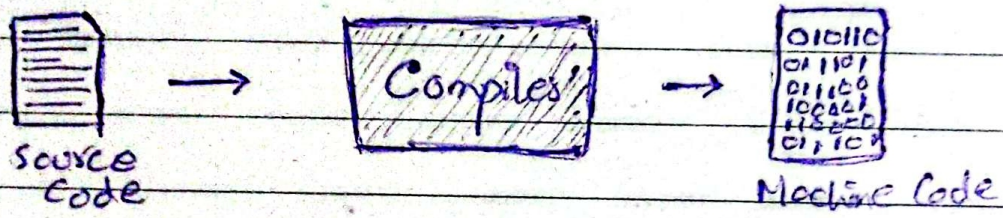
Date _____ Day _____

WEEK #01

→ Programming languages — Notations for describing computations to people and to machines.

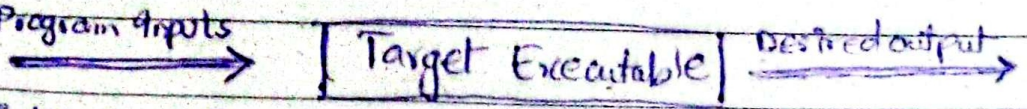
→ A language understood by a machine is called Machine language (ML), Machine Code, Object Code
  ↳ Consist of streams of 0's & 1's

→ The program that carries out the translation activity is called compiler



* A compiler is a program that translates source code written in one language into the target code of another language *

→ Executables — Files that actually do something by carrying out a set of instructions. Eg: .exe file

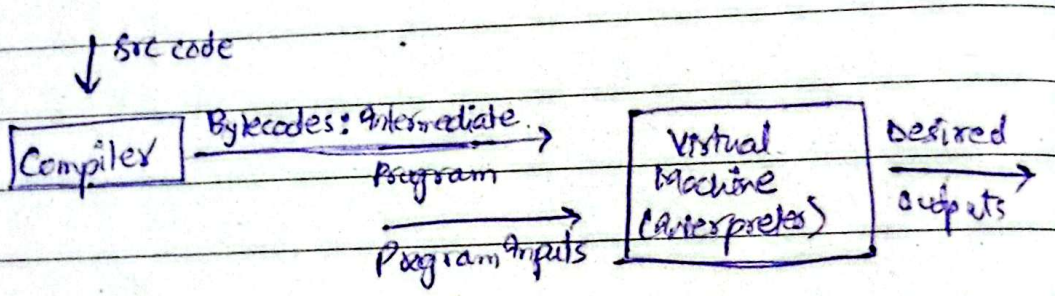


↳ Initially there is a compile phase followed by the run/execute phase

Date _____ Day _____

- Interpreter
  - ↳ A program that doesn't produce the target executable
  - → It reads the code line by line & directly executes the code by using given inputs

- Java Combines Compilation & Interpretation



Portability: Bytecode compiled on one machine can be interpreted on another one

Faster Execution thru Just-In-Time Compilers (JIC)

→ In compiler based language compiler & src code not needed to run .exe. Whereas interpreter requires both

→ In compiler if multiple error it shows them all whereas in interpreter it only shows 1st one

