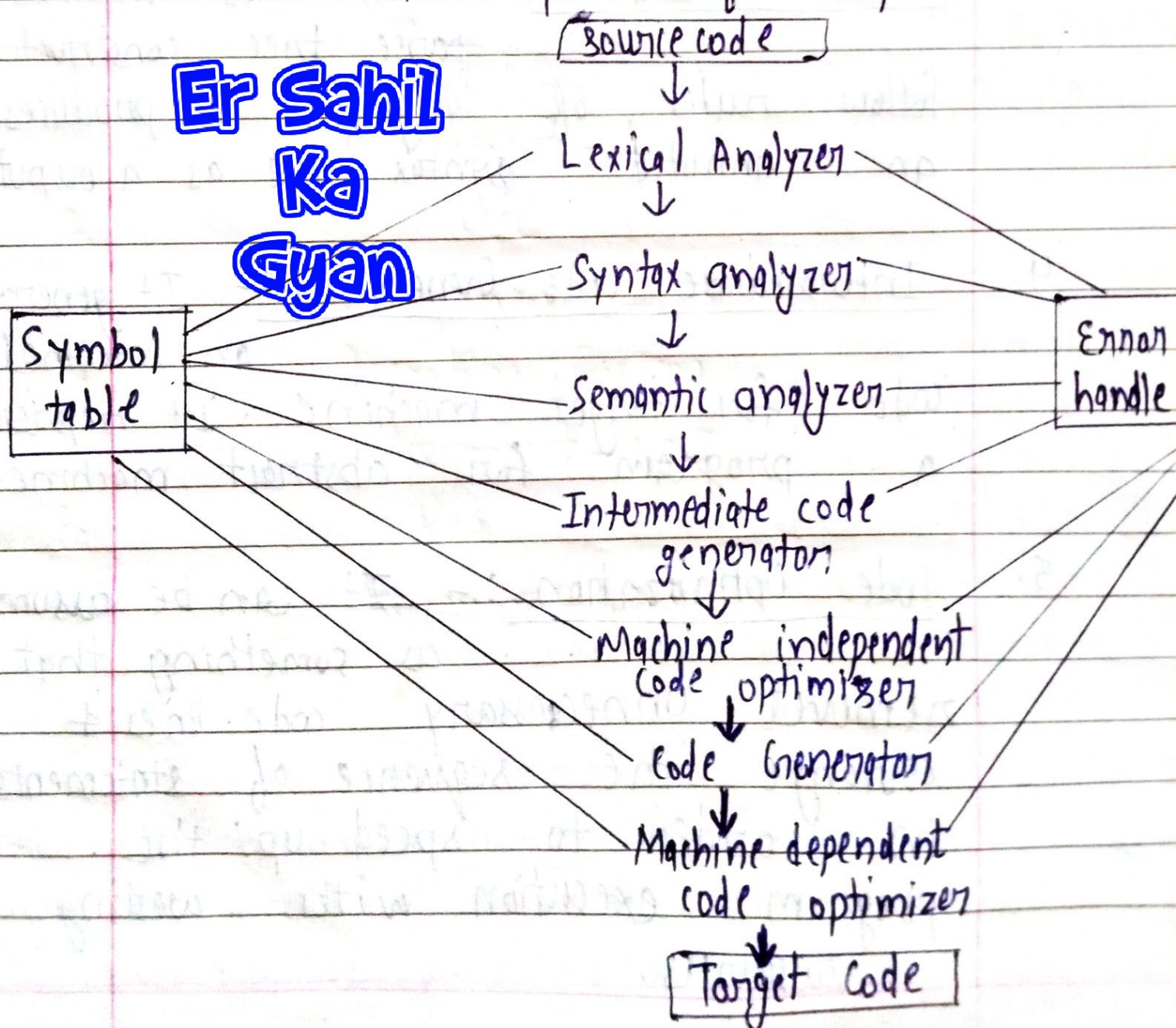


Q.1 Explain all phases of compiler give all phase result for given statement
"position = initial + rate * 60" with suitable diagram.

Ans — The compilation process is a sequence of various phase. Each phase takes input from previous stage has its own representation of source program & feeds its output to the next phase of compiler.

**Er Sahil
Ka
Gyan**



1. Lexical Analysis:- First phase of scanner works as a text scanner. This phase scans the code as a stream of character & converts it into meaningful lexemes.
2. Syntax Analysis:- It is also called parsing. It takes token from lexical & generates parse tree.
3. Semantic Analysis:- It checks whether parse tree constructed follow rules of language. It produces an annotated syntax tree as a output.
4. Intermediate Code Generation:- It generates an intermediate code for target machine. It represent a program for abstract machine.
5. Code Optimization:- It can be assumed as something that removes unnecessary code lines & arrange the sequence of statements in order to speed up the program execution with wasting resources.

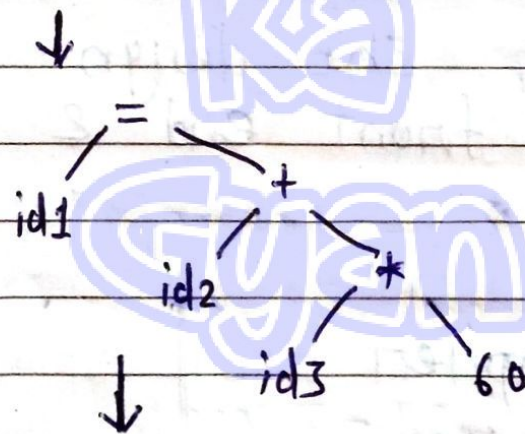
6. Code Generation: — It takes optimized representation of intermediate code & maps it to the target machine language.

Symbol table:- It is a data structure maintain through all the phases of compiler.

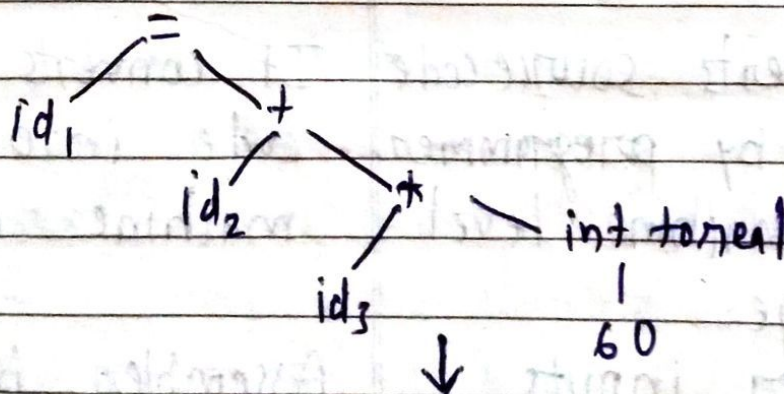
Statement \Rightarrow

position = initial + rate * 60 [Lexical Analyzer]

id1 = id2 + id3 * 60 [Syntax Analyzer]



[Semantic Analyzer]



[Intermediate code generator]

$t_1 = \text{int to real}(60)$

$t_2 = \text{id}_3 * t_1$

$t_3 = \text{id}_2 + t_2$

$id_1 = t_3$
↓

$t_1 = id_3 + 60$

$id_1 = id_2 + t_2$
↓

MOV id_3, R_2

MULF $\#60.0, R_2$

MOV id_2, R_1

ADD R_2, R_1

MOV R_1, id_2

[Code Generation]

[Code Optimizer]

Q.2 What are the difference b/w compiler & assembler. What is the advantage of dividing the design of a compiler into front end & back end design.

Ans-

Compiler

Assembler

1. It converts source code written by programmer to a machine level language.

It converts assembly code into the machine code.

2. Compiler inputs source code.

Assembler inputs assembly lang. code.

3. It checks & converts at one time.

It does not convert at one time.

4.

The output of compiler is a mnemonic version of machine code.

The output of assembler is a binary code.

5.

C, C++, Java & C# are example.

GNU, GNU is an example.

Advantage:-

You need only write one front end and then one back end for each target architecture. This approach cuts down the amount of work required to target another architecture.

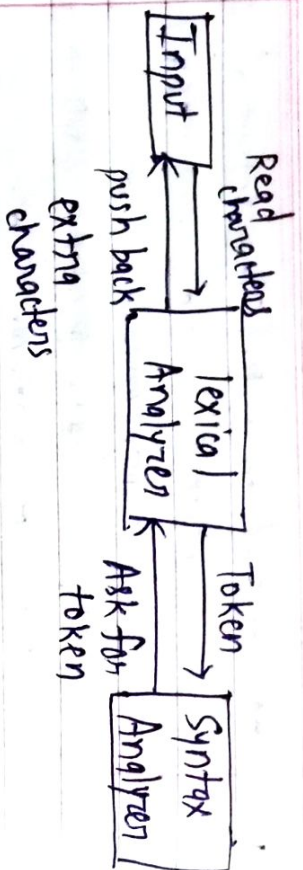
Q.3

What are the main function performed by lexical analyzer? write "C" language code for lexical Analyzer to eliminating white space & collecting no. from input code.

Ans-

The main task of lexical is to read input characters in the code & produce tokens.

The output is a sequence of tokens that is sent to the parser for syntax analysis.



Code:-

```

#include <stdio.h>
#include <string.h>
#include <conio.h>

void main()
{
    char str1[50] = {"My no. are 1234 & 9"};
    char str2[20] = "";
    int i, j;
    clrscr();
    j = 0;
    for (i = 0; str1[i] != '\0'; i++)
    {
        if (str1[i] == ' ') && (str1[i+1] != '\t')
        {
            if (str1[i] >= '0' && str1[i] <= '9')
            {
                str2[j] = str1[i];
                j++;
            }
        }
    }
}
  
```

```

str2[j] = '\0';
printf("\n Eliminating white space on\n\n");
for (i = 0; str1[i] != '\0'; i++)
    printf("%c", str1[i]);
getch();
}
  
```

Output:

Eliminating white space on collecting numbers	
1	
2	
3	
4	
9	

Q.4 Explain following methods of input buffering.

