Compiler:

2 Broad Phases.

- ① ② ③ } Analysis phase — Front End
- ④ ⑤ ⑥ } Synthesis Phase — Back End.



Symbol Table — is a database.

(stores the initializations (identifiers), which are connected to all 6 phases of compiler for interaction with the program.

Acts as an references)

Error Table — is also a database which is a reference to all the errors occuring in any of the given phases,

or error.

— throws exceptions which can be handled through error table.

— Example

$a = b + c * 60$

↓

LEXICAL ANALYSIS

↓

$\langle id, 1 \rangle = \langle id, 2 \rangle + \langle id, 3 \rangle * 60.$

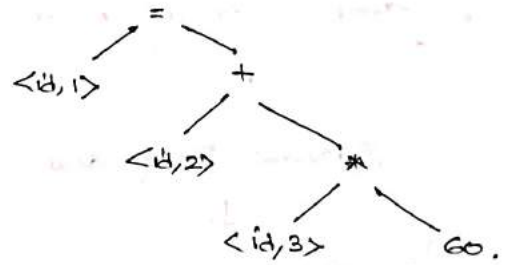
Symbol Table

1	id	a
2	id	b
3	id	c

↓

SYNTAX ANALYSIS

↓

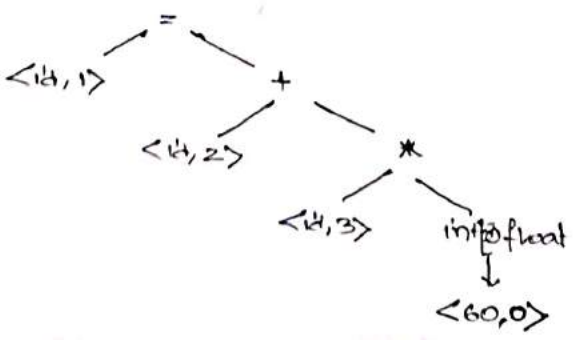


↓

SEMANTIC ANALYSIS

↓

(Conversion of a datatype in semantics)



↓

Intermediate Code Generation

↓

$t_1 = \text{int to float}(60.0)$

$t_2 = \langle id, 3 \rangle * t_1$

$t_3 = \langle id, 2 \rangle + t_2$

$t_4 = t_3$

$\langle id, 1 \rangle = t_4$

↓

CODE OPTIMISATION

↓

$t_1 = \langle id, 3 \rangle * \text{int to float}(60.0)$

$\langle id, 1 \rangle = \langle id, 2 \rangle + t_1$

↓

CODE GENERATION

↓

LD R_1, $\langle id, 3 \rangle$ — LD R_2, $\langle id, 2 \rangle$

MUL R_2, R_1, #60.0

ADD R_1, R_2, $\langle id, 2 \rangle$ — R_3, R_3, R_2

SD $\langle id, 1 \rangle$, R_3

——— $a = x + y * z + 60.$

——— $x = m/n + 30.0$

c $g = x + y * z + 60.$

→ Expression

$g = x + y * z + 60.$

↓

Lexical Analysis

↓

$\langle id,1 \rangle = \langle id,2 \rangle + \langle id,3 \rangle * \langle id,4 \rangle + 60.$

↓

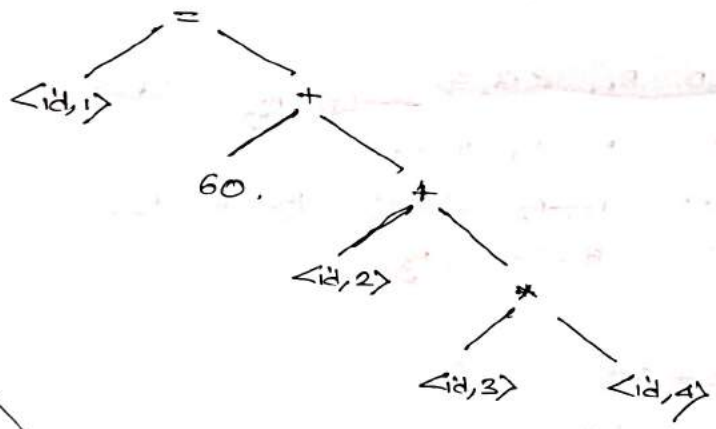
Syntax Analysis

↓

Symbol Table

1	id	g
2	id	x
3	id	y
4	id	z

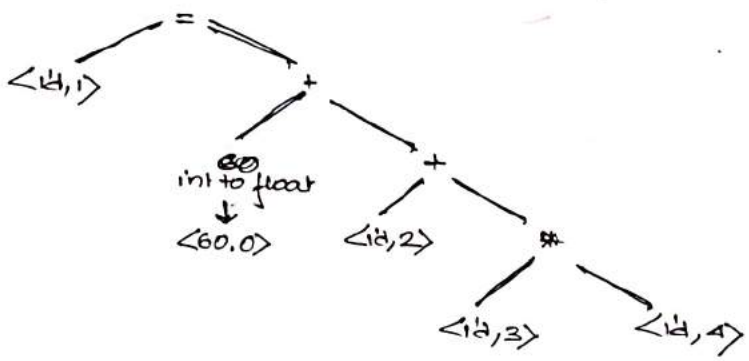
Symbol Table

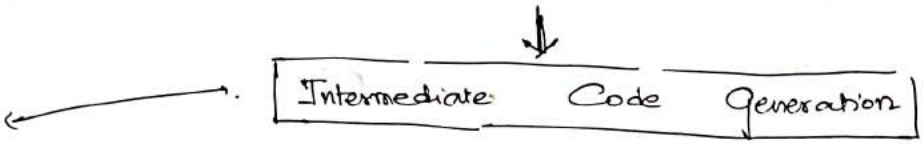


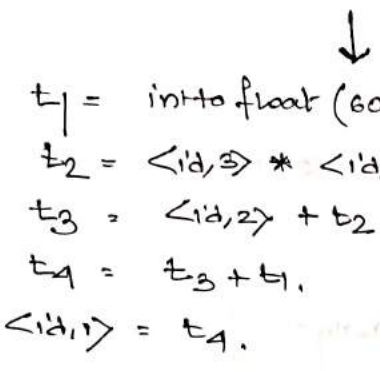
↓

Semantic Analysis

↓







$t_1 = \text{int-to-float}(60.0)$

$t_2 = \langle id, 3\rangle * \langle id, 4\rangle$

$t_3 = \langle id, 2\rangle + t_2$.

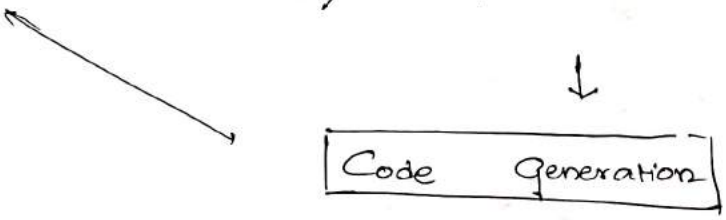
$t_4 = t_3 + t_1$.

$\langle id, 1\rangle = t_4$.

$t_1 = \langle id, 3\rangle * \langle id, 4\rangle$.

$t_2 = \langle id, 2\rangle + t_1 + \text{int-to-float}(60.0)$

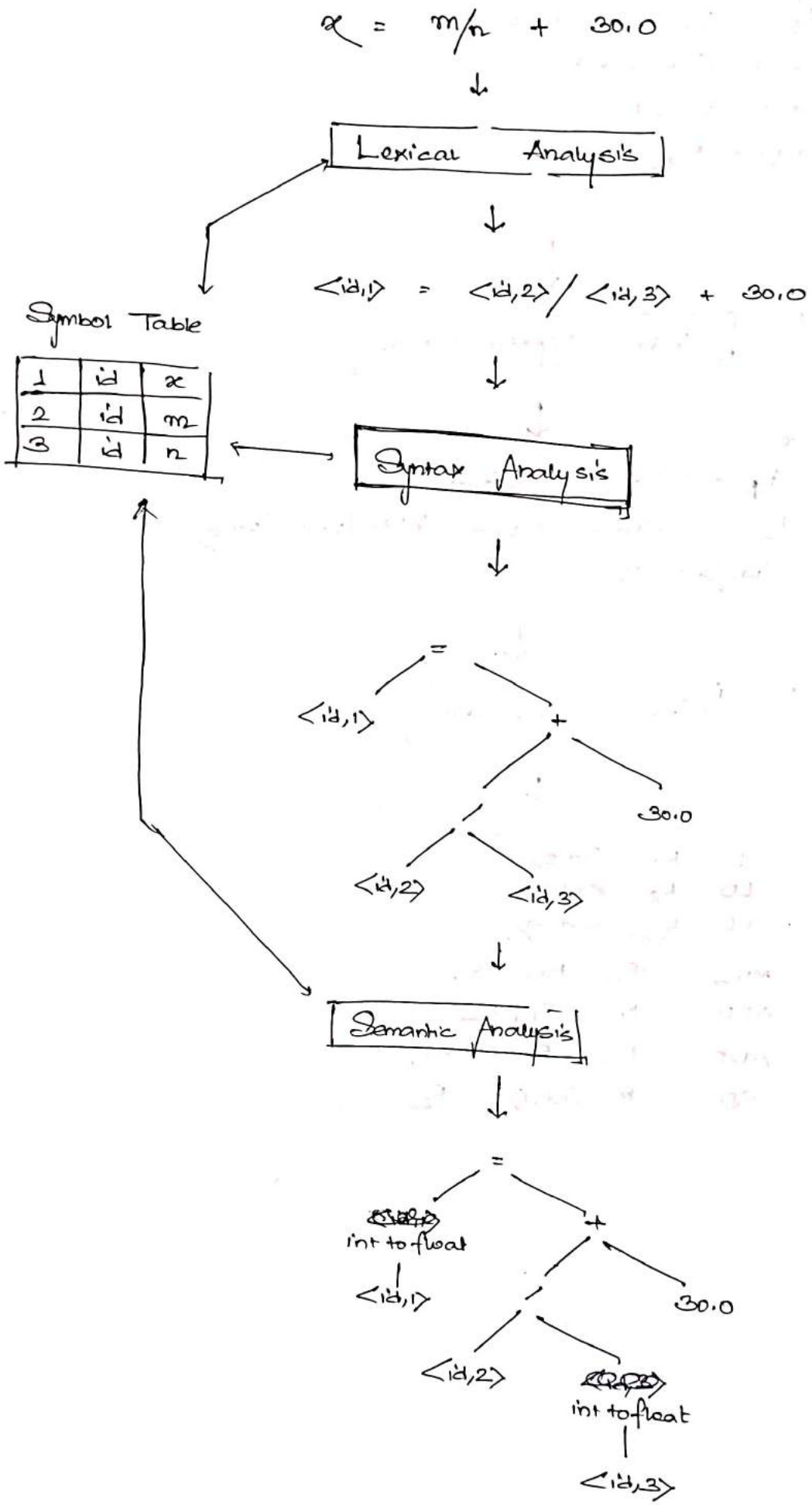
$\langle id, 1\rangle = t_2$,




```
LD   R1, <id,2>
LD   R2, <id,3>
LD   R3, <id,4>
MUL  R2, R2, R3.
ADD  R2, R1, R2
ADD  R2, R2, #60.
SD   <id,1>, R2
```

x = m/n + 30.0 5/2

Expression —



↓

Intermediate Code Generation.

↓

t_1 = int to float (<id, 3>)

t_2 = <id, 2> / t_1.

t_3 = t_2 + 30.0

~~t_4 = <id,1> = int to float (~~

t_4 = int to float (<id, 1>)

~~t_5~~ t_4 = t_3.

↓

Code Optimisation

↓

t_4 = <id, 2> / [int to float (<id, 3>)]

<id, 1> = t_1 + 30.0.

↓

Code Generation

↓

```
LD   R1, <id,2>
LD   R2, <id,3>.
DIVD. R1, R1, R2
ADD  R1, R1, #30.0.
SD   <id,1>, R1.
```

16/01/23

Lexical Analysis (lexer/ Scanner)

→ First phase of compiler is Lexical Analysis phase or scanner phase. It reads the stream of characters from source program., and groups them into meaningful sequences called lexemes.

→ For each lexeme, the Lexical Analyser produces a token in the form of. <token name, attribute value>. and passes it to the next phase.

The first component of token is token name, which is an abstract symbol, that is used during syntax analysis.

The second component is attribute value, which points to an entry in the symbol table for this token.

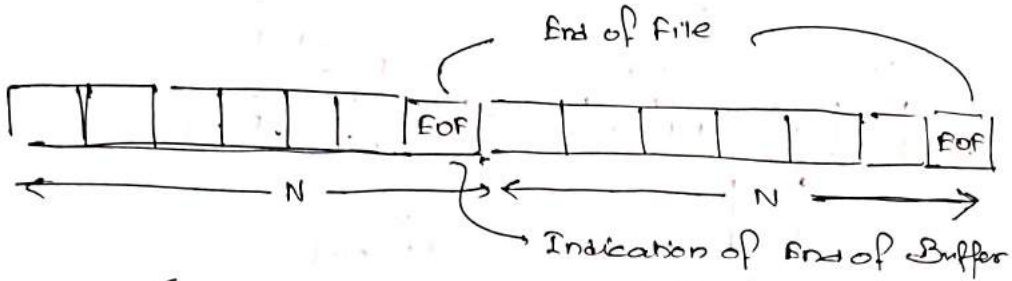
Role of Lexical Analyser.

→ Reads the input character from Source program.
→ removes comment lines & white spaces.
→ groups characters into lexemes.
→ produces sequence of tokens as output.
→ interacts with symbol table
→ co-relates error messages if found any.

if (a==0 && b==0)

Input Buffering (Look Ahead Concept)

17/01/2023



→ Pointers (Lexeme Begin, forward ptr).

→ another EOF marker is used to mark end of current instruction.

└ Sentinels.

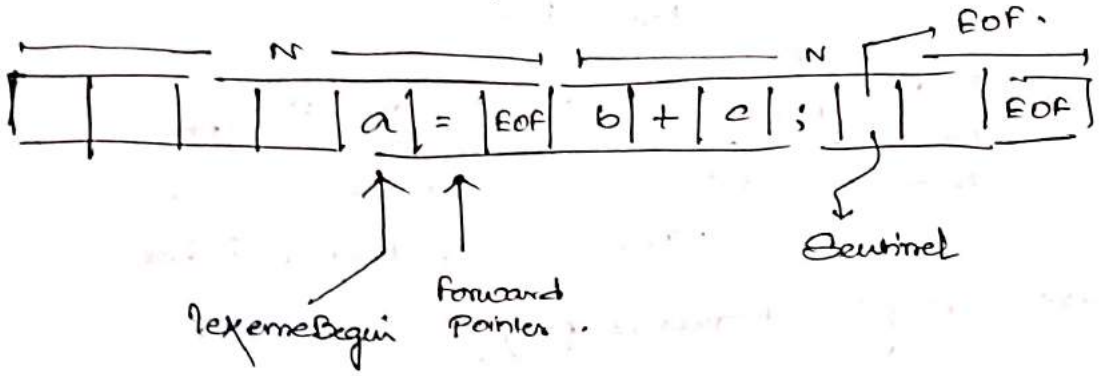
A source program that is fed to the compiler passes through the first phase (Lexical Analysis phase), in which characters of the program are scanned using buffer pairs, character by character. Two buffers are used inorder to handle large lookaheads easily. Amount of time taken to process characters are and large no. of characters, must be compiled using specialised buffering techniques to reduce the amount of overhead, in the buffers, Therefore, in buffer-pair technique, the buffers are reloaded alternatively.

⇒ Two pointers are used, →

(i) ~~lexeme Begin~~. lexemeBegin — marks the beginning of current lexeme.

(ii). PointerForward — scans ahead until a pattern match is found.

Sentinels — A special character denoted by EOF, marks the end of instruction, and is not a part of source program.



- lexical analysis proved above syntax analysis, ???

→ Definitions

A token is a pair constating of token name, and optional attribute value. The token name is an abstract symbol, representing a kind of lexical unit A particular keyword. or a sequence of input characters, denoting an identifier.

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

For e.g. — In a keyword the pattern is the sequence of a characters, that form the keyword,

Lexeme - A lexeme is a sequence of characters in the source program. that matches the pattern for a token and is identified by. lexical analyser. as an instance of a token.

Classes of tokens

→ covers most of C or all of the tokens.

- 1 token for each keyword.
- Tokens for Operators (either individual or in groups)
- 1 token representing identifiers.
- One/more tokens representing constants (numbers/literals)
- 1 tokens for each punctuation.

- → Lexical Errors can be spelling mistakes.
  (panic mode recovery errors).
  - insertion of extra chars ~~forr~~
  - missing chars.
  - swapping of chars.

forr → extra char. — for

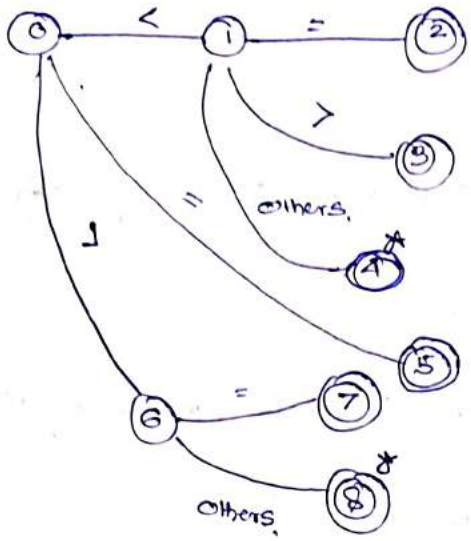
fo — for

fro — for.

} recovered by lexical analyser (panic mode).

18/01/2023

FSM (F.S.M).



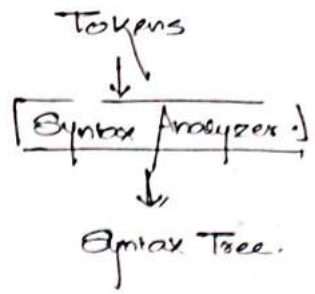
— Lex Tool is used above c resource file for running.
Lexical files (.extension: yy).

Definition section (Declaration)
%.%.
Rules section (Translational Rules)
%%.
C Code section. (Auxiliary function).

— .— .yy.c

03/01/2023

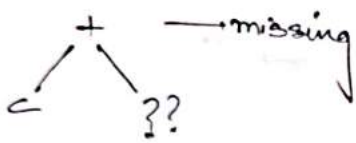
Context free Grammar (CFG).



main ( ) { ... }

missing parenthesis. (syntax error).

b = c + ;



→ missing operand. (syntax error).

- panic mode recovery.
- global correction.
- phase level recovery.
- error productions.

} error handling techniques of syntax analyzer.

CFG has 4 parts.

V_n — set of non terminals.
V_t — set of terminals.
P — set of productions.
S — start symbol.

} — required for parsing.

V_n — {A–Z}
V_t — {a–z, 0–9, {, }, (,)}
P — {A → a, B → b}
S — {S, B, A ...}.

S → AB
A → a
B → b.
} 1 CFG.

S → AB.
A → B
B → C
} not a CFG as no terminals are present.

E → E+E /
E → E*E .
E → id

⇒ E → E+E / E*E / id.

S → AB
A → a
B → b

⇒ S → AB | Ba

Two derivations of CFG.

→ Right Most Derivative (RMD)

→ Left Most Derivative (LMD).

↓

left most non-terminal to be derived first.

right most non-terminal to be derived.

first

LMD

E → E+E
E → E*E
E → id

id + id * id

E → E+E
→ id + E
→ id + E*E
→ id + id*E
→ id + id*id.

E → E*E
→ E*E+E
→ id*E+E
→ id*id+E
→ id*id+id.

RMD

E → E+E
→ E+E*E
→ E+E*id
→ E+id*id
→ id+id*id.

E → E*E
→ E*E+E
→ E*id+E
→ E+id*id
→ id+id*id.

↓

This parser cannot handle this grammar because of ambiguously.

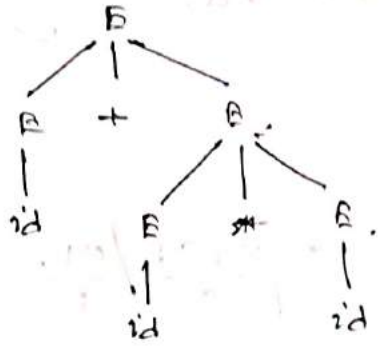
↓

A grammar is called ambiguous when the grammar has more than one LMD or more than one RMD or both cases, then there occurs ambiguity in the CFG., and thus,

we need to remove the ambiguity.

- Parse Tree

E ⟶ i'd + i'd * i'd



- A grammar which produces more than one LMD or more than one RMD for the same sentence/string, is called as ambiguous grammar. or. ambiguity.

⟶ Top Down Parser. (LMD) Input scan from Left to Right.
⟶ Bottom up Parser.

- Top Down Parser.

1) Left Recursive Grammar.
2) Left Factored Grammar.

⟶ method to remove ambiguity from grammar.
↓
make it worse for parse tree.

- Removal of Left Recursion

$A \longrightarrow A\alpha \mid \beta$

$\Rightarrow$

$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' / \epsilon$

$A \longrightarrow \underbrace{A}\ \underbrace{a}\ b\underbrace{B} / \underbrace{b}$
$\qquad\qquad\ \alpha \qquad\quad\ \beta$

$A \longrightarrow a\ bA'$

$A' \longrightarrow abBA' / \epsilon$

$S \longrightarrow AB.$
$A \longrightarrow Ab/b.$
$B \longrightarrow a.$

$\longrightarrow$

$S \longrightarrow AB.$
$A \longrightarrow bA'$
$A' \longrightarrow bA'/\epsilon$
$B \longrightarrow a$

(1) $S \longrightarrow \underbrace{S01S1}_{\alpha}/\underbrace{01}_{\beta}.$

$S \longrightarrow 01S'$
$S' \longrightarrow .01S1S'/\epsilon.$

(2) $L \longrightarrow L * S/p.$

$L \longrightarrow . pL'$
$L' \longrightarrow . *SL'/\epsilon.$

(3) $A \longrightarrow ABO1/B.$

$A \longrightarrow . BA'$
$A' \longrightarrow .BO1.A'/\epsilon.$

(4) $S \longrightarrow \underbrace{01S1}_{\beta}/S1.$

$S \longrightarrow .01S1.S'$
$S' \longrightarrow .1S'/\epsilon.$

$S \longrightarrow 01S1/$
$S \longrightarrow S1.$

(5) $T \longrightarrow T-B/a.$

$T \longrightarrow . aT'$
$T' \longrightarrow . -BT'/\epsilon.$

$A \longrightarrow Abb\,Bb \;/\; bB \;/\; aB \;/\; \emptyset \;/\; Aab$

$\alpha \qquad \beta_1 \qquad \beta_2 \qquad\qquad \alpha_2$

$\Rightarrow$

$A \longrightarrow bBA' \;/\; aBA'$

$A' \longrightarrow bBbA' \;/\; abA' \;/\; \epsilon.$

$A \longrightarrow A\alpha_1 / A\alpha_2 / \ldots / A\alpha_n / \beta_1 / \beta_2 / \ldots / \beta_n.$

~~$A \longrightarrow \beta_1 A' / \beta_2 A' / \beta \ldots / \beta_n A'$~~

$A \longrightarrow \alpha_1 A' / \alpha_2 A' /$

$A \longrightarrow \beta_1 A' / \beta_2 A' / \ldots / \beta_n A'$

$A' \longrightarrow \alpha_1 A' / \alpha_2 A' / \ldots / \alpha_n A' / \epsilon.$

$A \longrightarrow AbaABa$

↳ can be rewritten $\longrightarrow$

$A' \longrightarrow .\, AbaABa / \epsilon.$

$\Rightarrow$ $A \longrightarrow baA'$

$A' \longrightarrow .\, baBaA' / \epsilon.$

25/03/2023

Left Factored Grammar.

$A \longrightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha \ldots / \alpha\beta_n / \gamma$ → same leftmost symbol.

$A \longrightarrow \alpha A' / \gamma$.
$A' \longrightarrow \beta_1 / \beta_2 / \ldots / \beta_n$

→ Removal of left factored symbol.

C-Productions having same left most symbol be it terminal / non-terminal, we removed factoring symbol (s) to a single (leftmost) the ambiguity in the grammar.

Example :-

$A \Longrightarrow a/Ab/aaab/ab/b.$

Answer: $A \longrightarrow aA'/b.$
$A' \longrightarrow Ab/ab/b.$

$A \rightarrow Ab/Aa/\epsilon$
$\gamma \rightarrow \alpha A / \alpha\beta / \epsilon$

Q1. $S \longrightarrow aSSbS/abSb/aab/b.$

$S \longrightarrow aS'/b.$
$S' \longrightarrow SSbS/bSb/ab.$

$A \rightarrow AA'/\epsilon$
$A' \rightarrow b/a$

Q2. $S \longrightarrow bSaaS/bba/baS.$

$S \longrightarrow bS'$
$S' \longrightarrow SaaS/ba/aS.$

Q3. $S \longrightarrow ab/abA/abSc$

$S \longrightarrow abS'$
$S' \longrightarrow \epsilon/A/Sc.$

$S \rightarrow aS'$
$S' \rightarrow b/bA/bSc$
$S' \rightarrow bS''$
$S'' \rightarrow \epsilon/A/Sc$

4. $S \rightarrow Aab / Abc / Ab / a$.

~~$S \rightarrow Aas' / As$~~

$S \rightarrow AS' / a$.

$S' \rightarrow ab / bS''$

$S'' \rightarrow c / \epsilon$.

$S \rightarrow AS' / a$.

$S' \rightarrow ab / bc / b$

$S \rightarrow AbS' / Aab / a$.

$S' \rightarrow c / \epsilon$.

$x^2 + xy + x^3$

$x^2(x+1) + xy$

$x(x+y+x^2)$

$A \rightarrow Ab / Aa / \epsilon$.

$A \rightarrow A\alpha / \beta$.

Removing factored:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \epsilon$.

$A \rightarrow A''$

$A'' \rightarrow A' A'' / \epsilon$

$A' \rightarrow b / a$

Removing recursion

▨ $A \rightarrow A\underset{a_1}{ab} / A\underset{a_2}{b} / \underset{p}{bC}$

$A \rightarrow bcA'$

$A' \rightarrow ab \,/\, abA' / bA' / \epsilon$.

6. $A \rightarrow aaAaA / abAa / aaaA / ba$.

7. $B \rightarrow aBaBb / aB / baB / ab / b$,

8. $S \rightarrow aAd / aB$,

$A \rightarrow a / ab$,

$B \rightarrow cd / dc$,

9. $A \rightarrow Aa / bB$,

$B \rightarrow AcB / Ad / a$

10. $S \rightarrow A$.