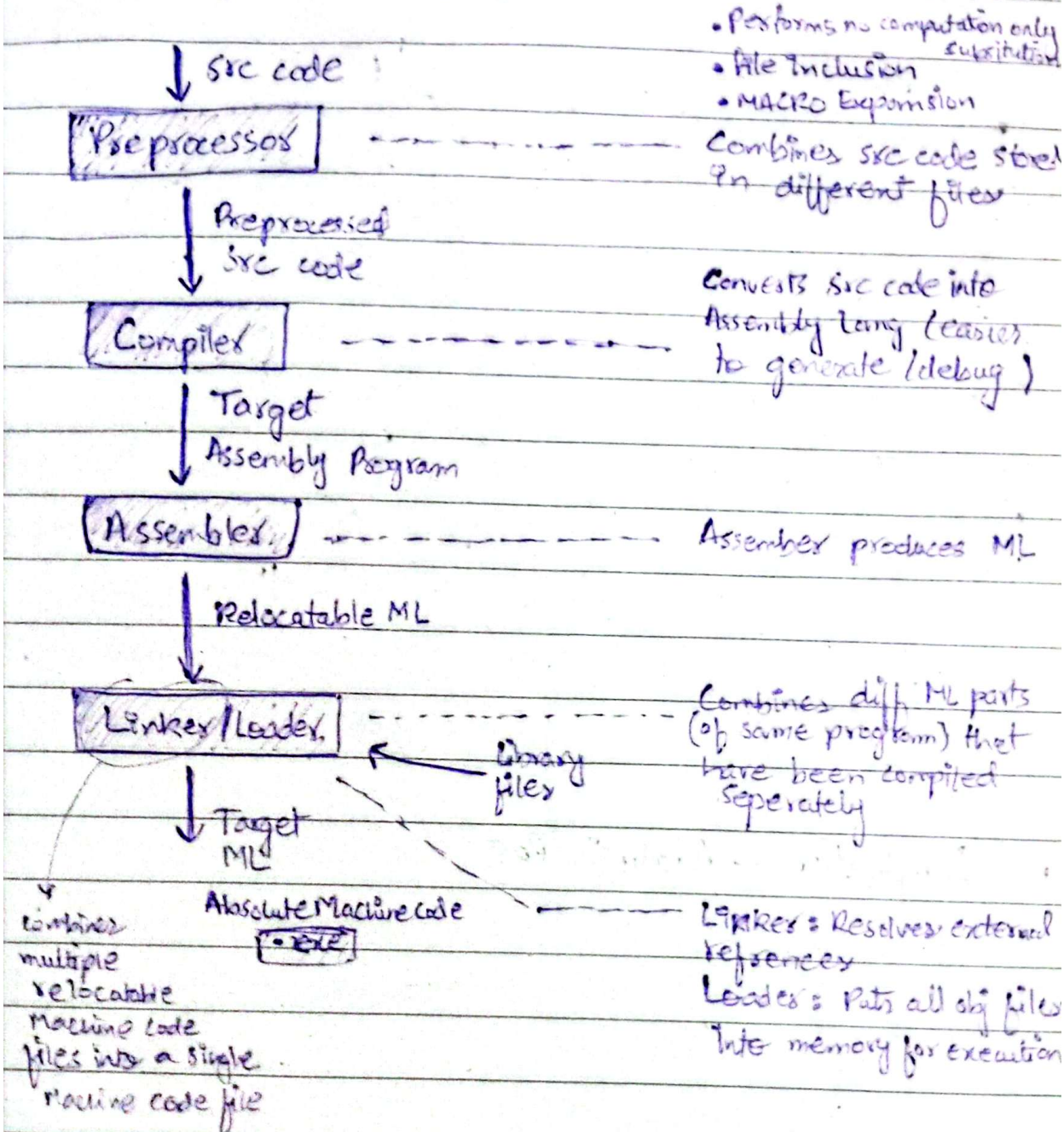
→ Compiles based language → Fast

→ Interpreter takes less space

→ In Interpreter debugging easy

Date _____ Day _____

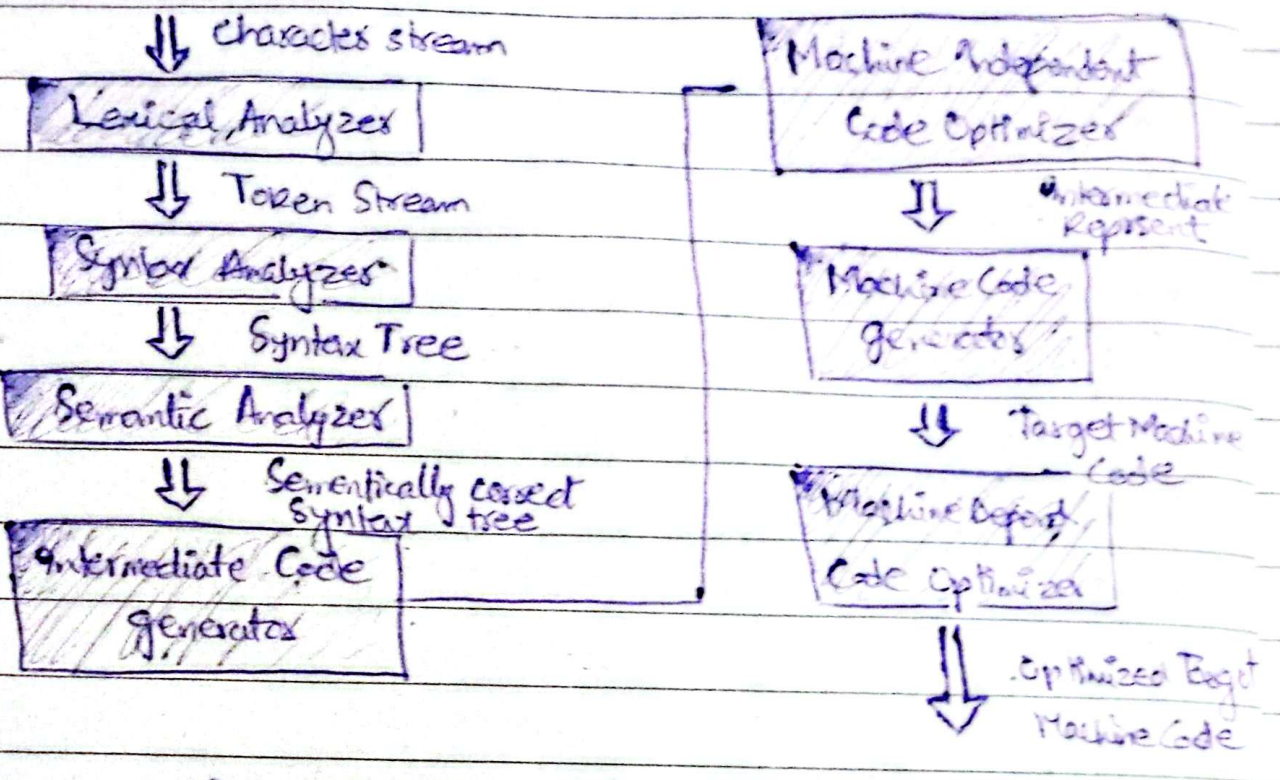
• Language Processing System:



Date _____ Day _____

These all phase don't run sequentially

▷ Structure of A Compiler



▷ Compilation – Analysis Part

→ Phases Involved

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis

↳ Determines the operations implied by the source program which are recorded in a tree structure called **Syntax Tree.**

↳ Breaks up src code into constituent pieces while storing info in symbol table

Date _____ Day _____

▷ Compilation — Synthesis Part

→ Phases involved

① Intermediate code generation
② " " Optimization
③ Machine Code Generator
④ " " Optimizer

→ Both intermediate & machine code are optimized to achieve efficiency

↳ Constructs target code from Syntax Tree & from info in Symbol Table

▷ Frontend & Backend

- Lexical Analysis
- Syntax "
- Semantic "
- Intermediate Code Generation
- " " Optimization

} Frontend (Machine Independent)

- Machine Code generation
- " " Optimization

} Backend (Machine Dependent)

