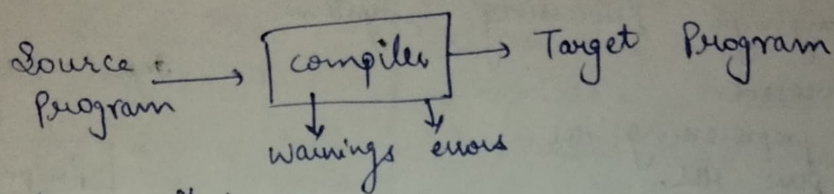
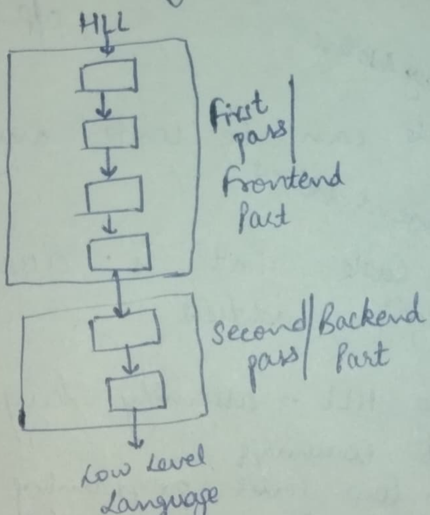
Compiler Design

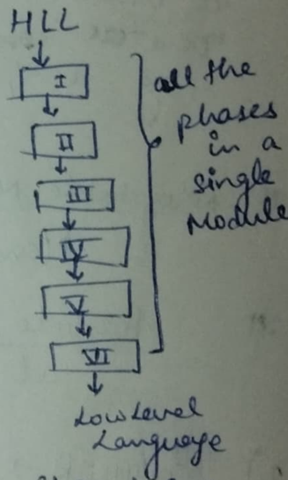
* Compiler → It is a software/program that converts a program written in HLL (source language) to a Low Level Language (object/Target language).
  • It also reports errors present in source programs.



* Types of compilers
1) Single Pass Compilers → Is a type of compiler that processes the source code only once.
2) MultiPass Compiler → It is a type of compiles that processes the source code multiple times (to convert HLL → Low Level Lang) i.e to convert source code to target/object code.



ii) Multi Pass (Two Pass)



i) Single Pass

* There are 2 parts of compilation Process / 2 major phases of Compiler.

Analysis
1) It breaks up the source code/prog. into small parts and creates an Intermediate representation of the source prog.

Synthesis
1) It takes the Intermediate representation of the source prog. as Input and creates the desired target code/program.

* Language Processing Systems :-

We write the programs in a HLL which is easy for us to understand in Compiler design

- These (HLL) programs are fed into a series of tools and OS components to get the desired code that can be used by the Machine. This is called Language Processing System.

Input = HLL / Source Code

↓

[Preprocessor]

↓ Pure HLL

[Compiler]

↓ Assembly Language

[Assembler]

↓ Relocatable Machine Code

[Loader / Linker]

↓

O/P → executable code / Absolute M/c Code

1) Pre - Processor →

- In preprocessing, HLL converted to pure HLL.
- Preprocessor removes the preprocessor directories (# include <stdio.h>) and will add the respective files i.e. file Inclusion
- It will do macro expansion, operator conversion (eg a--; preprocessing a = a-1)

Relocatable M/C Code (Temporary Address) → Code can be loaded anywhere in the Memory.

Absolute M/c Code (Permanent Address) → Code that is assembled to work at one specific address

2) COMPILER → • convert pure HLL → assembly Language

- also gives errors & warnings
- Assembly Language → low level programing language → not in binary form.

3) ASSEMBLER → (is a program) • for every platform (H/W + OS) we have a assembler.

- A assembler for one platform will not work for another platform.

H/W → Intel
OS → window

Assembly code $\xrightarrow{\text{converted}}$ $\boxed{\text{executable M/c Code}}$

Executable M/C Code → Can be loaded at any time & can be run.

4) LOADER / LINKER → Converts relocatable code → absolute M/c code.

A $\boxed{\text{Linker}}$ links diff. obj files into a single file (executable file) & then $\boxed{\text{Loader}}$ — loads that executable file in the Memory & executes it.