Ans

1. Buffer Pairs: — Because of amount of time taken to process characters and the large no. of characters that must be processed during the compilation of a large source program, specialized buffering

techniques have been developed to reduce the amount of overhead required to process a single input character

Two points are maintained:-

1. Pointer lexeme begin, marks the beginning of current lexeme, whose extent it is attempting to determine
2. Pointer forward, scans ahead until a pattern match is found,

2. Sentinels:-

If we use the scheme of buffer pairs, we must check, each time we advance forward, that we have not moved off of the buffers; if we do, then we must also reload the other buffer. Thus, for each each character read, we make two tests: one for the end of the buffer and one to determine what characters are read.

Q.5

Write a grammar to define simple arithmetic expression & identify terminal symbol & non-terminal symbols in the grammar.

Ans-

Let the grammar is

$E \rightarrow TA$

$A \rightarrow +TA / \epsilon$

$T \rightarrow FB$

$B \rightarrow *FB / \epsilon$

$F \rightarrow (E) / id$

Productions are $E \rightarrow TA$

$A \rightarrow +TA$

$T \rightarrow FB$

$B \rightarrow *FB$

$F \rightarrow (E)$

$F \rightarrow id$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Terminals in the grammar are:

$+, *, (,), id$

Non-terminals in the grammar are:

ϵ, T, A, F, B

Context Free Grammar

For specifying the syntax of a language we use a specified notation called a context-free grammar.

Context-free grammar has 4 components :

- (1) A set of tokens, known as terminal symbols.
- (2) A set of non-terminals.
- (3) A set of productions.
- (4) A designation of one of the non-terminals as a "start" symbol.

The syntax of programming language constructs can be described by context-free grammar or BNF. Grammars offer significant advantages to both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.

- From certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed.

- It imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors.

Parse Trees : A parse tree is a tree with following properties :

- (1) The root is labeled by start symbol.
- (2) Each leaf is labeled by token or by ϵ .
- (3) Each interior node is labeled by non-terminal.
- (4) If A is the non-terminal labeling some interior node and X_1, X_2, \dots, X_n are the labels of that node from left to right then

$A \rightarrow X_1 X_2 \dots X_n$ is a production, here, X_1, X_2, \dots, X_n stand for a symbol that is either terminal or non-terminal :

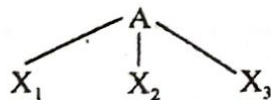


Fig. 1 : A root "A" with X_1, X_2, X_3 children

Ambiguity : A grammar can have, more than one parse tree generating a given string of tokens, such grammar

is said to be "ambiguous". So we have to be careful in taking the structure according to grammar. e.g.

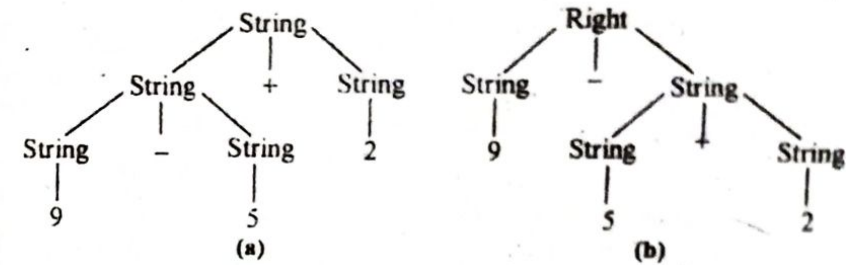


Fig. 2 : Parse trees for $(9 - 5 + 2)$

Eliminating Ambiguity : Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example.

stmt \rightarrow if expr then stmt
 /if expr then stmt else stmt
 /other

Here "other" stands for any other statement.

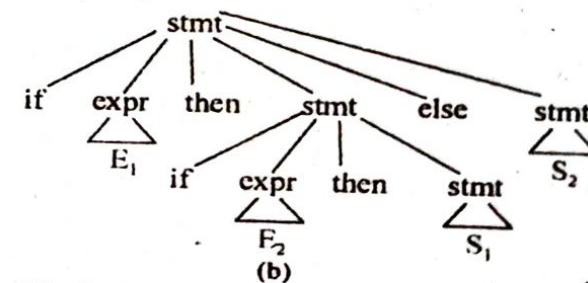
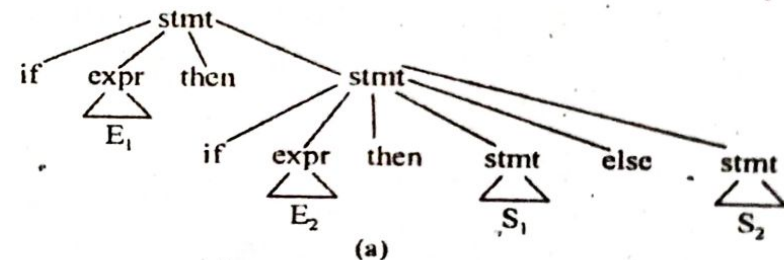


Fig. 3 : Parse trees for an ambiguous statement.

To eliminate ambiguity :

stmt \rightarrow matched_stmt /unmatched_stmt

matched_stmt \rightarrow if expr then matched_stmt else matched_stmt

/other

unmatched_stmt \rightarrow if expr then stmt

/if expr then matched_stmt else

unmatched_stmt

Parsing : Parsing is the "process of determining" if a string of token can be, generated by grammar.

There are different parsing method that can be applied to construct Syntax - directed translation.

Basically \rightarrow "top down method" and bottom-up method," for parsing.

Top-Down Parsing : The top-down construction of a parse tree is done by starting with the root, labeled with the starting non-terminal and repeatedly performing the following two steps.

1. At node n labeled with non-terminal A , select one of the productions for A and construct children at n for the symbols on the right side of the production.

2. Find the next node at which a subtree is to be constructed.

Associativity of Operators

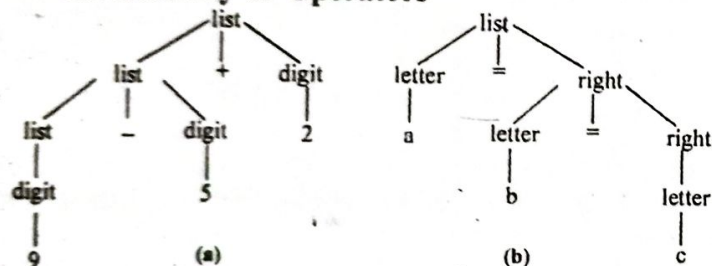


Fig. 4 : Parse trees for left and right associative operators

In the grammar there are various operators and each operator has different associative properties i.e. e.g., the contrast between a parse tree for a left -associative operator like "-" and parse tree for right-associative operator like "=" is shown above.

Precedence of Operators : In the grammar, each operator has a different precedence with other operator, so they operate accordingly to there precedence level (higher to lower).

Predictive Parsing : Recursive - descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A special form of Recursive - descent - parsing is called predictive parsing, in which the look-ahead symbol unambiguously determines the procedure selected for each non-terminal.

Non-Recursive Predictive Parsing

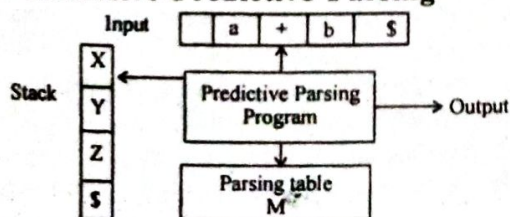


Fig. 5 : Model of a nonrecursive predictive parser

Non recursive predictive parsing can be build by maintaining a stack. Explicitly, rather than implicitly via recursive calls. The non recursive parser lookup the production to be applied in parse table. A table-driven predictor parser has an input buffer, a stack, a parsing table and an output stream.

FIRST and FOLLOW : The construction of predictive parser is aided by two functions associated with grammar G . These functions, FIRST and FOLLOW, allow us to fill in the entries at a predictive parsing table for G , whenever possible.

The Role of the Parser : The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.

There are three general types of parsers for grammars :

- Universal parsing method such as, Cocke Younger-Kasami algorithm and Earley's algorithm can parse any grammar.
- Top-down parsing method, it build parse trees from the top (root) to bottom (leaves),
- Bottom-up parsing method, parsers start from the leaves and work up to the root.

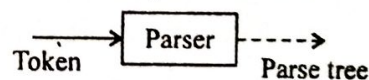


Fig. 6

Syntax Error Handling

Programs can contain errors at many different levels.

For example, errors can be :

- Lexical such as misspelling.
- Syntactic, such as an arithmetic expression with unbalanced parentheses.
- Semantic, such as an operator applied to an incompatible operand.
- Logical, such as an infinitely recursive call.

The error handler in a parser has simple-to-state goals :

- It should report error, clearly and accurately.
- It should recover from each error quickly.
- It should not significantly slow down the processing of correct program.

Error Recovery Strategies :

- Panic mode.
- Phrase level.
- Error production.
- Global correction.

Panic Mode : On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing token is found.

Q.1
 Write difference b/w bottom-up & Top-down parsing with suitable example.

Top-down parsing is a parsing technique that first look at the highest level of parse tree and works down the parse tree by using the rules of grammar thus parsing technique uses left most derivation.

Bottom-up is a parsing technique that first look at the lowest level of the parse tree and works up the parse tree by using the rule of grammar. This parsing technique uses right most derivation.