① Lexical Analysis

Let $x = a + b * c$ ⇒ Lexical Analysis generates Lexems & tokens

→ Lexems are similar to words but they individually don't have any meaning

Date _____ Day _____

→ Lexical Analysis Recognizes the tokens using regex

Eg: Regex for identifier

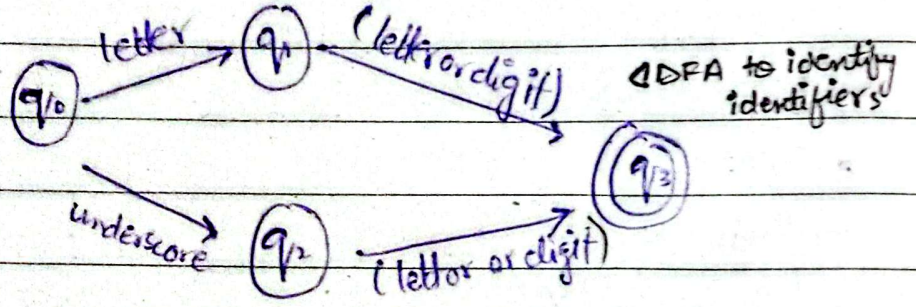
$l(l+d)^* \mid \_(l+d)^*$

l = letter

d = digit

_ = underscore

Lexems	Token
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier



→ It reads the character stream of source code and break it up into sequences call lexems. (A.K.A Scanning)

→ Each lexem is represented as token

② Syntax Analysis

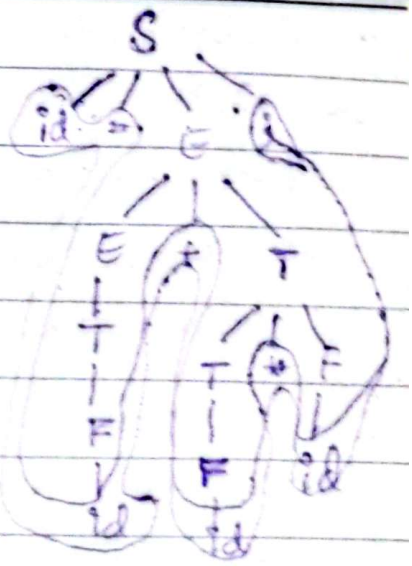
→ Also known as Parser

→ The stream of token is pass to this phase to identify grammatical structure of the stream.

→ Syntax analyzer depends on Type 2 or Content Free Grammars

Date _____ Day _____

→ It builds a parse tree, which replaces the linear sequence of tokens with a tree structure
→ The tree is built according to the rules of grammar.
→ It is analyzed, augmented & transformed by later phases



(3) Semantic Analyzer

→ Takes the parse tree as input
→ Performs semantic checks eg:
- Type checking
- object binding
- rejecting incorrect programs
- issue warning
- correctness of scope resolution
- multiple variables declaration

• undeclared variables

→ Performs logical analysis of parse tree

(4) Intermediate Code Generator

→ Takes the semantically verified parse tree
→ Three Address Code is used [3 operands/instruction]

$t_0 = b * c$
$t_1 = a + t_0$
$x = t_1$

→ bottom to top fashion to preserve precedence of parse tree

Date _____ Day _____

⑤ Code Optimizer

→ Takes the intermediate code as input and optimizes it.

→
```
t0 = b * c;
x = a + t0;
```
} optimized to 2 instructions

⑥ Target Code Generator

→ Converts the optimized intermediate code to target machine code.

→ This is done thru assembly language

⑦ Machine Dependent Optimizer

→ Optimizes the machine code

→ It is optional activity

⑧ Error Handler

→ It is a subroutine to take care of the continuation of compilation, even any error at any phase ie error handler is responsible to continue the compilation process even any error occurs at any phase

Date _____ Day _____

WEEK #02

▷ SYMBOL TABLE MANAGEMENT

→ Data Structure used for storing the names of variables and their associated attributes
- variables
- objects
- function
- classes

→ It should be design so that the compiler can
  ↳ find record for each variable quickly
  ↳ Store/retrieve data for a record quickly

→ Symbol table is ~~used~~ by Lexical analysis, Syntax & made & updated by Semantic Analysis. Then used by last three phases

Lexical Analysis → Creates entries for identifiers

Syntax Analysis → Adds info regarding attributes

Semantic Analysis → Using available info checks Semantics & updates the Symbol table

