Compiler Design

(1) Lexical Analysis.
(2) Syntax Analysis
(3) Syntax Directed Translation
(4) Intermediate Code Generation
(5) Runtime Environment
(6) Code optimization

Book : Aho, ullman, Sethi

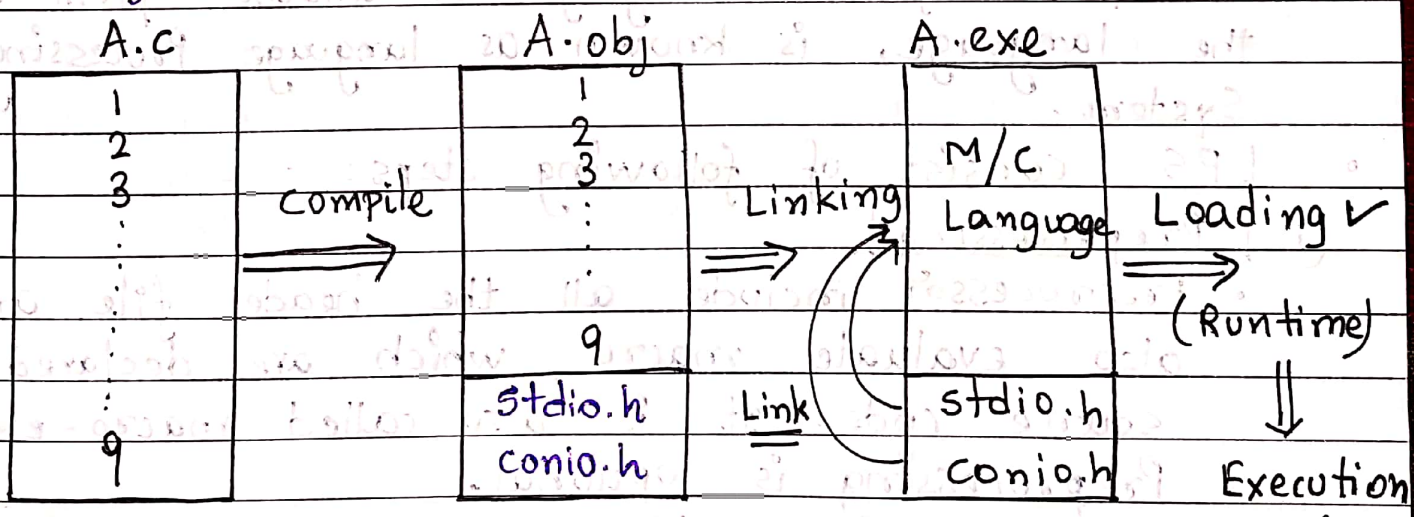
- Compiler is a Language translator.

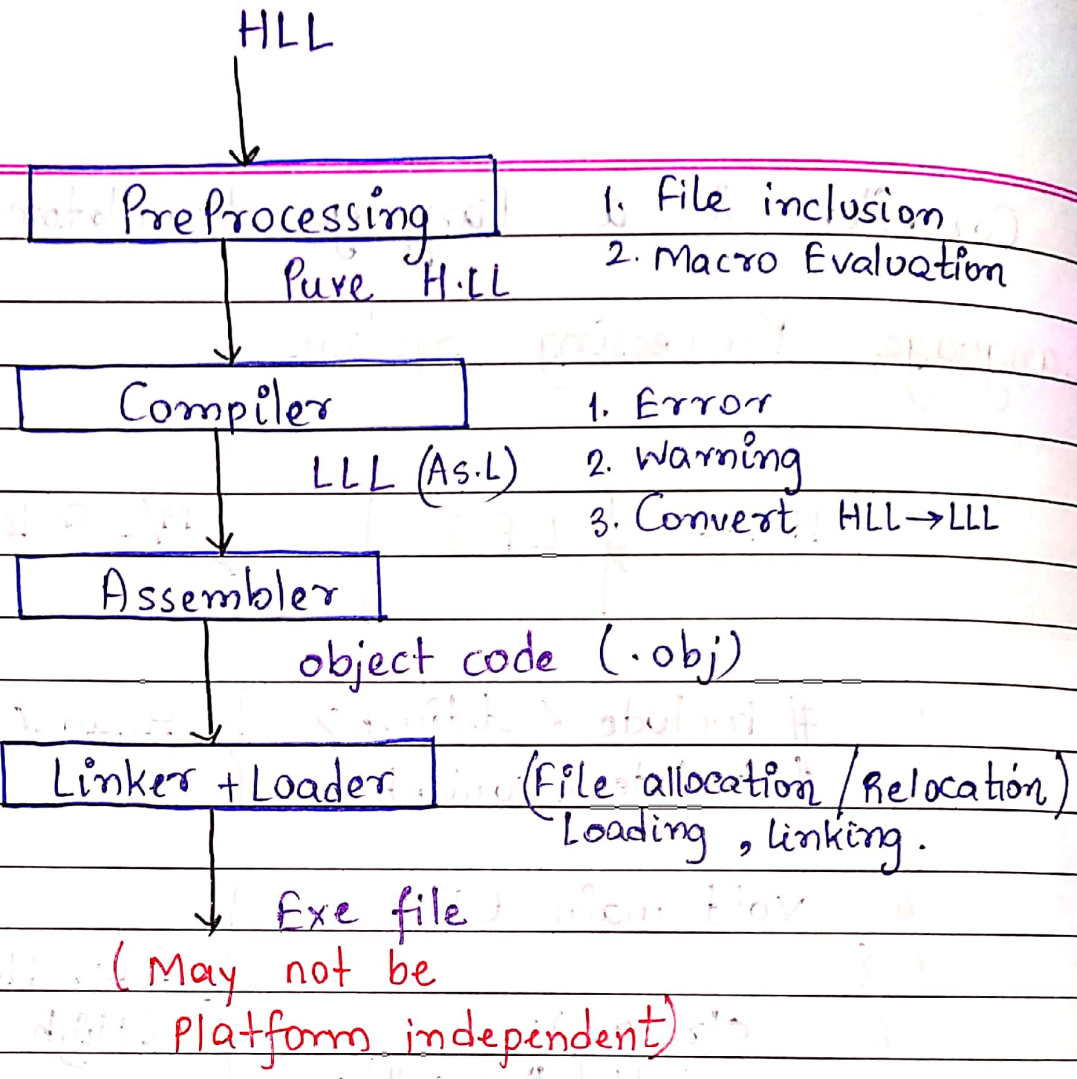
Language Processing System


```
1.  # include <stdio.h>
2.  # include <conio.h>
3.
4.  void main ()
5.  {
6.      clrscr();
7.      printf("Hello");
8.      getch();
9.  }
```

Header files

Pure H.L.L (High Level Language)





- A system program which convert source code from one form of the language to another form of the language, is known as language Processing System.
- LPS consist of following steps:

(i) **Preprocessing**

- Preprocessor include all the header file and also evaluate macro which are declared in source code. It is also called macro-evaluator.
- Preprocessing is optional.
- The language which does not support ~~has~~ #include or macro, does not require pre pocessing.

Ex: Pascal.

(2) **Compiler** : It checks the syntax error. If everything is OK, then convert H.L.L to Low level language.

- Compilation is optional, if the source code is directly written in low level language then no need of compiler.
- for example 8085 and 8086.

(3) **Assembler** : It is a translator, converts assembly language into ~~exe~~ object code. (Not executable code). and it is stored in a temporary file, and store in an address provided in an operating System.

(4.) **Linker or Loader** : It is used for allocation and relocation. Allocation means getting the memory partition fom O.S to store the object data.

- Rellocation means rellocation address mapped with physical address and moving object code to physical address after converting into exe code.

- Linker is a system program that combines all files into a single exefile.

- Loading loads the exe file into permanent memory.

- Difference between Compiler and interpreter :-

Compiler	Interpreter
• Input high level, output Low level.	• Input HLL output result.
• Process is translation	• Process is execution.
• Compline scans the entire text at a time.	• Scan the text line by line.
• Process is very fast	• Interpreter is very slow
• Requires heavy memory and less time	• Requires low memory. and more time
• code optimization	• Direct execution hence optimization not possib
• Static scope,	• Dynamic scope.
• FORTRAN is first language to have compiler	• Basic is the first language to implemen interpreter.
• C, C++, Java requires compilation	• Cmd prompt langua like SQL, Prolog, Python, smalltalk, lisp are interprete language.

Challenges to Design a Compiler

- Compiler writing is not an easy task. It is very challenging and you must have a lot of knowledge of various feilds of computer science.

- (i) Many variations:
  (a) Language level variation, Ex C, C++, java, .net...
  (b) Programming principle level variation, for exampl top-down and bottom up approach.
  (c) operating System level, for ex. linux, windows.
  (d) Hardware Level, Ex. intel, AMD, Qualcomm.

(ii) Quality of a good compiler. It must generale correct machine code and it must run fast. compiler must translale fast. Compiler must show error with line number.

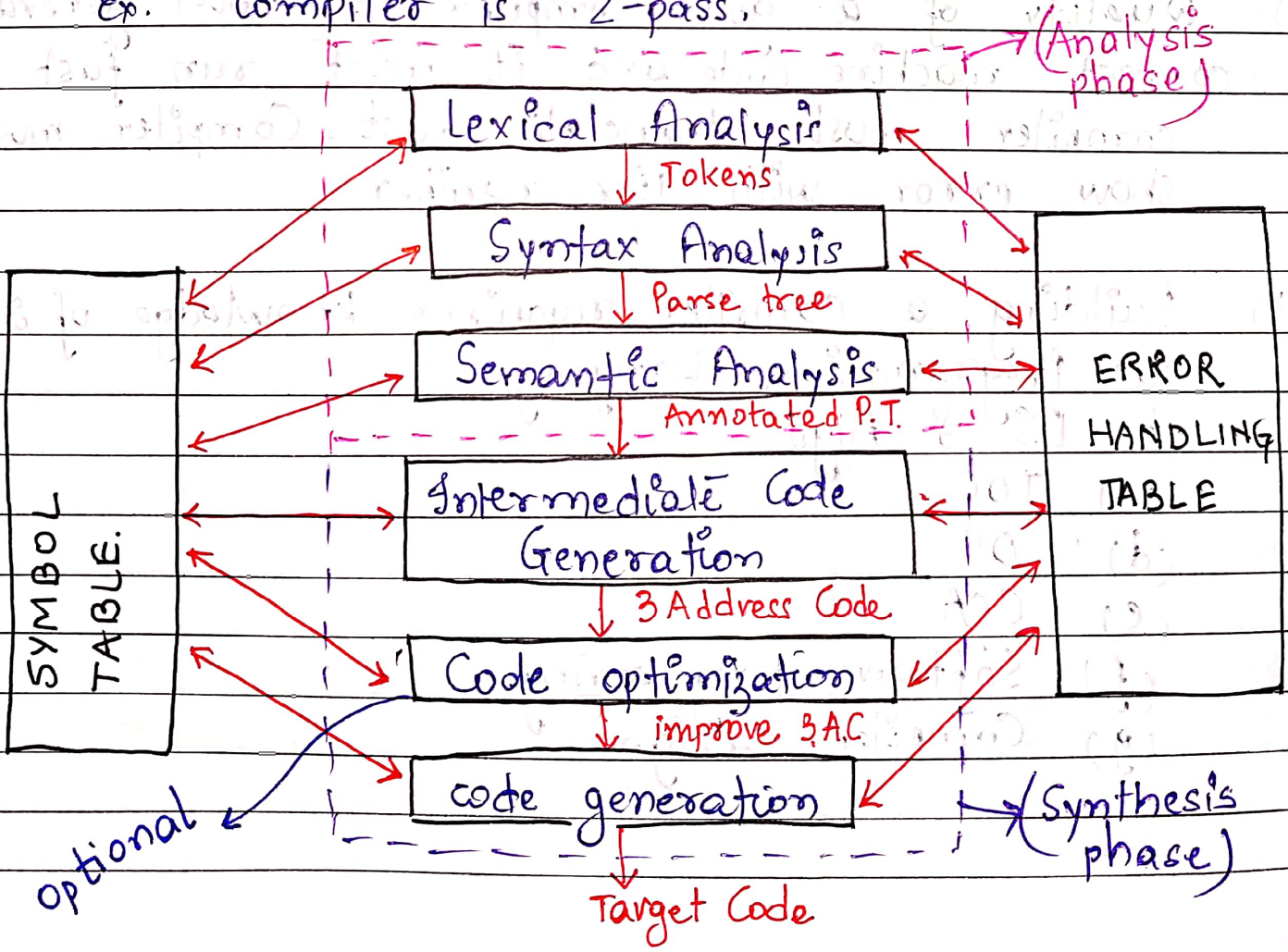
(iii) Building a compiler requires knowledge of:
  (a) Programming Language.
  (b) DS & Algo
  (c) TOC
  (d) OS
  (e) CSA
  (f) Software Engineering
  (g) Compiler Design.

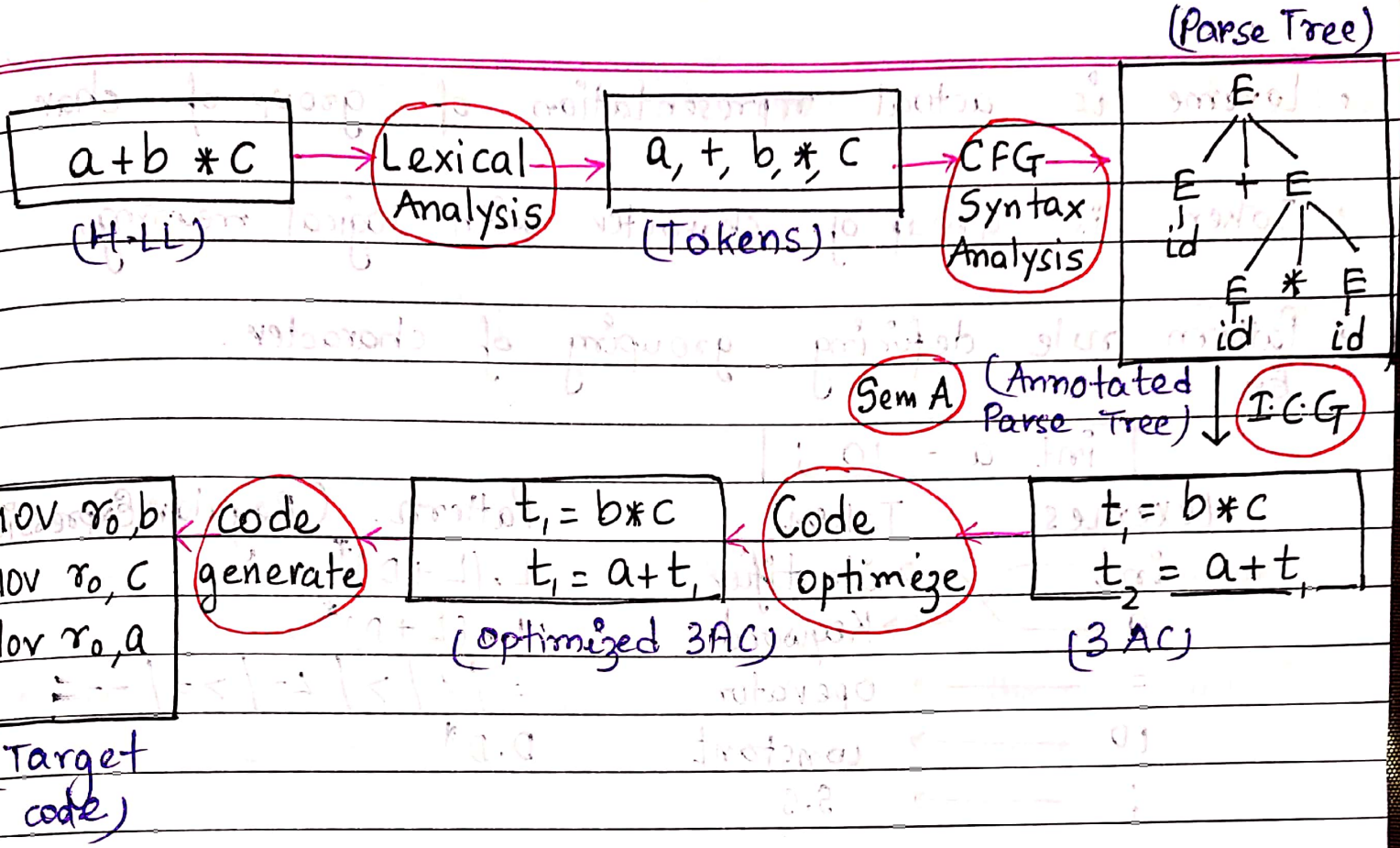
Generic Structure of Compiler

- Compiler is a very complex program, It is divided into phases. Each ~~face~~ phase take one representation of source program and output another representation.

- The conversion of source code from one form of representation to another form of representation is called phase.

Pass: It is the number of iteration required to scan the source code during the compilation Ex. compiler is 2-pass.

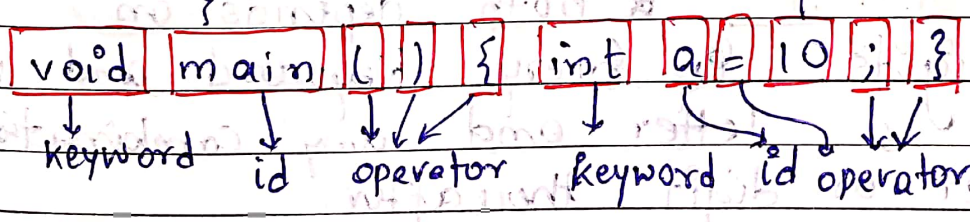




LEXICAL ANALYSIS PHASE

- The source code is scanned and grouped into lexeme and then generate the token. i.e output of lexical analyzer is stream of tokens.
- Tokens are stored in the symbol table.

```
void main()
{   int a=10;
}
```



{Lexemes}.
↓
FA Accept
↓
Tokens.

- Lexeme is actual representation of group of char
- Token is group of character with logical meaning.
- Pattern rule defining grouping of character.

Ex.

int a = 10 ;

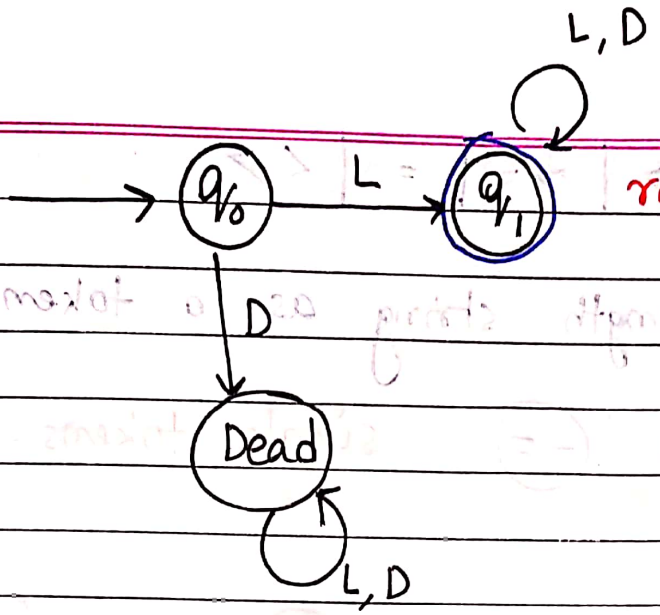
Lexemes.	Token	Pattern: (Regular Expression)
int	identifier	L.(L+D)*
a	Keyword	L.(L+D)*
=	operator	=/</>/<=/>=/--
10	constant	D.D*
;	S.S	

- Lexical Analyser Design: LA can be designed in hand code, lex tool. etc.

Hand Code:

(1) Design rules for pattern using regular expression.
(2) Construct the regular expression based on pattern.
(3) Convert R.E to F.A
(4) Acceptance of tokens by FA.

Keyword & Identifier: Both depends on letter and digits. Every Keyword and id begins with a letter and any combination of letter or digit thereafter.



return (gettoken(), install_id())

gettoken() → present in symbol table?

install_id() → create new record.

- The procedure install_id has access to the buffer where the lexeme has been located.
- The symbol table is examined. If the lexeme is found as a keyword, then install_id returns 0.
- If the lexeme is found in symbol table, and it is a program variable, then install id will return a pointer to the symbol table.
- If the lexeme is not found in the symbol table, then it installed in the symbd table and a pointer to the newly created entry is returned.

int a;

return

100

Name	Type	Scope	Life
a	int	local	local

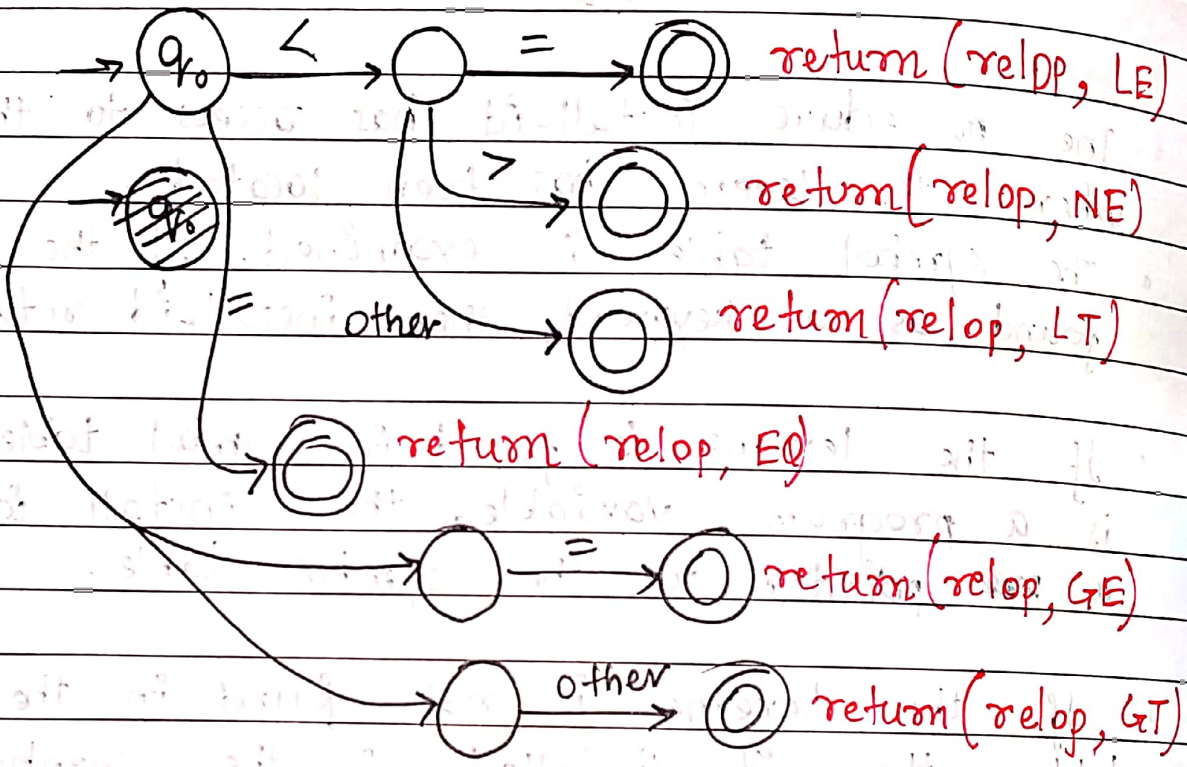
← entry for a.

Symbol table

- RE = < | <= | > | >= | = | <>
- taking maximal length string as a token.

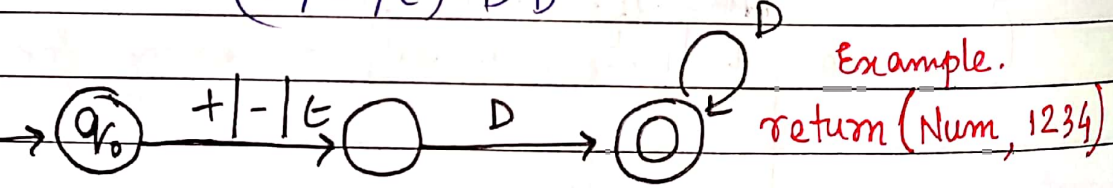
++ -- << += -= single tokens.

Finite Automata



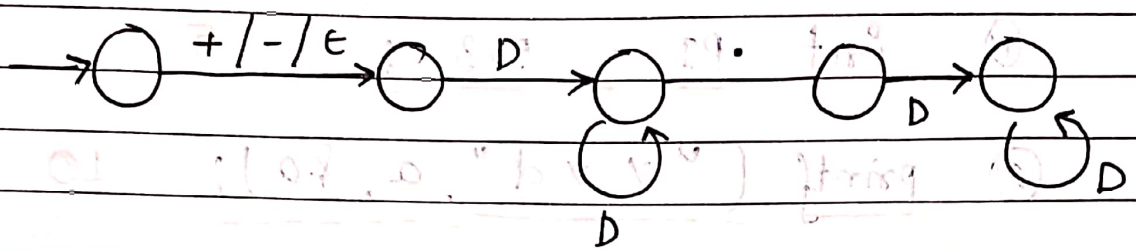
Integer Numbers: Combination of digits D = 0 | 1 | 2 -- | 9

R.E = (+ | - | ∈) D D*



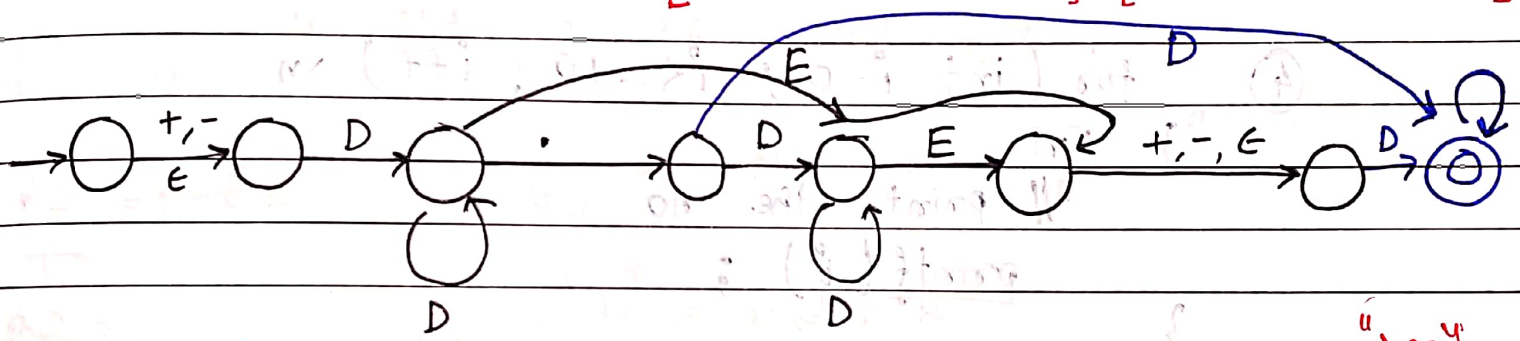
Example.
return (Num, 1234)

Floating Point : RE = (+/-/ϵ) D.D*(.) DD*.



