30/1/2023

Intro to Compiler →

↳ Assembler → converts assembly language code to machine level language.

Limitations

* depends on machine architecture
* remember n no. of instructions is difficult
* complexity is increased.
* code length is large.

Hence people started using High Level lang. ie C, C++, Java

↳ Compiler → converts source program to machine level program / target program.

* Program must be error free.

The same work can be done by Interpreter also

↳ Linux uses interpreter instead of Compiler.

↳ Working of compiler & interpreter

eg:-

```
void main()
{
    int a
    a = 5 + 6
}
```

Line 1
Line 2
3
4
5
.
10

* Compiler compiles entire source prog & list the errors.
* Interpreter will first report error in a particular line repeatedly. (Step by step execution).

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

◦ Interpreter
◦ Assembler

Source prog. → [Compiler] → Target Prog.

Input → [Target Prog] → Output

↳ Interpreter → An interpreter is another kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in source program on inputs supplied by user.

Source Prog. → [Interpreter] → Output
Input →

Diff. between Compiler & Interpreter

Compiler	Interpreter
◦) A compiler translates entire source code in a single run.	◦) An interpreter translates entire source code line by line.
◦) It consumes less time; faster than interpreter	◦) It consumes more time than Compiler.
◦) It is more efficient	◦) It is less efficient
◦) CPU utilization is more.	◦) CPU utilization is less.
◦) Both syntactic & semantic errors can be changed.	◦) Only syntactic errors are checke
◦) It is not flexible	◦) It is flexible.
◦) The localization of errors is difficult.	◦) The localization of error is ea
◦) A presence of error can cause whole prog. to be re-organized.	◦) A presence of error can cause o a part of prog. to be re-organi
◦) eg:- C, C++, JAVA, C#	◦) JAVA, Python, Perl, Matlab, Ruby

Phases of a Compiler

1.) Character Stream → [Lexical Analyzer] → Tokens. (Lexemes).

Reads char string to tokens. Lex is a tool. By using Lex Tool we build lexical Analyzer.

eg:- [3] Sum = a + b; → character stream

↓

[L. A]

↓

⟨id,1⟩ = ⟨id,2⟩ ⟨+⟩ ⟨id,3⟩ → tokens

→ Lexical Analysis → The first phase of a compiler is called Lexical Analysis or Scanning.

1) The lexical analyzer reads stream of characters making up the source program & groups character into meaningful sequence called lexemes. For each lexeme, lexical analyzer produces as output a token of the form ⟨token name, attribute value⟩

Symbol Table is already in Compiler. When Compiler scans, it stores the data in a symbol table according to position.

Position	Name
1	Sum
2	a
3	b

2) Lexical Analyzer removes the comments/spaces from the source code.

2nd Phase

→ Syntax Analyzer

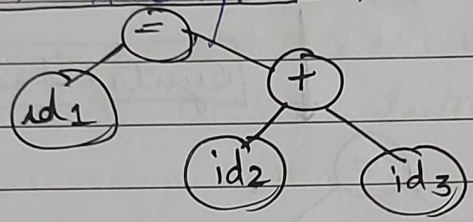
<id,1> = <id,2> + <id,3>

↓

Syntax Analyzer

They have their own grammar. They write their own syntax. For eg:- sum = a + b ; it is translated to E → E + E.

- It will check syntax of the program.
- Syntax Analyzer will generate a parse Tree,



(The root node contains operators & leaf node contains operands).

It is also known as "Parser"

↳ The second phase of the compiler is Syntax Analysis or Parsing. The parser uses the first components of tokens produced by lexical Analyzer to create tree like intermediate representation that depicts the grammatical structure of the token stream.

3rd Phase

Semantic Analyzer → check program whether it is semantically correct or not

- Type Mismatch.
- Array Indexing

→ It uses the syntax tree & info. in symbol ta to check source program for semantic consistency with language definition. An important part of this is type checking, where compiler checks that each operator has matching operands (same datatype)

For eg:- Many programming lang. definitions require an array index to be integer. Compiler must report

an error if a floating point number is used to index an ~~an~~ array.

eg →

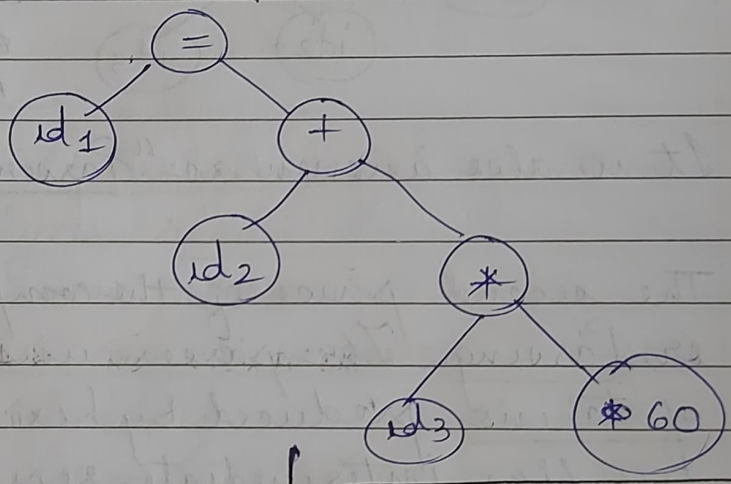
position = Initial + rate * 60

position }
Initial } float
Rate

Lexical Analyzer

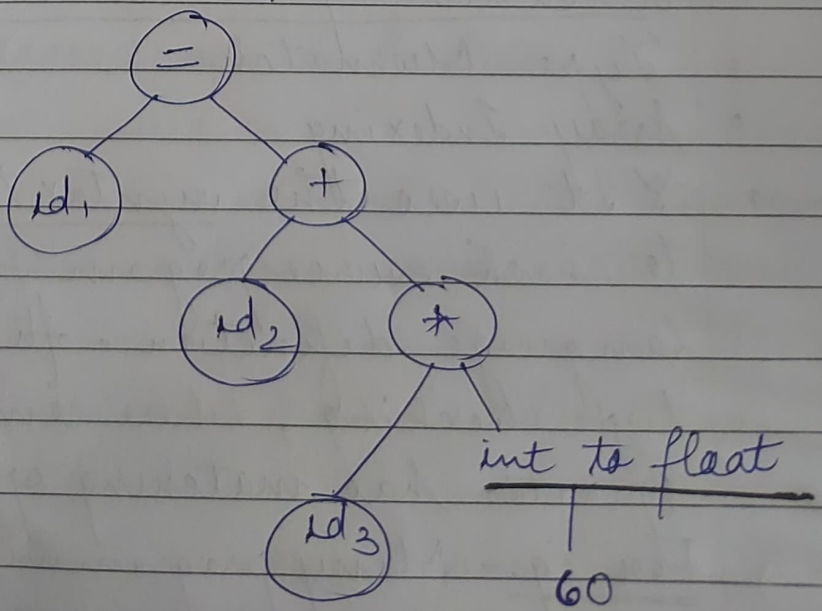
<id,1> = <id,2> + <id,3> * 60

Syntax Analyzer



Semantic Analyzer

Convert integer type to "float"



__/__/____

1.2.22

4th Phase Intermediate Code Generation

Imp Three addressing coding. Right handside we will have 1 operator or 2 operands.

↳ In the process of translating source prog. to target code, a compiler may construct one or more intermediate representation

eg: position = initial + rate * 60.

↓ Intermediate Code Generation

t_1 = int to float (60)
t_2 = id3 * t_1
t_3 = id2 + t_2
id1 = t_3

Q) Why only 1 operator in each line?

→ The ultimate machine level code will have either 2 operands or 1 operator. (Microprocessor coding)

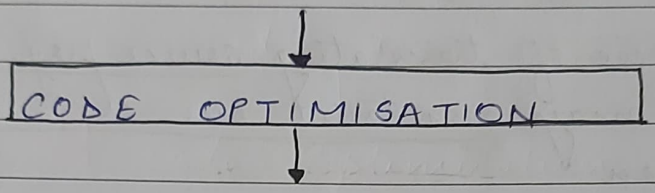
For eg: ADD R1, R2
1 2

In every instruction we have only 2 operands so in ICG we have this rule.

__/__/____

3.2.22

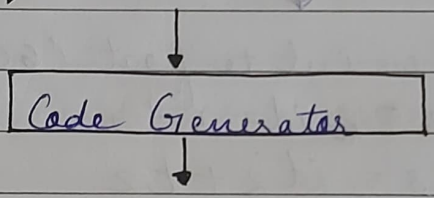
5th Phase **Code Optimisation** → The machine independent code optimisation phase attempts to improve the intermediate code so that faster target code will result.



$t_1 = id\,3 * 60$

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

hase 6 **Code Generator** →




```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

- Errors generated by each phase of compiler

__/__/____

f) pass the statement area = 3.14 × r² into compiler.
(float, int)

→ area = 3.14 * r * r.

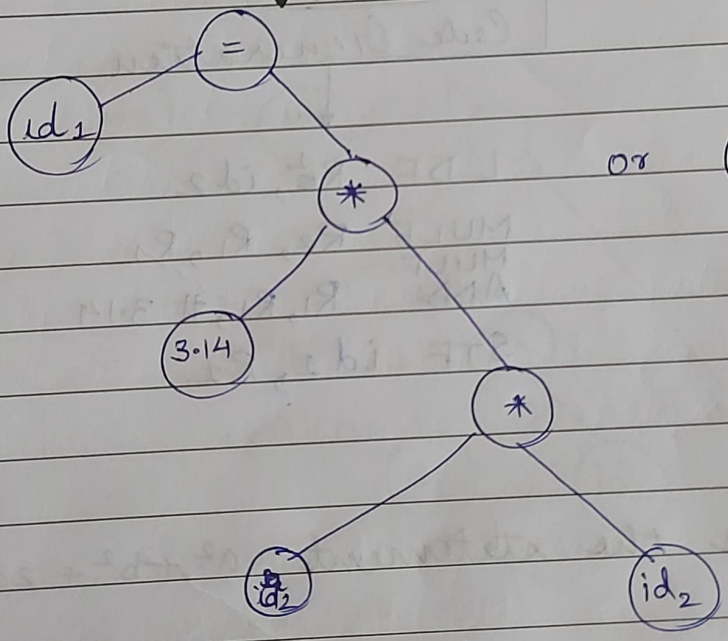
a) LA — Lexical Analyzer.

<id,1> <=> <3.14> <*> <id,2> <*> <id,2>

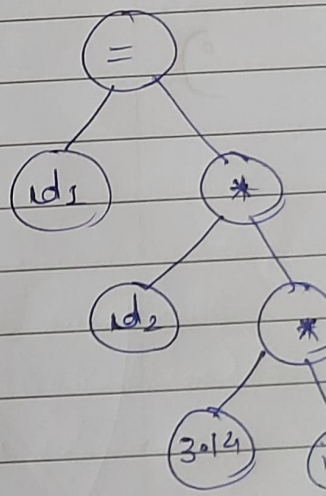
Symbol table

1	area
2	r

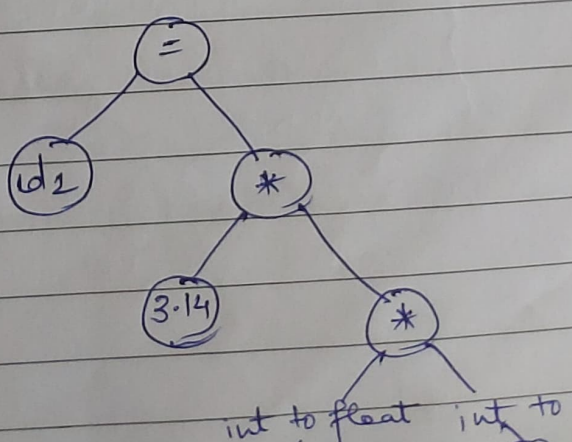
b) Syntax Analyzer



or



c) Semantic Analyzer



c)

Intermediate Code Generation

$t_1 = int\ to\ float\ (id_2)$

$t_2 = t_1 * t_1$

$t_3 = 3.14 * t_2$

$id_1 = t_3$

c)

Code Optimisation

$t_2 = id_2 * id_2$

$t_3 = 3.14 * t_2$

$id_1 = t_3$

c)

Code Generation

