driver

Q. What is a compiler and why we need it?

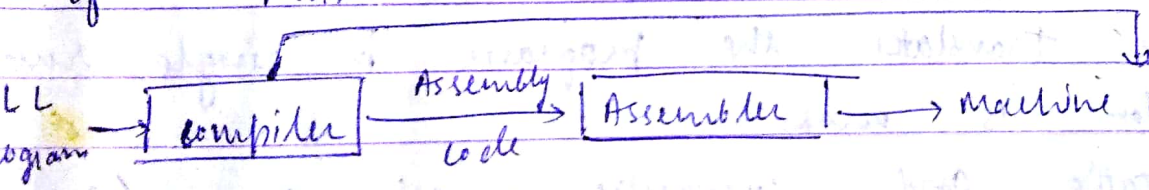
Languages like C, C++, java etc are HLL i.e. languages written in abstract Human readable and understandable form. But a computer understands only machine code i.e. language in terms of 0's & 1's.

Compiler:

It is a program or a s/w whose input is ~~it~~ high level program and o/p can be an assembly level program or it directly o/p's the machine code which after providing input to the computer provides you the result.

Assembly level language:-

Every computer has a processor & it works on specific set of instructions & a programs or language that is written using such set of instruction is called Assembly level language.

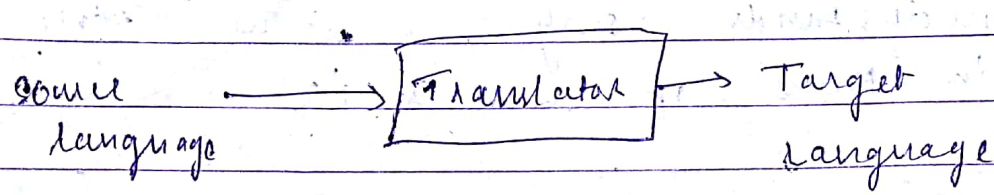


easier ← In terms of programing → harder

slower ← In terms of execution → faster

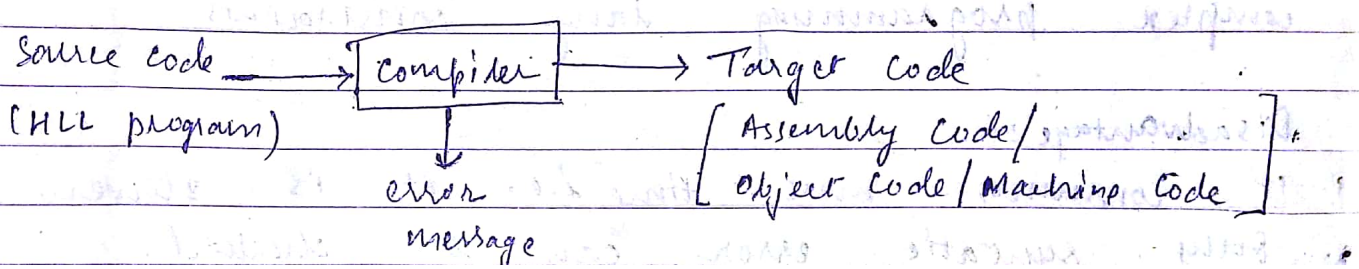
Translators :-

A translator is defined as a s/w program that takes as i/p a program written in some language & produces o/p a program in another language



There are diff type of Translators :

1. Compiler
2. Interpreter
3. Assembler
4. Macro Assembler



substitute for compilers.

Eg. ~~BA~~ Interpreter of BASIC language

Advantage :-

1. An interpreter translates a program line by line
2. Interpreter are smaller in size as compared to compilers
3. Error localisation is easier.
4. An interpreter facilitates the implementation of complex programming lang. constraints

Disadvantage :-

1. It consumes more time i.e. it is slower.
2. Only syntatic error can be checked.
3. CPU utilisation is less
4. Interpreters are less efficient as compared to compilers.

3. Assembler :-

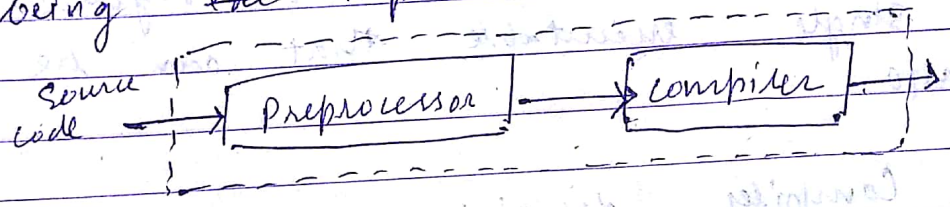
It is a translator that translates an assembly level lang. prog. into a low level machine program which can then executed directly.

Assembly level lang. prog. → [Assembler] → low level machine prog

Q: Preprocessor :-

1. Preprocessor are also part of compiler but it recognizes the statement to be processed first by means of certain directives & its meaning is substituted in the source program before it goes to final compilation.

2. This enables the statements of the source program to be processed before it is being inputted to the main compiler.



6. Loader :- Loader is system program that loads the binary code in the m/o ready for execution.

Eg. Loaders are responsible for locating program in the main m/o every time it is being executed.

7. Linker :- Linking is the process of combining various pieces of code and data together to form a single executable that can be loaded in m/o.

* Challenges in Compiler designing :-

(a) Many variations :

(i) Many programming languages (C, C++, Java, Pascal)

(ii) Many programming parameters (Object oriented, functional, logical)

(iii) Many computer architectures (Intel, alpha, MIPS)

(iv) Many operating system (Solaris, windows, Unix)

(b) Qualities of a Good Compiler :

(i) Compiler itself must be bug free

(ii) It must generate correct machine code

(iii) The generated machine code must run fast

(iv) The compiler itself run fast i.e. the

compilation time is directly proportional to program size.

(d) it must print good diagnostic and error messages.

(e) Building a compiler requires knowledge of :-

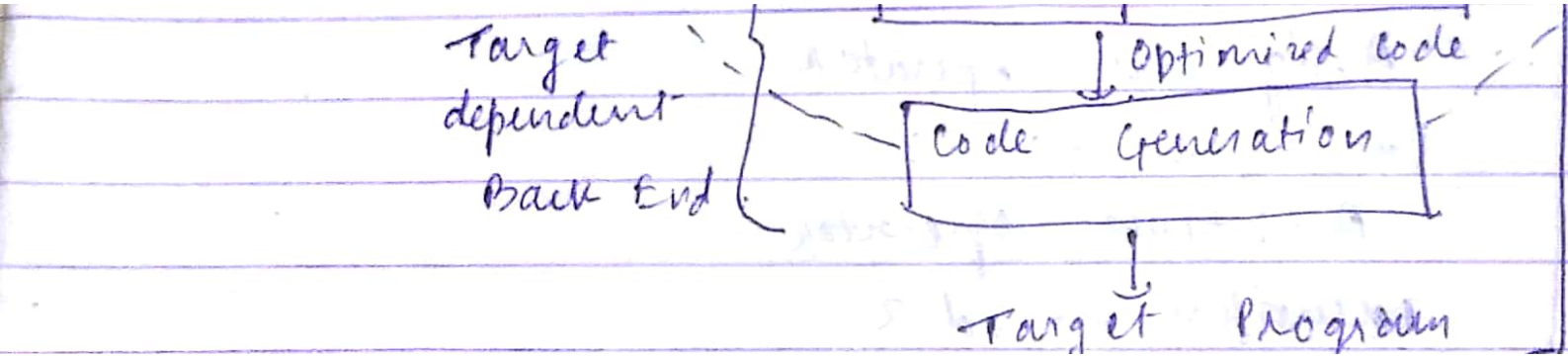
1. Programming languages
2. Computer architecture
3. Theory of automata
4. Algorithm and data structure
5. Software Engineering

** Difference between :-

Compiler	Interpreter
1. A compiler translates complete source program in a single run.	1. An interpreter translates source prog. line by line.
2. Error localisation is difficult in compiler	2. Error localisation is easier than compiler.

Compiler	Interpreter
5. Compilers are larger in size	Interpreters are smaller in size as comp. to compiler.
6. Intermediate object code is generated.	6. Due to alternate process bet translation & execution, no object code is produced.
7. m/c req. is more since object code is generated.	7. m/c req. is less
8. In compiler, it is not necessary to compile a prog. again & again to execute it.	8. In interpreter, it is necessary because no object code is produced and every time high level prog. is converted into low level prog.
9. Errors are displayed after entire prog. is checked	9. Errors are displayed for every inst. interpreted.
10. Compilers are more efficient	10. They are less efficient.
11. It consumes less time.	11. It consumes more time than compiler
12. Compiler is faster	12. It is slower
13. Eg C compiler	13. Eg Interpreter of BASIC lang.

Scanned by CamScanner



instance a keyword, identifier or symbol names

→ The token language is typically a regular lang so a finite state automata constructed from regular expression can be used to recognize it. This phase is also called lexing or Scanning.

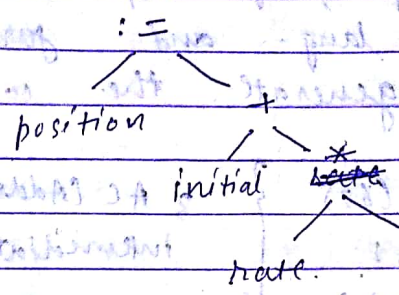
eg. position := initial + rate * 60 ;

Tokens	Token Name
position	identifier → id1
:=	Assignment operator
initial	identifier → id2
+	Addition operator
rate	identifier → id3
*	Multiplication operator
60	constant (number)
;	delimitor

3. (a) Compiler Front End :-

(1) Lexical Analysis

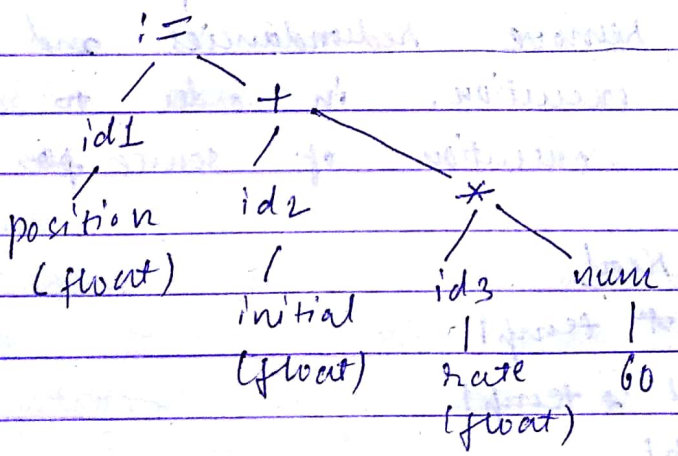
Eg: position := initial + rate * 60.



Parse tree

(3) Semantic Analysis :- This phase is used to recognise the meaning of program code. and start to prepare for o/p. In this phase ~~three~~ type checking is done and most of compiler error are checked.

Eg.



Annotated Parse Tree.

4) Intermediate Code Generation :- This phase uses syntactic structure to create stream of simple instructions. These can be in the form of

Simple statements or some of the intermediate forms like postfix, triples, quadruples

Depending upon the lang. and for which machine it will generate the code.

Eg.

```
temp 1 = int to real (60)
temp 2 = rate * temp 1
temp 3 = initial + temp 2
position = temp 3
```

} 3 AC (Address code) intermediate form.

(D) Compiler Back End :-

(1). Code optimization Phase :- It is optional phase of compiler which is executed only if the intermediate code needs to be optimized. Its purpose is to remove redundancies and reduce the time of execution. in order to increase the efficiency of execution of source program.

Eg.

```
temp 1 = int to real
temp 1 = rate * temp 1
temp 1 = initial + temp 1
position = temp 1
```

Intermediate lang. is translated into the o/p lang which is usually the native machine lang of the system.

Eg.
```
MOV  R0, 60
MOV  R1, id3
MUL  R0, R1
MOV  R1, id2
ADD  R0, R1
STORE M[id1], R0
```
} Assembly code.

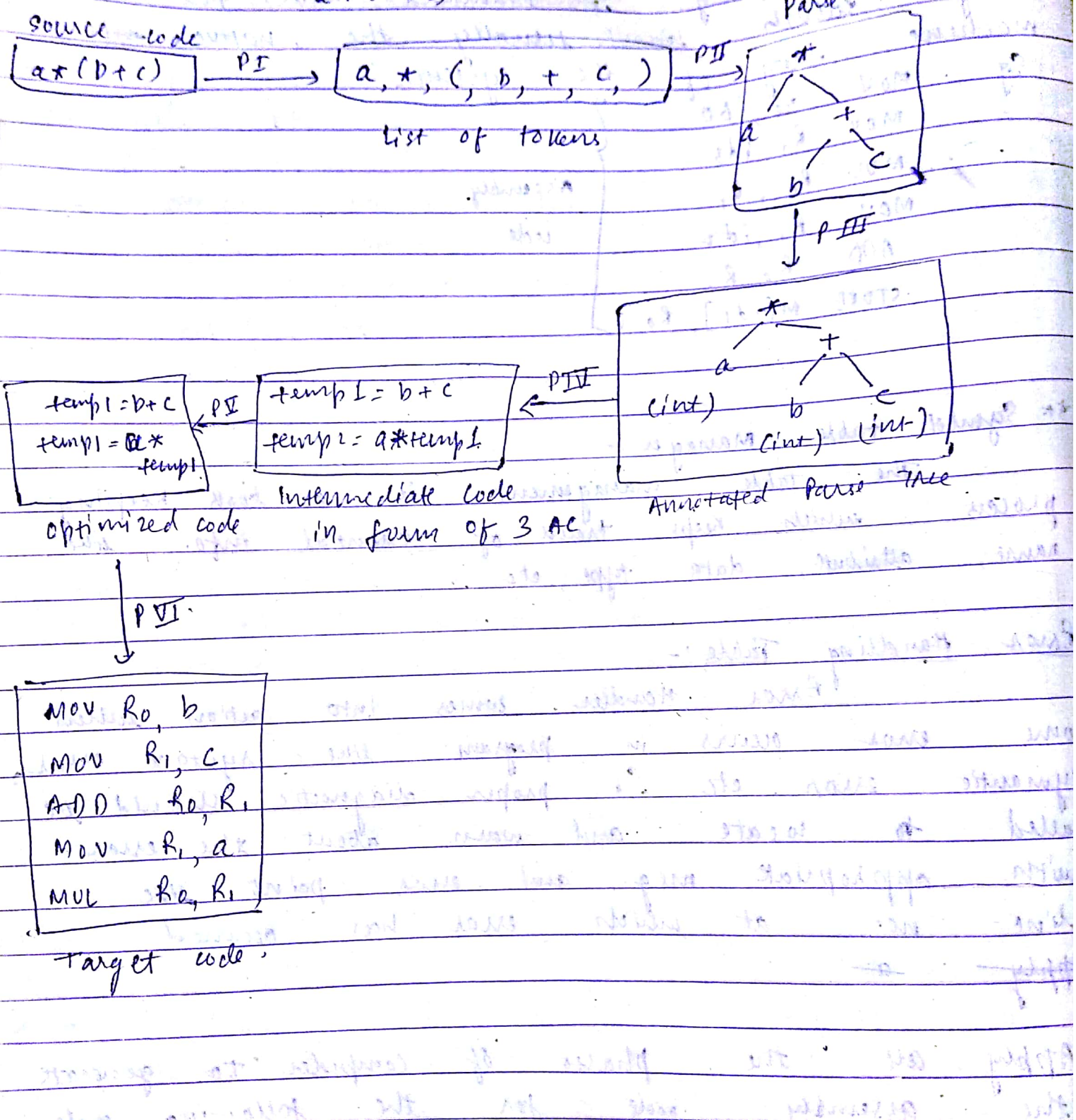
** Symbol table Manager :-

The table management is a book keeping process which keeps track of essential info. like name, attribute, data type, etc.

* Error Handling Table :-

Error Handler comes into action when some error occurs in program like syntax error, symantic error, etc. : proper diagnostic should be called to locate and warn about the error. with appropriate msg and error point like line no. at which error has occured.

~~Apply~~ ~~a~~



Lexical Analysis Phases:-

Lexical means anything related to words. The main task of lexical analyser in compiler design are:-

1. It process the input character that constitute a high level program into valid set of tokens.
2. It skips the comment and white spaces while creating these tokens.
3. If any input which contains error is provided by the user in the program lexical analyser along with error handling table correlate that error with source file and line number.

Terminologies used in Lexical Analysis Phase:-

1. Token:- A set of I/P stream which are related through a similar pattern is called token.

2. Lexeme:- Lexeme is the actual I/P stream which represents the token

eg. X = X * (acc + 123)

Token	Lexeme
Identifier	X
Assignment op.	=
Identifier	X
,	,
,	,
,	,
,	,

Syntax checker in a compiler serves as the master program.
It first sends a request to get a valid token from the lexical analyzer and scanner than do its pattern matching to create a valid token if possible and sends back to syntax checker.

Eg. of tokens :-

1. Operators +, =, >, <, >=, <=, !=

2. Keywords if, for, while, ~~do~~ int

3. Numeric literals 43, 6.05, -3.6e10, 0x13FA

4. Character literal 'a', '1', 'n'

5. String literals "Hello", "abc", "\n"

Non-tokens :-

white space Space (' '), Tab ('\t')

Comments //, /* */

Q If (MAX = 5) then GOTO 100

Construct symbol table & find the tokens for the above code.

Syntax analysis phase is able to construct a proper parse tree otherwise with the help of error handler, it declares syntax error.

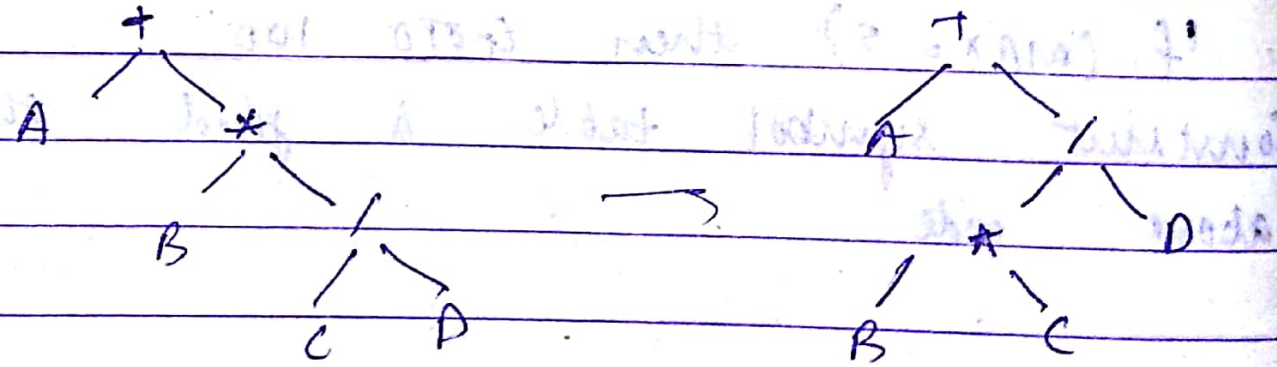
1.

Eg. A + * B / C.

2.

3

Eg. Consider expression A + B * C / D

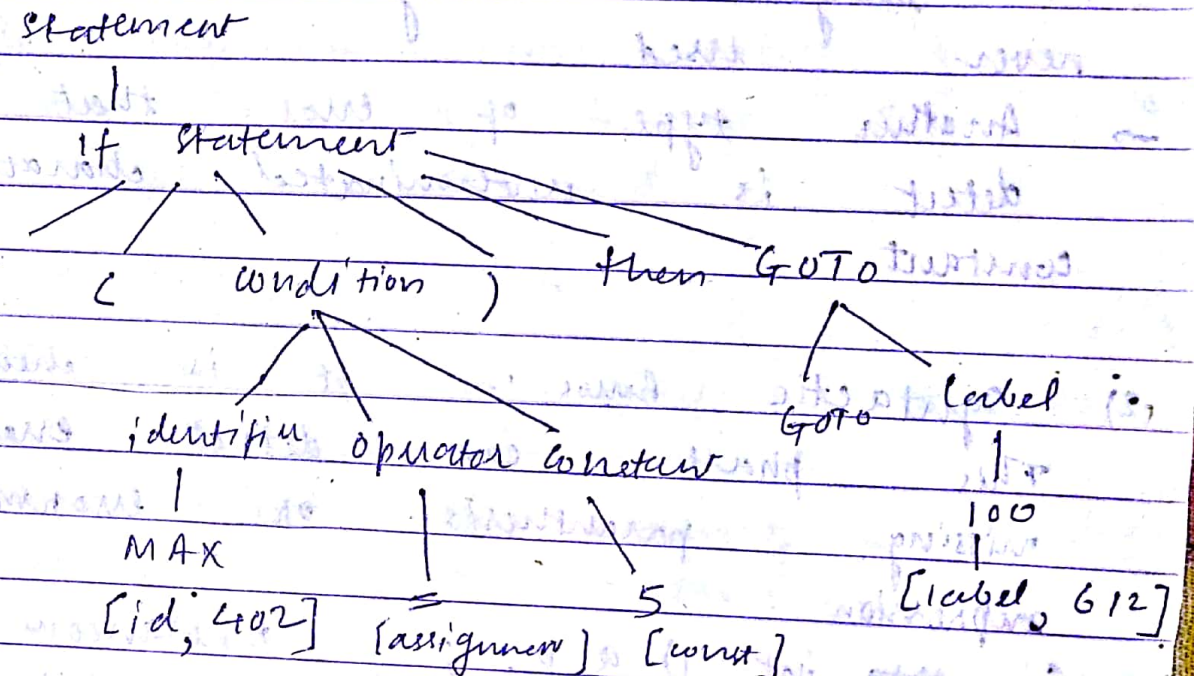


We have seen 2 parse tree for the given statement & the compiler select the parse tree based on the precedence

rule defined by grammar of compiler for a language.