Exponent : 12.56 E+10.
12 E-10

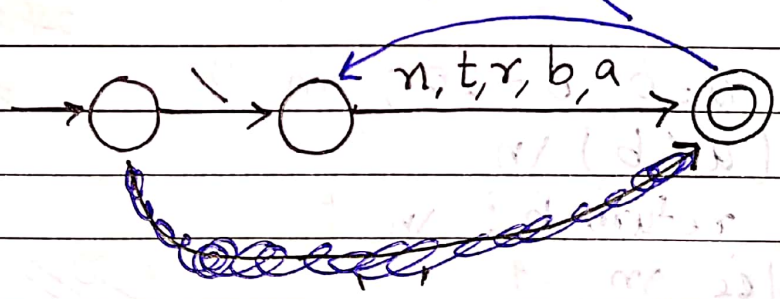
R.E : (+/-/ϵ) DD* [((.) D.D*) /ϵ] . [E(+/-/ϵ)DD* / ϵ]



"\n"

0.5 E 1.2 Invalid token

White spaces : W=(\n | \t | \r | \b | \a)+



Example
return (ws, \b)

R.E = W+

Ques.) Find the number of tokens :

① int PI = 5.3 ; 5

② printf (" %.2d " , a , &a) ; 10

③ if (x > y) \n
{ \n
pf (" CD ") ; \n
}

16 - 3 = 13

④ for (int i=0 ; i<=10 ; i++) \n
{ \n
// print the no \n
printf (i) ; \n
}

28 - 4 = 24
4
= 20

⑤ int main ( ) \n
{ \n
int a=0 b=20 ; \n
if (a<b) \n
return b ; \n
else \n
return a ; \n
}

⑥
```
int main ()
{   // print msg
    pf ("\"S", "G+H");
}
```
13

⑦
```
int main ()
{   a = b+c;
    pf ("____", a, &a, **b, a&&c);
    getch ();
}
```
32

⑧
```
main ()
{
    int x=10, *p;
    int y = x++;
    char *q;
    p=&x; q = 'A';
    if (*p >= 10)
    {
        *p = x+100;
    }
    else
    {   pf ("___", x);
        /* comment */
    }
}
```
-3

18 22 31 38 46 56

58

Ques) Which of the following is not a token in C language?

(A) int - keyword ✓
(B) main - id ✓
(C) "gate" - string. ✓
(D) None

Ques) Find invalid Preprocessor command in C?

(A) #define
(B) #include
(C) #if
(D) None

PARSING :

- Parser will verify syntax of grammar (set of rules)

int a = 10 ; // No lexical error

Nit a = 10 ; // No Lexical error

int 2ab ; // Error.

↓

Not defined as token (id begin with Letter)

- a = a + b ; // No Lexical Error.
  // No Syntax error.

a = % b ; // No lexical error.
// Syntax error.

- E → E * E | E + E | + E | - E | id

w = a + - b ; // valid, No lexical error

int a $ b = 10 ;

LE error if $ is not an operator.

a = $ b ;

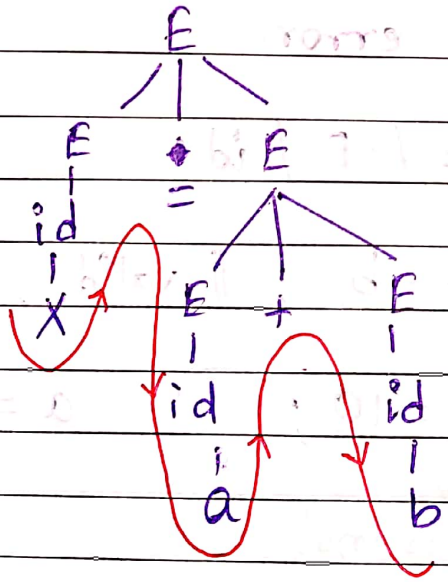
↳ Illegal operator.

Token → Parser → Parser.

- The parser takes the input in the form of stream of tokens and that input is recognized / regenerated, from the context free grammar, of language. Then there is no grammatical mistake.
- If that input is not re-gerated by the grammar then there is some grammatical mistake, and result is syntax error.

- The parser will take care only of formation of sentence, rather than meaning of sentence. since it does not consider type of identifier for parsing.

X = a+b



- for parsing, type of identifier is not required.

SEMANTIC ANALYSIS PHASE

- In this phase, we verify the meaning of each and every sentence apart from parsing.

```
int a, b;
bool c;
c = a + b;
```

bool int

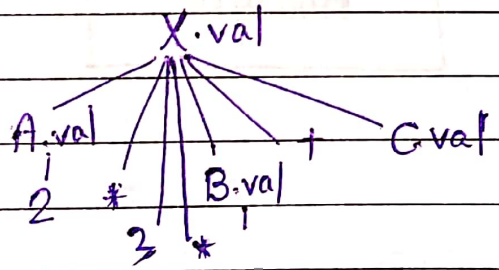
semantic Error.

Symantic Rules for CFG $E \rightarrow E_1 = E_2$

{ If $E_1.type = E_2.type$
$E_1.val = E_2.val$
else
return Error
}

$X \rightarrow ABC$ Rule { $X.val = A.val * 2 * B.val + C.val$ }

$X = A * 2 * B + C.$



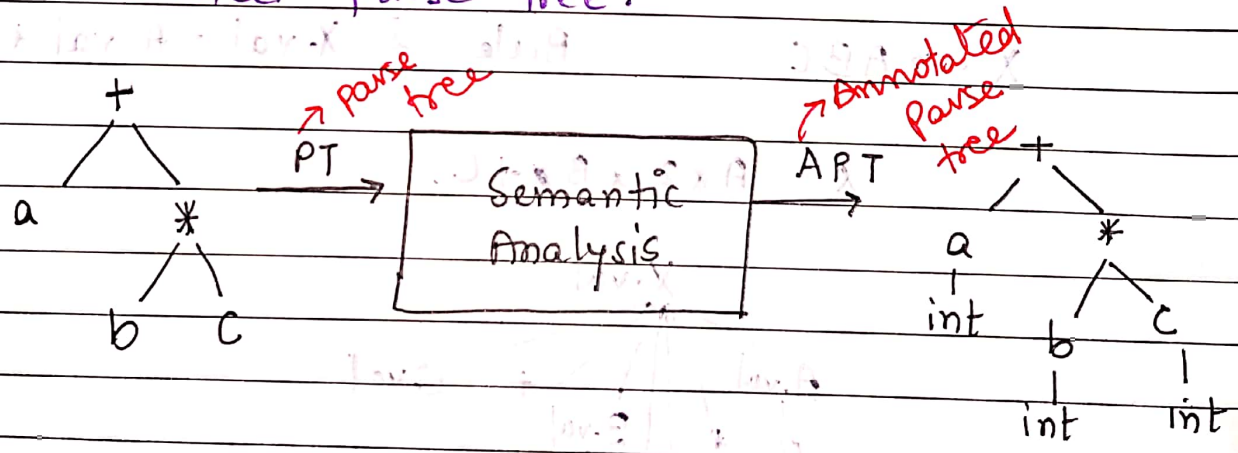
Ex $x = 2 * 3 * 3 + 5$

- In Semantics, we perform the type checking, we declare variable, verify each and every semantics of source code.

```
int a, b;

z = a + b      # No LE, No Syntax error
?   int int
```

- Semantic Analysis can be performed by attaching the semantic action for each and every grammar rule.
- Grammar with Semantic action is called Syntax directed translation. It is a kind of technology to carry out the semanticness of source code.
- for the semantic analysis, input is parse tree, o/p annotated parse tree.



INTERMEDIATE CODE GENERATION

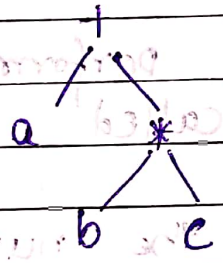
- Source code is converted into intermediate representation to improve performance of code generation, and to make code generation process

simple and easy.

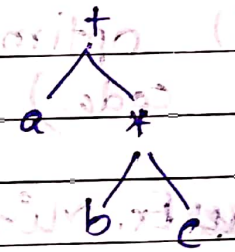
a+b*c

IC (Intermediate Code)

- Linear form
  - Postfix: abc*+
  - Three Address Code:
    $t_1 = b * c$
    $t_2 = a + t_1$
- Tree form
  - Syntax Tree



  - DAG (Directed Acyclic Graph)

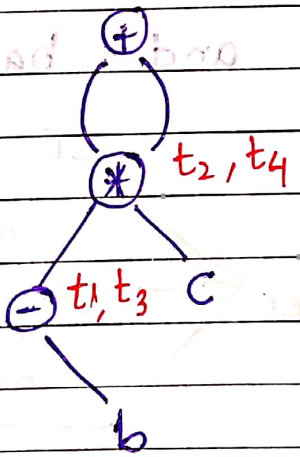


$X = (-b * c) + (-b * c)$

t_1 t_2 t_3 t_4

$t_1 = -b$
$t_2 = t_1 * c$
$t_3 = -b$
$t_4 = t_3 * c$
$X = t_2 + t_4$.

optimize ⟹ (DAG)



$t_1 = t_3$
$t_2 = t_4$
(Delete t_3, t_4)

$t_1 = -b$
$t_2 = t_1 * c$
$X = t_1 + t_2$.

CODE OPTIMIZATION

- The process of reducing the instruction without effecting the outcome of source code is known as optimization.

- There are two types of code optimization.

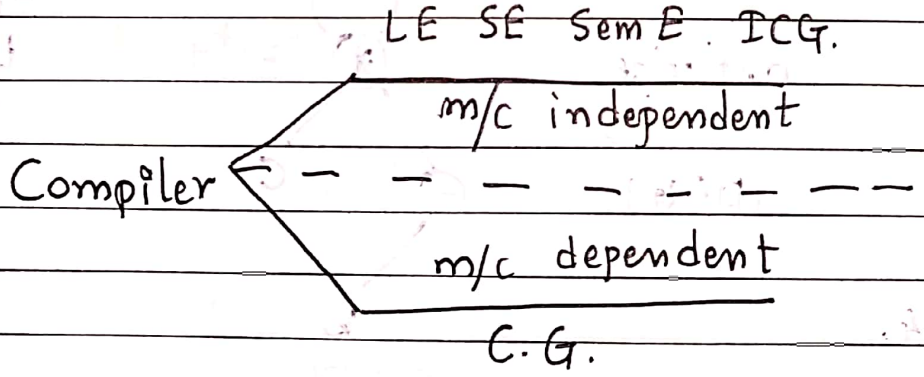
(i) Machine independent optimization: optimization performed on 3 address code is called m/c independent optimization.

(ii) optimization performed on target code (assembly code) is called m/c dependent optimization

Ex. Minimize the number of register

- Optimization phase divides compiler into two parts.

- front end and back end.



TARGET CODE GENERATION :

- The optimized and unoptimized source code will be converted into assembly language code. which is the target code.

Ex. $a+b*c$.

$t_1 = b*c$ ⟹ optimize ⟹ $t_1 = b*c$

$t_2 = t_1 + a$ $t_1 = t_1 + a$.

Target Code.

```
MOV  r0, b
MUL  r0, c
ADD  r0, a
```

Ex. Position = initial + rate * 60

LA: id_1 op_1 id_2 op_2 id_3 op_3 $const_1$

SA:

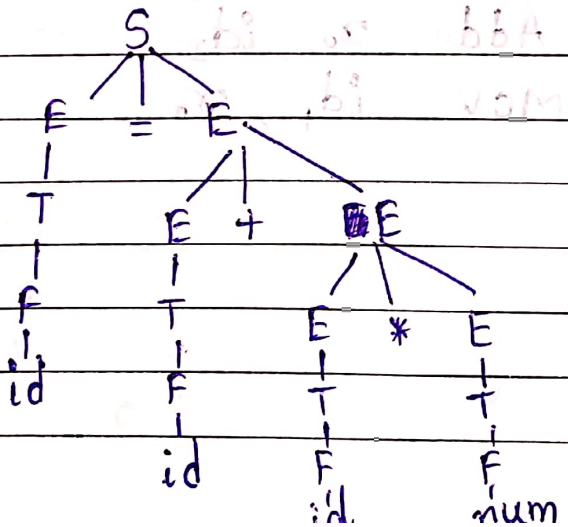
E → E+E / T

E → E*E / T

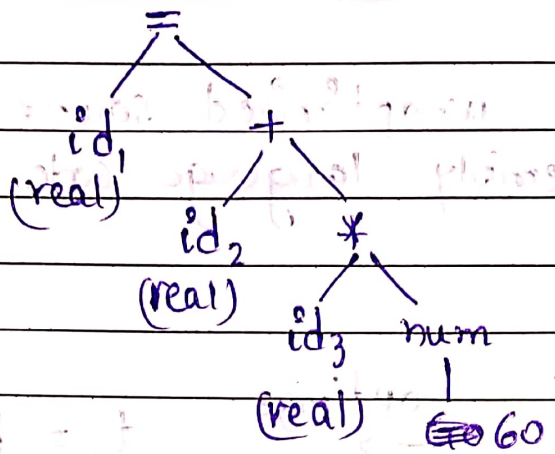
F → Num / id

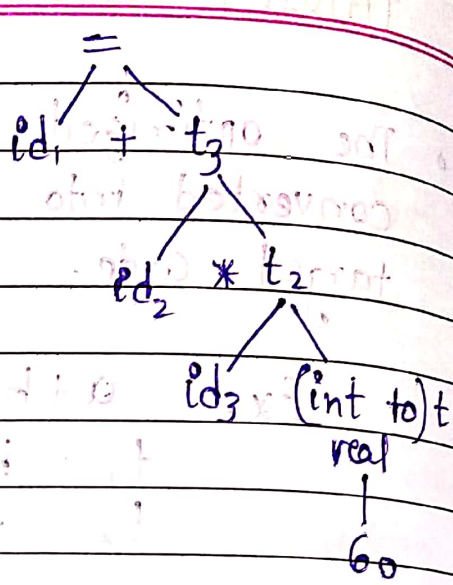
T → F

S → E = E



Sem A





(ICG)
$t_1 = int\_to\_real(60)$
$t_2 = id_3 * t_1$
$t_3 = id_2 + t_2$
$id_1 = t_3$

(CO)
$t_1 = int\ to\ real\ (60)$
$t_1 = id_3 * t_1$
$t_1 = id_2 + t_1$
$id_1 = t_1$

(CG)
MOV r_0 int to real(60)
MUL r_0, id_3
Add r_0 id_2
MOV id_1, r_0

Ex a[i] = 2+5 Home work.

- There are two object which are working along with the compiler.
  (i) Symbol table
  (ii) Error Handling table.
- It is a datastructure provided by the compiler, to store the complete information of source code.
- It is responsibility of compiler to provide memory for symbol table. (Dynamic)
- All the phases of compiler are interacting with symbol table.
- At any phase if any new variable is encountered, that variable is stored in the symbol table.
- Lexical Analysis is the first phase to communicate with the symbol table, hence during the lexical analysis phase, Symbol table is generated by compiler.
- During the first two phase, we store the information into symbol table and during the remaining phase, we make use of information available in symbol table.

Information stored in the symbol table:

- The following information is stored about identifier.
  - (i) Name
  - (ii) Type
  - (iii) Size of identifier
  - (iv) Location, address of identifier
  - (v) Scope
  - (vi) Life
  - (vii) value
  - (viii) other information in case of complex datatype (array, union, structure) etc. record, procedure

Operations on Symbol table :

- There are 4 operation that can be performed on symbol table.
  - (i) Insert
  - (ii) Lookup
  - (iii) Modify
  - (iv) Delete

Capability of symbol table.

- It should be able to enter the symbol in the table, and return the address.
- It should allow to search the symbol table to check whether a particular symbol already exist in the table, if it should return the address of the entry.
- It should be able to delete some element or group of element from the table.
- The search time must be as fast as possible. ∴ the entire table must be in memory.
- Symbol table should be able to grow as symbol table are added to it.
- It must support duplicate entries in different scope.

Implementation of symbol Table :

- Symbol table can be implemented using any one of the following datastructure.

- Linear table
- Linked list
- hash table

} DS

Gate 2010 Which DS in the compiler is used for managing information about variable and attribute? Symbol table.

Error Handling Table :

- Detection and reebooting of errors in source program is main function of compiler. Error may occur at any phase of compilation. A good compiler must determine the line number of program exactly where error have occured.

(i) **Lexical Error:** These are errors occured due to the declaration of the variable.

```
int a = b;     // No error
int $ = c;     // Error
printf ( Hello ); // No error
```

(ii) Syntax Error: There are error due to grammati mistake.

```
int a=10 ;  // No LE  No SE
nit a=10 ;  // No L.E  But S.E.
printf ( Hello );  // No L.E  No SE
pri ( Hello );  // No errors.. ( Normal function syntax)
for ( ; ) ;  //S.Error  No lexical error
for ( , , )  // S.E
while ( )  // SE
a = +a ;  // No LE, No SE
a = %a ;  // SE
```

```
int a;
bool b;
z = a+b;
```

(iii) Semantic Error: Error due to meaningless sentence or using variable without declaration.

```
int a;  // No L.E  No SE
bool b;  But Semantic Error.
c = a+b
```

Ex int a, b; // Semantic error.
c = a + b; (Type checking).
↓
type missing

Ex int *P1 *P2;
P1 = P1 + P2; // Semantic Errors
P1 = P1 * P2;
P1 = P1 / P2;
P1 = P1 % P2 %

Ex int a[10];
++a; a is array.
a++; Base address can't
a--; be modified.
--a;

Ex int x = "Yellow"; // NoLE
semantic No SE
Error.

- Lexical, syntax and Semantic action can be ended by programmer.

Code optimization Error: There can be error, Ex in control flow analysis, there exist some statement, which can never be reached.

- The 5th type of error may encountered during code generation.
  Ex. there may exist a constant that is too large, to fit in a word of the target machine.

- The sixth type of error may encounter when try to make symbol table entry for example, there may exist some identifier that has been multiple declaration in the same scope.

- There are some error which are occured during execution of program. They are called fatal error
  Ex memory insufficient

- No input output stream.

Ques) Find the type of error produced by the following code:-

```
main ( )
{  int  /* Comment
      float  /* comment. */ t gate ;
```

(Remove comment creates space)

```
      int t gate ;
```

// Syntax error

(A) L.E
(B) S.E
(C) Both
(D) None.

Ques:) Which of the following is not used as a token seperator during L.A phase?
(A) white space.
(B) comment.
(C) semicolon ?
(D) None.

Ques:) Find the statement which has syntax error ?
(A) fi(z);
(B) for(a,b,c); ✓
(C) while(a);
(D) All

Ques:) Three Address code is also known as
(A) Abstract code
(B) Intermediate code
(C) machine independent code
(D) ALL ✓

Ques:) Which of the following functionality is not of C compiler.
(A) Identifying token
(B) Identification of syntax error
(C) Linking ✓
(D) None.

Ques.) Which of the following is valid RE of identifier.

(A) $(L + D)^*$
(B) $D \cdot (L + D)^*$
(C) $L \cdot (L + D)^*$ ✓
(D) $L \cdot (L + D)^+$

Ques.) Type checking is normally performed by
(A) lexical
(B) Syntax Direction translation (Semantic) ✓
(C) Syntax
(D) Code optimization

Ques.) Which is NOT M/C machine independent phase of compiler?

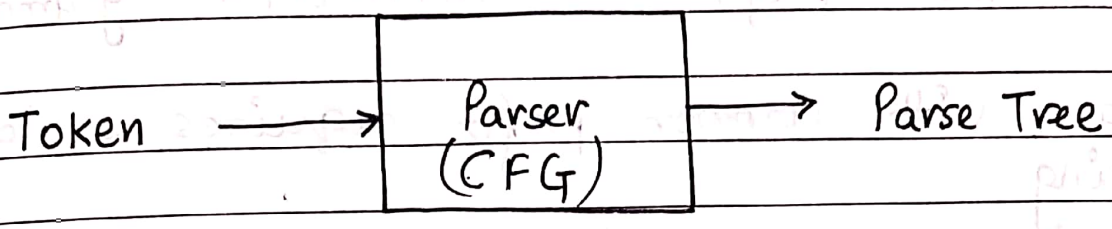
(A) Syntax Analysis
(B) Code Generation ✓ (Not in case of Java)
(C) Lexical Analysis
(D) Intermediate code generation

Difference between Single Pass & Multipass

- No of iteration required to scan the source code during the compilation is no of Pass.

Single pass	Characteristics	Multipass
High	Memory Requirement	Low (load one by one phase)
faster	Time	slower (perform read and write operation for intermediate result)
one pass	# Passes	Atleast 2 passes
single	# Phases	Multiple
low	Flexibility	High
low	Efficiency	high
low	Code Reusability	high

UNIT-2 PARSER



- The process of construction of parse tree for the given input string is known as parsing.
- The implementation of parse tree is parser
- To perform syntax analysis on the source code, the source language must be defined by context free grammar.
- CFG, Type of derivation, LMD, RMD
  Type of CFG : • Based on string → Rec (Dir → LR, Indir → RR), Non Rec
  • Based on Parse Tree
- Problem of left recursion
- LRG to RRG ✓

Non-Deterministic Grammar :

- Grammar with common prefix is known as non--deterministic grammar

$$S \rightarrow abc \mid abd \mid abe$$

Backtracking increase → parsing slow.

Left-Factoring : To avoid backtracking, we need to convert common prefix into deterministic Grammar.

- Grammar with common prefix requires a lot of backtracking.
- The process of conversion of Grammar with common prefix into deterministic Grammar (without common prefix) is known as left factoring.

$$S \to \alpha\beta_1 \mid \alpha\beta_2 \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \gamma_m$$

$$S \to \alpha S' \mid \gamma_1 \mid \gamma_2 \ldots \mid \gamma_m$$

$$S' \to \beta_1 \mid \beta_2 \ldots \mid \beta_n.$$

Ex. $S \to abc \mid abd \mid abe.$

$$S \to abS'$$
$$S' \to c \mid d \mid e$$

Ques Remove common prefix of the following grammar.

$$S \to aAb \mid abB \mid abcd \mid abcde$$
$$A \to \epsilon \mid e$$
$$B \to \epsilon \mid f$$

$S \rightarrow aAb \mid abB \mid abcdS'$

$S' \rightarrow \epsilon / e$

$A \rightarrow \epsilon / e$

$B \rightarrow \epsilon / f$

$S \rightarrow aAb \mid abS''$

$S'' \rightarrow B \mid cdS'$

$S' \rightarrow \epsilon / e$

$A \rightarrow \epsilon / e$

$B \rightarrow \epsilon / f.$

$\Rightarrow$

$S \rightarrow aS'''$

$S''' \rightarrow Ab \mid bS''$

$S'' \rightarrow B \mid cdS'$

$S' \rightarrow \epsilon / e$

$A \rightarrow \epsilon / e$

$B \rightarrow e \mid \epsilon$

Alternate :

$\Rightarrow S \rightarrow aS'$

$S' \rightarrow Ab \mid bB \mid bcde \mid bcd$

$S' \rightarrow Ab \mid bS''$

$S'' \rightarrow B \mid cde \mid cd$

$S'' \rightarrow B \mid cS'''$

$S''' \rightarrow de \mid d$

$S'' \rightarrow dS^{IV}$

$S^{IV} \rightarrow \epsilon \mid e$

Ques $S \rightarrow iEtS \mid iEtSeS \mid \epsilon$

$E \rightarrow b.$

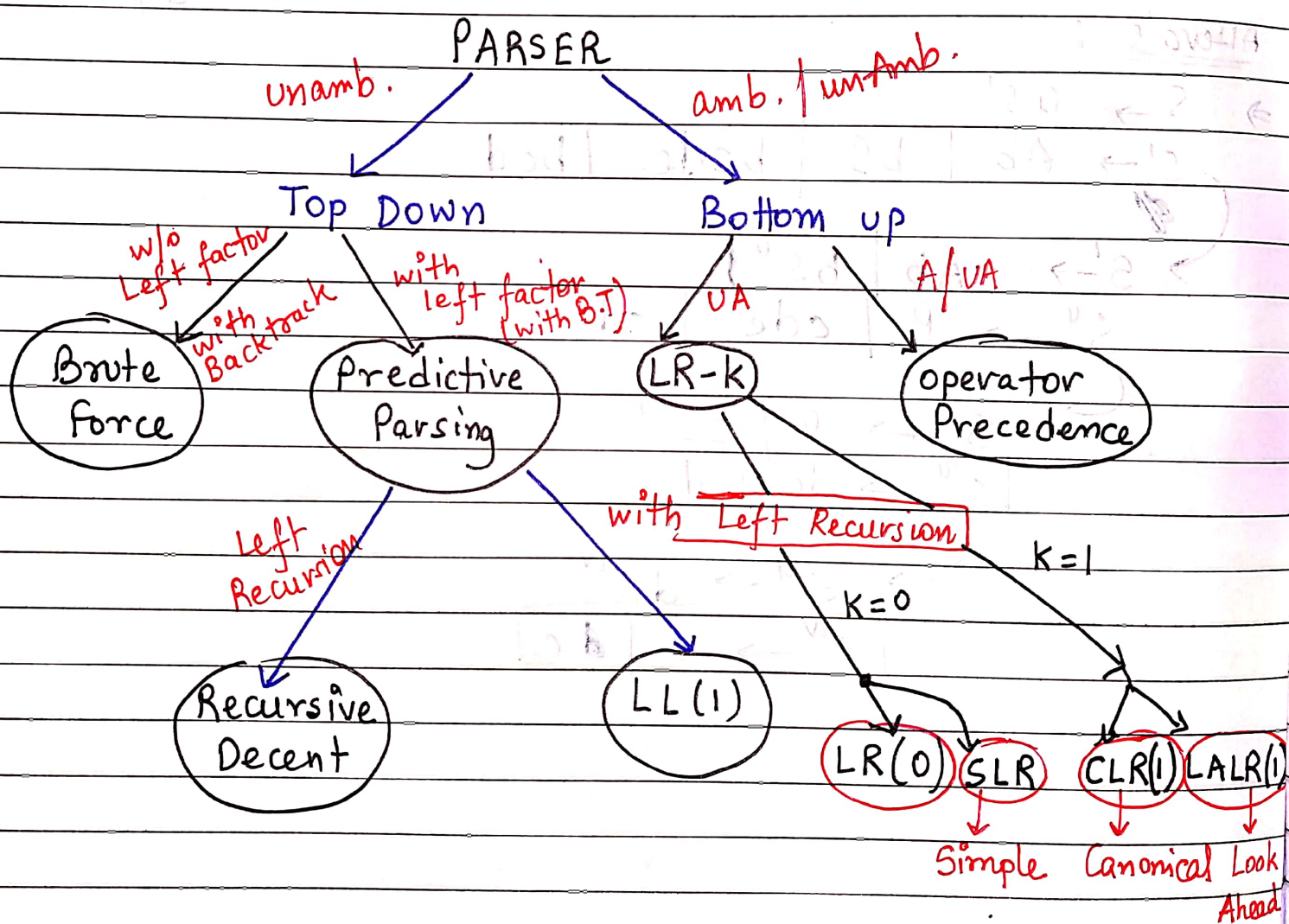
$S \rightarrow iEtSS' \mid \epsilon$

$S' \rightarrow \epsilon \mid eS$

$E \rightarrow b$

Ques [Type of Parser]

- Top Down parser
- Bottom up parser

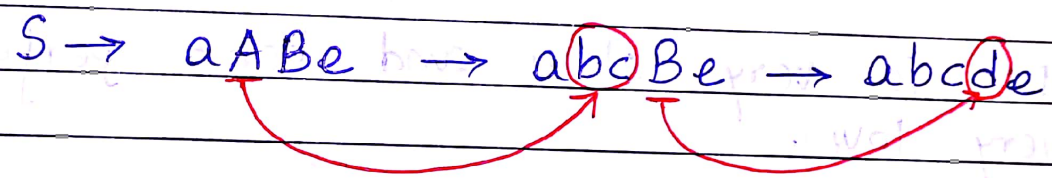


Top-Down Parser

- The process of construction of parse tree starting from root and working upto input string is known as top down parser.