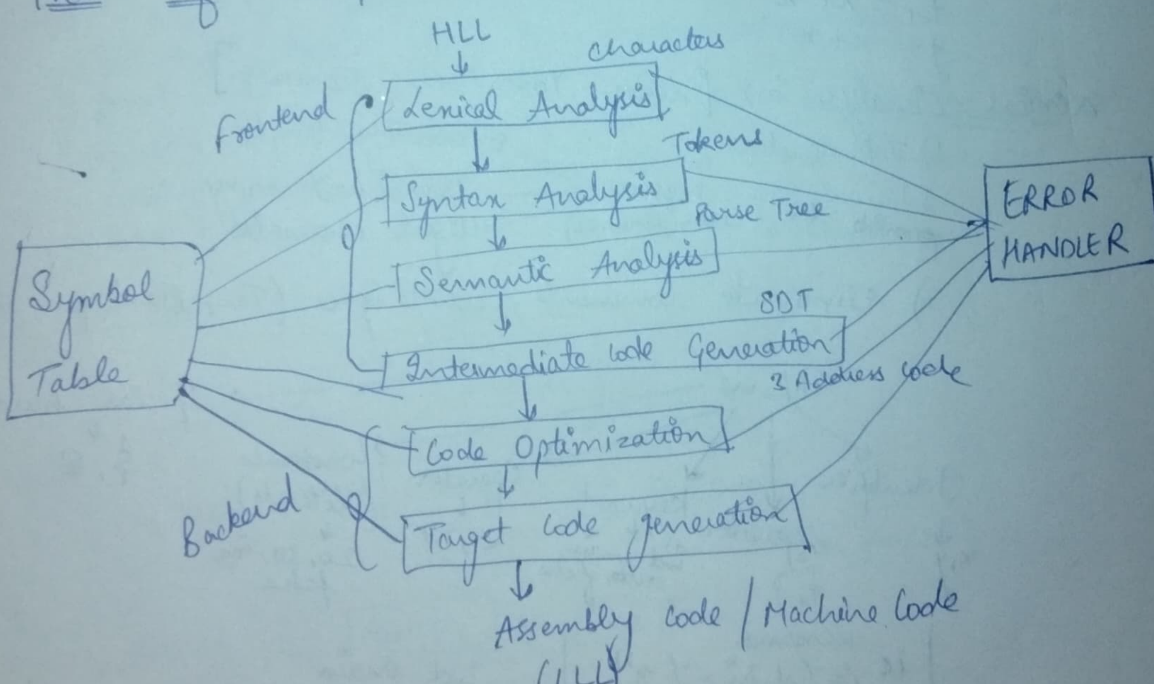
NOTE → If we talk of any compiler like Turbo C, gcc etc. all these phases are included to convert

Source code → M/c code.

* **Compiler** | **Interpreter**

Compiler	Interpreter
1) Takes entire prog at once as Input.	1) It takes single line of code at a time.
2) Speed – high	2) Speed – Low
3) generates Intermediate object code. So Memory requirement is More	3) Memory requirement is less because no Intermediate code created.
4) Eg → C, C++, Scala uses compiler	4) Example → perl, python, Matlab, Ruby
5) Errors → all errors are displayed together	5) Errors → continues translating the prog. until the 1st error is met, in which case it stops.
6) Error detection is difficult.	6) Error detection is easy
7) Compilers are larger in size.	7) Smaller in size.
8)	8)

* Phases of Compiler:-



example:- $y = 2 * x$

Lexical Analysis to convert into stream of tokens

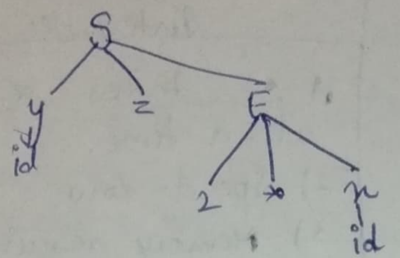
5 tokens: y → Identifier, $=$ → assignment, 2 → constant, $*$ → operator, x → Identifier

$\frac{a}{0}$ ⇒ Run Time Error

1) Lexical Analysis also known as Scanner / Tokenizer

Size, type, scope, reference of variable stored in Symbol Table.