Ex of top-down parser:—

$S \rightarrow aABe$
 $A \rightarrow Abc|b$
 $B \rightarrow d$

input = abbcde
 we use left most derivation

$S \rightarrow aABe$
 $S \rightarrow aAbcBe$ [A $\rightarrow Abc$]
 $S \rightarrow abbcBe$ [A $\rightarrow b$]
 $S \rightarrow abbcde$ [B $\rightarrow d$]

Hence, we get our input.

Ex of Bottom UP parsing

$S \rightarrow aABe$
 $A \rightarrow Abc|b$
 $B \rightarrow d$

we use right most derivation
 input = abbcde

$S \rightarrow aABe$ [B $\rightarrow d$]
 $S \rightarrow aAbde$ [A $\rightarrow Abc$]
 $S \rightarrow abbcde$ [A $\rightarrow b$]

Hence, we get our input.

Q.2 What do you mean by parser conflicts?

Explain different types of parser conflict in detail.

Ans— There are two kinds of conflicts that can occur in an SLL(1) parsing table

1. A shift Reduce (SR) conflict occur in a state that request both a shift action & a reduce action.

2. A reduce reduce (RR) conflicts occur in a state that request two or more different reduce actions.

1. Shift Reduce Conflict:-

It is most common conflict found in grammar. It is caused when the grammar allows a rule to be reduced for particular token, but at some time allowing another rule to be shifted for last same token. As a result the grammar is ambiguous since a program can be interpreted more than one way. This error is often caused by recursive grammar definition where the system can't determine when one rule is completed and another is just started.

2. Reduce Reduce Conflict:-

It occurs if there are two or more rules that apply to same sequence of input. This usually indicates a serious error in grammar when this

Recursive Decent Parsing \Rightarrow

Eg $\text{Expr} \rightarrow \text{term Rest}$
 $\text{Rest} \rightarrow + \text{term Rest} \mid - \text{term Rest} \mid \epsilon$
 $\text{term} \rightarrow 0 \mid 1 \mid \dots \mid 9$

Solⁿ -

```
Expr() {
    term();
    Rest();
    return();
}
```

```
Rest() {
```

```
    if (lookhead == '+')
    {
        match(); term(); Rest(); return();
    }
    else if (lookhead == '-')
    {
        match(); term(); Rest(); return();
    }
}
```

```
term() {
    if (isDigit[lookhead])
    {
        match();
        return();
    }
    else {
        error();
    }
}
```

Top-down Predictive Parser \Rightarrow Eg - $x + y \$$ Eg - input: $\text{id} + \text{id} * \text{id}$ $E \rightarrow TE'$, $E' \rightarrow +TE' \mid \epsilon$ $T \rightarrow FT'$, $T' \rightarrow *FT' \mid \epsilon$ $F \rightarrow \text{id} \mid (\epsilon)$

Stack	Input	Production
$E \$$	$\text{id} + \text{id} * \text{id} \$$	
$TE' \$$	$\text{id} + \text{id} * \text{id} \$$	$E \rightarrow TE'$
$FT'E' \$$	$\text{id} + \text{id} * \text{id} \$$	$T \rightarrow FT'$

$idT'E' \$$	$id + id * id \$$	$F \rightarrow id$
$T'E' \$$	$+ id * id \$$	$T' \rightarrow \epsilon$
$E' \$$	$+ id * id \$$	$E' \rightarrow +TE'1$
$+TE' \$$	$+ id * id \$$	
$TE' \$$	$id * id \$$	
$FTE' \$$	$id * id \$$	$T \rightarrow TE'1$
$idTE' \$$	$id * id \$$	$F \rightarrow id$
$T'E' \$$	$* id \$$	$T \rightarrow *ET'1$
$*ET'E' \$$	$* id \$$	
$FTE' \$$	$id \$$	$F \rightarrow id$
$idTE' \$$	$id \$$	$F \rightarrow id$
$T'E' \$$	$\$$	$T' \rightarrow \epsilon$
$E' \$$	$\$$	$E' \rightarrow \epsilon$
$\$$	$\$$	

First and Follow:- It is associated with a grammar that help us fill in the

entries of an M-table.

First():- It gives set of terminals that begin the strings derived from production rule.

Eg- $E \rightarrow TA$

$A \rightarrow +TA | \epsilon$, $T \rightarrow FB$, $B \rightarrow *EB | \epsilon$

$F \rightarrow (E) | id$

Soln- $First(E) = First(TA) = First(T) = First(FB) = First(F)$

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

$First(A) = \{ +, \epsilon \}$

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

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

$First(B) = \{ *, \epsilon \}$

Follow():- It gives set of terminals that can appear immediately to right of given symbol.

8. $S \rightarrow AaAb | BbBa$ $First(S) = ?$

$$First(S) = First(AaAb) \cup First(BbBa) \\ = (First(A) - \{ \epsilon \}) \cup (First(Ab) \cup (First(B) - \{ \epsilon \}) \cup First(a))$$

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

Eg- $E \rightarrow TA$, $A \rightarrow +TA | \epsilon$, $T \rightarrow FB$, $B \rightarrow *EB | \epsilon$
 $F \rightarrow (E) | id$

$$Follow[E] = \{ \$, \epsilon \}$$

$$Follow[A] = Follow[E]$$

$$= \{ \$, \epsilon \}$$

$$Follow[T] = Follow[E] \quad [F \rightarrow (E) | id]$$

$$= \{ + \}$$

and by production $A \rightarrow \epsilon$
 $E \rightarrow T$

$$Follow[T] = Follow[E] = \{ \$, \epsilon \}$$

$$So \quad Follow[T] = \{ \$, \epsilon, + \}$$

$$Follow[B] = Follow[T] \quad [T \rightarrow FB]$$

$$= \{ \$, \epsilon, + \}$$

$$Follow[E] = Follow[B] \quad [B \rightarrow *EB]$$

$$= \{ *, \epsilon \}$$

$$if \quad B \rightarrow \epsilon$$

$$T \rightarrow F$$

$$Follow[F] = Follow[T] = \{ \$, +, \epsilon \}$$

$$= \{ \$, +, \epsilon \}$$

Production	First	Follow
$E \rightarrow TA$	$\{ (, id, \$ \}$	$\{ \$, + \}$
$A \rightarrow +TA \epsilon$	$\{ +, \epsilon \}$	$\{ \$, +, \epsilon \}$
$T \rightarrow FB$	$\{ (, id, \$ \}$	$\{ +, \$, \epsilon \}$
$B \rightarrow *EB \epsilon$	$\{ *, \epsilon \}$	$\{ +, \$, \epsilon \}$
$F \rightarrow (E) id$	$\{ (, id, \$ \}$	$\{ *, +, \$, \epsilon \}$

$First(A)$ then substitute ϵ on TA $Follow(A)$

Select() for M-table :-

$$\text{Select}(A \rightarrow w) = \text{First}(\text{First}(w) \text{ Follow}(A))$$

$$\text{Select}(E \rightarrow TA) = \text{First}(\text{First}(TA), \text{Follow}(E))$$

$$\begin{aligned} &= \text{First}(\{ \epsilon, id, a, b \}, \{ \epsilon \}) \\ &= \text{First}(\{ \epsilon, id, a, b \}, \{ \epsilon \}) \\ &= \{ \epsilon, id, a, b \} \end{aligned}$$

$$\text{Select}(A \rightarrow +TA) = \text{First}(\text{First}(TA) \times \text{Follow}(A))$$

$$= \{ + \}$$

$$\text{Select}(A \rightarrow \epsilon) = \text{First}(\text{First}(\epsilon) \times \text{Follow}(A))$$

$$= \text{First}(\{ \epsilon \} \times \{ \epsilon, a, b \})$$

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

$$= \{ \epsilon, a, b \}$$

$$\text{Select}(T \rightarrow FR) = \text{First}(\text{First}(FR) \times \text{Follow}(T))$$

$$= \text{First}(\{ \epsilon, id, a \} \times \{ \epsilon, a, b \})$$

$$= \{ \epsilon, id, a \}$$

$$\text{Select}(B \rightarrow +FR) = \text{First}(\text{First}(FR) \times \text{Follow}(B))$$

$$= \{ + \}$$

$$\text{Select}(B \rightarrow \epsilon) = \text{First}(\text{First}(\epsilon) \times \text{Follow}(B))$$

$$= \{ \epsilon, a, b \}$$

$$\text{Select}(F \rightarrow (E)) = \text{First}(\text{First}((E)) \times \text{Follow}(F))$$

$$= \{ (\}$$

$$\text{Select}(F \rightarrow id) = \text{First}(\text{First}(id) \times \text{Follow}(F))$$

$$= id$$

Finally with help of select values we can construct M-table as follows:

$$L(L(1) \text{ grammar}) = \{ a^+ T A \} \cap \text{Select}(A \rightarrow \epsilon) = \{ \epsilon, a, b \} \cap \{ \epsilon, a, b \} = \emptyset$$

DATE: / /
PAGE NO.:

Non-terminals	(id	+	*)	\$
E	TA	TA				
A			+TA			
T	FR	FR				
B						
F	(E)	id		*FR		

Q. $S \rightarrow AaAb / BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

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

$$\text{First}(A) = \text{First}(B) = \epsilon$$

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

$$\text{Follow}(A) = \{ \epsilon, a, b \}$$

$$\text{Follow}(B) = \{ \epsilon, a, b \}$$

$$\text{Select}(S \rightarrow AaAb) = \text{First}(\text{First}(AaAb) \times \text{Follow}(S))$$

$$\text{Select}(S \rightarrow BbBa) = \text{First}(\text{First}(BbBa) \times \text{Follow}(S))$$

$$\text{Select}(S \rightarrow \epsilon) = \text{First}(\text{First}(\epsilon) \times \text{Follow}(S))$$

$$\text{Select}(B \rightarrow \epsilon) = \{ \epsilon, a, b \}$$

$$\text{Select}(A \rightarrow \epsilon) = \{ \epsilon, a, b \}$$

$$\text{Select}(S \rightarrow \epsilon) = \{ \epsilon, a, b \}$$

Non-terminal	a	b	\$
S	AaAb	BbBa	
A			
B			

BOTTOM-UP PARSING $\Rightarrow S \rightarrow aTde \quad T \rightarrow Tbc/b \quad U \rightarrow d$

Implication $S \rightarrow aTde \Rightarrow aTde \Rightarrow aTbcde \Rightarrow abbcde$

It is an attempt to reduce input string to start symbol of grammar by tracing out RMD of w in reverse.