$A \rightarrow B / BaA$.

$B \rightarrow bC$.

$C \rightarrow Cb / Cc / \epsilon$.

11. $S \rightarrow aAb / abB$.

$A \rightarrow Aa / \epsilon$.

$B \rightarrow Bb / b / \epsilon$.

12. $S \rightarrow aXYb$

$X \rightarrow Xda / YZ$

$Y \rightarrow q / \epsilon$.

$Z \rightarrow Zh / h / \epsilon$.

13. $A \rightarrow BC\alpha / \gamma$.

$B \rightarrow \gamma A / \epsilon / B\alpha$.

$C \rightarrow C\gamma / \alpha / \lambda\alpha$.

14. $S \rightarrow Ad / bAc / Ba / bBa$.

$A \rightarrow d$

$B \rightarrow d$.

15. $S \rightarrow assb / aScS / a$

Rahul Bhowmick — 20051824

6. $A \longrightarrow aaAaA / abAa / aaaA / ba.$

Ans:
$A \longrightarrow aA' / ba$
$A' \longrightarrow aAaA / bAa / aaA.$

7. $B \longrightarrow aBaBb / aB / baB / ab / b.$

Ans:
$B \longrightarrow aB' / bB''$
$B' \longrightarrow BaBb / B / b.$
$B'' \longrightarrow aB / \epsilon.$

$B \longrightarrow aB' / bB''$
$B' \longrightarrow BB''' / b.$
$B''' \longrightarrow aBb / \epsilon.$
$B'' \longrightarrow aB / \epsilon.$

8. $S \longrightarrow aAd / aB.$
$A \longrightarrow a / ab$
$B \longrightarrow cd / dc.$

Ans:
$S \longrightarrow aAS'$
$S' \longrightarrow Ad / B$
$A \longrightarrow aA'$
$A' \longrightarrow b / \epsilon.$
$B \longrightarrow cd / dc$

9. $A \longrightarrow Aa / bB.$
$B \longrightarrow ACB / Ad / a.$

Ans:
$A \longrightarrow bBA'$
$A' \longrightarrow aA' / \epsilon.$
$B \longrightarrow AB' / a.$
$B' \longrightarrow CB / d.$

10. $S \to A$

$A \to B / BaA.$

$B \to bC$

$C \to Cb / Cc / \epsilon.$

Ans: $S \to A.$

$A \to BA'$

$A' \to \epsilon / aA.$

$B \to bC.$

$C \to aC' / \epsilon.$

$C' \to b / c'$

$C \to aC''.$

$C'' \to C'C'' / \epsilon.$

11. $S \to aAb / abB.$

$A \to Aa / \epsilon$

$B \to Bb / b / \epsilon,$

Ans: $S \to aS'$

$S' \to Ab / abB.$

$A \to A'$

$A' \to aA' / \epsilon.$

$B \to bB' / B'$

$B' \to bB' / \epsilon$

15. $S \to aSSb / aScS / a.$

Ans: $S \to aS'$

$S' \to SSb / ScS / \epsilon.$

$S' \to SS'' / \epsilon.$

$S'' \to Sb / cS.$

14. $S \to Ad / bAc / Ba / bBa$

$A \to d$

$B \to d.$

Ans: $S \to Ad / Ba / bBS'$

$S' \to Ac / Ba.$

$A \to d$

$B \to d$

18. $A \to BCx / y$

$B \to yA / \epsilon / Bx$

$C \to Cy / x / Ax.$

Ans: $A \to BCx / y$

$B \to yAB' / B'$

$B' \to xB' / \epsilon.$

$C \to xC' / AxC'$

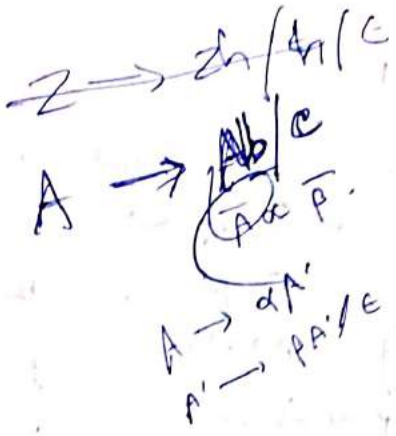
$C' \to yC' / \epsilon.$

12. $S \rightarrow aXYb$

$X \rightarrow Xd \mid a \mid YZ$

$Y \rightarrow q \mid \epsilon$

$Z \rightarrow Zh \mid h \mid \epsilon$.



Ans:

$S \rightarrow aXYb$.

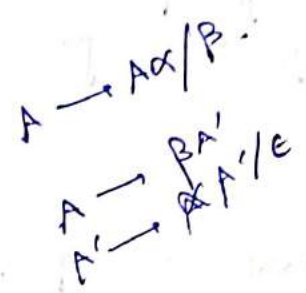
$X \rightarrow .\ YZX' \mid$

$X' \rightarrow daX' \mid \epsilon$.

$Y \rightarrow q \mid \epsilon$.

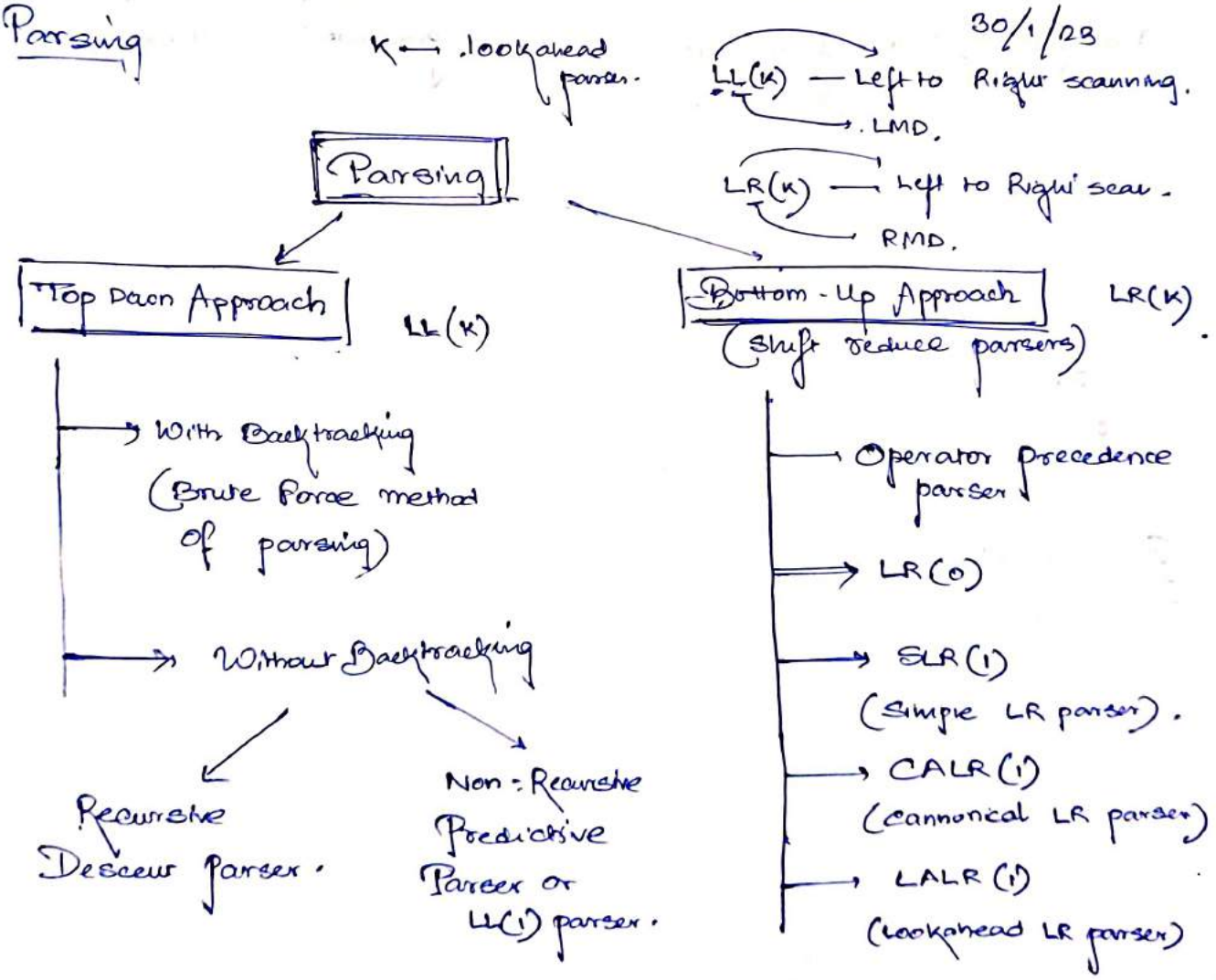
$Z \rightarrow .\ hZ' \mid Z'$

$Z' \rightarrow hZ' \mid \epsilon$.



Parsing

30/1/23



Top-Down Approach.

1) We start from the starting symbol and reach the string. If the string is derived successfully from the starting symbol, replacing diff. productions, successfully then parsing is completed.

2) uses special class of grammar

LL(k) ——— k is the lookahead.

- left to right scan.
- LMD.

3) This parsing cannot accept any type of ambiguous grammar.

Bottom-Up Approach.

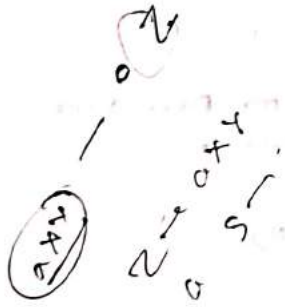
1) We start from the string, reduce the matching strings to the non-terminal symbol & finally reach the starting symbol. If the starting symbol is reached then parsing is successful.

2) uses special class of grammar

LR(k) ——— k ← lookahead.

- RMD
- left to Right scan.

3) can accept ambiguous grammar, sometimes.



Make this grammar for LL(1) parsing. → Question format

→ Parser eligible for Top Down.

(remove factoring or recursion if needed).

$S \rightarrow Sa$

$A \rightarrow \alpha\beta$

Brute - force Approach :-

$S \rightarrow bd / c / aAd / aBc.$

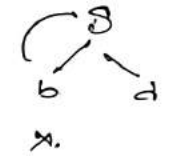
$A \rightarrow cf / c.$

$B \rightarrow d / c.$

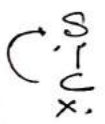
"adc".

- Brute force (start for first production of start symbol, and if not matches with the string format, it will result in backtracking and take up the next production)

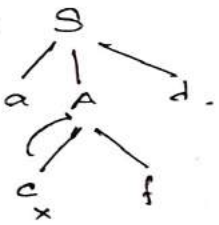
(1)



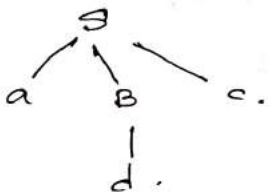
(2)



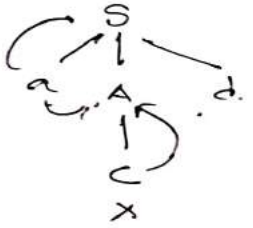
(3)(c)



(4).



(b)



→ "adc" ✓

↳ String generated as per need

↳ Parsing Done.

Recursive Descent Parser.

G: $E \longrightarrow E+T / T$.

$T \longrightarrow T*F / F$.

$F \longrightarrow id$.

G': $E \longrightarrow TE'$

$E' \longrightarrow +TE' / \epsilon$.

$T \longrightarrow FT'$

$T' \longrightarrow *FT' / \epsilon$.

$F \longrightarrow id$

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

```
E'()
{
 if (l=='+')
 { match(+);
   T();
   E'();
 }
 else
  return;
}
```

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

```
T'()
{
  if (l=='*')
  { match(*);
    F();
    T'();
  }
  else
   return;
}
```

```
main()
{
  E();
  if (l=='$')
  pf("Parsing successful");
  else
   pf("Error");
}
```

Rahul Bhowmick

1. $S \rightarrow A$.
   $A \rightarrow Bb / aCd$
   $B \rightarrow b / \epsilon$.
   $C \rightarrow c / \epsilon$.

2. $S \rightarrow ABD$.
   $A \rightarrow a / BSB$.
   $B \rightarrow b / D$
   $D \rightarrow d / \epsilon$

Ans1 :-

```
main()
{
  S();
  if (l == '$')
  pf("Parsing Successful");
  else
  pf("Error");
}
```

- B()

```
{
  if (l == 'b')
  { match(b);
  }
  else
  return;
}
```

- C()

```
{
  if (l == 'c')
  match(c);
  else
  return;
}
```

- S()

```
{
  A();
}
```

- A()

```
{
  B();
  if (l == 'a')
  { match(a);
    C();
    if (l == 'd')
    match(d);
  }
  else
  {
    B();
    if (l == 'b')
    match(b);
  }
}
```

2) S → ABD
A → a/BSB.
B → b/D.
D → d/ε

```
main()
{
  S();
  if (l == '$')
    pf ("Parsing success");
  else
    pf ("Error");
}
```

```
main()
{
  A();
  B();
  D();
```

```
main()
{
  S();
  if (l == '$')
  match ('$')
```

```
S()
{
  A();
  B();
  D();
}
```

```
A()
{
  if (l == 'a')
  {match (a)
  else
  {
    B();
    S();
    B();
  }
}
```

```
B()
{
  if (l == 'b')
    match (b);
  else
    D();
}
```

```
D()
{
  if (l == 'd')
  match (d);
  else
    return;
}
```

01/02/23.

Non Procedure Predictive Parser. OR. LL(1) parser.

⟹ FIRST Algorithm.

First (α) gives set of terminals that may begin in string derived from α.

1. If 'α' is a terminal symbol, then First(α) = $\{\alpha\}$
2. If 'α' is a non-terminal symbol
   (a) and is defined with a null production $\alpha \longrightarrow \epsilon$ then
   First(α) = $\{\epsilon\}$.
   (b) if it is defined with non-null prediction $\alpha \longrightarrow x_1, x_2 x_3$
   then First(α) = ~~First (x_1, x_2,~~
   = First ($x_1 x_2 x_3$)
   = First (x_1) $\{$if $x_1 \not\Rightarrow \epsilon\}$
   OR = First (x_1) $- \{\epsilon\} \cup$ First ($x_2 x_3$) $\{$if $x_1 \Rightarrow \epsilon\}$.

— Examples:- (non-terminal, terminal)

$S \longrightarrow aAB$.
$A \longrightarrow a$
$B \longrightarrow b$.

First(S) = First (aAB) (x_1, x_2, x_3). in the form First ($\alpha_1 x_2 x_3$)
= First (a) $\because \alpha \not\rightarrow \epsilon$ = First (α_1) $\because x_1 \not\Rightarrow \epsilon$
= $\{a\}$

First(A) = First (a) $\because$ $A \longrightarrow a$ $\alpha \not\rightarrow \epsilon$
= $\{a\}$

First(B) = First (b). " ".
= $\{b\}$

S → AB
A → a
B → b.

First (A) = First (a) = {a}

First (B) = First (b) = {b}

First (S) = First (AB) (x_1, x_2)

Now Since First (A) = {a}

and $x_1 \neq \epsilon$.

∴ First (S) = First (A)

= First (a)

= {a}.

S → AB
A → a/ϵ.
B → b.

First (B) = First (b) = {b}.

First (A) = First (a) ∪ First (ϵ)

= {a, ϵ}.

Substitute A = {a, ϵ}. in S,

S → aB/B.

So, First (S) = First (AB)

= First (A) − {ϵ} ∪ First (B) ∵ A → ϵ. ✓

= {a, ϵ} − {ϵ} ∪ {b}

= {a} ∪ {b} = {a, b}.

S → AB.
A → a/ϵ.
B → b/ϵ.

[S → ab
  S → a
  S → b
  S → ϵ]

First (B) = {b, ϵ}. ∵ S can produce ϵ when A → ϵ, B → ϵ.

First (A) = {a, ϵ}

First (S) = First (AB)

= First (A) − {ϵ} ∪ First (B)

= {a, ϵ} − {ϵ} ∪ {b, ϵ}

= {a, b, ϵ}.

S → ABx.
A → a/ϵ.
B → b/ϵ.

Here at least S → x, when both A → ϵ, B → ϵ. so S doesn't produce ϵ.

[S → abx
  S → ax
  S → bx.
  S → x.]

First (B) = {b, ϵ}

First (A) = {a, ϵ}.

First (S) = First (ABx)

= First (A) − {ϵ} ∪ First (B) − {ϵ} ∪ First (x)

= {a, ϵ} − {ϵ} ∪ {b, ϵ} − {ϵ} ∪ {x}

= {a, b, x}.

Note: When a non-terminal produces multiple productions then to find First(α), we need to take the union of all the first symbols of each of the productions.

1. $S \rightarrow aBDh$.
   $B \rightarrow eC$
   $C \rightarrow bC/\epsilon$.
   $D \rightarrow E/F$.
   $E \rightarrow g/\epsilon$.
   $F \rightarrow f/\epsilon$.

2. $S \rightarrow aABb$.
   $A \rightarrow c/\epsilon$.
   $B \rightarrow d/\epsilon$.

3. $S \rightarrow Bb/Cd$
   $B \rightarrow aB/\epsilon$.
   $C \rightarrow cC/\epsilon$.

4. $S \rightarrow ABCDE$.
   $A \rightarrow a/\epsilon$.
   $B \rightarrow b/\epsilon$.
   $C \rightarrow c$
   $D \rightarrow d/\epsilon$
   $E \rightarrow e/\epsilon$.

2. $S \rightarrow aABb$.
   $A \rightarrow c/\epsilon$.
   $B \rightarrow d/\epsilon$.

Ans:-

B

$First(B) = First(d) \cup First(\epsilon)$

$= \{d\} \cup \{\epsilon\}$

$= \{d, \epsilon\}$.

$First(A) = First(c) \cup First(\epsilon)$

$= \{c\} \cup \{\epsilon\} = \{c, \epsilon\}$.

$First(S) = First(aABb)$

$= First(a)$

$= \{a\}$.

1. $S \rightarrow aBDh$
$B \rightarrow eC$
$C \rightarrow bC/\epsilon$
$D \rightarrow E/F.$
$B \rightarrow g/\epsilon$
$F \rightarrow f/\epsilon.$

$D \rightarrow g$
$D \rightarrow f$
$D \rightarrow \epsilon.$

Ans:- First (F) = ~~{f, ε}~~.
= First (f) ∪ First (ε)
= {f, ε}.

First (E)
= First (g) ∪ First (ε)
= {g, ε}.

First (D)
= First (E) ∪ First (F)
= [First (E) − {ε} ∪ First (ε)] ∪ [First (F) − {ε} ∪ First (ε)]
= [{g, ε} − {ε} ∪ {ε}] ∪ [{f, ε} − {ε} ∪ {ε}]
= {g, ε} ∪ {f, ε}
= {ε, f, g}.

First (C) = First (bC) ∪ First (ε)
= First (b) ∪ {ε}
= {b} ∪ {ε}
= {b, ε}.

First (B) = First (eC)
= First (e)
= {e}.

First (S) = First (aBDh)
= First (a)
= {a}.

4.

$S \rightarrow ABCDE.$

$A \rightarrow a/\epsilon$

$B \rightarrow b/\epsilon$

$C \rightarrow c$

$D \rightarrow d/\epsilon$

$E \rightarrow e/\epsilon.$

Ans:-

First (A) = First (a) ∪ First (ϵ)

= {a} ∪ {ϵ}.

= {ϵ, a}.

First (B) = First (b) ∪ First (ϵ)

= {b} ∪ {ϵ}

= {ϵ, b}.

First (E) = First (e) ∪ First (ϵ)

= {e} ∪ {ϵ}

= {ϵ, e}.

First (D) = First (d) ∪ First (ϵ)

= ~~{d, ϵ}~~.

= {d} ∪ {ϵ}

= {ϵ, d}

First (C) = First (c)

= {c}.

First (S) = First (ABCDE)

= First (A) − {ϵ} ∪ First (BCDE)

= {ϵ, a} − {ϵ} ∪ [First (B) − {ϵ} ∪ First (CDE)]

= {a} ∪ [{b, ϵ} − {ϵ} ∪ First (CDE)]

= {a} ∪ [{b} ∪ {First (C)]

= {a} ∪ [{b} ∪ {c}]

= {a, b, c}.

3. $S \rightarrow Bb/Cd$

$B \rightarrow aB/\epsilon.$

$C \rightarrow cC/\epsilon.$

Ans:-

$First(C) = First(cC) \cup First(\epsilon)$

$= First(c) \cup First(\epsilon)$

$= \{c, \epsilon\}.$

$First(B) = First(aB) \cup First(\epsilon)$

$= First(a) \cup First(\epsilon)$

$= \{a, \epsilon\}$

$First(S) = First(Bb) \cup First(Cd)$

$= [First(B) - \{\epsilon\} \cup First(b)] \cup [First(C) - \{\epsilon\} \cup First(d)]$

$= [\{a, \epsilon\} - \{\epsilon\} \cup \{b\}] \cup [\{c, \epsilon\} - \{\epsilon\} \cup \{d\}]$

$= [\{a\} \cup \{b\}] \cup [\{c\} \cup \{d\}]$

$= \{a, b\} \cup \{c, d\}$

$= \{a, b, c, d\}.$

$S \rightarrow ACB / cbB / Ba$

$A \rightarrow da / BC$

$B \rightarrow g / \epsilon.$

$C \rightarrow h / \epsilon.$

(B) $\quad First(C) = \{ First(h) \cup First(\epsilon)$

$= \{h, \epsilon\}.$

$First(B) = First(g) \cup First(\epsilon)$

$= \{g, \epsilon\}.$

$First(A) = First(da) \cup First(BC)$

$= \{d\} \cup [First(B) - \{\epsilon\} \cup First(C)]$

$= \{d\} \cup [\{g\} \cup \{h, \epsilon\}]$

$= \{d, g, h, \epsilon\}.$

$First(S) = First(ACB) \cup First(cbB) \cup First(Ba)$

$= [First(A) - \{\epsilon\} \cup First(CB)] \cup First(c) \cup [First(B) - \{\epsilon\} \cup First(a)]$

$First(c) - \{\epsilon\} \cup First(B)$

$\{h\} \cup \{g, \epsilon\}$

$= [\{d, g, h\} \cup \{h, g, \epsilon\}] \cup \{c\} \cup \{a, g\}.$

$= \{a, c, d, g, h, \epsilon\}.$

EQ-8

1). $E \longrightarrow E + T / T.$

$T \longrightarrow T * F / F.$

$F \longrightarrow id.$

2) $S \longrightarrow AbC / ad$

$A \longrightarrow eS / C.$

$C \longrightarrow f / p$

3) $S \longrightarrow aXb$

$X \longrightarrow q / i$

4) $S \longrightarrow ABD$

$A \longrightarrow a / BSB.$

$B \longrightarrow b / D.$

$D \longrightarrow d / \epsilon.$

5) $A \longrightarrow BCx / y$

$B \longrightarrow yA / \epsilon.$

$C \longrightarrow dAy / x$

6) $A \longrightarrow BCD.$

$B \longrightarrow hB / \epsilon.$

$C \longrightarrow Cq / q / Ch / i$

$D \longrightarrow B / \epsilon.$

7) $S \longrightarrow [SX] / a.$

$X \longrightarrow \epsilon / +SY / Yb.$

$Y \longrightarrow \epsilon / -SXc.$

8) $S \longrightarrow TS / [S]S / ,S / \epsilon.$

$T \longrightarrow (x)$

$X \longrightarrow TX / [X]X / \epsilon.$

9) $S \longrightarrow XZY / ZbY / Ya.$

$X \longrightarrow da / YZ$

$Y \longrightarrow q / \epsilon.$

$Z \longrightarrow b / \epsilon.$

10) $S \longrightarrow XS / dS / \epsilon.$

$X \longrightarrow Y / Zb / aY$

$Y \longrightarrow cZ$

$Z \longrightarrow e.$

1) ~~First (A) = First (~~ (

$A \longrightarrow .A\alpha/\beta.$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A'/\epsilon.$

$E \cdot \longrightarrow E+T/T.$

$T \longrightarrow T*F/F.$

$F \longrightarrow id.$

$E \longrightarrow .\cdot TE'$

$E' \longrightarrow . +TE'/\epsilon.$

$T \longrightarrow FT'$

$T' \longrightarrow *FT'/\epsilon.$

$F \longrightarrow id.$

First (F) = First (id) = {id}.

First (T') = First (*FT') ∪ First (ε)

= First (*) ∪ First (ε)

= {ε, *}.

First (T) = First (FT').

= First (F)

= {id}.

First (E) = First (TE')

= First (T)

= {id}.

First (E') = First (+TE') ∪ First (ε)

= First (+) ∪ First (ε)

= {+, ε}.

2) $S \rightarrow AbC/ad$

$A \longrightarrow eS/C$

$C \rightarrow f/p.$

First (C) = {f, p}.

First (A) = First (eS) ∪ First (C)

= First (e) ∪ First (C)

= {e, f, p}

First (S) = First (AbC) ∪ First (ad)

= First (A) ∪ First (a)

= {a, e, f, p}

3) $S \rightarrow aXb$

$X \rightarrow q/i$

First(X) = {q, i}.

First(S) = First(aXb)

= First(a)

= {a}.

4) $S \rightarrow ABD$.

$A \rightarrow a/BSB$.

$B \rightarrow b/D$

$D \rightarrow d/\epsilon$

First(D) = {d, ϵ}.

First(B) = First(b) ∪ First(D)

= {b} ∪ [First(D) − {ϵ} ∪ First(ϵ)]

= {b} ∪ {d} ∪ {ϵ}

= {b, d, ϵ}.

First(A) = First(a) ∪ First(BSB)

= {a} ∪ [First(B) − {ϵ} ∪ First

~~A → ABaAB / ϵ~~

BaP

Follow {a}.. {a}.

A → A'

A

First(aA) − {ϵ}

A → ABAaA

First(aA) = a

5) $A \longrightarrow BCx/y$

$B \longrightarrow yA/\epsilon$

$C \longrightarrow dAy/x$.

First(B) = First (yA) ∪ First (ϵ)

= First (y) ∪ $\{\epsilon\}$

= $\{y, \epsilon\}$.

First (C) = First (dAy) ∪ First (x)

= $\{$ First (d) ∪ First (x)

= $\{d, x\}$.

First (A) = First (BCx) ∪ First (y)

= [First $(B) - \{\epsilon\}$ ∪ First (Cx)] ∪ $\{y\}$

= [$\{y\}$ ∪ First (ϵ)] ∪ $\{y\}$

= $\{y, d, x\}$ ∪ $\{y\}$

= $\{d, x, y\}$.

6). $A \longrightarrow BCD$.

$B \longrightarrow hB/\epsilon$

$C \longrightarrow Cq/q/Ch/\epsilon i$

$D \longrightarrow B/\epsilon$.