```
LDF   R1, id2
MULF  R2, R1, R1
MULF  R1, R1, #3.14
STF   id1, R1
```

Q2) pass the statement $a^2 + b^2 + 2ab$ through compiler?

→ ans = $a^2 + b^2 + 2ab$

↓

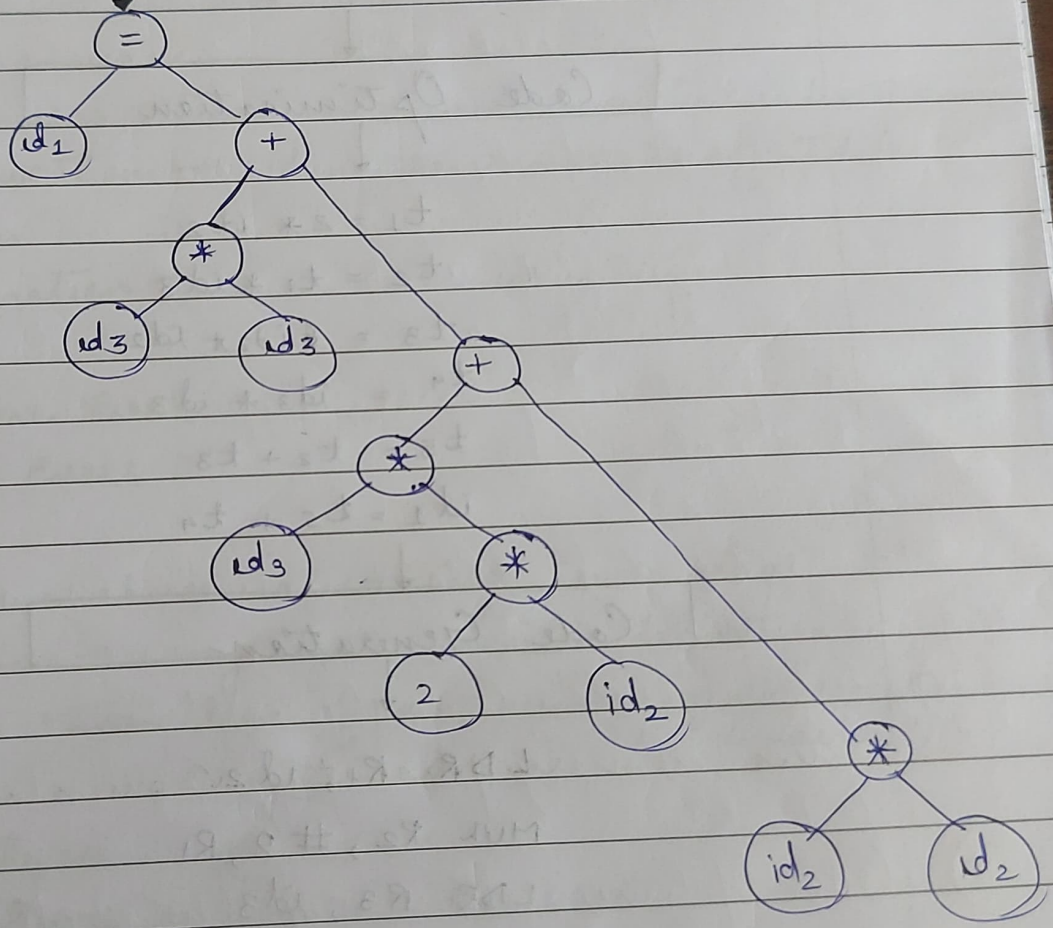
o) Lexical Analysis

↓

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

↓

Syntax Analysis



↓

Semantic Analysis

↓

same (assuming all integer)

Intermediate Code Generation

$t_1 = 2 * id_2$

$t_2 = t_1 * id_3$

$t_3 = id_2 * id_2$

$t_4 = id_3 * id_3$

$t_5 = t_2 + t_3$

$t_6 = t_5 + t_4$

$id_1 = t_6$

Code Optimisation

$t_1 = 2 * id_2$

$t_2 = t_1 * id_3$

$t_3 = id_2 * id_2$

$t_4 = id_3 * id_3$

$t_5 = t_2 + t_3$

$id_1 = t_5 + t_4$

Code Generation

```
LDR R1, id2
MUL R2, #2, R1
LDR R3, id3
MUL R4, R2, R3
MUL R5, R1, R1
MUL R6, R3, R3
ADD R7, R4, R5
ADD R7, R7, R6
STR id1, R7
```

__/__/____

Errors generated by different phases of compiler.

i) Lexical Analysis → errors occur in operation of tokens
eg:- incorrect identifier / variable declaration
ii) Syntax Analysis → errors occurred during construction of syntax tree.
eg:- missing parantheses
iii) Semantic Analysis → When compiler constructs that have high syntactical structure but no meaning.
- During type conversion.

eg:- type mismatch, undeclared variable, non-int index of array.
iv) Code Optimization → When the result is affected by optimization
v) Code Generation → When code is missing.

- Compile Time Error
- Run Time Errors

6.2.22

Error → Some abnormal things in a prog
i) Compile Time → wrong syntax; variable declaration
ii) Run Time → wrong logic of prog; memory issue; exception handl "divide by 0" /0

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer

} Compile Time Error

- Intermediate Code Generation
- Code Optimization ← Machine Dependent / Machine Independent
- Code Generation

} Run Time Err

__/__/____

Errors Faced at each step →

i) Lexical Analyzer → Compile Time Error;
If tokens are not received; so error.

eg:- ↳ int a; No Error.
↳ Int a; Error: Not a Datatype
↳ int $a; Error: $ special symbol

ii) Syntax Analyzer → Compile Time Error.
eg if we miss semicolon or comma b/w operands

↳ int a b Error

iii) Semantic Analyzer → Compile Time Error.

```
float a, b
b = a + 5
```

a → float
b → int
} Error type mismatch

↳ ar[2.2] → float value in error

RUNTIME ERROR

iv) Intermediate Code Generation → no memory / less memory present / segmentation fault / divide by '0'

v) Code Optimisation → If the o/p gets affected by optimising code.

vi) Symbol Table → if the variable is entered two times / duplicacy of variable

vii) Code Generation → Variable is used but not ~~dicted~~ declared.

Cousins of Compiler

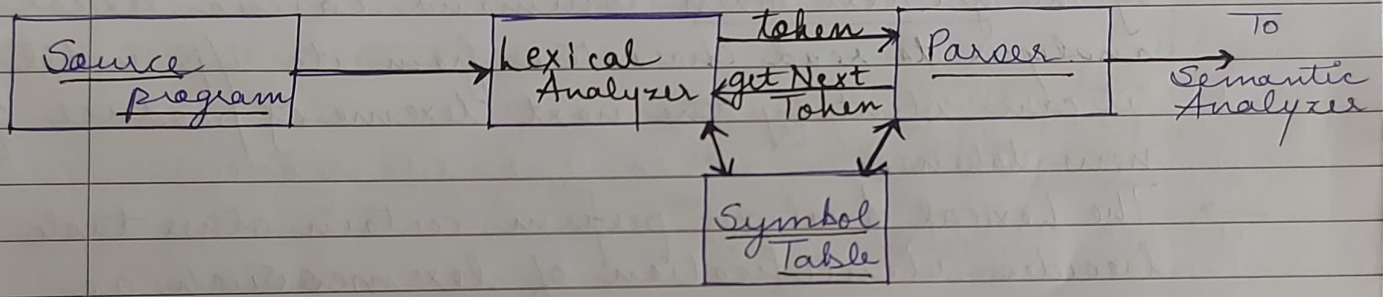
__/__/____

Cousins of Compiler

- Assembler
- Linker
- Loader
- Language Processor

Lexical Analyzer →

The role of the lexical analyzer



→ ; int a = 10, b = 5, c ;

→ 2. c = a + b ;

→ Reading first line, generating tokens; then tokens are passed to parser, scanning prog. line by line. Symbol Table is storing the variables a, b, c & method names.

1	a	10
2	b	5
3	c	

→ If I put commands /* in and unable to close command, then this error is identified by Lexical Analyzer. It will not generate any tokens for commands. It will be eliminating line space. It converts char stream to tokens.

__/__/____

As

↳ the first phase of compiler, is to read i/p characters of source program, group them into lexemes & produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for "Syntax Analysis".

The interaction b/w lexical analyzer & the parser are shown in the above figure. Commonly the interaction is implemented by having the parser call, the lexical analyzer. The call suggested by the "getNextToken" command causes lexical analyzer to read characters from its i/p until it can identify the next lexeme & produce for it next token.

→ The lexical analyzer perform certain other tasks besides identification of lexemes such as

* Stripping Out Commands and Whitespace

8.2.23

INPUT BUFFERING →

9 Write a LEX program to identify/count words & digits words. numbers.

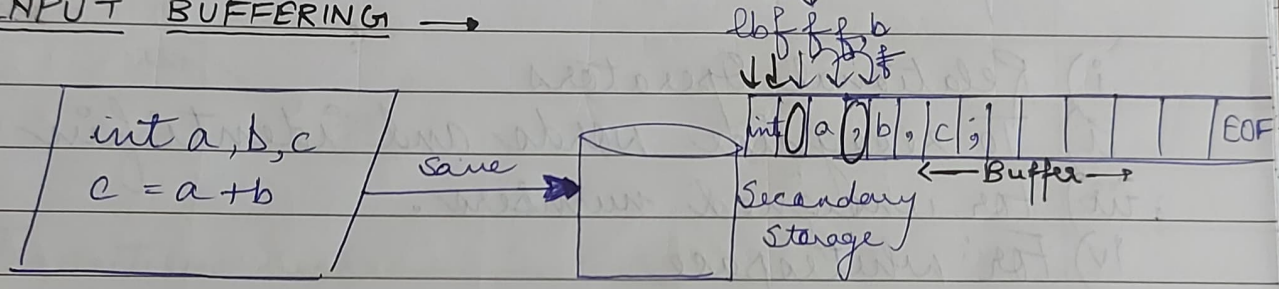
→

```
%{
  #include <stdio.h>
  int wc=0, n=0;
%}
%%
[a-zA-Z][a-zA-Z0-9]*   (0 or more times)   {wc++;}
[0-9][0-9]*            {n++;}
%%
int yywrap() {
    return 1; }
```