2) Imp phase → Syntax Analysis act as Parser



Compile Time Error

3) Semantic Analysis → check logical errors (scoping, properly declared)

SDT → Syntax Directed Translation

4) Imp Phase → Intermediate Code Generation → 3 Address Code → generate Machine Independent Code.

$t_1 = 2*x$

$y = t_1$

eg $z = a+b*c$

$t_1 = b*c$

$t_2 = a+t_1$

$z = t_2$

} Max. 3 Address] 3 Address Code

5) Code Optimization → Divide Code into Block.

Inside Block → Local optimization

Outside Block → Global optimization

$t_1 = 2*x$

$t_1 = x+x$

$y = x+x$

<u>Lexical Analysis</u> → [Lexer, Tokenizer, Scanner]

1) Tokenization

2) Give Error Message — Exceeding length / Unmatched string / Illegal character

provide Lexical error (Lexemes)

3) Eliminate Comments, white Space (Tab, Blankspace, Newline)

Tokens →
- Identifier → x, y
- Separator → {, }
- keyword → int, main, auto, if else
- Operator → <, >, +
- Constants (literals) → 20, 30, True, false
- Special character → $, @

Lexemes

" i = %d, &i = % x"

1

int, main

int main

1 2

Lexeme → Sequence of characters from the Input that match a pattern.

Tokens → Symbolic names for the entities that make up the text of the program. ex: ID, Constant, Keywords, Operators, Punctuations, Literal String

Source Code ⟶ Lexeme ⟶ Token ⟶ Parser

int x = 5;

Lexeme	Token
int	Keyword
x	id
=	Assignment operator
5	Constant
;	operator

Keyword id operator constant operator

* <u>YACC</u> → It stands for Yet Another Compiler-Compiler (developed by Stephen C Johnson)
  - It is a tool for generating Look Ahead Left-to-Right (LALR) parser.
  - It takes Input from the Lexical Analyzer and generates parse Tree.
  - Syntax Analyzer / Parser is the 2nd phase of the compiler which takes Input as tokens and generates a parse tree.

<u>3 Steps</u> →

Yacc specification ⟶ [Yacc compiler] ⟶ Y.tab.c

Y.tab.c ⟶ [C compiler] ⟶ a.out
↳ LALR parser

a.out ⟶ [Lexical Analyzer] ⟶ parse Tree

LEX TOOL → Lex is a tool / Computer program which generates <u>Lexical Analyzer</u>. It is used with YACC parser / generator.
- 1st phase of compiler which converts source code with HLL → stream of Tokens.
- Lex written by Mike Lesk and Eric Schmidt and described in 1975.

Lex Source program file.l → Lex compiler → lex.yy.c (c program) → Compiler → a.out (Lexical analyzer)

I/P Stream → a.out → Stream of Tokens

* Symbol Table → Data Structures that are used by compilers to hold Info. about source-program constructs.
  - It is used to store Info. about the occurence of various entities such as objects, classes, variable names, functions etc.
  - It is used by both analysis phase and synthesis phases.

used for following purposes →

1) It is used to store the name of all entities in a structured form at one place.
2) It is used to verify if a variable has been declared
3) It is used to determine the scope of a name.
4) It is used to Implement type checking by verifying assignments and expressions in the source code are semantically correct.

A symbol table can either be linear or hash table

example
```
int count;
char x[] = "NESO ACADEMY";
```

Name	Type	Size	Dimension	Line of Declaration	Line of Usage
count	int	2	0	—	—
x	char	12	1	—	—

Operations →

1) Non-Block Structured language.
   - contains single instance of the variable declaration
   - operations:
     a) Insert() → More frequently used in analysis phase (frontend) when tokens are identified and names are stored in table. The Insert() fn takes the symbol & its value in the form of argument.
        eg insert(x, int)
     b) lookup() → used to search a name & It determines
        - The existence of symbol in the table
        - Declaration of symbol before It is used
        - check whether the name is used in the scope
        - Initialization of the symbol.