$C \longrightarrow qC'/iC'$

$C' \longrightarrow qC'/hC'/\epsilon$.

First (B) = First (hB) ∪ First (ϵ)

= $\{\epsilon, h\}$.

First (D) = First (B) ∪ First (ϵ)

= $\{\epsilon, h\}$.

First (C') = First (qC') ∪ First (hC') ∪ First (ϵ)

= $\{q, h, \epsilon\}$.

First (C) = First (qC') ∪ First (iC')

= $\{q\}$ ∪ $\{i\}$

= $\{q, i\}$.

First (A) = First (BCD)

= First (B) - $\{\epsilon\}$ ∪ First (CD)

= $\{h\}$ ∪ First (C) - $\{\epsilon\}$ ∪ First (D)

= $\{h\}$ ∪ $\{q, i\}$ ∪ $\{\epsilon, h\}$ = $\{\epsilon, q, h, i\}$

7). $S \rightarrow [Sx] / a$.

$X \rightarrow \epsilon / +SY / Yb$

$Y \rightarrow \epsilon / -SX@c$

First(S) = First([Sx]) ∪ First(a)

= First([) ∪ First(a)

= {[, a}.

First(Y) = First(ϵ) ∪ First(−SXc)

= {ϵ} ∪ First(−) = {ϵ, −}.

First(X) = First(ϵ) ∪ First(+SY) ∪ First(Yb)

= {ϵ} ∪ First(+) ∪ [First(Y) − {ϵ} ∪ First(b)]

= {ϵ, +} ∪ [{−} ∪ {b}]

= {+, −, b, ϵ}.

8). $S \rightarrow TS / [S]S / , S / \epsilon$.

$T \rightarrow (X)$

$X \rightarrow TX / [X]X / \epsilon$.

First(T) = First((X))

= First(() = {(}

First(X) = First(TX) ∪ First([X]X) ∪ First(ϵ)

= First(T) ∪ First([) ∪ First(ϵ)

= {(, [, ϵ}.

First(S) = First(TS) ∪ First([S]S) ∪ First(,S) ∪ First(ϵ)

= First(T) ∪ First([) ∪ First(,) ∪ First(ϵ)

= {(, [, ,, ϵ}

9) $S \rightarrow XZY / ZbY / Ya.$

$X \rightarrow da / YZ$

$Y \rightarrow q / \epsilon$

$Z \rightarrow h / \epsilon$

First(Z) : $\{h, \epsilon\}$

First(Y) : $\{q, \epsilon\}$

First(X) : First(da) $\cup$ First(YZ)

$= First(d) \cup [First(Y) - \{\epsilon\} \cup First(Z)]$

$= \{d\} \cup \{q\} \cup \{h, \epsilon\}$

$= \{d, q, h, \epsilon\}$

First(S) = First(XZY) $\cup$ First(ZbY) $\cup$ First(Ya)

$= [First(X) - \{\epsilon\} \cup First(Z) - \{\epsilon\} \cup First(Y)]$

$\cup [First(Z) - \{\epsilon\} \cup First(bY)] \cup [First(Y) - \{\epsilon\} \cup First(a)]$

$= [\{d, q, h\} \cup \{h\} \cup \{q, \epsilon\}] \cup [\{h\} \cup \{b\}]$

$\cup [\{q\} \cup \{a\}]$

$= \{d, q, h, \epsilon\} \cup \{h, b\} \cup \{q, a\}$

$= \{a, b, d, q, h, \epsilon\}$.

10). $S \rightarrow Xs / ds / \epsilon$

$X \rightarrow Y / Zb / aY$

$Y \rightarrow cZ$

$Z \rightarrow e$.

First(Y) = First (cZ)
= {c}

First(Z) = First (e)
= {e}

First(X) = First(Y) ∪ First (Zb) ∪ First (aY)
= {c} ∪ {e} ∪ {a}.
= {a, e, c}.

First(S) = First (XS) ∪ First (dS) ∪ First (ϵ)
= First(X) ∪ First (d) ∪ {ϵ}
= {a, c, d, e, ϵ}.

follow (S)

Follow (A).

07/02/2023

(1) Place '$ ' in follow (S). where 'S' is the start symbol and '$' is the input right endmarker.

(2) If there is a production A → αBβ, then add everything found in first (β) except 'ϵ'. (terminal)

(3) If there is a production A → αB or A → αBβ and β contains 'ϵ', then add everything in ~~follow (B) to~~ Follow (A). to Follow (B)

(non-terminal)

	First .	Follow .
$S \longrightarrow aAb/bB.$	$\{a,b\}$	$\{\$\}$
$A \longrightarrow aB.$	$\{a\}$	$\{b\}$
$B \longrightarrow b.$	$\{b\}$	$\{\$, b\}.$

	First	Follow
$S \rightarrow aABb$	$\{a\}$	$\{\$\}$
$A \rightarrow c/\epsilon.$	$\{c, \epsilon\}$	$\{b, d\}$
$B \rightarrow d/\epsilon.$	$\{d, \epsilon\}$	$\{b\}.$

$$\text{Follow}(A) = \text{First}(Bb) - \{\epsilon\}$$
$$= [\text{First}(B) - \{\epsilon\} \cup \text{First}(b)] - \{\epsilon\}$$
$$= \{b, d\}$$

$$\text{Follow}(B) = \text{First}(b) - \{\epsilon\}.$$
$$= \{b\}.$$

	First .	Follow
$S \longrightarrow aBDh.$	$\{a\}$	$\{\$\}$
$B \longrightarrow cC.$	$\{c\}$	$\{f, g, h\}$
$C \longrightarrow bC/\epsilon$	$\{b, \epsilon\}$	$\{f, g, h\}.$
$D \longrightarrow EF.$	$\{g, f, \epsilon\}$	$\{h\}.$
$E \longrightarrow g/\epsilon.$	$\{g, \epsilon\}.$	$\{h, f\}$
$F \longrightarrow f/\epsilon.$	$\{f, \epsilon\}$	$\{h\}.$

08/02/2023

	First	Follow
$E \rightarrow TE'$	$\{id\}$	$\{\$,)\}$
$E' \rightarrow +TE'/\epsilon$	$\{+, \epsilon\}$	$\{\$,)\}$
$T \rightarrow FT'$	$\{id\}$	$\{+, \$\}$
$T' \rightarrow *FT'/\epsilon$	$\{\epsilon, *\}$	$\{+, \$\}$
$F \rightarrow id / (E)$	$\{id, (\}$	$\{+, *, \$\}$

Follow

E. — $\{\$,)\}$

E' — $\{\$,)\}$

T — $\{\$,), +\}$

T' — $\{\$,), +\}$

F. — $\{+, *, \$,)\}$

Q.

	First	Follow
$S \rightarrow ACB / CbB / Ba$	$\{a, c, d, g, h, \epsilon\}$	$\{\$\}$
$A \rightarrow da / BC$	$\{d, g, h, \epsilon\}$	$\{h, g, \$\}$
$B \rightarrow g / \epsilon$	$\{g, \epsilon\}$	$\{\$, a, h, g\}$
$C \rightarrow h / \epsilon$	$\{h, \epsilon\}$	$\{g, h, \$\}$

2)

	First	Follow
$S \rightarrow ABC / ad$	$\{a, e, f, p\}$	$\{\$, b\}$
$A \rightarrow eS / Cr$	$\{e, f, p\}$	$\{b\}$
$C \rightarrow f / p$	$\{f, p\}$	$\{r, \$, b\}$

3)

	First	Follow
$S \rightarrow aXb$	$\{a\}$	$\{\$\}$
$X \rightarrow q / i$	$\{q, i\}$	$\{b\}$

5)

	First	Follow
$A \rightarrow BCx / y$	$\{d, x, y\}$	$\{\$, d, x, y\}$
$B \rightarrow yA / \epsilon$	$\{y, \epsilon\}$	$\{d, x\}$
$C \rightarrow dAy / x$	$\{d, x\}$	$\{x\}$

6)

	First	Follow
$A \rightarrow BCD$	$\{q, h, i\}$	$\{\$\}$
$B \rightarrow hB / \epsilon$	$\{\epsilon, h\}$	$\{q, i, h, \$\}$
$C \rightarrow qC' / iC'$	$\{q, i\}$	$\{h, \$\}$
$C' \rightarrow qC' / hC' / \epsilon$	$\{q, h, \epsilon\}$	$\{h, \$\}$
$D \rightarrow B / \epsilon$	$\{\epsilon, h\}$	$\{\$\}$

7)

	First	Follow
$S \rightarrow [SX] / a$	$\{[, a\}$	$\{\$, -, +, b, c,]\}$
$X \rightarrow \epsilon / +SY / Yb$	$\{+, -, b, \epsilon\}$	$\{e,]\}$
$Y \rightarrow e / -SXc$	$\{\epsilon, -\}$	$\{c,], b\}$

8)

	First	Follow
$S \rightarrow T S / [S] S / , S / \epsilon$.	{c, [, , , ϵ}.	{$}
$T \rightarrow (X)$	{c}	{c, [, , , $,)}
$X \rightarrow T X / [X] X / \epsilon$.	{c, [, ϵ}	{],)}

9) $S \rightarrow XZY / ZbY / Ya$.
$X \rightarrow da / YZ$
$Y \rightarrow q / \epsilon$.
$Z \rightarrow h / \epsilon$.

First	Follow
{a, b, d, q, h, ϵ}	{$}
{d, q, h, ϵ}	{q, h, $}
{q, ϵ}	{$, a, h, q}
{h, ϵ}	{q, h, $, b}

10) $S \rightarrow XS / dS / e$
$X \rightarrow Y / Zb / aY$
$Y \rightarrow cZ$
$Z \rightarrow e$

First	Follow
{a, c, d, e, ϵ}	{$}
{a, e, c}	{a, c, d, e, $}
{c}	{a, c, d, e, $}
{e}	{b, a, c, d, e, $}.

( )

S

T

X

Algorithm for Construction of Predictive Parsing Table.

For each production $A \longrightarrow \alpha$ do the following:

(i). For each terminal 'a' in First(A), add $A \longrightarrow \alpha$ to $M[A, \alpha]$

(ii). If 'ϵ' is in First(α), then for each terminal 'b' in Follow(A), add $A \longrightarrow \alpha$ to $M[A, b]$.
If 'ϵ' is in First(α), then and '$\$$' is in Follow(b) add $A \longrightarrow \alpha$ to $M[A, \$]$ as well.

Note: If after performing the above, there is no production at any cells of $M[A, \alpha]$ then set the cells as error.

$E \longrightarrow E+T / T$
$T \longrightarrow T*F / F$.
$F \longrightarrow id / (E)$.

	First	Follow
$E \longrightarrow TE'$	{id, (}	{$,)}.
$E' \longrightarrow +TE' / \epsilon$	{+, ϵ}	{$,)}.
$T \longrightarrow FT'$	{id, (}	{+, $,)}
$T' \longrightarrow *F.T' / \epsilon$.	{*, ϵ}	{+, $,)}
$F \longrightarrow id / (E)$	{id, B ({	{*, +, $,)}

Table contains all terminal symbols in First, n Follow excluding ε, including $. (columns) Rows — all non-terminal symbols in modified grammar.

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε.
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε.
F.	F → id			F → (E)		

If exactly one entry for each production, in each cell of parsing table — then the grammar is in LL1.

If

$A \longrightarrow \alpha_1 / \alpha_2 / \dots / \alpha_n$.

$First(\alpha_1) \cap First(\alpha_2) \cap \dots First(\alpha_n) = \phi$.

. First is disjoint set of

} The given grammar is LL1 (available for parsing).

$A \longrightarrow \alpha / \epsilon$.

$First(\alpha) \cap$ or $Follow(A) = \phi$.

$A \longrightarrow \alpha_1 / \alpha_2 / \alpha_3 / \alpha_4 / \dots / \alpha_n / \epsilon$.

$First(\alpha_1) \cap First(\alpha_2) \cap \dots \cap First(\alpha_n) \cap Follow(A) = \phi$.

S → AB.	{a, b, ϵ}	{$}
A → a/ϵ.	{a, ϵ}	{b, $}
B → b/ϵ	{b, ϵ}	{$}

A → a/ϵ.

First(a) ∩ Follow(A)

= {a} ∩ {b, $} = ϕ.

B → b/ϵ.

First(b) ∩ Follow(B)

= {b} ∩ {$} = ϕ.

6.

	First	Follow
$A \rightarrow BCD$	$\{e, q, h, i\}$	$\{\$\}$
$B \rightarrow hB/\epsilon$.	$\{\epsilon, h\}$	$\{q, i, h, \$\}$
$e \rightarrow qC'/iC'$	$\{q, i\}$	$\{h, \$\}$
$C' \rightarrow qC'/hC'/\epsilon$.	$\{q, h, \epsilon\}$	$\{h, \$\}$
$D \rightarrow B/\epsilon$.	$\{\epsilon, h\}$	$\{\$\}$

	h	q	i	$
A	A → BCD.	A → BCD	A → BCD.	
B.	B → hB.	B → ϵ.	B → ϵ.	B → ϵ.
C		C → qC'	C → iC'	
C'	C' → ~~ϵ~~C'	C' → qC'		C' → ~~$~~ϵ.
D.	D → B.			D → ϵ.

8.

	First	Follow
$S \rightarrow TS/[S]S/,S/\epsilon$.	{C, [, , , ϵ}	{$}
$T \rightarrow (X)$	{C}	{C, [, , , $,)}
$X \rightarrow TX/[X]X/\epsilon$.	{C, [, ϵ}	{)}.

	[	q	]	,	(	)	$
S	S→[S]S ~~S→qS~~	~~S→qS~~		S→,S	S→T$		S→ϵ.
T					T→(X)		
X	X→[X]X				X→TX	X→ϵ.	

(a)

		First	Follow
Q1.	$S \rightarrow (L) / a$	$\{(, a\}$	$\{\$, , ,)\}$.
	$L \rightarrow SL'$	$\{(, a\}$	$\{)\}$
	$L' \rightarrow , SL' / \epsilon$.	$\{, , \epsilon\}$	$\{)\}$

	(	)	a	,	$
S	$S \rightarrow (L)$		$S \rightarrow a$.		
L	$L \rightarrow SL'$		$L \rightarrow SL'$		
L'		$L' \rightarrow \epsilon$.		$L' \rightarrow , SL'$	

Q2. $S \rightarrow aSbS / bSaS / \epsilon$.

First(S) = $\{a, b, \epsilon\}$.

Follow(S) = ~~$\{\$\}$~~ $\{\$, b, a\}$

	a	b	$
S.	$S \rightarrow aSbS$ $S \rightarrow \epsilon$	$S \rightarrow bSaS$ $S \rightarrow \epsilon$	$S \rightarrow \epsilon$

Q3. $S \longrightarrow aSA / bS / c$.

First(S): {a, b, c}

Follow(S): {$, a}

First(A) = {a}

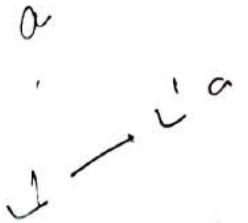
Follow(A) = {$, a}

$S \longrightarrow aSa / bS / c$

abSa

Q4.

	First	Follow
$S \longrightarrow iC + SS_1 / a$	{i, a}	{$, }
$S_1 \longrightarrow eS / \epsilon$	{e, ϵ}	
$C \longrightarrow b$	{b}	{+}



Recursive Descent Parser.

21/02/2023

Q. $E \longrightarrow iE'$

$E' \longrightarrow +iE/\epsilon.$

```
E()
{
  if (l == 'i')
  {
    match('i');
    E'();
  }
}
```

```
E'()
{
  if (l == '+')
  {
    match('+');
    match('i');
    E();
  }
  else
  return;
}
```

```
match(char t)
{
  if (t == l)
  {
    t = getchar();
  }
  else
  {
    printf("Error");
  }
}
```

```
main()
{
  E();
  if (l == '$')
  {
    printf("Parsing
           Complete");
  }
}
```

Bottom – Up Parser

Bottom-up can generate string from left recursive grammar, ambiguous Grammer. They are more powerful than Top Down Approach.

From a String, if we can generate starting symbol (root) — Bottom Up Parser.

Shift-Reduce Parser.

(i) Shift.
(ii) Reduce
(iii) Accept.

} Three Actions.

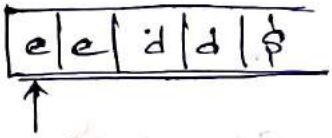
— Stack
— input buffer.

} Data Structures.

$S \rightarrow cc$
$C \rightarrow cc$
$C \rightarrow d$

Stack

I/P.



(i) Shift the input symbol onto the Stack, until the handle is on the top of the Stack.

(ii) After getting handle Reduce the string to left side of production.

(iii) (i) and (ii) will be repeated until we get error or Stack contains $S, starting symbol, and input buffer contains $.

↳ then accept.

Handle → substring which is a part of production. (Right hand side)

d, cc, cc are handles.

Stack	Input	Action
$	ccdd$	Shift
$c	cdd$	Shift.
$cc	dd$	Shift.
$ccd	d$	Reduce by C → d.
$ccC	d$	Reduce by C → cc
$cC	d$	Reduce by C → cc
$C	d$	Shift.
$Cd	$	Reduce by C → d
$cc	$	Reduce by S → cc.
$S	$	Accept.

Q.

T → T+T
T → T-T
T → c.

Input : $c_1 + c_2 - c_3$ $c + c - c$

Stack	Input	Action
$	$c_1+c_2-c_3$$	Shift.
$c1	$+c_2-c_3$$	Reduce T → c_1
$T	$+c_2-c_3$$	Shift.
$T+	$c_2-c_3$$	Shift.
$T+c2	$-c_3$$	Reduce T → c_2
$T+T	$-c_3$$	Reduce T → T+T
$T	$-c_3$$	Shift.
$T-	$c_3$$	Shift.
$T-c3	$$	Reduce T → c_3

$T-T	$	Reduce T → T-T
$T	$	Accept.

6. A → 5A5 / 7A7 / 9

75957.

Q. E → E+E / E*E / (E) / id

id* (id+id)

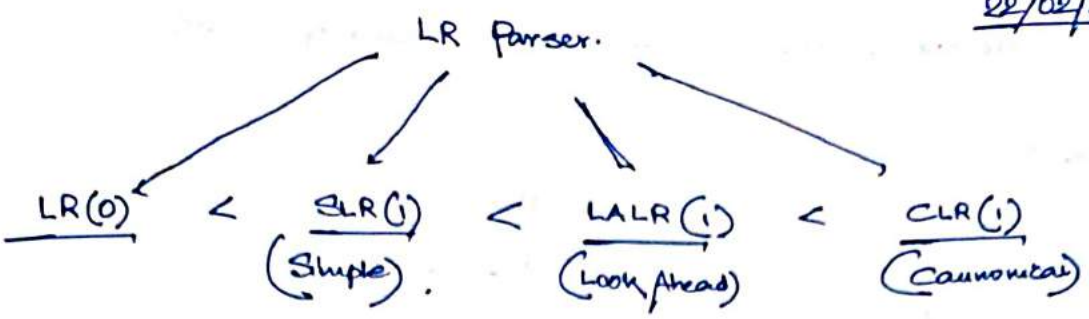
Stack	Input	Action
$	75957$	Shift.
$7	5957$	Shift.
$75	957$	Shift.
$759	57$	Reduce A → 9.
$75A.	57$	Shift.
$75A5	7$	Reduce A → 5A5
$7A.	7$	Shift.
$7A7	$	Reduce A → 7A7
$A	$	Accept.

Q.

Stack	I/p	Action
$	id*(id+id)$	Shift.
$id	*(id+id)$	Reduce E → id
$E	*(id+id)$	Shift.
$E*	(id+id)$	Shift.
$E*(	id+id)$	Shift.
$E*(id	+id)$	Reduce E → id
$E*(E	+id)$	Shift.
$E*(E+	id)$	~~Reduce E → id~~ Shift.
$E*(E+id	)$	Reduce E → id
$E*(E+E	)$	Reduce E → E+E.
$E*(E	)$	Shift.
$E*(E)	$	Reduce E → (E).
$E*E	$	Reduce E → E*E
$E	$	Accept.

S.

22/02/2023



L — Left to Right Parsing.

R — Right most derivation in Reverse.

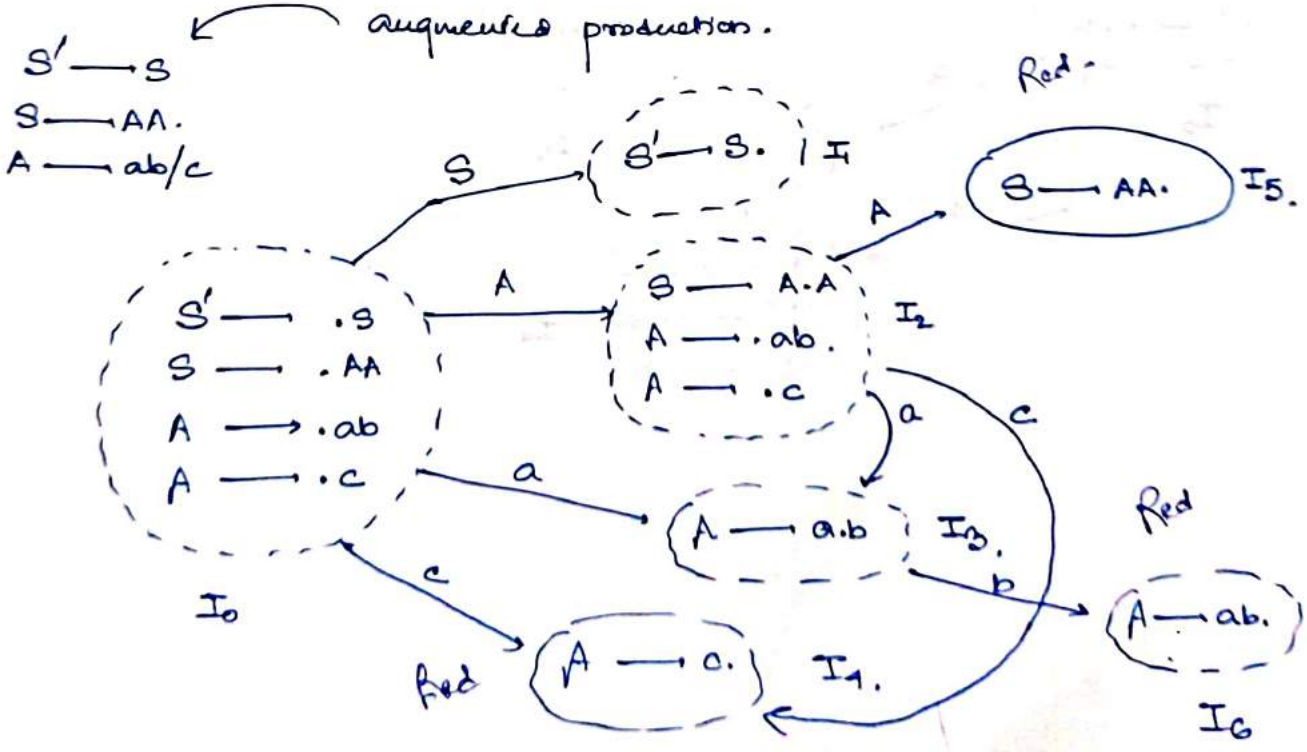
(1) — Look Ahead parser.

LR(0) parser.

— use of augmented production. in LR(0) parser. ← virtual production to be introduced.

In bottom - up approach. the whole input may not be processed, but still we can reach to the start symbol. So in case of these problems, augmented production to start symbol is used.

After introducing augmented productions, the grammar is known as Augmented Grammar.