```
int main () {
printf ("Enter the sentence :");
yylex();
printf("Total Digits %d \n", n);
printf("Total Words %d \n", wc);
return 0;
}
```

yylex() → yyin() → reading i/p from default i/p device
→ yyout() → write the o/p on default o/p device

INPUT BUFFERING →



Lexical analyzer reads the first line & store it in a buffer. The size of buffer is 4096 bytes. In the buffer the last field is "EOF". Then these are converted into tokens using 2 pointers.

- lexeme begin } To generate tokens we use 2 pointers
- forward

Initially these 2 point at same position

- Forward is going to next position.

→ Tokens are generated on basis of seperator.
→ <int> as a whole is a token. As space is seen,
→ it is a seperator so <int> is a token.
→ a is generated as a token & so on.
~~With~~ lexeme is a matched pattern <int>

Single Buffing { Lexical Analyzer is using 20% - 30% of the time to compile because read line by line & store it in buffer after which tokens are generated

i	n	t		a	,	b	;	c	;

4096.

c	=	a	+	b	;				

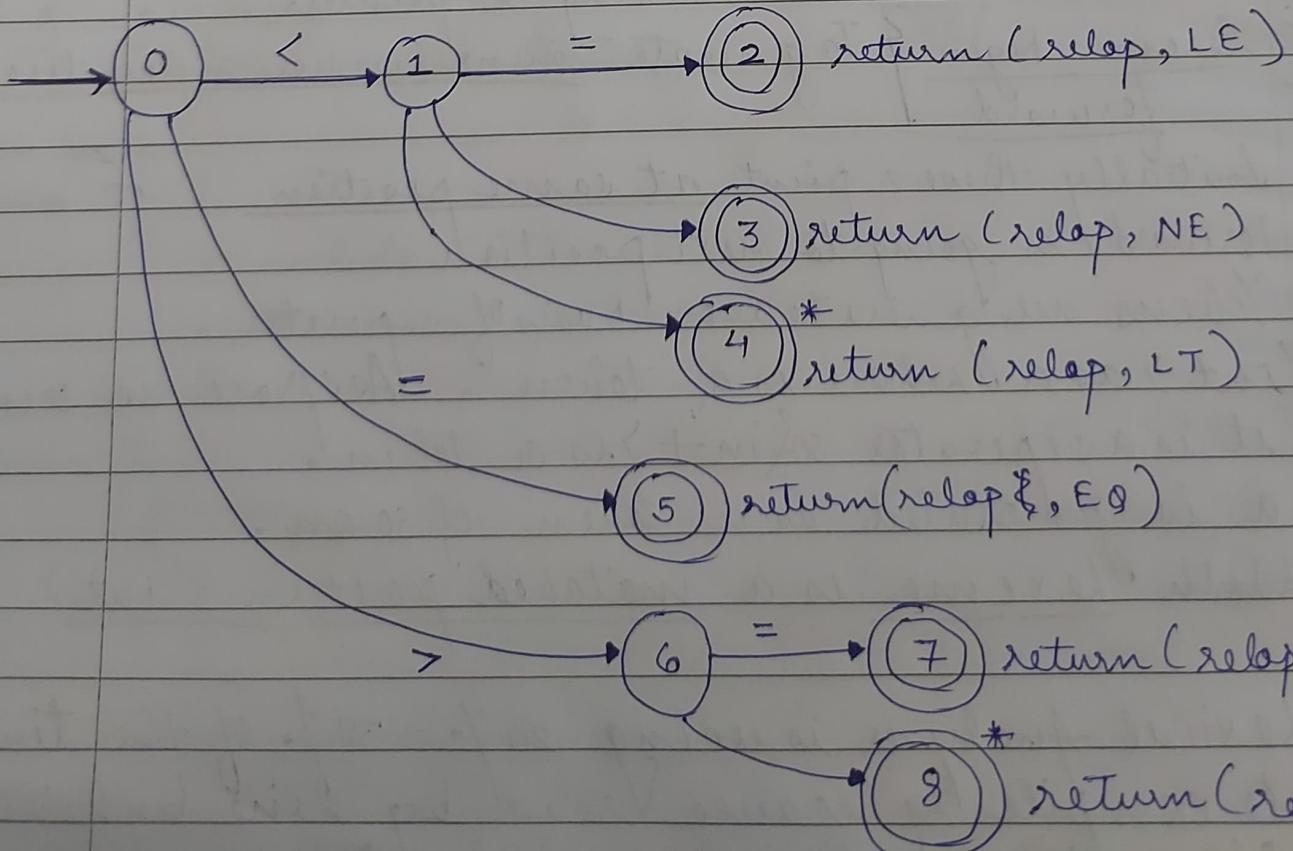
2-Buffer is used to speed the compile time.
We have 2 buffers, each of size 4096 bytes.

Working
We have data in 2 buffers. When int a+b,c; is converted to tokens & the pointer is set to second buffer (c=a+b), when first buffer is free, it will read next line.

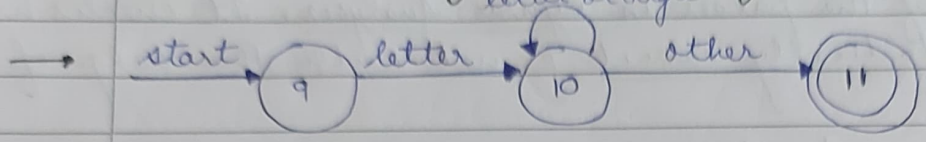
Q) Develop the transition diagram for the following topics (DFA).

i) Relational Operators
ii) To recognize words and identifiers
iii) For unsigned numbers.
iv) For whitespace

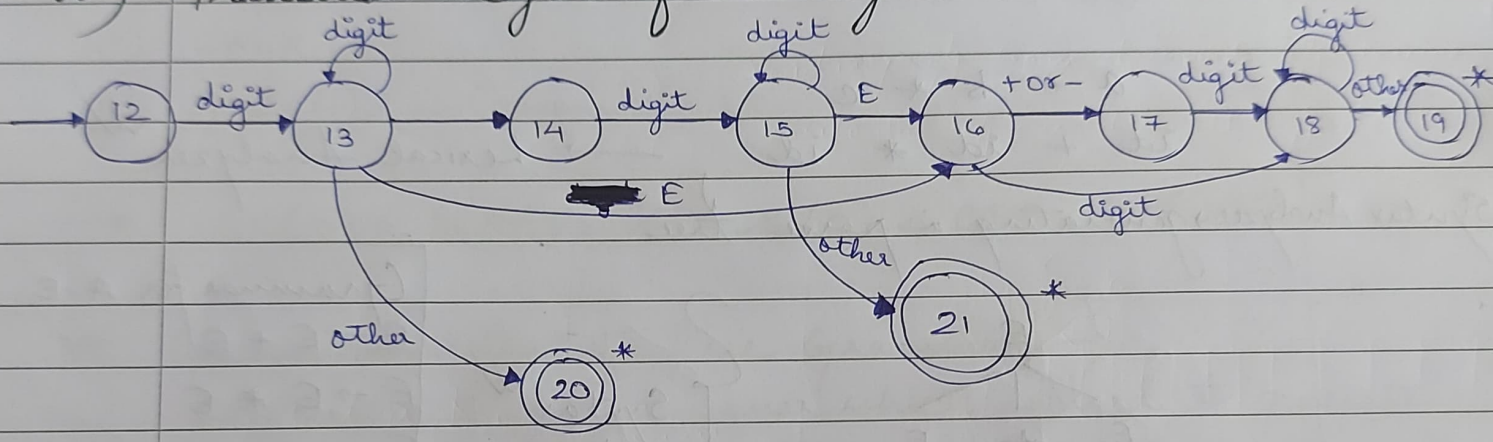
i) Transition Diagram for Relational Operators →



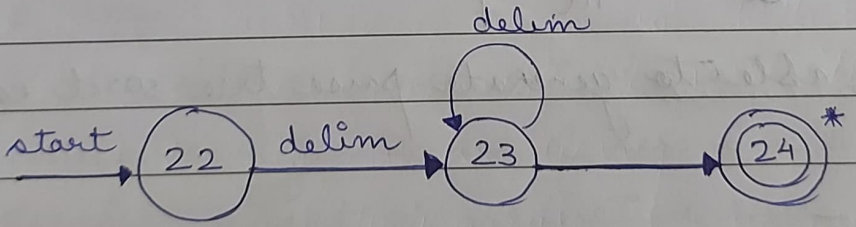
ii) Transition Diagram to recognise words of identifiers



iii) Transition Diagram for Unsigned Numbers



iv) Transition Diagram for whitespace



__/__/____

UNIT-2

PARSER

→ The name of the parser is Syntax Analyser. It checks whether prog. is syntatically correct or not with the help of grammar

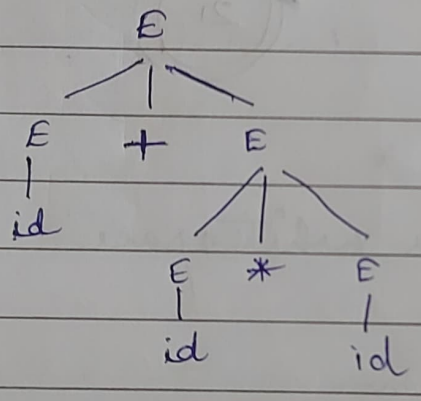
→ Grammar are of 2 types

- Ambiguous Grammar → If we have more than 1 parse tree
- UnAmbiguous Grammar

a + b * c

id + id * id ⟶ Lexical Analyzer ✓✓

Syntax Analyzer → generating a parse tree



Syntax Analyzer ✓✓

Parse Tree

Grammar for A.E

E → E + E

E → E * E

E → E - E

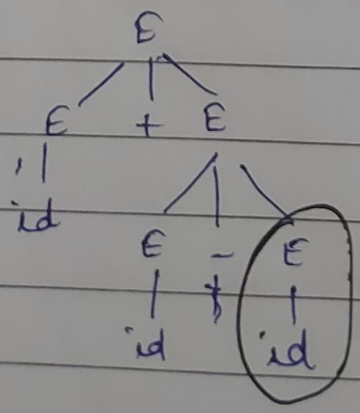
E → E / E

E → id

∴ we are able to generate parse tree so it is syntactically correct.

b) X + Y −

id + id −



missing, so error / parser is not able to generate a parse tree.

__/__/____

2 approaches to built syntax Analyzer. // (LL(1))

- TOP DOWN → Recursive Desent Parser; Non Recursive Desent
- BOTTOM UP. → LR Parsing / Operator Precedence

→ In LR parser there are 4 algorithm

a) LR (0) → weak
b) SLR (1) Simple LR
c) LALR (0)
d) CLR (1) Canonical LR

→ LR(0) < SLR(1) < LALR (0) < CLR(1) (strongest)

~~Grammar~~ → Context Free Grammar →

CFG is a formal grammar which is used to generate all possible strings in a given formal language.
CFG can be defined by $G = \{V, T, P, S\}$ four tuples

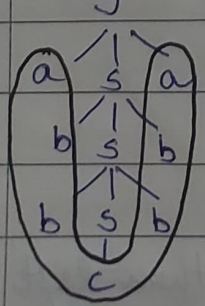
E → E + E
E → id

a) V describe a finite set of non terminal symbols (E)
b) T indicates a finite set of terminals symbols (id)
c) P describe a set of production rules
d) S describe start symbol

eg:- S → aSa
S → bSb
S → c

Check that abbcbba string can be derived from the given CFG

Ans



TYPES OF GRAMMAR

i) ambigitaeous
ii) unambiguous

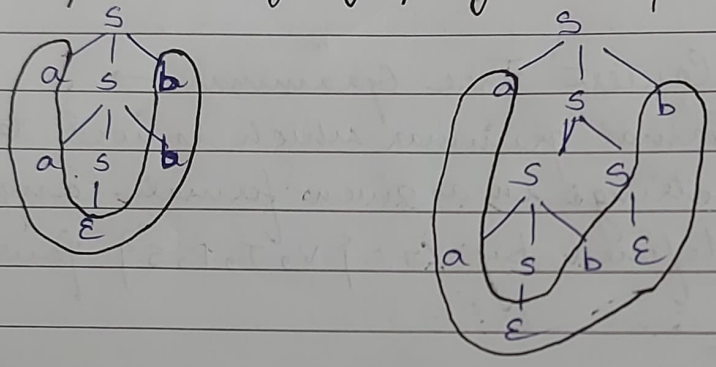
i) AMBIGUOUS → A grammar is said to be ambigious, if there exist more than one parse tree or more than one left most derivation or more than one RMD for given input string.

eg:- $S \rightarrow aSb / SS$

$S \rightarrow \varepsilon$

Check for ambiguity for given input string aabb

→



It is ambiguous as there are 2 parse trees

⇒ ∴

ELIMINATE LEFT RECURSION AND LEFT FACTORING

→ Top Down approach can't work on ambiguous grammar by eliminating left recursion & left factoring

1) Elimination of left Recursion → Top Down parsing method cannot handle left Recursive grammars, so a transformation is needed to eliminate left recursion

eg:- $E \rightarrow E+E$

$E \rightarrow E-E$ Check for Left Recursion

$E \rightarrow id$

→ If grammar is in form of $\underset{\text{LHS}}{A} \rightarrow \underset{\text{LHS}}{A}\alpha \mid \beta$ then left recursion

1) $E \to E+E$ → left recursion as LHS = RHS
2) $E \to E-E$ → "
3) $E \to id$ No left recursion

So rewrite the grammar by left recursion

$A \to A\alpha \mid \beta$
$E \to E+T \mid id$ ⇒ $A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

The left recursion pair of production $A \to A\alpha \mid \beta$ could be replaced by non left recursive production $A \to \beta A'$ & $A' \to \alpha A' \mid \varepsilon$

Q) Eliminate Left Recursion from the following grammar:

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

→ To eliminate left Recursion we have a rule such as

$A \to A\alpha \mid \beta$
$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$

→ $E \to E+T \mid T$ (A, A, α, β)
$A \to A\alpha \mid \beta$

→ $A \to \beta A'$
$E \to TE'$

→ $A' \to \alpha A' \mid \varepsilon$
$E' \to +TE' \mid \varepsilon$

$E \to A$
$\alpha \to +T$
$T \to \beta$

$T \to T*F \mid F$ (A, A, α, β)
$A \to A\alpha \mid \beta$

→ $A \to \beta A'$
$T \to FT'$

→ $A' \to \alpha A' \mid \varepsilon$
$T' \to *FT' \mid \varepsilon$

__/__/____

The grammar without left recursion is as follows:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

This grammar will not generate more than 1 parse tree.

Q2) Consider the following grammar & remove left recursion.

$A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

Soln To eliminate Left Recursion we have such rule as

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

$A \rightarrow \beta A'$

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

Rewrite the following grammar as

$A \rightarrow ABd \mid a$

$A \rightarrow Aa \mid a$

$B \rightarrow Be \mid b$

→ To eliminate Left Recursion we have rule such as

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

$A \rightarrow \beta A'$

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

$A \rightarrow ABd \mid a$

→ $A \rightarrow \beta A'$

$A \rightarrow \alpha A'$

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

$A' \rightarrow Bd A' \mid \varepsilon$

$A \rightarrow Aa \mid a$

→ $A \rightarrow \beta A'$

$A \rightarrow \alpha A'$

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

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

$B \rightarrow Be \mid b$

→ $A \rightarrow \beta A'$

$B \rightarrow b B'$

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

$A' \rightarrow$

The grammar without left recursion

$A \rightarrow aA'$

$A' \rightarrow BdA'/\epsilon$

$A \rightarrow aA'$

$A' \rightarrow aA'/\epsilon$

$B \rightarrow bB'$

$B' \rightarrow eB'/\epsilon$

Eliminate Left Recursion from the following grammar.

$S \rightarrow aB \mid aC \mid Sd \mid Se$

