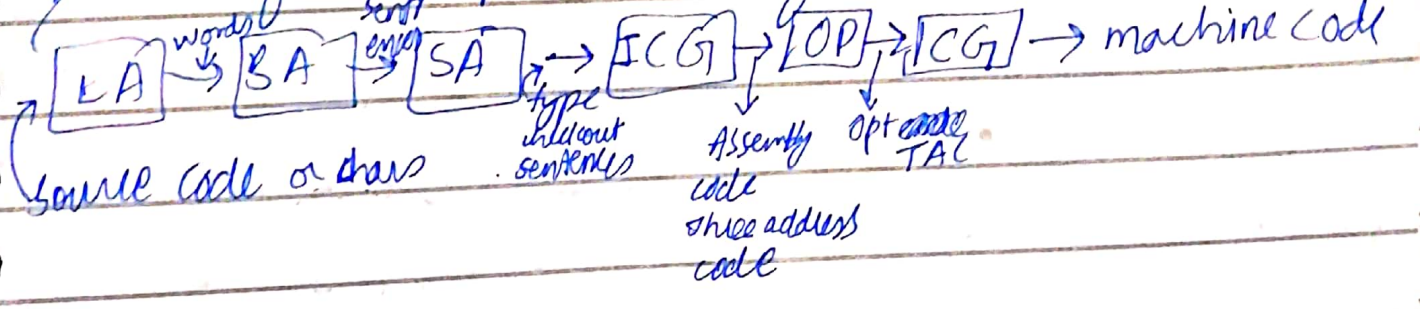
Compiler Construction

1. Lexical Analysis
2. Syntax Analysis (Parser)
3. Semantic Analysis
4. intermediate code Generation
5. Optimization
6. code Generation or interpretation / VM

* Parts of a pipeline all of them



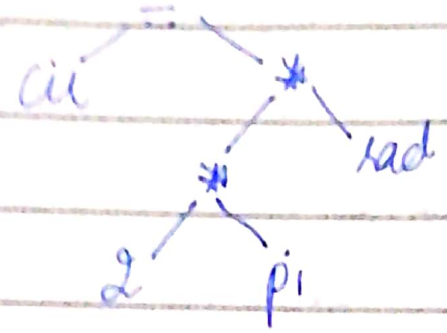

```
int x=2; int y;
int foo() { return ++x; }
cin >>y;
if (y >10 || foo() >5)
    cout << x <<endl;
cout << x;
```

Short circuit (Yes)	Short circuit (No)
y=15 2,2	3,3
y=5 ③	③

```
int sum = 100;
```

(INT, 1), (ID, "sum"), ('=', X), (NUM, 100), (';', X)

deg = 2 * Pi * rad



Syntax error (Pause)

(a) int x sum;
(b) x : int;
(c) a = b x t;

Semantic error (captured by semantic analyzer)

int x = foo(); → return ~~float~~ string

IC G) Three Address code

c = a + b - c

t1 = a + b
t2 = t1 - c
x = t2

optimizing →

t1 = a + b
x = t1 - c

Example machine code

0 10 14 18

Symbol Table → Data structure used in every phase of compiler (holds information about identifiers (var names, functions).

```
int foo() {
int x, y;
char ch;
double sal;
....
```