Q Construct the parse tree for the given statement.

if (MAX = 5.) then GOTO 100



Structure of Compiler

* Error Handling Table

level of compilation are (1) lexical or scanner errors
Such type of errors consist of illegal & or unrecognised character mainly caused by typing errors.
A common way for this to happen is:
for the programmer to type a character i.e. illegal in any instance of lang. & is never used.
→ Another type of error that scanner may detect is unterminated character or string constant.

(2) Syntactic Errors :- It is checked by parser. This phase can detect errors such as missing parantheses or erroneous arithmetic expression.
Eg. int 1) a, b, ; Extraneous comma
2) (a * b + c) d - ; Variable missing
3) void mains() → misspelled keyword

3) * Symantic Error :- Symantic errors may occur in a program related to the fact that some statements may be correct from the syntactical point of view, but they make no sense & there is no code that can be generated to carry out the meaning of the statement

* Eg undeclared name & type incompatibilities are sources of symantic error.

(4) Code optimization error : This type of error may encounter during code optimization. For eg. In control flow analysis, there may exist some statements which can never be reached.

(5) Code generation error :- It may occur during code generation :' In code generation, architecture of computer also play an imp role. For eg. there may exist a constant that is too large to fit in a word of the target machine.

(6) This may encounter when compiler try to make symbol table entries for eg. there may exist an identifier that has multiple declaration with contradictory attributes.

** Capabilities of error handling table :-
A good error diagnostic reduces the degugging and maintainence effort. It can have no. of properties :-
1. The error message should be in the form of source program rather than in some internal representation form.
2. The error msg should have clarity

3. It should properly locate the block or line number.

4. There should be only one single error msg for a error.

ERRORS

Lexical Phase - mis-spelled token

Syntax Phase - missing paranthesis or erroneous arithmetic expressions.

Semantic Phase - Undeclared names & type incompatibility

Intermediate code - Incompatible operand types

Code optimizer - certain statements that are not reachable due to wrongly written function calls.

Code generator - Some of the created words are too large to fit to registers of target machine

Symbol Table :- Book keeping routine may detect multiple declarations with contradictory attributes

PASSES OF A COMPILER :-

1. A pass in compiler design is the group of several phases of compiler to perform analysis or synthesis of source programs.
2. There are 2 types of compiler :-
   (i) One pass compiler
   (ii) 2 pass or multiple pass compiler
3. Pass also means that some of the inner operations are repeated several times
4. In one pass structure, both analysis & synthesis of source program is done in the flow from beginning to end of program.
5. In 2 pass structure, analysis of source program is done in 1st pass and synthesis of source program is done in 2nd pass.
6. Eg. if we have the source code

```
i++; i++; i++;
```

then inside 1 pass compiler, it would generate

```
i = i+1; i = i+1; i = i+1;
```

and if the compiler would be 2 pass or multiple pass, than we have

```
i = i+3;
```

7. The more passes a compiler has, the better optimized code it can generate but -

is slower because it must repeat some step again & again.

* Difference between

Attribute	Single pass	Multiple pass compiler
1) Grouping of phases	All phases are grouped into 1	Diff phases of compiler are grouped into more than 1 pass
2) Scanning	Source program is scanned only once.	Source prog. is scanned twice or more
3) Memory Requirement	It requires extremely high storage.	It requires less storage. ∴ the space used by the compiler prog. can be reused in the following pass. ∴ It use very small space
4) Overheads	Less overheads are involved	More overheads are involved
5) Time	Time spent is constructing the intermediate code & scanning it in the subsequent passes can be saved so it takes less time.	It takes more time because each pass reads & writes an intermediate file.
6. Implementation	One pass compiler was easy to implement	Difficult to implement
7. Process of Compilation.	It uses forward declarations to determine how to link & compile code.	A multipass compiler 1st passes the entire

	Single pass	Multi pass
	: there is only 1 pass, a function or variable cannot be used or referenced before it is declared	file as the 1st pass then compiles the parsed data in the 2nd pass. It do not require forward declaration & can optimize the compiled code by performing analysis on the parsed code before compiling
2. Imposing Restrictions	It imposes certain restrictions on the program.	It doesnot impose any kind of restriction on the prog
9 Code efficiency	Inefficient	Effiient for further use.
10 Cost	Less	More.
11 Complexity	High	Low
12. Ease to modify	Difficult	Easy
13 Another name	also known as narrow compiler	wide compilers
14. eg.	Pascal & C requires single pass compiler	Java requires multipass compiler.

8 Difference betn phases & passes of compiler.

→ when we talk about the phases of compiler, we talk about the steps of compiler which a compiler should perform in order to finish a job

Eg Phases could be creating local variables in symbol table, generating parse tree, lexical checking, optimization, etc

→ Passes means some operation was repeated. It means that before giving the results, it repeat some steps multiple times

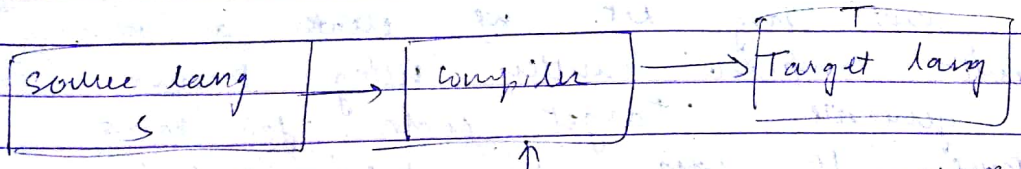
mostly the optimization step

** BOOT STRAPPING :-

1. It is a process for creating a compiler in its own language using the facilities offered by that lang.

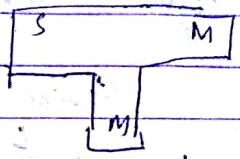
In bootstrapping process, for constructing any compiler, we require 3 languages.

→ Source language, S, which we are creating
→ Target language, T
→ Intermediate lang, I, also known as Implementation lang in which the compiler is written.

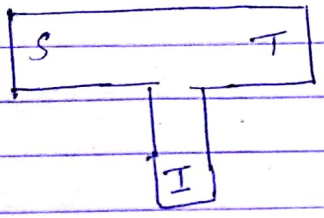


machine code (M)

T diagram →



C_M^{SM} | $S_M M$



T-diagram / Bratman diagram.

C_I^{ST} OR $S_I T$

→ It is represented by a T diagram as given above and this T diagram can be denoted by symbol C_I^{ST} or $S_I T$.

→ Bootstrapping is also defined as technique of writing the compiler in its own lang. by using the feature provided by that lang.

Eg. Consider the development of 1st compiler in which we were knowing only the machine code M of that machine and using this M, we can write a compiler for lang. S. ∴ We have created a lang. S, that can be used to write other compiler for

** Cross Compiler :- Using the concept of bootstrapping, we can create a compiler that may run on one machine and produce target code for some other machine. Such a compiler is called cross compiler.

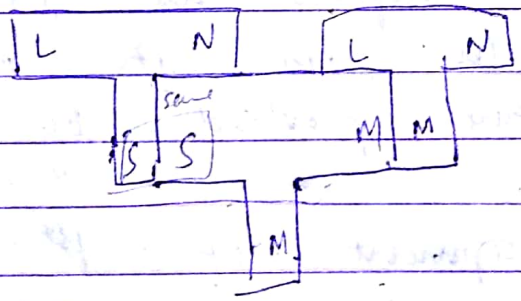
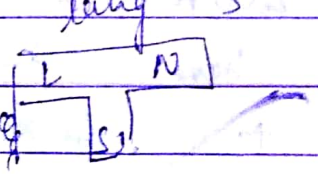
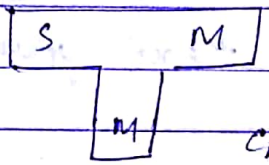


Fig (a) $C_S^{LN} + C_M^{SM} \rightarrow C_M^{LN}$

Suppose we write a cross compiler for a new language L in implementation language S to generate code for machine N



If an existing compiler for S runs on machine M and generate code for M i.e. we create

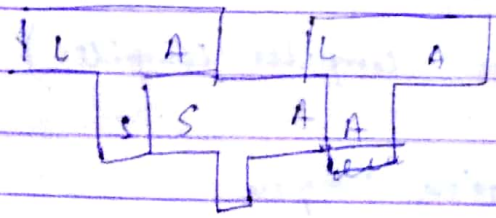


C_M^{SM}

If C_S^{LN} is run through C_M^{SM} we get a compiler C_M^{LN} i.e. a compiler from N to N which can run on machine N which is shown in fig (a)

Note :- The implementation lang, S of the compiler C_S^{LN} must be the same as the source lang. of the existing compiler C_M^{SM} and that the target lang, M of the existing compiler must be same as the implementation lang. of the translated form C_M^{LN}

Q Create a cross compiler C_S^{LA} using previously known lang C_A^{SA}



$$C_S^{LA} + C_A^{SA} = C_A^{LA}$$

Imp

** Compiler construction Tools
OR
Compiler writing Tools

1. Scanner generator Tools :-
   Eg. LEX , FLEX

2) Parser generator Tools :-
   eg. YACC (Yet Another compiler compiler)

5. Code generation.

6. Compiler Construction Tool kit.

Compiler Construction Tools are the miscellaneous s/w modules that help the compiler designer in compiler writing process so as to create the automated design of specific compiler components. These compiler construction tools are also called Compiler generators or translator ability system.

The tools that are needed in compiler construction are as follows :-

1. Scanner Generators :- They produce lexical analysis from a regular expression description of token of a language. In this module, the inputs are regular expression and o/p is a well defined DFA with a transition table.

CFG → Parser generation → Parsing Table

Parser Generators

- Top-Down Parser Generator
  - LL(1) Parser
- Bottom up Parser generator
  - ~~Simple~~ SLR Parser — Simple LR Parser
  - ~~Canonical~~ CLR Parser — Canonical LR Parser
  - LALR Parser — Look Ahead LR Parser

Powerful :- SLR ≤ LALR ≤ CLR : most powerful

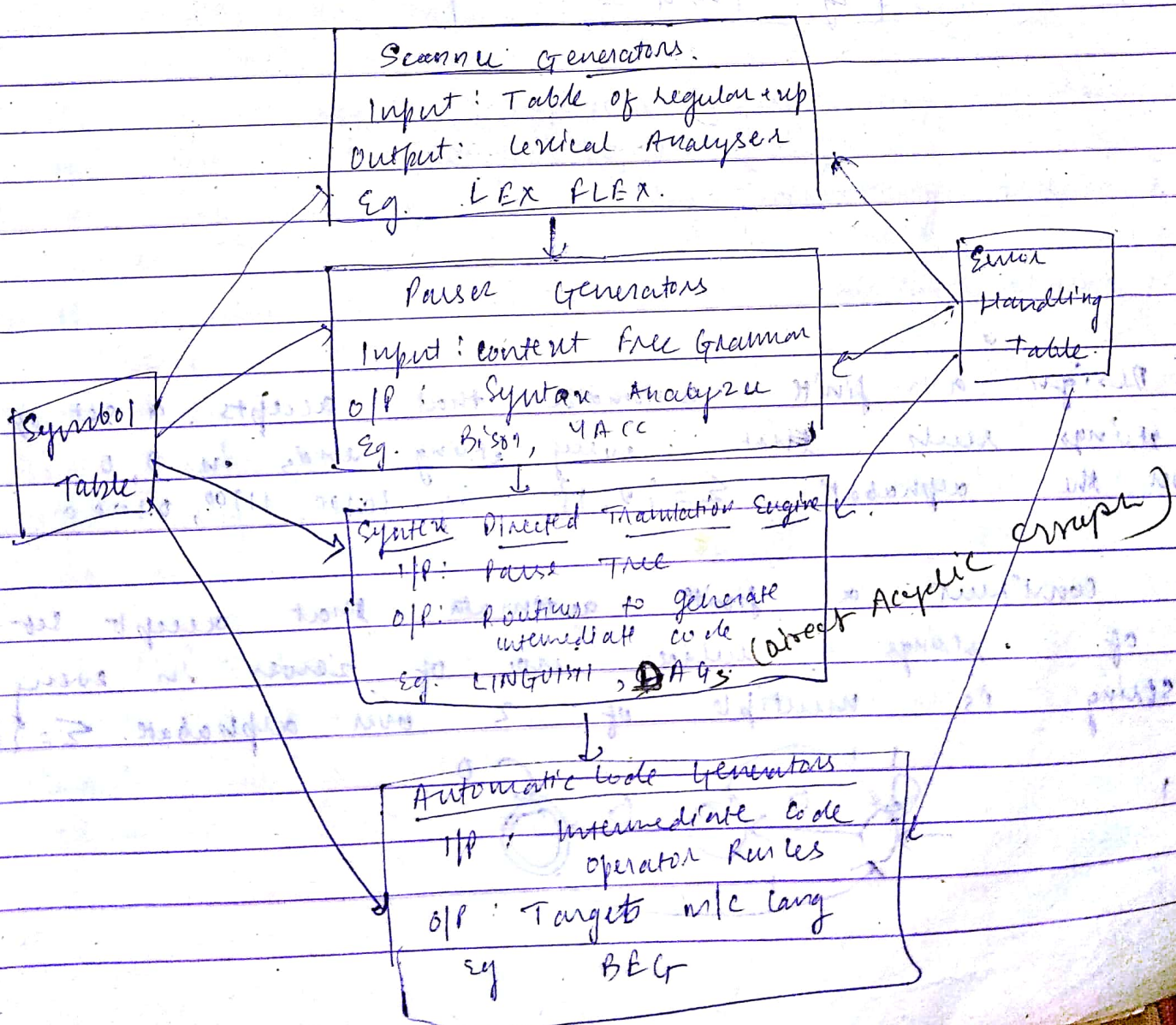
Space requirement :- SLR = LALR ≤ CLR

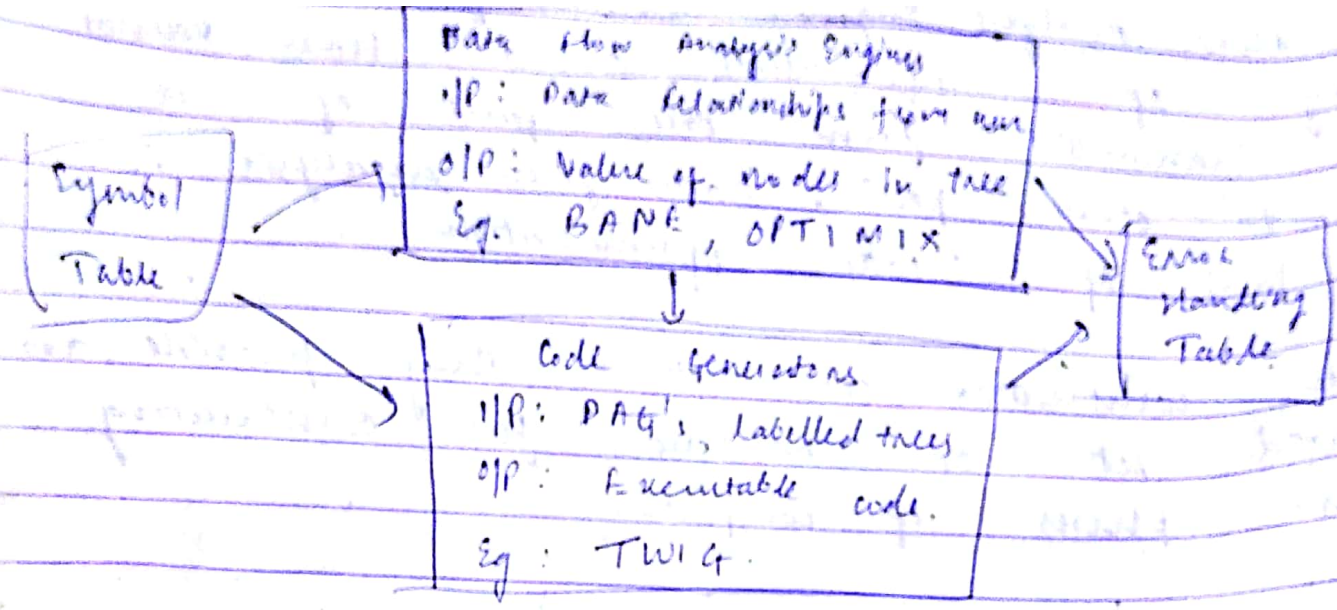
3. Syntax directed Translation engine :- They produce collection of routines for walking a parse tree and generating intermediate code.

4. Code generator generators :- They produce a code generator from a collection of rules for translating each operation of the intermediate lang into the machine lang ~~of~~ or a target machine.

5. Data flow Analysis Engine :- They facilitate the gathering of information about how values are transmitted from one part of a prog. to other part. Data flow analysis is key part of code optimization.

6. compiler construction tool kits :- They provide an integrated set of routines for constructing various phases of compiler.

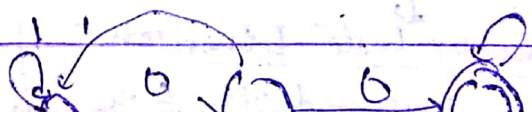




Q.1 Design a finite automata that accepts a set of strings such that every string ends in 0, 0 over the alphabet $\Sigma = \{0, 1\}$ Eg. 10100, 11100, 01000

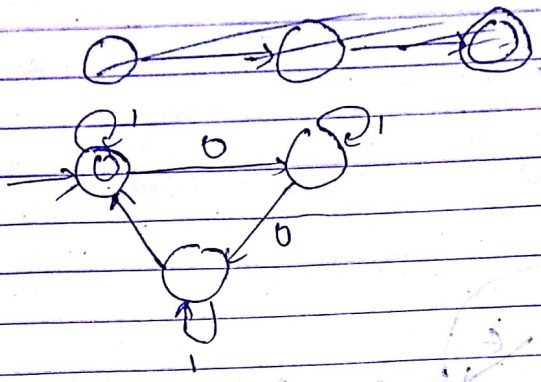
Q.2. Construct a finite automata that accept set of strings where no. of zeroes in every string is multiple of 3. over alphabate $\Sigma = \{0$

Ans 1



+ : logical or

Ans 2



Q Find a regular expression corresponding to each of the following subset {0, 1}.

(i) the language of all strings containing atleast 2 zeroes.

(ii) the language of all strings containing atmost 2 zeroes (either one 0 or 2 zero)

(iii) the lang. of all string ending with 1 and do not contain 00.

(iv) the lang. of all strings in which both the no. of zeroes and no. of 1's are odd.

Ans

(1) $(0+1)^* 0 (0+1)^* 0 (0+1)^*$

(2) $1^* + 01^* + 10 1^*$ | $1^* + 01^* + 101^*$

(3) ~~$1^* + 01^* + 10 1^* +$~~ $(1 + 01 + 10)^*$

(4) ~~$0^* + 1^* +$~~ $(00 + 11)^* 01$ | $(00)^* 0 + (11)^* 1$

010

a/b = a + b. a.a

Q Convert the following regular expression into NFA

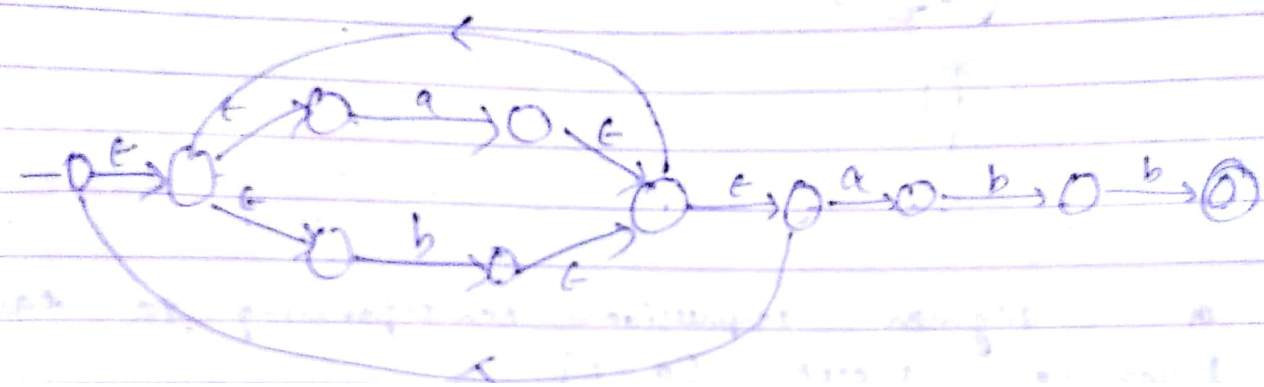
(1) (a/b)* abb

(2) aa* / bb*

(3) letter / (letter / digit)*

(4) (a+b) / (b+c)

(1)



a/b = a + b.