Ex → S → aABe
A → bc
B → d

w = abcde.

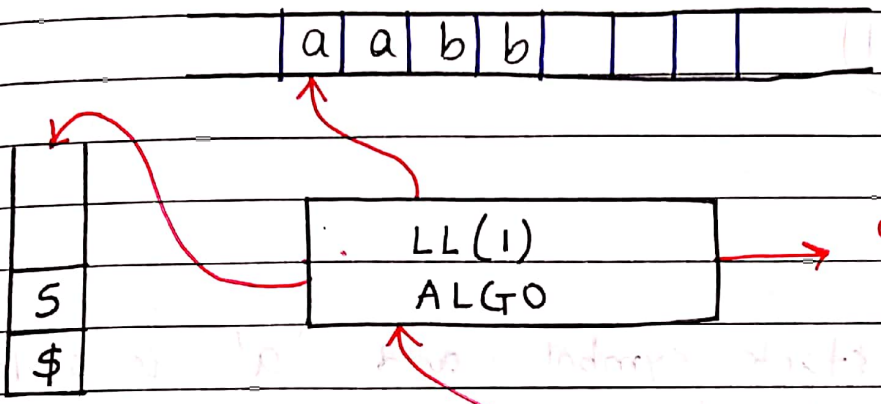


Note : Top down parser internally use leftmost derivation

- TDP can be constructed for grammar which may or may not be left factor.
- Hence TDP can be constructed with and without back tracking.
- TDP without backtracking is called as predictive parser.
- Predictive Parser can be constructed for recursive and non-recursive Grammar.

- Recursive Predictive parser is constructed only for Rec. Grammar only. (May be Left or Right recursive).
- Left recursion may cause looping of parser.
- Non-recursive predictive parser is constructive for the grammar which is free from left recursion, ambiguity and left factor. Hence this parser is more powerful and efficient than other kind of top-down parser.
- TDP is very slow and hence performance is very low.
- Top down parser are constructed for grammar which has less complexity.
- Average time complexity is $O(n^4)$.

LL(1) PARSER



NT \ T	T's	$
N-T's	Productions	

Parser Table.

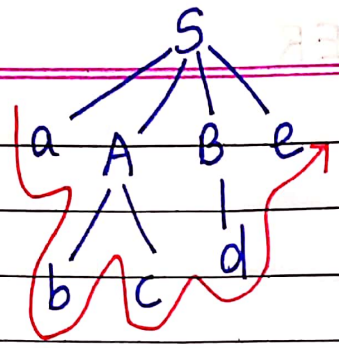
m x n

m → No of Non-Terminals

n → No of Terminals + 1 (for $)

Ex: S → aABe
A → bc
B → d

SNo	Stack		Input	Operation	
1	$ S		abcde$	S → aABe	(1)
2	$ eBA a	match →	a bcde$	Pop	
3	$ eBA	×	bcde$	A → bc	(2)
4	$ eBcb	✓	bcde$	Pop	
5	$ eBc	✓	cde$	Pop	
6	$ eB	×	de$	B → d	(3)
7	$ ed	✓	de$	Pop	
8	$ e	✓	e$	Pop	
9	$		$	Success. ✓	



LL(1) ALGO:

- Let X be a stack symbol and 'a' is a lookhead symbol of input string.

(1) if X = a = $ then parsing is successful.

(2) if X = a ≠ $ then pop and increment the input pointer.

(3) if X ≠ a ≠ $ then, and M[X, a] contain the production X → uvw then replace X in reverse order.

(4) output the production which is used for expanding the parse tree.

FIRST and Follow functions:

FIRST : First(α) : α ∈ T's / NT's contains all the terminals that may begin in any right hand side of the grammar.

(i) If α is terminal then $First(\alpha) = \{\alpha\}$
[$First(a) = \{a\}$]

(ii) If α is non terminal, then if $\alpha \to \epsilon$ then simply add ϵ in [$First(\alpha) = \{\epsilon\}$]

(iii) If α is non-terminal and $\alpha \to aX$ then
[$First(\alpha) = First(aX)$] $= \{a\}$

Ex $S \to aSb$.
$First(S) = \{a\}$

(iv) If α is non-terminal and $\alpha \to X_1 X_2 \cdots X_n$
If X_1 is NT and $First(X_1)$ does not have ϵ
then [$First(\alpha) = First(X_1)$]

If X_1 is NT and First of X_1 contain ϵ,
then [$First(\alpha) = First(X_1) - \{\epsilon\} \cup First(X_2 X_3 \cdots X_n)$]

Ques) find first :

① $A \to a \mid (A)$ $\quad First(A) = \{a, (\}$ ✓

② $S \to aAB$ $\quad First(S) = \{a\}$ ✓
$A \to bA \mid c$ $\quad First(A) = \{b, c\}$ ✓
$B \to c \mid d \mid \epsilon$ $\quad First(B) = \{c, d, \epsilon\}$ ✓

(3) $S \to aAB \mid b$ — first(S) = {a, b} ✓
$A \to bA \mid Bd$ — first(A) = {b, c, d} ✓
$B \to c \mid d$. — first(B) = {c, d} ✓

(4) $S \to aAB \mid b$ — first(S) = {a, b}
$A \to bA \mid Bd$ — first(A) = {b, c, d, ϵ} ✓
$B \to c \mid d \mid \epsilon$ — first(B) = {c, d, ϵ}

(5) $A \to (A) \mid b \mid a \mid \epsilon$ — first(A) = {(, a, b, ϵ} ✓

(6) $S \to aSb \mid bSa \mid \epsilon$ — first(S) = {a, b, ϵ} ✓

(7) $S \to AB$ — first(S) = {a, ϵ} ✓
$A \to aA \mid \epsilon$ — first(A) = {a, ϵ}
$B \to \epsilon$ — first(B) = {ϵ}

(8) $S \to (L) \mid a$ — first(S) = {(, a} ✓
$L \to SL'$ — first(L) = {(, a}
$L' \to ,SL' \mid \epsilon$ — first(L') = {, , ϵ}

(9) $S \to AA$ — first(S) = {a, b, ϵ} ✓
$A \to aA \mid b \mid \epsilon$ — first(A) = {a, b, ϵ}

(10) $S \to ABCDE$ | $A \to a \mid \epsilon$ | $B \to b \mid \epsilon$ | $C \to c \mid \epsilon$ | $D \to d$ | $E \to e \mid \epsilon$

first(S) = {a, b, c, d}
first(A) = {a, ϵ}
first(B) = {b, ϵ}
first(C) = {c, ϵ}
first(D) = {d}
first(E) = {e, ϵ}

(11) $S \to ACB \mid CbB \mid Ba$ | $A \to da \mid BC$ | $B \to h \mid \epsilon$ | $C \to g \mid \epsilon$

first(S) = {g, d, h, b, ϵ}
first(A) = {d, ϵ, h, g}
first(B) = {h, ϵ}
first(C) = {g, ϵ}

(12) $S \to AaAb \mid BbBa$ | $A \to \epsilon$ | $B \to \epsilon$.

first(S) = {a, b}
first(A) = {ϵ}
first(B) = {ϵ}

FOLLOW : Follow(A) is the set of terminals that may follow immediately to right of A in any product of the grammar.

Rules : (i) If A is start symbol then follow(A) contains $

(ii) If $S \to \alpha A \beta$ and F(β) does not have ϵ, then [Follow(A) = First(β)]

(iii) If $S \to \alpha A \beta$ and First(β) contains ϵ then Follow(A) = [First(β) − {ϵ}] ∪ Follow(S)

Ex. $S \to AB$ — Follow(S) = {$}
$A \to a \mid \epsilon$ — Follow(A) = {$, b}
$B \to b \mid \epsilon$ — Follow(B) = {$}

$S \to AB$ — Follow(S) = {$}
$A \to aA \mid \epsilon$ — Follow(A) = {$, b}
$B \to b \mid \epsilon$

ϵ is Never present in follow

(iv) If $S \to \alpha A$ then
Follow(A) = Follow(S).

Find the follow :

(1) $S \to a \mid b \mid \epsilon$ — follow(S) = {$}

(2) $S \to aA \mid \epsilon$ — follow(S) = {$}
$A \to Ab \mid \epsilon$ — follow(A) = {b, $}

(3) $S \to aAb \mid aB$ — follow(S) = {$}
$A \to Aa \mid aB \mid b$ — follow(A) = {b, a}
$B \to Bc \mid d$ — follow(B) = {b, a, c, $}

(4) $S \to AaAb \mid BbBa$ — follow(S) = {$}
$A \to \epsilon$ — follow(A) = {a, b}
$B \to \epsilon$ — follow(B) = {a, b}

(5) $S \to aAB$ — follow(S) = { $ }
$A \to aA \mid \epsilon$ — follow(A) = { b }
$B \to b$ — follow(B) = { $ }

(6) Silly Mistake
$S \to aAbB$ — follow(S) = { $, a } ∪ { b, c }
$A \to Aa \mid bBa \mid \epsilon$ — follow(A) = { a, b }
$B \to SB \mid c \mid d$ — follow(B) = { $, a } ∪ { b, c }

(7) $S \to aSb \mid aAb$ — Follow(S) = { b, a, $ }
$A \to bA \mid aBS \mid a$ — follow(A) = { b, b, a }
$B \to ASB \mid a \mid b$ — follow(B) = { a }

(8) $S \to AA$ — Follow(S) = { $ }
$A \to aA \mid b \mid \epsilon$ — follow(A) = { $, a, b }

(9) $S \to AB$ — follow(S) = { $ }
$A \to a \mid \epsilon$ — follow(A) = { b, $ }
$B \to \epsilon \mid b$ — follow(B) = { $ }

(10) $S \to ABCDE$ — follow(S) = { $ }
$A \to a \mid \epsilon$ — follow(A) = { b, c, d }
$B \to b \mid \epsilon$ — follow(B) = { c, d }
$C \to c \mid \epsilon$ — follow(C) = { d }
$D \to d$ — follow(D) = { $, e }
$E \to e \mid \epsilon$ — follow(E) = { $ }

(11) $S \rightarrow (L) / a$ — Follow(S) = { , $, (,) }

$L \rightarrow SL'$ — Follow(L) = {) }

$L' \rightarrow , SL' (\ / \epsilon$ — Follow(L') = {) (}

(12) $S \rightarrow ACB / CbB / Ba$ — Follow(S) = { $ }

$A \rightarrow da / BC$ — Follow(A) = { g, h, $ }

$B \rightarrow hB / \epsilon$ — follow(B) = { $, a, g, h }

$C \rightarrow g / \epsilon$ — follow(C) = { b, $, a, g, h }

(13) $S \rightarrow ABCDE$ — Follow(S) = { $ }

$A \rightarrow a / \epsilon$ — follow(A) = { $, b, c, d, e }

$B \rightarrow b / \epsilon$ — follow(B) = { $, c, d, e }

$C \rightarrow C / \epsilon$ — follow(C) = { $, d, e }

$D \rightarrow d / \epsilon$ — follow(D) = { $, e }

$E \rightarrow e / \epsilon$ — follow(E) = { $ }

Procedure to construct LL(1) parse Table.

$A \to \alpha$

(i) Add $A \to \alpha$ in $M[A, a]$ for every terminal a in $First[\alpha]$

(ii) If first of α contain ϵ, then add, $A \to \alpha$ in $M[A, b]$ for every terminal in follow of A.

Ex $A \to (A) \mid a$

	(	)	a	$
A	A→(A)		A→a	

$w = (a)$

$A \to (A) \to (a)$

Stack	I/P String	Operation
$A	(a)$	M[A, (] = A→(A).
$)A(	(a)$	POP.
$)A	a)$	M[A, a] = A→a
$)a	a)$	POP
$)	)$	POP
$	$	Success

Ex $S \to aA \mid BaAb$

$A \to aA \mid \epsilon$

$B \to a \mid b$

Conflict

	a	b	$
S	S→aA, S→BaAb	S→aAb	
A	A→aA	A→ϵ	A→ϵ
B	B→a	B→b	

$S \to aA \mid BaAb$ First(aA) ∪ First(BaAb).

- check for every transition, what the left hand side is able to derive from that transition.
- for ex. $A \to \epsilon$, find Follow(A) and add entries for every non terminal in follow(A).
- If First(NTerminal$_1$) $\cap$ ~~First Follow~~ First(NT_2) then the grammar is Not LL(1).
- LL(1) grammar is free from multiple entries.

LL(1) Grammar :

- G is LL(1) if its LL(1) parse table is free from multiple entries.
- for grammar G, LL(1) parser can be constructed if and only if
- It is **free from ambiguity**

Ex $S \to A \mid a$
$A \to a$

	a
Table (S) :	$S \to a$, $A \to a$ } conflict
Table (A) :	$A \to a$

- It should be free from left recursion.

Ex $A \to Aa \mid b$ cannot generate LL(1) Parse tree.

Ex It must not have common prefix. It should be "left factored."

- Every Grammar which is ambiguous is not LL(1).
- Every Grammar which has left recursion is NOT LL(1).
- Grammar which is not LL(1). No need to check for these.

Rules to check LL(1) or Not

- G without ϵ-production is LL(1) for
  $A \to \alpha_1 \mid \alpha_2 \mid \alpha_3 \ldots \alpha_n$.
  The set $First(\alpha_1)$, $First(\alpha_2)$, $First(\alpha_3) \ldots First(\alpha_n)$ are mutually disjoint.
  i.e intersection of any two firsts are not LL(1).
- If G has ϵ-production is LL(1) if for every production of the form.

$A \to \alpha \mid \epsilon$
then $First(\alpha) \cap Follow(A) = \phi$.

Ex. $S \to aSa \mid \epsilon$

$S \to aSa$: $First(aSa) = \{a\}$

$S \to \epsilon$: $Follow(S) = \{a\}$.

Conflict

③ $S \to AaAb \mid BaBb$
$A \to \epsilon$
$B \to \epsilon$

Not LL(1) as.
$S \to AaAb \mid BaBb$ present in column a.

	a	b	$
S	S→AaAb S→BaBb		
A	A→ϵ	A→ϵ	
B	B→ϵ	B→ϵ	

Ex ① $S \to aS \mid baS \mid c \mid d$ is LL(1)

② $S \to AaAb \mid BbBa$ = LL(1)
$A \to \epsilon$
$B \to \epsilon$

③ $S \to aSb \mid ab$. is NOT LL(1)

④ $S \to aSb \mid \epsilon$ is LL(1)

⑤ $S \to A \mid a$ Not LL(1)
$A \to a$

⑥ $A \to A(A) \mid a$ Not LL(1) left recursion

⑦ $S \to aAB \mid Bab$ ✓
$A \to aA \mid \epsilon$ ✓ $A \to aA = \{a\}$ is LL(1)
$B \to dBb \mid c$ ✓ $A \to \epsilon = \{d, c\}$

(8) $S \to aAb \mid Bc$ Not LL(1)
$A \to bA \mid \epsilon$
$B \to Bc \mid d$

(9) $S \to 0S1 \mid 1S0 \mid \epsilon$ Not LL(1)
$\{0\}$ $\{1\}$ $\{1, 0\}$.

(10) $S_1 \to S\#$ ✓
$S \to qABC$ ✓
$A \to a \mid bbD$ ✓ is LL(1).
$B \to a \mid \epsilon$ $\{a\} \cap \{\$, \#, b\}$ ✓
$C \to b \mid \epsilon$ $\{b\} \cap \{\$, \#\}$ ✓
$D \to c \mid \epsilon$ $\{c\}$ $\{\$, \#, a, b\}$ ✓

(11) $S \to A$ Not LL(1)
$A \to aB \mid Ad$
$B \to bBC \mid f$
$C \to g$

(12) $S \to iEtSS_1 \mid a$ Not LL(1).
$S_1 \to eS \mid \epsilon$ $\{e\} \cap \{\$, e\}$
$E \to b$

(13) $S \to aBDh$ ✓ is LL(1).
$B \to CC$ ✓
$C \to bC \mid \epsilon$ $\{b\} \cap \{g, f, h\}$ ✓
$D \to EF$ ✓
$E \to g \mid \epsilon$ $\{g\} \cap \{f, g, h\}$ ✓
$F \to f \mid \epsilon$ ✓

(14) $S \to aAbB \mid bAaB \mid \epsilon$

$A \to S$

$B \to S$

Not LL(1).

for $S = \{a\} \cap \{b\} \cap \{b\} \times$

LL(1) table for 14 :

	a	b	$	
S	$S \to aAbB$, $S \to \epsilon$	$S \to bAaB$, $S \to \epsilon$	$S \to \epsilon$	conflict.
A	$A \to S$	$A \to S$	~~$A \to S$~~	→ careful.
B	$B \to S$	$B \to S$	~~$B \to S$~~	Follow(A) does not contain $

(Q.11.) $S \to A$

$A \to aB \mid Ad$

$B \to bBC \mid f$

$C \to g$

	a	b	c	d	f	$
S	$S \to A$					
A	$A \to Ad$, $A \to aB$					
B		$B \to bBC$			$B \to f$	

Entry ~~double~~ possible for $A \to Ad$? Yes

12 $S \to iEtSS_1 \mid a$ follow(S_1) = { e, $ }

$S_1 \to eS \mid \epsilon$

$E \to b$

	a	b	i	t	e	$
S	$S \to a$		$S \to iEtSS_1$			
S_1					$S_1 \to eS$ $S_1 \to \epsilon$ $S_1 \to \epsilon$	
E		$E \to b$				

13 $S \to aBDh$

$B \to cC$

$C \to bC$

$D \to EF$

$E \to g \mid \epsilon$

$F \to f \mid \epsilon$

g

C → g.

Ques) $S \rightarrow FR$

$R \rightarrow *S / \epsilon$

$F \rightarrow id$

In the predictive table M of the grammar the entries M[s, id] & M[R, $] respectively are ?

	*	id	$
S	~~S→FR~~	S→FR ✓	
R	R→*S		R→ϵ ✓
F		F→id	

$S \rightarrow FR = \{id\}$

$R \rightarrow *S = \{*\}$

$R \rightarrow \epsilon = \{\$\}$

$F \rightarrow id$

BOTTOM-UP PARSER (shift Reduce)

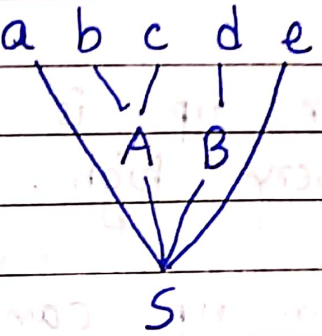
The process of constructing parse tree starting from input string and continuing upto start symbol of grammar is known as bottom-up parsing.

Ex. $S \rightarrow aABe$

$A \rightarrow bc$

$B \rightarrow d$

$w = abcde$

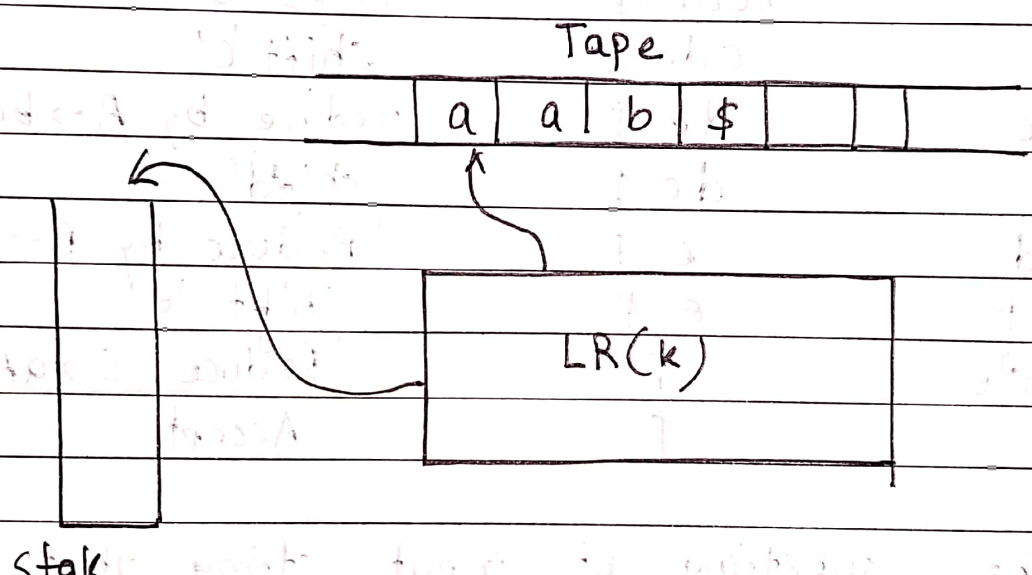


stack	i/p string	opr
$	abcde $	shift 'a'
$a	bcde $	shift 'b'
$ab	cde $	shift 'c'
$ abc	de $	reduce by A → bc
$ aA	de $	shift 'd'
$ aAd	e $	Reduce by B → d
$ aAB	e $	shift 'e'
$ aABe	$	Reduce S → aABe
$ S	$	Accept

Handle : The substring of input string that match with the right hand side of any production is called as an handle.

- Bottom up parsers internally use reverse of next derivation. It is also called as shift reduce parsers.

- It can be constructed for both ambiguous & unambiguous.
- Bottom up also allows the grammar which is left recursive and not left factored.
- Bottom up is very fast and hence performance is very high.
- Bottom up can be used for more complex Grammar
- The average time complexity is $O(n^3)$.
- Block Diagram of bottom up:



Item	Action		Goto (N.T)
	T	$	N.T
I			
T	shift/		~~Reduce~~
E	Reduce		shift
MS			only

LR(k) parse table

Tape : Divided into multiple cell and each cell contain multiple symbol. At any point of time the input buffer string.

Stack : The stack contain all the grammar symbol. The grammar symbol can be pushed into stack or pop from the stack using shift reduce operation.

Parse table : It is constructed using terminal non-terminal and LR items. The parse table is consist of two parts. Action and Goto.

Action : It contains shift or reduce operation performed on terminals.

Goto : Contain only shift operation which are performed on non-terminals.

- Action in shift reduce parser
  - (i) Shift : Whenever handle does not occur from the top symbol of stack, then perform the shift operation.
  - (ii) Reduce : Whenever handle occur from the top symbol of stak, then perform the reduce operation.

- After reading input string, if stack contain only start symbol of grammar as top-most symbol, then input is accepted.

- Error: After processing the complete input string, if the stack does not contain the stack symbol as topmost symbol, then the parsing is failed and result is error.

Ex.1 $A \rightarrow aA / b$

$w = aab$

stack	i/p	operation
$ a	aab $	Shift a
$ a	ab $	shift a
$ aa	b $	shift b
$ aab	$	Reduced A→b
$ aaA	$	reduced A→aA
$ aA	$	reduced A→aA
$ A	$	Accept.

Ex.2 $S \rightarrow AA$

$A \rightarrow aA / b$

$w = abab$.

stack	input	operation
$	abab$	shift a
$a	bab$	shift b
$ab	ab$	Reduce b A→b
$aA	ab$	Reduce A→aA
$A	ab$	shift A
$Aa	b$	shift b
$Aab	$	Reduce b A→b
$AaA	$	Reduce A→aA
$AA	$	Reduce S→AA
$S	$	Accept.

LR(k) Parser :

- It is a kind of bottom up parser, first L stands for Left to right scan, R stand for right most derivation in reverse order. K is # look ahead.
- LR(K) is constructed only for unambiguous grammar.

Classification of LR(k) parsers :

(a) k=0 → LR(0)
→ SLR (simple LR)

(b) k=1 → CLR LR(1) canonnical
→ LALR LALR(1) Look Ahead.

Procedure to construct :

- obtain the augmented grammar for the given grammar.
- Create the cannonical collection LR(0) item of compiler.
- Draw the DFA
- Prepare parse table using DFA.

Augmented Grammar: Obtained by adding one more production as unit production. S' drives to S.