$B \rightarrow bBc \mid f$

$C \rightarrow g$

Rewrite the following grammar as

~~$S \rightarrow aB$~~ $S \rightarrow Sd/aB$

$S \rightarrow Se/aC$

$B \rightarrow bBc/f$

$C \rightarrow g$

→ To eliminate left Recursion we have ~~such~~ rule such as

$A \rightarrow A\alpha/\beta$

$A \rightarrow \beta A'$

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

$S \rightarrow Sd/aB$

→ $A \rightarrow \beta A'$

$S \rightarrow aBS'$

→ $A' \rightarrow \alpha A'/\epsilon$

$S' \rightarrow dS'/\epsilon$

$S \rightarrow Se/aC$

→ $A \rightarrow \beta A'$

$S \rightarrow aCS'$

→ $A' \rightarrow \alpha A'/\epsilon$

$S' \rightarrow eS'/\epsilon$

__/__/____

The grammar without left recursion is

$S \rightarrow aBS'$

$S' \rightarrow dS' \mid \varepsilon$

$S \rightarrow aCS'$

$S' \rightarrow eS' \mid \varepsilon$

$B \rightarrow bBC \mid f$

$C \rightarrow g.$

9) Eliminate the left recursion from the following grammar.

→ $A \rightarrow Ac \mid Aad \mid bd \mid c$

→ $L \rightarrow L, S \mid S$

$S \rightarrow a \mid (L)$

__/__/____

1) Eliminate Left Recursion from the following grammar:

$$A \to Ac \mid Aad \mid bd \mid c$$

Sol^n Rewrite the following grammar as

$$A \to Ac \mid bdc$$
$$A \to Aad \mid bd$$

→ To eliminate left Recursion we have rule such as

$$A \to A\alpha \mid \beta$$
$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

A Aα β A → Ac | c	A Aα β A → Aad | bd
→ A → βA'	→ A → βA'
A → cA'	A → bdA'
→ A' → αA' | ε	→ A' → αA' | ε
A' → cA' | ε	A' → adA' | ε.

The grammar without left recursion is

$$A \to cA' \mid bdA'$$
$$A' \to cA' \mid adA' \mid e$$

2) Eliminate the left recursion from the following grammar.

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

Sol^n Rewrite the following grammar as

$$L \to L, S \mid S$$
$$S \to a \mid (L).$$

To eliminate left Recursion we have rule such as.

$$A \to A\alpha \mid \beta$$
$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

__/__/____

$A \quad A \quad \alpha \quad \beta$

$\rightarrow \quad L \rightarrow L, S \mid S$

$\rightarrow \quad A \rightarrow \beta A'$

$L \rightarrow SL'$

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

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

The grammar without Left Recursion is

$L \rightarrow SL'$

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

$S \rightarrow a \mid (L)$

INDIRECT RECURSION ⟶

Q1) Eliminate Left Recursion from the following grammar

$A \rightarrow Ba \mid Aa \mid c$

$B \rightarrow Bb \mid Ab \mid d$

⟶ Rewrite the grammar as.

$A \rightarrow Aba \mid Aa \mid c$

$B \rightarrow Bb \mid Bab \mid d$

To eliminate left Recursion we have rule as

$A \rightarrow Aba \mid c$	$A \rightarrow Aa \mid c$	$B \rightarrow Bb \mid d$	$B \rightarrow Bab \mid d$
$\rightarrow A \rightarrow \beta A'$	$\rightarrow A \rightarrow \beta A'$	$\rightarrow A \rightarrow \beta A'$	$\rightarrow A \rightarrow \beta A'$
$A \rightarrow cA'$	$A \rightarrow cA'$	$B \rightarrow dA'$	$B \rightarrow dB'$
$\rightarrow A' \rightarrow \alpha A' \mid \varepsilon$	$\rightarrow A' \rightarrow \alpha A' \mid \varepsilon$	$\rightarrow A' \rightarrow \alpha A' \mid \varepsilon$	$\rightarrow A' \rightarrow \alpha A' \mid \varepsilon$
$A' \rightarrow baA' \mid \varepsilon$	$A' \rightarrow aA' \mid \varepsilon$	$B' \rightarrow bB' \mid \varepsilon$	$B' \rightarrow abB' \mid \varepsilon$

The following grammar now can be written as.

$A \rightarrow cA'$

$A' \rightarrow baA' \mid aA' \mid \varepsilon$

$B \rightarrow dB'$

$B' \rightarrow bB' \mid abB' \mid \varepsilon$.

__/__/____

g2) $S \to Aa \mid b$
$A \to Ac \mid Sd \mid \varepsilon$

Soln →
$S \to Sda \mid b$
$A \to Ac \mid Aad \mid \varepsilon$

For removing left recursion we have rule such as:

$A \to A\alpha \mid B$

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

$S \to Sda \mid b$	$A \to Ac \mid \varepsilon$	$A \to Aad \mid \varepsilon$
→ $A \to \beta A'$	→ $A \to \beta A'$	→ $A \to \beta A'$
$S \to bS'$	$A \to \varepsilon A'$	$A \to \varepsilon A'$
→ $A' \to \alpha A' \mid \varepsilon$	→ $A' \to \alpha A' \mid \varepsilon$	→ $A' \to \alpha A' \mid \varepsilon$
$S' \to daS' \mid \varepsilon$	$A' \to cA' \mid \varepsilon$	$A' \to adA' \mid \varepsilon$

The following can be written as.

$S \to bS'$

$S' \to daS' \mid \varepsilon$

$A \to \varepsilon A'$

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

LEFT FACTORING → (first few terms of RHS is common)

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \cdots \mid \alpha\beta_n$.

for eg:- $A \to aaB \mid aaC \mid aaD \mid aaE$

To eliminate left factoring the rule is:

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

Q. Do the left factoring for the following grammar:

$S \to iETS \mid iETSeS \mid a$

$E \to b$

Sol → Rewrite the grammar as follows:

$S \to iETS \mid iETSeS$ → i)

$S \to a$ → ii)

$E \to b$ → iii)

→ Removing left Factoring we have rule such as.

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \alpha\beta_n$

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

→ $S \to \underbrace{iETS}_{\alpha} \mid \underbrace{iETS}_{\alpha}\underbrace{eS}_{\beta}$

$A \to \alpha A'$

$S \to iETSS'$

$S' \to eS \mid \varepsilon$

$S \to a$

$E \to b$

Answer

Q2) $A \to aAB \mid aA \mid a$

$B \to bB \mid b$

→ Rewrite the grammar as

$A \to aAB \mid aA$

$A \to a$

$B \to bB \mid b$

→ Removing left Factoring we have rule as.

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n$

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \ldots \beta_n$

→ $A \to \underbrace{a}_{\alpha}\underbrace{AB}_{\beta} \mid \underbrace{aA}_{\alpha}\underbrace{\ }_{\beta} \mid \underbrace{a}_{\alpha}$ $A \to \underbrace{aAB}_{\alpha\beta_1} \mid \underbrace{aA}_{\alpha\beta_2} \mid \underbrace{a}_{\alpha}$

$A \to \alpha A'$ $A \to aA'$

$A \to aA'$ $A' \to AB \mid A \mid \varepsilon$

$A' \to$

$B \to \underbrace{bB}_{\alpha\beta_1} \mid \underbrace{b}_{\alpha}$

$B \to bB'$

$B' \to B \mid \varepsilon$

Q1. $S \rightarrow assbs / asasb / abb / a$

Q2. $S \rightarrow a / ab / abc / abcd$

1) $S \rightarrow assbs / asasb / abb / a$ → i)

Sol^n. To remove left factoring we have rule such as:

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \cdots / \alpha\beta_n$

$A \rightarrow \alpha A'$

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

→ $S \rightarrow a\,ssbs / a\,sasb / abb / a$

$S \rightarrow as'$

$S \rightarrow ssbs / sasb / bb / \varepsilon$ } Ans.

2) $S \rightarrow a / ab / abc / abcd$

→ Rewrite the grammar as follows:

$S \rightarrow ab / abc / abcd$ → i)

$S \rightarrow a$ → ii)

→ Removing left factoring we have rule such as

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \alpha\beta_4 / \cdots / \alpha\beta_n$

$A \rightarrow \alpha A'$

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

→ $S \rightarrow ab / abc / abcd$

$S \rightarrow abs'$

$s' \rightarrow c / cd / \varepsilon$

$S \rightarrow a.$ } Ans.

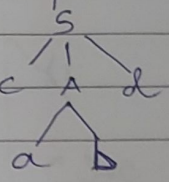