Q Convert the following regular expression into NFA

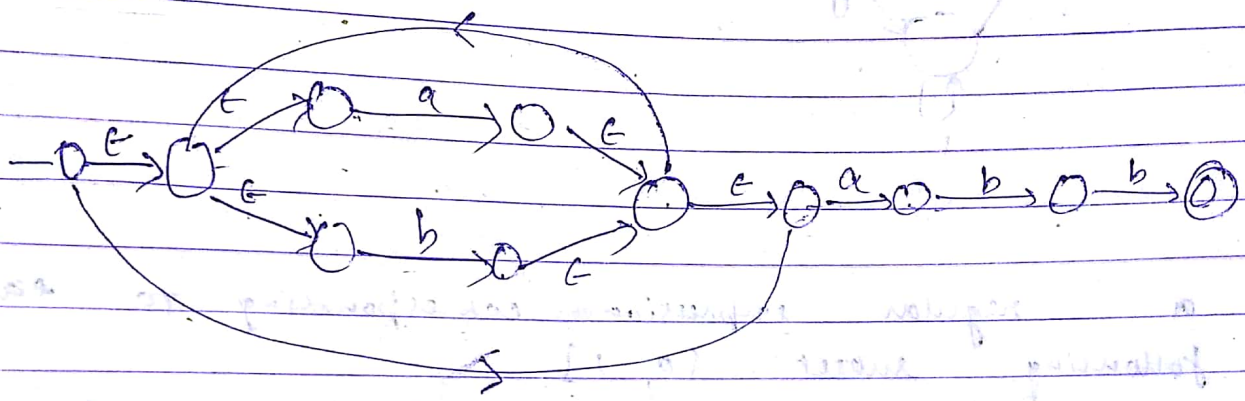
(1) (a/b)* abb

(2) aa* / bb*

(3) letter / (letter / digit)*

(4) (a+b) / (b+c)

(1)



a/b = a + b. a.a

Q Convert the following regular expression into NFA

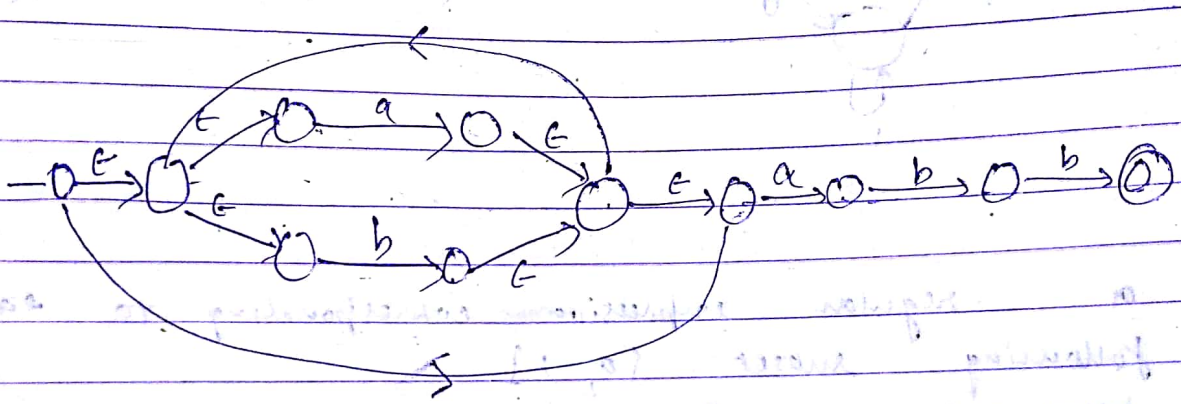
(1) (a/b)* abb

(2) aa* / bb*

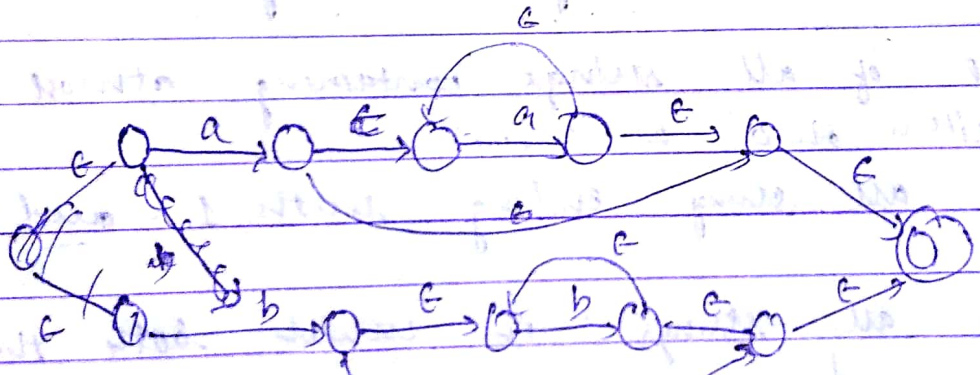
(3) letter / (letter / digit)*

(4) (a+b) / (b+c)

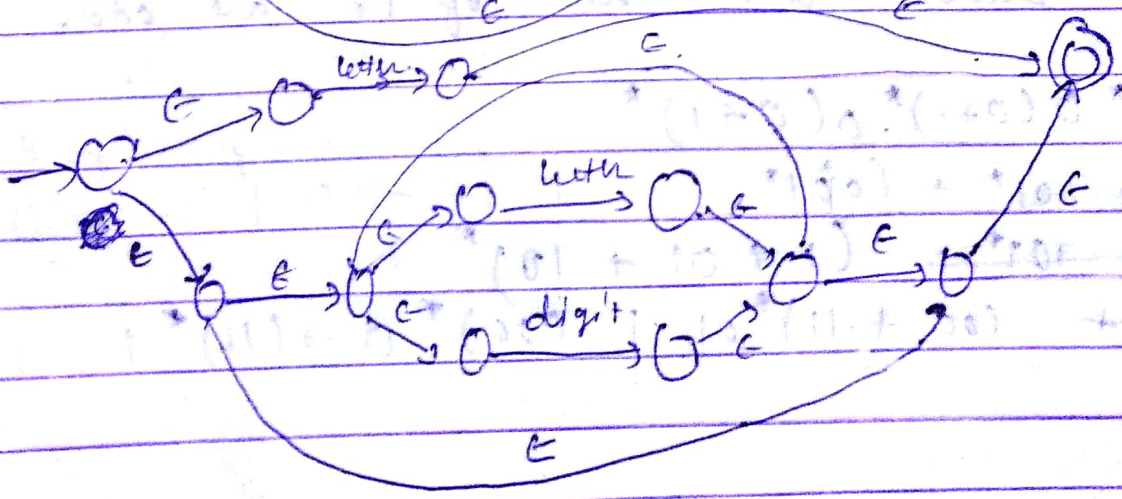
(1)



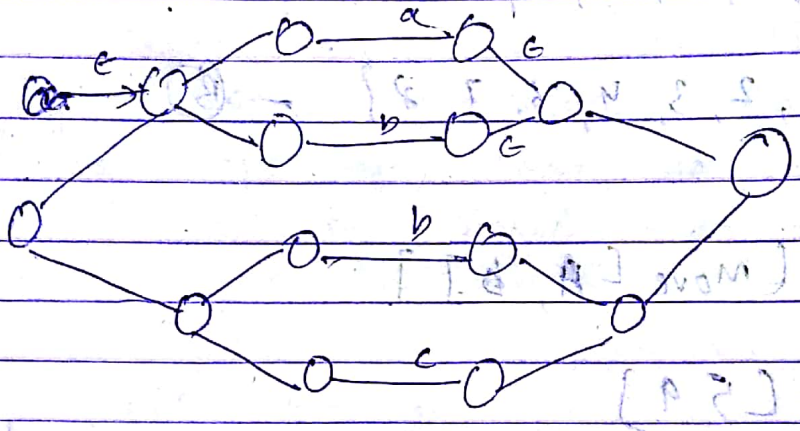
(2)



(3)



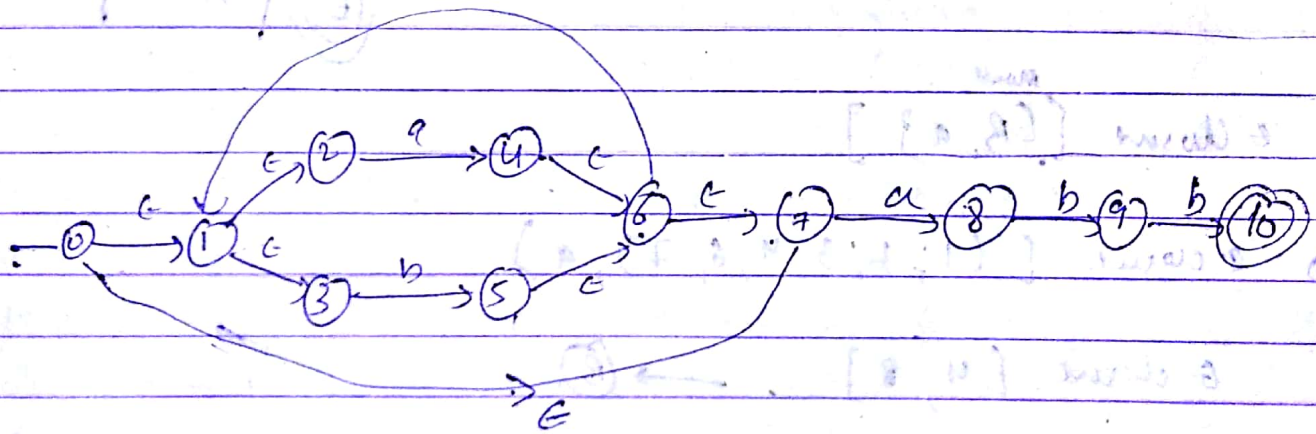
(4)



Q. Construct DFA for the following regular expression

(1) $(a|b)^* abb$

⇒



* ε closure (0) = {0, 1, 2, 3, 7} → (A)

ε closure [Move [A, a]]
δ

= ε-closure [4, 8]

= ε-closure {4} ∪ ε-closure {8}

$= \{1, 2, 3, 4, 6, 7\} \cup \{8\}$

$= \{1, 2, 3, 4, 6, 7, 8\}$ → (B)

ϵ closure [Move [A, b]]

ϵ closure [5, 9]

ϵ closure $\{5\} \cup \epsilon$ closure $\{9\}$

$\Rightarrow \{5, 6, 1, 2, 3\} \cup \{9\}$

$= \{1, 2, 3, 5, 6, 9\}$ → (C)

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C

ϵ closure [Move [B, a]]

$\Rightarrow \epsilon$ closure [(1, 2, 3, 4, 6, 7), a]

ϵ closure [4, 8] → (B)

ϵ closure [Move [B, b]]

$\Rightarrow$ ϵ

Short note*

* Specification of tokens :-

Regular expressions are imp. notations for specifying lexeme patterns. Each pattern match's with a set of string so regular expression will serve as names for set of strings.

Alphabate, string and language :-

* Alphabate :- Alphabate is any finite set of symbols. The typical example of symbols are letters, digits and punctuation.

Alphabate Σ = { a ~~finite~~ finite set of symbols }
= { letters / digits / punctuations }
= { a ---- z / A ---- z / 0 - 9 }

* Strings :- A string over an alphabate is a finite sequence of symbols drawn from that alphabate. Eg. Banana is a string of length 6.

(a)- Prefix of String :- A prefix of string 's' is any string obtained by removing 0 or 1 symbol from the end of 's'.

Eg banana, ana are suffixes of string banana.

(c) Substring of string:- It is obtained by deleting any prefix and any suffix from string.
Eg. ban, ana etc are substring of string banana.

(d) Subsequence of string:- A subsequence is any string formed by deleting zero or more not necessarily consequtive position of string.
Eg. baan is subsequence of string banana.

* Language:- A language is any countable set of string over some fixed alphabate.
There are several important operation that can be applied to a language.

operations on languages:- Let L be the set of {A|B|...|Z|a|b...|z} and M be set of all the digits {0|1|..|9}
then following operations can be performed
1. Union:- LUM = {s|s is in L or s is in M}
2. Concatenation:- LM = {st | s is in L and t is in M}
3. Kleene Closure:- $L^* = \bigcup_{i=0}^{\infty} L^i$

4. Positive closure :- $L^+ = \bigcup_{i=1}^{\infty} L^i$

* Regular expression for token :-

(1) Identifier :- identifier is the set of string of letters and digits beginning with a letter. So the regular expression for identifier is represented as letter. (letter / digit)*

where letter = A | B | ... | z | a | b | ... | z

digit = 0 | 1 | ... | 9

2. Number :-

num = digits. optional-fraction optional-exponentiation

digits = digit. digit* = digit+

optional-fraction = (.) digits

= (.) digit. digit*

= (.) digit+

optional exponent = E(+ / - / ϵ) digits

= E (+ / - / ϵ) digit. digit*

= E (+ / - / ϵ) digit+

General regular expression for a number is

num = digit+ (.) digit+ E(+ / - / ϵ) digit+

③ Regular expression for white spaces:-

(a) ws = (space | Newline | tab)$^+$

4. Relational operators

$(< | <= | > | >= | =)^+$

** Recognition of Tokens:-

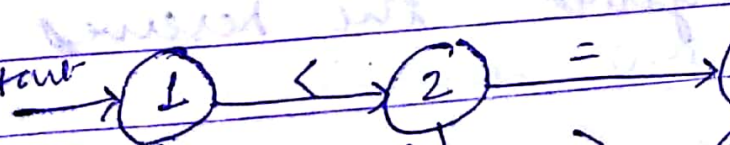
	Lexemes	Token Name	Attribute Value
1.	Any ws	—	—
2.	if	if	—
3.	then	then	—
4.	else	else	—
	Any id	identifier	Pointer to symbol table
	Any number	number	Pointer to symbol table
5.	<	relop	LT
	<=	—"—	LE
6.	>	—"—	GT

Transition diagrams have a collection of nodes or circles called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of the pattern.

Some imp. conventions about transitions diagrams are :-

1. Some states are said to be accepting or final state. These states indicate that a lexeme has been found.
2. It is necessary to retract the forward pointer 1 position and then we additionally place a * symbol near that accepting state.
3. One state is designated as the start or initial state. It is indicated by an edge labelled start

* Transition Diagram for Relational operators :-



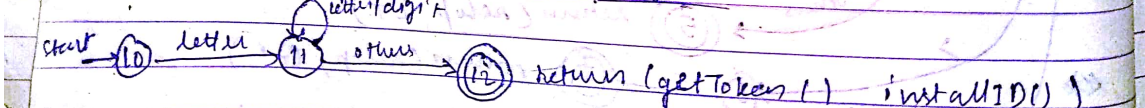
* Recognition of ~~Reserved~~ Reserved words & identifiers:-

There are 2 ways that we can handle reserved words that look like identifiers.

1. Install the reserved words in the symbol table initially. A field of the symbol table entry indicates that these strings are not ordinary identifiers and they told us which tokens to represent.
2. Whenever we find an identifier, a call to installId() places it in the symbol table if it is not already there, and returns a pointer to the symbol table entry for the lexeme found.
3. The function gets tokens, examines the symbol table for the lexeme found, and returns whatever token name the symbol table lexeme represents.

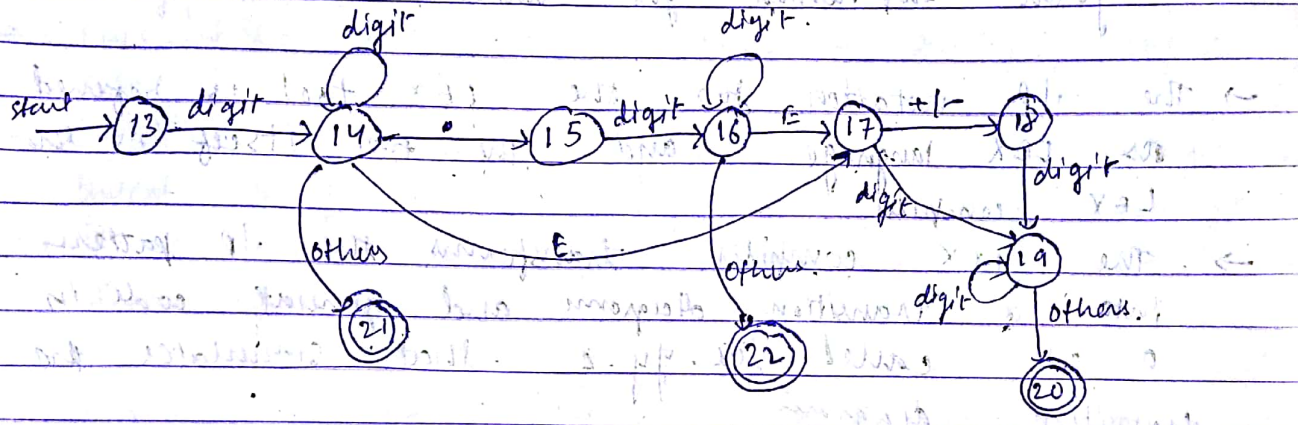
2. Create separate transition diagrams for each keyword and treat recognize the reserved words

* Transition diagram for identifiers:-



id = letter . (letter / digit)*

* Transition diagrams for numbers :-



num = digits optional_fraction optional_exponent

digits = digit . (digit)* = digit+

optional_fraction = (.) digits
= (.) digit . (digit)*
= (.) digit+

1254

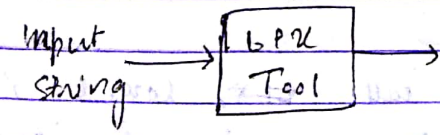
123.451

123.451E +/-23

123.451E46

123E +/-47

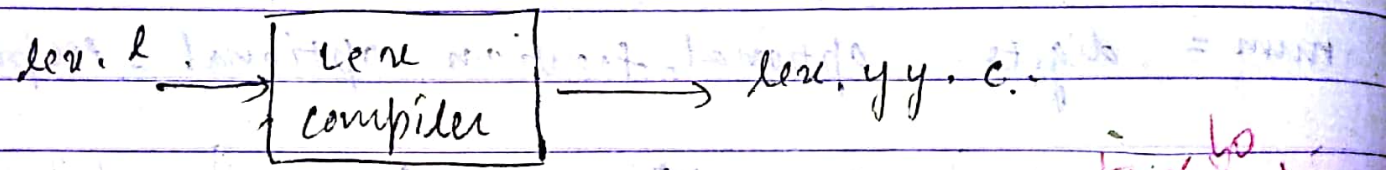
** lexical Analyzer Generator - LEX :-

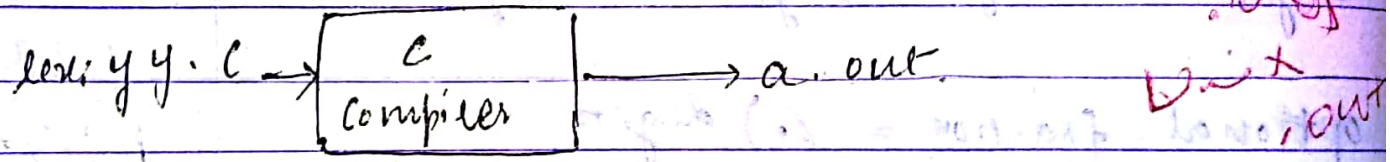


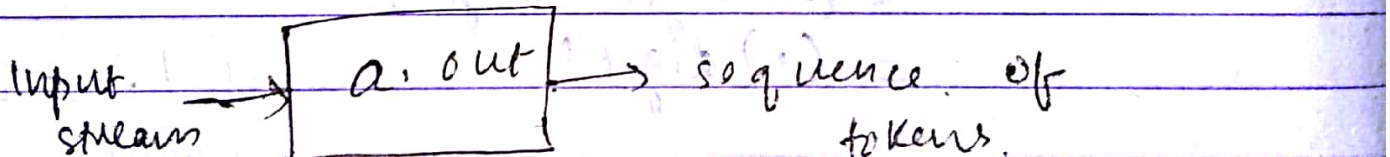
→ The lexical analyzer generator tool - LEX allows us to describe pattern of tokens by specifying regular expression for various tokens.

→ The I/P notation for the LEX tool is referred as LEX language and the tool itself is the LEX compiler

→ The LEX compiler transforms the I/P pattern into a transition diagram and generate code in a file called lex.yy.c that simulates the transition diagram







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

→ This file lex.yy.c is compiled by C compiler into a file called a.out.

→ The C compiler o/p a.out is a working lexical analyzer that can take a stream of i/p characters and produce a sequence of tokens.

Structure of lex Program :-

A lex program has following 3 parts :-

declarations
%% %%
Translation Rules
%% %%
Auxiliary Functions

1. declaration :- The declaration section include declarations of variables and regular definition
The declarations are surrounded by the special bracket in the form %{ and %}

- Lex rules may use the regular definition of declaration section and the actions are fragment of code typically written in C or in any other implementation lang.

3. Auxiliary functions :- The 3rd section holds the additional functions that are used in the action part of translation rules.

Lexical analyzer for token :-

```
%{
/* definition of manifest constants
   LT, LE, EQ, NE, GT, GE, IF,
   THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions */
delim   [ \t \n]
ws      {delim}+
letter  [A-Z a-z]
digit   [0-9]
id      {letter}.({letter/digit})*
number  {digit}+(\.{digit}+)(E[+-]{digit}+)
%. %.
{ws}    {/* no action and no return */}
if      {return(IF);}
then    {return(THEN);}
else    {return(ELSE);}
{id}    {yylval=(int)  install_ID(); return(ID);}
{number} {yylval=(int) install_NUM(); return(NUMBER);}
```

definitions

```
"<"   {yylval = LT;  return (RELOP);}
"<="  {yylval = LE;  return (RELOP);}
"="   {yylval = EQ;  return (RELOP);}
"<>"  {yylval = NE;  return (RELOP);}
">"   {yylval = GT;  return (RELOP);}
">="  {yylval = GE;  return (RELOP);}
%%
int installID() {/* function to install the lexem, whose
                    first character is pointed by yytext.
                    and whose length is yyleng into
                    the symbol table & return a pointer */}
int installNUM() {/* similar to installID but put
                     numerical constants into a separate
                     table */}
```

Code for Recognizing identifier

```
#include <stdio.h>
#include <ctype.h>
void main()
{
char ch;
ch = getchar();
if (isalpha(ch))
ch = getchar();
else error();
while (isalpha(ch) || isdigit(ch))
        ch = getchar();
}
```

abc123

UNIT - II SYNTAX ANALYSIS

→ Checks syntax of the program and constructs abstract syntax tree

→ It also helps in error reporting and recovery

→ This phase is modeled using context free grammar

→ This phase recognise using pushdown automata or table driven parser

** What syntax analysis cannot do :-

1. Check whether a variable has been declared before use
2. To check whether a variable has been initialized

* Role of Parser :-

→ The parser report the syntax error in an intelligible fashion and to recover from commonly occuring errors to continue the processing of the remaining prog
→ The parser also constructs a parse tree and passes it to the rest of the compiler for further processing. Thus the parser & rest of the frontend can be implemented by a single module.