Symbol Table

Name	Type	Address
foo	FUN	0
x	INT	0
y	INT	4
ch	char	8
sal	DOUBLE	9

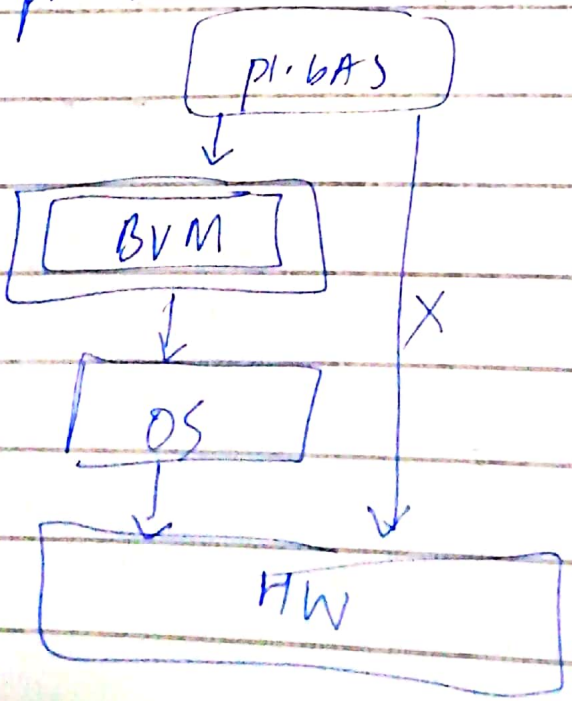
Because var in data segment. function in code segment

First 2 phases generate this info, then other phases use this information.

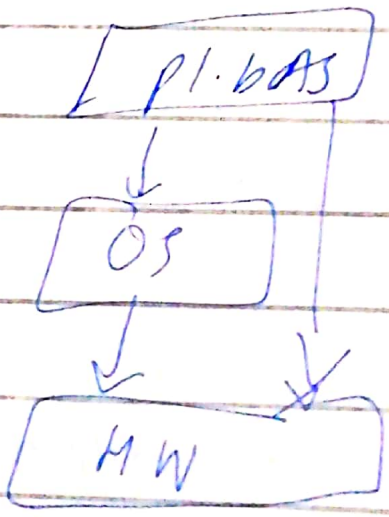
Types of Language Processors

1. Compiler
2. Interpreter
3. Hybrid systems

~~Compiler~~ Interpretor



Compiler

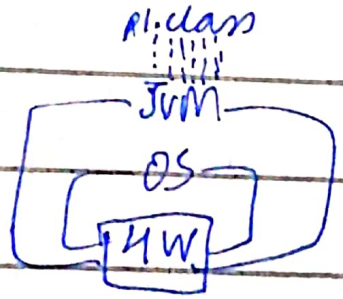


compiler	Interpreter
(1) FAST execution	(1) Slow execution
(2) Source code protection	(2) NO protection here because we need to give source code
(3) No scene here	(3) Portability (Platform independence)
(4) Less debugging	(4) Better debugging

out of array index error → given sometimes by C → compiler used, given everytime by JAVA → interpreter

Java uses hybrid → compiler + interpreter → produces class file

pl.java → [javac] → pl.class




```
for(int i=0 ; i<arr.length ; i++){
    if(special(arr[i])){
```

```
token = strtok(all, " ");
while (token != NULL){
    for (int i=0; i< token.len(); i++){
        if (special(token[i])){
            cout << "\n";
        }
        cout << token[i];
    } //for end
    token = strtok(NULL, " ");
}//while end
```

func special = returns true if character is +, -, ÷, *, ;, %, ,

Regular Expression (Let

(1) ϕ is an RE. If $r \in \Sigma$ then "r" is RE

(2) If "r" and "s" are RE then so are.