• checking whether the name is declared multiple items.
lookup(Symbol)

2) Block Structured Languages → variable declaration may happen ~~man~~ multiple times.

a) Insert()
b) Lookup()
c) Set() → Specify [The scope of variable]
d) Reset() → Redefine the scope of variable

Imp'→ T1: $a?\ (b|c)^{*}\ a$
T2: $b?\ (a|c)^{*}\ b$
T3: $c?\ (b|a)^{*}\ c$ if $c? = 1$ then $(b|a)^{*} c$ not processed

Lexical Analyser uses these patterns to recognize 3 ~~typ~~ tokens T1, T2, T3 over the alphabet a, b & c.
Note that $\boxed{x?}$ specifies 0 or 1. occurence of the symbol x
Note also that the anlyser outputs the token that Matches the longest possible prefix.
If the string "bbaacabc" is processed by the Analyser then which one the following is the sequence of tokens it outputs

$\underset{T_1}{\underline{bba}}\ \underset{T_1}{\underline{aca}}\ \underset{T_3}{\underline{bc}}$ — (b)

a) $T_1 T_2 T_3$ b) $T_1 T_1 T_3$
c) $T_2 T_1 T_3$ d) $T_3 T_3$

$\underset{T_2}{\underline{bb}}\ \underset{T_1}{\underline{aa}}\ \underset{T_3}{\underline{cabc}}$ — (c)

→ all options are right (d) is most suitable

d) $\underset{T3}{\underline{bbaac}}\ \underset{T3}{\underline{abc}}$

* Recursive Grammar:→ The grammar G is said to be recursive if in atleast one production there is same variable at both L.H.S & R.H.S

Example: $S \to Sa|b$ Left Recursion
$S \to aS|b$ Right Recursion
$S \to aSb\ |\ \epsilon$ Middle Recursion

Types:→ 1) Left Recursion → The Grammar is said to be left recursion if the leftmost variable of R.H.S. is same as the variable at L.H.S

eg:→ 1) $S \to Sa|b$
2) $S \to AB$
$A \to Aa|b$
$B \to aB|b$

2) Right Recursion → The grammar is said to be right recursion if the rightmost variable of R.H.S is same as the variable of L.H.S.
eg → $S \to aS|b$

Grammar which is both left & right recursion is ambiguous.

eg → $S \to SS \mid AB$
$A \to Aa \mid a$
$B \to Bb \mid b$

$\boxed{\text{NOTE}}$ → If the grammar is left Recursive then the parser goto Infinite Loop so to avoid looping we need to convert the left recursive grammar into right recursive grammar.

* Conversion of LR into RR or Removal of LR →

① $A \to A\alpha \mid \beta$ LR

⇓ removed

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

② $A \to A\alpha_1 \mid A\alpha_2 \mid \text{--} \mid A\alpha_n$
$A \to \beta_1 \mid \beta_2 \mid \text{--} \mid \beta_n$

⇓

$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \text{---} \mid \beta_n A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \text{---} \mid \alpha_n A' \mid \varepsilon$

Example
$E \to E + T \mid T$ LR
$T \to T * F \mid F$ LR
$F \to id \mid (E)$ No Recursion

Soln →
$E \to E + T \mid T$
$A \to A\alpha \mid \beta$
⇓
$A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

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

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

$F \to id \mid (E)$

Eg2 → $S \to \underbrace{SaSb}_{\alpha_1} \mid \underbrace{SbSa}_{\alpha_2} \mid \underbrace{\varepsilon}_{\beta}$

$S \to \varepsilon S'$
$S' \to aSbS' \mid bSaS' \mid \varepsilon$

Eg3 → $S \to S\underbrace{SS}_{\alpha} \mid \underbrace{0}_{\beta}$
$S \to 0S'$
$S' \to SSS'$

Eg4 → $S \to S\underbrace{0S1}_{\alpha} \mid \underbrace{\varepsilon}_{\beta}$
$S \to \varepsilon S'$
$S' \to 0S1S' \mid \varepsilon$

* Grammars → It is basically set of rules that defines the valid structure of a particular language.

G (V, T, P, S) — Start Symbol
Productions
Variables
Terminals

i) Set of terminals (i.e. that terminate, they are not replaced by any other thing further)

ii) Set of non terminals (values/variables that are replaced by terminals). LHS

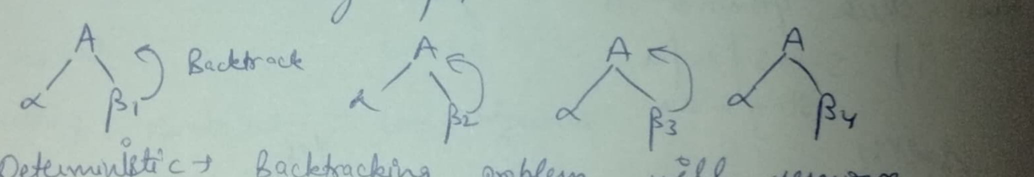
iii) Set of productions →
on LHS → Non Terminal followed by arrow & on RHS we can have T or V or combo of both

iv) Start Symbol → One of the non-terminal is designated as the start symbol from where the production begins.