* limitations of regular langs :-
→ Many langs are not regular. for eg the string of balanced paranthesis
(((( ))))
It can be represented by a grammar
S → (S) | ε

→ Regular expression cannot be used for syntax analysis because
(i) The pumping lemma for regular languages prevents the representation of constructs like a string of balanced paranthesis where there is no limit on the no. of paranthesis. This is because a finite automata may repeat states. However it does not have the power to remember no. of times a state has been reached
(ii) Also regular expressions do not provide

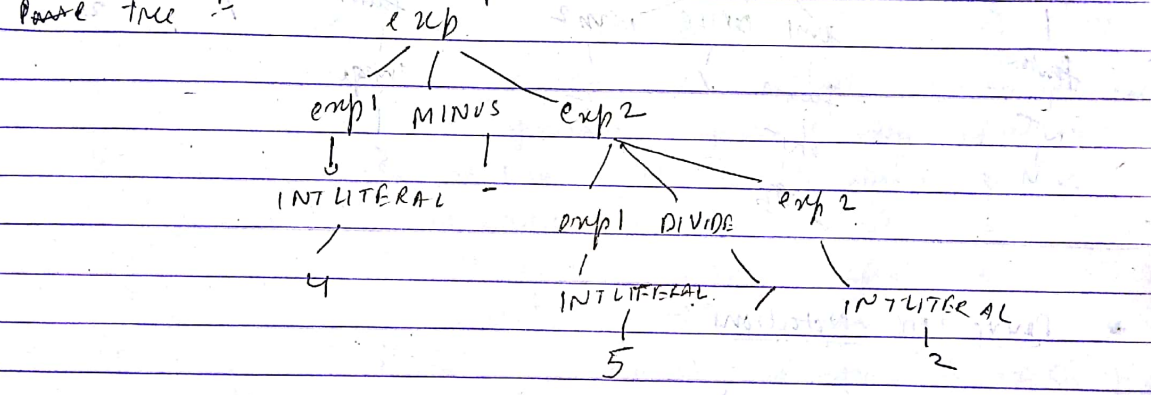
Type 1	Content Sensitive Grammar	A = any string with non terminals B = any string with terminals or non-termin equal to or longer than A. $\|A\| \leq \|B\|$
Type 2	CFG	A = any string with single non terminal B = any string with terminals & non-terminal
Type 3	Regular Grammar	A = any string with single non-terminal B = a or aX where a → Terminal

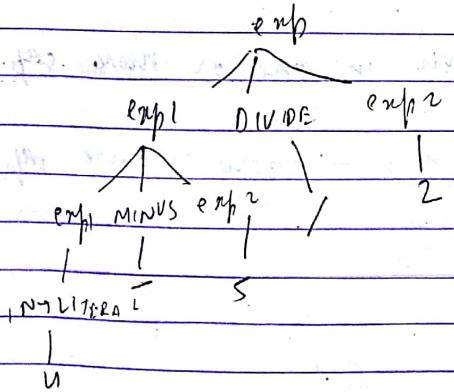
* Representing Simple Arithmetic Expressions using CFG:

Design grammar 4 - 5/2.

exp → INT LITERAL.
exp → exp1 MINUS exp2
exp → exp1 DIVIDE exp2
→ Ambiguous grammar.

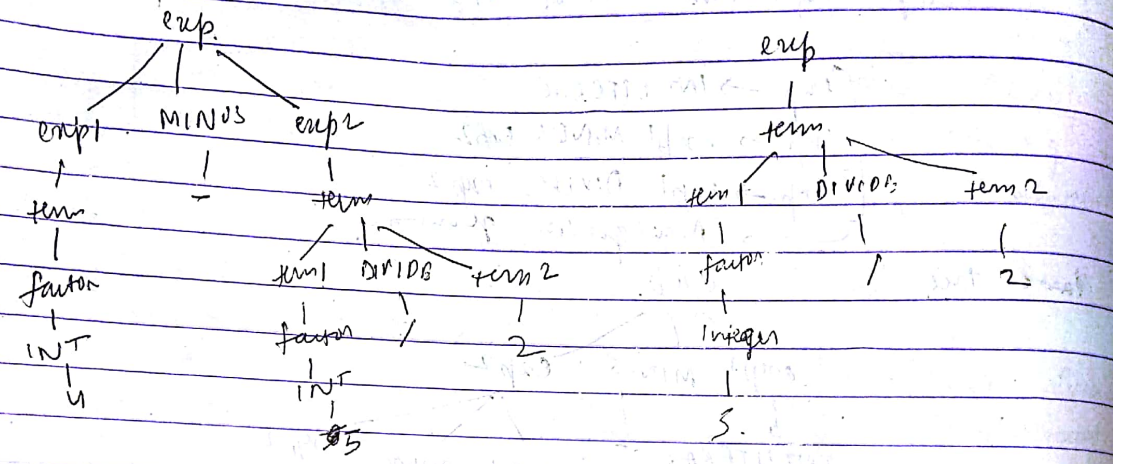
Parse tree :





This is ambiguous, so we have to remove ambiguity.

exp → exp1 MINUS exp2 | term → to next.
term → term1 DIVIDE term2 | factor
factor → INTEGER.



* Derivation Notations :-

$S \Rightarrow +$

- $\Rightarrow$ derives in one step.
- $\Rightarrow +$ or $\overset{+}{\Rightarrow}$ derives in one or more step.
- $\Rightarrow *$ or $\overset{*}{\Rightarrow}$ —— " —— zero or more step.

* Recursion in CFG :-

Rec. { X → abc X AB ab

Left { X → [X]. . . . .

Recursive grammar some non terminal or terminal or non terminal

Same

$X \rightarrow \ldots\ldots X$ → Right recursive grammar

CFG definition :-

A mathematical model for the grammar is defined as

$G = (V_n, V_t, P, S)$

where G is the notation for the grammar

V_N : a finite set of non-terminals generally represented by capital letters

V_t : A finite set of terminals generally represented by small letters.

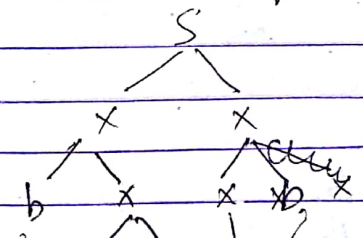
S :- Starting non-terminal also called Start symbol of grammar

P : Set of rules or productions in ~~the~~ CFG

Q. Consider the ~~to~~ CFG $S \rightarrow XX$

$X \rightarrow XXX \mid bX \mid Xb \mid a$

Find the Parse tree for the string "bbaaaab"



↓

yield of parse tree.

Q. Write CFG for the lang. $L = \{ x\,0^n\,y\,1^n\,z \mid n \geq 0 \}$ and find the parse tree for the string "x00y11z"

$n = 0$, x y z
$n = 1$, x 0 y 1 z
$n = 2$, x 00 y 11 z

constant

$S \rightarrow x\,A\,z$
$A \rightarrow y \mid 0\,A\,1$

Q. Let G be a CFG $S \rightarrow bB \mid aA$
$A \rightarrow b \mid bS \mid aAA$
$B \rightarrow a \mid aS \mid bBB$

"bbaababa" For the string, find :-

(1) Left most derivation
(2) Right most derivation
(3) Parse tree

$S \overset{lm}{\Rightarrow} bB$
$\overset{lm}{\Rightarrow} bbBB$
$\overset{lm}{\Rightarrow} bbaSB$
$\overset{lm}{\Rightarrow} bbaaABB$
$\overset{lm}{\Rightarrow} bbaabSB$
$\overset{lm}{\Rightarrow} bbaababB$
$\overset{lm}{\Rightarrow} bbaababa$

Q. Write a CFG for regular expression.

$r = 0^* | (0+1)^*$

$S \rightarrow A \mid B$

$A \rightarrow 0A \mid \epsilon$

$B \rightarrow 0B \mid 1B \mid \epsilon$.

Q. Design CFG for R.E $(a+b)^* aa (a+b)^*$

$S \rightarrow A\,aa\,B$ ~~$S \rightarrow A\,a\,a\,A$~~

$A \rightarrow aA \mid bA \mid \epsilon$ ~~$A \rightarrow aA \mid bA \mid \epsilon \leftarrow A$~~

$B \rightarrow aB \mid bB \mid \epsilon$

Q Write CFG for the lang.

$L = \{a^{2n} b^m \mid n > 0, m \geq 0\}$

$S \rightarrow aa\,AB$ | aab

$A \rightarrow aa\,A \mid \epsilon$

$B \rightarrow bB \mid \epsilon$.

** Parser : Parser is a prog. that takes as I/P string 'w' and produces o/p either a parse tree for 'w'. If 'w' is a valid sentence of a grammar or an error message indicating that w is not a valid sentence of given grammar.

3.

4.

SLR
CLR
LALR]

look ahead

LL (δ) Parser

↙ Left to right

↘ Leftmost derivation.

LR.

↙ Left to right

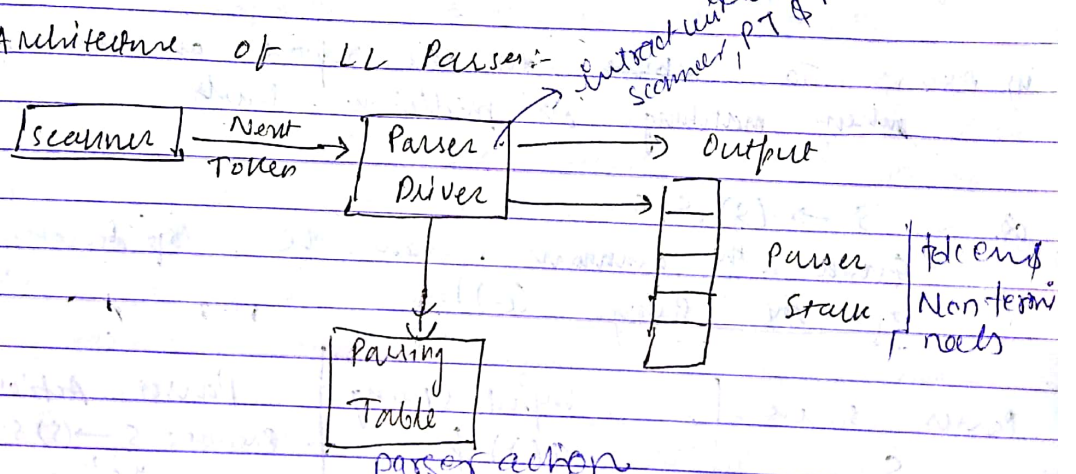
↘ Rightmost deri in reverse

Top Down :-

It constructs the parse tree starting with start symbol and applying the productions rules of grammar to generate given string at the leaves

** LL Parsing :- It uses an explicit stack rather than recursive calls to perform parsing.
→ LL(k) parsing means that k-tokens of look-ahead are used. Here the 1st L means that token sequence is read from left to right and the 2nd L means leftmost derivation is applied at each step.

Architecture of LL Parser :-



LL Parser consist of :-
1. Parser stack : It holds grammar symbols i.e non-terminals & tokens
2. Parsing table :- It specifies the parser action
3. Parser driver :- This function interacts with parser stack, parsing table, & scanner.

LL Parsing Actions :-

(1) Match :- To match top of parser

$: End marker of i/p string

stack with next i/p token

2) Predict :- To predict a production & apply it in a derivation stack.

3) Accept :- To accept & terminate parsing of a sequence of tokens

4) Error :- To report an error msg when matching or prediction fails.

Eg. $S \rightarrow (S)\,S \mid \epsilon$

Consider the grammar. Show the top down parsing of the string. (())

Parser stack	Input string	Parser Actions
S	(()) $	Predict: S → (S) S
(S) S	(()) $	Match "("
S) S	()) $	Predict: S → (S) S
(S) S) S	()) $	Match "("
S) S) S	)) $	Predict S → ε
) S) S	)) $	Match ")"
S) S	) $	Predict S → ε
) S	) $	Match ")"
S	$	Predict S → ε.
–	$	Accept.

Q. Problems with top down parsing.

(1) Backtracking : After studying the concept of top-down parsing, it is clear that top-down parsing attempts to find the leftmost derivation for I/P string

→ Basically in top down mechanism, every terminal symbol generated by some production of the grammar is matched with the I/P string pointed by the string marker

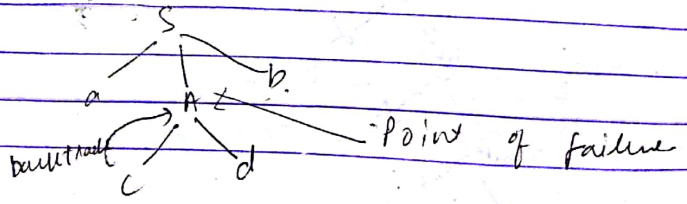
→ If the match is successful, the parse can continue and if a mismatch occurs, then our predictions may gone wrong

→ The prediction which leads to the mismatching terminal symbol is rejected and the string marker is reset to the posn where the rejected production was made. This is known as Backtracking

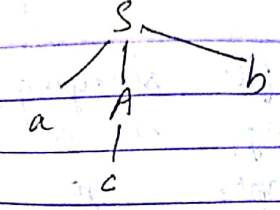
→ Backtracking is one of the major drawback of top down parsing.

Eg. Consider the given gramar $S \rightarrow aAb$
$A \rightarrow cd \mid c$

Show the backtracking for the string w = acb



Backtracking



2. Left recursion :- The grammar, G is ~~send~~ ~~set~~ + said to be left recursive. if it has a non terminal A in the form $A \rightarrow A\alpha / \beta$ where α & β are strings of terminals or non terminals or both.

$A \rightarrow A\alpha$

$A \xRightarrow{\text{on}} A\alpha$

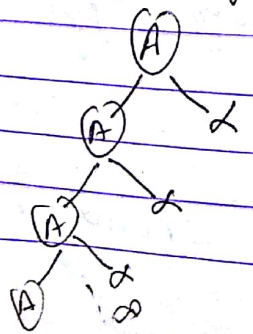
$\Rightarrow A\alpha\alpha$

$\Rightarrow A\alpha\alpha\alpha$

⋮

∞ times.

If left recursive grammar is used for specification of the lang, then the top down parser design for the grammar language may enter into an infinite lo



This is because a top down parser always attempts to obtain the left most derivation for the I/P string 'w', the parser may see the same non terminal 'A' every time as the leftmost non-terminal & every time it may do the derivation $A \rightarrow A\alpha$ ... For top down parsing, non-left recursive grammar should be used.

Left recursion can be eliminated from the grammar by replacing $A \rightarrow A\alpha \mid \beta$.

$$\boxed{\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon \end{array}}$$

Q. Eliminate left recursion from the grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Sol^n $E \rightarrow E+T \mid T$

$A = E$ $\quad \alpha = +T$, $\beta = T$

Replacing with

$$\boxed{\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \end{array}}$$

$T \rightarrow T*F \mid F$

Here $A = T$ $\quad \alpha = *F$, $\beta = F$

$T \rightarrow FT'$

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

Resulting grammar after eliminating left recursion

$E \rightarrow TE'$
$E' \rightarrow +TE'/\epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT'/\epsilon$
$F \rightarrow (E)/id$

8. Consider the following CFG $S \rightarrow Bb/a$
$B \rightarrow Bc/Sd/e$

Find left recursion & remove it

~~$B \rightarrow Bc/Sd/e$~~
$B \rightarrow Bc/Bbd/ad/e$
$B \rightarrow Bc/ad$
$B \rightarrow Bbd/e$

$B \rightarrow adB'$
$B' \rightarrow cB'/\epsilon$

$B \rightarrow eB'$
$B' \rightarrow bdB'/\epsilon$

* Left Factoring :- Left factoring is useful for producing a grammar suitable for predictive parser

If there is any production of the form

$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$.

This can be eliminated by replacing above production

$A \longrightarrow \alpha A'$

$A' \longrightarrow \beta_1 \mid \beta_2$

Q Consider grammar $S \longrightarrow iEtSeS \mid iEts \mid a$

$E \longrightarrow b$.

find left factoring & remove it

Sol:

$S \longrightarrow iEtSeS \mid iEtS$

$S \longrightarrow a$

$E \longrightarrow b$.

$A = S$, $\alpha = iEtS$, $\beta_1 = eS$, $\beta_2 = \epsilon$

final grammar

$S \longrightarrow iEtSS'$

$S' \longrightarrow eS \mid \epsilon$

$S \longrightarrow a$

$E \longrightarrow b$

** Terminologies Related with Top down Parsing :-

FIRST() and ~~First~~ Follow() are 2 functions associated with a grammar that help us in filling the entries of M-table.

→ FIRST() function is a function which gives the set of terminals that begin the strings derived from the production rule.

→ FOLLOW() is a function which gives set of terminals that can appear immediately to the right of a given symbol.

* Benefits of FIRST() & FOLLOW() :-

1. Can be used to proof LL(k) characteristics of the grammar.
2. Can be used in the construction of predictive parse tables
3. Provides selection info. for recursive descent parser

* Rules for Calculating FIRST()

If $\alpha = XYZ$

(i) FIRST() of α = FIRST() of (XYZ) if $X \in$ Terminal then $FIRST(\alpha) = \{x\}$

(ii) if $X \in$ Non-Terminal (A, B, X) then

(a) if $X \to \epsilon$ in the grammar then add $\{\epsilon\}$ to FIRST(X)

Eg. $A \to \epsilon$

$FIRST(A) = \{\epsilon\}$

Terminal : small letter.

(b) if $X \in$ Non Terminal (A, B, X) and $X \to Y_1 Y_2 Y_3 \ldots Y_k$
then FIRST (X) = FIRST(Y_1)
if FIRST (Y_1) does not contain ϵ i.e. ϵ free.

(c) if $X \in$ Non-Terminal and $X \to Y_1 Y_2 Y_3 \ldots Y_k$ &
FIRST (Y_1) contains ϵ
then

$$FIRST(X) = FIRST(Y_1) - \{\epsilon\} \cup FIRST(Y_2, Y_3 \ldots Y_k)$$

Q $S \to ACB \mid CbB \mid Ba$ Calculate FIRST() of given grammar.
$A \to da \mid BC$
$B \to g \mid \epsilon$
$C \to h \mid \epsilon$

$\Rightarrow$ FIRST (C) = FIRST (h) $\cup$ FIRST (ϵ)
$= \{h\} \cup \{\epsilon\}$
$= \{h, \epsilon\}$.