(a) $r+s$, $r|s$ (b) rs (c) (r) (d) r^*

→ All possible strings over a and b

$(a+b)^* = [\lambda, a, b, aa, ab, ba, bb, \ldots]$

[Lexicographic ordering:-

[Dictionary ordering Exactly 1 a

$(a + ab^* + b^*a)\ b^*ab^*$

Exactly two a's

$b^*ab^*ab^*$

At least 2 a's

(3) At most three a's

$(b^*ab^* + b^*ab^*ab^* + b^*ab^*ab^*ab^* + b^*)$

or $b^*(a+\lambda)b^*(a+\lambda)b^*(a+\lambda)b^*$

(4) Third digit is 1 from left

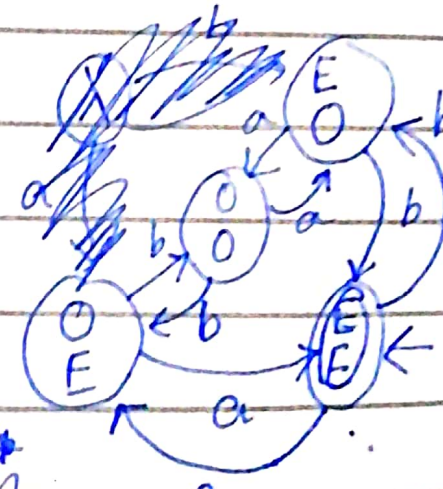
$(0+1)^*1(0+1)^*$

(5) Third digit is 1 from right

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

abaabba

Even-Even



$aa + bb + (ab + ba)(aa + bb)^*(ab + ba)$

Is naming alag alag tarah samjh ai

Regular Definition → (No recursion here)