$S' \rightarrow S$

Ex. $A \rightarrow aA \mid b$. $\xrightarrow{\text{augment}}$ $A' \rightarrow A$

$A \rightarrow aA \mid b$.

LR(0) item : Items of compiler.

Any production which has . any where on the right hand side of grammar. It is known as LR(0) item.

Ex $A \rightarrow XYZ$

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$ } Incomplete / Non final / shift.

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$ } final / Reduce / Complete.

- final item used for reducing.
- Non-final item used for shift.
- final item represents a final state in DFA.

Canonical collection

- The set $C = \{ I_0, I_1, I_2 \ldots \}$ is known as canonical collection of items.

$$I_i = \text{state } i$$

Function used for creation of LR(0) item.

1. closure of (I)
2. Goto (I, X).

where I is LR(0) item and $X \in (V+T)$

Closure of I:

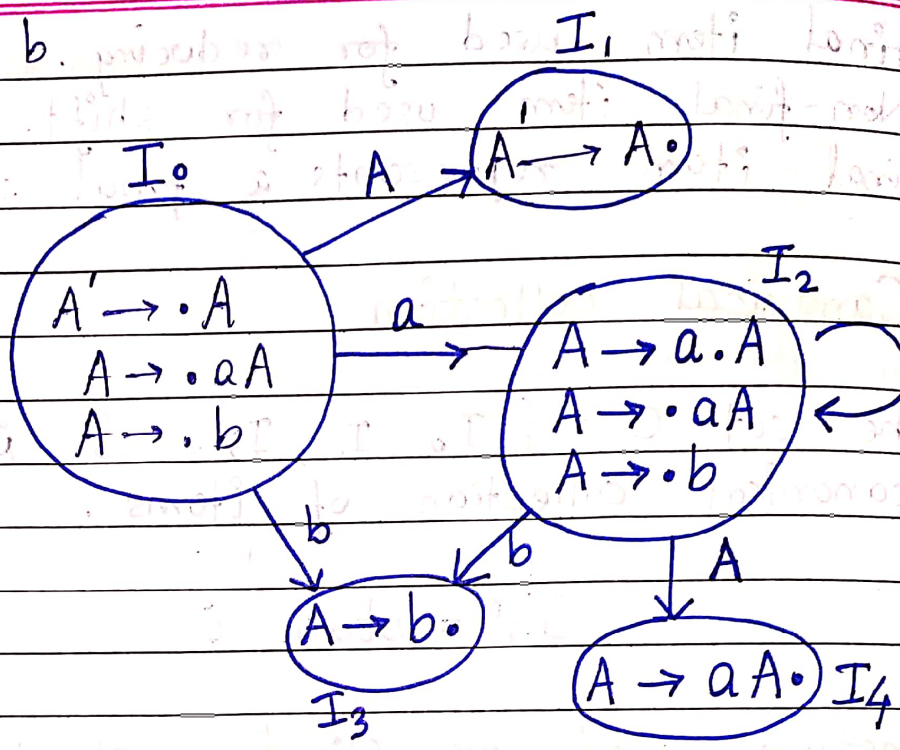
- The input is set of items and output is set of items

Procedure to find

i) Add everything from input to output.
ii) If $A \to \alpha . B\beta$ in I and $B \to \gamma$ is present in grammar, then add $B \to .\gamma$ in I.
iii) Repeat (ii) for every newly added item.

Ex $A \to aA \mid b$.

$A' \to A$
$A \to aA$
$A \to b$



LR(0) table :

Item	a (Action)	b	$	A (Goto)
0	S_2	S_3		1
1			Acc	
2	S_2	S_3		4
3	r_2	r_2	r_2	
4	r_1	r_1	r_1	

- Shift Entry :
  - (i) If $Goto(I_i, X) = I_j$ then
    - (a) If $X \in T$ then $Action(i, X) = S_j$ (shift)
    - (b) If $X \in NT$ then $Goto(i, X) = j$

Reduce Entry :

(ii) If $Goto(I_i, X) = \phi$ then

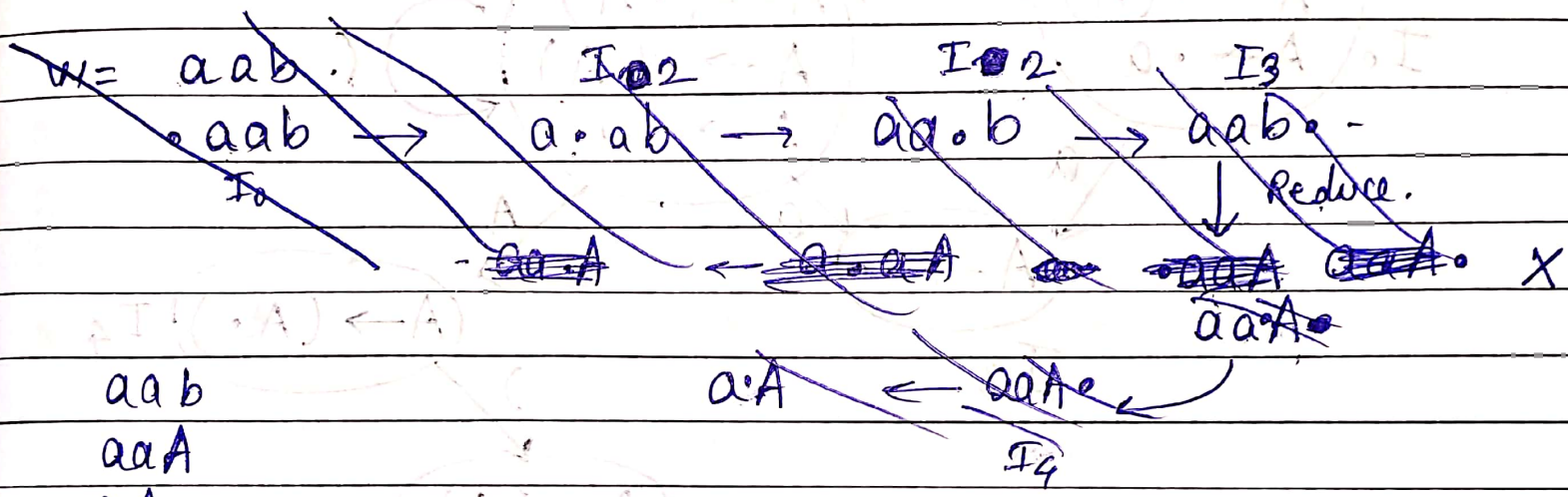
Reduce only for final item

Accept : $A' \to A\bullet$ — I_1 — { $ }

r_1 : $A \to aA\bullet$ — I_4 — { a, b, $ }

r_2 : $A \to b\bullet$ — I_3 — { a, b, $ }

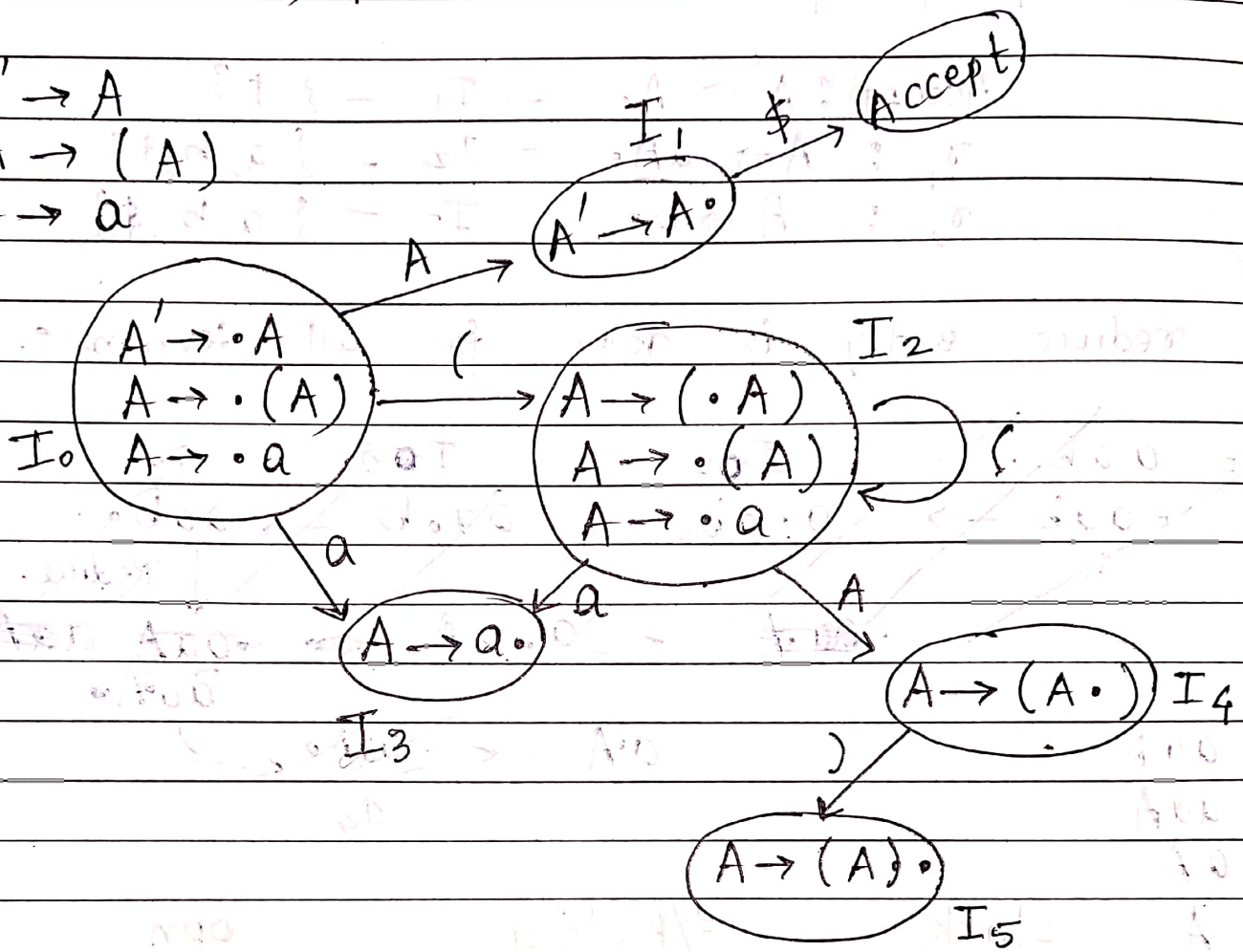
for reduce entry is done for all terminals.



A	stack	I/P string	opr.
0	$	aab$	S_2
2	$a	ab$	S_2
2	$aa	b$	S_3
3	$aab	$	r_2 $A \to b$
4	$aaA	$	r_1 $A \to aaA$
1	$aA	$	r_1 $A \to aA$
1	$A	$	Accept ✓

Ex $A \to (A) \mid a$

$A' \to A$
$A \to (A)$
$A \to a$



LR(0) table

	Action				Goto.
Item	a	(	)	$	A
0	s_3	s_2			1
1				Accept	
2	s_3	s_2			4
3	r_2	r_2	r_2	r_2	
4			s_5		
5	r_1	r_1	r_1	r_1	

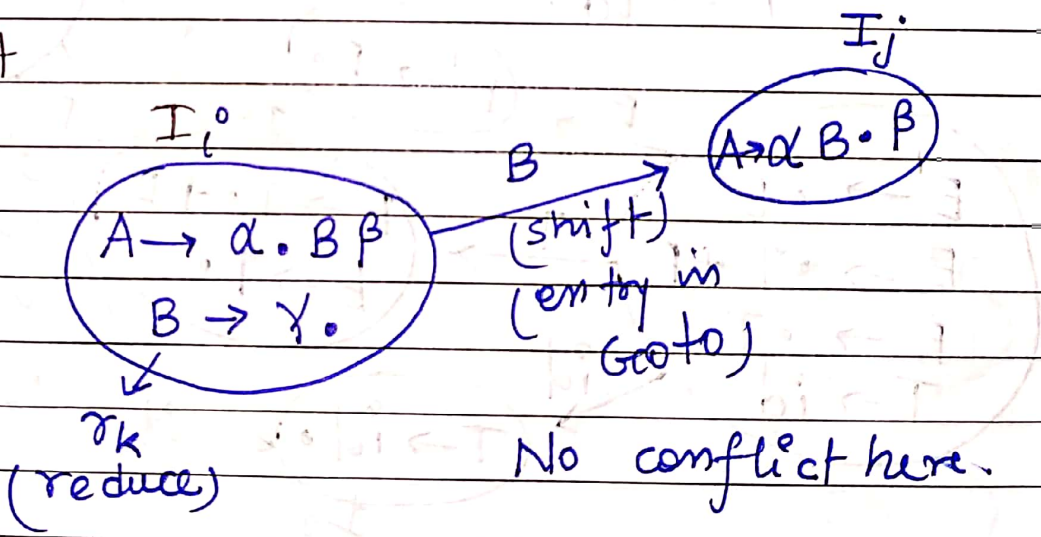
Reduce entries.	$A' \to A\cdot$	(Accept)	I_1
	$A \to (A)\cdot$	(r_1)	I_5
	$A \to a\cdot$	(r_2)	I_3
		↓	↓
		Entry	Row

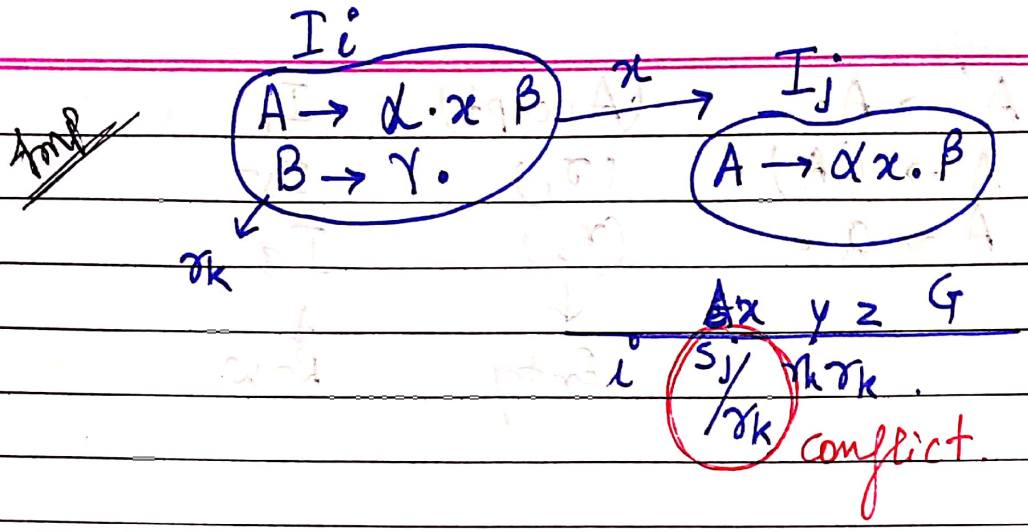
$w = (a)\$$.

	Stack	I/p	operation.
0	$	(a) $	S_2
2	$ (	a) $	S_3
3	$ (a	) $	r_2
4	$ (A	) $	S_5
5	$ (A)	$	r_1
1	$ A	$	Accept

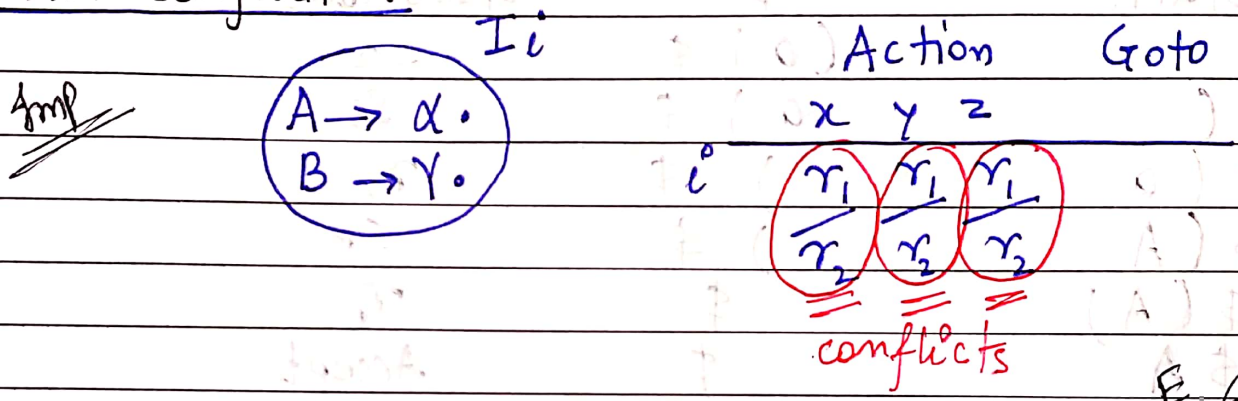
Conflicts : in LR(0).

(i) SR conflict

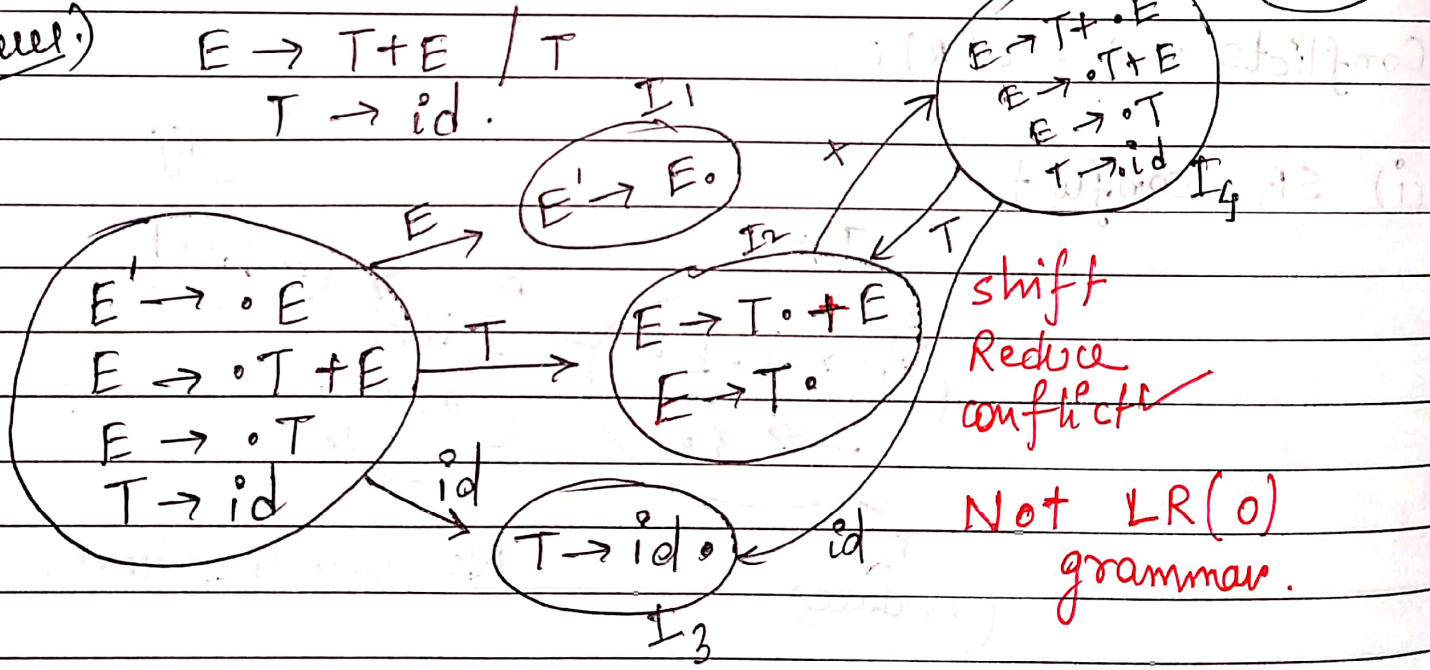




(2) RR Conflict :



Ques: E → T+E / T
T → id.



	Action			Goto	
Item	id	+	$	E	T
0	S_3			1	2
1					
2		S_4			
3					
4	S_3			5	2
5					

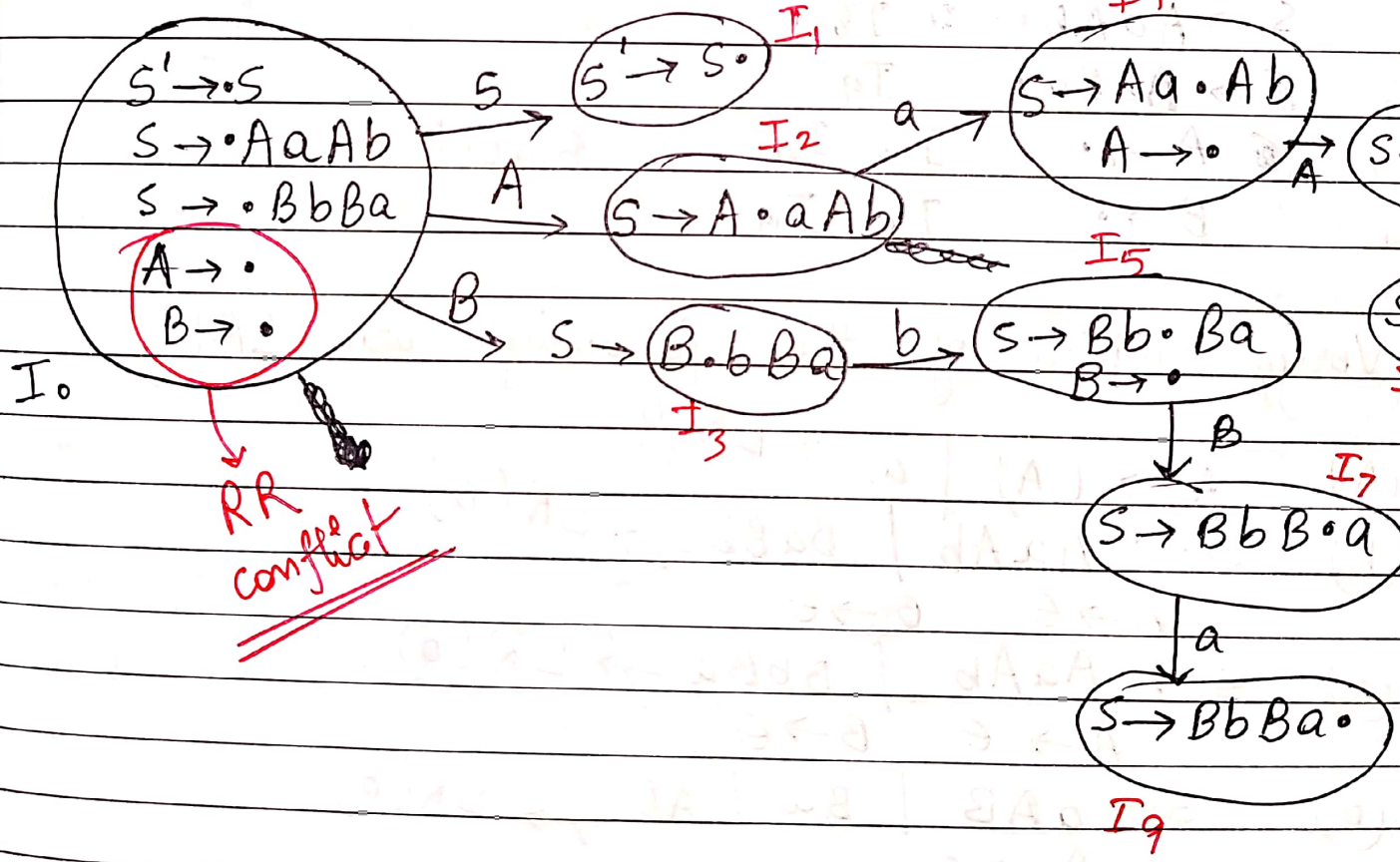
Reduce : $E' \to E.$ I_1 Accept

Entry $E \to T+E.$ I_5 r_1

$E \to T.$ I_2 r_2

$T \to id.$ I_3 r_3

Ques $S \to AaAb \mid BbBa. \quad A \to \epsilon \quad B \to \epsilon$



	Action			Goto		
Item	a	b	$	S	A	B
0	r_3/r_4	r_3/r_4	r_3/r_4	1	2	3
1						
2	S_4					
3		S_5				
4	r_3	r_3	r_3		6	
5	r_4	r_4	r_4			7
6		S_8				
7	S_9					
8	r_1	r_1	r_1			
9	r_2	r_2	r_2			

Reduce Entries :

Accept $S' \to \bullet S \bullet$ I_1

r_1 $S \to AaAb \bullet$ I_8

r_2 $S \to BbBa \bullet$ I_9

r_3 $A \to \bullet$ I_0, I_4 } important

r_4 $B \to \bullet$ I_0, I_5 }

Q) Verify which of the following are LR(0)

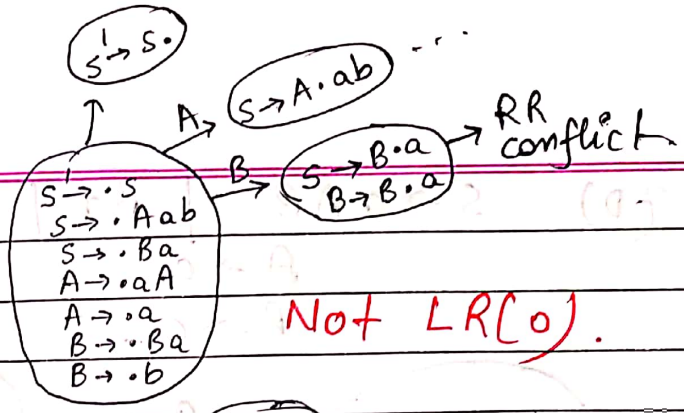
(A) $A \to (A) \mid a$ → LR(0)

(B) $S \to AaAb \mid BaBb$ → LR(0)
$A \to \epsilon$ $B \to \epsilon$

(C) $S \to AaAb \mid BbBa$ → LR(0).
$A \to \epsilon$ $B \to \epsilon$

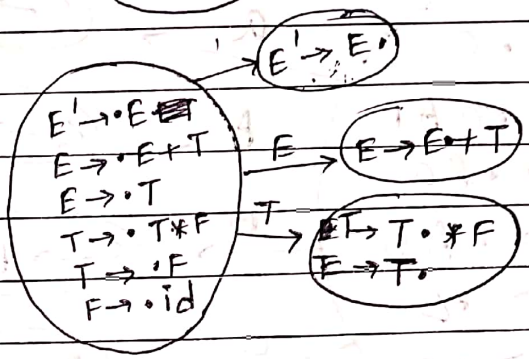
(D) $S \to aAB \mid Ba \mid Ab$ → LR(0)
$A \to C$
$B \to C$.