Intermediate → Available info helps in adding temporary variables information

Code Optimization → Info is used in machine dependent optimization

Target Code Gen → Generates the code using address info of identifiers

Date _____ Day _____

→ Attributes of table

1. Name → name of var
2. Type → type [int, float--]
3. Size → How many bytes
4. Line of Declaration → ∞
5. Line of Usage → if used in multiple place this will be a linklist
6. Address → Address info of identifies

- Non-block structured Language (FORTRAN..)
  - → Contains single instance of the variable declaration
  - → Operations
    - i- Insert() ii- Lookup()
- Block Structured Language (C, C++, JAVA)
  - → Variable decleration may happen multiple times
  - → Operations
    - i- Insert() ii- Lookup() iii- Set() iv- Reset()

defining & redefining
scope

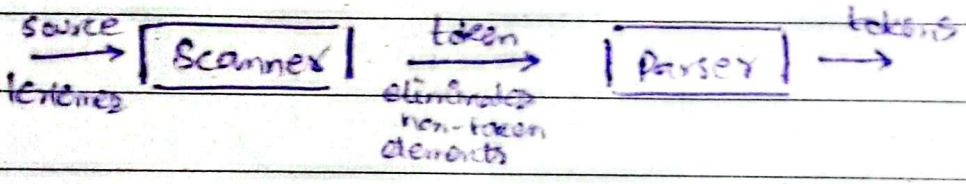
Date _____ Day _____

▷ Parsing & Scanning

→ In compilers the recognizer is split in to two phases

(1) Scanner → Translate input characters to tokens
↳ Also, report lexical errors like illegal characters and illegal symbols

(2) Parser → Read token stream and reconstructs the derivation (parse tree)



→ These two modules are seperate to maintain simplicity & seperation of concerns
↳ Scanner hides details from parser (comments, whitespace, input files etc)
↳ Parser therefore has simpler stream of tokens and also due to efficiency
↳ Scanner uses simple faster design but still consumes a suprising amount of the compiler's total execution time.

Date _____ Day _____

Tokens

↳ It is like a class which is assigned to each lexem (character of src code)

↳ Some tokens have attribute associated with it

- Operators & Punctuations
  - ↳ +, /, ., ;, *, (,), {, }, <, = ---
    - ↳ Each of these has seperate lexical class (token name)
- Keywords
  - ↳ while, do, int, float, switch, if, else ...
    - ↳ Each of these also have seperate class
- Identifier
  - ↳ All identifiers have class ID, with a value that represents it's "id" in symbol table
- Integer constants
  - ↳ They have INT class, with the value of that constant. INT (42)

return maybe b = iffy ;

[RETURN] [ID (maybe)] [NEQ] [ID (iffy)] [SCOLON]