It is also known as 'shift reduce parsing'.

A string reduced to starting symbol

g- $S \rightarrow AABb$ $A \rightarrow aA/a$ $B \rightarrow bB/b$
input: $aabb \Rightarrow aabb \Rightarrow aabb \Rightarrow aabb \Rightarrow s$

Q. Find the handle of the ~~excess~~ ^{sentential} form occurring in the derivation of string $id + id * id$ by following grammar:
 $E \rightarrow E + E \mid E * E \mid id$

Ans- String can be reduced to E in following steps:

1. $id + id * id$
2. $E + id * id$
3. $E + E * id$
4. $E + E * E$
5. $E + E$
6. E

Q. String: ~~id~~ $(a, (a, a))$, Show handle of each right $S \rightarrow (L) / a$ sentential form:
 $L \rightarrow L, S / S$

1. $(a, (a, a))$
2. $(S, (a, a))$
3. $(S, (S, a))$
4. ~~$(S, (S, a))$~~ $(S, (L, S))$
5. (S, L)
6. (L)
7. S

Type of Shift reduce Parsing:-
Operation - Precedence Parser

- (i) LR-Parser :- lower wide range of grammars
(ii) SLR :- simple LR parser
(b) CLR :- most general LR parser LR(1)



(iii) LALR :- intermediate LR parser
SLR, LR, LALR are same, only their tables are different

Operation precedence parsing :- Simple, but only a small class of grammars.

It follows 2 property:-

- (i) No RHS of any production has a ϵ .
(ii) No two non-terminals are adjacent.

There are 3 operator precedence relations:

- (i) $a > b$ a has higher OP
(ii) $a < b$
(iii) $a = b$

Both end of given input string, add the \$ symbol.

Scan from LR until $>$ is encountered.

Scan towards left over all equal precedence until the first left most $<$ is encountered.

g- $E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow id$

why-

$w = id + id * id$

+	>	<	*	<	id	\$
*	>	<	>	<	-	>
id	>	<	>	<	>	>
\$	<	<	<	<	<	A

$\$ id + id * id \$$

Stack	Relation	Input
\$	<	id + id * id \$
\$ id	>	id + id * id \$
\$ F	<	id + id * id \$

Comment

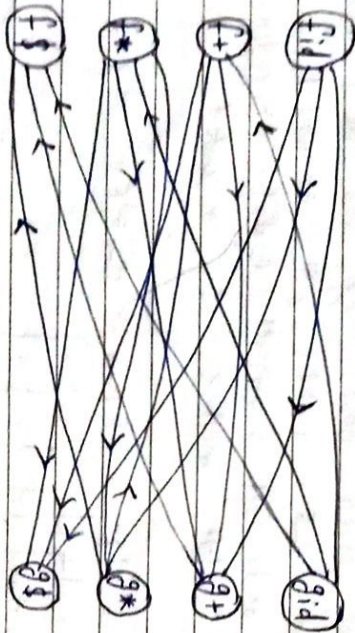
shift id

Reduce $F \rightarrow id$

shift +

\$f_1	<	id id \$	shift id
\$f+id	>	* id \$	Reduce f → id
\$f+f	<	* id \$	shift +
\$f+f*	<	id \$	shift id
\$f+f*id	>	\$	Reduce f → id
\$f+f*f	>	\$	T → F
\$f+T*f	>	\$	T → T * F
\$f+T	>	\$	T → F
\$T+T	>	\$	E → T
\$E+T	>	\$	E → E+T
\$E	A	\$	

Operation function table :- To reduce the size it is used instead of OP.



• $f+id \rightarrow f+ \rightarrow f+ \rightarrow f+$
 • $g+id \rightarrow g+ \rightarrow g+ \rightarrow g+$
 • $f*id \rightarrow f* \rightarrow f* \rightarrow f*$
 • $g*id \rightarrow g* \rightarrow g* \rightarrow g*$

Here we can compare id-id, +, + etc. so this is a

disadvantage.

SLR parser.

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

Q

$E' \rightarrow \cdot E$ - 0 Augmented grammar

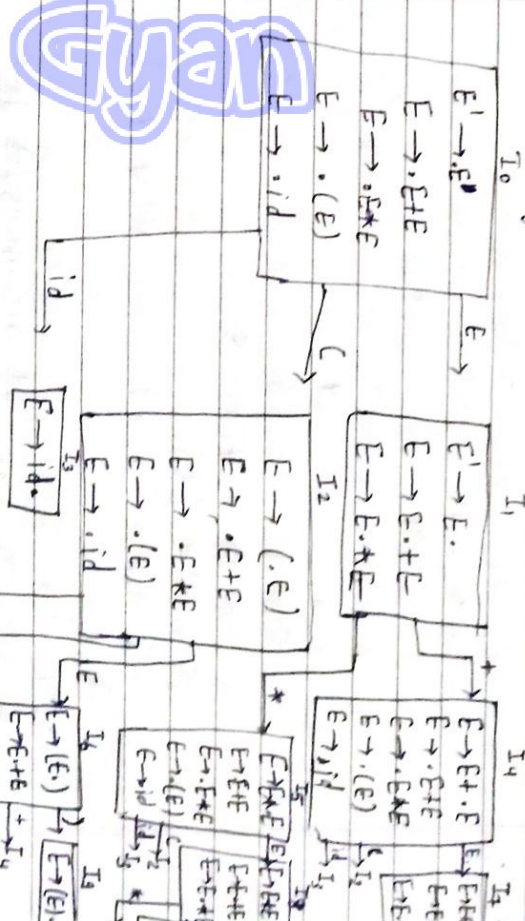
$E \rightarrow \cdot E+E$ - 1

$E \rightarrow \cdot E * E$ - 2

$E \rightarrow \cdot (E)$ - 3

$E \rightarrow \cdot id$ - 4

Transition Diagram



$Final(E) = Final(E+E) \cup Final(E * E)$

$\cup Final(E * E)$

$Follow(E) = \{ \$, +, *,) \}$

State	Action	()	id	\$	goto
I_0	S_4	S_2		S_3	Accept	1
I_1	S_5	S_2		S_3		6
I_2	R_4	S_2		S_3		
I_3	R_4	S_2		S_3		
I_4	R_4	S_2		S_3		
I_5	S_2	S_2		S_3		
I_6	S_4	S_2		S_3		
I_7	S_4	S_2		S_3		
I_8	S_4	S_2		S_3		
I_9	R_3	S_2		S_3		

If $S \mid S \rightarrow LR(0)$
 $\left. \begin{matrix} S \mid R \\ R \mid S \\ R \mid R \end{matrix} \right\} \rightarrow \text{Not LR}(0)$
 Result: - I_7 &
 I_8 has
 $S \mid R$ so
 Not LR(0)

Let grammar contain multiple entries.

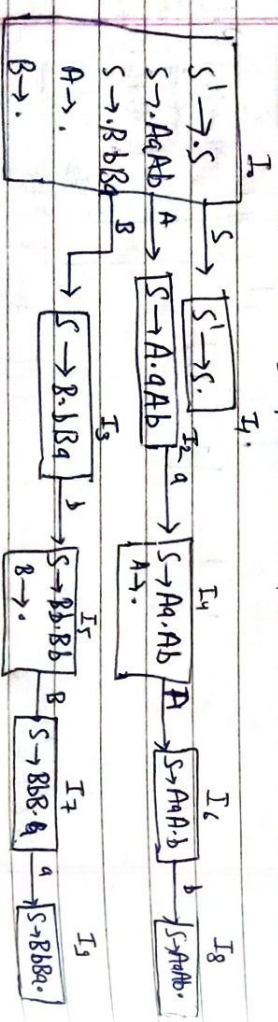
Q. $S \rightarrow AqAb \mid BbBq$ $A \rightarrow \epsilon$ $B \rightarrow \epsilon$

$S' \rightarrow \cdot S$ — 0
 $S \rightarrow \cdot AqAb$ — 1
 $S \rightarrow \cdot BbBq$ — 2
 $A \rightarrow \cdot$ — 3
 $B \rightarrow \cdot$ — 4

$\text{First}(S) = \{a, b\}$
 $\text{First}(A) = \text{First}(B) = \epsilon$

$\text{Follow}(S) = \{a, b\}$
 $\text{Follow}(A) = \{a, b\}$
 $\text{Follow}(B) = \{a, b\}$

Er Sahil Ka Gyan



State	a	Action	
I_0	R_3, R_4		
I_1	R_3, R_4		
I_2	S_4		
I_3	S_5		
I_4	R_3		
I_5	R_4		
		Accept	

I_6		S_8	
I_7	S_9		
I_8			
I_9		R_1	R_2

Thus grammar contains multiple entries. Thus grammar is not LR(0)

LR(1)/CLR parser :-

(i) Augmented grammar (ii) Calculation of first

(iii) Transition Diagram

LR(0), LR (look ahead)