(A|B|C|D---|Z|a|b|c|---|z)*

" " (0|1|2|3---|9)*

id → letter (letter|digit)*

letter → A|B|---|Z|a|b|c---|z

digit → 0|1|2|3---|9

CFG → recursion allowed but * not allowed

Easy to use interface, details hidden

Regular Exp for numeric constant:-

(-|+|λ)(Numeric)*(.)(+|λ)(Numeric)*(E+|λ)(-+|λ)(Numeric)*

N → OS digit+ OF OE

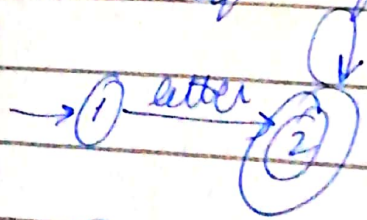
OS → λ | + | -

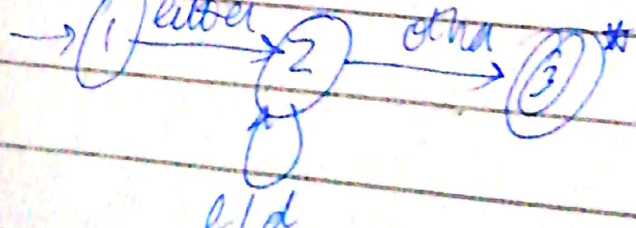
digit → 0|1|---|9

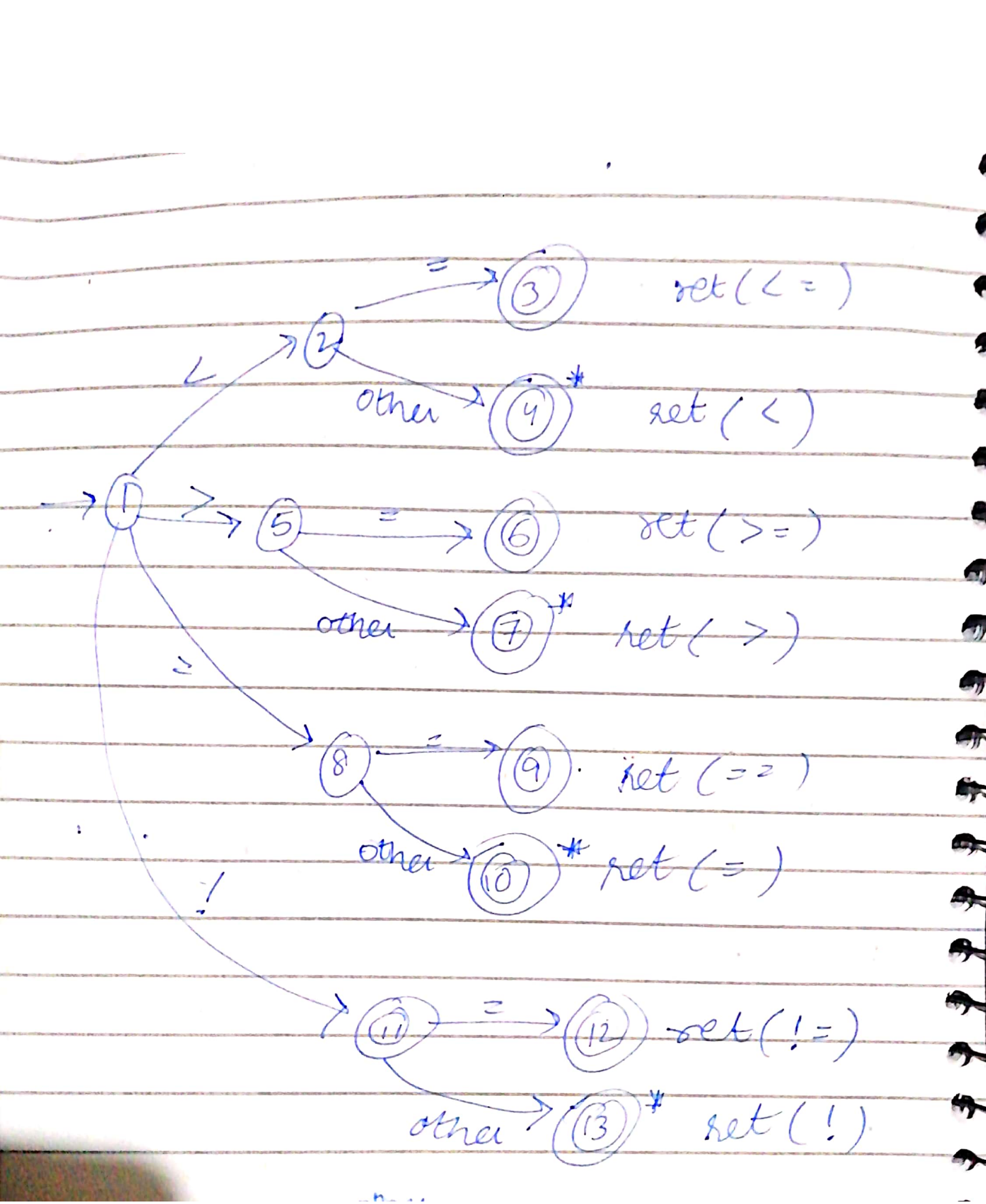
OF → λ | . digit+

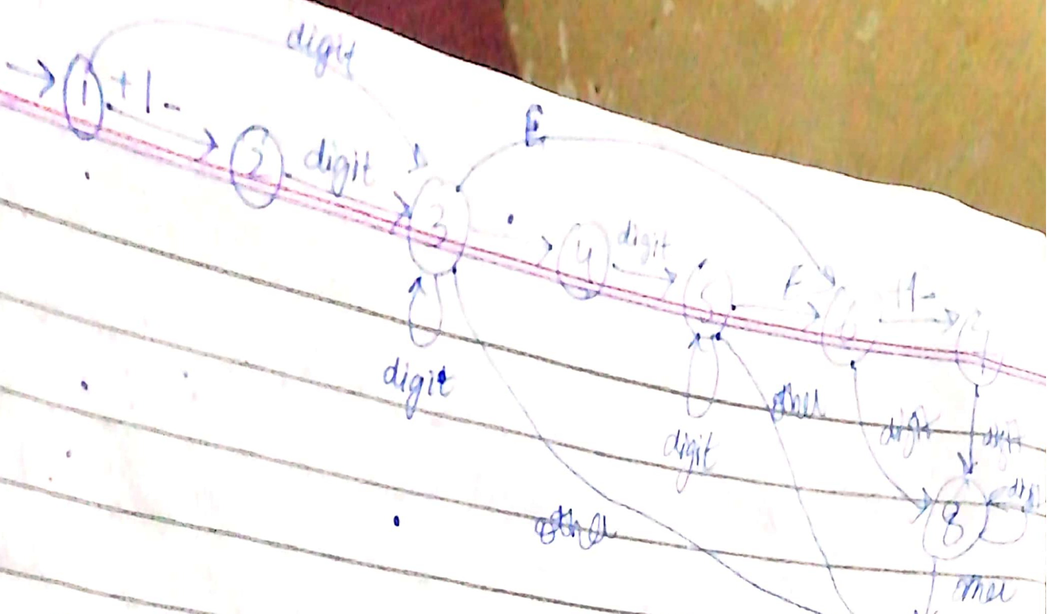
OE → λ | E OS digit+

DFA → Deterministic finite Automata









Tokens and Lexemes :-

→ A token is a meaningful sequence of characters
→ A lexeme is a particular instance of a token

```
int max (int x, int y){
    int m;
    if (x >= y)
        m=x
    else
        m=y
    return m;
}
```

('+', ∧)	(RO, GT)	(AND, ∧)
('-', ∧)	(RO, LT)	(OR, ∧)
('*', ∧)	(RO, LE)	('!', ∧)
('/', ∧)	(RO, EQ)	
('%', ∧)	(RO, NE)	
	(RO, GE)	

*) an identifier (ID) can have multiple lexemes
↳ (ID, "x"), (ID, "y"), (ID, "max"), (ID, "m")

*) some tokens have 1 lexeme
↳ (INT, ∧), (IF, ∧)

256

enum Names [INT, ID, RO, - - - - -];

```
* struct Pair {
      int tok;   // RO    // ID
      int len;   // GE    // (int)'s
      char* s; "max"
```