__/__/____

Recursive Descent Parsing →

$S \to cAd$

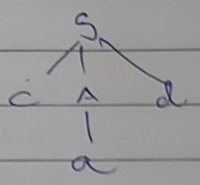
$A \to ab/a$

$w = cad$ (Input)



cabd ✗

PROBLEM OF RECURSIVE DESCENT PARSING { When we have 'n' production, it will check for each production, so recursive descent will take a lot of time.

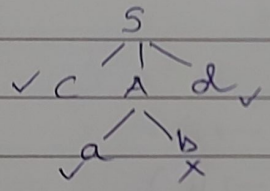


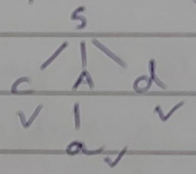
cad ✓

eg:- Consider the grammar $S \to cAd$

$A \to ab/a$

To construct a parse tree (TOP-DOWN) for the input string $w = cad$, begins with a tree consisting of a single node labelled 'S' and the input pointer pointing to 'c'.





$cad \neq cabd$ $cad = cad$.

Non Recursive Descent Parsing →

FIRST AND FOLLOW FUNCTION →

★ FIRST FUNCTION → Following are the rules used to compute first function.

1) If the terminal symbol 'a' then FIRST{a} = {a}

eg:- $X \to abc$

FIRST OF (X) = First (a) = {a}

2) If there is a rule $X \to \varepsilon$ then First of X = First of $\{\varepsilon\} = \{\varepsilon\}$

3) For the rule $X \to abc$ & $A \to mnq$, First of (X) = First (A) = First (m) = {m}

4) For the rule $A \to X_1 | X_2 | X_3 | \dots | X_n$ First (A) = First (X_1) ∪ First (X_2) ∪ .. First (X_n)

__/__/____

Q. Construct the first sets for the given grammar:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

→ After removing left recursion grammar is as follows:

$E \to TE'$

$E' \to +TE' \mid \varepsilon$

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

Compute First(E) = First of (T) $= \{(, id\}$

$T \to FT'$

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

$F \to (E) \mid id$

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

First(E') = First(+) ∪ First(ε)

$= \{+, \varepsilon\}$

First(T') = First(*) ∪ First(ε)

$= \{*, \varepsilon\}$

__/__/____

Q2. Consider the grammar

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

→ After removing left recursion

$L \rightarrow SL'$

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

$S \rightarrow a \mid (L)$

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

$First(S) = First\{a\} \cup First\{(\}$

$= \{a, (\}$

$First(L') = First\{,\} \cup First(\varepsilon)$

$= \{, , \varepsilon\}$

Q3. Construct first sets for the given grammar

$S \rightarrow aABb$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow d \mid \varepsilon$

→ No left recursion

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

$First(A) = First(c) \cup First(\varepsilon) = \{c, \varepsilon\}$

$First(B) = First(d) \cup First(\varepsilon) = \{d, \varepsilon\}$

$M \to aDE \quad E \to a/b$

Follow of D = First(E)

First(E) = {a, b}

$M \to aDe$

Follow(D) = First(e)

= {e}

__/__/____

FOLLOW FUNCTION →

The rules for computing 'follow function' are as follows:-

1. For the start symbol 'S' place $ in Follow(S)
2. If there is a grammar $X \to \overset{\alpha}{a}\overset{B}{B}\overset{\beta}{c}$ then Follow(B) = First(β)

   Follow(B) = First(c) = {c}
3. If the grammar is in the form of

   $A \to \alpha B \beta$

   $\beta \to a/\varepsilon$

   then we apply Follow(B) = First(β)

   = {a, ε}

   Then Eliminate ε

   = **First(β) − ε ∪ Follow of (B)**

eg:-

$M \to aBE$

$E \to b/\varepsilon$

Follow(B) = First(E)

= {b, ε}

Eliminate ε

First(β) − ε ∪ Follow(β)

4. If the grammar is in form

   $A \to \alpha B$ then

   Follow(B) = Follow(A)

Q1. Consider the grammar $E \to E+T$

$T \to T*F/F$

$F \to (E)/id$

- Make the grammar suitable for Top-Down Parsing.
- Construct First & Follow sets.

Soln →

The grammar after removing left recursion is

$E \to TE'$

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

$T \to FT'$

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

$F \to (E)/id.$

__/__/____

	First set of	Follow
E →	{ (, id }	{), $ }
E' →	{ +, ε }	{), $ }
T →	{ (, id }	{ +,), $ }
T' →	{ *, ε }	{ +,), $ }
F →	{ (, id }	{ *, +,), $ }

Follow sets of → Check grammar E is placed where .....

Consider F → (E)

A → αBβ

Follow (E) = First ( ) = {) }

Since E is start symbol hence place $ in Follow(E)

Follow (E) = {), $ } ⟶ i)

Calculate Follow (E')

Check grammar where is E' in α

E → TE'

E' → + T E'

This is in form of A → αβ

First grammar is appropriate so

E → T E'

A → α B so by Rule 4

Follow (E') = Follow (E)

= {), $ }

Follow (T)

E → T E' ⟶ i)

E' → + T E'

Considering i)

E → T E'

A → α B β

Follow (T) = First (E')

= {), $ } { +, ε }

Eliminate ε

Follow(E') - ε ∪ Follow(E')

{+, ε} - ε ∪ {), $}

= {+,), $}

Follow of T'

T → FT' → i)

T' → * FT' | ε → ii)

Considering i)

T → F T'

A → α B

Follow of T' = Follow of T

= {+,), $}

Follow(F)

Checking grammar

T → F T' → i)

T' → * FT' | ε

By 1st grammar

A → Bβ

Follow of F = First(T')

(First(T') - ε) ∪ Follow(T')

({* ε} - ε) ∪ {+,), $}

{*, +,), $}

__/__/____

Q) Consider grammar

$S \rightarrow aABb$

$A \rightarrow c / \varepsilon$

$B \rightarrow d / \varepsilon$

→ No Left Recursion

→

	First	Follow
S	{a}	{$, b}
A	{c, ε}	{d, b}
B	{d, ε}	{b}

→ Follow of S

As S is in start symbol so place $

→ Follow of A

$S \rightarrow \underset{\alpha}{a} \underset{B}{A} \underset{\beta}{B} b$

Follow (A) = First (B)

= {d, ε}

⇒ Eliminate ε

First (B) − ε ∪ Follow (B)

{d, ε} − ε ∪ {b}

= {d, b}

→ Follow of B

$S \rightarrow aA\underset{B}{B}\underset{\beta}{b}$

Follow of B = First (b)

= {b}

23/2/23

The work of Syntax Analyzer is to check whether prog is syntactically correct or not.

Q. Construct predictive parsing table for the following grammar: & show the moves made by the predictive parser on i/p id + id * id.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

↳ Construction of predictive parsing table / LL(1) parsing table

For each production $A \rightarrow \alpha$

1. Find First(α) & for each terminal in First(α), Make entry $A \rightarrow \alpha$ in the table
2. If First(α) contains ε as terminal, then find Follow(A) & for each terminal in Follow(A) make entry $A \rightarrow \varepsilon$ in the table....
3. If First(α) contains 'ε' and Follow(A) contain '$' as terminal then make entry $A \rightarrow \varepsilon$ in the table for the '$'

→ Eliminating the left recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

→ Computing First & Follow Set

1

__/__/____

	First	Follow
E	{ (, id }	{), $ }
E'	{ +, ε }	{), $ }
T	{ (, id }	{ +,), $ }
T'	{ *, ε }	{ +,), $ }
F	{ (, id	{ +, *,), $ }

Row → Non Terminal
Column → Terminals

:) Check for E, we have (, id in First so write it in table
:) If ε then go to Follow

→ Generating a predictive parsing table

	+	*	(	)	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	

Stack & Input must be empty for right ans.

→ Checking for id + id

→

MATCHED	STACK	INPUT	ACTION
	E$	id + id $	E → TE'
	TE'$	id + id $	T → FT'
	FT'E'$	id + id $	F → id
id	idT'E'$	id + id $	Match id
	T'E'$	+ id $	T' → ε
	E'$	+ id $	E' → +TE'
id +	+TE'$	+ id $	Match +
	TE'$	id $	T → FT'
	FT'E'$	id $	F → id
id + id	idT'E'$	id $	Match id
	T'E'$	$	T' → ε
	E'$	$	E' → ε
	$	$	