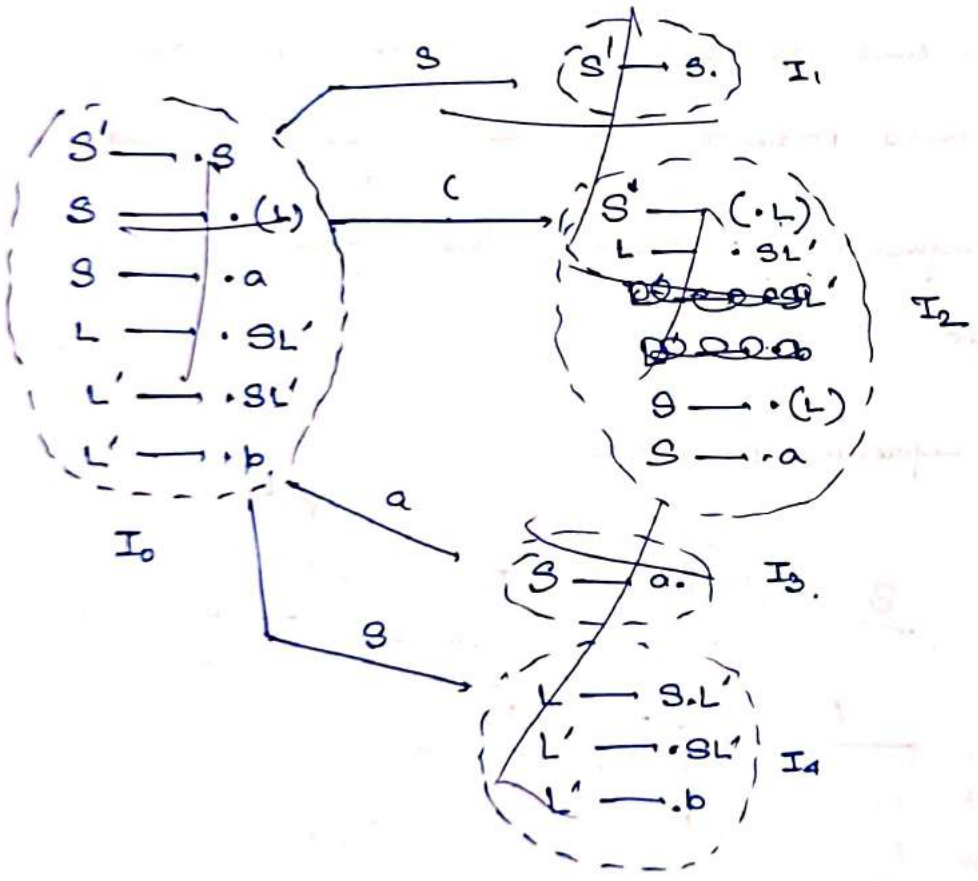


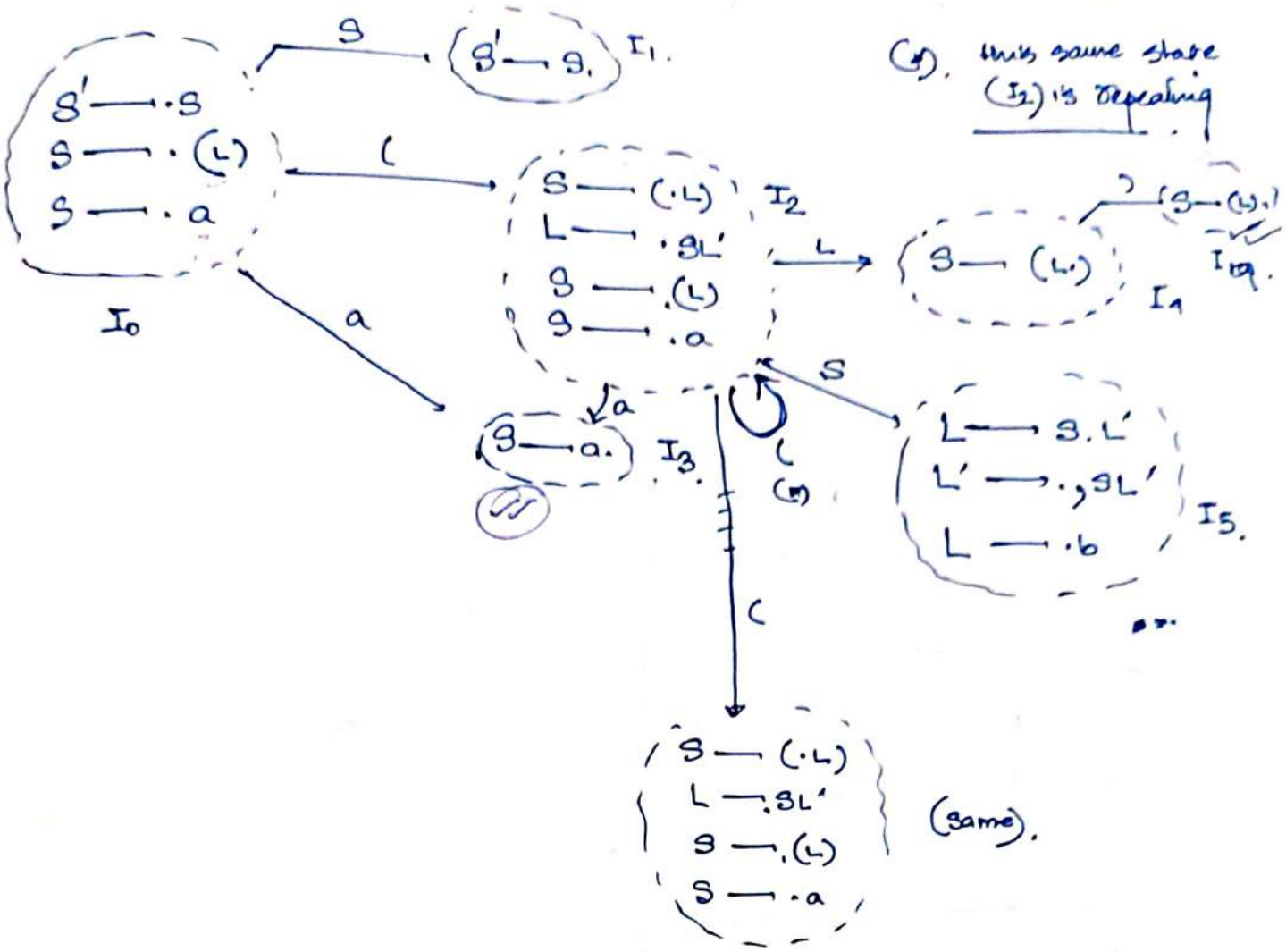
I_0 - I_6 are known as canonical items of LR(0).

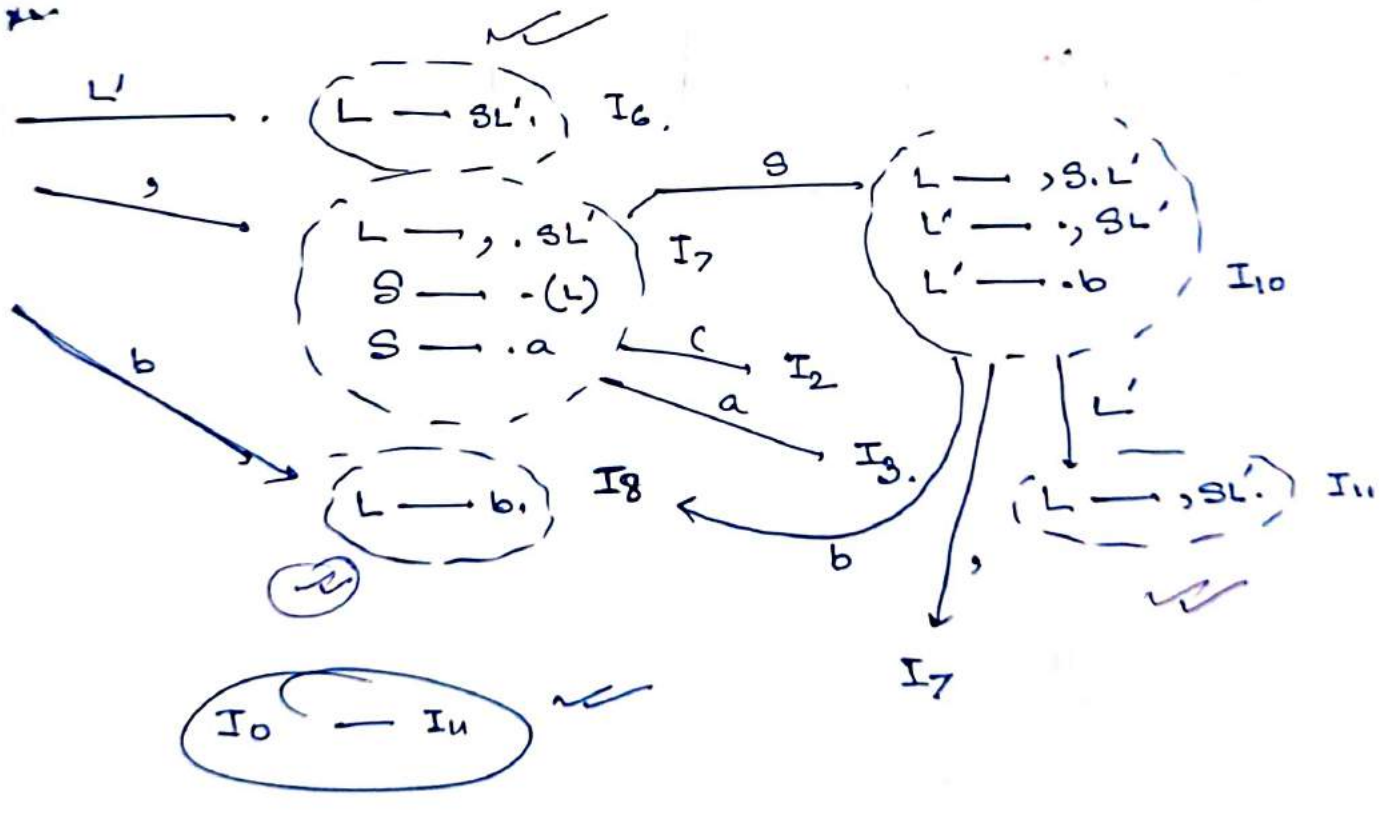
S' — S
S — (L)/a
L — SL'
L' — ,SL'/b

E — E+T/T.
T — (E)/id

S' — S
S — (L)/a
L — SL'
L' — ,SL'/b




S
S' → S.
I1.
S' → .S
S → .(L)
S → .a
I0
(
S → (.L)
L → .SL'
S → .(L)
S → .a
I2
L
S → (L.)
I4.
)
S → (L).
I5.
a
S → a.
I3.
S
L → S.L'
L' → .,SL'
L → .b
I5.
(
(same)
C


L'
L → SL'.
I6.
,
L → ,.SL'
S → .(L)
S → .a
I7
S
L → ,S.L'
L' → .,SL'
L' → .b
I10
b
C
I2
a
I3.
L → b.
I8
b
L'
L → ,SL.
I11
,
I7
I0 — I4

$S \rightarrow \dot{W}A\check{B} / A\check{B}C\check{S}$

$A \rightarrow \check{B} / \dot{W}\check{B}$

$B \rightarrow y\check{B} / \epsilon$

$C \rightarrow z$

$W \rightarrow x.$

First
$\{x, y, z\}$
$\{y, \epsilon, x\}$
$\{y, \epsilon\}$
$\{z\}$
$\{x\}$

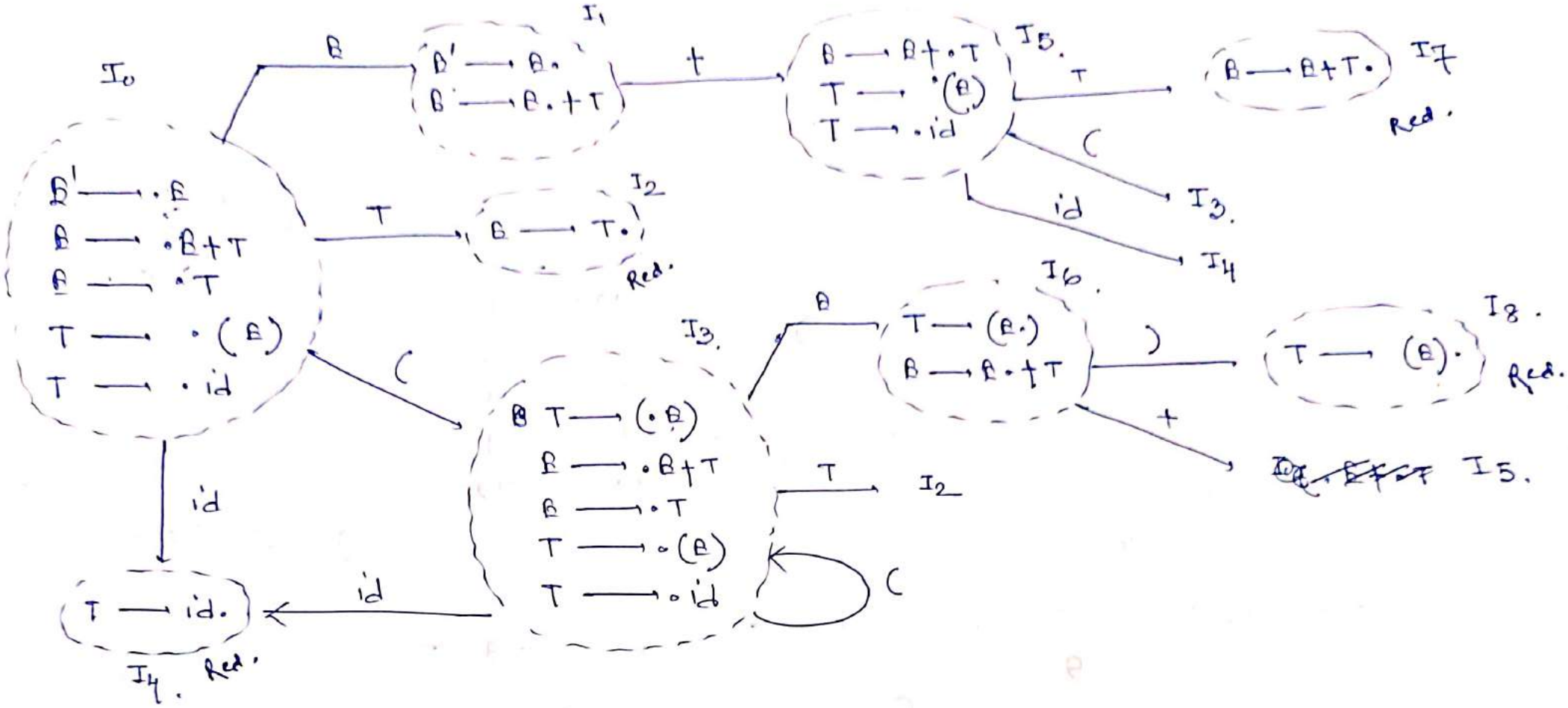
Follow
$\{\$\}$
$\{y, \$, z\}$
$\{\$, z, y\}$
$\{x, y, z\}$
$\{y, x, \$, z\}$

	x	y	z	$\$$
S	$S \rightarrow WAB$ $S \rightarrow ABCS$	$S \rightarrow ABCS$	$S \rightarrow ABCS$	
A	$A \rightarrow WB.$	$A \rightarrow B.$ $A \rightarrow B,$	$A \rightarrow B.$	$A \rightarrow B.$
B		$B \rightarrow yB.$ $B \rightarrow \epsilon$	$B \rightarrow \epsilon$	$B \rightarrow \epsilon.$
C			$C \rightarrow z$	
W	$W \rightarrow x.$			

Bottom-up parser (LR(0))

E → E + T / T
T → (E) / P1

E' → E .
E → E + T / T
T → (E) / P1

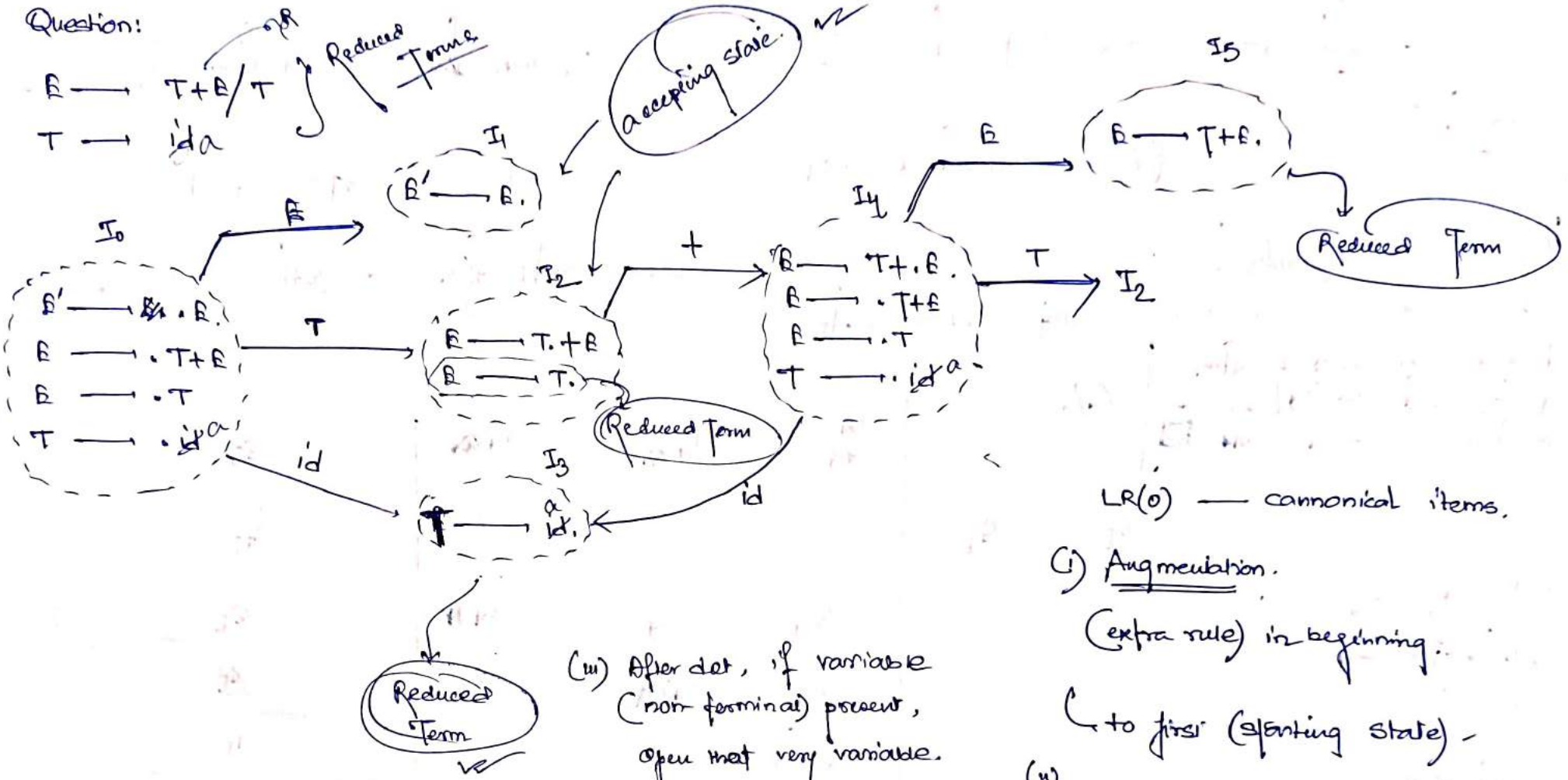

I0
E' → .E
E → .E+T
E → .T
T → .(E)
T → .id
E
I1
E' → E.
E → E.+T
+
I5.
E → E+.T
T → .(E)
T → .id
T
I7
E → E+T.
Red.
(
I3.
id
I4
T
I2
E → T.
Red.
(
I3.
T → (.E)
E → .E+T
E → .T
T → .(E)
T → .id
E
I6.
T → (E.)
E → E.+T
)
I8.
T → (E).
Red.
+
I5.
T
I2
(
id
I4.
T → id.
Red.
id

LR(0).

Question:

E → T+E / T OR Reduced Terms

T → id a



LR(0) — cannonical items.

(i) Augmentation.

(extra rule) in beginning.

↳ to first (starting state).

(ii)

(a) Dot indicates whether a parser has parsed a symbol or that.

(before the dot symbols, are already parsed by parser).

(after id. dot, the symbols are not parsed).

(iii) After dot, if variable (non-terminal) present, open that very variable.

If terminal after dot, no need for open (stop).

↳ After dot check for, if next symbol after dot is terminal/non-terminal

Items $(I_0 - I_5)$ ——— 6 Canonical Items.

State (Items)	Action Parts			Goto Part	
	id	+	$	E	T
I_0	S_3			I_1	I_2
I_1			accept		
I_2	R_2	S_4/R_2	R_2		
I_3	R_3	R_3	R_3		
I_4	S_3			I_5	I_2
I_5	R_1	R_1	R_1		

all terminals including $ are in (Action parts) -
all variables or (non-terminals) are in GoTo part.

Productions

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

Here there is a multiple entry or (S_4/R_2) in parsing table. Shift + reduce in action part

(R/R)

Not LR(0) parser

$I_0 \xrightarrow{id} S_3$ (Shift 3) — Shift Item 3 ✓

for terminals — Shift ✓

$I_2 \xrightarrow{+} S_4$ (Shift 4) — shift Item 4.

I_1 — accept state, since the production is present by default

Reduced production (R_i)

(I_3, I_5, part of I_2 are reduced states) ✓

Question:

$E \longrightarrow E+T \;/\; T$ (1, 2)

$T \longrightarrow (E) \;/\; id$ (3, 4)

$E' \longrightarrow E$

$E \longrightarrow E+T \;/\; T$

$T \longrightarrow (E) \;/\; id$

State (Items)	Action Part					GoTo Part	
	+	(	)	id	$	E	T
I_0		S_3		S_4		1	2
I_1	S_5				accept		
I_2	R_2	R_2	R_2	R_2	R_2		
I_3		S_3		S_4		6	2
I_4	R_4	R_4	R_4	R_4	R_4		
I_5		S_3		S_4			7
I_6	S_5		S_8				
I_7	R_1	R_1	R_1	R_1	R_1		
I_8	R_3	R_3	R_3	R_3	R_3		

The above grammar is LR(0) — (each cell has one entry)

LR(0) parsing table

S → S
S → AA.
A → ab
A → c

	Action (terminal)				Goto (non-terminal)	
	a	b	c	$	S	A
0	S_3		S_4		1	2
1				accept		
2	S_3		S_4			5
3		S_6				
4	R_3	R_3	R_3	R_3		
5	R_1	R_1	R_1	R_1		
6	R_2	R_2	R_2	R_2		

S → (L)
S → a
L → SL'
L' → ,SL'
L' → ε

	Action (terminal)						Goto (non-terminal)		
	(	)	a	,	b	$	S	L	L'
0	S_2		S_3				1		
1						accept			
2	S_2		S_3				5	4	
3	R_2	R_2	R_2	R_2	R_2	R_2			
4		S_9							
5				S_7	S_8				6
6	R_3	R_3	R_3	R_3	R_3	R_3			
7	S_2		S_3				10		
8	R_5	R_5	R_5	R_5	R_5	R_5			
9	R_1	R_1	R_1	R_1	R_1	R_1			
10				S_7	S_8				11
11	R_4	R_4	R_4	R_4	R_4	R_4			

	First	Follow
S → A.		
A → aB / Ad	{b}	
B → b.	{g}	
C → g		

S → A.
A → aB / Ad
B → b.
C → g.

A → Aα / β
A → βA'
A' → αA' / ϵ.

S → A'.
A → aBA'
A' → dA' / ϵ.
B → b.
C → g

	First	Follow
S → A'.	{a}	{$}
A → aBA'	{a}	{$}
A' → dA' / ϵ.	{d, ϵ}	{$}
B → b.	{b}	{d, $}
C → g	{g}	NA.

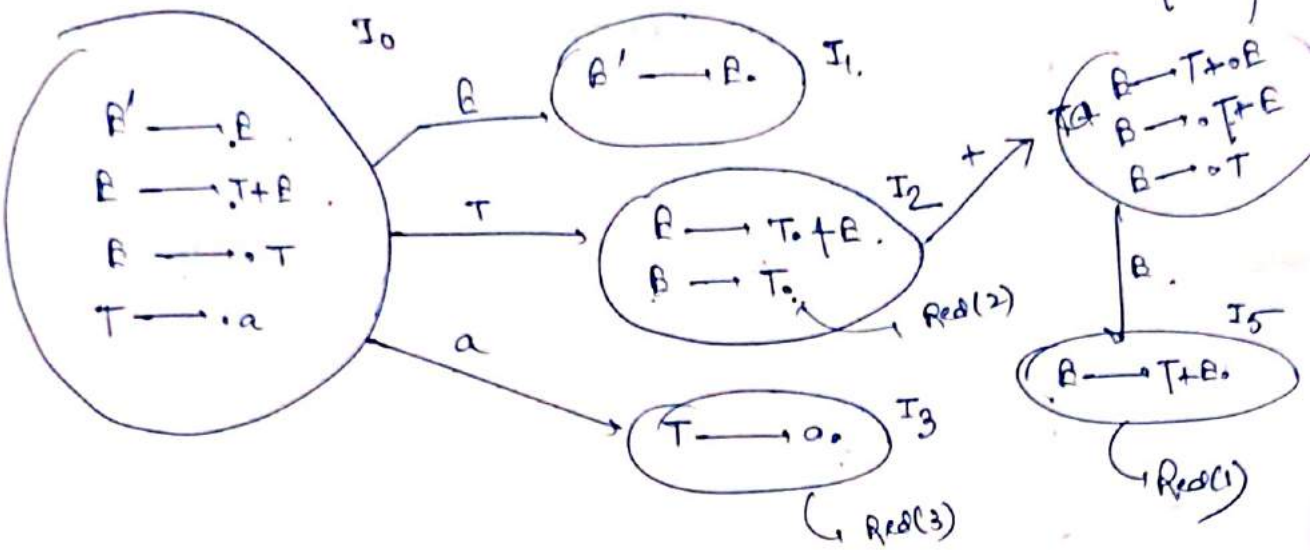
Parsing Table

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

2nd Question

E → T+E / T
T → a

E' → E
E → T+E / T
T → a



Parsing Table

	+	a	$	E	T
0		S_3		1.	2
1			accept		
2	S_4/R_2	R_2	R_2		
3	R_3	R_3	R_3		
4				5.	2
5	R_1	R_1	R_1		

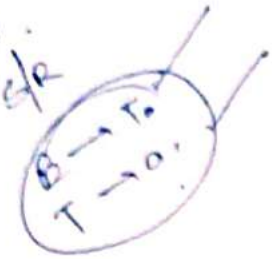
LR(0) Parsing Table

S_4/R_2 ——— conflict b/w S/R

So not LR(0) parser.

in one cell, there are multiple actions.

R/P c/R



LR(0) is blindly Reducing in every terminals.

↳ not depending upon look-ahead ✓

→ Simple Reduce Parser

→ So to avoid this conflict S/R, R/R.

we use SLR.(1). (Same) ✓

↳ Reduce in terminal

↳ for any production, we should find follow for that production.

↳ for that follow, R2 in only those ∮ items.

LR(0) Reduction blindly in all terminals

for the follow of the Non terminal

09/03/2023

	(1) (2)	First	Follow
E → ~~E+T~~.	T+E / T	{a}	{$}
T → a (3)		{a}	{+, $}

Parsing Table

	+	a	$	E	T
0		S3.		1	2
1			accept		
2	S4		R2		
3	R3		R3		
4				5	2
5			R1		

✓ SLR(1) Parsing ✓

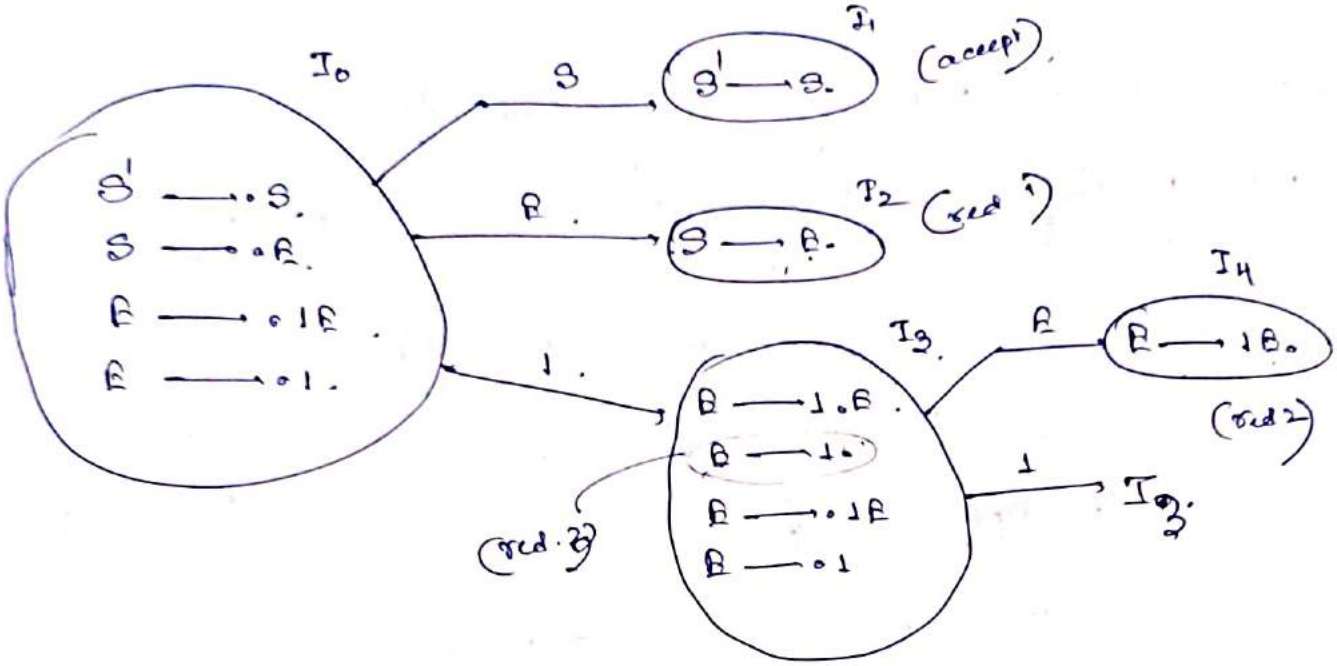
① ② ③

S → E

E → 1E / 1.