(E) $S \to Aab \mid Ba$
$A \to aA \mid a$
$B \to Ba \mid b$



Not LR(0).

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



Not LR(0)

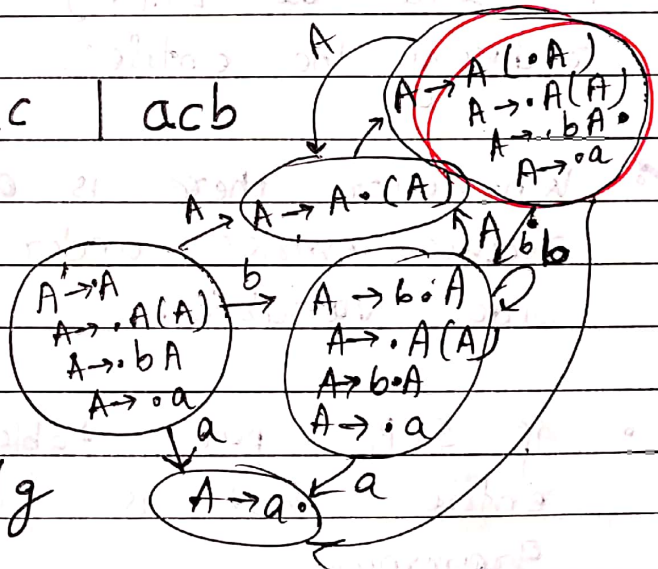
(G) $S \to AB \mid BA$
$A \to Aab \mid b$
$B \to BaA \mid a$

Not LR(0)
SR-conflict?

(H) $S \to A\#a \mid @bB$
$A \to \#A \mid \#$
$B \to *B\# \mid \#$

Not LR(0)

(I) $S \to AaA \mid bab \mid bac \mid acb$
$A \to aBA \mid b$
$B \to b$



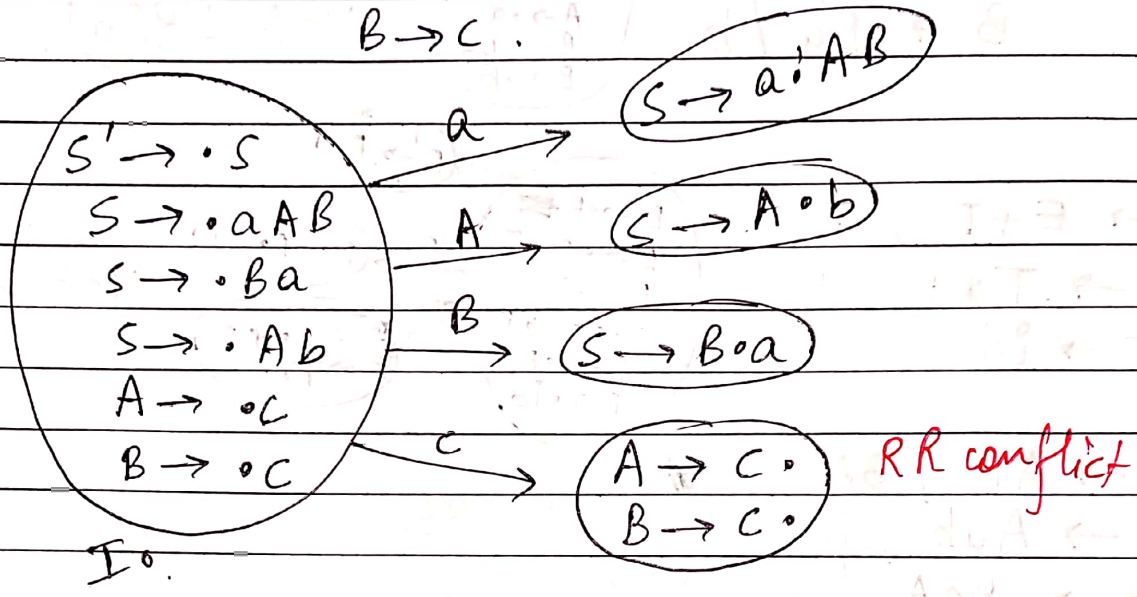
(J) $A \to A(A) \mid bA \mid a$
(Not LR(0))

(K) $E \to EF \mid e$, $T \to ET \mid g$
$F \to FT \mid f$.
Not LR(0)

(D) $S \to aAB \mid Ba \mid Ab$

$A \to c$

$B \to c$.



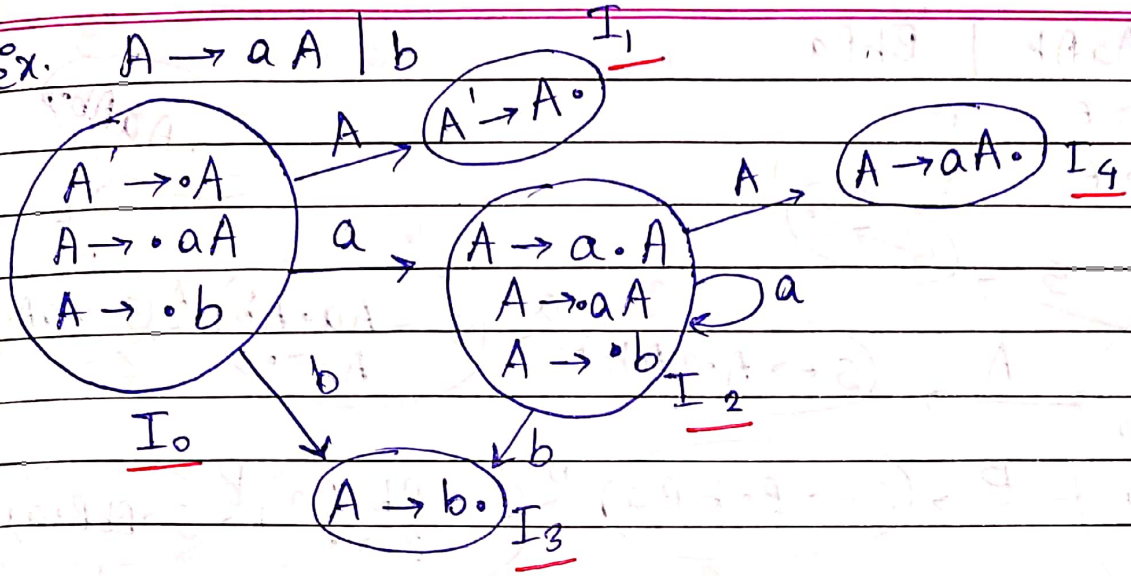
SLR :

- The process of construction of parse table for SLR1 is same as LR(0) but there is 1 restriction on reducing the entries.

- Whenever there is a final item then place the reduce entries under the follow symbol of left hand side variable.

- If SLR1 parse table does not contain multiple entries that is free from conflict is SLR1 grammar.

Ex. $A \to aA \mid b$



Item	Action			Goto
	a	b	$	A
0	S_2	S_3		2
1			Accept	
2	S_2	S_3		4
3			r_2	
4			r_1	

Reduce entries — Column : under follow of LHS

Accept	$A' \to A\cdot$	I_1	{$}
r_1	$A \to aA\cdot$	I_4	{$}
r_2	$A \to b\cdot$	I_3	{$}

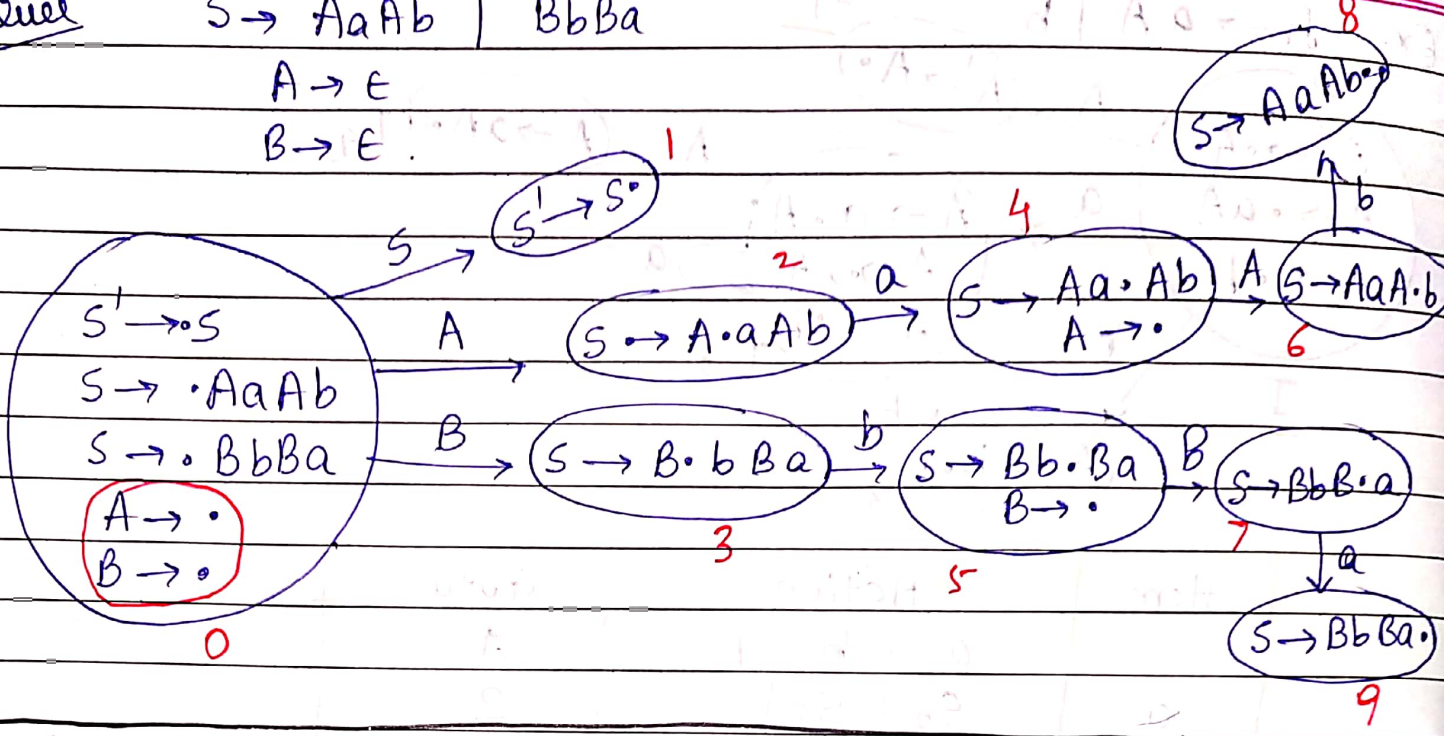
shift entry in SLR = # shift Entry in LR(0)

Reduce entries in SLR ≤ # shift entries in LR(0)

Ques $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$



Item	Action			Goto		
	a	b	$	S	A	B
0	$r_3 \mid r_4$	$r_3 \mid r_4$			2	3
1			Accept			
2	S_4					
3		S_5				
4	r_3	r_3			6	
5	r_4	r_4				7
6		S_8				
7	S_9					
8			r_1			
9			r_2			

Reduce entries

Accept : $S' \to S\cdot$ I_1 $\{\$\}$

r_1 : $S \to AaAb\cdot$ I_8 $\{\$\}$

Follow(S) = {$}

Follow(A) = {a, b}

Follow(B) = {a, b}

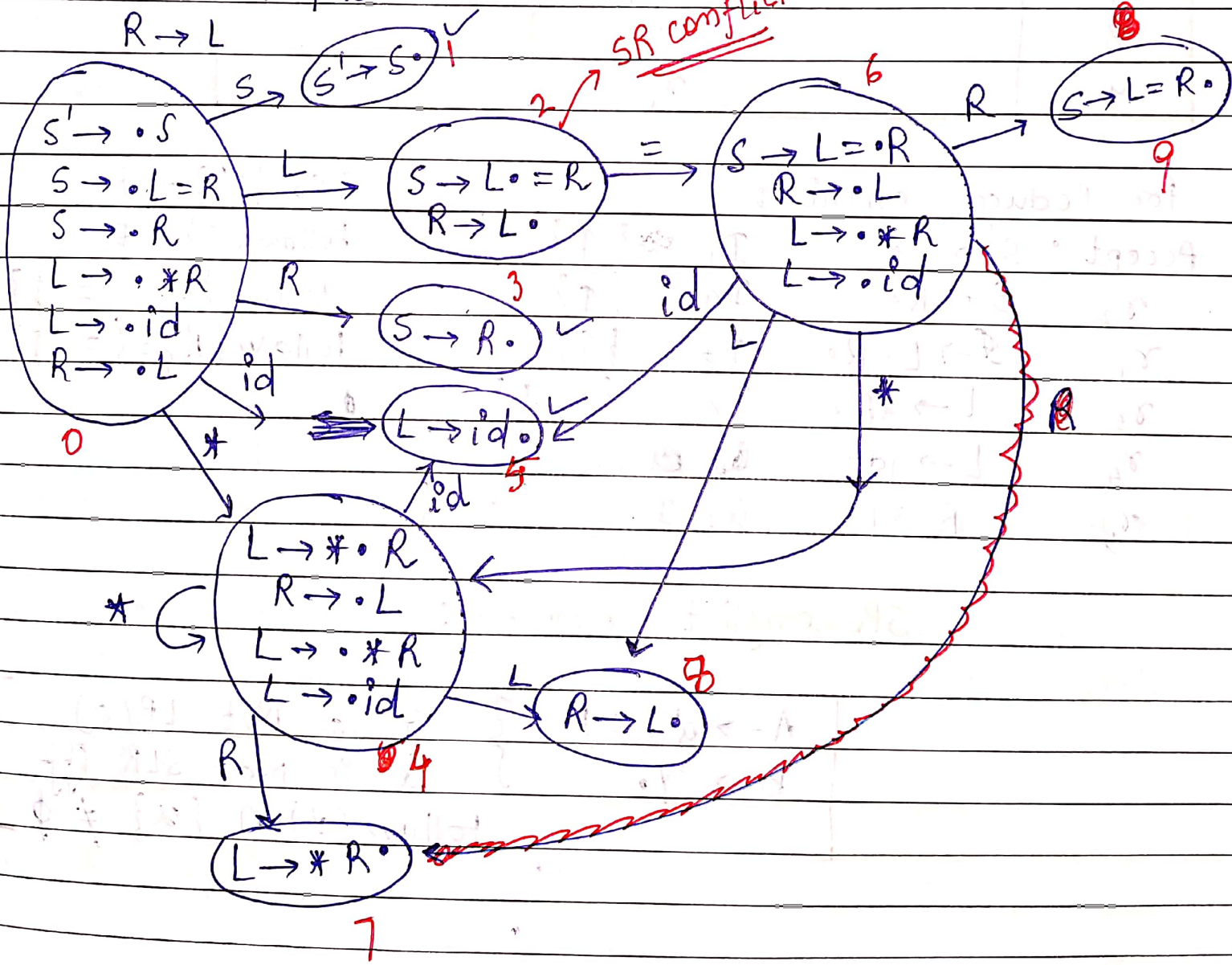
r_2	$S \to BbBa\,\cdot$	I_9	$\{\$\}$
r_3	$A \to \cdot$	$I_0\ I_4$	$\{a, b\}$
r_4	$B \to \cdot$	$I_0\ I_5$	$\{a, b\}$

Bottomline $A \to \alpha\cdot$, $B \to \beta\cdot$ } G is not LR(0) / G is not SLR (iff) Follow(A) ∩ Follow(B) ≠ ϕ.

Ques $S \to L=R \mid R$

$L \to *R \mid id$

$R \to L$



Stem	Action				Goto		
	id	*	=	$	S	L	R
0	S_5	S_4			1	2	3
1				Accept			
2			SR Conflict ← S_6/r_5	r_5			
3				r_2			
4	S_5	S_4				8	7
5			r_4	r_4			
6	S_5	S_4				8	9
7			r_3	r_4			
8			r_5	r_5			
9				r_1			

for Reduce entries :

Accept :	$S' \to S.$	I_1	$\{\$\}$
r_2	$S \to R.$	I_9	$\{\$\}$
r_1	$S \to L=R.$	I_3	$\{\$\}$
r_3	$L \to *R.$	I_7	$\{=, \$\}$
r_4	$L \to id.$	I_5	$\{=, \$\}$
r_5	$R \to L.$	I_2 I_8	$\{=, \$\}$

Follow$\{S'\}$ = $
Follow$\{S\}$ = $
Follow$\{L\}$ = $\{=, \$\}$
Follow$\{R\}$ = $\{=, \$\}$

SR Conflict condition :

$A \to \alpha . x \beta$ }
$B \to \gamma .$ }

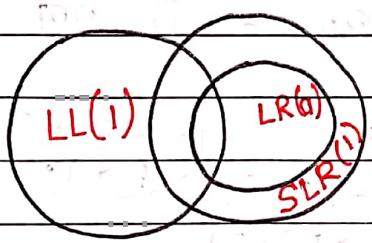
G is Not LR(0)
G is Not SLR iff
follow(B) $\cap \{x\} \neq \phi$

Is every LL(1) grammar LR(0) and vice versa?
No, not necessarily.

Ex $S \to 01 \mid 00$. is LR(0) but Not LL(1)
$S \to Aa \mid b$ is Not LR(0) but not LL(1)

$S \to AaAb \mid BaBb$
$A \to \epsilon$
$B \to \epsilon$
} is LL(1) but Not LR(0)

- Every LL(1) need not be LR(0)
- Every LR(0) need not be LL(1).



- Every LL(1) need not be SLR(1)
- Every SLR(1) need not be LL(1)
- Every LR(0) is SLR(1) but converse need not be true.
- # Entries in SLR(1) parse table is less than or equal to number of entries in LR(0) parse table.

Conflics

① SR conflict :

SR conflict :

$A \to \alpha \cdot x \beta$
$B \to \gamma \cdot$

Not LR(0)

$A \to \alpha \cdot x \beta$
$\beta \to \gamma \cdot$

$\{x\} \cap follow(B) \neq \phi$
then G is Not SLR(1)

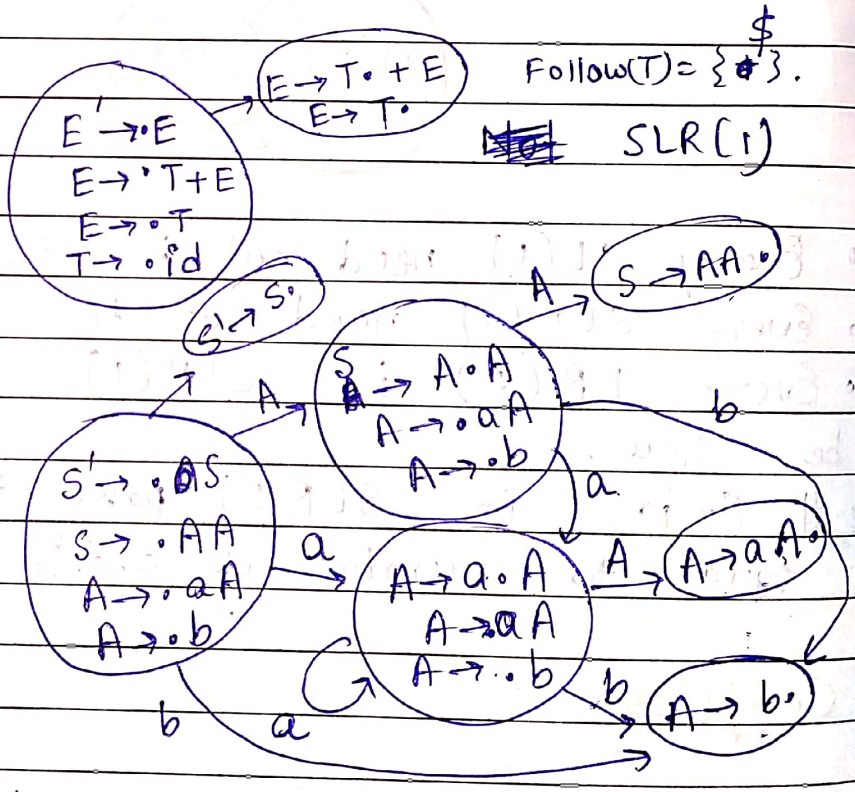
RR conflict :

$A \to \alpha \cdot$
$\beta \to \gamma \cdot$

Not LR(0).

$Follow(A) \cap follow(B) \neq \phi$
then G is not SLR(1)

Ex $E \to T + E \mid T$
$T \to id$



Ex $S \to AA$
$A \to aA \mid b$

Also LL(1)

Not any conflict
Both LR(1) and SLR(1)

LR(1) PARSER : CLR

- It depends on one Lookahead symbol.
- Procedure to construct LR(1) parser.

(1) Closure (I)

(a) Add everything from input to output.

(b) if $A \to \alpha . B\beta$, x and $B \to \gamma$ is in production then add every production of B in form

$B \to .\gamma$, First (βx) in the G.

(c) Repeat step(b) for every newly added product

$A \to \alpha . B\beta$, x ← lookahead

$B \to .\gamma$, First (βx).

Ex $S \to AaAb \mid BbBa$

$A \to \epsilon$

$B \to \epsilon$

∴

$S' \to .S, \$$

$S \to .AaAb, \$$

$S \to .BbBa, \$$

$A \to ., a$

$B \to ., b$

Ex $E \to E+T \mid T$

$T \to T*F \mid F$

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

$E' \to .E, \$$

$E \to .E+T, \{\$\} \cup \{+\}$

$E \to .T, \{\$\} \cup \{+\}$

$T \to .T*F, \{\$, +, *\}$

$T \to .F, \{\$, +, *\}$

$F \to .(E), \{\$, +, *\}$

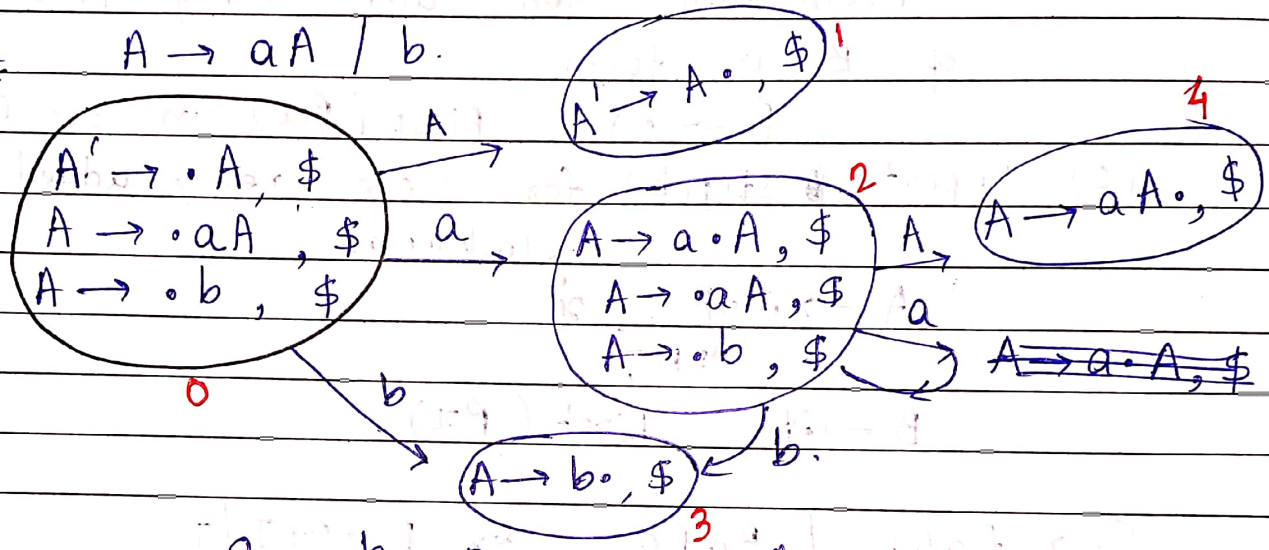
$F \to .id, \{\$, +, *\}$

(2) Goto (I, X)

(a) While finding the transition, the lookahead part remains same.

(b) while finding the closure, there will be change in look ahead part.

Ex $A \rightarrow aA \mid b$.



Item	Action			Goto
	a	b	$	A
0	S_2	S_3		1
1				
2	S_2	S_3		4
3			r_2	
4			r_1	

Reduce Entries :

Accept $A' \rightarrow A \bullet$ $(I, \$)$

r_1 $A \rightarrow aA \bullet$ $(I_4, \$)$

r_2 $A \rightarrow b \bullet$ $(I_3, \$)$

Row, column.