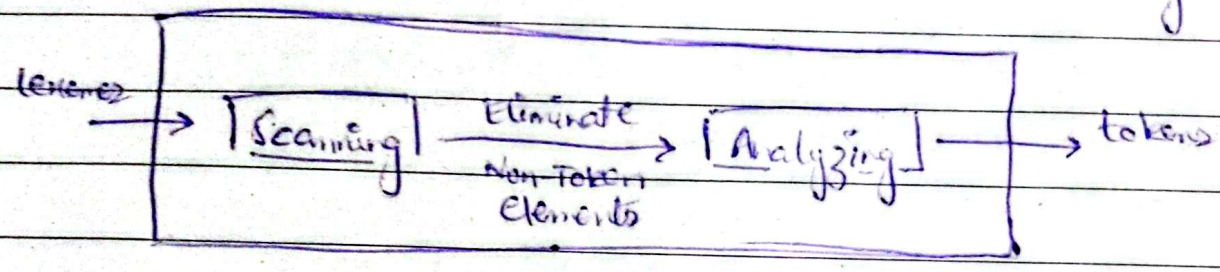
Date _____ Day _____

- Lexical Analyzer

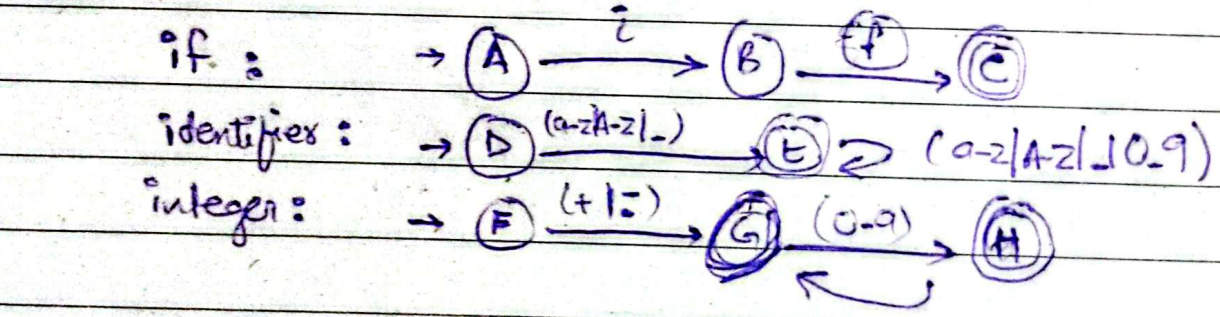
↳ Scans the pure High Level langiage (HLL) code line by line

↳ Takes lexemes as input & produces tokens as o/p

↳ Uses DFA for pattern matching



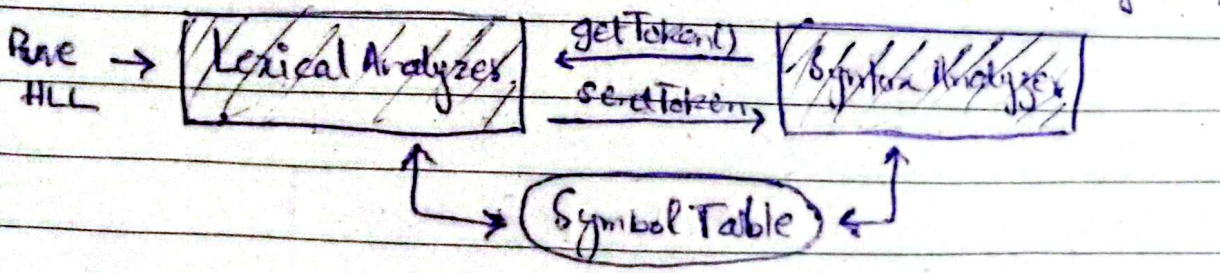
C - tokens:



↳ Removes comments & whitespaces from pure HLL code

' ' Space
'\t' horizontal tab
'\n' newline
'\v'
'\r' carriage return

↳ Helps in MACRO expansion in HLL code



Date _____ Day _____

→ Lexical Analyzer helps in giving error message by providing row no & col no.

→ Sabse phele Syntax Analyzer chalta hai wo get_token() call karta hai or phir LA send_token()

→ Agr syntax Analyzer ko esa token recv hoa which was not expected by the parse tree so it reports to error handler.

→ But it doesn't stop!

→ LA used DFA to do tokenization.

→ While doing tokenization always gives importance to longest matching

```
main()
{
    a2 = ab * c ;          12
    int a, b, c;           21
    y = x = 3 ;            27
}                          28
```

This program has syntax & semantic error but lexical analyzer will not stop ∴ there is no lexical err

```
char * c = "string";       6
in t f = 200 ;             6
in /* comment */ t f = 200;  6
```

↳ comment hata ke space ajayega

Date _____ Day _____

→ x = = 4 = y /* abc * * * / ; abc df ; / ; 10

→ int * * * p ; 6

→ x = x p. + + + + y ; 9

→ char ch = "Hello ; ?