FIRST (B) = FIRST (g) $\cup$ FIRST (ϵ)
$= \{g\} \cup \{\epsilon\}$
$= \{g, \epsilon\}$

FIRST (A) = FIRST (da) $\cup$ FIRST (BC)
$= \{d\} \cup [FIRST(B) - \{\epsilon\} \cup FIRST(C)]$ (Rule 6)
$= \{d\} \cup [\{g, \epsilon\} - \{\epsilon\} \cup \{h, \epsilon\}]$
$= \{d\} \cup [\{g\} \cup \{h, \epsilon\}]$
$= \{d, g, h, \epsilon\}$

$S \rightarrow ACB \mid CbB \mid Ba$

$FIRST(S) = FIRST(ACB) \cup FIRST(CbB) \cup FIRST(Ba)$

$= [FIRST(A) - \{\epsilon\} \cup FIRST(CB)] \cup [FIRST(C) - \{\epsilon\} \cup FIRST(b)]$
$\cup [FIRST(B) - \{\epsilon\} \cup FIRST(a)]$

$= [\{d, g, h, \epsilon\} - \{\epsilon\} \cup [FIRST(C) - \{\epsilon\} \cup FIRST(B)]]$
$\cup [\{h, \epsilon\} - \{\epsilon\} \cup \{b\}] \cup [\{g, \epsilon\} - \{\epsilon\} \cup \{a\}]$

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

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

$= \{a, b, d, g, h, \epsilon\}$

Q Calculate FIRST()

$E \rightarrow TA$

$A \rightarrow +TA \mid G$

$T \rightarrow FB$ (terminal)

$B \rightarrow *FB \mid \epsilon$

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

$FIRST(F) = FIRST(\epsilon) \cup FIRST(E) \cup FIRST(\cdot) \cup FIRST(id)$

$= \{\epsilon\} \cup FIRST(CE)) \cup FIRST(id)$

$= FIRST\{(\} \cup \{id\}$

$= \{(, id\}$

- Rules
(i) Place.
(ii) If.
exempt
in
(iii) If.
B.
exe

FIRST (B) = FIRST (*F B) U FIRST {ϵ}
= {*} U {ϵ}
= {*, ϵ}

FIRST (A) = FIRST (+TA) U FIRST (ϵ)
= {+} U {ϵ}
= {+, ϵ}

FIRST (T) = FIRST {F B}
= FIRST (F)
= {(, id}

FIRST (E) = FIRST (TA)
= FIRST (T)
= {(, id}

- Rules for calculating FOLLOW()

(I) Place $ in FOLLOW(S) if S is a start symbol

(II) If there is a production $B \rightarrow \alpha A \beta$ then everything in FIRST(β) except for ϵ is place

Capital :- Non terminal

In short,

i) $B \rightarrow A\alpha$

- $\alpha \in T \rightarrow FOLLOW(A) = FIRST(\alpha) = \{a\}$
- $\alpha \in NT$'s and is ϵ free: $FOLLOW(A) = FIRST(\alpha)$
- $\alpha \in NT$'s and is ϵ - containing: $FOLLOW(A) = FIRST(\alpha) - \{\epsilon\} \cup FOLLOW(B)$

ii) $B \rightarrow \alpha A \longrightarrow FOLLOW(A) = \{\epsilon\} - \{\epsilon\} \cup FOLLOW(B)$
$= FOLLOW(B)$

Q. $S \rightarrow ACB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

Calculate FOLLOW() of given grammar

Soln $FOLLOW(S) = \{\$\}$

$FOLLOW(A) = FOLLOW(S \rightarrow ACB)$
$= FIRST(CB) - \{\epsilon\} \cup FOLLOW(S)$
$= [FIRST(C) - \{\epsilon\} \cup FIRST(B)] - \{\epsilon\} \cup \{\$\} = [\{h, \epsilon\} - \{\epsilon\} \cup \{g, \epsilon\}] - \{\epsilon\}$
$= \{h, \$\}$ $= \{g, h, \$\}$

$FOLLOW(B) = FOLLOW(S \rightarrow ACB) \cup FOLLOW(S \rightarrow CbB) \cup FOLLOW(S \rightarrow Ba) \cup FOLLOW(A \rightarrow BC)$
$= FOLLOW(S) \cup FOLLOW(S) \cup FIRST(a) \cup [FIRST(C) \cup FOLLOW(A)]$

= {~~$~~} ∪ {$} ∪ {$} ∪ {a} ∪ [{h, ∈} − {∈}] ∪ {g, h, $}

= {g, a, h, $}

FOLLOW(C) = FOLLOW(S → ACB) ∪ FOLLOW(C → bB) ∪ FOLLOW(B → c)

= [FIRST(B) − {∈} ∪ FOLLOW(S)] ∪ FIRST(B) ∪ FOLLOW(A)

= [{g, ∈} − {∈} ∪ {$}] ∪ {h} ∪ {h, $, g}

= {b, g, h, $}

Q

E → TA

A → +TA | ∈

T → FB

B → *FB | ∈

F → (E) | id

Soln FOLLOW(E) = {$} ∪ {FOLLOW(F → (E))}

= {$} ∪ FIRST())

= {), $}

FOLLOW(A) = FOLLOW(E → TA) ∪ FOLLOW(A → +TA)

= FOLLOW(E) ∪ FOLLOW(A)

= {), $}

= { *, +,), $ }

* Construction of a predictive Parse Table.

Algorithm :-

Input : Grammar G
O/P : Predictive Parse Table [M - table]

Method :

1) For each production $A \to \alpha$ of the grammar.
do step 2 & step 3

2) For each terminal "a" in FIRST (α) & $A \to \alpha$
to M [A, a]

3) if ε is in FIRST (α), add ~~A → ε~~ $A \to \alpha$ to M[A, b]
for each terminal "b" in FOLLOW (A)
if ε is in FIRST (α) & $ is in FOLLOW(A)
then add $A \to \alpha$ to M [A, $]

4) Make each undefind entry of M as error

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

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

find FIRST() & FOLLOW() and construct the predictive parse table and also check that the given string $w = id * (id + id)\$$ is successfully parsed or not.

Sol

Step 1 :- Elimination of left Recursion :-

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

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Step 2 Calculation of FIRST().

FIRST(F) = FIRST('(') ∪ FIRST(id)

= {(, id}

FIRST(T') = FIRST(*) ∪ FIRST(ε)

= {*, ε}

FIRST(E') = FIRST(+) ∪ FIRST(ε)

Follow (E) = {$} ∪ {), $}

Follow (E') = {), $}

Follow (T) = First (E') - {ε} ∪ Follow (E)
= {+,), $}

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

Follow (F) = First (T') - {ε} ∪ Follow (T)
= {+, *,), $}

Skip Construction of Predictive Parse Table :-

M-Table.

NTs \ Ts	+	*	(	)	id	$
E			E → TE'		E → TE'	
E'	E' → +TE'			E' → ε		E' → ε
T			T → FT'		T → FT'	
T'	T' → ε	T' → *FT'	T' → ε			T' → ε
F			F → (E)		F → id	

Select ε-free productions :- ~~E → TE'~~ F → (E)

First (F) = First ((E))

= {(}

$M[F, (] = \{F \to (E)\}$

2) $F \to id$

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

$M[F, id] = \{F, id\}$

3) $T' \to *FT'$

$FIRST(T) = FIRST(*FT')$

$= \{*\}$

$M[T', *] = T' \to *FT'$

4) $T \to FT'$, $FIRST(T) = FIRST(FT')$

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

$M[T, (] \setminus T \to FT'$

$M[T, id] /$

5) $E' \to +TE'$, $FIRST(E') = FIRST[+TE']$

$M[E', +] \to E' \to +TE'$

6) $E \to TE'$

$FIRST(E) = FIRST(TE')$

$= FIRST(T)$

selecting ε-containing productions

1) $E' \rightarrow \epsilon$

FOLLOW(E') = FOLLOW($E \rightarrow TE'$) ∪ FOLLOW($E' \rightarrow +TE'$)

= FOLLOW(E) ∪ FOLLOW(E')

= {), $}

M[E',)] = $E' \rightarrow \epsilon$.

M[E', $]

2) $T' \rightarrow \epsilon$.

FOLLOW(T') = FOLLOW($T \rightarrow FT'$) ∪ FOLLOW($T' \rightarrow *FT'$)

= FOLLOW(T) ∪ FOLLOW(T')

= {+,), $}

M[T', +]

M[T',)] — $T' \rightarrow \epsilon$.

M[T', $]

As the predictive parse table or M table has only 1 entry in each cell. So the given grammar is LL(1).

Eg: Parsing the I/P string

w = id * (id + id) $

Stack	I/P string	Action
$E	id * (id + id) $	M[E, id] = E → TE'
$ E'T	id * (id + id) $	M[T, id] ⟹ T → FT'
$ E'T F	id * (id + id) $	M[F, id] = F → id
$ E'T' id	id * (id + id) $	pop id
$ E'T'	* (id + id) $	M[T', *] = T' → *FT'
$ E'T'F*	* (id + id) $	pop *
$ E'T'F	(id + id) $	M[F, (] = F → (E)
$ E'T')E(	(id + id) $	pop (
$ E'T')E	id + id) $	M[E, id] = E → TE'
$ E'T')E'T	id + id) $	M[T, id] =

M Imp

Q Construct the predictive parse table for the following grammar :-

$S \rightarrow i C t S S' \mid a$

$S' \rightarrow e S \mid \epsilon$

$C \rightarrow b$.

Also find FIRST() & FOLLOW(). Show that whether grammar is LL(1) or not.

Soln Step I :- FIRST(C) = {b}

FIRST(S') = FIRST(e S) ∪ FIRST(ε)

= {e, ε}.

∴ FIRST(S) = FIRST(i C t S S') ∪ FIRST(a)

= {a, i}.

Step II FOLLOW(S) = {$} ∪ FOLLOW(S → i C t S S') ∪ FOLLOW(S'

~~FOLLOW(S') = FOLLOW(S → i C t S S')~~

~~= FOLLOW(S)~~

~~= {$}~~

~~FOLLOW(C) = FOLLOW(S → i C t S S')~~

= FIRST

= {$} ∪ [FIRST(S') − {ε}] ∪ FOLLOW(S) ∪ FOLLO

= {$} ∪ [{e, ε} − {ε}] ∪ FOLLOW(S) ∪ FOLLOW(S')

[FOLLOW(S → iC

$= \{\$\} \cup [\{e\} \cup FOLLOW(S)] \cup FOLLOW(S)$

$= \{\$, e\}$

$FOLLOW(S') = follow(S \rightarrow iCtSS')$

$= FOLLOW(S)$

$= \{\$, e\}$

$FOLLOW(C) = FOLLOW(iCtSS')$ if $\alpha\beta$

$= FIRST(tSS')$

$= \{t\}$

Step III Construction of Predictive parsing Table

NT's \ T's	i	t	a	e	b	$
S	$S \rightarrow iCtSS'$		$S \rightarrow a$			
S'				$S' \rightarrow eS$ $S' \rightarrow \epsilon$		$S' \rightarrow \epsilon$
C					$C \rightarrow b$	

Select ϵ-free production. $S \rightarrow iCtSS'$

$FIRST(S) = FIRST(iCtSS')$

$= \{i\}$

$M[S, i] = \{S \rightarrow iCtSS'\}$

G containing

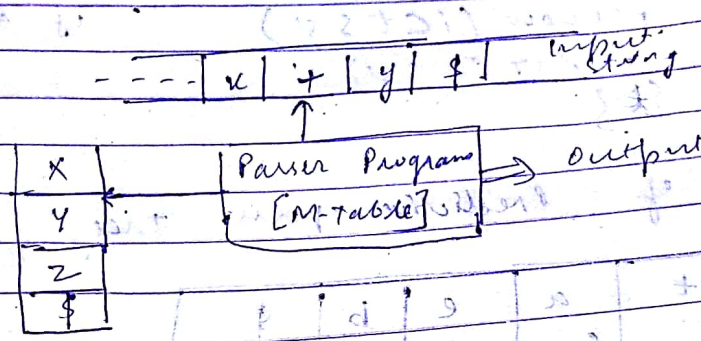
$S' \rightarrow G$

FOLLOW(S') = {$, e}

∴ in one cell, there are 2 productions,

So it is not LL(1)

** TOP DOWN PREDICTIVE PARSER :- / Algorithm for Predictive Parsing



If X is the top most symbol of the stack & 'a' is the current I/P string than.

(i) If X = a = $ than the parser handts and it announce the successful completion of parsing.

(ii) If X is a terminal and X = a ≠ $ than parser POP X from stack and advance the I/P pointer to the next symbol

(iii) If X is a non terminal than the parser consult the parse table M[X, a] if M[X, a] contains the production in the form X → UVW than parser pus

the production in the reverse order i.e.. W V U

(4) If M[x, A] is blank then the parser announces error and the parser cause the error recovery routines.

** LL(1) Grammar :-

A grammar is said to be LL(1) grammar if and only if its M table has no multiple entries. We can also define LL(1) grammar as the first L stands for left to right scanning of I/P and the second L stands for left most derivation and the (1) indicates that next I/P symbol is used to decide next parsing process. i.e. the length of ~~look Ahead~~ lookahead is 1.

For a grammar to be LL(1), the following condition must be satisfied :-

For every pair of production, $A \rightarrow \alpha / \beta$.

{

contain ϵ than $FIRST(\alpha) \cap FOLLOW(A) = \phi$

}

Q Consider the grammar

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Test whether given grammar is LL(1) or not.

Sol" $A \rightarrow \alpha \mid \beta$

$A = S$, $\alpha = AaAb$, $\beta = BbBa$

$FIRST(\alpha) \cap FIRST(\beta) = \phi$ (ϵ free).

$FIRST(AaAb) \cap FIRST(BbBa)$

$= \{FIRST(A) - \{\epsilon\} \cup FIRST(aAb)\} \cap [FIRST(B) - \{\epsilon\} \cup FIRST(b$

$= \{\{\epsilon\} - \{\epsilon\} \cup \{a\}\} \cap \{\{\epsilon\} - \{\epsilon\} \cup \{b\}\}$

$= \{a\} \cap \{b\} = \phi$.

This proves above grammar is LL(1)

Q. Consider the following grammar $S \rightarrow IAB \mid \epsilon$

$A \rightarrow IAC \mid OC$

$B \rightarrow OS$

$C \rightarrow I$

Check whether grammar is LL(1) or not

$S \rightarrow IAB \mid \epsilon$.

$FIRST(IAB) \cap FOLLOW(S)$

G. 8.11.21, 22, 24, 30/5 - 1

= {1} ∩ [{$} ∪ FOLLOW (B → OS)]
= {1} ∩ [{$} ∪ FOLLOW (B)]
= {1} ∩ [{$} ∪ FOLLOW (S → 1AB)]
= {1} ∩ [{$} ∪ FOLLOW (S)]
= {1} ∩ [{$} ∪ FOLLOW (S)] = {1} ∩ {$} = ϕ

Q S → ict ss' / a , S' → es / ∈ , C → b

FIRST (ict ss') ∩ FIRST (a)
= {i} ∩ {a}
= ϕ

(ii) FIRST (e S) ∩ FOLLOW (∈)
{e} ∩ FOLLOW S' → e
{e} ∩ [FOLLOW S → ict ss']
{e} ∩ [FOLLOW (S' → e S)]
FOLLOW {S']
= {e} ∩ {FOLLOW (S')}
= {e} False.

** BOTTOM UP PARSING / Shift Reduce Parser / LR Parser :- (Left to right Parsing ? Rightmost derivation in Reverse)

→ It attempts to traverse a parse tree in a bottom up manner
→ It reduces a sequence of tokens to the

start symbol

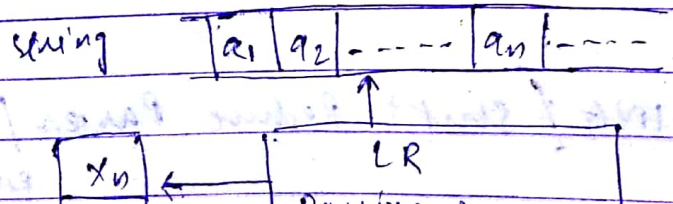
→ At each reduction step, the RHS of a production is replaced with LHS. and this reduction step corresponds to reverse of a rightmost derivation.

* Advantages of Bottom up parsing :-

1. LR parser can recognise all programming lang constructs
2. It is most general non-backtracking shift reduced parsing method
3. It can detect syntactic errors as soon as possible in left to right scan of I/P.
4. It can parse a large class of grammar which is proper super set of the class of grammar parsed with predictive parser

* Drawbacks :-

→ Too much work is needed to construct LR parser for a typical programming lang.



* Construction of SLR Parsing Table :-

1) Augmented Grammar :- Let a given grammar G, $G = (V_N, V_t, P, S)$ where V_N : non-empty finite set of non terminals.
V_t : non-empty set of terminal.
S : Starting non-terminal
P : Production or rules of grammar.

If we modify the above grammar by adding a unit production $S' \rightarrow S$ to the given grammar and now the starting non-terminal becomes S' & grammar can be written as $G' = (V_N \cup \{S'\}, V_t, P \cup \{S' \rightarrow S\}, S')$

Eg. $E \rightarrow E + T / T$
$T \rightarrow T * F / F$

Augment grammar $E' \rightarrow E$, $T \rightarrow T * F$
$E \rightarrow E + T$, $T \rightarrow F$
$E \rightarrow T$

2) LR(0) ITEMS :-

LR(0) item of a grammar, G is a production of G with a dot at some position of the right side of the production i.e. if we have a production $T \rightarrow XYZ$. Then the LR(0) item of this production are

I_0 :
$T \rightarrow .XYZ$
$T \rightarrow X.YZ$
$T \rightarrow XY.Z$
$T \rightarrow XYZ.$

if G has ε-production i.e. we have $T \to \epsilon$ then LR(0) item of this production is $T \to .$

If we have an LR(0) item as $T \to X . Y Z$ This indicates that a string derivable from X has been seen so far on the I/P and we hope to see a string derivable from YZ next on the I/P.

* **Closure operation on a set of items :-**

If I is a set of item for a grammar then closure (I) is a set of items constructed from I by the following 2 rules:

(1) Initially every LR(0) item in I is added to closure (I)

(2) If $A \to \alpha . B\beta$ is in closure (I) & $B \to \gamma$ is a production rule of G then $B \to . \gamma$ will be in the closure (I)

Eg

$E' \to E$
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

$E' \to . E$
$E \to . E + T$
$E \to . T$
$T \to . T * F$
$T \to . F$

closure

$E' \to . E$
$E \to . E + T$
$E \to . T$
$T \to . T * F$

A LR parser consist of :-

(1) Driver Program or Parsing Program : It reads the i/p string one symbol at a time and maintains a parsing stack. The so driver prog. is used for all types of LR parser.

(2) Parsing stack :- It contains state info. wch states are obtained from grammar and

(3.) Parsing Table :- Parsing table has 2 parts

(i) Action Part : It specifies the parser action

(ii) Goto Part : It specifies successor states

(4) i/p string : It is a sequence of tokens supplied to the parser

(5) o/p :-

Entries are labelled in parsing table as

Sn & : shift token & go to state n (and

4. Error: Parser discovers that a syntax error has occured.

In LR parser, the 2 main actions are shift & Reduce. ∴ LR parser is also known as shift reduce parser.

Handle of a Right Sentential form :-

If we have the production $A \to \beta$ then in the right sentential form of the form $S \xrightarrow[rm]{*} \alpha A \omega \Rightarrow \alpha \beta \omega$, this is known as handle.

> A handle of a right sentential form is :-
A substring β that matches the RHS of a production $A \to \beta$

The reduction of β to A is a step along the reverse of a right most derivation.

$E \to E + E$
$E \to E * E$
$E \to id$

Illustrate that the bottom up parser is a reverse of right most derivation.

E is the start symbol & $w = id1 + id2 * id$

Now show the handle of each right

$E \xrightarrow{rm} E + E$

$\xrightarrow{rm} E + E * E$

$\xrightarrow{rm} E + E * id$

$\xrightarrow{rm} E + id * id$

$\Rightarrow id + id * id$

↑ Reverse of Rightmost derivation

stack	Input String	Action	Handle
$	id1 + id2 + id3 $	Shift id1	—
$ id1	+ id2 + id3 $	reduce by $E \to id$	Handle id
$ E	+ id2 * id3 $	Shift +	—
$ E +	id2 * id3 $	Shift id2	—
$ E + id2	* id3 $	reduce by $E \to id$	Handle id
$ E + E	* id3 $	Shift *	—
$ E + E *	id3 $	shift id3	—
$ E + E * id3	$	reduce by $E \to id$	Handle id
$ E + E * E	$	reduce by $E \to E * E$	Handle E * E
$ E + E	$	reduce $E \to E + E$	Handle E + E
$ E	$	accept	—

* Shortcomings of Shift/Reduce Parsing :-

1. Shift / Reduce conflict :- Parser gets confused to make a shift operation or a reduction

2. Reduce / Reduce conflict : The parser cannot decide which of the several reductions to be made

Types of LR Parser :-

- SLR (Simple LR) : It is easier to implement but it may fail to produce a table for certain class of grammar.
- CLR (Canonical LR) : It is most powerful and work on very large class of grammar but it is very expensive & difficult to implement
- LALR (Look Ahead LR) :- This is intermediate in power bet SLR & CLR. It works on most of the class of grammar and with some effort, it can be implemented easily

Powerful : SLR $\leq$ LALR $\leq$ CLR

Memory Require : SLR = LALR $\leq$ CLR

(4) GOTO operation :- If I is the set of items and X is a grammar symbol (terminal or non terminal) than goto (I, X) is defined as ~~goto (I, X)~~

goto (I, X) = closure ({ [A → αX . β] })
such that [A → α . Xβ] is in I

goto (I_0, E) = [E' → E.
E → E. + T]

goto (I_0, T) = [T → T. * F
E → T.]

goto (I_0, ( ) = [F → (. E)]

Q Consider the following grammar :-

E → E + T | T
T → T * F | F
F → (E) | id

Construct the SLR parsing table.