$1) S' \rightarrow \cdot S$

$2) A \rightarrow \cdot AqAb$ $A \rightarrow \epsilon, q$

(iv) LR(1) Parsing table

Q. $S \rightarrow AqAb \mid BbBq$ $A \rightarrow \epsilon$ $B \rightarrow \epsilon$

Solⁿ (i) Augmented grammar

$S' \rightarrow \cdot S$ — 0

$S \rightarrow \cdot AqAb$ — 1

$S \rightarrow \cdot BbBq$ — 2

$A \rightarrow \cdot$ — 3

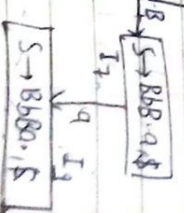
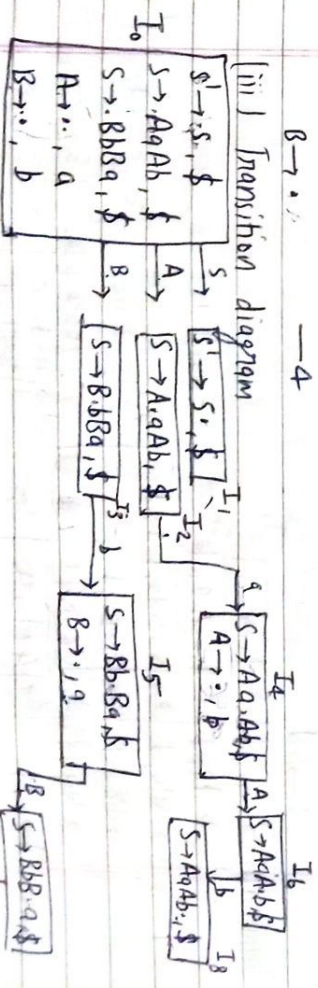
$B \rightarrow \cdot$ — 4

(ii) First

$\text{First}(B) = \text{First}(A) = \{a, b\}$

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

(iii) Transition diagram



Result:- Table doesn't contain multiple entries. Thus grammar is LR(1).

Diagram illustrating the construction of a parse tree for the sentence "The cat sat on the mat" using LR(0) items and transitions.

The root node is I_0 , which contains the LR(0) item $S \rightarrow \cdot A A, \$$. Transitions from I_0 lead to I_1 (on A) and I_2 (on A).

I_1 contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_1 lead to I_3 (on c) and I_4 (on A).

I_2 contains the LR(0) item $S \rightarrow b A c, \$$. Transitions from I_2 lead to I_5 (on A) and I_6 (on A).

I_3 contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_3 lead to I_7 (on A) and I_8 (on A).

I_4 contains the LR(0) item $S \rightarrow \cdot b A c, \$$. Transitions from I_4 lead to I_9 (on b) and I_{10} (on A).

I_5 contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_5 lead to I_{11} (on A) and I_{12} (on A).

I_6 contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_6 lead to I_{13} (on A) and I_{14} (on A).

I_7 contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_7 lead to I_{15} (on A) and I_{16} (on A).

I_8 contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_8 lead to I_{17} (on A) and I_{18} (on A).

I_9 contains the LR(0) item $S \rightarrow b \cdot b A c, \$$. Transitions from I_9 lead to I_{19} (on b) and I_{20} (on A).

I_{10} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{10} lead to I_{21} (on c) and I_{22} (on A).

I_{11} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{11} lead to I_{23} (on A) and I_{24} (on A).

I_{12} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{12} lead to I_{25} (on A) and I_{26} (on A).

I_{13} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{13} lead to I_{27} (on A) and I_{28} (on A).

I_{14} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{14} lead to I_{29} (on A) and I_{30} (on A).

I_{15} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{15} lead to I_{31} (on A) and I_{32} (on A).

I_{16} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{16} lead to I_{33} (on A) and I_{34} (on A).

I_{17} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{17} lead to I_{35} (on A) and I_{36} (on A).

I_{18} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{18} lead to I_{37} (on A) and I_{38} (on A).

I_{19} contains the LR(0) item $S \rightarrow b \cdot b A c, \$$. Transitions from I_{19} lead to I_{39} (on b) and I_{40} (on A).

I_{20} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{20} lead to I_{41} (on c) and I_{42} (on A).

I_{21} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{21} lead to I_{43} (on c) and I_{44} (on A).

I_{22} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{22} lead to I_{45} (on c) and I_{46} (on A).

I_{23} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{23} lead to I_{47} (on A) and I_{48} (on A).

I_{24} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{24} lead to I_{49} (on A) and I_{50} (on A).

I_{25} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{25} lead to I_{51} (on A) and I_{52} (on A).

I_{26} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{26} lead to I_{53} (on A) and I_{54} (on A).

I_{27} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{27} lead to I_{55} (on A) and I_{56} (on A).

I_{28} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{28} lead to I_{57} (on A) and I_{58} (on A).

I_{29} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{29} lead to I_{59} (on A) and I_{60} (on A).

I_{30} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{30} lead to I_{61} (on A) and I_{62} (on A).

I_{31} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{31} lead to I_{63} (on A) and I_{64} (on A).

I_{32} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{32} lead to I_{65} (on A) and I_{66} (on A).

I_{33} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{33} lead to I_{67} (on A) and I_{68} (on A).

I_{34} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{34} lead to I_{69} (on A) and I_{70} (on A).

I_{35} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{35} lead to I_{71} (on A) and I_{72} (on A).

I_{36} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{36} lead to I_{73} (on A) and I_{74} (on A).

I_{37} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{37} lead to I_{75} (on A) and I_{76} (on A).

I_{38} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{38} lead to I_{77} (on A) and I_{78} (on A).

I_{39} contains the LR(0) item $S \rightarrow b \cdot b A c, \$$. Transitions from I_{39} lead to I_{79} (on b) and I_{80} (on A).

I_{40} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{40} lead to I_{81} (on c) and I_{82} (on A).

I_{41} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{41} lead to I_{83} (on c) and I_{84} (on A).

I_{42} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{42} lead to I_{85} (on c) and I_{86} (on A).

I_{43} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{43} lead to I_{87} (on c) and I_{88} (on A).

I_{44} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{44} lead to I_{89} (on c) and I_{90} (on A).

I_{45} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{45} lead to I_{91} (on c) and I_{92} (on A).

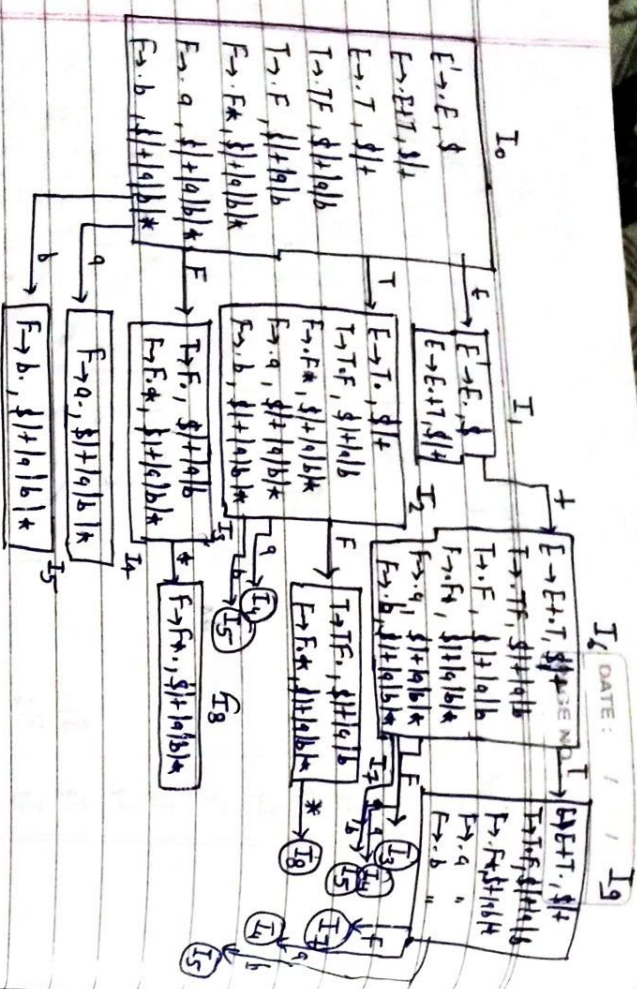
I_{46} contains the LR(0) item $S \rightarrow b A \cdot c, \$$. Transitions from I_{46} lead to I_{93} (on c) and I_{94} (on A).

I_{47} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{47} lead to I_{95} (on A) and I_{96} (on A).

I_{48} contains the LR(0) item $S \rightarrow b A c \cdot, \$$. Transitions from I_{48} lead to I_{97} (

Result:- Table doesn't contain multiple entries. Thus programmer is LR(1).