E → 1

First	Follow
{1}	{$}
{1}	{$}



LR(0) parsing table.

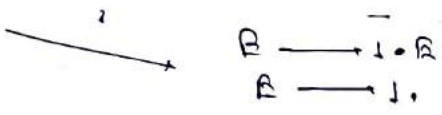
Item/State	1	$	S	E.
0	S_3.		1.	2
1		accept.		
2	R_1	R_1.		
3	S_3/R_3	R_3.		4.
4	R_2	R_2		

Not LR(0) — S/R conflict.

SLR(1)

Item.	A	$	A	B.
0	S_2		1.	2
1		accept.		
2		R_1		
3	S_3	R_3		4
4.		R_2		

SLR(1) ~ .

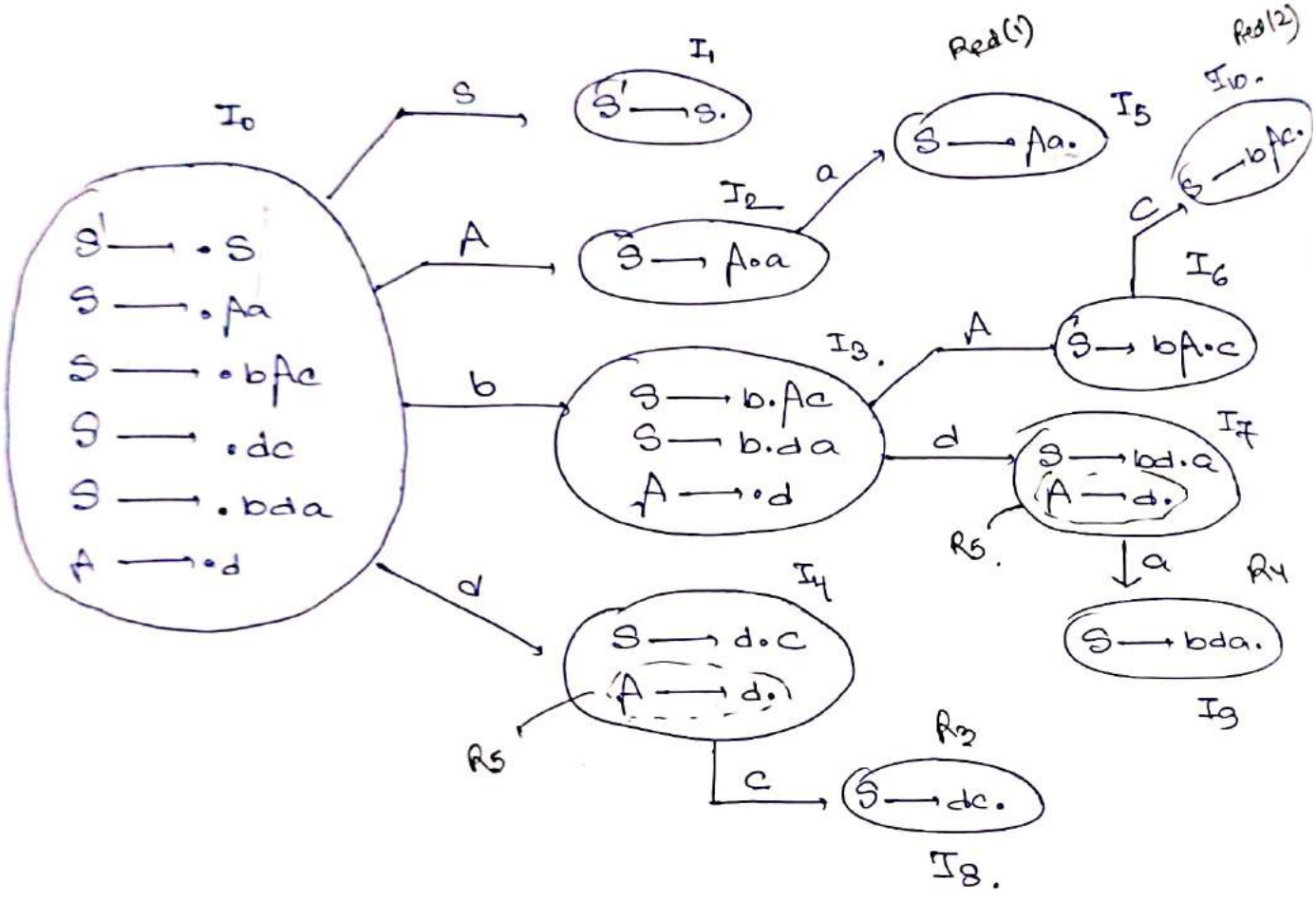


$S \rightarrow Aa / bAc / dc / bda$

$A \rightarrow d$

Check LR(0), SLR(1).

	First	Follow
$S \rightarrow Aa / bAc / dc / bda$	{d, b}	{$}
$A \rightarrow d$	{d}	{a, c}



LR(0).

State/Item	Action a	b	c	d	$	GoTo S	A.
0		S_3		S_4		1.	2
1					accept.		
2	S_5.						
3				S_7			6
4.	R_5	R_5	S_8/R_5	R_5	R_5.		
5	R_1	R_1	R_1	R_1	R_1		
6			S_{10}				
7	S_9/R_5	R_5	R_5	R_5	R_5		
8	R_3	R_3	R_3	R_3	R_3		
9	R_4	R_4	R_4	R_4	R_4		
10	R_2	R_2	R_2	R_2	R_2		

NOT LR(0) ✓ NOT SLR(1) ✓

SLR(1).

State/Item	a	b	c	d	$	S	A.
0		S_3		S_4		1	2
1					accept		
2	S_5						
3				S_7			6
4	R_5		S_8/R_5				
5					R_1		
6			S_{10}				
7	S_9/R_5		R_5				
8					R_3		
9					R_4		
10					R_2		

20/03/2023

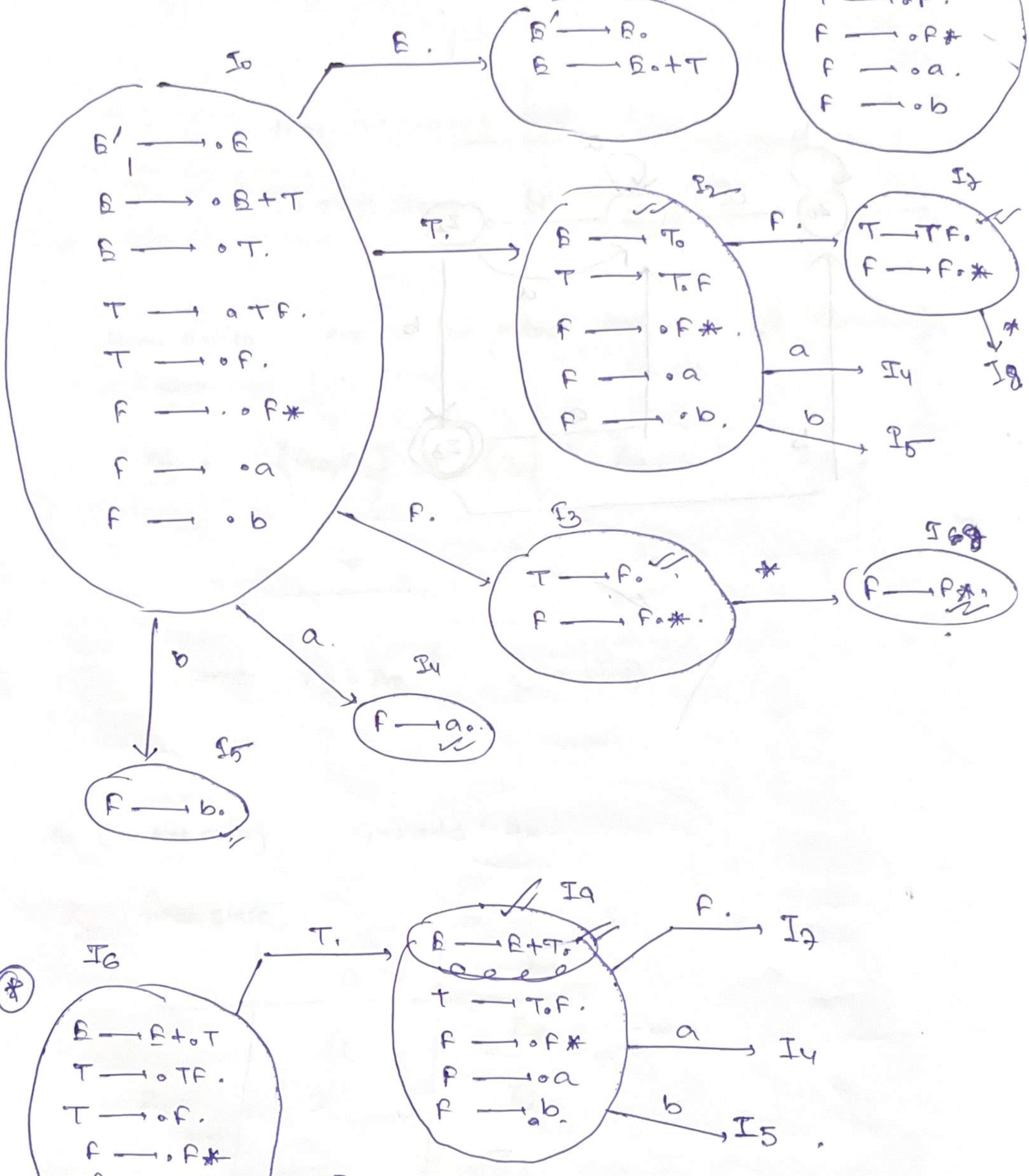
$E' \longrightarrow E$ (1)

$E \longrightarrow E+T$ (2) / T (3)

$T \longrightarrow TF$ (4) / F (5)

$F \longrightarrow f*$ (6) / a (7) / b (8)

	First	Follow
E'	{a,b}	{$}
E	{a,b}	{$,+}
T	{a,b}	{$,+,a,b}
F	{a,b}	{$,+,a,b,*}



Not LR(0)

SLR(1) ✓

State	Action					Goto		
	+	*	a	b	$	E	T	F
0			s_4		s_5	1	2	3
1	s_6				accept			
2	R_2	R_2	R_2/s_4		R_2/s_5	R_2		
3	R_4	R_4/s_8	R_4		R_4	R_4		
4	R_6	R_6	R_6		R_6	R_6		
5	R_7	R_7	R_7		R_7	R_7		
6			s_4		s_5		9	3
7	R_3	R_3/s_8	R_3		R_3	R_3		
8	R_5	R_5	R_5		R_5	R_5		
9	R_1		R_1/s_4		R_1/s_5	R_1		7

SLR(1)

State	Action					Goto		
	+	*	a	b	$	E	T	F
0			s_4		s_5	1	2	3
1	s_6				accept			
2	R_2		s_4		s_5	R_2		7
3	R_4	s_8	R_4		R_4	R_4		
4	R_6	R_6	R_6		R_6	R_6		
5	R_7	R_7	R_7		R_7	R_7		
6			s_4		s_5		9	3
7	R_3	s_8	R_3		R_3	R_3		
8	R_5	R_5	R_5		R_5	R_5		
9	R_1		s_4		s_5	R_1		7

(1) (2)

$A \longrightarrow BaBb \mid CbCa$

$B \longrightarrow \epsilon$. (3)

$C \longrightarrow \epsilon$. (4)

Augmented production:

$A' \longrightarrow A$.

New Grammar: $A' \longrightarrow A$

$A \longrightarrow BaBb \mid CbCa$

$B \longrightarrow \epsilon$.

$C \longrightarrow \epsilon$.

	First	Follow
$S \longrightarrow BB$	{a, b}	{$}
$B \longrightarrow aB \mid b$	{a, b}	{a, b, $}
$B \longrightarrow b$.		

$S' \rightarrow .S$

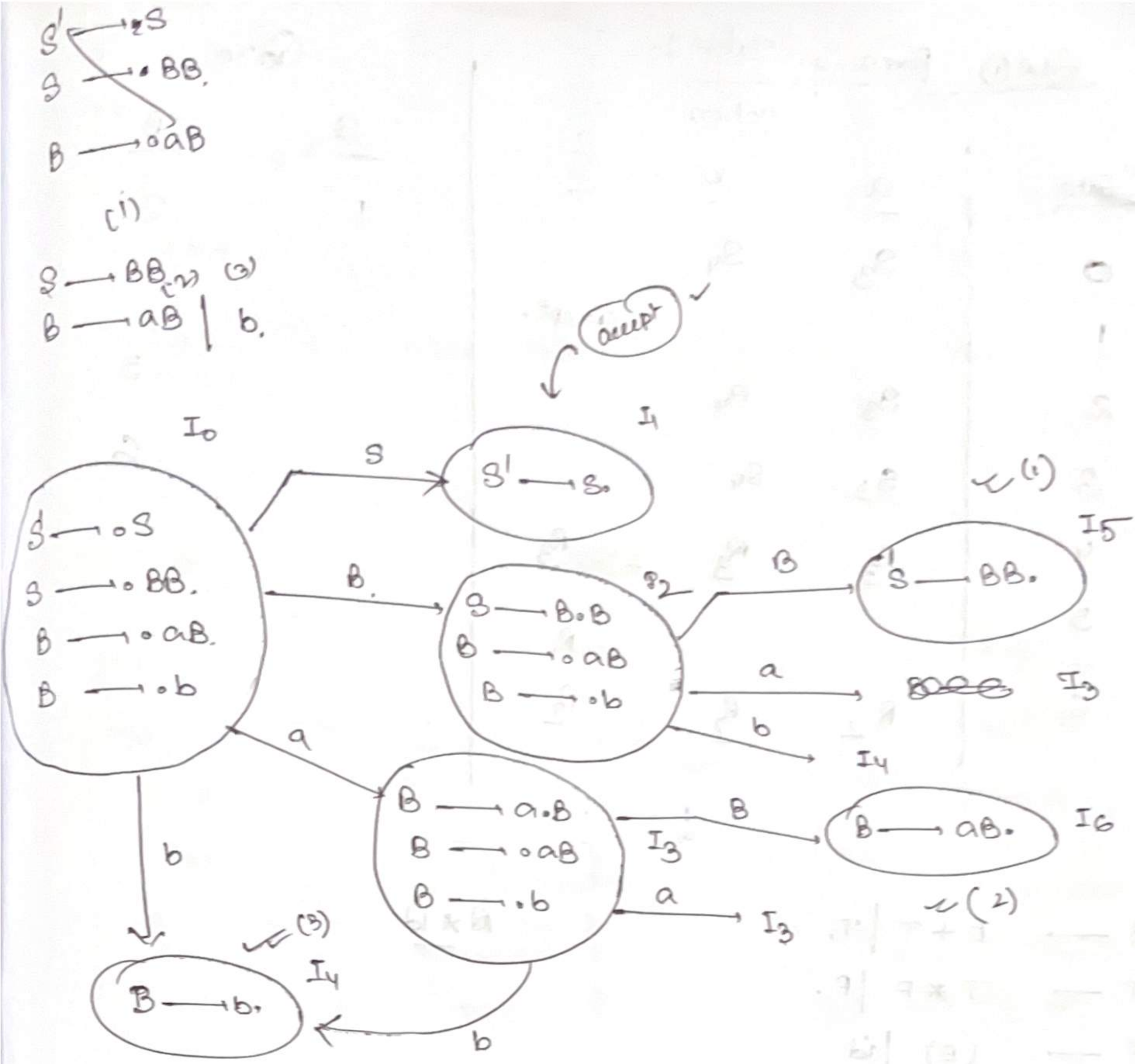
$S \rightarrow .BB$

$B \rightarrow .aB$

(1)

$S \rightarrow BB$ (2)

$B \rightarrow aB \mid b$ (3)



LR(0) Parsing Table :

State	Action			Go-To	
	a	b	$	S	B
0	S_3	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

SLR(1) Parsing Table :-

State	Action			GoTo	
	a	b	$	S	B
0	S_3	S_4		1	2
1			accept.		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5			R_1		
6.	R_2	R_2	R_2		

B ⟶ B + T | T.

T ⟶ T * F | F.

f ⟶ (E) | id

id * id

Stack	Input 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 B ⟶ T.
$ B	$	Accept.

parser has choice to Shift as well, Reduce simultaneously.

here we can push * in stack (Shift operator)

+ also T can be reduced to B (B ⟶ T)

↳ (Reduce) -

↳ Shift / Reduce conflict -

If reduce $E \rightarrow T$ performed,

then

$\$E$

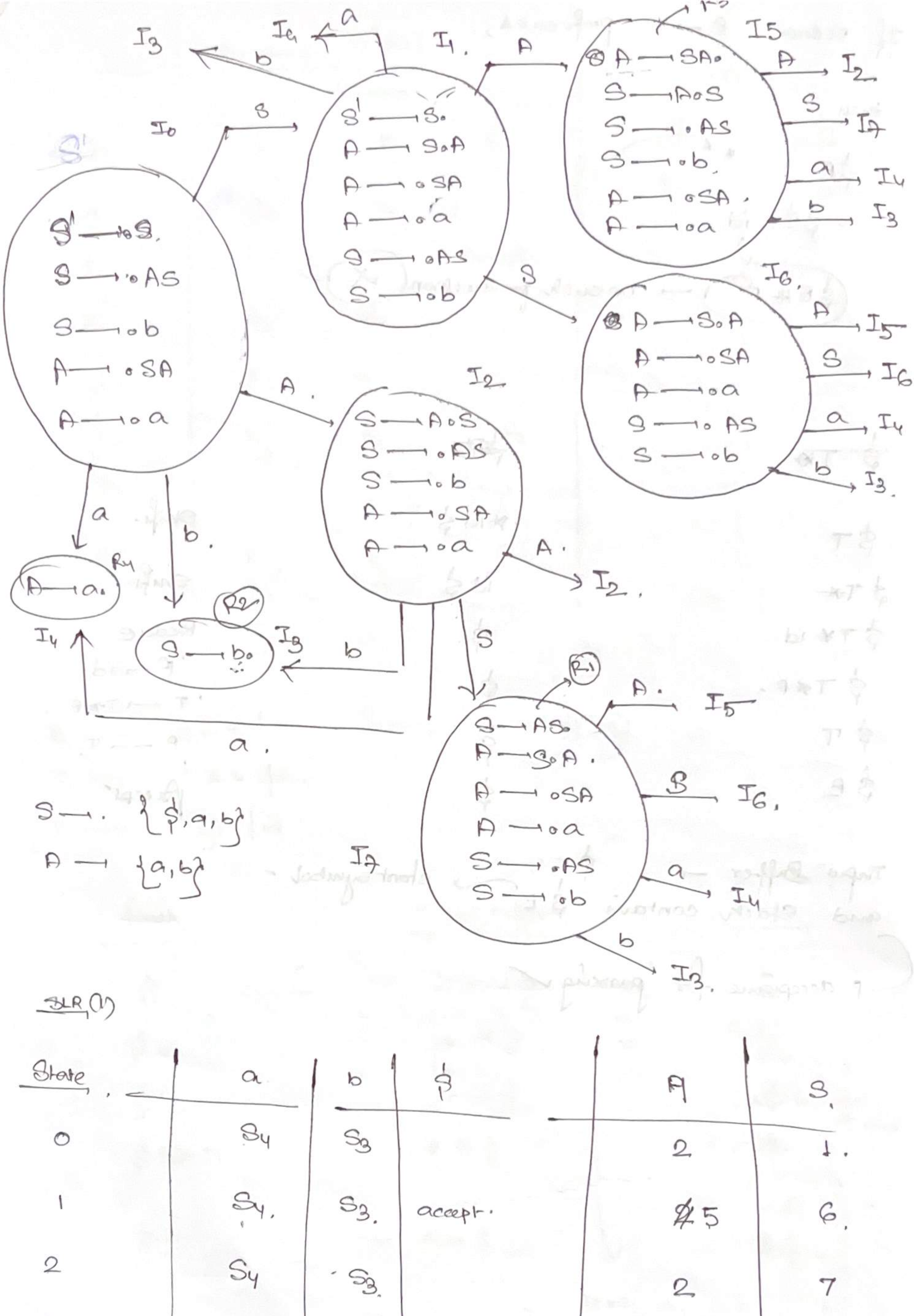
$= \$E * id$

$\$E * F$ → no such production ✗

$\$$	$\$$	
$\$T$	$*id\$$	shift.
$\$T*$	$id\$$	shift.
$\$T*id$	$\$$	Reduce $F \rightarrow id$
$\$T*F$	$\$$	$T \rightarrow T*F$.
$\$T$	$\$$	$E \rightarrow T$.
$\$E$	$\$$	Accept

Input Buffer → $\$$ and Stack contain $\$E$ → start symbol

↳ acceptance for parsing ✓



SLR(1)

State	a	b	$	A	S
0	S_4	S_3		2	1
1	S_4	S_3	accept	5	6
2	S_4	S_3		2	7
3	R_2	R_2	R_2		
4	R_4	R_4	R_4		
5	S_4/R_3	S_3/R_3		2	7
6	S_4	S_3		5	6
7	S_4/R_1	S_3/R_1	R_1	5	6

22/03/2023

CLR(1) and LALR(1) — we use LR(1) cannonical items.

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

$S \longrightarrow AA.$
$A \longrightarrow aA$
$A \longrightarrow b$
} — create CLR(1) parsing table.

= Neceessity: Augmented Grammar.

$S' \longrightarrow S$
$S \longrightarrow AA.$
$A \longrightarrow aA$
$A \longrightarrow b.$

(CLR) ✓

$S' \longrightarrow \circ S, \$)$
$S \longrightarrow \circ AA, \$)$
$A \longrightarrow \circ aA, a/b)$
$A \longrightarrow \circ b, a/b)$

lookah.

$S' \longrightarrow$ lookahead will be always $.

if First of remaining. is ε, then lookahead will be $.

$S \longrightarrow$ First(—) = ε
↳ lookahead = $

First(A) = a/b.

$\circ A, \$$

$\therefore$

I_0.

$S' \longrightarrow \circ S, \$$
$S \longrightarrow \circ AA, \$$
$A \longrightarrow \circ aA, a/b.$
$B \longrightarrow \circ b, a/b.$

(P.T.O) ✓

I_0

$S' \longrightarrow .S, \$$

$S \longrightarrow .AA, \$$

$A \longrightarrow .aA, a/b$

$A \longrightarrow .b, a/b$

S → I_1: $S' \longrightarrow S., \$$

A → I_2: $S \longrightarrow A.A, \$$; $A \longrightarrow .aA, \$$; $A \longrightarrow .b, \$$

a → I_3: $A \longrightarrow a.A, a/b$; $A \longrightarrow .aA, a/b$; $A \longrightarrow .b, a/b$

b → I_4: $A \longrightarrow b., a/b$

A → I_8: $A \longrightarrow aA., a/b$

a → I_3 (self-loop)

b → I_4

I_2

$S \longrightarrow A.A, \$$

$A \longrightarrow .aA, \$$

$A \longrightarrow .b, \$$

A → I_5: $S \longrightarrow AA., \$$

a → I_6: $A \longrightarrow a.A, \$$; $A \longrightarrow .aA, \$$; $A \longrightarrow .b, \$$

b → I_7: $A \longrightarrow b., \$$

A → I_9: $A \longrightarrow aA., \$$

a → I_6 (self-loop)

b → I_7

LALR(1) — merge states having same productions but diff. lookaheads. (these states need to be combined)

3,6 — . are combined

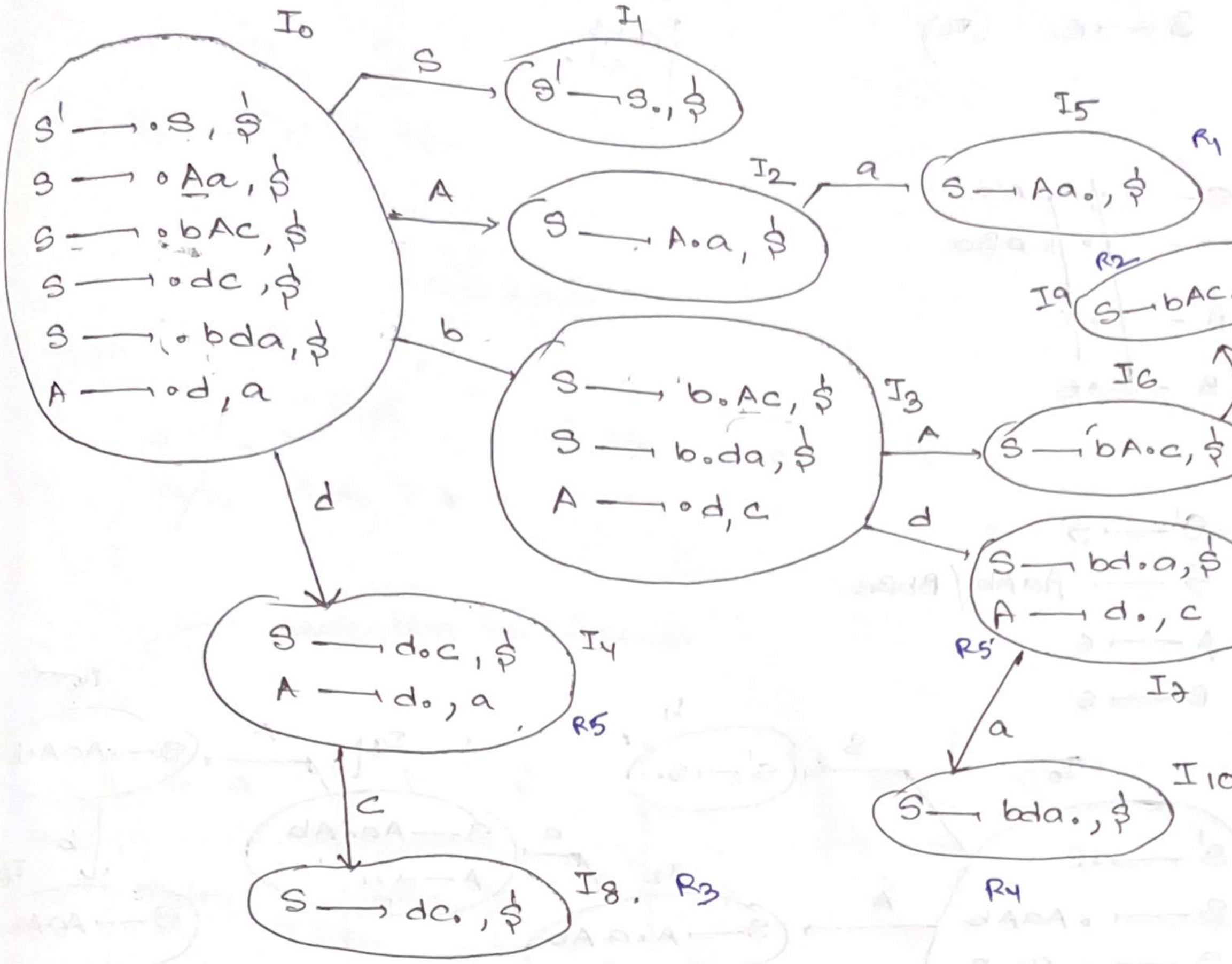
thus (4,7) (8,9) — combined.