∴ As Stack & input is empty this means the grammar is syntactically correct.

__/__/____

→ Checking for id + + id

MATCHED	STACK	INPUT	ACTION
	E$	id + id $	E → TE'
	TE'$	id + id $	T → FT'
	FT'E'$	id + id $	F → id
id	id T'E'$	id + id $	Match id
	T'E'$	+ id $	

__/__/____

Q) Given the grammar

$S \to (L) \mid a$

$L \to L, S \mid S$

- Do the necessary change to make it suitable for LL(1) Parser
- Show the move made by predictive parser on i/p (a, a)

→ After removing left recursion

$L \to SL'$

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

$S \to a \mid (L)$

$S \to (L)$

	First	Follow
$S \to (L) \mid a$	{a, (}	{$}
$L \to SL'$	{a, (}	{)}
$L' \to , SL' \mid \varepsilon$	{, , ε}	{)}

Follow(L) = First ())

{)}

Follow(L') = $L \to SL'$

= αB

Follow(L') = Follow(L)

= {)}

→ Generating a predictive parsing table

	,	(	)	$	a
S		$S \to (L)$			$S \to a$
L		$L \to SL'$			$L \to SL'$
L'	$L' \to , SL'$		$L' \to \varepsilon$		

⇒ (a,a)

Match	Stack	Input	Action
	S $	(a,a) $	S → (L)
(	(L) $	(a,a) $	Matched (
	L) $	a,a) $	L → SL'
	SL') $	a,a) $	S → a
(a	aL') $	a,a) $	Matched a
	L') $	a,a) $	L' → ,SL'
(a,	,SL') $	,a) $	Matched ,
	SL') $	a) $	S → a
a	aL') $	a) $	Matched a
	L') $	) $	L' → ε
)	) $	) $	Matched)
	$	$	

∴ As input & stack is empty so grammar is syntactically correct.

__/__/____

Q) Given the grammar

$S \rightarrow iETs / iE\,TS\,eS / a$

$E \rightarrow b$

a) Make the grammar suitable for LL(1) Parsing.

b) Construct parsing table for given grammar ...

Soln Rewrite the grammar as.

$S \rightarrow iETS / iE\,TSeS$

$S \rightarrow a$

$E \rightarrow b$

→ Removing left Factoring we have rule such as ....

$A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 / \alpha\beta_4 / \ldots\ldots / \alpha\beta_n$

$A \rightarrow \alpha A'$

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

→ $S \rightarrow \overbrace{iETS}^{\alpha} / \overbrace{iETS}^{\alpha}\overbrace{eS}^{\beta}$

$S \rightarrow iETSS'$

$S' \rightarrow eS / \varepsilon$

$S \rightarrow a$

$E \rightarrow b.$

→ Generating First & Follow Sets

	FIRST	FOLLOW
S	{i, a}	
S'	{e$, ε}	
E	{b}	

Follow(S) = $S' \rightarrow \cancel{es/\varepsilon}\ \overbrace{ie}^{\alpha}\,T\,\overbrace{S}^{\beta}\,\overbrace{S'}^{\beta}$

= First(S')

= {e, ε}

Removing ε

First(S') − ε ∪ Follow(S')

= {e, ε) − ε ∪

Follow(S') = $\overbrace{ie\,T\$}^{\alpha}\,\overbrace{S'}^{\beta}$

= Follow(S') = Follow(S)

=

g)

$S \to Aa \mid B$

$A \to Bb \mid Sc \mid \varepsilon$

$B \to d$

Rewrite grammar as

→

$S \to Sca \mid B$

$A \to Bb \mid Aac \mid \varepsilon$

$B \to d$

→ To eliminate Left Recursion we have rule as.

$S \to Sca \mid B$ (A, A α, B)

→ $A \to \beta A'$

$S \to BS'$

→ $A' \to \alpha A' \mid \varepsilon$

$S' \to caS' \mid \varepsilon$

$A \to Aac \mid Bb$ (A, A α, B)

→ $A \to \beta A'$

$A \to Bb\, A'$

→ $A' \to \alpha A' \mid \varepsilon$

$A' \to acA' \mid \varepsilon$

$S \to BS'$

$S' \to caS' \mid \varepsilon$

$A \to BbA' \mid \varepsilon$

$A' \to aCA' \mid \varepsilon$

$B \to d$

→ Generating First & Follow Sets

	First	Follow
S	{d}	{$}
S'	{c, ε}	{$}
A	{d, ε}	{$}
A'	{a, ε}	{$}
B	{d}	{b, $}

27/02/23

BOTTOM UP PARSING →

LR(0)

Q. Construct LR parsing table for the given grammar:

$S \rightarrow AA$

$A \rightarrow aA \mid b$

→ 1. $S \rightarrow AA$

2. $A \rightarrow aA$

3. $A \rightarrow b$

STEP 1. Add augmented grammar

$S' \rightarrow .S$

$S \rightarrow .AA$

$A \rightarrow .aA$

$A \rightarrow .b$

} State I_0

STEP 2 Apply GOTO Function (Shift "." DOT to end of rule)

→ GOTO (I_0, S)

$S' \rightarrow S.$ } I_1

→ GOTO (I_0, A)

$S \rightarrow A.A$

$A \rightarrow .aA$

$A \rightarrow .b$

} I_2

→ GOTO (I_0, a)

$A \rightarrow a.A$

$A \rightarrow .aA$

$A \rightarrow .b$

} I_3

→ GOTO (I_0, b)

$A \rightarrow b.$ } I_4

→ Apply GOTO with I_2 as in I_1 we have "." in end.

GOTO (I_2, A)

$S \to AA.$ } I_5

GOTO (I_2, a)

$A \to a.A$
$A \to .aA$
$A \to .b$ } I_3

GOTO (I_2, b)

$A \to b.$ } I_4

→ Apply GOTO with I_3

GOTO (I_3, A)

$A \to aA.$ } I_6

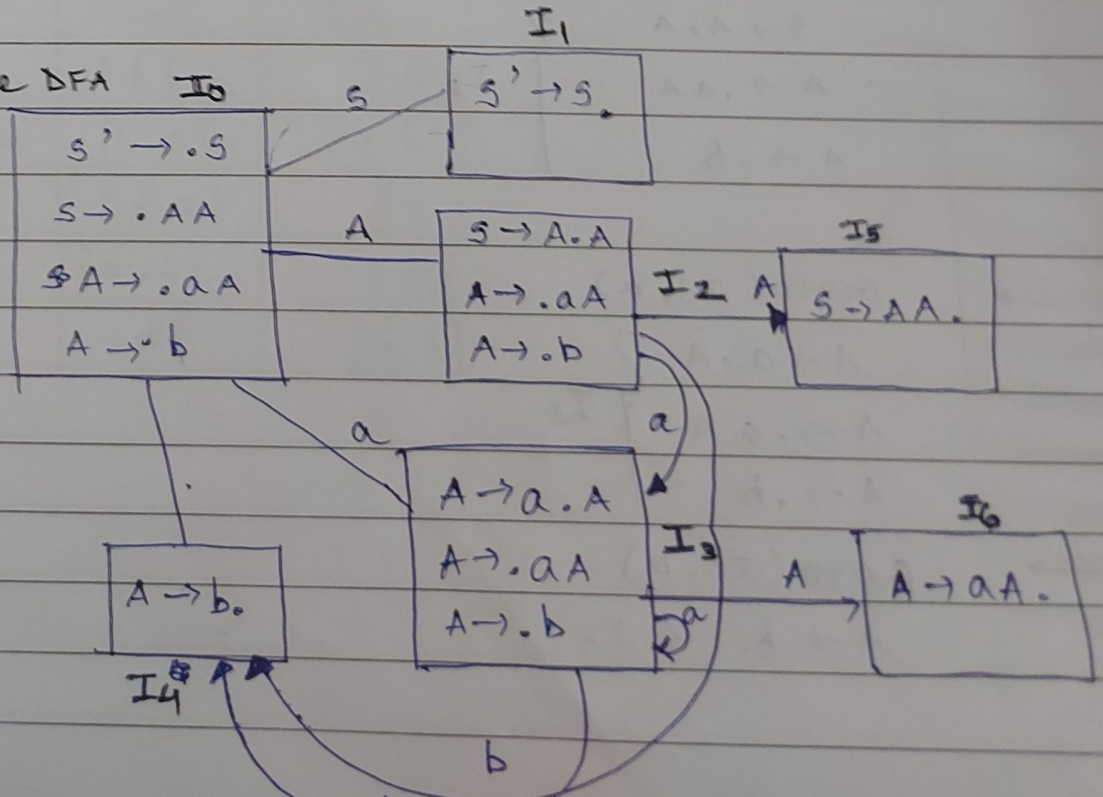
GOTO (I_3, a)

$A \to a.A$
$A \to .aA$
$A \to .b$ } I_3

GOTO (I_3, b)

$A \to b.$ } I_4

STEP 3:- Note the DFA



__/__/____

Parsing Table → Rows → States

	ACTION			GOTO	
	a	b	$	S	A
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	Reduce 3	Reduce 3	Reduce 3		
I_5	Reduce 1	Reduce 1	Reduce 1 ACCEPT		
I_6	Reduce 2	Reduce 2	Reduce 2 ACCEPT		

$I_0 \xrightarrow{S} I_1$ so write 1 in GOTO

Shift of Reduce in Action (terminals)

Write Reduce where . reaches END

Reduce 3 as we have written 3rd grammar

→

Stack	Input	Action
$0	a a b b $	Shift a and goto 3
$ 0 a 3	a b b $	Shift a and goto 3
$ 0 a 3 a 3	b b $	Shift b and goto 4
$ 0 a 3 a 3 b 4	b $	Reduce A → b
$ 0 a 3 a 3 A	b $	
$ 0 a 3 a 3 A 6	b $	Reduce A → aaA
$ 0 a 3 A 6	b $	Reduce A → aA
$ 0 A 2	b $	Shift B of goto 4
$ 0 A 2 b 4	$	Reduce A → b
$ 0 A 2 A 5	$	Reduce S → AA
$ 0 S 1	$	Accept

In the table see 'A' in I_3 its '6'

State of $ = S

Q2. Construct LR(0) parsing table for the given grammar

$S \rightarrow (S)$

$S \rightarrow id$

Parse the input (id)

→ Adding augmented grammar

1. $S \rightarrow (S)$
2. $S \rightarrow id$

STEP 1 Adding augmented grammar

$S' \rightarrow .S$
$S \rightarrow .(S)$ } I_0
$S \rightarrow .id$

STEP 2 Applying GOTO

* GOTO (I_0, S)

$S' \rightarrow S.$ ⟶ I_1

* GOTO $(I_0, ()$

$S \rightarrow (.S)$ $S \rightarrow .(S)$ $S \rightarrow .id$ → I_2

* GOTO (I_0, id)

$S \rightarrow id.$ → I_3

* GOTO (I_2, S)

$S \rightarrow (S.)$ → I_4

*

*

* GOTO $(I_2, ()$

$S \rightarrow (.S)$
$S \rightarrow .(S)$ } → I_2
$S \rightarrow .id$

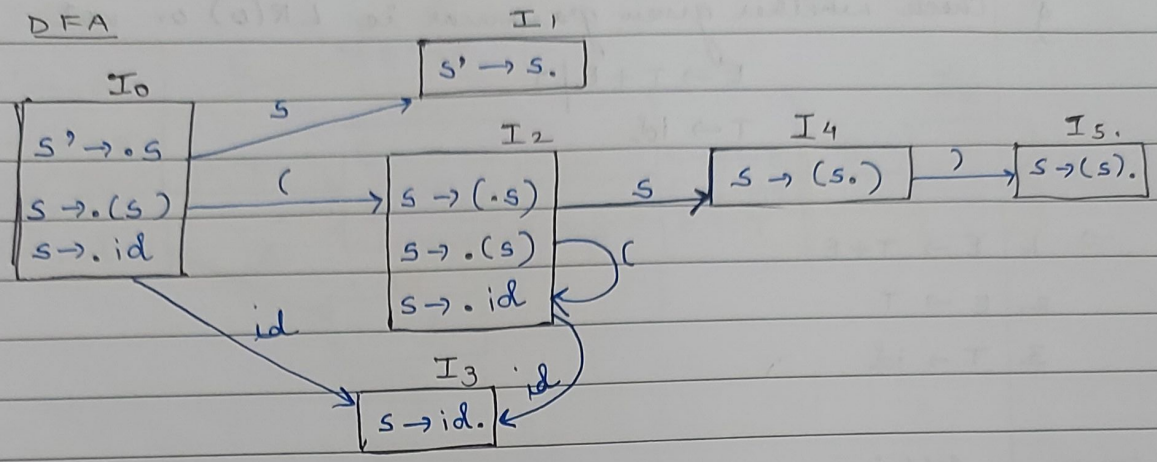
GOTO (I_2, id)

$S \rightarrow id$ → I_3

GOTO $(I_4,))$

$S \rightarrow (S).$ → I_5

STEP 3 DFA



Parsing Table

	ACTION				GO TO
	(	~~ACTION~~)	id	$	S
I_0	S2		S3		1
I_1				ACCEPT	
I_2	S4	~~S2~~	S3		4
I_3	r2	r2	r2	r2	
I_4		S5			
I_5	r1	r1	r1	r1	

#

Stack	Input	Action
$0	(id)$	Shift (and goto 2
$0(2	id)$	Shift id and goto 3
$0(2id3	)$	Reduce S → id
$0(2S4	)$	Shift) goto 5
$0(2S4)5	$	Reduce S → (S)
$0S1	$	Accept

Q Check whether given grammar is LR(0) or not

$E \rightarrow T + E \mid T$

$T \rightarrow id$

→ 1. $E \rightarrow T+E$

2. $E \rightarrow T$

3. $T \rightarrow id$

STEP 1. Adding augmented grammar.

$E' \rightarrow .E$

$E \rightarrow .T + E$

$E \rightarrow .T$

$T \rightarrow .id$

} I_0

STEP 2 Applying GOTO

0 GOTO (I_0, E)

$E' \rightarrow E.$ ⟶ I_1

0 GOTO (I_0, T)

$E \rightarrow T. + E$

$E \rightarrow T.$

} I_2

0 GOTO (I_0, id)

$T \rightarrow id.$ → I_3

0 GOTO $(I_2, +)$

$E \rightarrow T + .E$

$E \rightarrow .T + E$

$E \rightarrow .T$

} → I_4

0 GOTO (I_0, I_4, E)

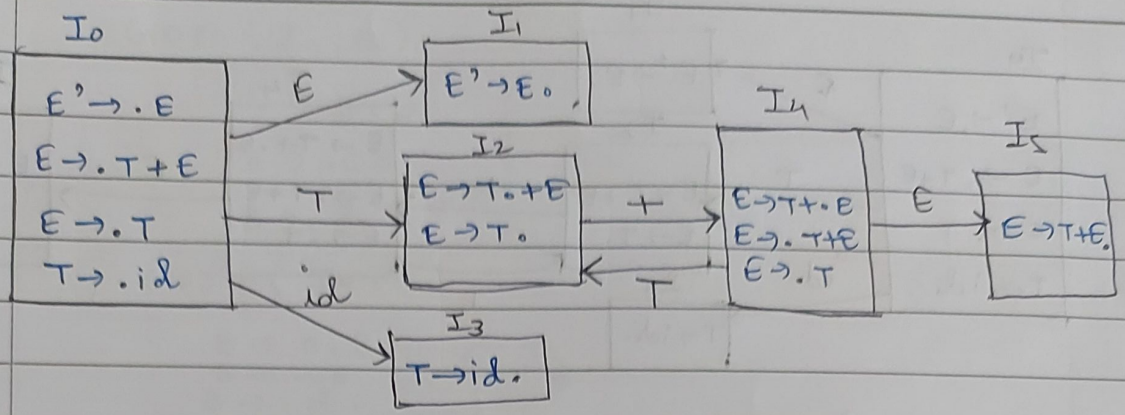
$E \rightarrow T + E.$ ⟶ I_5

0 GOTO (I_4, T)

$E \rightarrow T. + E$

$E \rightarrow T.$

} I_2



Parsing Table

	+	id	$	E	T
I_0		S_3		1	2
I_1			Accept		
I_2	S_4/R_2	R_2	R_2		
I_3	R_3	R_3	R_3		
I_4				5	2
I_5	R_1	R_1	R_1		

As in 2) '.' reaches end, so reduce