Soln step 1 : Make the given grammar in augmented form by adding a unit production of the form S' → S

0 — E' → E 7

Step II. Construct the LR(0) item set & label it as I_0

I_0 =
$E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

closure [goto (I_0, E)] = $\begin{bmatrix} E' \rightarrow E. \\ E \rightarrow E. + T \end{bmatrix} = I_1$

closure [goto (I_0, T)] = $\begin{bmatrix} E \rightarrow T. \\ T \rightarrow T. * F \end{bmatrix} = I_2$

$$= \begin{bmatrix} T \to .T * F \\ T \to .F \\ F \to .(E) \\ F \to .id \end{bmatrix} = I_6$$

closure [goto $(I_2, *)$] $= \begin{bmatrix} T \to T * .F \\ F \to .(E) \\ F \to .id \end{bmatrix} = I_7$

closure [goto (I_4, E)] $= \begin{bmatrix} F \to (E.) \\ E \to E. + T \end{bmatrix} = I_8$

closure [goto (I_6, T)] = closure $\begin{bmatrix} E \to E + T. \\ T \to T. * F \end{bmatrix}$ = I_9

closure [goto (I_6, F)] = [T → F.] = I_3

closure [goto (I_6, ()] = I_4

closure [goto (I_6, id)] = I_5

closure [goto (I_7, F)] = [T → T * F.] = I_{10}

closure [goto (I_7, ()] = I_4

closure [goto (I_7, id)] = I_5

closure [goto (I_8,))] = [F → (E).] = I_{11}

closure [goto (I_8, +)] = I_6

closure [goto (I_9, *)] = I_7

STATE	ACTION						GOTO		
	+	*	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1	S_6					accept			
2	r_2	S_7		r_2		r_2			
3	r_4	r_4		r_4		r_4			
4			S_4		S_5		8	2	3
5	r_6	r_6		r_6		r_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8	S_6			S_{11}					
9	r_1	S_7		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

$E' \rightarrow E. \longrightarrow I_1$

$r_1 - E \rightarrow E+T. \longrightarrow I_9$

$r_2 - E \rightarrow T. \longrightarrow I_2$

$r_3 - T \rightarrow T * F. \longrightarrow I_{10}$

$r_4 - T \rightarrow F. \longrightarrow I_3$

$r_5 - F \rightarrow (E). \longrightarrow I_{11}$

$r_6 \rightarrow F \rightarrow id. \longrightarrow I_5$

FOLLOW (E) = FOLLOW ($E' \rightarrow E$) $\cup$ FOLLOW ($E \rightarrow E+T$) $\cup$ FOLLOW ($F \rightarrow (E)$)

= FOLLOW (E') $\cup$ FIRST (+) $\cup$ FIRST ())

= { $ } $\cup$ { + } $\cup$ {) }

= { +,), $ }

FOLLOW(T) = FOLLOW(E → E + T) ∪ FOLLOW(E → + T) ∪ FOLLOW(T → T * F)

= FOLLOW(E) ∪ FOLLOW(E) ∪ FOLLOW(T)

= {+,), $} ∪ {+,), $} ∪ {*}

= {+,), $, *}

FOLLOW(F) = FOLLOW(T → T * F) ∪ FOLLOW(T → F)

= FOLLOW(T) ∪ FOLLOW(T)

= {+,), $, *}

⇒ w = id * (id + id) $: Parse by using SLR parsing

STACK	INPUT STRING	ACTION
0	id * (id + id) $	Shift 5
0 id 5	* (id + id) $	r6, F → id
0 F	* (id + id) $	goto 3
0 F 3	* (id + id) $	r4, T → F
0 T	* (id + id) $	goto 2
0 T 2	* (id + id) $	Shift 7
0 T 2 * 7	(id + id) $	Shift 4
0 T 2 * 7 (4	id + id) $	Shift 5
0 T 2 * 7 (4 id 5	+ id) $	r6, F → id
0 T 2 * 7 (4 F	+ id) $	goto 3
0 T 2 * 7 (4 F 3	+ id) $	r4, T → F
0 T 2 * 7 (4 T	+ id) $	goto 2
0 T 2 * 7 (4 T 2	+ id) $	r2, E → T

0T2*7(4E8	+id).$	Shift 6
0.F2*(4E8+6id 5	).$	r6 F→id
0T2*7(4E8+6F	).$	goto 3
0T2*7(4E8+6F3	).$	r4 T→F

** Algorithm for Constructing SLR Parsing Table:-

Step I: If $A \to \alpha . a\beta$ in I_i and goto$(I_i, a) = I_j$ then action$[i, a] = S_j$ [. shift. j]

Step II: If $A \to \alpha . B\beta$ in I_i and goto$(I_i, B) = I_j$ then goto$(i, B) = j$

Step III: If $A \to \alpha .$ is in I_i then place reduce $A \to \alpha$ in Action$[i, a]$ for every 'a' in FOLLOW(A)

Step IV: If $S' \to S$ in I_i then Action$[i, \$]$ = accept.

Step V: All other entries will be error entries

Q. Consider the following grammar:-

$S \to L = R$

$S \to R$

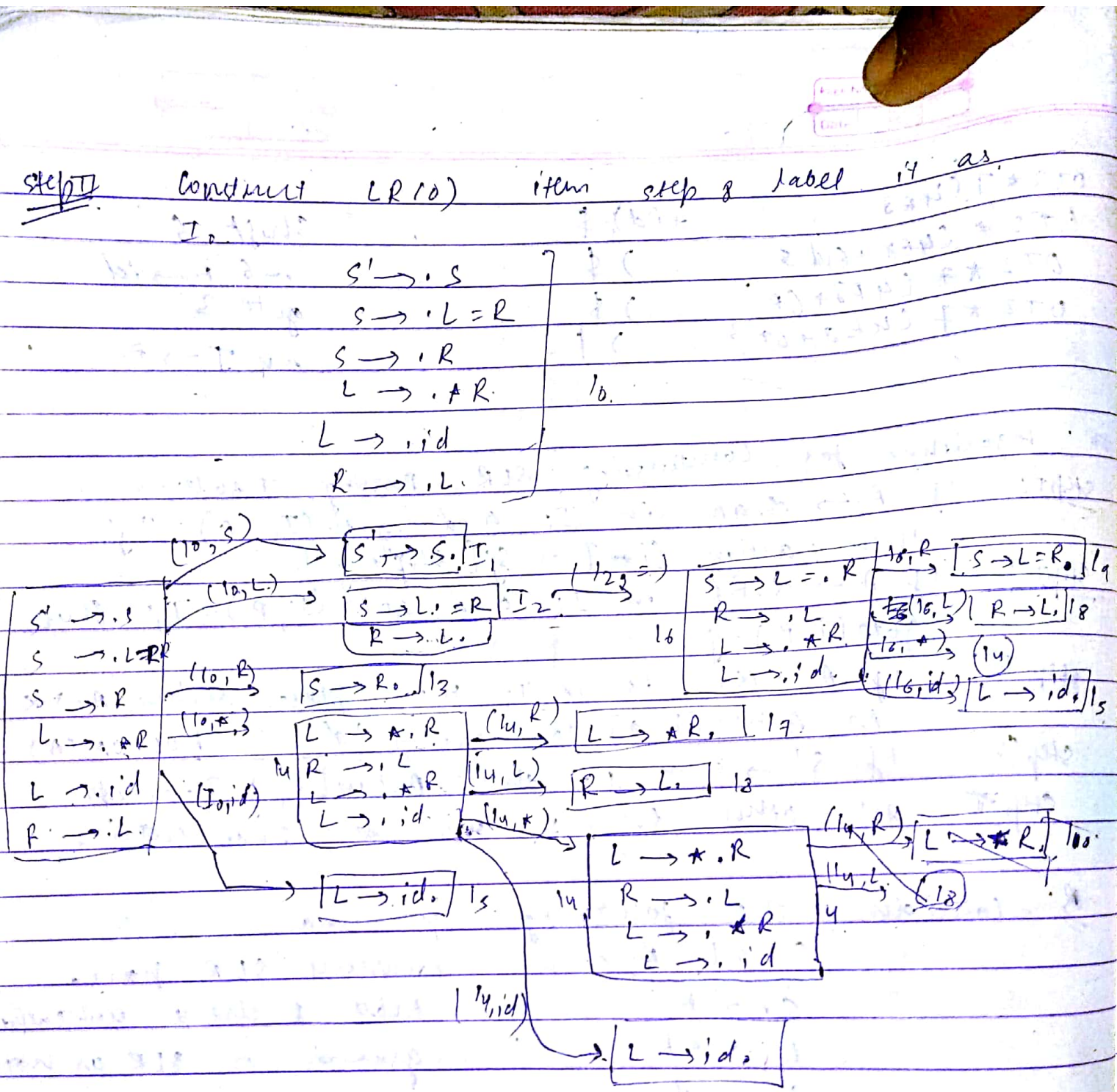
$L \to *R$

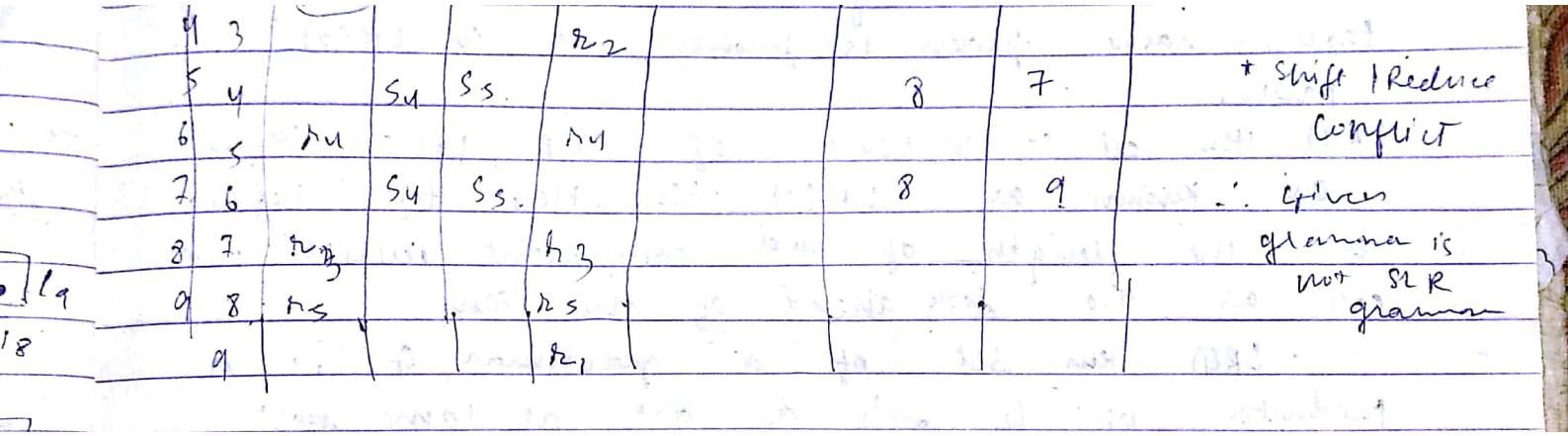
$L \to id$

$R \to L$

construct SLR parse table & check whether grammar is SLR or not

Step II Construct LR(0) item step & label it as I_0.





4	3				R_2					
5	4		S_4	S_5			8	7		* Shift / Reduce
6	5	R_4			R_4					Conflict
7	6		S_4	S_5			8	9		∴ given
8	7	R_3			R_3					grammar is
9	8	R_5			R_5					not SLR
	9				R_1					grammar

FOLLOW(R) = { = , $ }

Goto

** CLR Parsing Table :-

For constructing the CLR or LR(1) parsing table, same process is followed as in LR(0) parser.

LR(1) item set :- In case of CLR, parsed items are known as LR(1) item. Here the 1 refers to the length of 2nd component which we call as the look ahead of the item.

LR(1) item set of a grammar, G, is a production of G with a dot at some position on the right side and a look ahead is also attached. The look ahead of the production can be calculated by given 2 rules

(i) Add entry item in I to closure (I)

(ii) Repeat for every item of the form
A → α . B β, a in closure (I)
do
for every production B → γ
do
add B → . γ, FIRST(βa) to closure (I)
End

Q. E → E + T | T
T → T * F | F
F → (E) | id.

Calculate look ahead for each production

Q6)" Augmented grammar:- $E' \to E$

$E \to E + T$

$E \to T$

$T \to F$

$F \to (E)$

$F \to id$

$E' \to .E$, $\$$ (Rule 3)

$E \to .E + T$, FIRST $(.\$) = \$$

$E \to .T$, $\$$, $\$$

$E \to .E + T$, FIRST $(+T\$) = +$

$E \to .T$, $+$

$T \to .T * F$, FIRST $(\$) = \$$

$T \to .F$, $\$$

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

$T \to .F$, $+$

$T \to .T * F$, FIRST $(*F\$) = *$

$T \to .F$, $*$

$F \to .(E)$, $\$$

When non terminal only then we have to proceed

F → .(E) , *
I → .id , *

I_0 =

E' → . E	$
E → . E + T	+/$
E → . T	+/$
T → . T * F	+/*/$
T → . F	+/*/$
T → .(E)	+/*/$
F → .id	*/+/$

Q. S → L = R.
S → R
L → * R
L → id
R → L

Soln

S' → S.	S' → .S , $
S → L=R	S → .L=R , FIRST(=$) = $
S → R	S → .R , $
L → *R	L → .*R =
L → id	L → .id , =
R → L	R → .L , $
	L → .*R , $
	L → .id , $

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Soln

$S' \rightarrow S$	$S' \rightarrow .S, \$$
$S \rightarrow AaAb$	$S \rightarrow .AaAb, \$$
$S \rightarrow BbBa$	$S \rightarrow .BbBa, \$$
$A \rightarrow \epsilon$	$A \rightarrow ., a$ (FIRST(aAβ))
$B \rightarrow \epsilon$	$B \rightarrow ., b$

Q. Construct CLR parsing table for the given grammar

So

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

Soln Step 1 Make grammar in augmented form

Step 2 Construct LR(1) item set.

$I_0 =$

$S' \rightarrow .S, \$$
$S \rightarrow .L = R, \$$
$S \rightarrow .R, \$$
$L \rightarrow .*R, /\$$

closure [goto(I_0, R)] = [S → R., $] = I_3

closure [goto(I_0, *)] = [L → *.R, =/$; R → .L, =/$; L → .*R, =/$; L → .id, =/$] = I_4

closure [goto(I_0, id)] = [L → id., =/$] = I_5

closure [goto(I_2, =)] = [S → L = .R, $; R → .L, $; L → .*R, $; L → .id, $] = I_6

closure [goto(I_4, R)] = [L → *R., =/$] = I_7

closure [goto(I_4, L)] = [R → L., =/$] = I_8

closure [goto(I_4, *)] = [L → *.R, =/$; R → .L, =/$; L → .*R, =/$; L → .id, =/$] = I_4

closure [goto(I_4, id)] = [L → id., =/$] = I_5

closure [goto(I_6, R)] = [S → L=R., $] = I_9

closure [goto(I_6, L)] = [R → L., $] = I_{10}

closure [goto (I_6, id)] = [L → id ., $] = I_{12}

closure [goto (I_{11}, R)] = [L → *R ., $] = I_{13}

closure [goto (I_{11}, L)] = [R → L ., $] = I_{10}

closure [goto (I_{11}, *)] = I_{11}

closure [goto (I_{11}, id)] = I_{12}

		ACTION			GOTO		
state	=	*	id	$	S'	L	R
0		S_4	S_5		1	2	3
1				accept			

* Diff b/w SLR(1) and CLR(1)

By comparing the SLR(1) parser with the CLR(1) parser, we find that CLR parser is more powerful but CLR has greater no. of states than SLR. Hence the storage req of CLR is greater than SLR parser

SLR(1)	CLR(1)
(1) Concept of ~~follow~~ Follow() is used for constructing SLR parsing table	(1) No calculation of Follow is reqd. on the basis of look ahead we can construct CLR parse state
(2) Less powerful	(2) More powerful
(3) No. of states in SLR are less.	(3) No. of states are greater.
(4) less m/o is reqd	(4) M/O requirement is high.
(5) Less efficient	(5) More efficient
(6) Req to handle ambiguous	(6) Does not require

where the parser power will be in between SLR and CLR and its storage requirement will be same as SLR(1).

3. Construct CLR & LALR parse table.

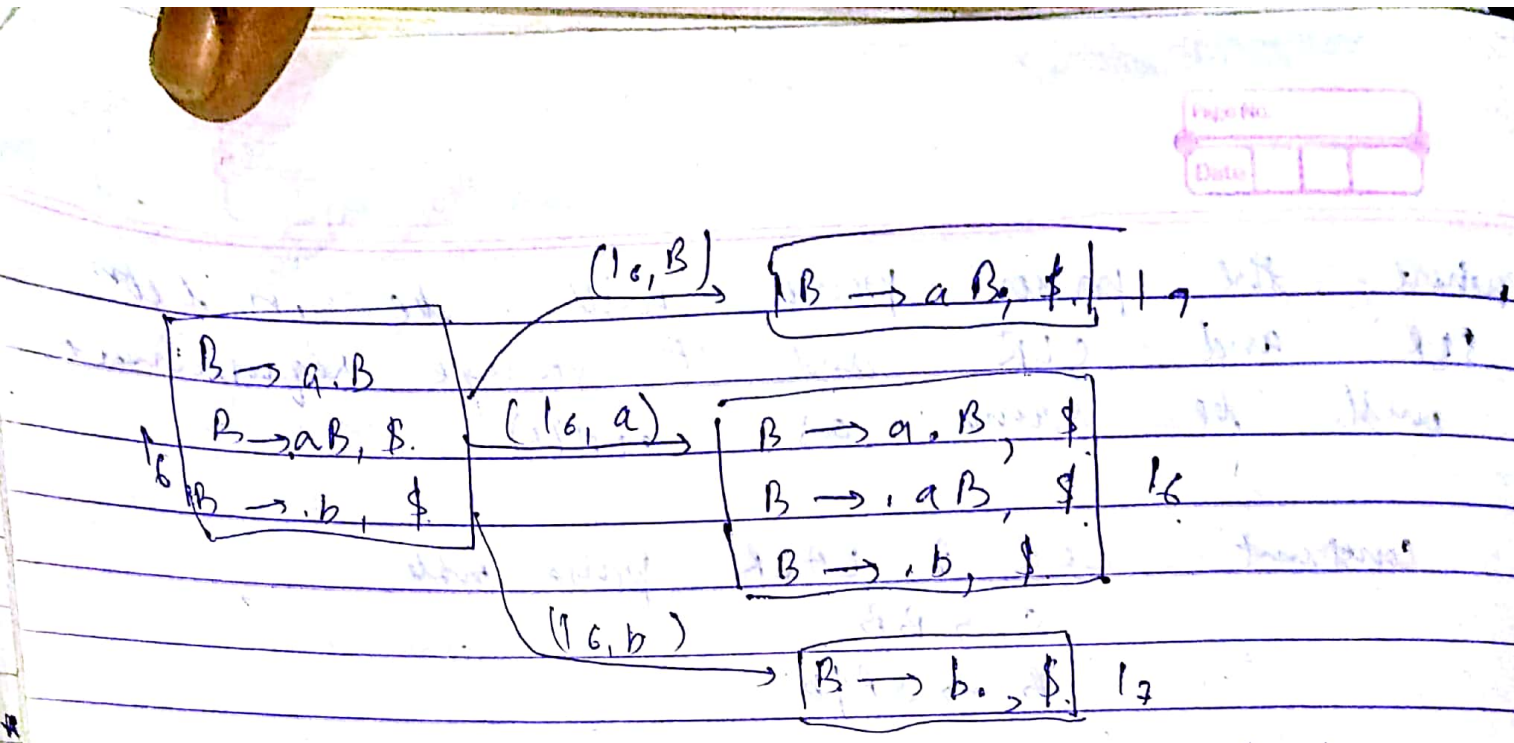
$S \rightarrow BB$

$B \rightarrow aB \mid b$

Soln Step 1

$S' \rightarrow S$

$S \rightarrow BB$



State	Action a	Action b	Action $	Go to S	Go to B
0	S_3	S_4		1	2
1					
2	S_6	S_7			5
3	S_3	S_4			
4					
5					
6	S_6	S_7			9
7					
8					

I_3 = $B \rightarrow a.B,\ a/b$
$B \rightarrow .aB,\ a/b$
$B \rightarrow .b,\ a/b$

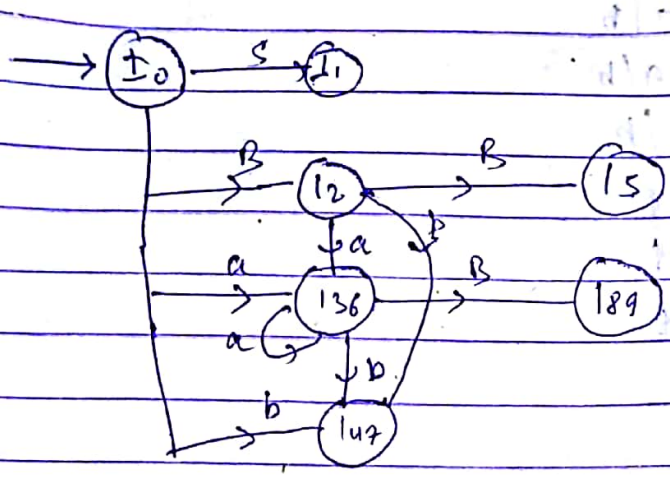
I_{36}

I_6 = $B \rightarrow a.B,\ \$$
$B \rightarrow .aB,\ \$$
$B \rightarrow .b,\ \$$

I_{47}

I_{89}

state	Action				GO TO	
	a	b	~~$ $~~	$	S	B
0	S_{36}	S_{47}			1	2
1	~~S_{36}~~	~~S_{47}~~		accept		
2	S_{36}	S_{47}			5	



D.F.A for LALR.

Q. Construct LALR(1) parsing table for the given grammar

$S \rightarrow Ba \mid bBc \mid dc \mid bda$

$B \rightarrow d.$

as it is CLR.

Q. Difference betn predective parser (top down parser) & shift reduce parser (bottom up parser)

Predictive Parser	Shift reduce Parser
1. It is top down parser (LL parser)	1. It is bottom up parser (LR parser)
2. Stack is used for predicting what is to come	2. Stack show what has been seen so far
3. The stack initially contains start symbol of the grammar	3. Stack is initially empty

Predictive	Shift Reduce
4. I/p tokens are popped off from the stack	4 I/P tokens are pushed on the stack.
5. Left side of production are popped off from the stack	5. Right side of production are popped off from the stack
6. Right side of production are pushed on the stack	6. Left side of productions are pushed on the stack.
7. The stack is empty when the accept state is reached	7. The stack contains the start symbol of the grammar when the accept state is reached
8. There is backtracking problem	8. No such problem of backtracking exist
9. Less efficient	9. More efficient
10 Less powerful	10. More powerful
11. Left recursion problem also exist in predictive parser	11. No left recursion problem because it is bottom up parsing

An operator precedence parser is a comp. program that interprets an operator precedence grammar.

* Operator Precedence Grammar :- The CFG enjoys have the following properties
  (i) No RHS of ε or any production has ε.
  (ii) No 2 non-terminals in the RHS of a production are adjacent.