	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
3,6	~~R_3~~ S_{36}	~~R_3~~ S_{47}	~~R_3~~		89
4,7	R_3	R_3	R_3		
5			R_1		
8,9	R_2	R_2	R_2		

Q. $S \Rightarrow Aa \;(R_1) \;/\; bAc \;(R_2) \;/\; dc \;(R_3) \;/\; bda \;(R_4)$

$A \rightarrow d \;(R_5)$.

CLR(1) ✓



	a	b	c	d	$	S	A
0		S_3		S_4		1	2
1					accept		
2	S_5						6
3				S_7			
4	R_5			S_8			
5					R_1		
6				S_9			
7	S_{10}			R_5			
8					R_3		
9					R_2		
10.					R_4		

r1 r2 Follow LR(0) / SLR(1) 24/03/2023

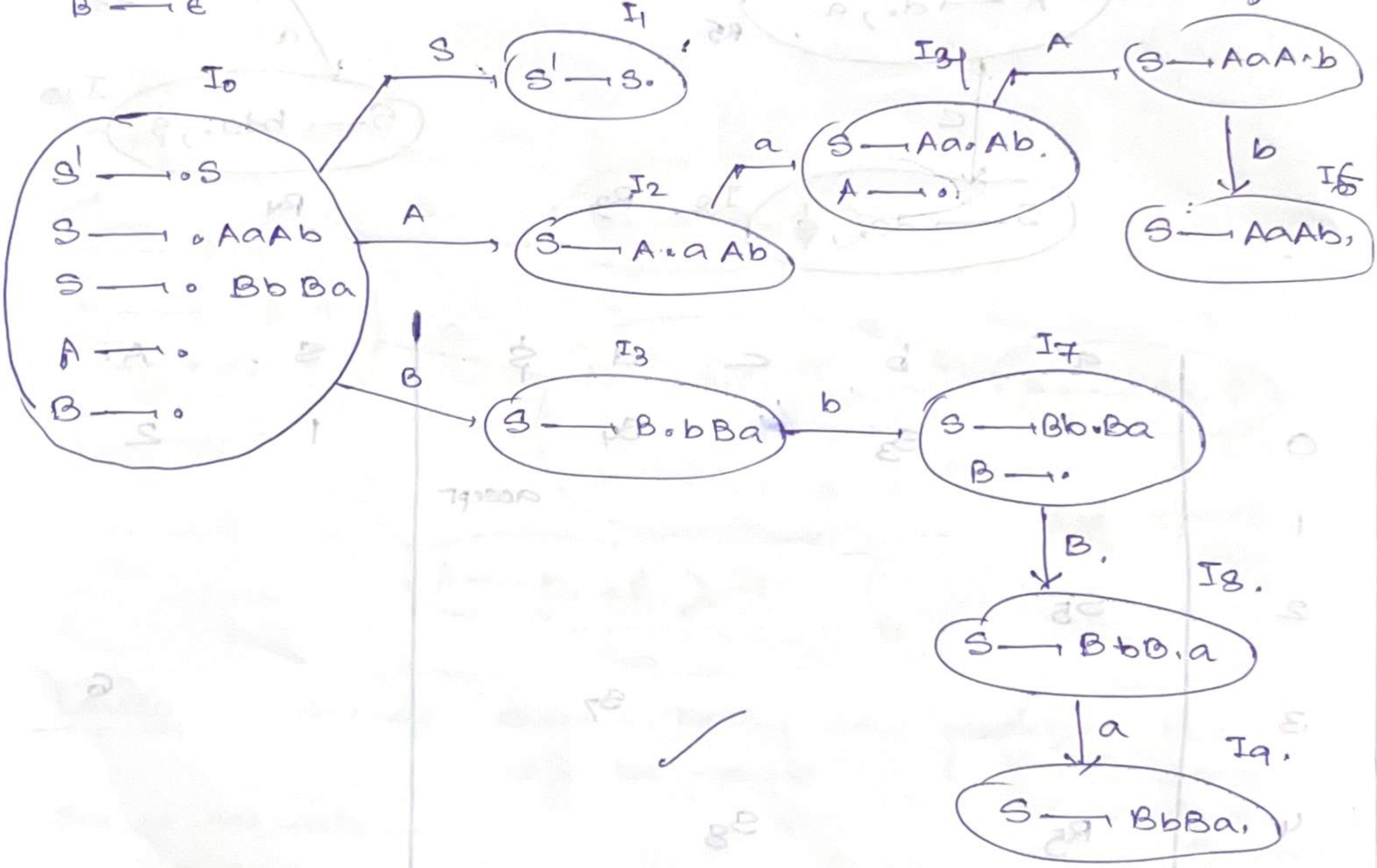
S ⟶ AaAb / BbBa. {$}.

A ⟶ ∈ (r3) {a, b}

B ⟶ ∈. (r4) {a, b}.

S ⟶ ·AaAb
S ⟶ ·BbBa
A ⟶ ·∈
B ⟶ ·∈

S' ⟶ S
S ⟶ AaAb / BbBa
A ⟶ ∈
B ⟶ ∈



LR(0)

Reduce

for state I0 ~

	a	b	$	A	B	S
0	r3/r4	r3/r4	r3/r4	2	3	1

Reduce conflict

SLR(1)

	a	b	$	A	B	S
0	r3/r4	r3/r4		2	3	1 ~

reduction is follow.

SLR(1)

	a	b	$	A	B	S
0	R3/R4	R3/R4		2	3	1.
1			accept.			
2	S4					
3		S7				
4	R3	R3.		5.		
5		S6.				
6			R1			
7	R4	R4			8.	
8	S9					
9			R2			

S' → .S, $

S → .AaAb, $ / .BbBa, $

A → . , a

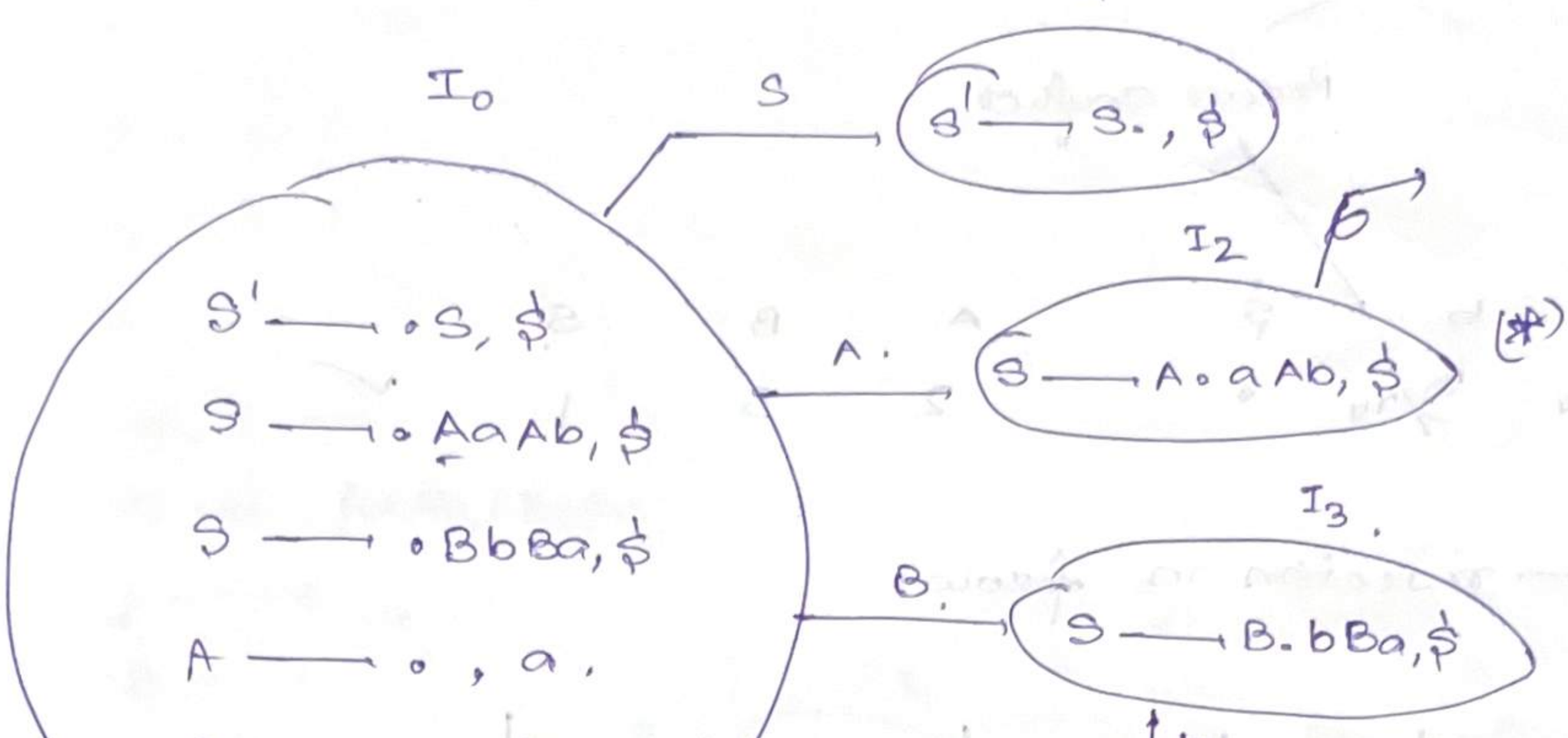
S → .Aa/Ab, a/b

A → . , a

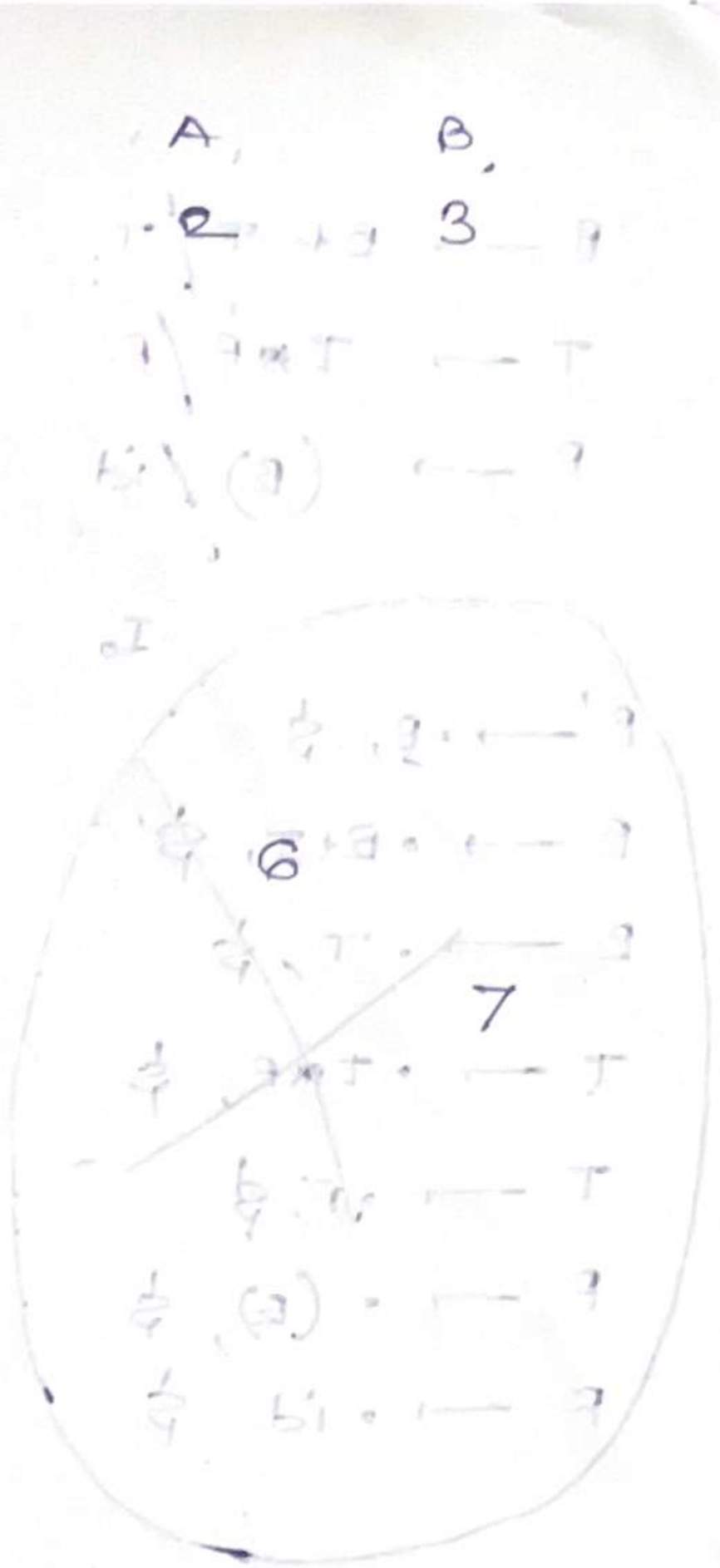
reduction on lookaheads

A → . , (a)

CLR(1)



	a	b	$	S	A	B
0	R_3	R_4.		1	2	3
1.			accept			
2	S_4.					
3.		S_5				
4		R_3				
5.	R_4.					
6		S_8.				
7	S_9					
8.			R_1			
9			R_2			



CLR(1) ✓ parsing table

$S' \rightarrow S$ R_1
$S \rightarrow AA$ R_2
$A \rightarrow aA$ R_3
$A \rightarrow b$. R_4

$S \rightarrow AA$. R_1
$A \rightarrow aA$. R_2
$A \rightarrow b$. R_3

	a	b	$	S	A.
0	S_3	S_4.		1.	2.
1			accept.		
2	S_6	S_7			5
3	S_3	S_4.			8
4	R_3	R_3	R_3		
5			R_1.		
6	S_6	S_7			9
7	R_3	R_3.	R_3		
8	R_2	R_2	R_2		
9	R_2	R_2	R_2		

37/3

12,33

$B \longrightarrow B+T \,/\, T$

$T \longrightarrow T*F \,/\, F$

$F \longrightarrow (E) \,/\, id$

$\longrightarrow$ CLR(1)

I_0:

$B' \longrightarrow \cdot B, \$$

$B \longrightarrow \cdot B+T, \$$

$B \longrightarrow \cdot T, \$$

$T \longrightarrow \cdot T*F, \$$

$T \longrightarrow \cdot F, \$$

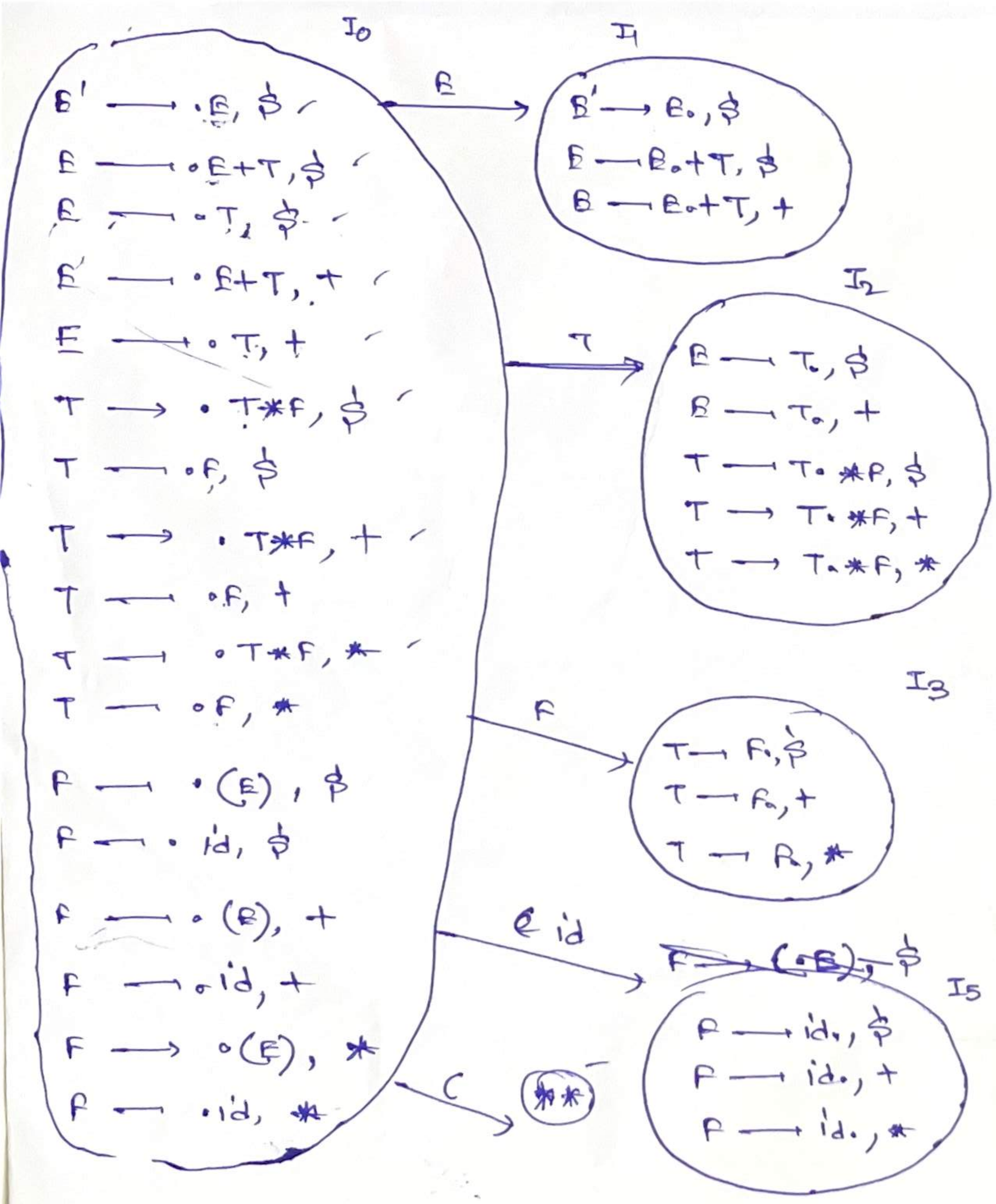
$F \longrightarrow \cdot (E), \$$

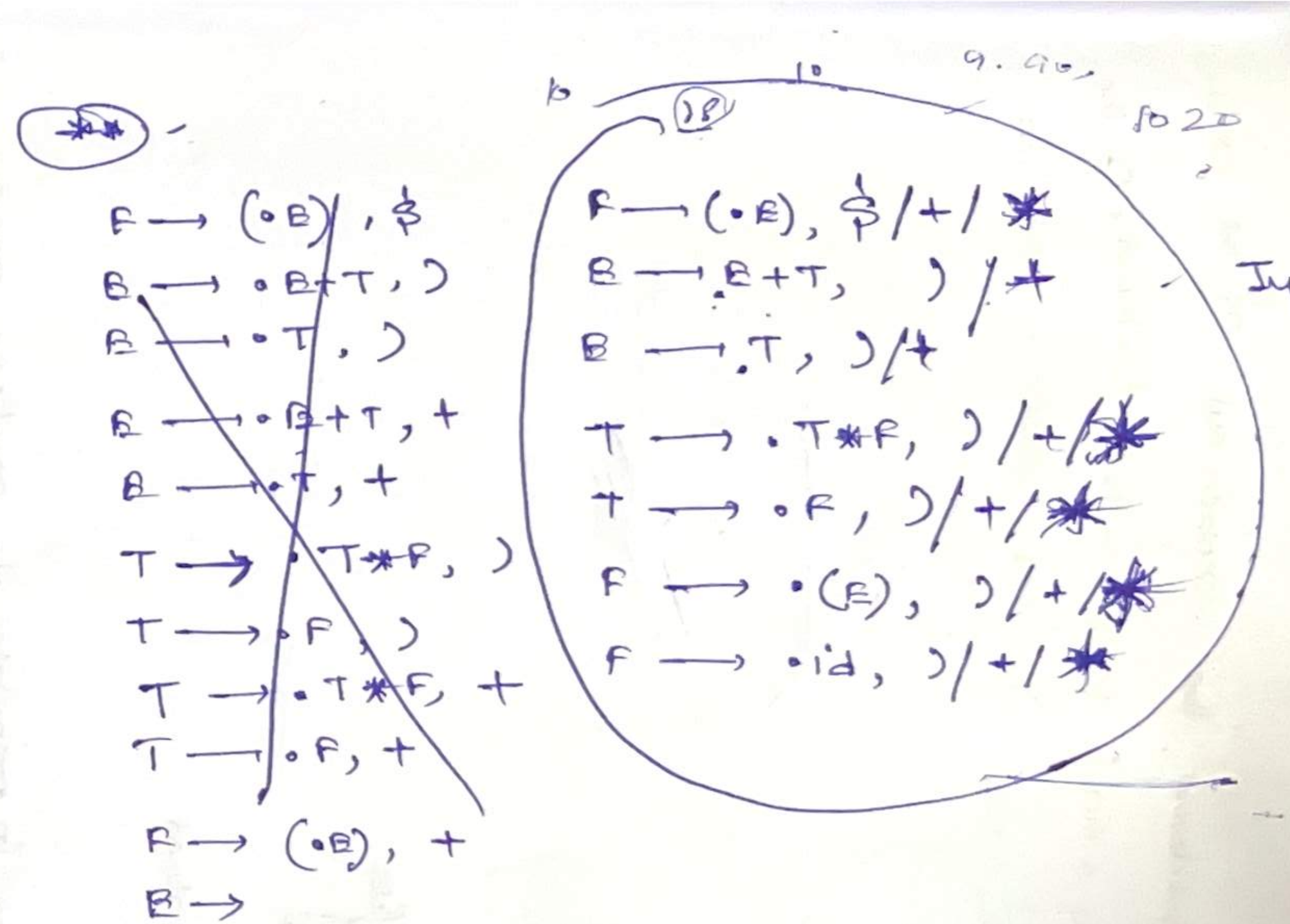
$F \longrightarrow \cdot id, \$$

(29.25) 28.x 29.) 29 25.