NOTE → For state '2' + symbol we have 2 operations 'shift' & 'reduce' so there is a conflict. The given grammar is not in LR(0).

SLR

Q Check whether given grammar is in SLR or not.

$E \rightarrow T+E$

$E \rightarrow T$

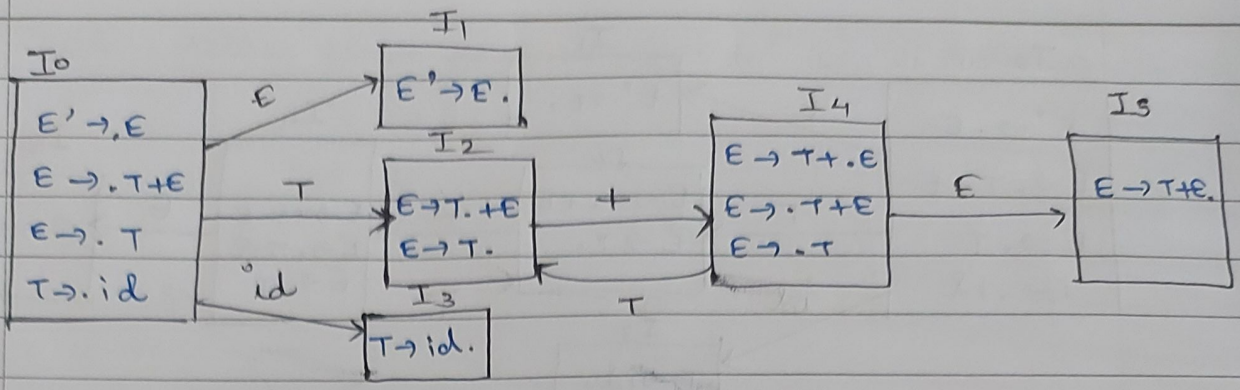
$T \rightarrow id$

i) Everything is same as LR(0). Calculate Follow(E)

Follow(E) = {$}

Follow(T) = {+, $}

$E \rightarrow T + E$
$A \rightarrow B\ \beta$
$E \rightarrow T$



	+	id	$	E	T
I_0		S_3		1	2
I_1			ACCEPT		
I_2	S_4		r_2		
I_3	r_3		r_3		
I_4				5	2
I_5			r_1		

Follow(T) = {+, $}

Check for E→T.
Follow T = Follow E
so in $ write r_2

g. Construct SLR parsing Table for given grammar

$E \to BB$

$B \to cB \mid d$.

→ 1. $E \to BB$

2. $B \to cB$

3. $B \to d$

Augmented Grammar.

$\hat{E} \to .E$

$E \to .BB$

$B \to .cB$

$B \to .d$

} I_0

0 GOTO(I_0, E)

$E' \to E.$ → I_1

__/__/____

- GOTO (I_0, B)

  $E \to B.B$
  $B \to .CB$
  $B \to .d$ } I_2

- GOTO (I_0, C)

  $B \to C.B$
  $B \to .CB$
  $B \to .d$ } I_3

- GOTO (I_0, d)

  $B \to d.$ $\longrightarrow I_4$

- GOTO (I_2, B)

  $E \to BB.$ $\to I_5$

- GOTO (I_2, C)

  $B \to C.B$
  $B \to .CB$
  $B \to d.$ } I_3

- GOTO (I_2, d)

  $B \to d.$ $\to I_4$

GOTO (I_3, B)

$B \to CB.$ $\longrightarrow I_6$

~~$B \to CB$~~

~~$B \to d$~~

GOTO (I_3, C)

$B \to C.B$
$B \to .CB$
$B \to .d$ } I_3

GOTO (I_3, d)

$B \to d.$ $\to I_4$

I_0

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

13/3/23

Q) Construct LR of SLR Parsing Table for the given grammar

$A \rightarrow Bb$

$B \rightarrow a$

→ 1. $A \rightarrow Bb$

2. $B \rightarrow a$

Adding Augmented Grammar

$A' \rightarrow .A$

$A \rightarrow .Bb$ } I_0

$B \rightarrow .a$

Applying GOTO

→ GOTO (I_0, A)

$A' \rightarrow A. \longrightarrow I_1$

→ ~~$A \rightarrow .Bb$~~ GOTO (I_0, B)

$A \rightarrow B.b \longrightarrow I_2$

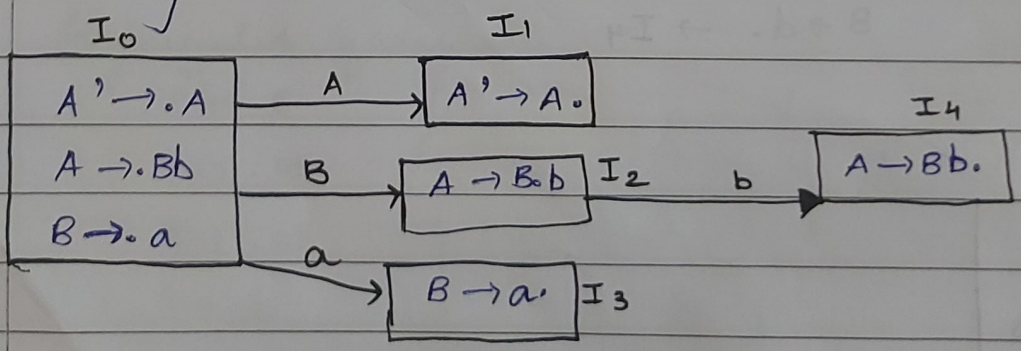
GOTO (I_0, a)

$B \rightarrow a. \longrightarrow I_3$

GOTO (I_2, b)

$A \rightarrow Bb. \longrightarrow I_4.$

Making DFA



Making LR parsing Table

	ACTION			GOTO	
	b	a	$	A	B
I_0		S_3		1	2
I_1			ACCEPT		
I_2	S_4				
I_3	r_2	r_2	r_2		
I_4	r_1	r_1	r_1		

After making LR parsing table, construct SLR parsing table.

Follow Function

Follow(A) = { $ }

Follow(B) = { b }

SLR Parsing Table →

	ACTION			GOTO	
	b	a	$	A	B
I_0		S_3		1	2
I_1			ACCEPT		
I_2	S_4				
I_3	r_2				
I_4			r_1		

__/__/____

Q) Construct LR & SLR Parsing Table for given grammar

$S \to AB$

$A \to a$

$B \to a$

Adding Augmented Grammar

$S' \to .S$

$S \to .AB$

$A \to .a$

$B \to .a$

} I_0

Applying GOTO

GOTO(I_0, S)

$S' \to S.$ $\to I_1$

GOTO(I_0, A)

$S \to A.B$

$B \to .a$

} $\to I_2$

~~GOTO(I_0, B)~~ GOTO(I_0, a)

$A \to a.$

$B \to a.$

} I_3

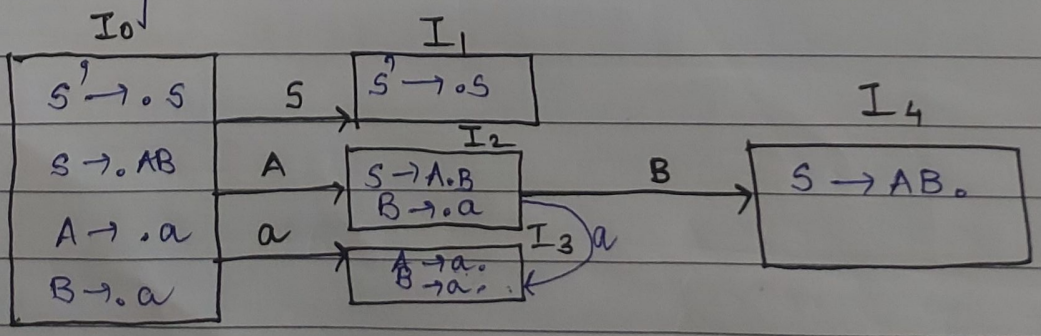
GOTO(I_2, B)

$S \to AB.$ $\to I_4$

GOTO(I_2, a)

$B \to a.$ $\to I_5$

Making DFA



__/__/____

LR Parsing Table

	ACTION		GOTO		
	a	$	S	A	B
I_0	S_3		1	2	
I_1		Accept			
I_2					4
I_3	r_2/r_3	r_3/r_2			
I_4	r_1	r_1			
I_5	r_3	r_3			

Now moving for SLR Parsing Table.

Follow (S) = { $ }

Follow (A) = { }

Follow (B) = { }

Lookahead पहले का देखते हैं।

16/3/23

In Bottom Up's parsing we have 2 conflicts
- Shift Reduce conflict
- Reduce Reduce conflict

CLR (Canonical LR parsing)

Q. Construct CLR parsing table for the following grammar.

$S \to CC$

$C \to cC \mid d$

Soln LR(1) = LR(0) + Look Ahead

Step 1 Add the augmented grammar

Given
1. $S \to CC$
2. $C \to cC$
3. $C \to d$

I_0
- $S' \to .S$, $ {For S' (start) we write $}
- $S \to .CC$, $ {We will look for after S, we have $ so we will write First ($)}
- $C \to .cC$, c/d {For . C we have written C productions so calculate First(C) = {c, d}}
- $C \to .d$, c/d

Step 2.

GOTO(I_0, S)

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

GOTO(I_0, C)

I_2
- $S \to C.C$, $
- $C \to .cC$, $
- $C \to .d$, $

First ($) as we have term after

GOTO(I_0, c)

I_3
- $C \to c.C$, c/d
- $C \to .cC$, c/d
- $C \to .d$, c/d

GOTO(I_0, d)

$C \to .d$, c/d $\longrightarrow I_4$

__/__/____

GOTO(I_2, C)

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

GOTO (I_2, c)

$C \to c.C\ ,\ \$$
$C \to .cC\ ,\ \$$
$C \to .d\ ,\ \$$
} I_6

GOTO (I_2, d)

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

GOTO(I_3, C)

$C \to cC.\ ,\ c/d \longrightarrow I_8$

GOTO(I_3, c)

~~$C \to cC$~~ $C \to c.C\ ,\ c/d$
$C \to .cC\ ,\ c/d$
$C \to .d$
} I_3

GOTO (I_3, d)

$C \to .d\ ,\ c/d \to I_4$

GOTO(I_6, C)

$C \to cC.\ ,\ \$ \longrightarrow I_9$

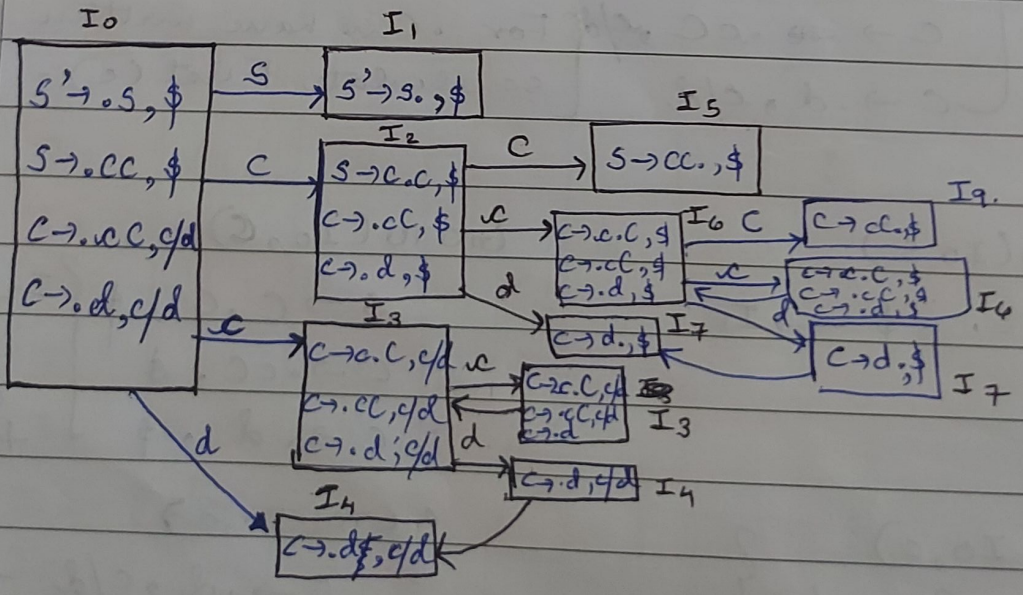
GOTO(I_6, c)

$C \to c.C\ ,\ \$$
$C \to .cC\ ,\ \$$
$C \to .d\ ,\ \$$
} I_6

GOTO(I_6, d)

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

DFA



CLR Parsing Table →

	c	d	$	S	C
I_0	S3	S4		1	2
I_1			Accept		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	R3	R3			
I_5			R1		
I_6	S6	S7			9
I_7			R3		
I_8	R2	R2			
I_9			R2.		

Q2. Check whether the given grammar is CLR or not. If parse the i/p aab

$E \rightarrow TT$

$T \rightarrow aT \mid b$

→ $E \rightarrow TT$

$T \rightarrow aT$

$T \rightarrow b$

→ Adding augmented matrix grammar

$E' \rightarrow .E, \$$

$E \rightarrow .TT, \$$

$T \rightarrow .aT, a/b$

$T \rightarrow .b, a/b$ } I_0

GOTO(I_0, E)

$E'' \rightarrow E. \rightarrow I_1$

goto(I_0, T)

$E \rightarrow T.T, \$$

$T \rightarrow .aT, \$$

$T \rightarrow .b, \$$ } I_2

__/__/____

GOTO (I_0, a)

$T \to a.T, a/b$
$T \to .aT, a/b$
$T \to .b, a/b$
} I_3

GOTO (I_0, b)

$T \to b., a/b \to I_4$

GOTO (I_2, T)

$E \to TT., \$ \to I_5$

GOTO (I_2, a)

$T \to a.T, \$$
$T \to .aT, \$$
$T \to .b, \$$
} I_6

GOTO (I_2, b)

$T \to b., \$ \to I_7$

GOTO (I_3, T)

$T \to aT., a/b \to I_8$

GOTO (I_3, a)

$T \to a.T, a/b$
$T \to .aT, a/b$
$T \to .b, a/b$
} I_3

GOTO (I_3, b)

$T \to b., a/b \to I_4$

GOTO (I_6, T)

$T \to aT., \$ \to I_9.$

GOTO (I_6, a)

$T \to a.T, \$$
$T \to .aT, \$$
$T \to .b, \$$
} I_6

GOTO (I_6, b)

$T \to b., \$ \to I_7.$

	a	b	$	E	T
I_0	S3	S4		1	2
I_1			ACCEPT		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	R3	R3			
I_5			R1		
I_6	S6	S7			9
I_7			R3		
I_8	R2	R2			
I_9			R2		

__/__/____

Parsing the input

Stack	Input	Action
$0	aab$	Shift a and goto 3
$0a3	ab$	Shift a and goto 3
$0a3a3	b$	Shift b and goto 4.
$0a3a3b4	$	

Not Accepted

Stack	Input	Action
$0	abb$	Shift a & goto 3
$0a3	bb$	Shift b & goto 4
$0a3b4	b$	Reduce T → b
$0a3T8	b$	Reduce T → aT
$0T2	b$	Shift b & goto 7
$0T2b7	$	Reduce T → b
$0T2T5	$	Reduce E → TT
$0E1	$	Accept.

RMD in reverse order

LALR PARSER (Look-Ahead LR Parser)

1. LALR parser is Look Ahead LR parser
2. It combines the similar states of CLR parsing table into one single state.

Q. Check whether given grammar is LALR or not

$S \to CC$

$C \to cC / d$

Step 1 Add Augmented Grammar

$S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .d, c/d$
} I_0

Step 2

GOTO (I_0, S)

$S' \to S., \$ \to I_1$ ✓

GOTO (I_0, C)

$S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$
} I_2 ✓

GOTO (I_0, c)

$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$