~~E⁻, E₊, S
E⁻, L⁺, I, S, H
L⁺, I, S, H
~~I⁻, I⁺, S, H, A, B~~
~~F⁻, F⁺~~
~~S, B~~~~



State	+	*	Action		E	GoTo	
			a	b		T	F
I ₀			S ₄	S ₅		1	3
I ₁	S ₆				Accept		7
I ₂	R ₂		S ₄	S ₅			
I ₃	R ₄	S ₈	R ₄	R ₄			
I ₄	R ₆	R ₆	R ₆	R ₆			
I ₅	R ₇	R ₇	R ₇	R ₇			
I ₆			S ₄	S ₅		9	3
I ₇	R ₃	S ₈	R ₃	R ₃			
I ₈	R ₅	R ₅	R ₅	R ₅			
I ₉	R ₁		S ₄	S ₅			7

Result:- Table does not contain multiple entries. Thus given grammar is LR(1)

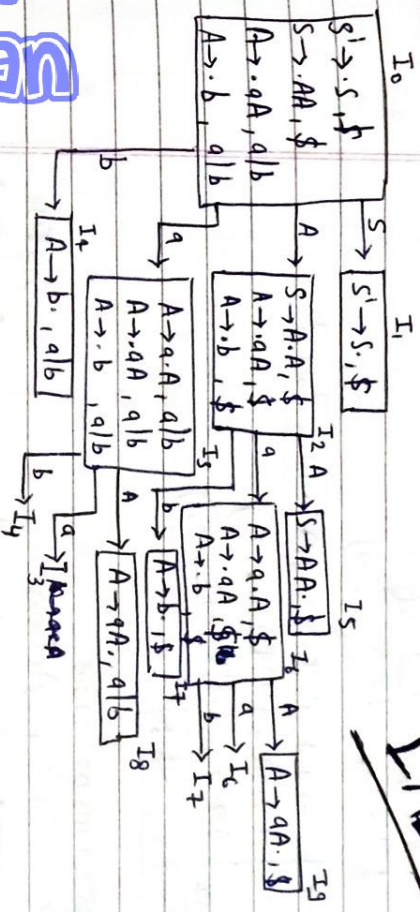
LALR [Look ahead left to right] Parser :-

- (i) Augmented gr.
- (ii) Calc. of first (iii) Item: (iv) LR(1) table
- (v) Find states having same production & merge both states
- (vi) Transition dia. (vii) LALR parsing table

Er Sahil Ka Gyan

Eg- $S \rightarrow AA$ $A \rightarrow aA/b$
 At- $S \rightarrow S$ -0 $\text{first}(A) = \{a, b\}$
 $S \rightarrow AA$ -1 $\text{first}(S) = \{a, b\}$
 $A \rightarrow aA$ -2
 $A \rightarrow \cdot b$ -3

LALR



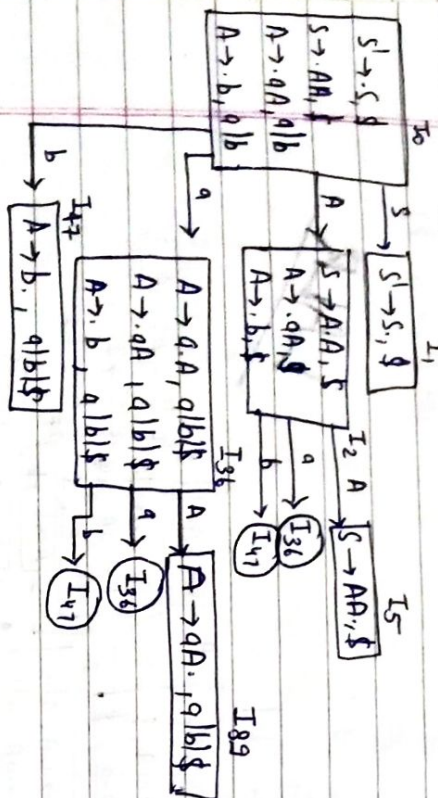
State	a	b	Action	E	S	A	GoTo
I ₀			S ₃			2	
I ₁			S ₄				
I ₂	S ₆	S ₇		Accept		5	
I ₃	S ₃	S ₄				8	
I ₄	R ₃	R ₃					
I ₅	S ₆	S ₇					
I ₆							
I ₇							
I ₈	R ₂	R ₂					
I ₉							

Step-2: Design of LALR

I₃₆ = $A \rightarrow aA, aAb/b$
 $A \rightarrow aA, aAb/b$
 $A \rightarrow \cdot b, aAb/b$

Key: Table does not contain multiple entries So it is LR(1).

$I_{47} = A \rightarrow b, a|b| \$$
 $I_{89} = A \rightarrow aA, a|b| \$$



LR(0) Parsing table

State	Action		GOTO	
	a	b	\$	A
I_0	S_{36}	S_{47}		1
I_1				2
I_2	S_{36}	S_{47}		5
I_3	S_{36}	S_{47}		89
I_4	R_3	R_3		
I_5	R_4	R_3		
I_6	R_2	R_2		
I_7				
I_8				
I_9				

Table does not contain multiple entries. Thus grammar is LR(0).

Syntax Directed Translation:-

Grammar + Semantic rule = SDT

Informal notations

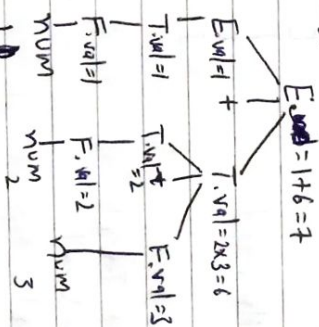
* In SDT, Every non-terminals can get 0 or more attributes depending on the type of attribute.
 * In Semantic rule, attributes are called string, no, memory location.... It is represented by VAL

Eg-

Production	Semantic Rule
$E \rightarrow E+T$	$E.VAL = E.VAL + T.VAL$
$E \rightarrow T$	$E.VAL = T.VAL$
$T \rightarrow T * F$	$T.VAL = T.VAL * F.VAL$
$T \rightarrow F$	$T.VAL = F.VAL$
$F \rightarrow \text{Num}$	$F.VAL = \text{num.lexval}$

* STD is implemented by constructing a parse tree and performing the action in L to R depth first order.

Eg- $1+2*3$



Top-down

Synthesized attribute

Inherited attribute

Here A is S.A. coz it is depended on B, C & D. (LHS of production rule.)

$A \rightarrow BCD$	RHS of production rule of non-terminals $C.S = A.S, C.S = B.S, C.S = D.S$
---------------------	--

Syntax Directed Definition :-

It is a generalization of a context free grammar in which each grammar symbol has an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol.

Grammar + Semantic Rule \Rightarrow SDD

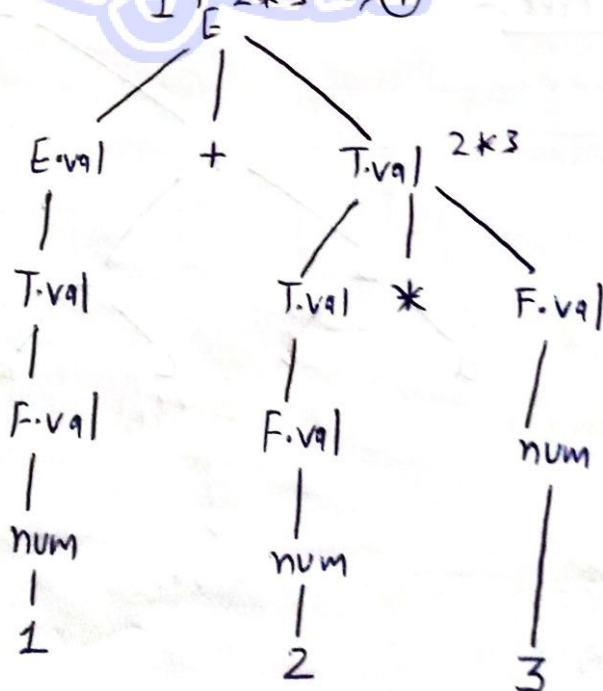
→ In semantic rule, attributes are called string, no., type, memory location ---. It is represented by VAL.

Eg -

	<u>Semantic Rule</u>
$E \rightarrow E + T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = num.lexval$ num (0-9)

→ SDD is implemented by construction a parse tree and performing the action in Left to Right depth first order.

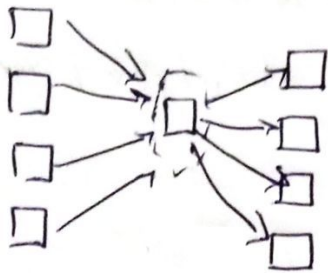
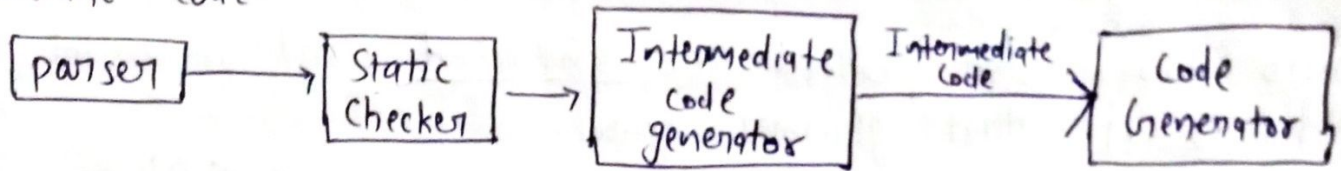
Eg - $1 + 2 * 3$ $1 + 2 * 3 \Rightarrow 7$



SDD

Intermediate Code Representation \Rightarrow

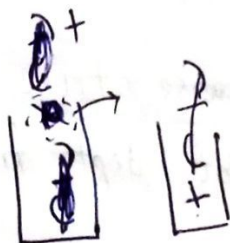
Intermediate code is used to translate the source code into the machine code.



Er Sahil

- (i) Postfix Notation (ii) Syntax tree (iii) Three address code

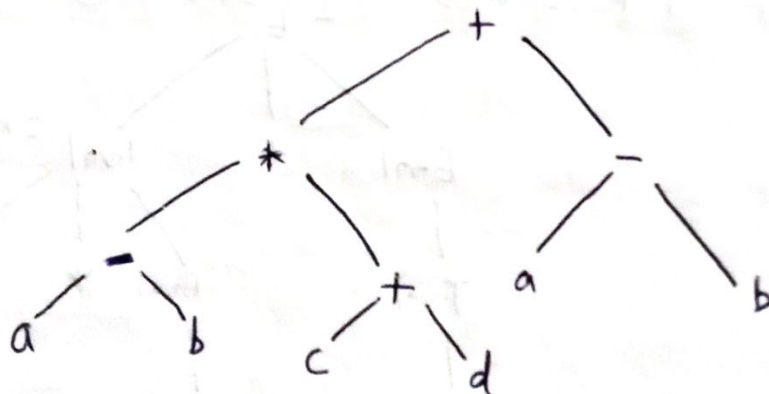
(i) Postfix Notation:-



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

$ab - cd + * ab - +$

(ii) Syntax tree:-



Three Address Code:- It is a type of intermediate code which is easy to generate & can be easily converted to machine code.