Classification of grammar

1) Deterministic & Non-Deterministic →

Non-deterministic → for a given grammar if we have many options on single symbol/variable than that grammar is called Non deterministic.
In NDG we have problem of back tracking.
Example $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \alpha\beta_4$
we need string $\alpha\beta_4$



Deterministic → Backtracking problem will remove

$A \to \alpha A'$
$A' \to \beta_1 / \beta_2 / \beta_3 / \beta_4$

This process is known as left factoring or eliminating non-deterministic.

2) Recursive and Non-Recursive →

Left Recursive → If the leftmost symbol in RHS is equal to symbol of LHS.

$A \to A\alpha / \beta$

Right Recursive → If the rightmost symbol in RHS is equal to symbol of LHS.

$A \to aA / \beta$

3) Ambiguous and Non-Ambiguous →

A CFG is said to be ambiguous if there exists more than one derivation trees for the given i/p string. or
More than one Leftmost derivation.
or " " " rightmost "

Example: $E \to E+E \mid E*E \mid id$

we have to derive $id+id*id$

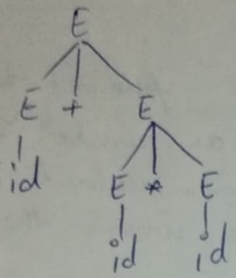
Sol^n:

LMD: $E \to E+E$
$id+E$
$id+E*E$
$id+id*E$
$id+id*id$

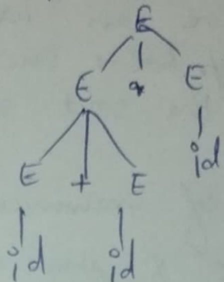
RMD: $E \to E+E$
$E+E*E$
$E+E*id$
$E+id*id$
$id+id*id$

RMD: $E \to E*E$
$E*id$
$E+E*id$
$E+id*id$
$id+id*id$

Parse Tree-1 (Is Valid)



Parse Tree-2



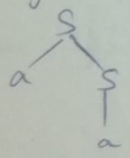
So we have more than 1 parse tree possible ∴ Grammar is ambiguous.

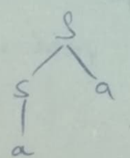
Ambiguous in the sense that "if we have 2 parse trees possible then parser will get confused about which one to generate or which one is correct.

Ques. Check Grammar is ambiguous or not?

i) $S \to aS \mid Sa \mid a$
String – aa

Sol^n:



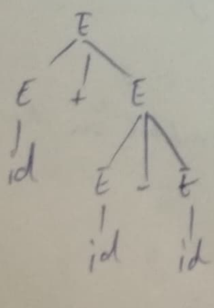


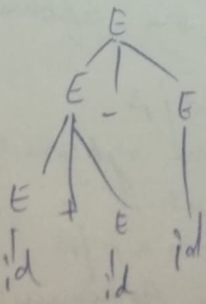
Yes

ii) $E \to E+E$
$E \to E-E$
$E \to id$

$id+id-id$

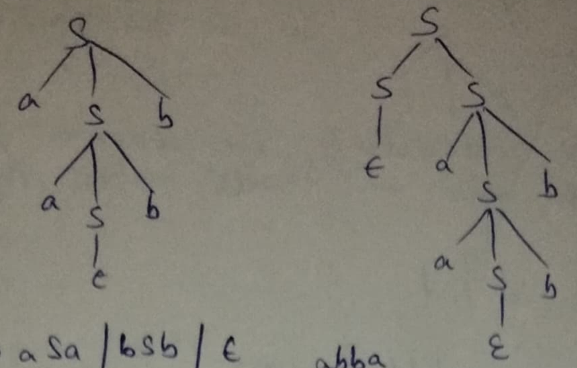
Sol^n:





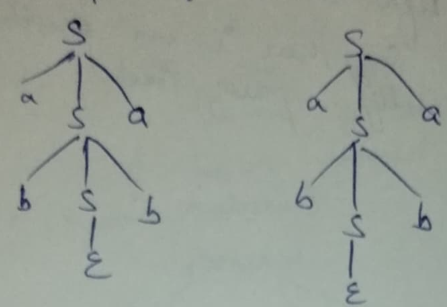
Yes

iii) $S \to aSb \mid SS$ aabb
$S \to \epsilon$



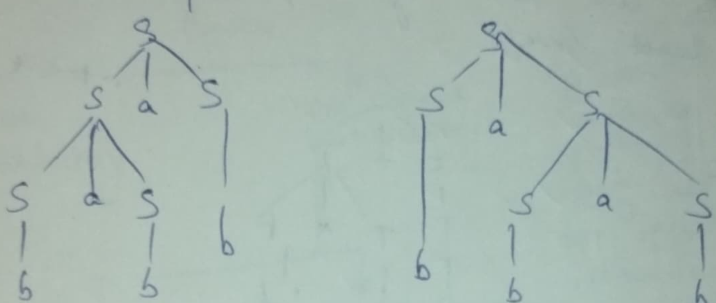
Yes

iv) $S \to aSa \mid bSb \mid \epsilon$ abba



Unambiguous.

v) $S \to SaS \mid b$



Yes

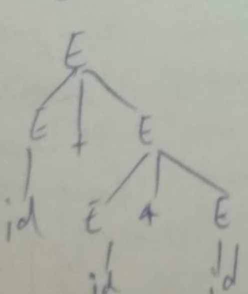
babab   babab

* Converting Ambiguous → Unambiguous.

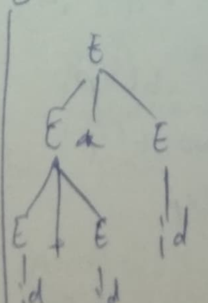
$E \to E+E \mid E*E$
$E \to id$

① id+id+id   Grammar failed Because rules of associativity failed

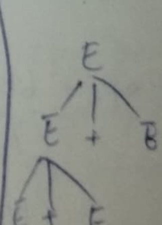
② id+id*id   Grammar failed ∵ precedance is not taken care of



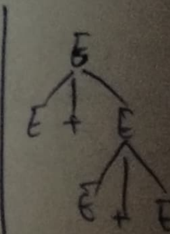
* evaluated 1st so correct



+ is evaluated yet ∴ Wrong

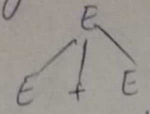


Left Associative



Right Associative

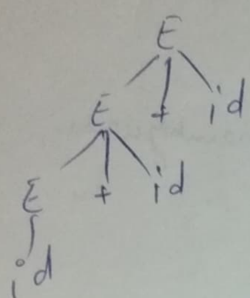
Getting Both associativities ∵ we are defining the GRAMMAR without any order



So we can grow in any direction.

To achieve left associativity, we have to grow in left direction only. So now we will restrict the growth of parse tree.

$E \to E + id \mid id$



Left Recursive

So, there is no possibility of getting diff. parse Tree.

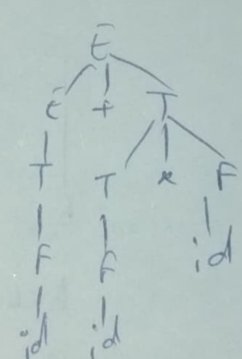
<u>for precedence Problems</u>

we should take care that highest precedence operator should be at the least level.

$+ < *$

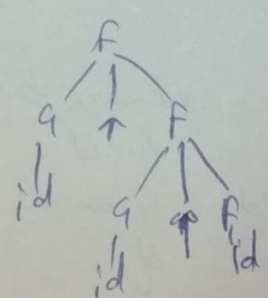
$E \to E+T \mid T$
$T \to T * F \mid F$
$F \to id$

id+id*id



<u>Example2</u> $2 \uparrow 3 \uparrow 2$ $\quad 2^{3^2}$ Right Associative

$F \to G \uparrow F \mid G$
$G \to id$



<u>Eg3</u> bExp → bExp OR bExp
/ bExp AND bExp
/ NOT bExp
/ True
/ False

OR — Less
AND
NOT — least level

Precedence
NOT > AND > OR

OR, AND] Left Associative

<u>Soln</u> $E \to E \text{ Or } F \mid F$
$F \to F \text{ And } G \mid G$

$G \to \text{Not } G \mid \text{True} \mid \text{false}$

Eg:-

$A \to A \$ B \mid B$
$B \to B \# C \mid C$
$C \to C @ D \mid D$
$D \to d$

Tell associativity & precedence

Sol^n:-

$\$ \to$ Left Associative
$\# \to$ " "
$@ \to$ " "