I_1 on B $\longrightarrow$ $B \cdot +T, \$/+$

I_2 on T $\longrightarrow$ $B \cdot T, \$/+$


I0
E
I1
B' → .E, $
E → .E+T, $
E → .T, $
E → .E+T, +
E → .T, +
T → .T*F, $
T → .F, $
T → .T*F, +
T → .F, +
T → .T*F, *
T → .F, *
F → .(E), $
F → .id, $
F → .(E), +
F → .id, +
F → .(E), *
F → .id, *
B' → E., $
E → E.+T, $
E → E.+T, +
I2
T
E → T., $
E → T., +
T → T.*F, $
T → T.*F, +
T → T.*F, *
I3
F
T → F., $
T → F., +
T → F., *
C
**
I5
id
F → id., $
F → id., +
F → id., *

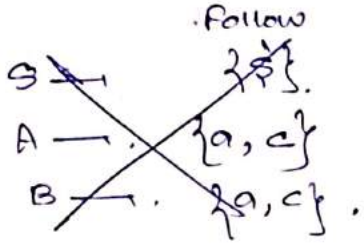

**
F → (.E), $
B → .E+T,)
B → .T,)
E → .E+T, +
E → .T, +
T → .T*F,)
T → .F,)
T → .T*F, +
T → .F, +
F → (.E), +
E →
9.40
8020
F → (.E), $/+/*
E → E.+T,)/+
E → .T,)/+
T → .T*F,)/+/*
T → .F,)/+/*
F → .(E),)/+/*
F → .id,)/+/*
I4

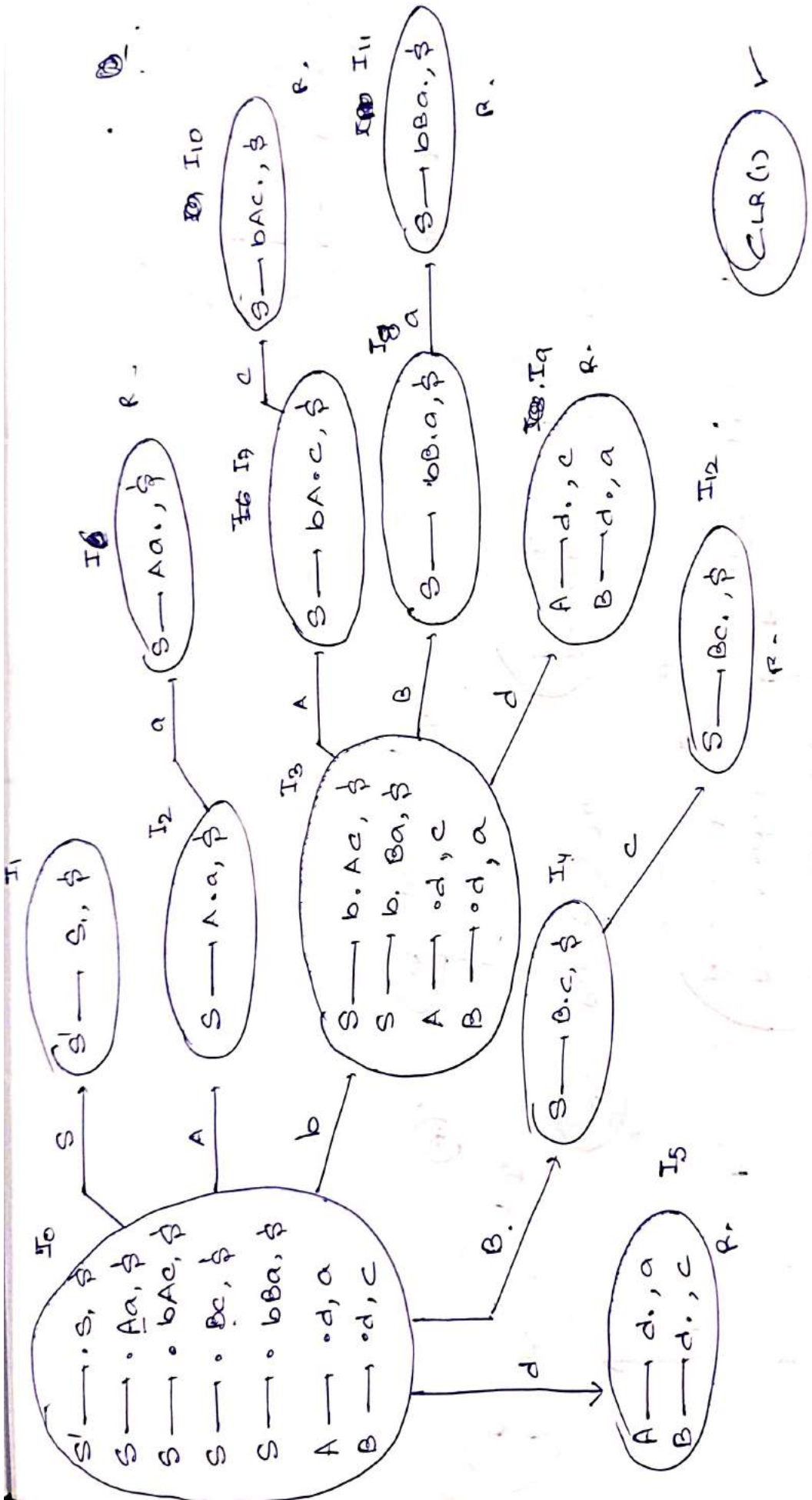
1. S → . Aa / bAc / Bc / bBa

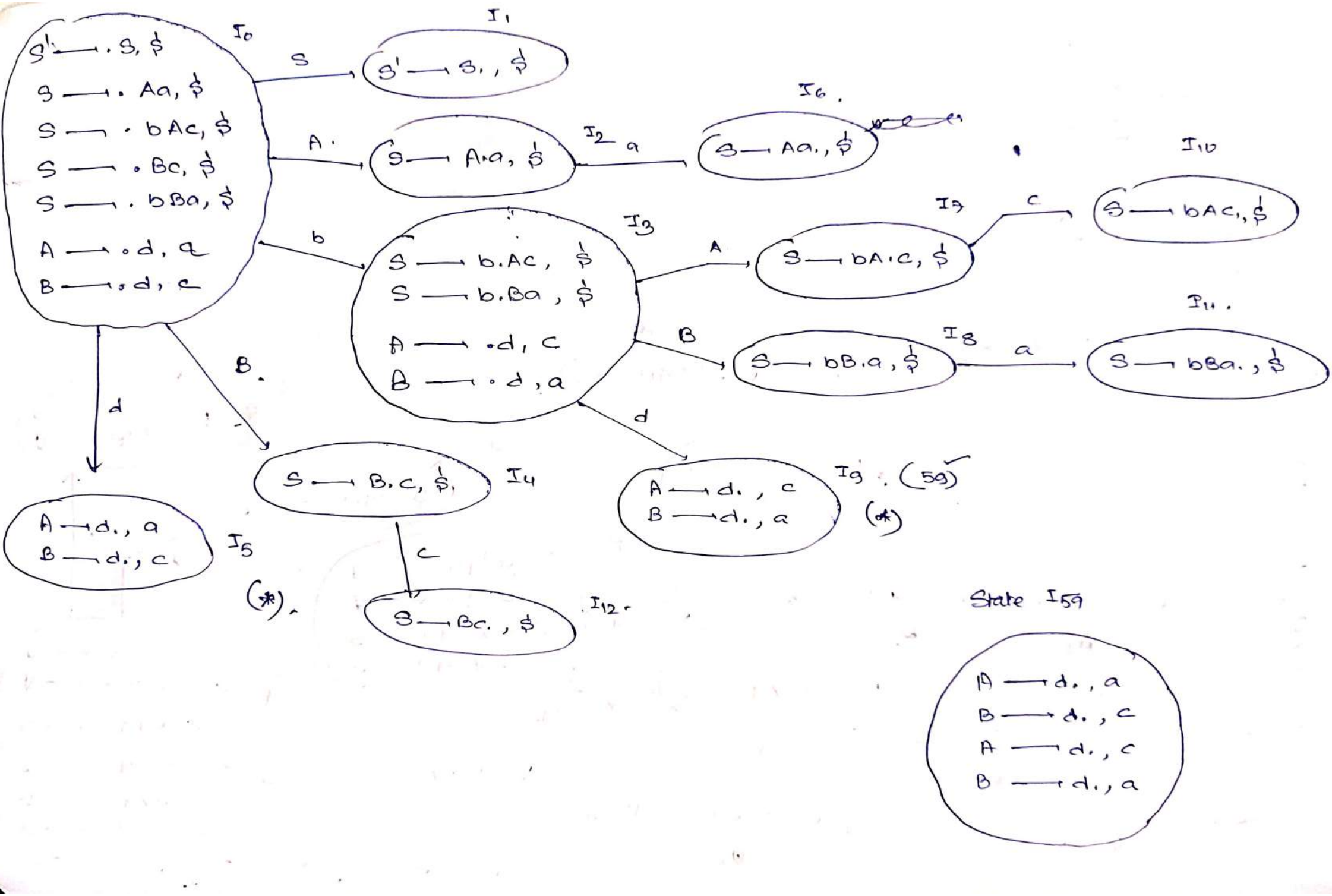
A → d (3)

B → d (4)

CLR(1) & LALR(1)






I0
S' → . S, $
S → . Aa, $
S → . bAc, $
S → . Bc, $
S → . bBa, $
A → . d, a
B → . d, c
S
I1
S' → S. , $
A
I2
S → A.a, $
a
I6
S → Aa. , $
b
I3
S → b.Ac, $
S → b.Ba, $
A → . d, c
B → . d, a
A
I7
S → bA.c, $
c
I10
S → bAc, $
B
I8
S → bB.a, $
a
I11
S → bBa. , $
d
A → d. , a
B → d. , c
I5
(*)
B
I4
S → B.c, $
c
I12
S → Bc. , $
d
A → d. , c
B → d. , a
I9 (59)
(*)
State I59
A → d. , a
B → d. , c
A → d. , c
B → d. , a

CLR(1)

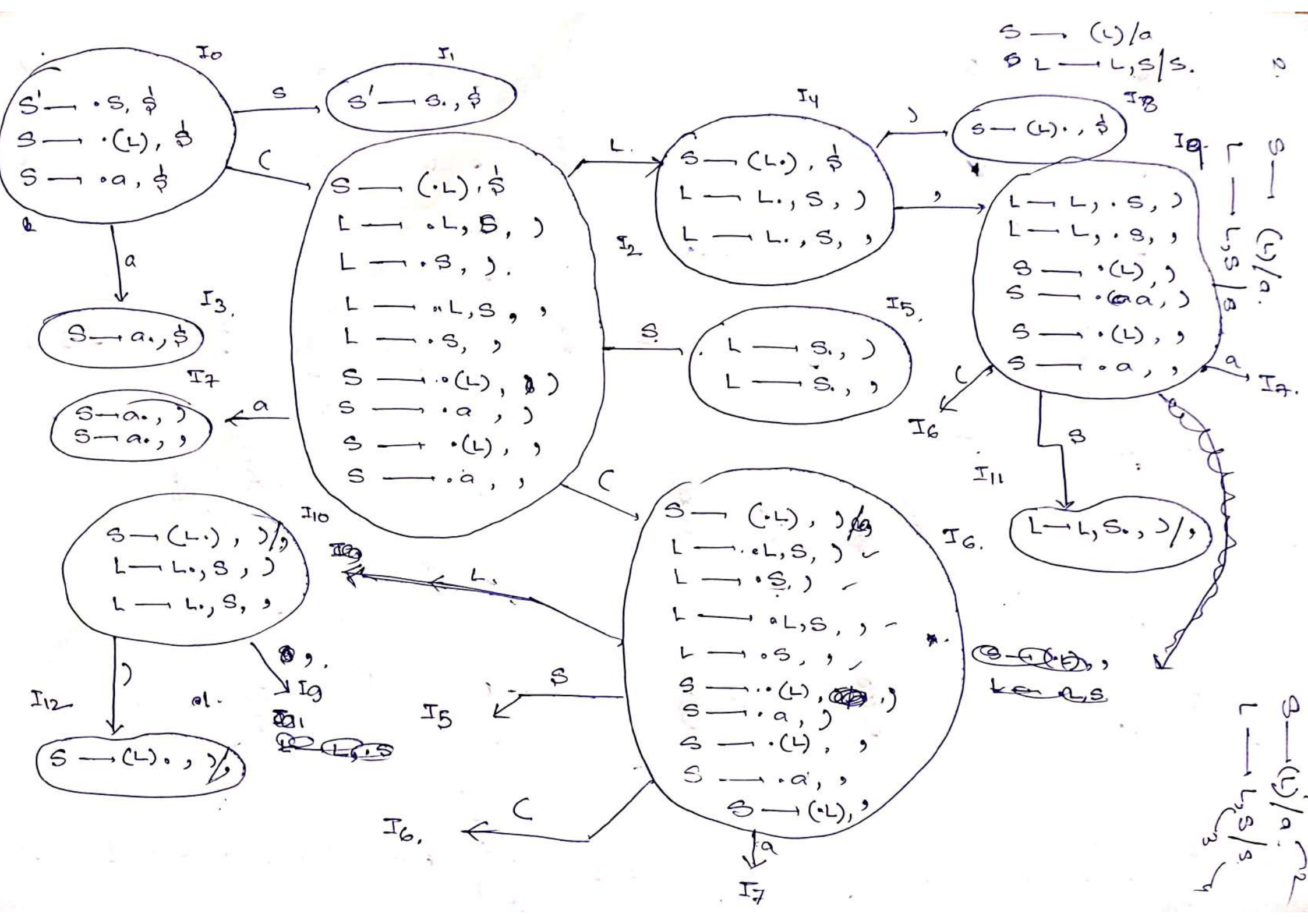
State / Item	a	b	c	d	$	S	A	B
0		S_3		S_5		1	2	4
1					accept			
2	S_6							
3				S_9			7	8
4			S_{12}					
5	R_5		R_6					
6					R_1			
7			S_{10}					
8	S_{11}							
9	R_6		R_5					
10					R_2			
11					R_4			
12					R_3			

CLR(1) ✓

LALR(1)

State / Item	a	b	c	d	$	S	A	B
0		S_3		S_{59}		1	2	4
1					accept			
2	S_6							
3				S_{59}			7	8
4			S_{12}					
59	R_5/R_6		R_5/R_6					
6					R_1			
7			S_{10}					
8	S_{11}							
10					R_2			
11					R_4			
12					R_3			

LALR(1) ✗


S → (L) / a
L → L, S / S.
I0
S' → .S, $
S → .(L), $
S → .a, $
S
I1
S' → S., $
(
a
I3
S → a., $
I7
S → a.,)
S → a., ,
I2
S → (.L), $
L → .L, S,)
L → .S,)
L → .L, S, ,
L → .S, ,
S → .(L),)
S → .a,)
S → .(L), ,
S → .a, ,
L
S
I4
S → (L.), $
L → L., S,)
L → L., S, ,
)
,
I8
S → (L)., $
I9
L → L, .S,)
L → L, .S, ,
S → .(L),)
S → .a,)
S → .(L), ,
S → .a, ,
I5
L → S.,)
L → S., ,
I6
I11
L → L, S.,)/,
I10
S → (L.),)/,
L → L., S,)
L → L., S, ,
I12
S → (L).,)/,
I9
I11
I5
I6.
I7
S → (L)/a
L → L, S/S
3

CLR(1)

State	(	)	a	,	$	S	L.
0	S_2		S_3			1	
1					acc.		
2	S_6		S_7			5,	4
3					R_2		
4		S_8		S_9			
5		R_4		R_4.			
6	S_6		S_7			5,	10
7		R_2		R_2			
8					R_1		
9	S_6		S_7			11	
10		S_{12}		S_9			
11		R_3		R_3.			
12		R_1		R_1 .			

LALR (1). ⟶ unique states: 0, 1, {2,6}, {3,7}, {4,10}, 5, {8,12}, 9, 11.

State	(	)	a	,	$	S	L.
0	S_{26}		S_{37}		acc.	1	
1.						5.	{4,10}.
{2,6}	S_{26}		S_{37}				
{3,7}		R_2		R_2	R_2		
{4,10}		$S_{8,12}$		S_9			
5		R_4.		R_4.			
{8,12}		R_1		R_1	R_1		
9	S_{26}.		S_{37}			11	
11		R_3		R_3			

SLR(1) ✓ LALR(1) ✓.

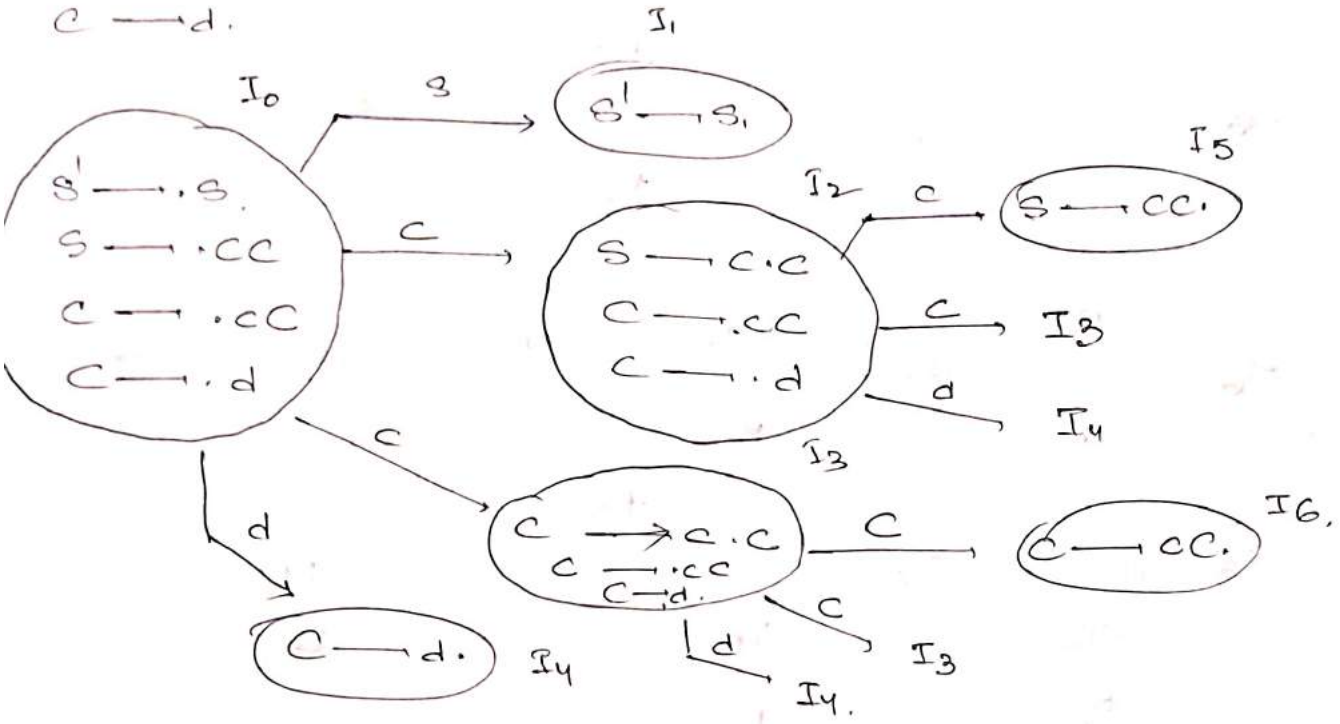
05/04/2023

LR(0)

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d.$



	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
2	S_3	S_4			5
3	S_3	S_4			6
4	R_3	R_3	R_3		
5	R_1	R_1	R_1		
6	R_2	R_2	R_2		

I/p string **ccdd**

Stack	Input	Action
$0	ccdd$	shift c $3.
$0c3	cdd$	shift c $3
$0c3c3	dd$	shift d $4.
~~$0c3c3d4~~	~~d$~~	
$0c3c3d4.	d$	reduce C → d
$0c3c3C	d$	

top of stack has no state

If not. go to previous state 3 and check. for production.

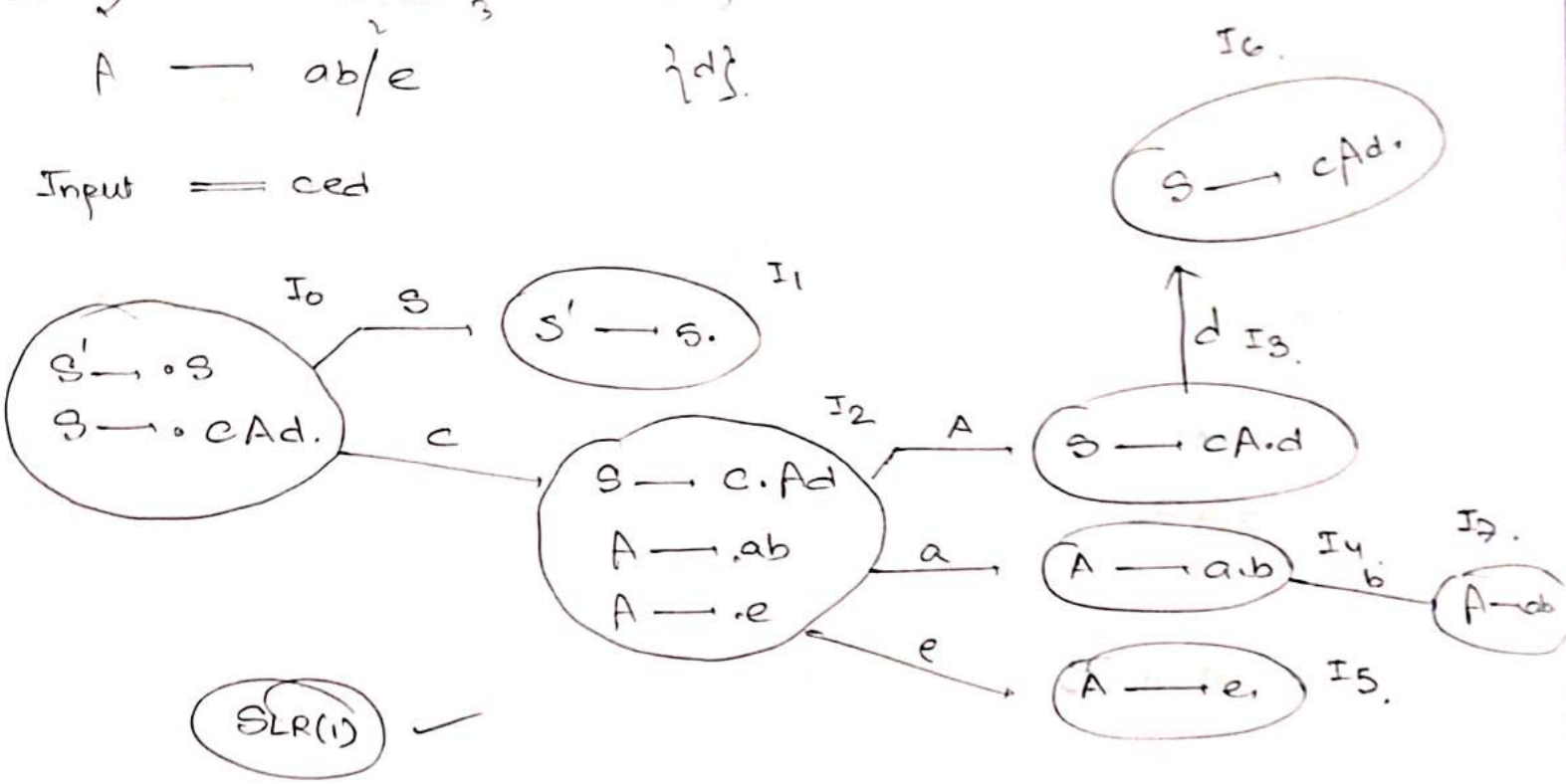
$0c3c3C6.	d$	reduce C → cC.
$0c3C6.	d$	reduce C → cC.
$0C2	d$	or shift d $4.
$0C2d4.	$	reduce C → d
$0C2C5.	$	reduce S → cC.
$0S1.	$	accept.

∴ String is accepted

Q. $S \longrightarrow cAd$ (1) {$, }

$A \longrightarrow ab/e$ (2) {d}.

Input = ced



	a	b	c	d	e	$	S	A.
0			S_2.				1.	
1.						acc.		
2.	S_4.				S_5.			3.
3				S_6.				
4		S_7						
5.				R_3				
6						R_1		
7				R_2				

Stack	Input	Action
$0	ced$	Shift c S_2.
$0c2	ed$	Shift e S_5
$0c2e5.	d$	Reduce A → e.
$0c2A3.	d$	Shift d S_6
$0c2A3d6.	$	Reduce S → cAd
$0S1.	$	accept.

LR(0) –

	a	b	c	d	e	$	S	A.
			S_2				1	
0.						acc.		
1								
2	S_4.				S_5.			3.
3				S_6.	S_5.			
4.		S_7						
5.	R_3	R_3	R_3	R_3	R_3	R_3.		
6	R_1	R_1	R_1	R_1	R_1	R_1		
7.	R_2	R_2	R_2	R_2	R_2	R_2		

Stack	Input	Action
$0	ced$	Shift c S_2
$0c2	ed$	Shift e S_5
$0c2e5	d$	Reduce A → e
$0c2A3	d$	Shift d S_6.
$0c2A3d6	$	Reduce $S \rightarrow cAd$
$0S1	$	accept.

16/04/2022

Semantic Analysis :

Syntax directed definition :

Gramma + Semantic rules

$E \longrightarrow E_1 + T$

$\{E.val = E_1.val + T.val\}$

$L \longrightarrow E$ → same variable as E. { $T.val = E.val$ }

$E \longrightarrow E_1 + T$ { $E.val = E_1.val + T.val$ }

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

$T \longrightarrow T_1 * F$ { $T.val = T_1.val * F.val$ }

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

$F \longrightarrow id.$ { $F.val = digit$ }

Two types of attribute :

(Synthesized attribute

Inherited attribute)

$S \longrightarrow ABC.$

{ $S.val =$ ·

parent node is calculated from child node.

When the node value is calculated from parent/sibling of the node.

- At any place, where parent attribute is calculated by its Children's attribute. ⟶ Synthesized attribute.

- **Inherited:** child's attribute is calculated by parent's / sibling's attribute.

Syntax Directed Definition.

- Convert given infix to postfix expression. (using SDD).

$E \longrightarrow E + T$ {print(+)}

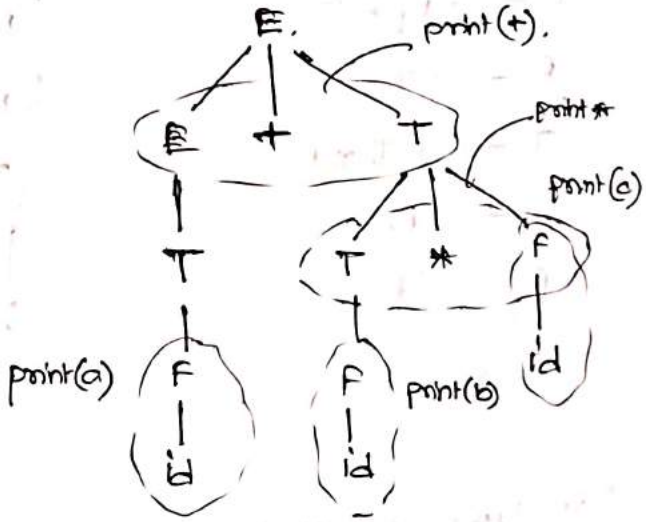
$E \longrightarrow T$

$T \longrightarrow T * f.$ {print(*)}

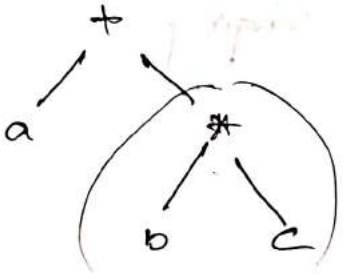
$T \longrightarrow f.$

$f \longrightarrow id$ {print(id)}.

a + b*c
= abc*+



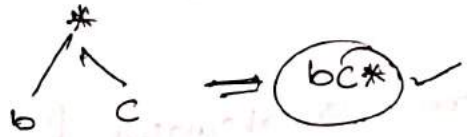
Annotated parse tree.

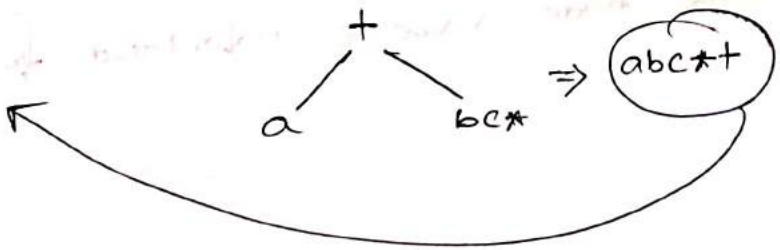


Left Right Root.