(INT, Λ), (ID, "sum"), ('(', Λ), (INT, Λ), (ID, "a"),
('[', Λ), (']', Λ), (',', Λ), (CONST, Λ), (INT, Λ),
(ID, "N"), (')', Λ), ('{', Λ), (INT, Λ), (ID, "s"), (';', Λ),
(NUM, 0), (FOR, Λ), ('(', Λ), (INT, Λ), (ID, "i"),
('=', Λ), (NUM, 0), (';', Λ), (ID, "i"), (RO, LT),
(ID, "N"), (';', Λ), (INC, Λ), (ID, "i"), (')', Λ),
(ID, "s"), ('+', Λ), (ID, "a"), ('[', Λ), (ID, "i"),
(']', Λ), (';', Λ), (RET, Λ), (ID, "s"), (';', Λ),
('}', Λ)

```
int sum (int a[], const int N){
  int s=0;
  for (int i=0; i<N; i++){
    s=s+a[i];
  return s;
}
```

$\Sigma = [a, b]$

(1) strings having exactly 2 a's

$S \rightarrow Ba\,Ba\,B$

$B \rightarrow Bb \mid \lambda$

(2) at least two a's

$S \rightarrow R \mid X\,X$

$[aa + bb + (ab + ba)(aa + bb)^*(ab + ba)]^*$

$S \rightarrow Saa \mid SbB \mid SABDAB \mid \lambda$

$AB \rightarrow Ab \mid ba$

$D \rightarrow Daa \mid DbB \mid \lambda$

Questions :-

1-. Which code must to generate for incorrect program

2-. SymbolTable would have unique ID's?

3-. Test.cmm ? what is this file

4-. Strings and comments to print in file?

5-. $x = x - 1 \Rightarrow$ Now should it be (+, ^) and (^, num, 1) or (−, ^) and (^, num, 1)

6. Incorrect Program partially give token-lexeme pa until its correct.

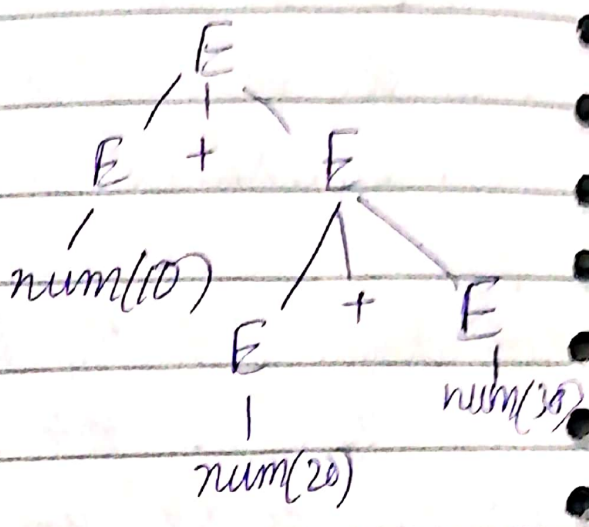
How to Develop Parser?