Eg Consider the following CFG :- E → EAE | (E) | id
A → + | - | * | ↑

E → E+E | E-E | E*E | E↑E | (E) | id

* operator precedence Relation: The precedence relations are only established between the terminals of a grammar and with $ markers present at both end of string, non-terminals are ignored

Relation	Meaning
a <· b	Terminal a has lower precedence than b
a ≐ b	Terminal a has equal precedence with b
a ·> b	Terminal a has higher precedence over terminal b

Q. Show that weather the given grammar is operator precedence grammar or not.

$S \rightarrow SAS \mid a$

$A \rightarrow bSb \mid b$

Sol^n Not ope. prec. gram.

$S \rightarrow SbSbS \mid SbS \mid a$

$A \rightarrow bSb \mid b$

Q. Construct the operator relation table for the given grammar.

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

Sol^n

	+	*	id	$
+	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$
*	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$
id	$\cdot>$	$\cdot>$	–	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	accept

$ lowest precedence

id : highest precedence

$+ \cdot> +$

grammar is ambiguous

* Computation of LEADING() :-

(i) "a" is in LEADING(A) if,

(i) $A \rightarrow a\gamma$, here γ is grammar symbol

b.> $ for all "b" in TRAILING(S) where S is start symbol of grammar.

2) if A → xy and if x, y are terminals then set x ≐ y

3). If A → xBy and if x, y are terminals & B is non terminal then set x ≐ y.

4) if A → xB and if x is a terminal for all "a" in LEADING(B) then set x <. a

5) if A → Bx. & if x is a terminal then for all "b" in TRAILING(B) set b.> x

* Operator Precedence Parsing Algo :-

Input : Precedence table for the grammar, G & I/P string, w

output : Parse tree of I/P string, w if w is in the language of grammar otherwise indicate error.

Method: The method consist of following steps:-

Step1: Compute the operator precedence relation & construct operator precedence table using algorithm. There should be exactly one precedence relation betn 2 terminals a & b

Step2: Initially the stack contains $ and i/p buffer contains i/p string w$. Input pointer points to the first symbol of w$

Step3: Repeat the following steps :-

(a) If top symbol of on the stack is $ & i/p pointer points to the $ than return successfully

(b) else let 'a' be the terminal on the top of the stack and 'b' is the terminal pointed by i/p pointer then

Step1 : Computation of Leading ( ).

LEADING (F) = { (, id }

LEADING (T) = { * } ∪ LEADING (F)
= { *, (, id }

LEADING (E) = { + } ∪ LEADING (T)
= { +, *, (, id }

Step2 : Computation of TRAILING ( )

TRAILING (F) = {), id }

TRAILING (T) = { * } ∪ {), id }
= { *,), id }

TRAILING (E) = { + } ∪ { *,), id }
= { +, *,), id }

Step 3 : Computation of precedence relation table :-

	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

Operator Precedence Relation table.

③ $A \longrightarrow x\,BM$, $A \longrightarrow (E)$

i.e. $(=)$

④ $E \longrightarrow E + T$

TRAILING (+) :> +

{+, *,), id} :> +

+ :> +, * :> +,) :> +, id :> +

T → T * F

TRAILING (T) :> *

{*,), id} :> *

+ :> *,) :> *, id :> *

F → (E)

TRAILING (E) :>)

{+, *,), id} :>)

+ :>), * :>),) :>), id :>)

STACK	INPUT STRING	ACTION
$	id + id * id $	$ <· id, push onto stack, sh
$ <· id	+ id * id $	id :> +, pop from stack, redu
$	+ id * id $	$ <· +, push onto stack, shift
$ <· +	id * id $	+ <· id, push onto stack, shift
$ <· + <· id	* id $	id :> *, pop from stack, reduce
$ <· +	* id $	+ <· *, push onto stack, shift
$ <· + <· *	id $	* <· id, push onto stack, shift
$ <· + <· * <· id	$	id :> $, pop, reduce
$ <· + <· *	$	* :> $, pop, reduce
$ <· +	$	+ :> $, pop reduce
$	$	accept

R → L.

w = *id = id.

step 1 :- Leading (L) = {*, id}.

Leading (R) = {*, id}

Leading (S) = {=, *, id}

Trailing (S) :> $
{=, *, id} :> $

(2) · Not applicable

(3). NA.

(4). S → L=R
= <· Leading (R)
= <· {*, id} $

S → *R.
* <· Leading (R)
* <· {*, id}

(5) S → L=R.
Trailing (L) ·> =.
{*, id} ·> =

Stack	INPUT STRING	ACTION.
$.	*id = id $	$ <· *, push onto stack
$ <· *	id = id $	* <· id, push onto stack
$ <· * <· id	= id $	id ·> =, pop from stack, red
$ <· *	= id $.	* ·> =, pop.
$ *	= id $	$ <· =, push.
$ <· =	id $	= <· id, pop push
$ <· = <· id	$.	id ·> $, pop.
$ <· =	$	= ·> $, pop
$	$	accept.

Algorithm for constructing operator precedence function table :-

Input : A table containing operator precedence relation

Output : Precedence function table.

cycle then no precedence function exist.

steps: If the constructed graph has no cycle, then collect the length of longest paths from the group of f & g's respectively.

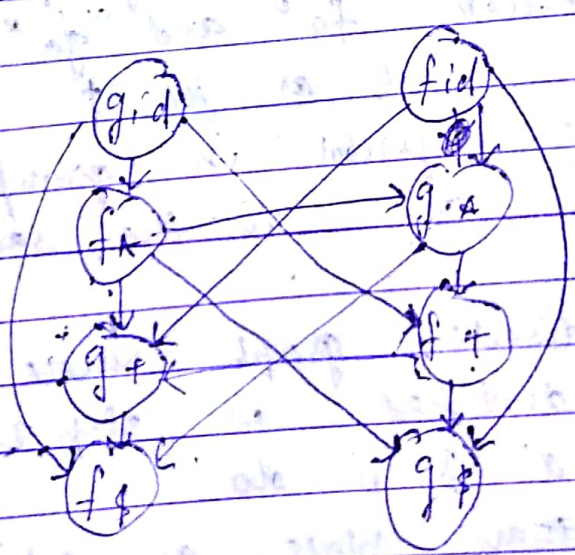
Q. $E \rightarrow E + E \mid E * E \mid id$

Construct a graph representing the precedence function

		id	*	+	$
	id	-	.>	.>	.>
f()	*	<.	.>	.>	.>
	+	<.	<.	.>	.>
	$	<.	<.	<.	-

g()

fid ---> g*



Precedence graph

Precedence function table:

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

fib → g* → f* → g* → f*

f0 * f g0

4 => 1

We have to consider longest path from each node.

Syntatic structure of prog. lang specify the properties of construct. We know that Syntatic structure CFG is used to specify Syntatical structure of prog. language & we add attributes with grammar symbols & semantic rules with production to make the translation of construct easier.

Semantic Rules:-

value for attributes are computed using semantic rules associated with grammar production. The semantic rules also impose an evaluation order on the attributes

eg.

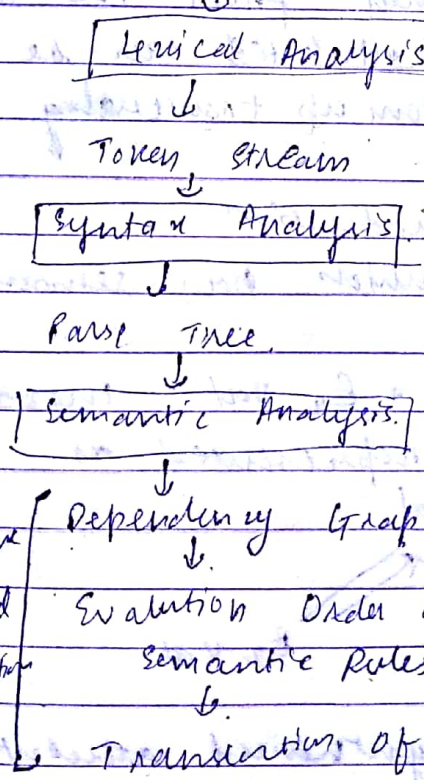
Production	Semantic Rules
E → E + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = (E.val)
F → id	F.val = id.lexval

Desk calculator

when we associate semantic rules with

1. S.D.D. (Syntax directed Definition)
→ It give high level specification for translation
→ Hide many implementation details such as order of evaluation of semantic actions
→ We associate a production rule with a set of semantic actions and we do not say when they will be evaluated

2. SDTS (Syntax directed Translation Scheme) :-
→ It indicates order of evalution of semantic actions associated with a production rule.
→ Translation scheme gives a little bit info about implementation details.



** ATTRIBUTES :- Attributes are associated info. with grammar symbols representing language construct by attaching them to it. An attribute can represent anything that we select i.e. a string, number, type, m/c location, code fragment, etc.

Eg when we are considering an attribute for identifier, it may include name of identifier, scope & type of identifier.

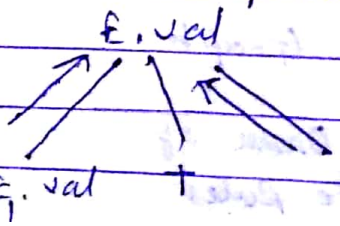
Types of attributes :-

1. synthesized attribute (S-attribute) :- An attribute at a node is synthesized if its value at the parent node can be determined from the attributes of child node. Synthesized attributes can be evaluated by single bottom up traversal of the parse tree.

Eg. If we have the production $E \rightarrow E_1 + E_2$ which has semantic rule as

$E.val = E_1.val + E_2.val$ then it can be represented as

Eg:



(2) Inherited Attribute: An attribute at a node is inherited if its value is computed from attribute values at the sibling or parent of that node in the parse tree

eg Consider production $A \rightarrow XYZ$

with the semantic rule

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

Here Y.val is an inherited attribute

SDD with inherited attributes :-

Production	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$; addtype(id.entry, L.in);
$L \rightarrow id$	addtype(id.entry, L.in);

* Attribute grammars :-

1. S Attributed
   - Uses only synthesized attributes.

2. L - Attributed :
   - Uses both inherited & synthesized attributes. Each inherited attributed is restricted to inherit either from parent or left-sibling only

Eg : $A \rightarrow XYZ$

$Y.a = A.a$, $Y.a = X.a$, $Z.a = Y.a$

$A.a = X.a$:- synthesized

SDD with inherited attributes :-

Production	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = integer$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$; addtype(id.entry, L.in)
$L \rightarrow id$	addtype(id.entry, L.in)

* Attribute grammars :-

S Attributed	L- Attributed :
1. Uses only synthesized attributes.	1. Uses both inherited & synthesized attribute. Each inherited attributed is restricted to inherit either from parent or left-sibling only
Eg : $A \rightarrow XYZ$	inherited
$Y.a = A.a$, $Y.a = X.a$, $Z.a = Y.a$	
$A.a = X.a$: synthesized	
2. Semantic actions are placed	2. Semantic actions are placed

Page No.
Date

* Give the syntax directed translation scheme for the following desk calculator program.

$S \to E \$$
$E \to E + E$
$E \to E * E$
$E \to (E)$
$E \to I$
$I \to I$ digit
$I \to$ digit

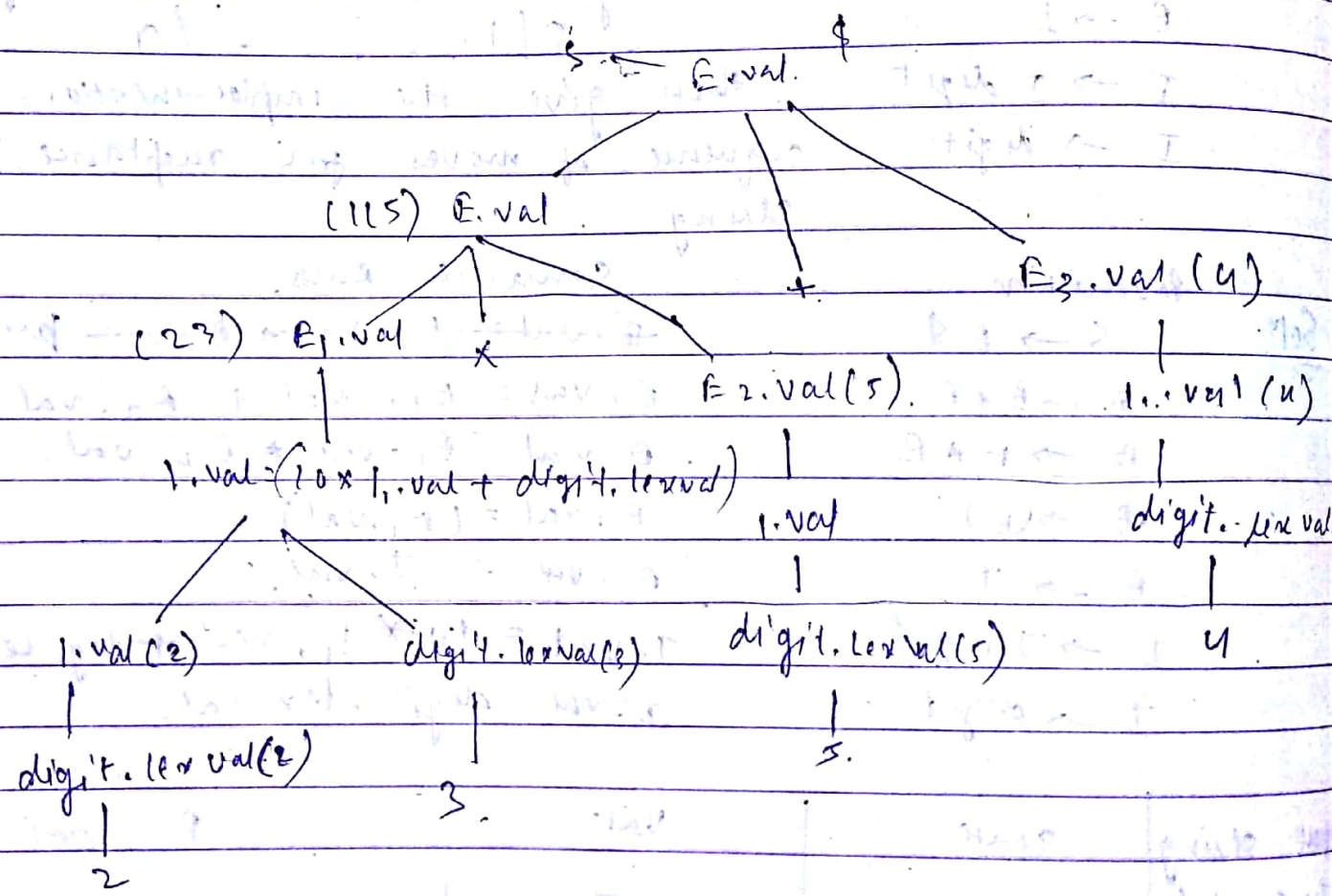
Also construct the syntax directed translation tree for the string $23*5+4\$$ where digit $\to 0/1/\ldots/9$ then give the implementation & sequence of moves for acceptance of string

Production | Semantic Rules

$S \to E \$$ | ~~E.val = E1.val + E2.val~~ print(E.val)
$E \to E + E$ | $E.val = E_1.val + E_2.val$
$E \to E * E$ | $E.val = E_1.val * E_2.val$
$E \to (E)$ | $E.val = (E_1.val)$
$E \to I$ | $E.val = I.val$
$I \to I$ digit | $I.val = 10 * I_1.val + dig.Lexval$
$I \to$ digit | $I.val = digit.lexval$

State	Val.	Production Rules
-	-	-
2	2	$I.val = digit.lexval(2)$
I.val	2	-
I.val 3	2 3	$I.val = 10 * I_2.val + digit_3.lexval$
I.val	23	-

+ 4 $	$E_1.val * I_2.val$	23 5	$E_2.val = I_2.val$
+ 4 $	$E_1.val * E_2.val$	23 5	$E.val = E_1.val * E_2.val$
+ 4 $	E.val	115	–
4 $	E.val +	115	–
$	E.val + 4	115 4	I.val = digit.lexval



$S \rightarrow L.L \mid L$ Use synthesized attribute to determine $S.val$.

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$.

Determine $S.val$ with a SDD in which the only synthesized value of B is 'c' giving the contribution of the bit generated by B to final value. For eg. the contribution of 1st and last bit in 101.101 to value 5.625 is 4 & .125.

Sol^n

Productions	Semantic Rules.
$S \rightarrow L.L$	$S.val := L_1.val + L_2.val$
$S \rightarrow L$	$S.val = L.val$
$L \rightarrow LB$	

0 1	L.val . B.val	S . 1	L.val = B.val * C
0 1	L.val . B L.val	S . 0 . 5	—
0 1	L.val . L.val 0	S . 0 . 5 0	B.val = 0
↑ 1	L.val . L.val B.val	S . 0 . 5 0	L.val = L1.val + B.val * C
. 1	L.val . L.val	S . 0 . 5 .	—
1	L.val . L.val 1	S 0 . 5 1	B.val = 1
↑ —	L.val . L.val B.val	S 0 . 5 1	L.val = L1.val + B.val
-	L.val . L.val	S 0 . 6 2 5	S.val = L.val + L.val

(1)

2nd method

Productions

$S \to L.L$ { S.dval = L_1.dval + L_2.dval / $2^{L_2.l}$ }

$S \to L$ { S.c = L.c ; S.dval = L.dval }

$L \to LB$ { L.c = L_1.c + B.c ; L.dval = L_1.dval + B.dval * 2 }

$L \to B$ { L.c = B.c ; L.dval = B.dval * 2 }

$B \to 0$ { B.c = 1 ; B.dval = 0 }

$B \to 1$ { B.c = 1 ; B.dval = 1 }

Input string	state	val	Semantic Rules
101.101	1	1	B.c = 1 ; B.dval = 1
01.101	B.c	1	L.c = B.c ; L.dval = B.dval
	B.dval	1	
01.101	~~B.c~~ L.c	1	—
	L.dval	4	—
01.101	L.c 0	1 0	B.c = 1 ; B.dval = 0
↑	L.dval 0	4 0	
1.101	L.c B.c	1 1	L.c = L_1.c + B.c
	L.dval B.dval	4 0	L.dval = L_1.dval + B.dval

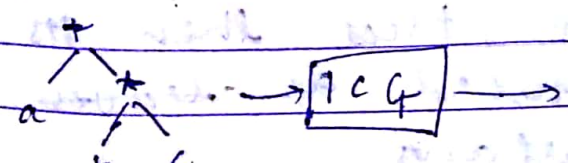
	L.dval	4.	4 (4+0*2')	
10	L.c	1	2 4	B.c = 1
	L.dval	1	4 1	B.dval = 1
10	L.c B.c		2 1	L.c = L1.c + B.c
	L.dval B.dval		4 1	L.dval = L1.dval + B.dval * 2
10	L.c		3	—
	L.dval		5	

* Intermediate code generation :-

A compiler can translate a source code into a file consist of machine code or assembly code in a syntax directed manner. The prog. which is used for generating intermediate code is called intermediate code generator. The I/P to the code generator consist of a parse tree and the tree is converted into linear sequence of instruction in an intermediate lang. such as 3 address code.

This can be represented as :-

Parse Tree → [ICG] → Linear sequence of instructions.



(i) Expresses high level structure of program
(ii) Easy to generate from I/P programs.
(iii) Closure to the source language.
Eg. Syntax Trees & DAG's (Directed Acyclic Graphs)

(ii) Low level intermediate Representation :-
- Expresses low level structure of a programs.
- Closure to the target machine
- Generation from I/P prog. they involve some work.
Eg. 3AC, Post fix Notation.

* SYNTAX TREES :-
It is a compact form of a parse tree that represents the ~~tight~~ hierarchical structure of the program. Here nodes represent operators & the children of a node represent what it operates on.

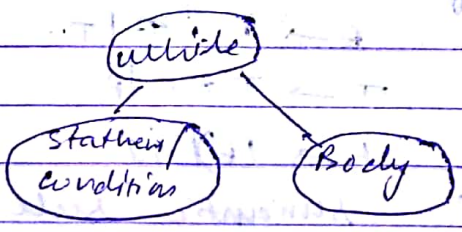
Structure of syntax trees :-
In case of representing an arithmetic expression in the form of syntax trees then its structure can be described as ~~leaves~~
for { leaves : identifiers or constants

children of internal nodes are its operands.

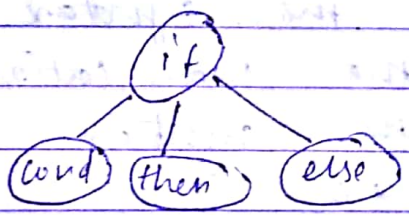
For representing statements in the form of syntax tree their structure will be as follows:-

1. A node's label indicate what kind of statement it is eg. while (condition) { }
2. The children of a node corresponds to the component of the statement.

eg while (condition)
{
}



Eg if cond
then
else - - -



* Construction of Syntax tree :-

pointer to the symbol table entry for the identifier.

3) mkleaf (number, value) : It creates a no. node with label number and with a field containing value of the number.