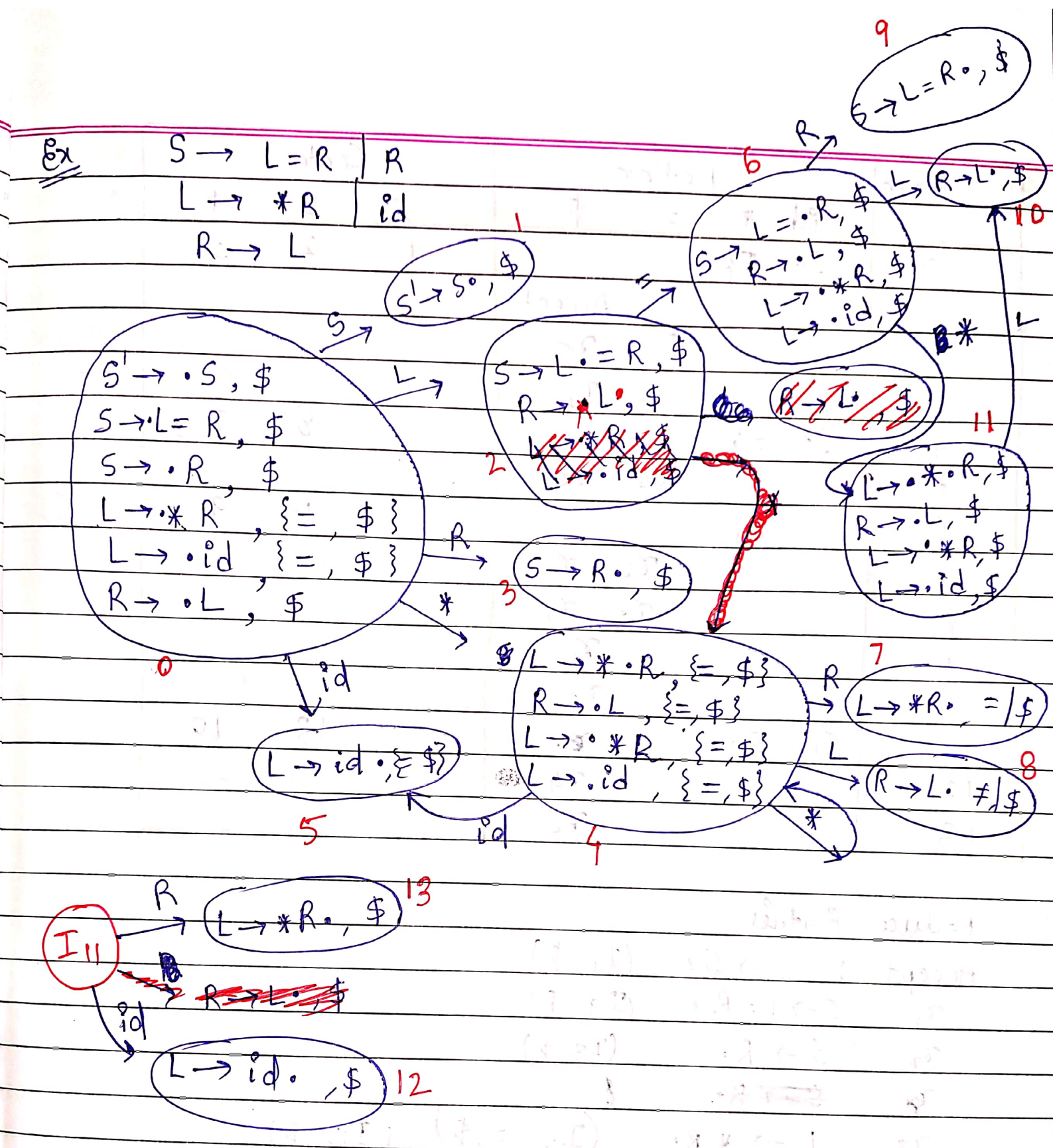
Grammar is LR(1) (CLR)

Ex

$S \rightarrow L = R \mid R$

$L \rightarrow *R \mid id$

$R \rightarrow L$



P.T.O

Stem	Action				Goto		
	id	*	=	$	S	L	R
0	S_5	S_4			1	2	3
1				Accept			
2			S_3	r_5			
3				r_2			
4	S_5	S_4				8	7
5			r_4	r_4			
6	S_{12}	S_{11}				10	9
7			r_3	r_3			
8			r_5	r_5			
9				r_1			
10				r_5			
11	S_{12}	S_{11}				13	10
12				r_4			
13				r_3			

Reduce Entries

Accept : $S' \to S.$ $(I_1, \$)$

r_1 $S \to L = R.$ $(I_9, \$)$

r_2 $S \to R.$ $(I_3, \$)$

~~r_3 $S \to R.$~~

r_3 $L \to *R.$ $(I_7, =, \$)$ $(I_{13}, \$)$

r_4 $L \to id.$ $(I_5, =/\$)$ $(I_{12}, \$)$

r_5 $R \to L.$ $(I_2, \$)$ $(I_8, =/\$)$ $(I_{10}, \$)$

SR

$A \to \alpha . x \beta$

$B \to \gamma . , x$

} Not CLR

RR

$A \to \alpha . , x$

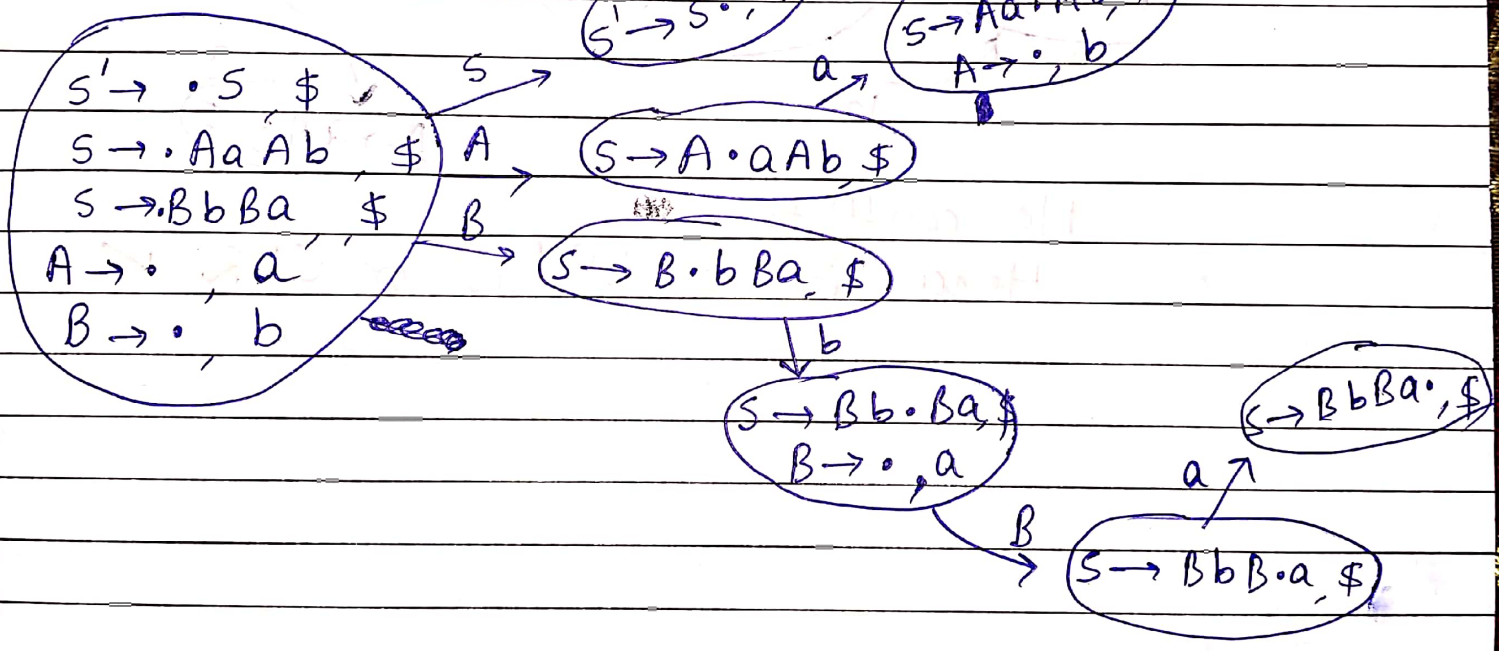
$B \to \beta . , x$

} Not CLR

Ques.) $S \to AaAb \mid BbBa$

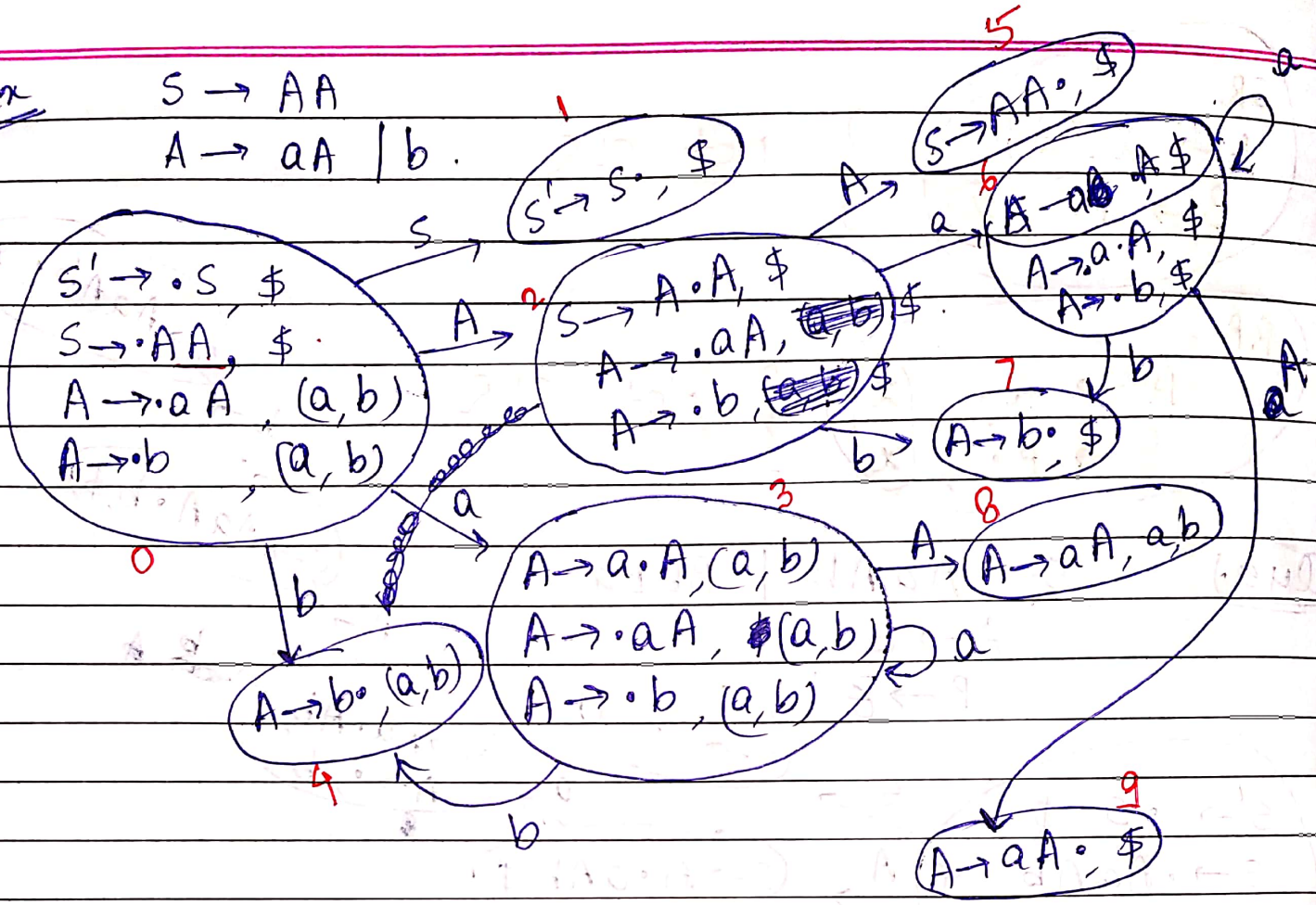
$A \to \epsilon$

$B \to \epsilon$



Ex $S \to AA$

$A \to aA \mid b$.



No conflicts

Hence it is LR(1) or CLR ✓

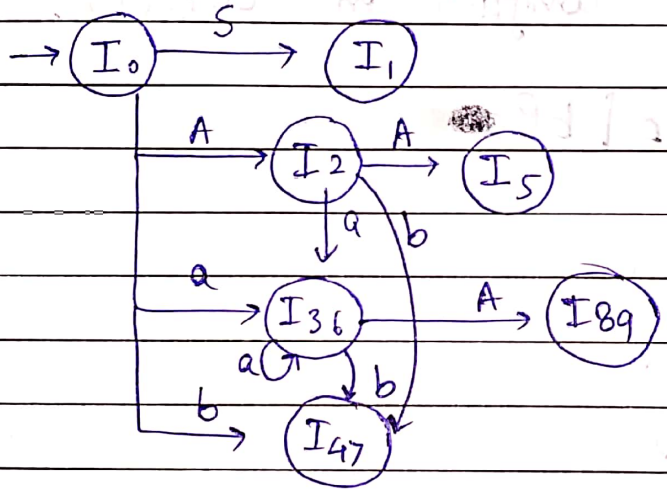
[LALR-1]

- The DFA of CLR(1) contain some state with common production part differ with lookahead part, combine these kind of state with single state and draw the DFA again and construct the parse table.

- If parse table is free from multiple entries, then the grammar is LALR(1).

Ex In previous example, $S \to AA$

$A \to aA \mid b$



Item	Action			Goto	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

states (SLR) = # (LALR) $\leq$ # (CLR).

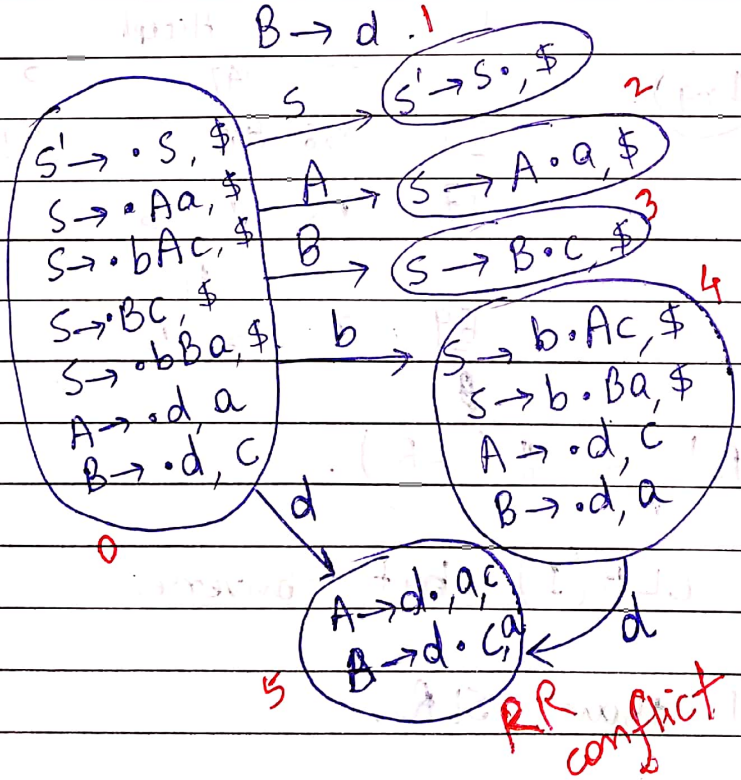
- Every LALR Grammar is CLR(1) but converse need not be true.
- LALR is more powerful than SLR.

- Every SLR1 grammar is LALR1 but converse need not be true.
- The number of entries in LALR1 table is less than or equal to number of entries in CLR table.

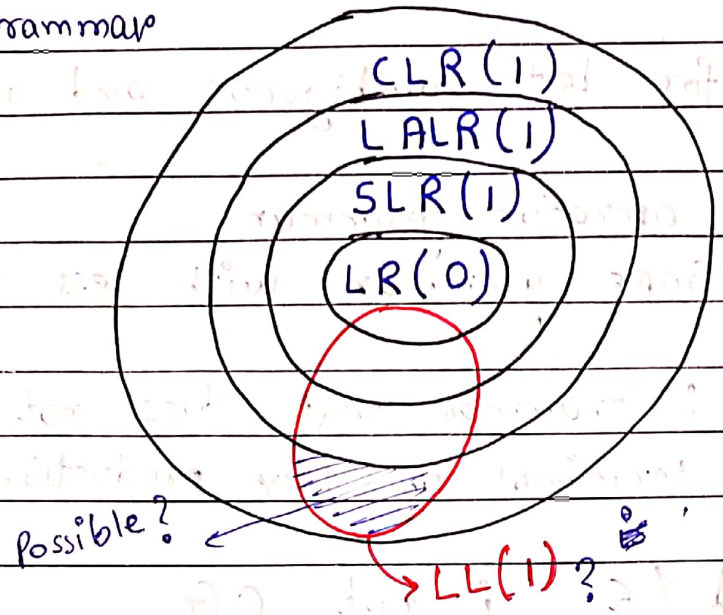
Conflict in LALR(1)

- If there is no SR conflict in CLR(1) then no SR conflict in LALR(1).
- There may be RR conflict in LALR(1) even when there is RR conflict in CLR(1).

Ex: S → Aa | bAc | Bc | bBa
A → d
B → d



Ques.) Grammar



S.No	Grammar	LL(1)	LR(0)	SLR(1)	LALR(1)	CLR(1)
1.	S → AA A → aA / b					
2.	E → E+T / T T → i					
3.	E → E+T / T T → TF / F F → F* / a / b					
4.	S → Aa / bAc / dc / bda A → d					
5.	S → Aa / bAc / Bc / bBa A → d B → d					
(6)	S → A / a A → a					
7.	S → SS / a / ∈					
8.	S → SS+ / SS* / a					
9.	S → xAy / xBy / xAz A → aS / q B → q					

OPERATOR PRECEDANCE PARSER

- It is constructed for both ambiguous and unambiguous grammar.
- Constructed only for operator grammar.
- used only for simple grammar with less complexity.

Operator Grammar : Grammar that does not contain ϵ and adjacent non-terminal of any production.

Ex. $A \to aA \mid \epsilon$ is not OG.

Ex $A \to EAE \mid id$ is not OG.

(Consecutive N.T.)

Ex $E \to E+E \mid E*E \mid E-E \mid id$ is OG. ✓

Ex $E \to E+T \mid T$ is O.G. ✓

$T \to T*F \mid F$

$F \to id$

Ex $S \to AB$

$A \to aA \mid b$ is not OG

$B \to bB \mid c$

Ex $S \to aS \mid b$

Operator Precedance Grammar :

- OG is said to be OPG, if its relational parse table is free from multiple entries.

Ex $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id.$

Ts	id	*	+	$
id	e_1	⋗	⋗	⋗
*	⋖	⋗	⋗	⋗
+	⋖	⋖	⋗	⋗
$	⋖	⋖	⋖	e_2

Errors

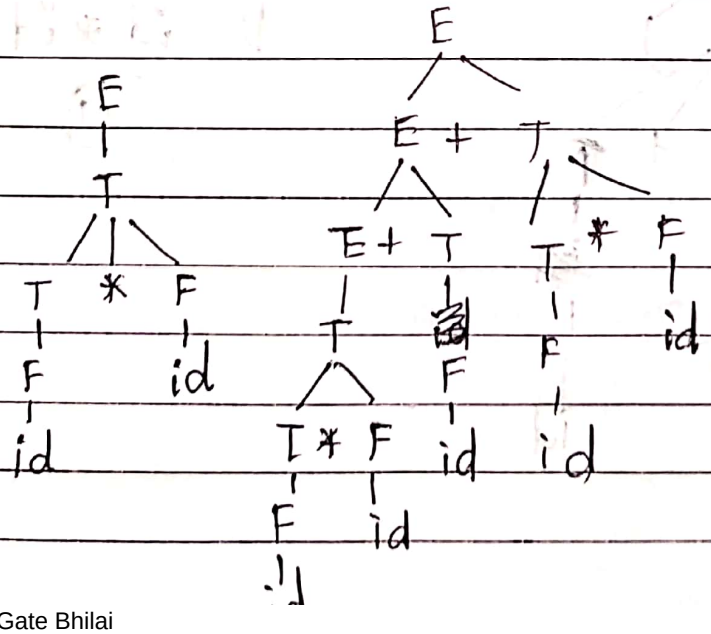
e_1 : Operator missing

e_2 : string missing (id missing)

if $a \lessdot b$: a has lower precedance than b

if $a \gtrdot b$ a has higher precedance than b

if $a \doteq b$ both a and b have equal precedance.



('id')

Que

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

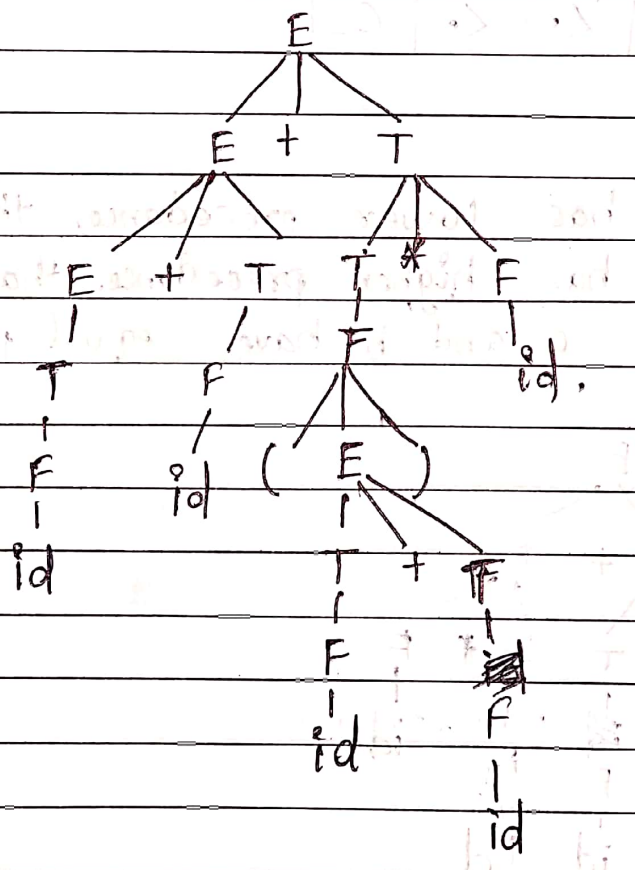
T's	id	+	(	)	*	+	$
id	e_1		e_1	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$		$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$	e_2
)	e_1		e_1	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
*	$\lessdot$		$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$		$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$		$\lessdot$	e_3	$\lessdot$	$\lessdot$	e_4

e_1 : operator m

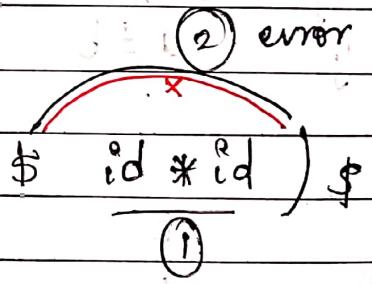
e_2 : close Missing

e_3 : open missing

e_4 : id / open missing



id * id + (id + id) * id



Operator Precedance Function

- The operator Precedance parser usually do not store the precedance table with the relation. Because high memory requirement, rather they are implemented in a special way.

- Operator precedance parser use precedance functions that map terminal symbol to integer and so the precedance relation between the symbol is implemented by comparison. (numerical comparison).

- Not every table of precedance relation has precedance function. But in practice, for most grammar, such funct" can be designed.

Algorithm for constructing precedance functions

1) create function f_a and g_a for each grammar terminal 'a' and '$'.

2) Partition the symbol in group so that f_a and g_b are in the same group if $A = B$.

3) Create a directed graph whose nodes are in the group.
   if $a \doteq b$: $f_a \rightarrow g_b$

   if $a \lessdot$: $f_a \leftarrow g_a$

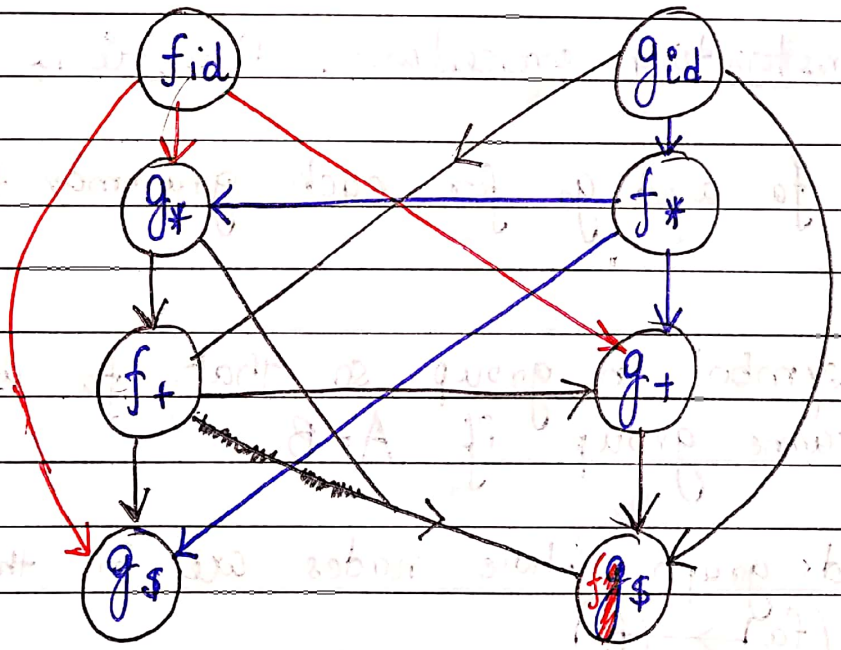
4. If connected graph has a cycle then no precedance function exist. When there are no cycle, collect the length of the longest path, from the group of f_a and g_b respectively.

Ex. $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

	g_{id}	g_*	g_+	$g_\$$
	Ts	id	*	+	$
f_{id}	id	-	$\cdot>$	$\cdot>$	$\cdot>$
f_*	*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
f_+	+	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$
$f_\$$	$	$<\cdot$	$<\cdot$	$<\cdot$	-



	id	*	+	$	
f	4	4	2	0	
g	5	3	1	0	

HT = 9

0

0 <— <9

Ques Given the following expression grammar.

$E \to E * E \mid F + E \mid F$

$F \to F - F \mid id$

True ?

(A) $* \cdot> +$

✓ (B) $- \cdot> *$

(C) $+ \doteq -$

(D) $+ \cdot> *$

Ques.) Consider following Grammar.

$S \to T * P$

$T \to U \mid T * U$

$P \to Q + P \mid Q$

$Q \to id \qquad U \to id.$

+ RA

* LA

+ >*

+ <· +

* >*

(A) + is Left asso * is RA

✓ (B) * is left asso + is RA

(C) Both are LA

(D) Both are RA

Ques Consider : $S \to CC$

$C \to cC \mid d$.

(A) LL(1)

(B) SLR(1) but not LL(1)

(C) LALR(1) but not SLR(1)

(D) LR(1) but not LALR(1)

Ques:) What will be the precedance and associativity of operator ~~and~~ $, @, \uparrow$

$A \to B \uparrow A \mid B$.

$B \to B \$ C \mid C$

$C \to D @ C \mid D$

$D \to \# - E \mid a$

$E \to b$.

$\uparrow$ is RA

$ is LA

@ is RA

precedance.

$\uparrow \lessdot \$ \lessdot @$

Ques:) Consider following grammar.

$S \to SXE \mid E$

$E \to FXE \mid F$

$F \to id$.