(1) Write CFG
(2) Rewrite CFG
(3) Implementation

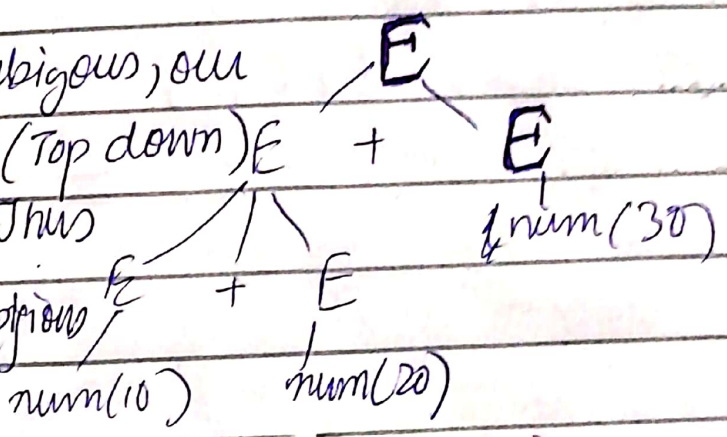
(a) Ambiguity
(b) Associativity
(c) Procedure
(d) Left recursion
(e) Left factoring

$E \to E+E \mid E-E \mid num$

$10+20+30$



if CFG is ambigous, our parsing algo (Top down) gets confused. Thus make unambigious



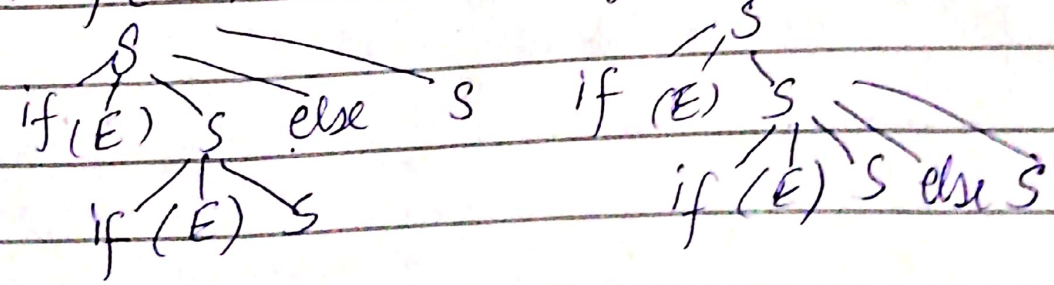
$S \to S(S)S \mid \wedge$ (Ambigous)

$S \to S(S) \mid \wedge$ (unambigous)

$S \to (S)S \mid \wedge$ (unambigous)

$S \to SS \mid (S) \mid \wedge$ (ambigous)