Precedence
$@ > \# > \$$

Eg:- $E \to E + F \mid E * F$
$F \to id$

Sol^n:- + and * are at same level
∴ same precedence
& both defined as left Associative

* **Parsers:-** (Syntax Analyzer)

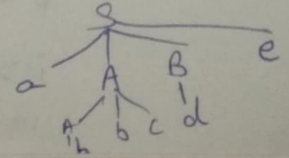
- Parsers
  - Top Down Parser
    - TDP with full backtracking → Brute force method
    - TDP without backtracking
      - Recursive Descent Parser
      - Non-Recursive Descent LL(1) Parser
  - Bottom up Parsers → or Shift Reduce Parsers
    - operator Precedence parser → only used for operator Grammar
    - LR Parsers
      - LR(0)
      - SLR(1)
      - LALR(1)
      - CLR(1)

LR parsers can scan string from L→R
use reverse of RMD

* **Bottom up parser:-** generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e, it starts from the terminals and ends on the start symbol.
It uses reverse of Right most derivation.

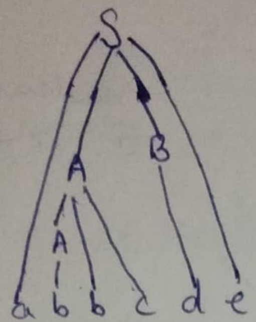
Top Down:- generates parse tree for the given I/P string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol & ends on the terminals.
It uses LMD.

Eg:- $S \to aAbe$ $\quad A \to Abc \mid b$ $\quad B \to d$



abbcde

Top Down → what is the next production we should use
Bottom Up → when to reduce the given terminal



* Operator Precedence Parser :- Ambiguous grammar are not allowed in any parser except operator precedence parser.

Eg :- $E \to E+E \mid E*E \mid id$

Soln operator relation Table / Operator precedence relation table

	id	+	*	$
id	–	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	–

id > +

- Two id's will never compared b/c they will never come side by side.
- Identifier will be given highest precedence compared to any other operator
- $ has least precedence compared to any other operator
- + is left associative
  1+2+3
  3+3
- * precedence > + precedence

Input → id+id*id

↑ look ahead :- Pointer

id + id * id $
↑ ↑

| $ | id |

Stack

$ < id Push id

id + id * id $
↑

| $ | id |

id > + Pop E
id – E id + id * id
$ < + Push

id + id * id $
↑

| $ | + |

id + id * id $
↑

| $ | + | id |

id > + Push
id > * Pop id – E id + id * id

id + id * id $
↑

| $ | + |

Top of stack < look ahead $\Rightarrow$ Push.

| $ | id | + | id | * |

id + id * id $

| $ | + | * | id

id+id*id$

| $ | + | * | ~~id~~

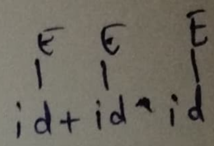
| $ | + | ~~*~~

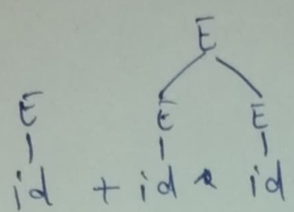
* < id Push

id > $ Pop

id - E

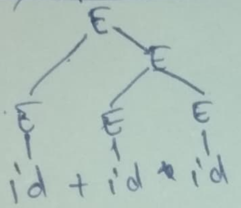
* > $ Pop





| $ | ~~+~~ |

+ > $ Pop



Disadvantage ↳ No. of entries i.e. if we have 4 operators we will have 16 entries
10 operators we will have 100 entries
for N operator → $O(n^2)$ Size of table will be very big

So to dec. the size of the table, we use operator fn table.