→ TAC instruction has at most 3 operands.

$$a = b \text{ op } c$$

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

Er Sahil
Ka
Gyan

TAC:-

$$\text{Let } t_1 = a + b$$

$$\text{Let } t_2 = c + d$$

$$\text{Let } t_3 = t_1 + c$$

$$\text{Let } t_4 = t_1 * t_2$$

$$\text{Let } t_5 = t_4 + t_3 \quad [\text{The TAC desired}]$$

There are 3 representations of TAC namely.

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple:- It is structure with consist of 4 fields namely op, arg1, arg2 & result.

2. Triples:- This representation doesn't make use of extra temporary variable to represent a single operation.

3. Indirect Triples:- This representation makes use of pointer to listing of all references to computations which is made separately & stored.

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)

#	op	arg1	arg2
(14)	+	a	b
(15)	+	c	d
(16)	*	(14)	(15)
(17)	+	(14)	c
(18)	+	(16)	(17)

In SDD, two attributes are used one is Synthesized & another is inherited attribute.

Synthesized attribute:- If parse tree node value is determined by the attribute value at child nodes.

- The production must have non-terminal as its head.
- Synthesized attribute is used by both S-attributed SDT & L-attributed STD.

Eg-

$E.val \rightarrow F.val$

$E \rightarrow T \rightarrow F \rightarrow num \rightarrow 1$

$E.val$



$F.val$

Er Sahil
Ka
Gyan

Inherited attribute:- An attribute is said to be IA if its parse tree node value is determined by the attribute value at parent and/or siblings node.

- Inherited attribute is used by only L-attributed SDT.

Eg-

$E.val = F.val$

$E.val$



$F.val$

S-attributed SDT:-

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- Semantic actions are placed in rightmost place of RHS.

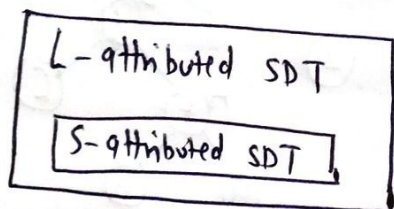
Eg - $L \rightarrow E \wedge E \Rightarrow L.val = E.val \wedge E.val$
 $F \rightarrow digit \Rightarrow F.val = digit.lexval$

L-attributed SDT:- If an attribute of an SDT is synthesized or inherited with some restriction on inherited, it can inherit values from left siblings only.

- Semantic actions are placed anywhere in RHS.

Eg - $X \rightarrow ABC$

$B.P = X.P$, $B.P = A.P$, $B.P = C.P$ X



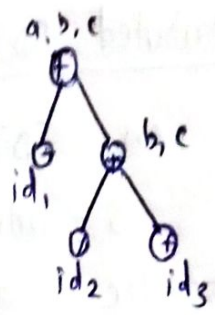
Eg - $P_1: S \rightarrow MN \quad \{ S.val = M.val + N.val \} \checkmark$
 $P_2: M \rightarrow PQ \quad \{ M.val = P.val * Q.val \ \& \ P.val = Q.val \} X$

DAG :- The directed Acyclic Graph is used to represent the structure of basic blocks, to visualize the flow of values b/w basic blocks & to provide optimization techniques in the basic blocks.

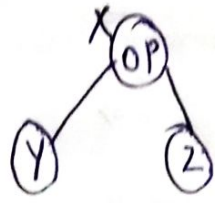
DAG is a 3-address code that is generated as the result of an Intermediate Code Generation (ICG).

- leaves have a unique identifier
- Interior nodes are labelled with operator symbol.
- Nodes are given a string of identifiers to use as labels for storing computed value.

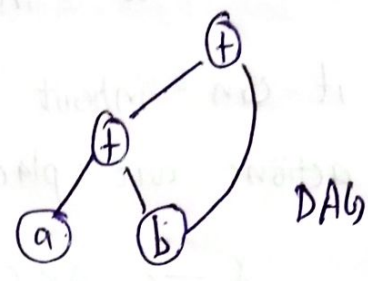
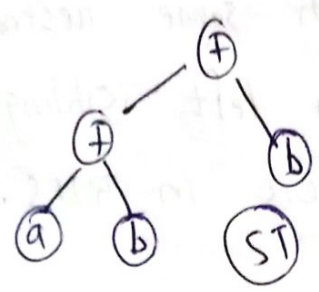
Characteristics of DAG



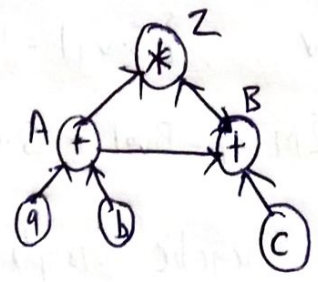
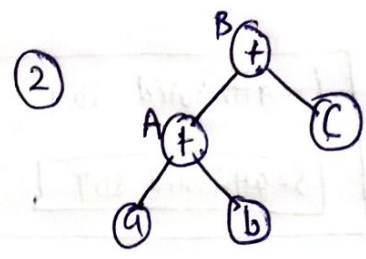
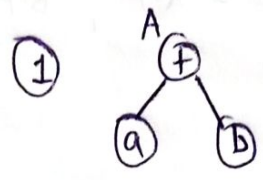
- case-(i) $x = y \text{ op } z$
- case-(ii) $x = \text{op } y$
- case-(iii) $x = y$



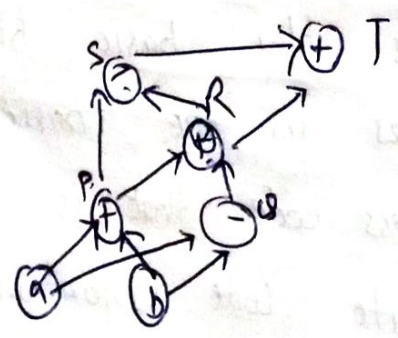
Eg - $(a + b) + b$



- Eg -
- $A = a + b$
 - $B = A + c$
 - $Z = A * B$



- Eg -
- $P = a + b$
 - $Q = a - b$
 - $R = P * Q$
 - $S = P - R$
 - $T = S + R$



Ex -

$$a = b * c$$

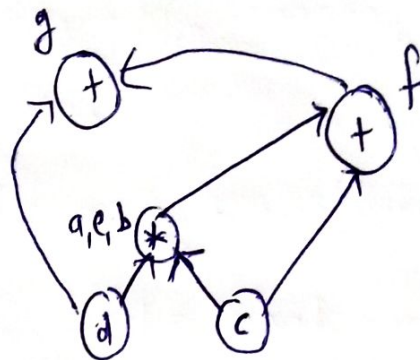
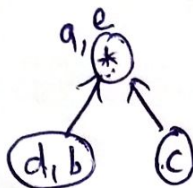
$$d = b$$

$$e = d * c$$

$$b = e$$

$$f = b + c$$

$$g = f + d$$



Eg

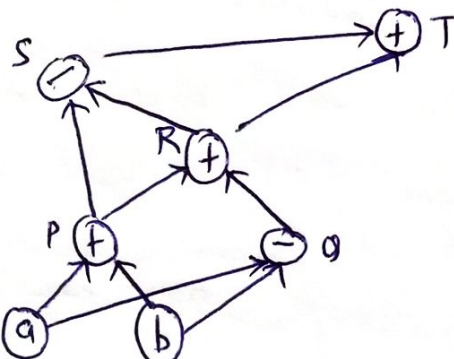
$$P = a + b$$

$$Q = a - b$$

$$R = P + Q$$

$$S = P - R$$

$$T = S + R$$

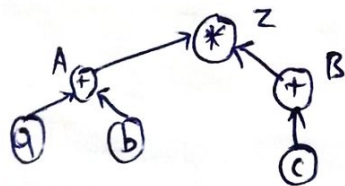


Eg

$$A = a + b$$

$$B = A + c$$

$$Z = A * B$$



Eg

$$a = b * c$$

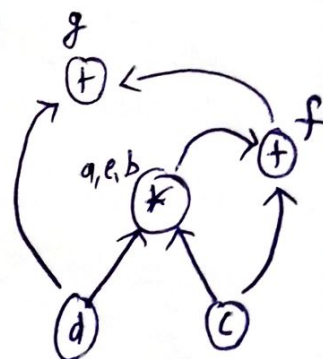
$$d = b$$

$$e = d * c$$

$$b = e$$

$$f = b + c$$

$$g = f + d$$



Q.1 Explain the key issues in runtime organization

Ans - Issues in runtime organization are:

Activation trees → It is used to depict the way control enters & leaves activation.

In an activation tree,

- (i) Each node represents an activation of a procedure.
- (ii) The root represents the activation of program.
- (iii) The node for a is parent of node for b if & only if the lifetime of a occurs before the lifetime of b .

Control stack:- It is used to keep track of line procedure activation the idea is to push the node for an activation on control stack as the activation begins & to pop the node when activation ends. The content of control stack are selected to paths to root of activation tree. When node n is at the top of control stack, the stack contains the nodes along path from n to the root.

Q.2. What are the various parameter passing methods?