} I_3 ✓

GOTO (I_0, d)

$C \to d., c/d \to I_4$ ✓

GOTO (I_2, C)

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

GOTO (I_2, c)

$C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$
} I_6 ✓

GOTO (I_2, d)

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

GOTO (I_3, C)

$C \to cC., c/d \to I_8$

GOTO (I_3, c)

$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$
} I_3

GOTO (I_3, d)

__/__/____

GOTO(I6, C)

C → cC. , $ → I9

GOTO(I6, d)

C → d. , $ → I7

GOTO(I6, c)

C → c.C , $
C → .cC , $
C → .d , $

} I6

Merge the similar states →

I3 & I6 production are same but only difference in look Ahead terms hence combine them.

→ I0

→ I1

→ I2

→ I36 C → c.C , c/d/$
C → .cC , c/d/$
C → .d , c/d/$

I4 & I7 production are same but only difference in look Ah
terms so we combine them ....

→ I47 C → d. , c/d/$

→ I5

I8 & I9 rules are same but only difference in look Ahe
terms so we combine them

→ I89 C → cC. , c/d/$

Whether LALR is powerful or CLR is powerful...

__/__/____

Parsing Table →

	ACTION			GOTO	
	c	d	$	S	C
I_0	S36	S47		1	2
I_1			Accept		
I_2	S36	S47			5
I_{36}	S36	S47			89
I_{47}	R3	R3	R3		
I_5			R1		
I_{89}	R2	R2	R2		

Q2) State whether given grammar is LALR or not...

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

→ 1. $S \rightarrow Aa$

2. $S \rightarrow bAc$

3. $S \rightarrow Bc$

4. $S \rightarrow bBa$

5. $A \rightarrow d$

6. $B \rightarrow d$

Step1 Adding Augmented Grammar.

$S' \rightarrow .S$, $

$S \rightarrow .Aa$, $

$S \rightarrow .bAc$, $

$S \rightarrow .Bc$, $

$S \rightarrow .bBa$, $

$A \rightarrow .d$, a

$B \rightarrow .d$, c

} I_0

__/__/____

- GOTO(I0, S)
  $S' \to S \cdot , \$ \to I_1$
- GOTO(I0, b)
  $S \to b \cdot Ac , \$$
  $S \to b \cdot Ba , \$$
  $A \to \cdot d , c$
  $B \to \cdot d , a$
  $\}\ I_3$
- GOTO(I0, d)
  $A \to d \cdot , a$
  $B \to d \cdot c$
  $\}\ I_5$
- GOTO(I0, A)
  $S \to A \cdot a , \$ \to I_2$
- GOTO(I0, B)
  $S \to B \cdot c , \$ \to I_4$
- GOTO(I2, a)
  $S \to Aa \cdot , \$ \to I_6$
- GOTO(I3, d)
  $A \to d \cdot , c$
  $B \to d \cdot , a$
  $\}\ I_8$
- GOTO(I3, A)
  $S \to bA \cdot c , \$ \to I_7$
- GOTO(I3, B)
  $S \to bB \cdot a , \$ \to I_9$
- GOTO(I4, c)
  $S \to Bc \cdot , \$ \to I_{10}$
- GOTO(I7, c)
  $S \to bAc \cdot , \$ \to I_{11}$
- GOTO(I9, a)
  $S \to bBa \cdot , \$ \to I_{12}$

Merge the similar states

→ I_0

→ I_1

→ I_2

→ I_3

→ I_4

→ I_{58} . $A \to d \cdot , a/c$
$B \to d \cdot , c/a$

→ I_6

→ I_7

→ I_9

→ I_{10}

→ I_{11}

→ I_{12}.

Parsing Table →

	ACTION					GOTO		
	a	b	c	d	$	S	A	B
I_0		S3		S58		1	2	4
I_1					Accept			
I_2	S6							
I_3				S58			7	9
I_4			S10					
I_{58}	r5/r6		r5/r6					
I_6					r1			
I_7			S11					
I_9	S12							
I_{10}					r3			
I_{11}					r2			
I_{12}					r4			

__/__/____

OPERATOR PRECEDENCE PARSER →

Operator Precedence grammar is a kind of Shift Reduce parsing method. It is applied to a small class of operator grammar.

→ A grammar is said to be operator precedence grammar if it has 2 properties :

i) No RHS of any production has a ε (epsilon)

ii) No Two Non-Terminals are Adjacent

eg
$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow id$

i) We don't have ε in RHS. So (i) is satisfied

ii) Rule (ii) is satisfied as no 2 adj. N.T are adjacent.

So this operator precedence grammar.

Step 1 We have to check whether grammar OPG or not.

Step 2 If yes then construct precedence table ...

Step 3. Parse input.

STEPS TO SOLVE problem

1. Check whether given grammar is OPG or not.
2. Construct operator precedence relation table.
3. Parse the given input string

> Greater → (Reduce) Consider only terminals.
< Lesser → (Shift)

8. Consider the following grammar. Apply OPG to parse i/p id+id*id
E → E+E / E*E / id

→ As per the rules of OPG, we don't have any ε in RHS of No Two Non Terminals are adjacent. ∴ given grammar is OPG.

2. Construction of operator precedence relation table

Stack ↓ / Input →	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	NO relation	>
$	<	<	<	Accept

1. highest precedence → terminals (id, a, b) etc
2. lowest precedence → $

If operators on both stack & input are same then we write (>)

Eg:-

Stack		Input	
+	(>)	+	greater
*	(>)	+	greater
+	(<)	*	less
*	(>)	*	
id	NO RELATION	id	
$	Accept	$	

3.

__/__/____

8. Parse input id + id * id (Look at Table)

	STACK	RELATION	INPUT	ACTION
$ < id	$	<	id+id*id$	Shift
id > +	$ id	>	+id*id$	Reduce E→id
$ < +	$E	<	+id*id$	Shift as $ < +
+ < id	$E+	<	id*id$	Shift
id > *	$E+id	>	*id$	Reduce E→id
+ < *	$E+E	<	*id$	Shift
* < id	$E+E*	<	id$	Shift
id > $	$E+E*id	>	$	Reduce E→id
* > $	$E+E*E	>	$	Reduce E→E*E
+ > $	$E+E	>	$	Reduce E→E+E
$ A $	$E	Accept	$	Accept

9. Check whether given grammar is OPG or not.

E → EAE | id

A → + | *

→ Rewrite the grammar as given grammar is not OPG as per rule (i) (Two N.T are adjacent to each other).

After rewriting the given grammar is in form of

E → E+E | E*E | id ....

__/__/____

Q. Apply operator precedence parsing to parse the input $a+b*c*d$

Consider the grammar $E \to E+T \mid T$

$T \to T*V \mid V$

$V \to a \mid b \mid c \mid d$

Sol^n... Given grammar is OPG as there is no ε in RHS and no two NT are adjacent.

→ Construction of operator precedence table →

	a	b	c	d	+	*	$
a	–	–	–	–	>	>	>
b	–	–	–	–	>	>	>
c	–	–	–	–	>	>	>
d	–	–	–	–	>	>	>
+	<	<	<	<	>	<	>
*	<	<	<	<	>	>	>
$	<	<	<	<	<	<	Accept

→ Parsing the i/p $a+b*c*d$.

Stack	Relation	Input	Action
$	<	a+b*c*d	Shift a
$a	>	+b*c*d	Reduce V→a
$V	<	+b*c*d	Shift +
$V+	<	b*c*d	Shift b
$V+b	>	*c*d	Reduce V→b
$V+V	<	*c*d	Shift *
$V+V*	<	c*d	Shift c
$V+V*c	>	*d	Reduce V→c
$V+V*V	>	*d	Reduce T→V
$V+V*T	<	*d	Reduce T→T*V

__/__/____

$ ~~V+T~~ $V+T~~*~~	<	* d	~~shift~~ *

The ip is not accepting ...

→ Operator Precedence parsing method is failed to process/ parse the given input d+b* c* d

g. Construct CLR parsing table for given grammar

S → L = R | R

L → * R | id

R → L

Parse the ip string id = id

→ Adding augmented grammar.

S' → . S , $

S → . L = R , $

S → . R , $

L → . * R , =

L → . id , =

R → . L , $

I_0

__/__/____

o) GOTO (I0, S)
S' → S. , $ → I1

o) GOTO (I0, L)
S → L. = R , $
R → L. , $
} I2

o) GOTO (I0, R)
S → R. , $ → I3

o) GOTO (I0, *)
L → *. R $, =
R → .L , $ =
L → .* R , =
L → . id , =
} I4

o) GOTO (I0, id)
L → id. , = → I5

o) GOTO (I2, =)
S → L = . R , $
R → .L , $
L → .* R , $
L → . id , $
} I6

o) GOTO (I4, R)
L → * R. , = → I7

o) ~~GOTO (I6, R)~~
~~S → L = R , $ → I8~~

o) GOTO (I4, L)
R → L. , $ → I8

~~GOTO (I6, R)~~
~~S → L = R , $~~

o) ~~GOTO (I4, *)~~
~~L → * R , =~~
~~R → L , $~~

GOTO (I6, R)
L → L = R. , $ → I9

GOTO (I6, L)
R → L. , $ → I10

o) GOTO (I6, *)
L → *. R , $
R → .L , $
L → .* R , $
L → . id , $
} I11

GOTO (I6, id)
L → id. , $ → I12

GOTO (I11, R)
L → * R. , $ → I13

o) GOTO (I11, L)
R → L. , $ → I10

o) GOTO (I11, *)
L → *. R , $
R → .L , $
L → .* R , $
L → . id , $
} I11

o) GOTO (I4, *)
L → *. R , =
R → L. , =
L → .* R , =
L → . id , =
} I4

GOTO (I6, id)
L → id. , = → I5