↳ Token error (Strings Not closed)

- Lexical Errors

① Identifiers that are way to long

② Exceeding length of numeric constant

③ Numeric constants which are ill-formed

int i = 457 $ 92 ;

④ Illegal characters that are absent from the src code

char x[ ] = " Hello " ; $ → this is not any keyword or an identifier

- Lexical Error Recovery

↳ Panic-mode recovery

↳ If we encounter error than skip the sucessive characters until delimiter is found

↳ Transpose of two adjacent characters

↳ while (——) ⇒ whiel , ill styled

↳ insert a missing character

↳ it sam ; ⇒ int sam ;

↳ Delete an unknown / extra character

↳ intt a ; ⇒ int a ;

↳ Replace a character

itt a ; ⇒ int a ;

Date _____ CONTEXT FREE GRAMMAR Day _____

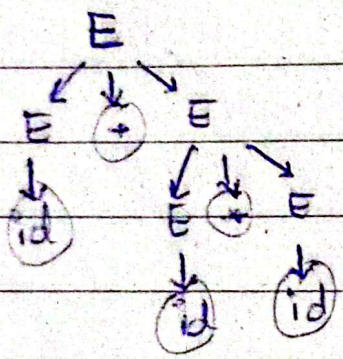
▷ AMBIGUOUS GRAMMAR

→ Ek grammar k liye koi esi string exists krti ho jiske liye ek se zyada parse tree bn sakte ho to it is ambigous.

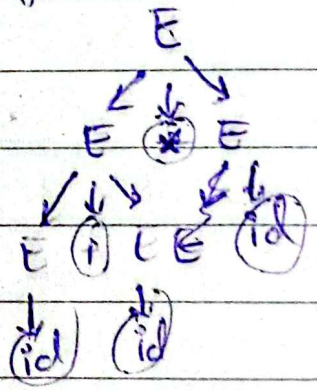
$E \rightarrow E+E \mid E*E \mid id$

$w = id+id*id$

Left Most Derivation:



Right Most Derivation -



▷ Converting Ambigous To Unambigous Grammar

(1) Left Recursive Grammar

$S \rightarrow Sb \mid a$

right hand expression mein agr Non terminal extreme left par ho.

(2) Right Recursive

$S \rightarrow aS \mid b$

extreme right par non-terminal

Date _____ Day _____

▷ SYNTAX ANALYZER

Stream of Tokens ↓

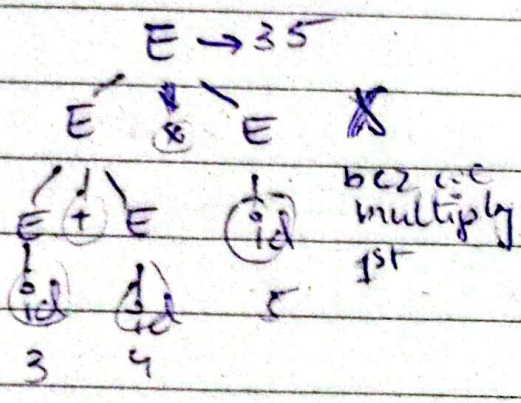
[Context Free grammar] ⇒ [Parser]