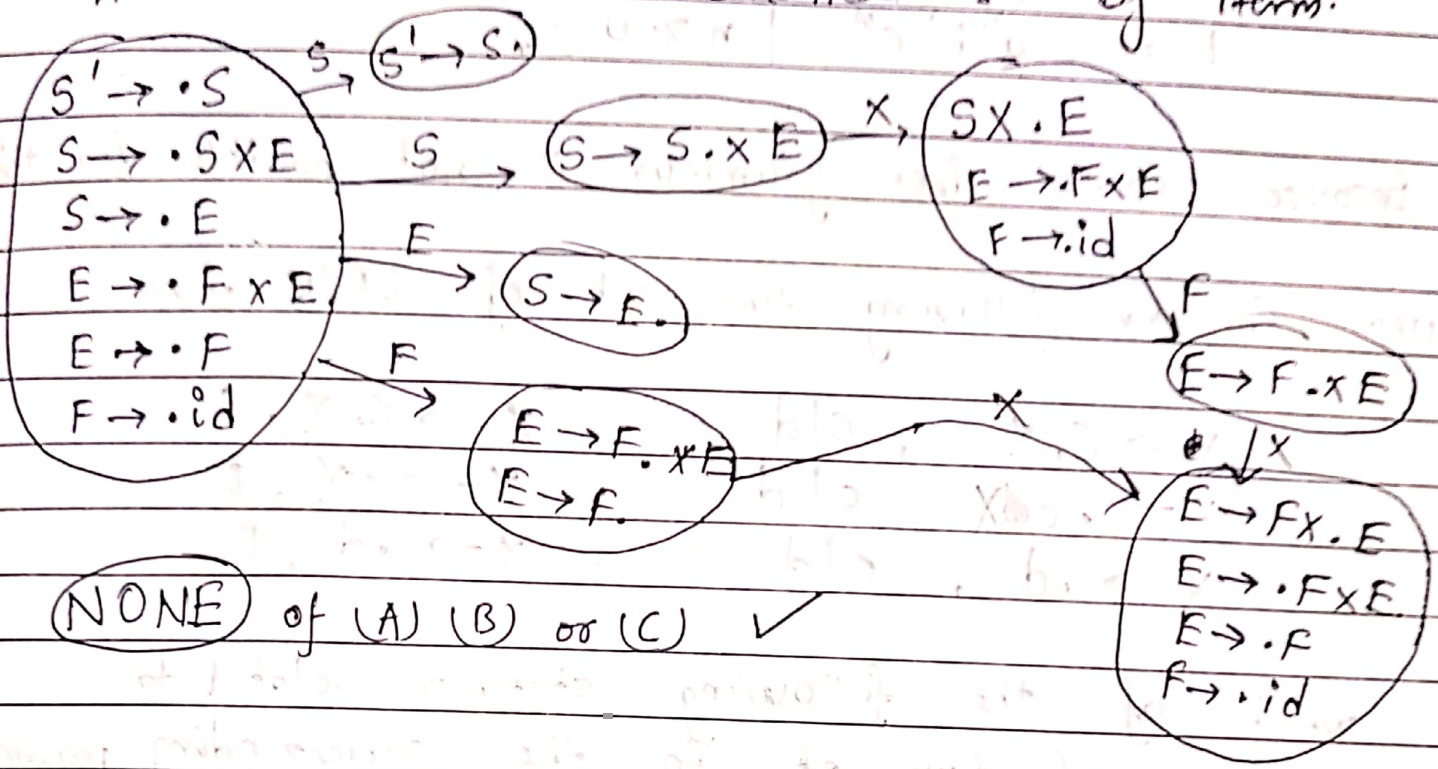
Consider following LR(0) items.

(A) $S \to SX.E$

(B) $E \to F.XE$

(C) $E \to FX.E$

Given the item above, which two of them will appear in the same canonical set of items?



NONE of (A) (B) or (C) ✓

(Ques) Consider following statement.

(A) Every regular grammar is LL(1) (False).
(B) Every regular set has LR(1) grammar (True)

which of the following is true?

Note: Every regular language is unambiguous. for every regular language there exist atleast one unambiguous grammar therefore it is true. (Point B).

(Ques:) Why is the following not LR(1)

$L = \{ a^n b^n c^n \mid n > 0 \}$.

Because : Context free grammar is not possible for this

(Ques:) Consider following two set of LR(1) item

$X \to C.X$, c|d	$X \to C.X$, $
$X \to .cX$, c|d	$X \to .cX$, $
$X \to .d$, c|d	$X \to .d$, $

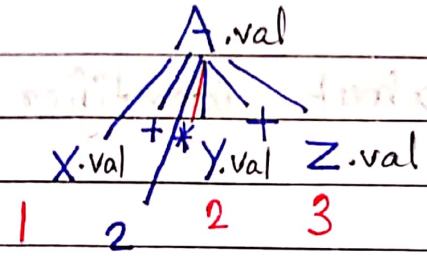
which of the following statemen related to merging of two set in the corresponding parser FALSE?

1.) Cannot be merged since look ahead are different. (false)
2.) Can be merged but will result in SR conflict. (false)
3.) Can be merged but will result in RR conflict. (false)
4.) ~~Since~~ can't be merge because goto on c will lead to two different set. (false)

SYNTAX DIRECTED TRANSLATION

- CFG — Semantic action

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



[Annotated / Decorated Parse Tree].

$1 + 2*2 + 3 = 8$

val = Attribute of id

- The context free grammar with semantic action is called as syntax directed translation.

- Apart from parsing, syntax directed translation can be used :-
  (i) To store information in symbol table.
  (ii) To perform type checking.
  (iii) Evaluation of algebric expression.
  (iv) To check variable conversion
  (v) To perform automatic type conversion.
  (vi) To construct a directed ~~acy~~ acyclic graph.

Annotated Parse Tree :

- The parse tree which show the attribute value with each and every node, is called annotated or decorated parse tree.

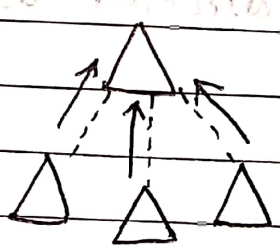
Attributes : Information about identifier is known as attribute.

(1) Name
(2) value
(3) Location
(4) Scope
(5) life
(6) Address.

Type of attribute : Based on the process of evaluation attributes can be classified into two types.

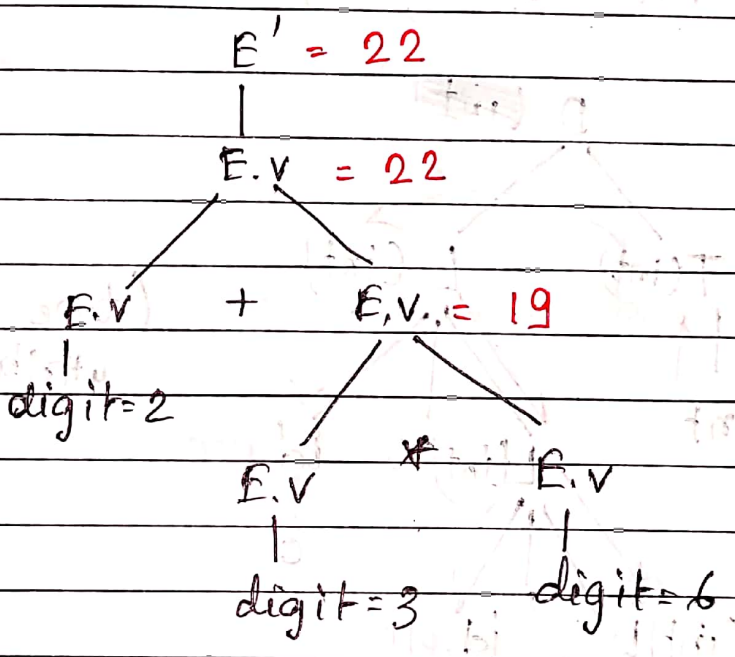
(1) Synthesized attribute.
(2) Inherited attribute.

Synthesized Attribute : The attribute whose value is evaluated from child to parent node is called synthesized attribute.



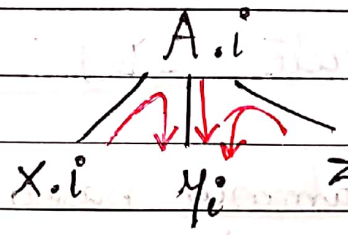
CFG	Semantic Action
$E' \rightarrow E$	{ print (E.val) }
$E \rightarrow E_1 + E_2$	{ E.val = E_1.val + E_2.val + 1 }
$E \rightarrow E_1 * E_2$	{ E.val = E_1.val * E_2.val + 1 }
$E \rightarrow$ digit	{ E.val = digit.lexval }

i/p = 2+3*6



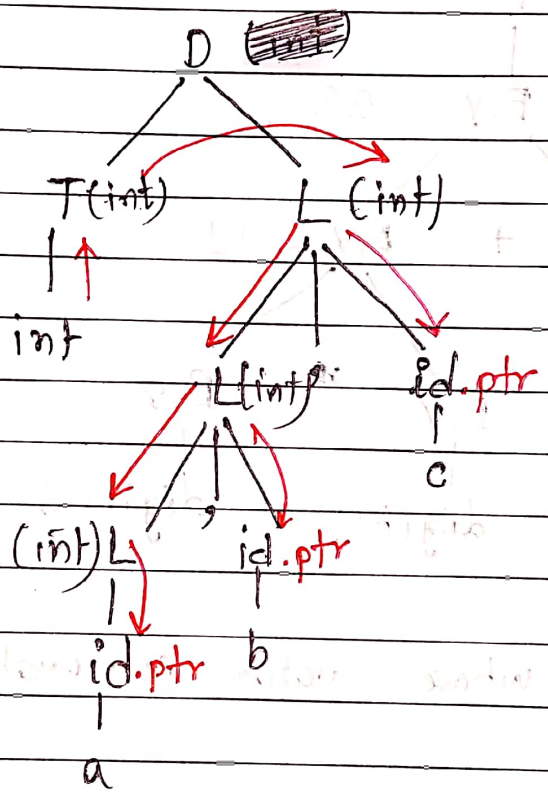
Inherited Attribute :

- The active root whose value is evaluated from its parent or sibling



CFG	Semantic Action
$D \rightarrow TL$	{ L.type = T.type }
$T \rightarrow$ int / real	{ T.type = int / real }
$L \rightarrow L_1$, id	{ L_1.type = L.type, enter (id.ptr, L.type) }
$L \rightarrow$ id	{ enter (id.ptr, L.type) }

i/p = int a, b, c ;



(type is inherited attribute)

Procedure to generate SDT :

(1) Construct the grammar based on input string.
(2) Construct the parse tree.
(3) Attach the semantic action by expecting the o/p.

Type of Syntax directed translation can be constructed in two ways :

1. S-attributed SDT
2. L-attributed SDT

S-Attributed :

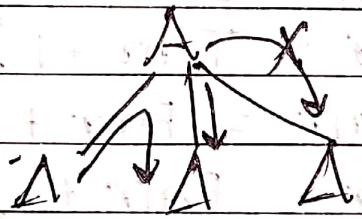
- S-Attributed only the synthesized attribute
- Symantic Action can be placed on the right side of the production.

$$A \rightarrow \alpha, \{ S.A \}$$
$$B \rightarrow \beta, \{ S.A \}$$

- Attributes are evaluated during bottom up or top down parsing.

L-attributed SDT :

- It uses both synthesized and inherited.
- Inherited attributes are restricted to inherit from parent or left sibling.



- Semantic action can be placed any where in producti

$$A \rightarrow \{ S.A \}\ \alpha$$
$$B \rightarrow \alpha_1 \{ S.A \}\ \beta_2$$

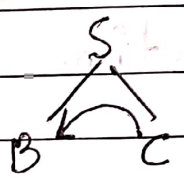
• Attribute can be evaluated using depth first left to right process.

• Note • Every S-attributed is L-attributed SDT but converse need not be true.

Ques Consider following SDT

$S \rightarrow BC \{ B.S = C.S \}$

S is ____



(A) S-Attributed

(B) L-attributed

(C) Both

✓(D) None.

Ques. $S \rightarrow BC \{ B.S = C.S \}$ L Attributed

Ques. $S \rightarrow BC \{ S.a = C.a \}$ Both S-L attributed

Ques

$E \rightarrow E * T \{ E.nr = E.nr + T.nr + 1 \}$

$E \rightarrow T \{ E.nr = T.nr + 1 \}$

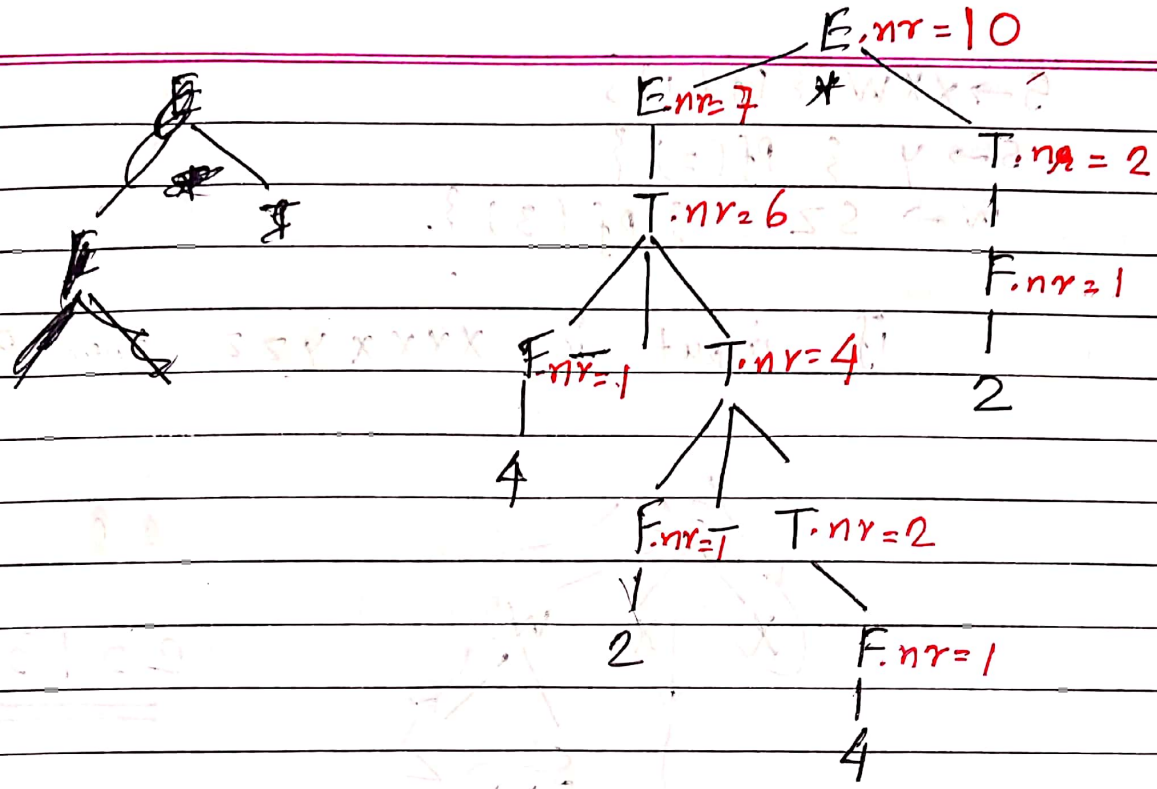
$T \rightarrow F - T \{ T.nr = F.nr + T.nr + 1 \}$

$T \rightarrow F \{ T.nr = F.nr + 1 \}$

$F \rightarrow 2 \{ F.nr = 1 \}$

$F \rightarrow 4 \{ F.nr = 1 \}$

i/p = 4-2-4*2 then o/p ____



Ques.) Consider following SDT :

$E \rightarrow E_1 + E_2$ { E.type = if (E_1.type == E_2.type && E_1.type == int) then return int else return error. }

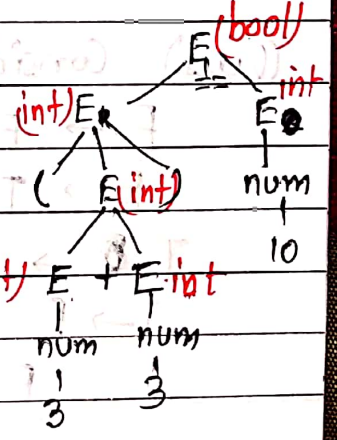
$E \rightarrow E_1 == E_2$ { E.type = if (E_1.type == E_2.type && E_1.type = int/bool) then return bool else return error. }

$E \rightarrow (E)$ { E.type = E_1.type }

$E \rightarrow num$ { E.type = int }

$E \rightarrow true$ { E.type = bool }

$E \rightarrow false$ { E.type = bool }



if ⇒ (3+3) == 10 then, output is ?

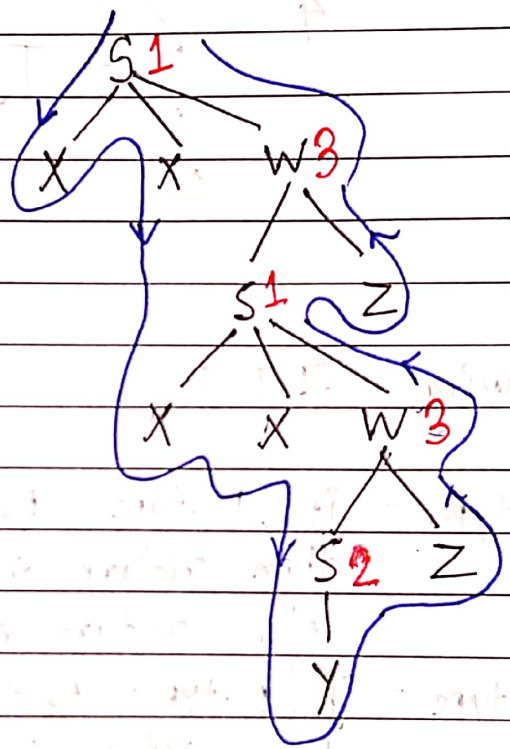
bool

Ques $S \rightarrow XXW \{ Pf(1) \}$

$S \rightarrow y \{ Pf(2) \}$

$W \rightarrow SZ \{ pf(3) \}$.

if input is xxxyzz then o/p = ?



2 3 1 3 1

(Ques.) Consider following SDT

$E \rightarrow E+T \{ print + \}$

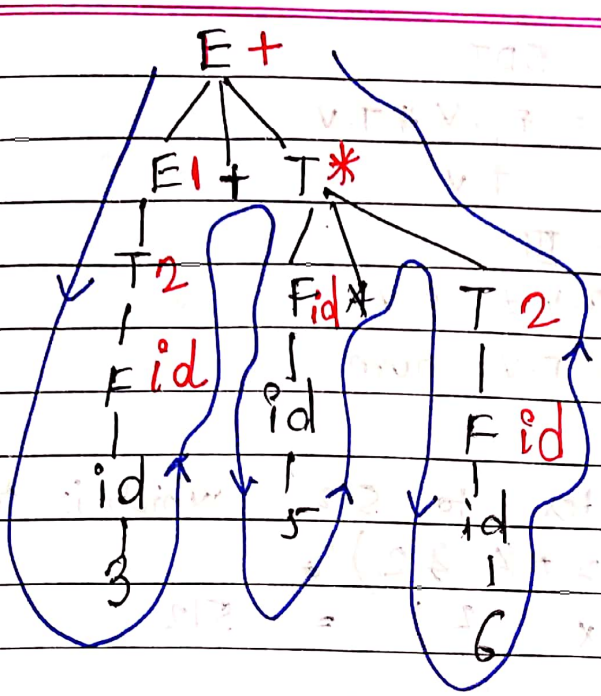
$E \rightarrow T \{ print\ 1 \}$

$T \rightarrow F*T \{ print * \}$

$T \rightarrow F \{ print\ 2 \}$

$F \rightarrow id \{ print\ id \}$

i/p = 3 + 5 * 6 : then



id 2 1 id id 2 * +
3 2 1 5 6 2 * +

Ques.) Consider following SDT

$E \rightarrow TE'$

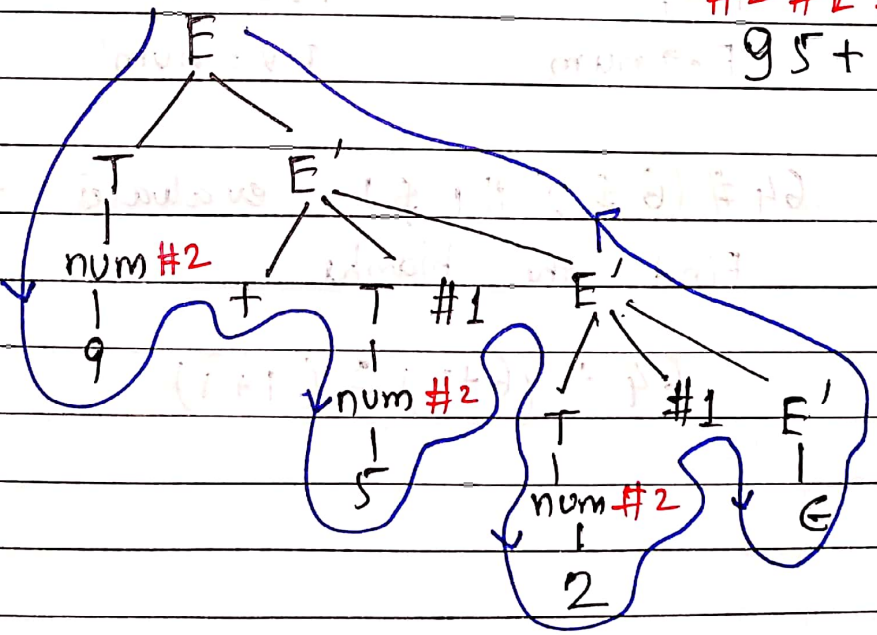
$E' \rightarrow +T \{ print + \} E'$

$E \rightarrow \epsilon$

$T \rightarrow num \{ print\ num.val \}$

ip. 9 + 5 + 2

#2 #2 #1 #2 #1
9 5 + 2 +



Ques) Consider the following SDT

$E \rightarrow E \# T$	$E.V = E_1.V * T.V$
$E \rightarrow T$	$E.V = T.V$
$T \rightarrow T \$ F$	III
$T \rightarrow F$	$T.V = F.V$
$F \rightarrow num$	$F.V = num$

If expression evaluates to 512, which is true?

$(8 \# (12 \$ 4)) \# (16 \$ 1) \# (4 \$ 2))$

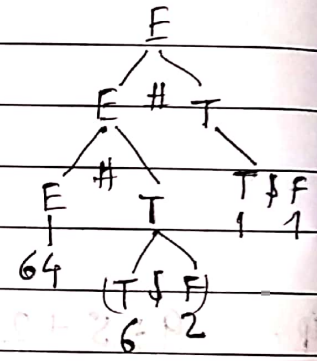
$(8 \times 8 \times 4 \times 2) = 512$

$\therefore$

$T.V = T_1.V - F.V$ ✓

Ques) Consider following SDT :

$E \rightarrow E \# T$	(1) _____
$E \rightarrow T$	$E.V = T.V$
$T \rightarrow T \$ F$	(2) _____
$T \rightarrow F$	$T.V = F.V$
$F \rightarrow num$	$F.V = num$



$64 \# (6 \$ 2) \# 1 \$ 1$ evaluates to 4.

Find the blanks

$64 \div (6+2) \div (1+1)$

Ques) Consider following SDT (Binary to decimal)

$L \to L \cdot L$	$\{ L.v = L_1.v + L_2.v \}$
$L \to L_1 B$	$\{ L.len = L_1.len + 1,\ L.v = L_1.v + 2^{L.len} * B.v \}$
$L \to B$	$\{ L.len = 1 \quad L.v = B.v / 2 \}$
$B \to 0$	$\{ B.v = 0 \}$
$B \to 1$	$\{ B.v = 1 \}$

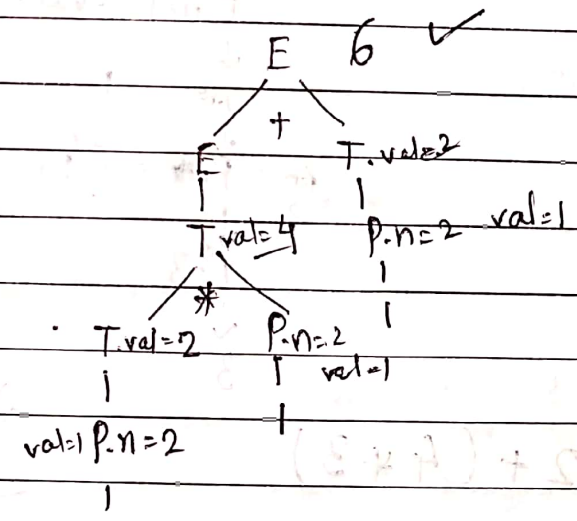
ip = 0.101 $\quad \frac{5}{8} = 0.625$ ==

Ques)

$E_1 \to E_2 + T$	$\{ E_1.v = E_2.v + T.v \}$
$E \to T$	$\{ E.v = T.v \}$
$T \to T * P$	$\{ T_1.v = T_2.v * P.v * P.n \}$
$T \to P$	$\{ T.v = P.v * P.num \}$
$P \to (E)$	$\{ P.v = E.v \}$
$P \to 0$	$\{ P.n = 1,\ P.v = 2 \}$
$P \to 1$	$\{ P.n = 2,\ P.v = 1 \}$

v = val
n = num

(ip) 1 * 1 + 1



val num

Ques.)

$S \to E$ $\quad S.V = E.V$ $\quad E.n = 1$

$E_1 \to E_2 \times T$ $\quad E_1.V = 2 * E_2.V + 2 * T.V$

$E_2.n = E_1.n + 1$

$T.n = E_1.n + 1$

$E \to T$ $\quad E.V = T.V$

$T.n = E.n + 1$

$T_1 \to T_2 + P$ $\quad T_1.V = T_2.V + P.V$

$T_2.n = T_1.n + 1$

$P.n = T_1.n + 1$

$T \to P$ $\quad T.V = P.V$

$P.n = T.n + 1$

$P \to (E)$ $\quad P.V = E.V$

$E.n = P.n$

$P \to i$ $\quad P.V = i / P.n$

(i) If i/p = 3 then o/p ? = 1

S = 1

↓

E.n=1 . val=1

↓

T.n=2 1

↓

P.n = P.V = i/P.n = 1

=3

↓

i.=3

↓

3

S.n=?

E=1

T_n=2

1/2

T_n=3 + P_n=3 38/5

2nd → P_4

(E_n=?)