Q. Construct the SDD (syntax directed defn) of the following grammar.

$E \rightarrow E + T \mid T$

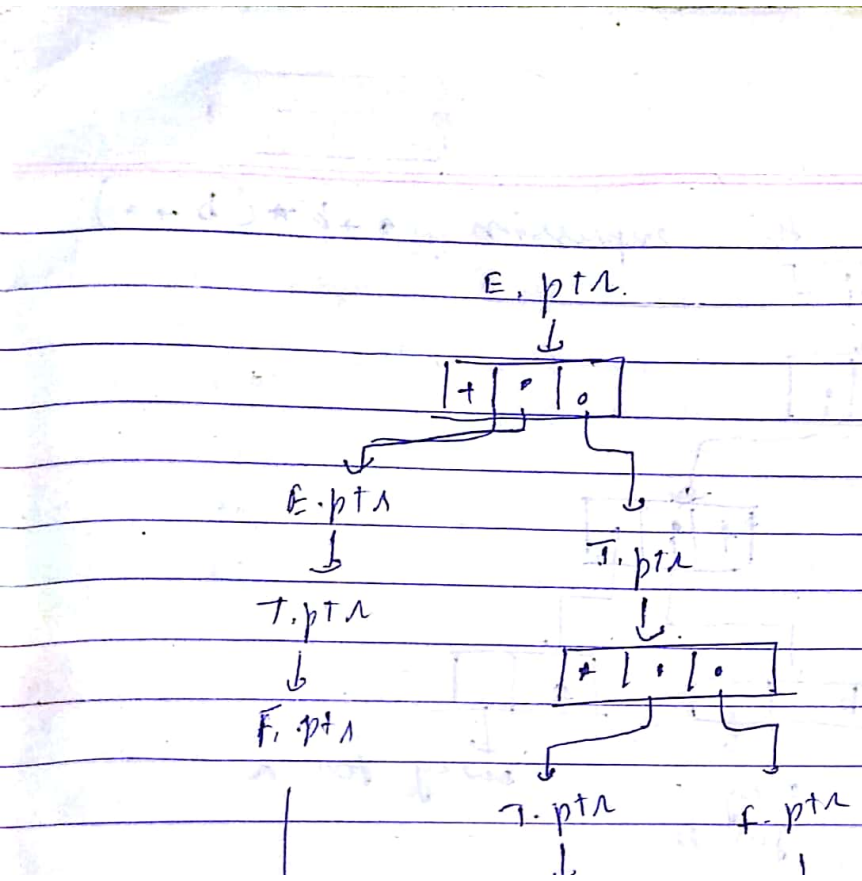
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

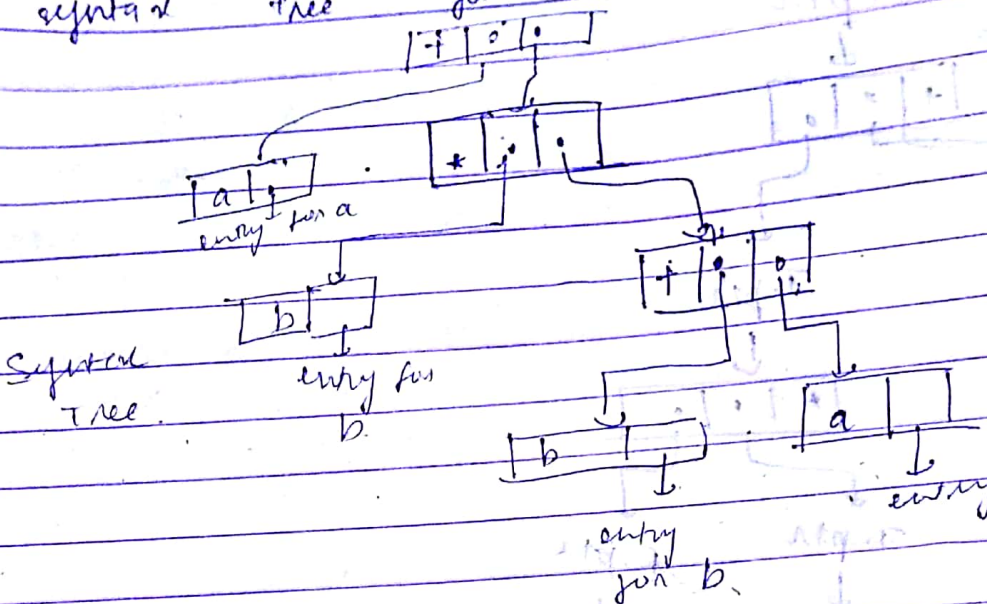
Associate semantic rule with the productions for construction of the syntax tree for an expression using the translation scheme. & also construct the syntax tree for the expression $a + b * c$

Sol.

Productions	Semantic Rules
$E \rightarrow E + T$	E.ptr = mknode (+, E.ptr, T.ptr)
$E \rightarrow T$	E.ptr = T.ptr
$T \rightarrow T * F$	T.ptr = mknode (*, T.ptr, F.ptr)
$T \rightarrow F$	T.ptr = F.ptr
$F \rightarrow (E)$	F.ptr = (E.ptr)
$F \rightarrow id$	F.ptr = mkleaf (id, id.entry)



Q Find syntax tree for the expression a + b * (b + a)



TAC :- Three Address Code :-

- It is one of the popular type of intermediate lang
- The TAC statement is of the form

A := B OP C

where A, B, C are operands & OP are operators

- The given representation consist of atmost 3 address field :-

to 2 address field for operands & 1 address field for result. ; It is called 3 address code

Advantages of TAC :-

2. It allows the rearrangement of intermediate code in a convenient manner during code optimization

3. It is most appropriate intermediate lang. for most of the optimization done in compiler such as common sub expression elimination & algebraic simplification.

4. Generate 3AE of the following program fragments

(i) if (a<b), x = y + z
else p = q + r.

i) if (a < b), goto (i + 2)
i+1) goto i+5
i+2) t_1 = y + z
i+3) x = t_1
goto i+7.
i+4) ~~t_2 = q + r~~

5 while (a < b)

i) $t_1 = y + 2$
i+1) $x = t_1$
i+2) if (a < b) goto (i)
i+3) STOP.

i) $t_1 = y + 2$
i+1) $x = t_1$
i+2) if (a < b) goto (i+4)
i+3) goto (i)
i+4) STOP.

(5) for (i=1; i<=20; i++)
$x = y + 2$

i) i = 1
i+1) if (i <= 20) goto i+6
i+2) goto (i+8)
i+3) $t_1 = i + 1$
i+4) $i = t_1$
i+5) goto (i+1)
i+6) $t_2 = y + 2$
i+7) $x = t_2$
i+8) goto (i+3).

6. switch (i+j)
{
case 1 : $x = y + 2$; break;
default : $p = q + r$; break;
case 2 : $u = v + w$; break;
}

Sol" i) $t_1 = i + j$

i+1) goto (i+11)

i+2) $t_2 = y + 2$

i+3) $x = t_2$

i+4) goto (i+14)

i+5) $t_3 = q + k$

i+6) $p = t_3$

i+7) goto (i+14)

i+8) $t_4 = v + w$

i+9) $u = t_4$

i+10) goto (i+14)

i+11) if $t_1 = 1$, goto (i+2)

i+12) if $t_1 = 2$, goto

i+13) goto (i+5)

i+14) STOP

7) while (A < C and B > D) do

if A = 1 then C = C + 1

else

while A <= D do

A = A + 3

i) if (A < C) ~~& B > D~~ goto (i+2)

i+1) goto (i+4)

i+9) if (A <= D) goto (i+11)
i+10) goto i
i+11) t2 = A + 3
i+12) A = t2
i+13) goto (i+9)
i+14) ~~Stop~~. STOP.

* Addressing Array elements :-

~~address~~ addr (a[i]) = Base(a) + w (i - lower bound)

a[3] = 100 + 2(3-0)
= 106.

addr (a[i]) Base(a) + w * i - w * lb

addr (a[i]) = w * i + (Base(a) - w * lb)

where (= Base(a) - w * lb.

Q write 3AC :-

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

1) i = 1
2) if (i <= 10) goto 4
3) goto 9
4) t1 = w * i
5) t2 = Base(a) - w
6) t3 = t2[t1]
7) t3 = i
8 goto 2
9 STOP

Q. sum = 0
for (i = 1; i <= 20; i++)
sum = sum + a[i] + b[i]; with
w = 4 bytes per word.

1) sum = 0
2) i = 1
3) if (i <= 20) goto 6
4) goto 14
5) i = i + 1 goto 3
6) $t_1 = 4 * i$
7) t_2 = Base(a) - 4.
8) $t_3 = t_2[t_1]$
9) t_4 = Base(b) - 4
10) $t_5 = t_4[t_1]$
11) $t_6 = t_5 + t_3$
12) sum = sum + t_6.
13) goto 5

2-D array is normally stored in 2 forms
1. Row major form
2. Column major form

Row major form
loc(A[i, j]) = Base(A) + w * [N_c(i - LB_2) + (j - LB_2)]

loc(A[i, j]) = Base(A) + w * [i * N_c - N_c * LB_2 + (j - LB_2)]

Q Generate 3AC of the program where a & b are array of size 20 x 20. & there are 4 bytes per word.

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

1) add = 0
2) i = 1
3) j = 1
4) $t_1 = i * 20$
5) $t_2 = t_1 + j$
6) $t_3 = t_2 * 4$
7) $t_4 = Base(a) - 84$
8) $t_5 = t_4[t_3]$ // a[i,j]
9) $t_6 = j * 20$
10) $t_7 = t_6 + i$
11) $t_8 = t_7 * 4$
12) $t_9 = Base(b) - 84$

13) $t_{10} = t_9[t_8]$ // b[i,j]
14) $t_{11} = t_5 * t_{10}$
15 addd = addd + t_{11}
16) ~~i = i + 1~~ i = i + 1
17) j = j + 1
18 if i <= 20 go to 20
19 go to 21
20 if j <= 20 goto 4
21 stop

- Generate 3AC c[a[i,j]] = b[i,j] + c[a[i,j]] + d[i+j] given dimension of a & b are 30 x 40 & that of c & d is 20. Assume subscript values are started with 1 for all array dimensions & each array element req. 4 bytes.

Soln
=
1) $t_1 = i * 40$
2) $t_2 = t_1 + j$
3) $t_3 = t_2 * 4$
4) $t_4 = Base(a) - 164$
5) $t_5 = t_4[t_3]$ // a[i,j]
6) $t_6 = Base(b) - 164$
7) $t_7 = t_6[t_3]$ // b[i,j]
8) $t_8 = 4 * t_5$
9) $t_9 = Base(c) - 4$
10) $t_{10} = t_9[t_8]$ // c[a[i,j]]
11) $t_{11} = i + j$
12) $t_{12} = 4 * t_{11}$
13) $t_{13} = Base(d) - 4$ (w.r.t.b)

15) $t_{15} = t_7 + t_{10}$

16) $t_{16} = t_{15} + t_{14}$

17) $t_9[t_3] = t_{16}$

* Implementation of TAC statement :-

A TAC statement can be implemented as a record having field for operators & operands We describe the following ways to implement TAC

1) Quadruple : A quadruple is a record structure having 4 fields such that-

(i) first field represent the operator

(ii) second " " first argument

(iii) 3rd " " 2nd " "

(iv) 4th " " the result obtained after applying operator OP to argument 1 & argument 2.

OP	ARG 1	ARG 2	Result
+	b	c	a

$a = b + c$

2) Triple :- A triple is a record structure having 3 fields such that :-

(i) 1st field represent an operator

2nd " " argument 1

3rd " " 2

3) Indirect Triple :- An indirect triple representation of TAC in which a additional array is used to list the operation to the triple in desired order
∴ An indirect triple representation consist of listing the pointers to triples rather than listing the triples themselves

Q write quadruple, triple & indirect triple of given expression

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

$t_1 = a+b$

$t_2 = -t_1$

UNIT : 5

Basic Blocks

A basic block b of a program P is a sequence of consecutive prog. instructions $I_1, I_2, \ldots I_m$ such that

1. flow of control enters at the beginning i.e. at I_1 of in basic block, b.
2. flow of control leaves at the end i.e. I_m in block b

Algorithm : Partition into basic blocks ()

Input : sequence of TAC

Output : list of basic blocks with each TAC in exactly one block

Method : i. We first determine the set of leaders, the first statement of basic block. The rules for determining leader are as follows :-

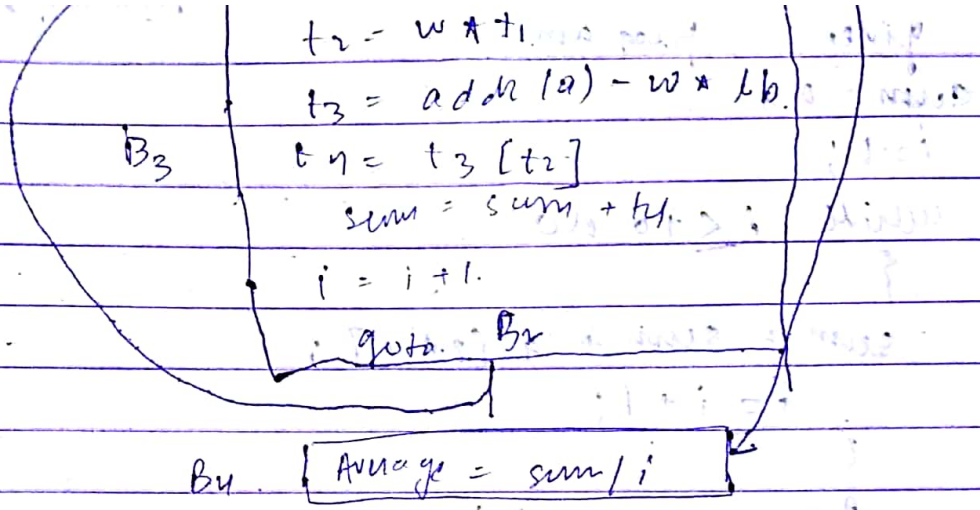
(i) The first statement is a leader.

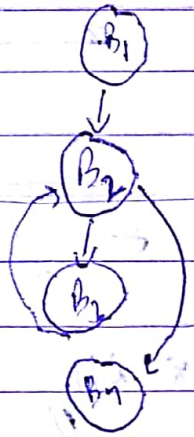
(ii) Any statement that is target of conditional or unconditional goto is a leader

(iii) Any statement that immediately follows goto or conditional goto is a leader

Q5 Construct basic block & draw flow graph for given program:-

```
sum = 0;
i = 1;
while i <= 10 do
{
sum := sum + a[2*i];
i = i + 1;
}
Average = sum / i;
```





flow graph.

CT

M Imp

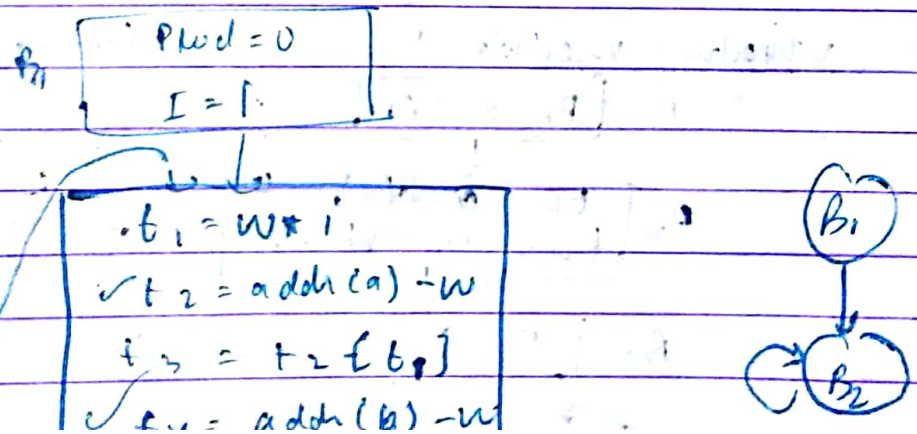
Q

```
begin
    prod = 0
    I = 1.
do
```

Q Construct basic block and draw flow graph of given program and also optimize the code by applying various loop optimization technique.

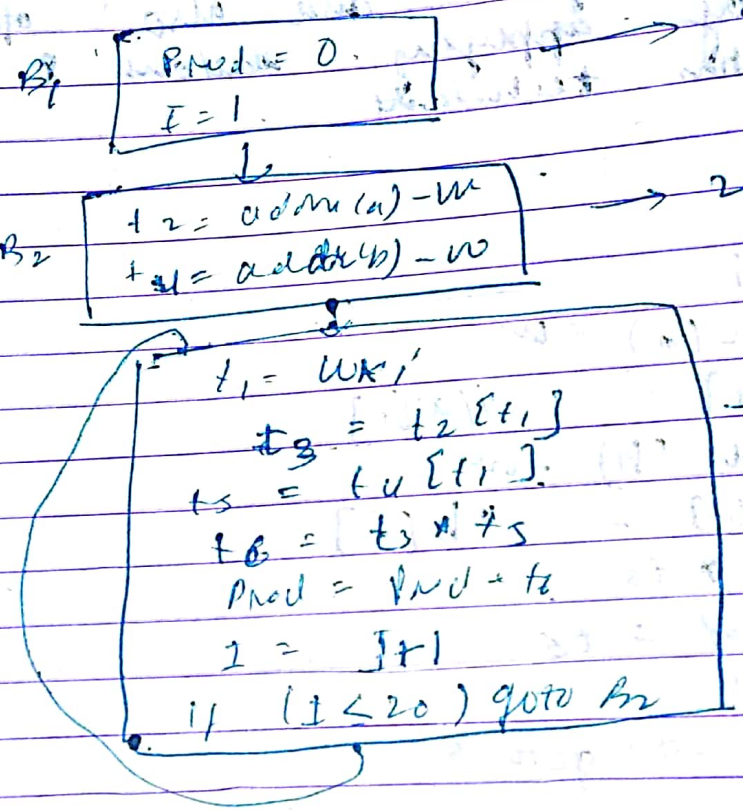
Soln

1) Prod = 0 ——— L
2) I = 1
3) t_1 = w * i ——— L2
4) t_2 = addr(a) - w
5) t_3 = $t_2[t_1]$ — //a[i]
6) t_4 = addr(b) - w
7) t_5 = $t_4[t_1]$ — //b[i]
8) t_6 = t_3 * t_5
9) Prod = Prod + t_6
10) I = I + 1
11) if (I ≤ 20) goto 3

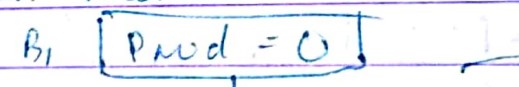


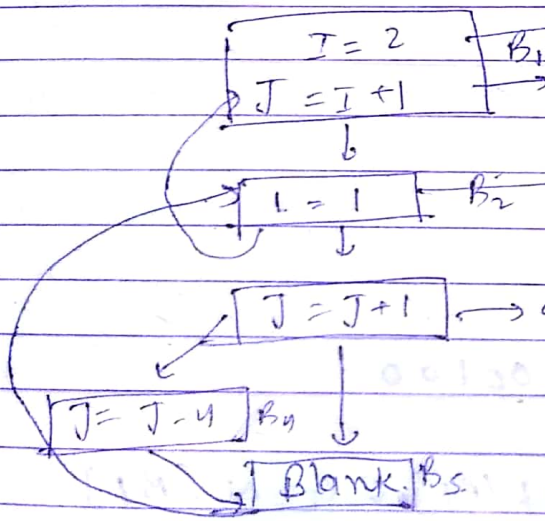
Codes

Code optimization techniques :-



Induction variable method :- (Assuming w = 4)





Compute GEN() for B
kill for each block
b) Compute in B out for reaching definition
c) Compute UD chains

		d_1 d_2 d_3 d_4 d_5		d_1 d_2 d_3 d_4 d_5
B_1	$\{d_1, d_2\}$	1 1 0 0 0	$\{d_3, d_4, d_5\}$	0 0 1 1 1
B_2	$\{d_3\}$	0 0 1 0 0	$\{d_1\}$	1 0 0 0 0
B_3	$\{d_4\}$	0 0 0 1 0	$\{d_2, d_5\}$	0 1 0 0 1
B_4	$\{d_5\}$	0 0 0 0 1	$\{d_1, d_4\}$	0 1 0 1 0
B_5	ϕ	ϕ	ϕ	ϕ

Pass 1 : $IN[B] = \phi$

$OUT[B] = GEN[B]$

Block B	IN [B]	OUT [B] (gen w GEN)
B1	0 0 0 0 0	1 1 0 0 0
B2	0 0 0 0 0	0 0 1 0 0 ✓
B3	0 0 0 0 0	0 0 0 1 0
B4	0 0 0 0 0	0 0 0 0 1
B5	0 0 0 0 0	0 0 0 0 0

Pass 2 :- $IN[B1] = OUT[B_2] = 00100$

$OUT[B_1] = IN[B_1] - KILL[B_1] \cup GEN[B_1]$

$= IN[B_1] \wedge \neg KILL[B_1] \vee GEN[B_1]$

$= 00100 \wedge 11000 \vee 11000$

$= 00000 \vee 11000$

$= 11000.$

$IN[B_2] = OUT[B_1] \wedge OUT[B_5]$

$= 11000 \wedge 00000 = 00000$

wp

Q. Consider the following expression

$w = (a-b) + (a-c) + (a-c)$

Translate into 3AC

Apply code genration algo on it

3AC

$t_1 = a - b$

$t_2 = a - c$

$t_3 = t_1 + t_2$

$t_4 = t_3 + t_2$

$w = t_4$

Sol^n Statements	Location L	Instruction Generated	Register Description	Addr. Description
$t_1 = a - b$	R_0	MOV a, R_0	R_0 contains t_1	t_1 is in R_0
		SUB b, R_0		
		$R_0 = R_0 - b$.		
$t_2 = a - c$	R_1	MOV a, R_1	R_1 contains t_2	t_2 is in R_1
		SUB c, R_1		
$t_3 = t_1 + t_2$	R_0	ADD R_1, R_0	R_0 contains t_3	t_3 is in R_0
		$R_0 = R_0 + R_1$.		
$t_4 = t_3 + t_2$	R_1	ADD R_0, R_1	R_1 contains t_4	t_4 is in R_1
		$R_1 = R_0 + R_1$		
$w = t_4$		MOV R_1, w		