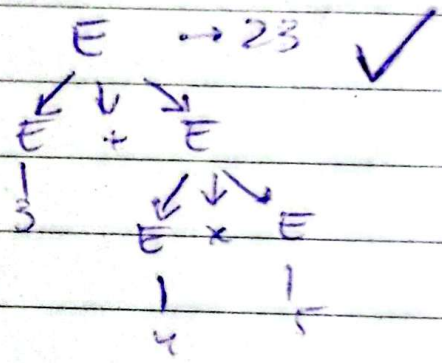
- Problems Due to Ambiguity

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

W = id+id*id
3 + 4 *5

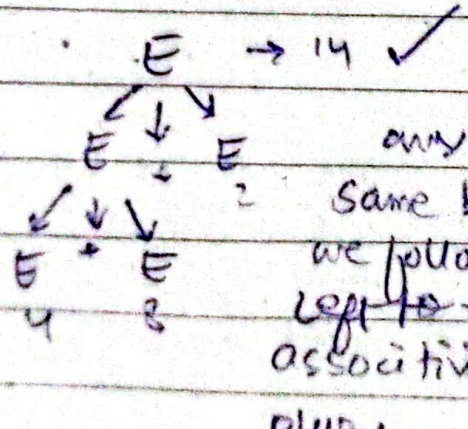
3+(4*5) → Precedence property violation



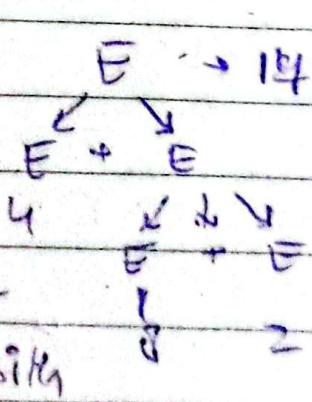


Associativity property voilation

W = 4+8+2 (4+8)+2



any is same but we allow left to right associativity with plus.

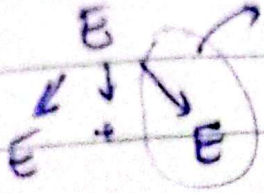


Day

Date

- Solution for Associativity

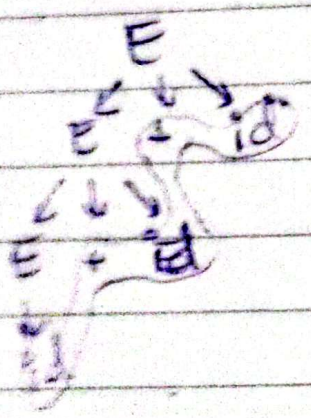
$E \to E + E \mid id$



this is causing the issue. Since '+' is L→R Associativity we will replace the right most terminal so the tree doesnot grow in that direction

$E \to E + id \mid id$

- We converted the grammar to Left recursive to ensure L→R associativity



- For R→L associativity we convest to Right Recursive

$E \to id + E \mid id$

- Solution for Precedence

we keep this down to levels to remove violations

'+' is LR associative we replace the right non-terminal

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

$E \to E + T$

$T \to T * T$ → But (x) is also LR associative so we change it to $T * F$

$E \to E + T \mid T$ → added this if we only want multiplication

$T \to T * F$

$F \to id$

With the precedence symbols we lower level first then change

Date _____ Day _____

low precedence will be on top

Q. bExp → bExp AND bExp | bExp OR bExp | NOT bExp | True | F.

bExp → bExp OR bExp1 | bExp1

bExp1 → bExp1 AND bExp2 | bExp2

bExp2 → NOT bExp2 | True | False

Q: R → R+R | R.R | R* | a | b | c

Precedence → * , . , + (High → Low)

Associativity → Left → Right

R → R + R1 | R1

R1 → R1 . R2 | R2

R2 → R * R2 | F

F → a | b | c

▷ Problem with Left Recursion

Left Recursive

↳ A → Aα | β

↳ If we consider this in form of code

```
A()
{
    A(); → it goes into infinite recursion.
    α;   → this statement never gets executed!
}
```

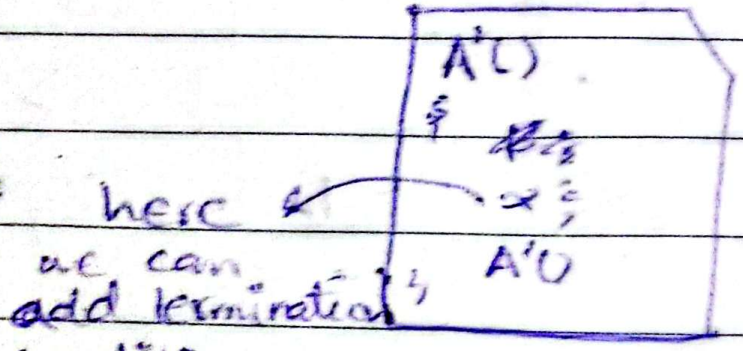
↳ But this is only a problem in Top Down Parser

Date _____ Day _____

Therefore we need to convert ~~E~~ left recursive to right recursive without changing the language

$A \rightarrow \beta A'$

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



here we can add termination condition.