4/6 E_n=4 * T_n=4. 3/5

E_n=4 → T_5 P_5 → i (3)

(4). i → P_6 → T_5

(ii) 2 + (4 * 3)

$2 \times \frac{1}{6} + 2 \times \frac{3}{5}$ $\quad \frac{38}{15} + \frac{1}{2}$

$\frac{1}{3} + \frac{6}{5}$

16+15 = 91/30

16/20

Intermediate Code Generator

C → obj code

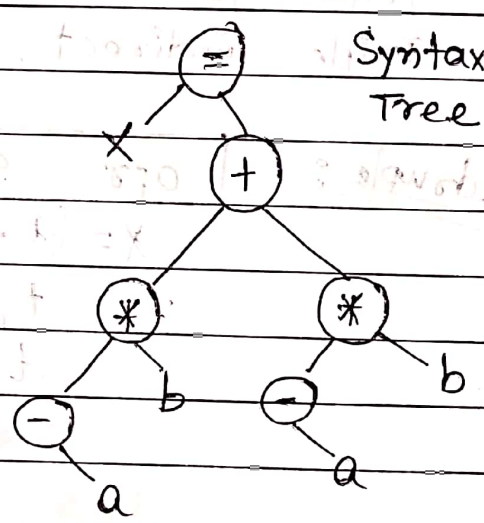
C → I.C → obj code

I.C

- Tree
  - Syntax Tree
  - DAG
- Linear Form
  - Three Address Code
  - Postfix Notation

Ex $X = (-a * b) + (-a * b)$

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



Syntax Tree

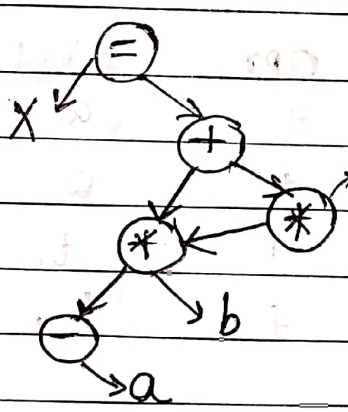
3AC: $t_1 = -a$

$t_2 = t_1 * b$

$t_3 = -a$

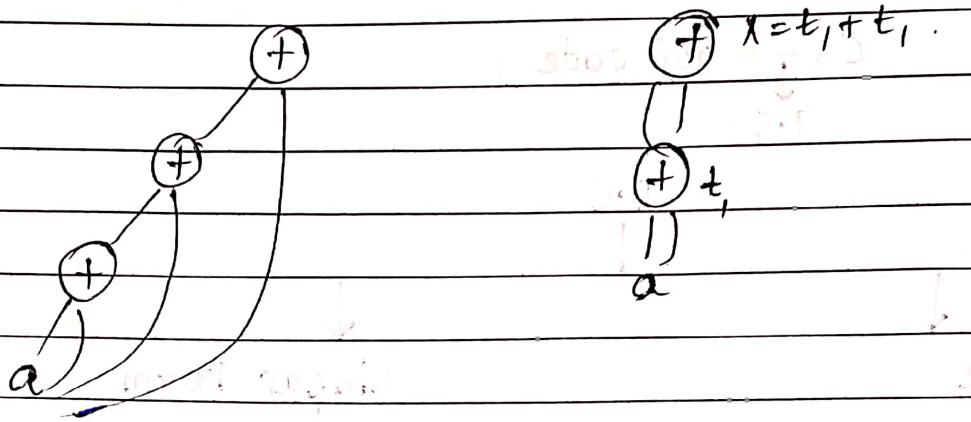
$t_4 = t_3 * b$

$t_5 = t_2 + t_4$



DAG : Eliminates Common Subexpression

Ex: $X = a+a+a+a$.



Implementation of 3AC:

It can be represented in 3 ways.

(i) Quadruple form

(ii) Triple form

(iii) Triple indirect.

Quadruple:

opr	src1	src2	Result

$X = (a+b) + (a+b+c)$

$t_1 = a+b$

$t_2 = a+b$

$t_3 = t_2 + c$

$X = t_1 + t_3$

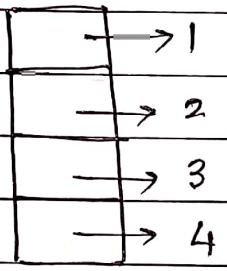
S.No	opr	src1	src2	Result
1	+	a	b	t_1
2	+	a	b	t_2
3	+	t_1	c	t_3
4	+	t_1	t_3	X

Tripple

S.No	Opr	src 1	src 2
1	+	a	b
2	+	a	b
3	+	(2)	c
4	+	(1)	(3)

Disadvantage: statements cannot be moved around for later purpose

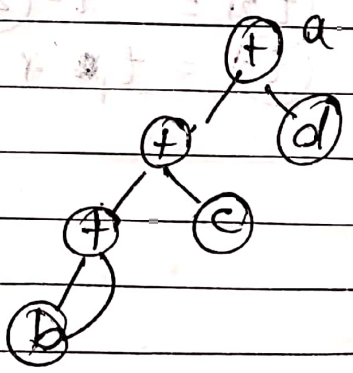
Indirect Array of Pointer



Ques Consider following statements.

a = b+c	~~a = b~~	~~a = b+c~~
c = a+d		~~c = a+d~~
d = b+c	d = a	~~d = a~~
e = d-b	e = c	~~e = c~~
a = e+b		~~a = b+c~~

c = b+c+d
d = b+b+c+d
e = b+c+d
a = b+c+d+b



6, 6 ✓

(7)

Ques The least number of temporary variable required to create 3AC in static single assignment.

Q $q + r / 3 + 5 - t * 5 + u * v / w$.

- In static single assignment, every variable assigned only single time (write only once) and can be read multiple times without assignment.

Ques Consider following code segment.

$x = u - t$
$y = x * v$
$x = y + w$
$y = t - z$
$y = x * y$.

minimum number of total variable required to convert the above code segment, to static single assignment form is ?

$x = u - t$
$y = x * v$
$t_1 = y + w$
$t_2 = t - z$
$t_3 = t_1 * t_2$

Total 10 variables

Quel) Consider following : 3AC code segment

```
p = a - b
q = p * c
p = u * v
q = p + q
```

static single assignment?

(A)
```
p1 = a - b
q1 = p1 * c
p1 = u * v
q1 = p1 + q1
```

(B) ✓
```
p3 = a - b
q4 = p3 * c
p4 = u * v
q5 = p4 + q4
```

(C)
```
p1 = a - b
q1 = p2 * c
p3 = u * v
q2 = p4 + q3
```

(D)
```
p1 = a - b
q1 = p * c
q2 = u * v
q3 = p + q
```

Ques.) Find 3AC of the following code:

```
if (a > b)
    x = y + z;
else
    p = p + r;
```

```
1. if a > b goto 4
2. p = q + r;
3. goto 5
4. x = y + z
5. End
```

Ques

```
switch (i+j)
{   case 1 : x = y+z ; break;
    case 2 : p = p+r ; break;
    default : u = v+w;
}
```

1. $t_1 = i+j$
2. goto 9
3. $x = y+z$
4. goto 12
5. $p = q+r$
6. goto 12
7. $u = u+w$
8. goto 12
9. if $t=1$ goto 3
10. if $t=2$ goto 5
11. goto 7
12. end.

Ques

```
for(i=1 ; i<=n ; i++)
    x = y+z.
```

(way 1)

1. $i = 1$
2. if $i<=n$ goto 7
3. goto 10
4. $t_1 = i+1$
5. $i = t_1$
6. goto 2

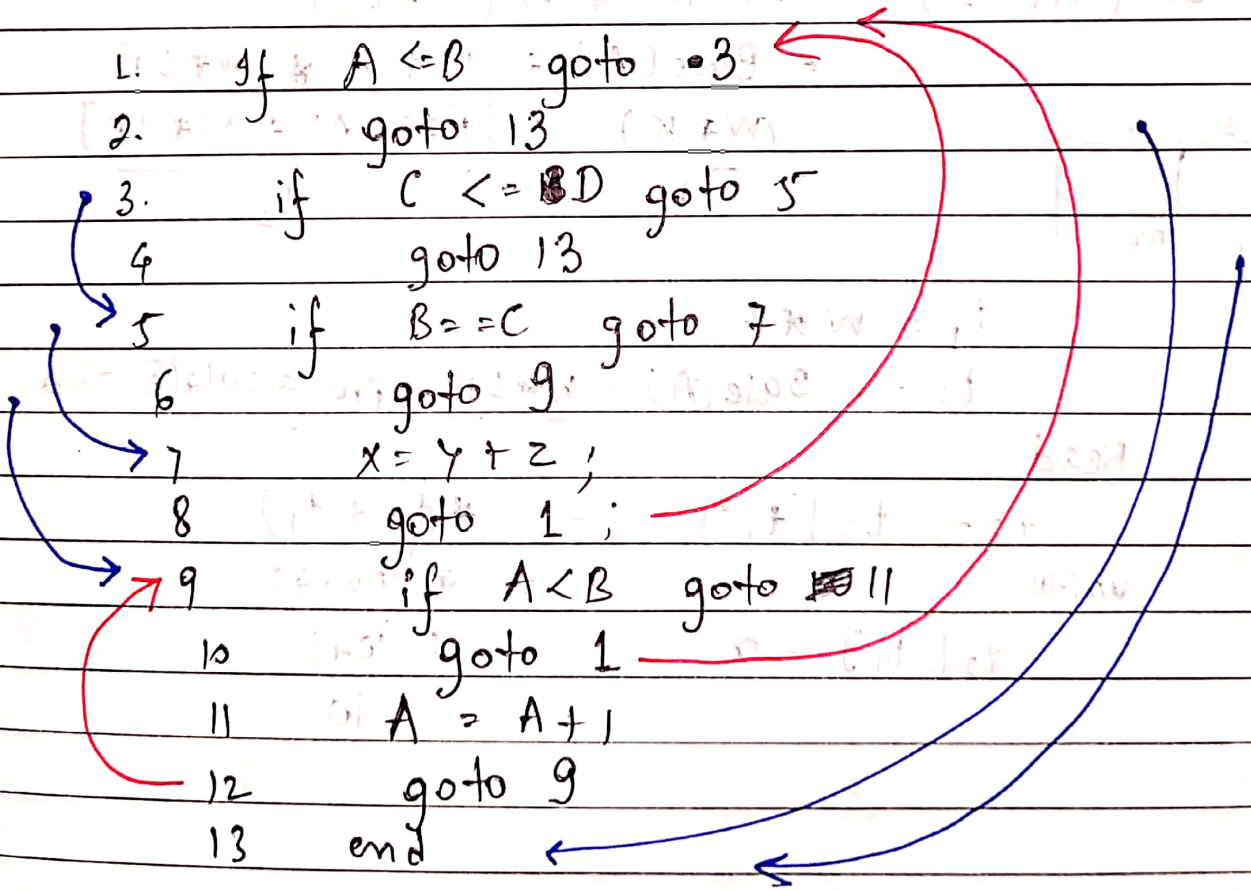
(way 2)

1. $i = 1$
2. if $i<n$ goto 8
3. $t_1 = y+z$
4. $x = t_1$
5. $t_2 = i+1$
6. $i = t_2$

7	$t_2 = Y + Z$	7	goto 2
8	$X = t_2$	8	end
9	goto 4		
10	end		

Ques.)

```
while ( A<=B && C<=D )
{  if (B==C)
       X= Y+Z;
   else
       while (A<B)
           A= A+1 ;
}
```

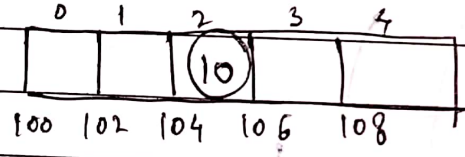


Ques Generate 3AC :

```
main()
{   int i=1;
    int a[10];          ⟶ *(a+i)
    while (i<=10)
    {   a[i] = i;
        i = i+1;
    }
}
```

1D : Array

$$Loc(A[k]) = Base(A) + w(K - LB)$$
$$= Base(A) + w*K - w*LB$$
$$= \underbrace{(w*K)}_{t_1} + \underbrace{[Base(A) - w*LB]}_{t_2}$$



$t_1 = w * k$

$t_2 = Base(A) - w*LB$ → precalculate this.

Read

$t_3 = t_2[t_1]$ ⟶ $*(t_2 + t_1)$
$*(100+4)$
$*(104)$
$= 10.$

write

$t_2[t_1] = x$

1.) ~~i=2~~ $i = 1$

2.) if $i <= 10$ goto 4

3.) goto 10

4.) $t_1 = 2 * i$

5.) $t_2 = Base(a) - 2$

6.) $t_2[t_1] = i$

7.) $t_3 = i + 1$

8.) $i = t_3$

9.) goto 2

10.) end.

} can be written directly $a[i] = i$

Ques.) Generate 3AC

```
sum = 0 ;
for ( i=1 ; i<= 20 ; i++)
    sum = sum + a[i] * b[i] ;
```

There are 4 bytes per word.

1. sum = 0
2. $i = 1$
3. if $i <= 20$ goto 5
4. goto 8
5. $t_1 = a[i] * b[i]$
6. $sum = sum + t_1$
7. goto 3
8. end.

→ Implicit

→ increment i
$t_{10} = i + 1$
$i = t_{10}$

(Explicit)
$t_2 = 4 * i$
$t_3 = Base(a) - 4$
$t_4 = t_3[t_2]$
$t_5 = Base(b) - 4$
$t_6 = t_5[t_2]$
$t_7 = t_4 * t_6$
$sum = sum + t_7$

Q111

```
begin
    add = 0 ;
    i = 1 ;
    j = 1 ;
    do
      begin
        add = add + a[i,j] * b[i,j] ;
            i = i+1 ;
            j = j+1 ;
      end
      while  i<=20  &&  j<=20 ;
end.
```

a & b are always of size 20×20 and there are 4 byte words.

1. add = 0
2. i = 1
3. j = 2
4. ~~add = add + a[i,j] *~~
4. t_1 = a[i,j] * b[i,j];
5. add = add + t_1
6. t_2 = i+1
7. i = t_2
8. t_3 = j+1
9. j = t_3
10. if i <= 20 goto 12
11. goto 14
12. if j <= 20 goto B 4
13. goto 14
14. end

for a & b evaluation

for a[i,j]

t_1 = 20 * i
t_1 = t_1 + j
t_1 = 4 * t_1 - 84
t_2 = Base(a)
t_3 = t_2[t_1]

RMO in 2D array.

$$Loc(A[i,j]) = Base(A) + W * [N_c * (i - LB_r) + (j - LB_c)]$$

$$= Base(A) + W * [N_c * i - N_c * LB_r + j - LB_c]$$

$$= Base(A) + W * [(N_c * i + j) - N_c * LB_r + LB_c]$$

$$= W * \underbrace{(N_c * i + j)}_{t_1} + Base(A) - \underbrace{W(N_c + LB_r + LB_c)}_{\text{Constant}}$$

$\rightarrow$

$t_1 = N_c * i$

$t_1 = t_1 + j$

$t_1 = 4 * t_1$

$t_2 = Base(A) - \underbrace{Constant}_{\text{precalculated}}$

Read $t_3 = t_2[t_1]$;

write $t_2[t_1] = x$

Ques.) c[a[i,j]] = b[i,j] + c[a[i,j]] + d[i+j]

where dimensions of array a & b are 30×40 & c & d is 20.

Assume subscript values start with 1 for all array dimension & every array element require 4 Bytes.

Code Optimization

- The process of reducing the instruction without effecting the outcome of source code is known as code optimization.

Type of code optimization :

- Machine independent optimization
- Machine dependent optimization.

- optimization of 3AC [Intermediate code] is known as machine independent optimization.

```
X = a + b * c
t1 = b * c          ⟹    t1 = b * c      X = b * c
t2 = a + t1               X = a + t1      X = a + X
X = t2.                                   ?
```

Type of machine independent optimization

(i) Local optimization
(ii) Global optimization
(iii) Loop optimization.

Local : Performed within block level is known as local optimization.

Global : Performed at program level is known as global.

Note : i) The complete source code is divided into blocks with the help of leader
ii) DAG is used with block level optimization
iii) Program flow graph is used in program level optimization.

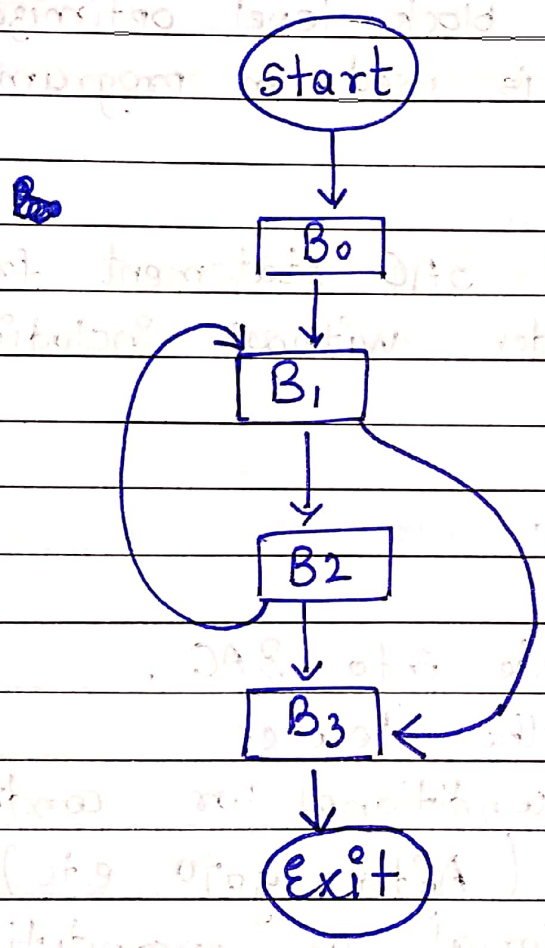
Basic block - Collection of 3AC statement from a leader to the next leader without including next leader

Rules to identify Leader

(i) Convert the source code into 3AC.
(ii) First statement is the leader
(iii) Statement following unconditional or conditional jump is a leader (After goto etc)
(iv) The target of conditional or unconditional or unconditional jump is a leader.

```
int i=1, f=1;
for (i=1; i<=n; i++)
    f = f * i;
return f;
```

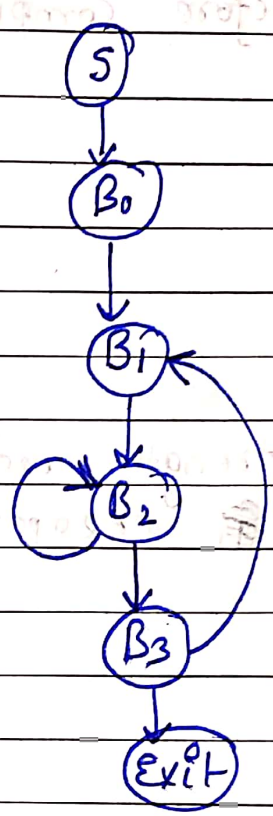
B_0	1. $i = 1$	L
	2. $f = 1$	
B_1	3. if $i > n$ goto 9	L
B_2	4. $t_1 = f * i$	L
	5. $f = t_1$	
	6. $t_2 = i + 1$	
	7. $i = t_2$	
	8. goto 3	
B_3	9. end	L



Nodes = 6
Edges = 7

Ques.) Consider the intermediate code given below :

```
1.  i = 1          L   B0
2.  j = 1          L   B1
3.  t1 = 5 * i     L   B2
4.  t2 = t1 + j
5.  t3 = 4 * t2
6.  t4 = t3
7.  a[t4] = -1
8.  j = j + 1
9.  if j <= 5 goto 3
10. i = i + 1      L   B3
11. if i < 5 goto 2
```



No of edges and Nodes in control flow graph. constructed for above code is ?

Nodes = 6

edges = 7

LOOP OPTIMIZATIONS :

- characteristics of Local / Global optimization.
- Replacing the value of constant before compile time is known as constant folding. It reduces runtime

Ex $X = y + 3 * 4$

$\downarrow$

$X = y + 12$.

- Constant propogation : Replacing value of expression before compiletime is known as constant propogation.

```
float pi = 22/7;          ⟹   Pi = 3.14
Area = pi * r * r                  ⇓
                          Area = 3.14 * r * r
```

- Strength reduction : Replacing the costlier operator by cheapest operator is known as strength reduction.

```
x = 2 * y;
    ↓
x = y + y    or    x = y << 1;

{ y = x * 32          }
{     ⇓               }
{ y = x << 5;         }

{ y = x / 32;         }
{     ⇓               }
{ y = x >> 5;         }
```

```
for( i=0; i<=10; i++)        ⟹    count = 0;
{ count = 5 * i }.                 for(i=1; i<=10; i++)
                                   { count = count + 5;
```

- Redundant code reduction : Don't repeat stored calculation.

```
x = a+b
y = b+a
  ⇓
x = a+b
y = x
```

- Code elimination : Dead code can be eliminated.

```
x=5; y=10;                    x=5; y=10
if (x=0)          ⟹          y=y+1
   x=x+1
else
   y=y+1
```

- ~~Loop~~ Loop optimization :

(1) Loop Jamming : Combine the body of two loops, when ever they are sharing the same index variable and same number of iteration.

```
for (int i=0 ; i<=100 ; i++)
    for (int j=0 ; j<=20 ; j++)
        a[i,j] = "TOC";

for (int i=0 ; i<=100 ; i++)
        printf ("CD");
```

```
for ( int i=0 ; i<=100 ; i++)
    { for ( int j=0 ; j<=20 ; j++)
        { a[i,j] = "TOC" ;
        }
      printf ("CD");
    }
```

Loop Fusion :

```
for ( int i=1 ; i<=100 ; i++)
    a[i] =100

for ( int j=1 ; j<=100 ; j++)
    b[j] = 50
```

⇓

```
for ( int i=1 ; i<=100 ; i++)
    { a[i] = 100
      b[i] = 50
```

Drawback : Locality of reference might be hampered

Loop fission : opposite of loop fusion.

- Loop interchange :

```
for ( i=1 ; i<=64 ; i++)
    for ( j=1 ; j<=64 ; j++)
        printf : a[j,i];
```

interchange.

Loop splitting :

```
for ( i=1 ; i<=100 ; i++)
  { if (i<=30)
      pf ("A");
    else if (i>=31 && i<=70)
      pf ("B");
    else
      pf ("C");
  }
```

```
for (i=1 ; i<=30 ; i++)
    printf ("A");

for (i=31 ; i<=70 ; i++)
    pf ("B");

for (i=71 ; i<=100 ; i++)
    pf ("C");
```

Loop unrolling :

```
int i=1 ;
while (i<=5)
  { pf ("*"); }
```

⇒

```
int i=1
pf
pf
pf
pf
pf .
```

Code Movement :

```
int a=2, b=3, i=1 ;

while ( i<=50)
  {   i= i+a*b;
      pf ( "%d", i);
  }
```

⇩

```
int a=2, b=3, i=1 ;
int c= a*b;
while ( i<=50)
  { i= i+c;
    pf ("%d", i);
  }
```

Ques.) `for(i=1 ; i<=(MAX-1) ; i++)` → take it outside.

Count++

`t = max-1` evaluate once.

Loop Peeling :

```
for (int i=1 ; i<=100 ; i++)
  { if ( i==1)
      pf ("Toc");
    else
      pf ("CD");
  }
```

pf ("Toc")

⇒ `for(int i=2; ...)`

pf ("CD");

Qeeee)
```
for (i=0 ; i<n ; j++)
{
    for (j=0 ; j<n ; j++)
    {
        if (i%.2)
        {
            X = (X + 4*j + 5*i);
            Y = (Y + 7 + 4*j);
        }
    }
}
```

which of the following is false?

(A) The code contain loop invariant computation. T

(B) There is a scope of common subexpression elimination T

(C) There is scope of strength reduction. T
$4*j = 2^2 \times j$ $j<<2$

(D) There is scope of dead code elimination
False (All code is reachable)

Question : Consider following code.

```
r0 r1
c = a+b
r2 r1 r0
d = c*a
r3 r1 r0
e = c+a
r1 r1 r1
X = c*c;
if (X > a)
    Y = a*a;
else { d = d*d;
       e = e*e;
     }
```

Why evaluate now?

Move in else.

Processor only has 2 registers.
Apply code motion optimization.
what is minimum no of spills to memory in the complied code :

```
r1  r0  r0,1
C = a+b
t1 [X] = c * c ;
   if ([X] > a)
        y = a * a ;
   else
   { d = c * a ;
     e = c + a ;
     d = d * d ;
     e = e * e ;
   }
```

(ii) what is minimum no of register needed in the processor without any spill to memory. and without any optimization. 4 ✓

Ques.) X[i][j][k] size int = 4B
char = 1B.

```
t0 = i * 1024
t1 = j * 32
t2 = k * 4
t3 = t1 + t0
t4 = t3 + t2
t5 = X[t4]
```

Which of the following statement about source code for the C program is correct.

(A) X is declared as int [32][32][8]

(B) X is declared as int [4][1024][32]

(C) X is declared as char X[4][32][4]

(D) X is delcared as char X[32][16][2]

$n_1 \times n_2 \times n_3$

$$X[i \times 1024 + j \times 32 + k \times 4]$$

$(n_2 \times n_3 / 4)$ $(n_3/4)$

$$n_2 = \frac{1024}{32} = 32/4 = 8$$

Alternate

$$X[i \times 256 + j \times 8 + k] \times 4$$

$$X[i \times 32 \times 8 + j \times 8 + k] \times 4$$

n_2 n_3 n_3

Ques.) How many minimum number of registers required to evaluate :

$(A+B) + (A-B) * (C+F)$

$t_1 = C + F$

$t_2 = A - B$

$t_1 = t_1 * t_2$

$t_2 = A + B$

$t_1 = t_1 + t_2$

2 registers

- Machine dependent optimization:
  The process of optimizing target code (assembly code) is known as machine dependent optimization.

- Type of machine dependent allocation :
  It is process of finding minimum number of registers that are allocated during target code optimization.

Peephole optimization :

- Eliminate redundant load and store instructions.

  Load R_0, a
  store a, R_0

- Avoiding unnecessary jump : (unnecessary Goto)
- Eliminating Dead code.

- Algebraic simplification:

$y = x + 0 \Rightarrow y = x$

$y = x * 1 \Rightarrow y = x$.