Ans- (i) Call by value:- The simplest possible method of passing parameters.

- The caller computes n-values for actuals.
- The caller places the resulting values on stack, in the AR of caller.
- The caller may change parameters, but this has no effect on caller.
- This is difficult protocol in Pascal & the only protocol is C.

(ii) Call by Reference:- When parameters are passed by reference.

- The caller passes the called procedure's pointer to storage address at actual parameter.
- If actual has an I-value, it is used.
- C++ uses call by reference if "&" operator is specified.

Example: Consider a swap example:

$x = a$

$y = b$

$temp = x$

$x = y$

$y = temp$

1. Program reference (input, output);

2. VAR a, b: integer

3. Procedure swap (var x, y: integer);

4. Var temp: integer; 5. begin
6. temp = x; 7. x = y; 8. y = temp;
9. end;

10. begin 11. a = 1; b = 2

12. swap(a, b);

13. write In ('a = ', a) write In ('b = ', b)

14. end

(iii) Copy restore:- This is hybrid b/w CbV & CbR.

- Refere caller is activated, we evaluate the actuals & put their n-values in AR for caller.
- But we also compute & save the I-values of actuals.

In return sequence, we copy the updated n-values from caller's AR to location for saved values. FORTRAN used this approach.

(iv) Call by Name [Macro Expansion]:- In this method

the body of procedure for procedure call. we just substitute

In copied body, the formal parameters are replaced by text of actuals.

ex - call swap(i, a[i])

temp = i; 1 = a[i]; a[i] = temp

The under CBN, swap set it to a[i] as expected but unexpected result of setting a[i] to i rather than a[i] to i.

Q.3 Explain the term nesting depth by taking suitable example.

Ans- Nesting depth:- It is used to implement lexical scope. It can be calculated as follows -

- i Nesting depth of main program is 1.
- ii Add 1 to depth when a new procedure begins.
- iii Subtract 1 from depth each time when you exit.
- iv The variable declared in specific procedure is associated with nesting depth.

For example, all digits match their nesting depth
 $0((22)1), ((33)1(22), ((44)))$
, $((22))((22))((1)$

The first 3 strings have min length among those that have the same digits in same order but last are not since $((22)1)$ also has digits 221 and is shorter.

eg - input: seq = "((11))"
output [0, 1, 1, 1, 1, 0]

Q4 How can we access local & non local names at block structured?

- Ans - local data can be accessed with the help of activation record.
- Offset relative to base pointer of an

- activation record points to local data variables within an record.
- Reference to any variable x is procedure is

$x = \text{Base pointer} + \text{offset of variables}$

Access to nonlocal names:-

A procedure may sometimes refer to variables which are not local to it. Such variables are called as non-local variables.

Two scoping rules for accessing non-local data

① Lexical or static Scoping:-

→ PASCAL, C & FORTRAN

→ The correct address of a non-local name can be determined at compile time by checking syntax.

② Dynamic Scoping:-

→ LISP

→ A use of nonlocal variable corresponds to declaration is the most recently called "still active" procedure.

It can only be determined at run time.

Q.5

Explain all factors of symbol table which are considered during evaluation of data structures.

Ans- Symbol table typically need to support multiple declaration of same identifier within a program.
Symbol table is an important data structure created & maintained by compiler in order to store information about the occurrence of various entities such as variable names, function names, object, classes, interfaces etc.
Symbol table is used by both analysis & the synthesis parts of a compiler.
A symbol table may serve the following purposes depending upon the lang. in hand:

(i) To store the names of all entities in structural form at one place.

(ii) To verify if a variable has been declared.

(iii) To implement type of checking, by verifying assignment & expression in the source code are semantically correct.

(iv) To determine the scope of a name (scope resolution)

→ A compiler contains two types of symbol table
• Global symbol table
• Scope symbol table

Q.1 Explain in detail the various issues of a design of a code generator.

Ans - Code generator converts the intermediate representation of source code into a form that can be rapidly executed by a machine.

Following issues arises:

(i) ILP to code generator:- The ilp of code generator is intermediate code generated by the front end.

→ Several choices for intermediate language:

- | | | |
|-----------------|---|--------------------|
| Linear | - | Postfix notation |
| 3 address | - | Quadruples |
| Virtual Machine | - | Stack machine code |
| Graphical | - | Syntax tree & dags |

(ii) Memory Management:- Mapping names in source program to address of data objects in run time memory.

- Done by front end & the code generator.
- A name in 3 add. stat. refers to a S.T. entry for name.
- A relative address can be determined.

(iii) Target Program:- The TP is o/p of code generation. The o/p may be absolute machine language, relocatable machine lang., assembly lang.

(iv) Instruction Selection:- Selecting the best instruction will improve the efficiency of program. It includes ins. that should be complete & uniform.

(v) Register Allocation Issues:- Use of registers makes the computations faster in comparison to that of memory, so efficient utilization of registers is important.

(vi) Evaluation Order:- The code generator decides the order in which ins. will be executed. The order of computations affects efficiency of the target code.

Q.2 "Code Optimization is an optional phase of compilation process" Explain.

Ans:- It is used to improve the intermediate code so that o/p of prog. could run faster & take less space. It removes unnecessary lines of code & averages the sequence of state. In order to speed up the program execution.

This phase is called optional phase because if the inter. rep. is good enough to develop the object code, then the compiler will skip the process of optimizing the intermediate representation. Optimizations may be turned off to speed up the translation.

Q.3 Explain the following terms:

(i) Common Sub expression elimination [CSE]:- CSE is compiler optimization that searches for instances of identical expressions and analyzes whether it is worth while replacing them with a single variable holding the computed values.

(ii) Variable propagation:- It is process of replacing direct assignments with their values. A direct assignment is an instruction of form $x=y$, which simply assigns the value of y to x .
Ex: // Before optimization
$$c = a * b \quad x = a \quad \text{till } d = x * b + 4$$

// After optimization

$$c = a * b \quad x = a \quad \text{till } d = a * b + 4$$

(iii) Loop invariant Computation:- A fragment of code that resides in loop &

computer basic value or each iteration is called loop-variant code. This code can be moved out of loop by saying it to be computed only once, rather than with each iteration.

(4.) Strength Reduction: - There are expressions that consumes more

CPU cycles time & memory. These expressions should be replaced with cheaper expressions without compromising the o/p of expressions.
Ex - $(x * 2)$ is expensive in terms of CPU cycles than $(x < 1)$ and fields the same results.

(5) Dead Code Elimination: - It is also known

as DCE, dead code removal, dead code stripping or dead code strip.

It is compiler optimization to remove code which does not affect the program result. It can also enable further optimizations by simplifying program structure.

Q4

Ans- How can we generate code from DAG?
The advantage of generating code for a basic block from its DAG representation

is that from DAG we can easily see how to rearrange the order of the final computation sequence then we can start from a linear sequence of 3 address statements or quadruples

To prepare the list of DAG nodes to compute, start at root of the right most sub tree put this node as the list, L, node to left and continue by adding a left most node to list after all its parents are already on the list then generate code for the nodes in L by starting at end of L & proceeding to the beginning.

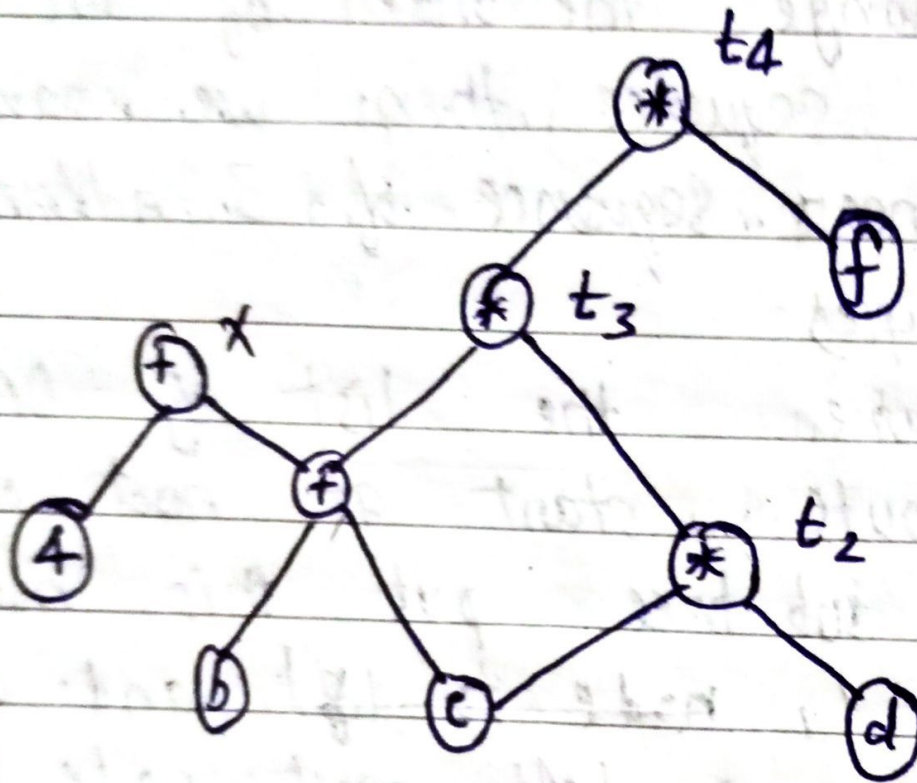
Q5

Construct DAG for basic block whose code is given below:

$t_1 = b + c$
 $t_2 = d * c$
 $t_3 = t_2 * t_1$
 $t_4 = t_3 * f$
 $x = t_1 + t_4$

Er Sahil Ka Gyan

DAG Representation -



Playlist is available on my channel

Er Sahil
Ka
Gyan