(abc*+)

a + b*c
= a + (bc*)
= abc*+

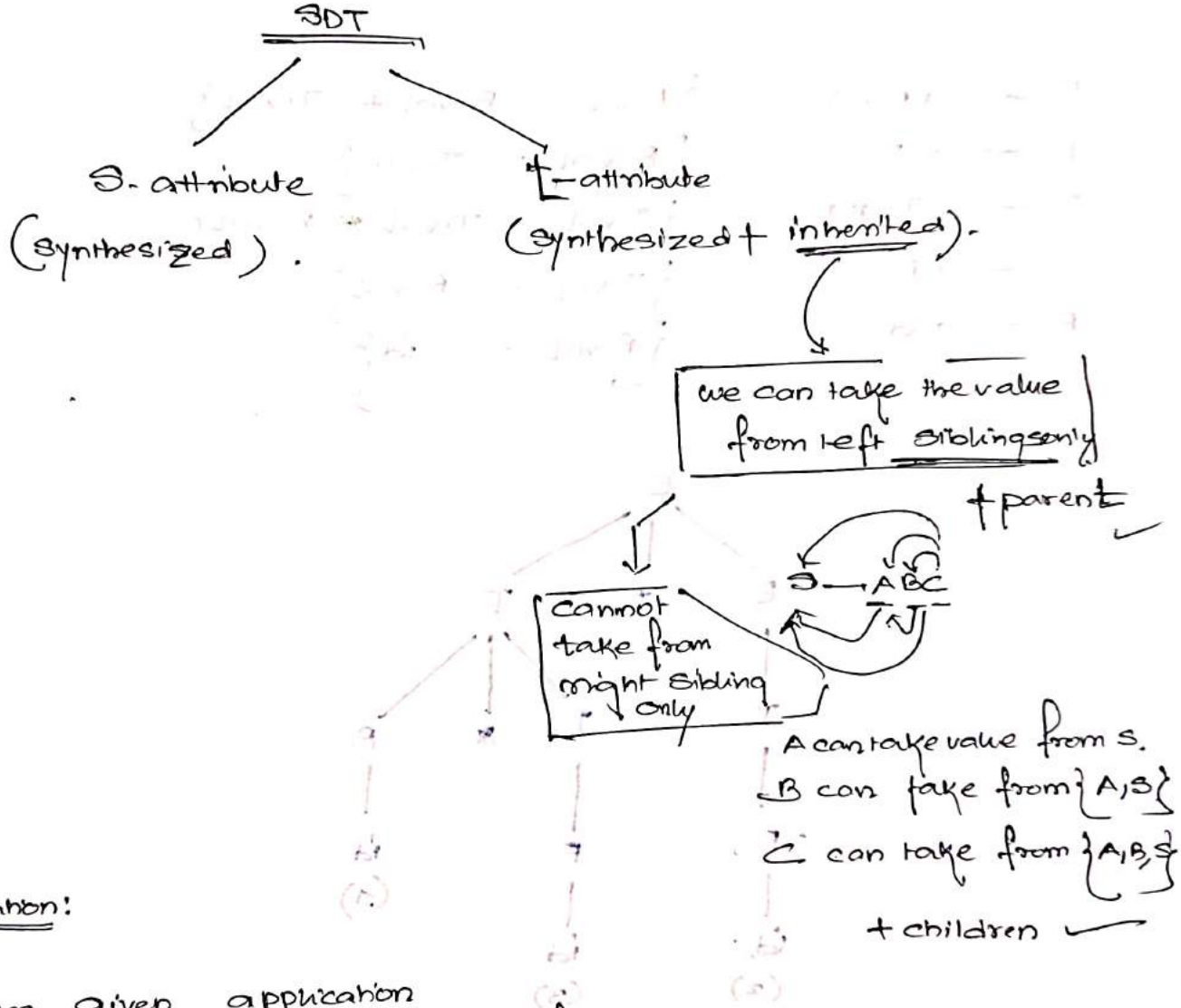




Syntax Directed Translation (SDT):

(interpreting the definitions).

└ output for the parse tree. (translation).



Application:

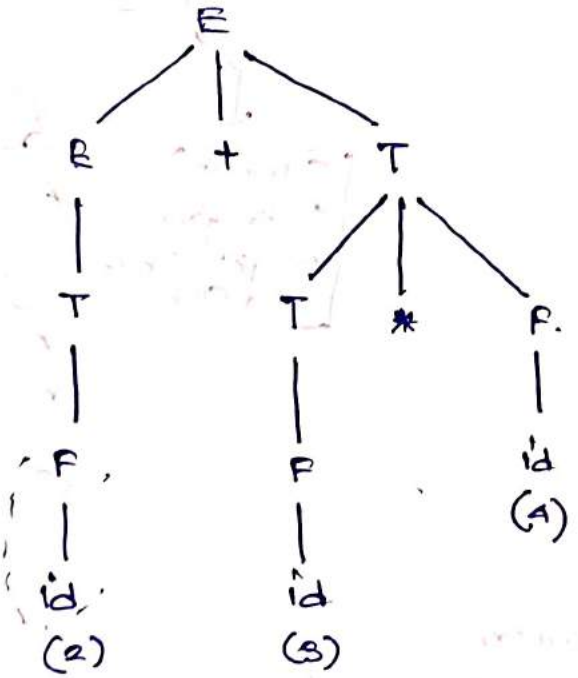
- executing given application
- conversion of infix to postfix
- conversion of infix to prefix.
- conversion of binary to decimal
- ~~conver~~ creating intermediate code. ~~genera~~
- type checking.
- finding no. of reductions.

1) Construct SDT for executing given arithmetic expression.

I/p — 2+3*4.

O/p — 14.

$E \longrightarrow E + T$	$\{E.val = E.val + T.val\}$
$E \longrightarrow T$	$\{E.val = T.val\}$
$T \longrightarrow T * f.$	$\{T.val = T.val * F.val\}$
$T \longrightarrow f.$	$\{T.val = f.val\}$
$F \longrightarrow id$	$\{f.val = id\}$



2+03.

Construct a SDT. to create a syntax tree for given arithmetic expression.

Input: x = a+b*c

Syntax tree ← Output

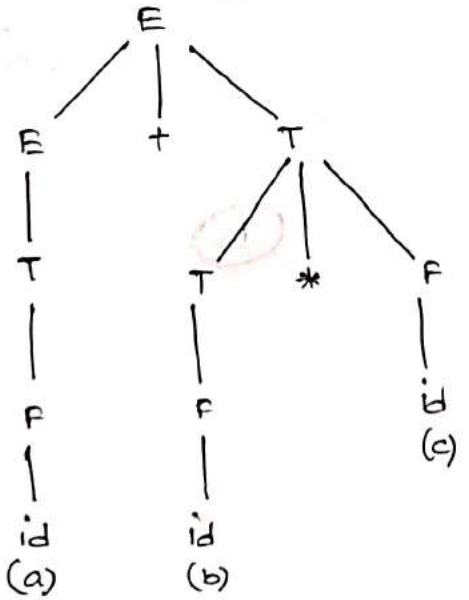
Construct a SDT to generate intermediate code for the given Arithmetic expression

Input → x = a+b*c

Output → Intermediate code.

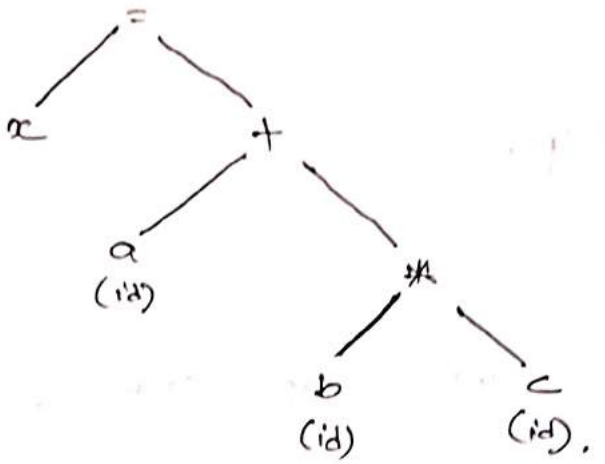
E	E+T	{E.val = E.val + T.val}
E	T	{E.val = T.val}
T	T*F.	{T.val = T.val * F.val}
T	F.	{T.val = F.val}
F	id	{F.val = id}

Parse Tree



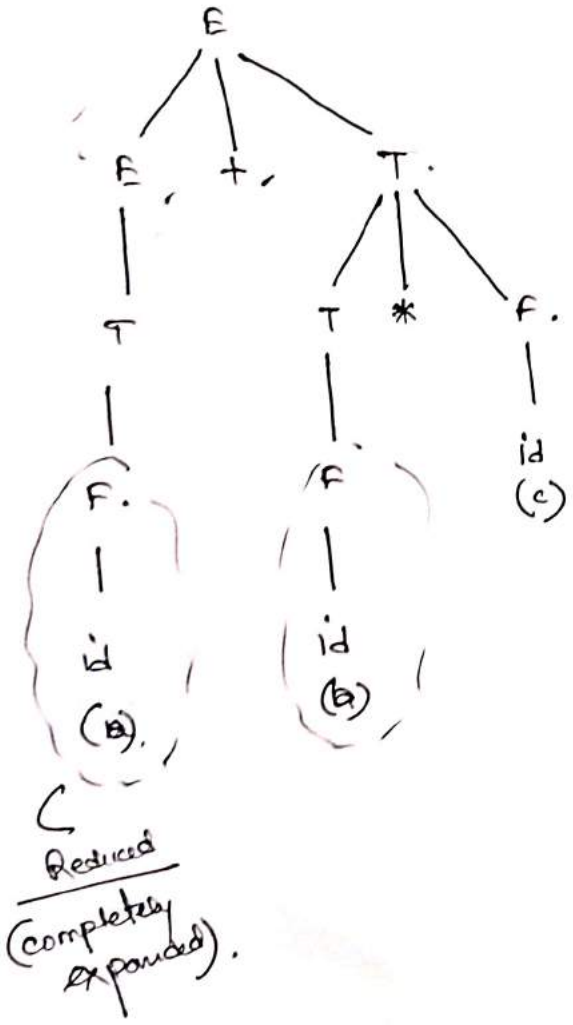
Syntax Tree

b*c → more priority

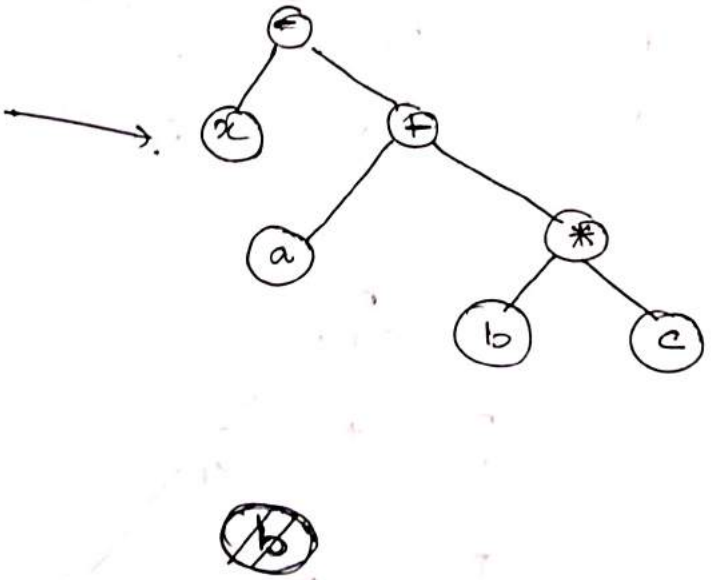


E → E+T. For expressing E, all the non-terminals in its production must be expanded i.e. E_1 and T_0

E → E+T / T
T → T*F / F
F → id



Reduced (completely expanded).



b

Intermediate Code Generation

$x = a + b * c$ — [Three Address Code]

→ only 1 operation + 2 operand

$R_1 := b * c$ ... (i)

$R_2 := a + R_1$ ... (ii)

$x := R_2$ ... (iii)

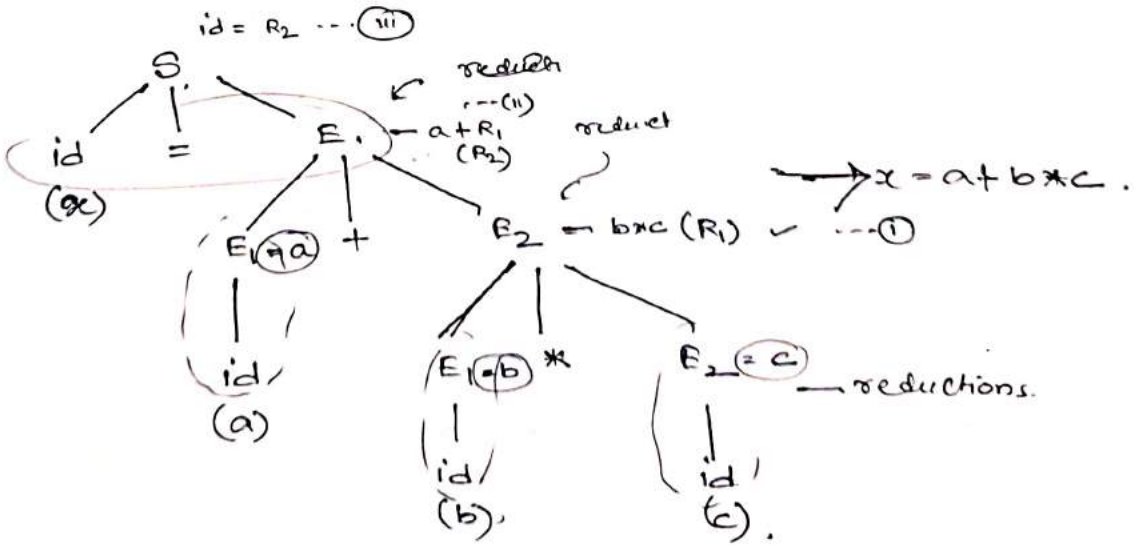
*, + — operators

:= assignment

$E \rightarrow E_1 = E_2$

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

$x = a + b * c$.



Intermediate Code Generation:

Syntax Tree (ST).

↓

ICG

↓

O/P Intermediate Code.

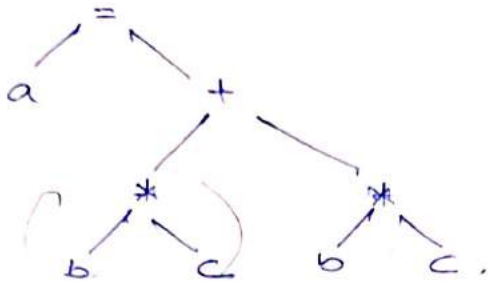
In an Analysis / Synthesis model of a computer, the front-end analyses a source program, and creates intermediate representation., from which the backend generates the target code

Variants of Syntax tree:

(1) Directed Acyclic Graph (DAG):

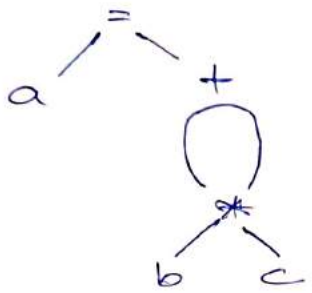
- identifies common subexpression i.e. subexpressions that occur more than once of a given expression.
- DAG has leaves., corresponding to atomic operands and interior nodes corresponding to operators.
- Difference is that, a node in DAG has more than one parent. if the node represents a common subexpression.

$a = b * c + b * c$



(Syntax Tree) -

↳ DAG :

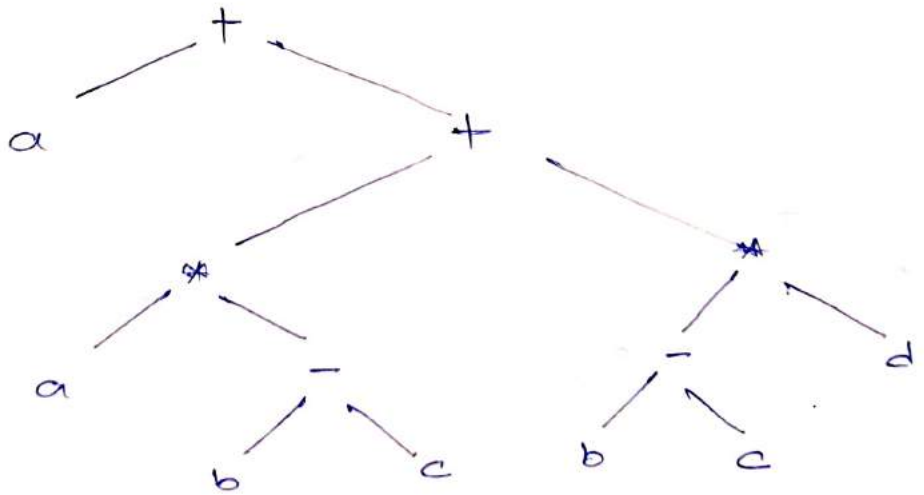


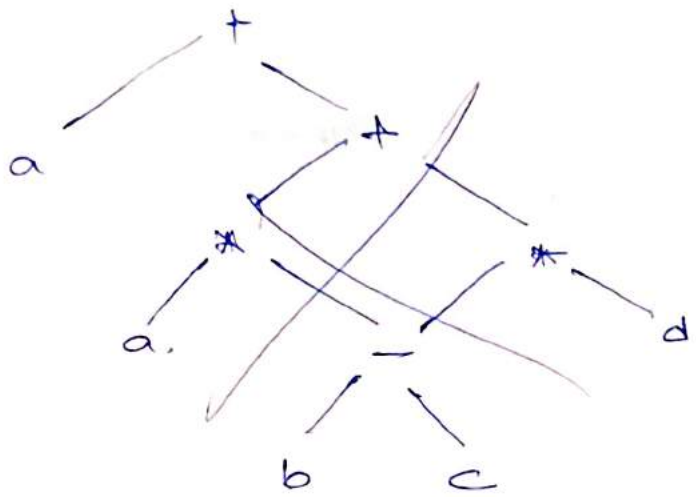
(DAG = Syntax Tree. if no common subexpression found) -

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

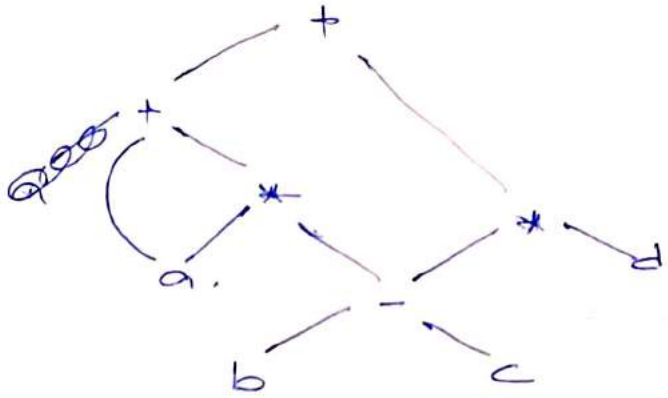
Syntax Tree

$a + b + c$

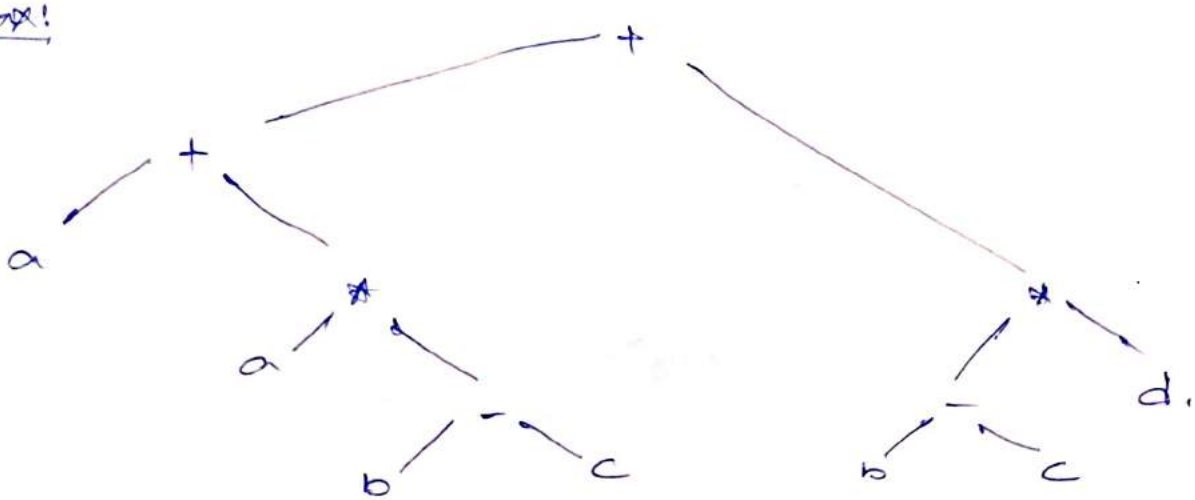




DAG:

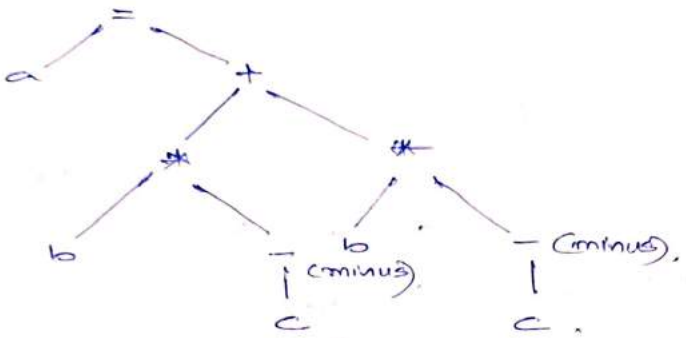


Syntax:

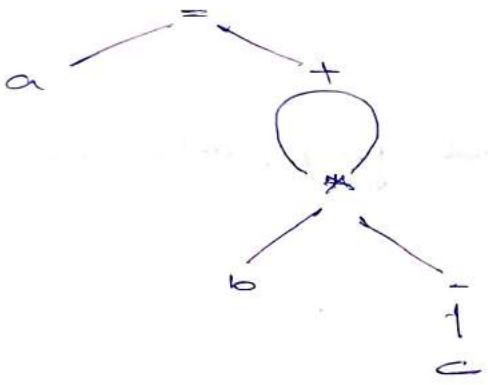


Q. $a = b * -c + b * -c$.

Syntax:



DAG:



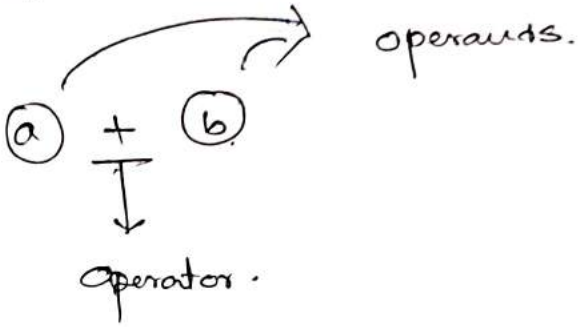
→ What are the different types of Three Address Code.?
(Intermediate Code.
generation) -

18/04/2023

Three Address Code

Add 1. Add 2 Add 3.

1. 2 for operands.
1 for operator.



- binary code
- ~~ternary code~~ unary code.
- assignments
- compare.
- conditional and unconditional jump. Statements
- array.

→ Three Address Code :-

- Quadraple.
- Triples
- Indirect Triples.

} representation.

$a = b * -c + b * -c$

Code →

1. $t_1 = \text{minus } c$
2. $t_2 = b * t_1$
3. $t_3 = \text{minus } c.$
4. $t_4 = b * t_3.$
5. $t_5 = t_2 + t_4.$
6. $a = t_5.$

Quadraple

	Op	arg1	arg2	Res
(unary).	minus.	c		t_1.
	*	b	t_1	t_2
	minus	c		t_3.
	*	b	t_3	t_4.
	+	t_2	t_4	t_5
	=/ assign	t_5		a

Triples

(Ref. no. of order of instructions).

	op	arg1	arg2
(0).	minus	c.	
(1).	*	b	(0)
(2)	minus	c	
(3)	*	b.	(2).
(4)	+	(1)	(3)
(5).	=/ assign.	~~(4)~~ .a	~~(4)~~ (4)

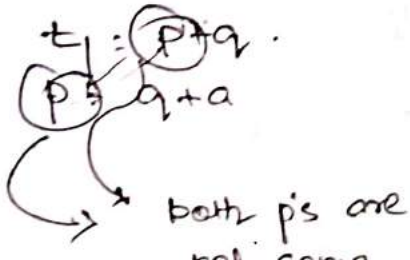
Indirect Triples

	Reference	op	arg1	arg2
(0)	(32)	minus	c	
(1)	(33)	*	b.	(32).
(2)	(34)	minus.	c	
(3)	(35)	*	b	(34)
(4)	(36)	+	(33)	(35)
(5)	(37)	=/assign.	(36)	~~x~~ (36)
			a	

Three Address Code :-

1. $A = -B + C * D.$
2. $A = B + C * D + E + C * D - F.$
3. S: $p + p * q + r * s - c + r * s.$

SSA (Static Single assignment)



1. t_1 = minus B.

2. t_2 = C * D.

3. t_3 = t_1 + t_2

4. A = t_3.

Quad:

op	arg1	arg2	result
minus	B		t_1
*	C	D.	t_2
+	t_1	t_2	t_3.
=	t_3		A.

Triple:

	op	arg1	arg2
(0)	minus	B.	
(1).	*	C	D.
(2)	+	(0)	(1).
(3).	=	A.	(2).

← Code Optimisation

Common Sub-expression elimination:

$a = b * -c + b * -c$

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

→

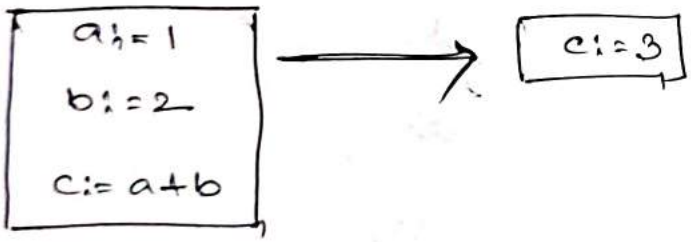
$t_1 := -c$
$t_2 := b * t_1$
$t_4 := b * t_1$
$t_5 := t_2 + t_4$
$a := t_5$

→

$t_1 := -c$
$t_2 := b * t_1$
$t_5 := t_2 + t_2$
$a := t_5$

Constant Folding:

deducing at compile time, that the value of E(expression) is constant, and we would directly use the value itself.



Both common sub-expression elimination and constant folding, contains redundancies,

like $t_5 := t_4$ → both variables are present. So, instead of t_5, we can directly write t_4 (or use)

(memory usage will reduce, if we remove t_5)

Replacing expensive operations with cheaper operations resp., is known as Strength reduction.

Peephole Optimisation:

The assignment instruction can often be merged with the previous instruction.

Program	Generated Quadraple	Peephole
$a = x+y$	$t_1 = x+y$ $a = t_1$	$a = x+y$.

Unreachable Codes :-

← Should be removed.

```
goto L2
x := x+1   ← No need.
```

Simple algebraic identities like :-

$x*1 = x$; $x = x+0 = 0+x = x$ can also be used.

Generated Quadruples: $t_5 = t_4 * 1$.

Peephole Optimised: $t_5 = t_4$.

Repeated Jumps :

Generated Quadruple:

(1) if false (t1) goto (8)

....

goto (16).

Peephole Optimised:

if false (t1) goto (16).

Dead Code :

x := 32 ← x is further not used after statements

y := x + y ⟶ y := y + 32.

Basic Blocks and Flow Graphs:

— Basic blocks introduces graph representation of intermediate code, helpful for code generation.

— The representation is constructed as follows:

1) Partition the intermediate code, into basic blocks., which are maximal sequence of consecutive 3-addr. instruction with following property.

- Flow of control can only enter the basic block through the first instruction of the block.
- Control will leave the block, without halting/branching except the last ins. of the block.

2) The basic block becomes the nodes of the flow graph. whose edges indicate which blocks can follow which other block.