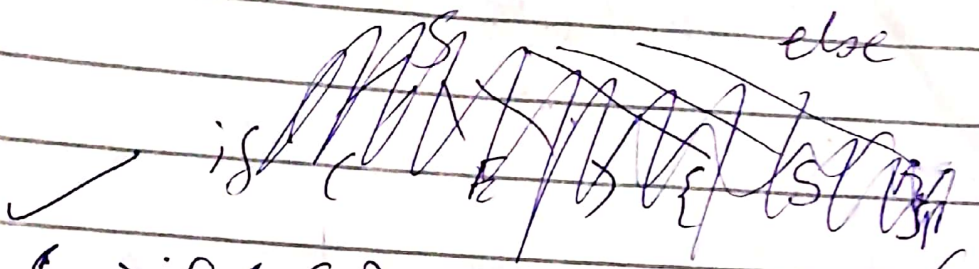
$S \to if(E)\ S\ else\ S \mid if(E)\ S$



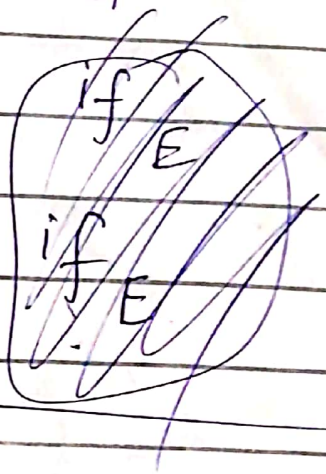
Dangling else problem

$S \rightarrow if(E)\{S\}else\{S\} \mid if(E)\{S\}$

$S \rightarrow if(E)\,S\,R$
$R \rightarrow \wedge \mid else\ S$



$S \rightarrow if(E)[S]\,R$
$R \rightarrow \wedge \mid else[S]$



Operator Associativity :-

$x = a + b + c$; ← left assoc (addition)

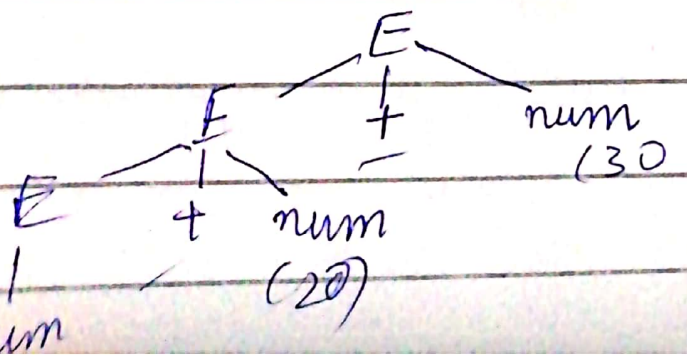
$u = v = w$; ← right assoc (assignment)

$y = e \wedge f \wedge u$ ← right assoc (exponent) e^{f^u}

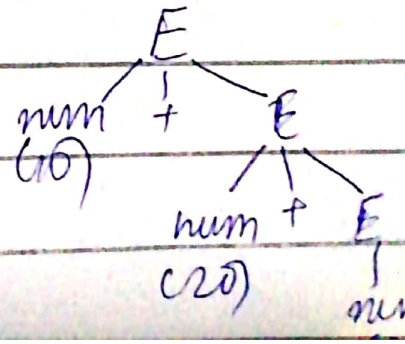
left recursive → associativity left

right recursive → associativity right

$E \rightarrow E + num \mid num$



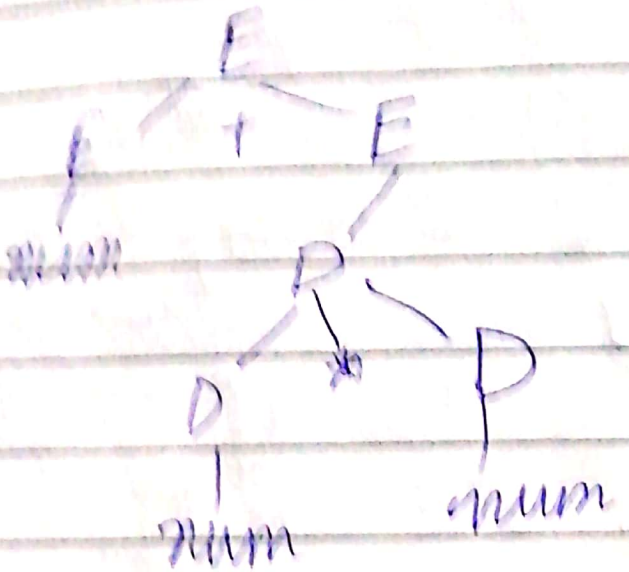
$E \rightarrow num + E \mid num$



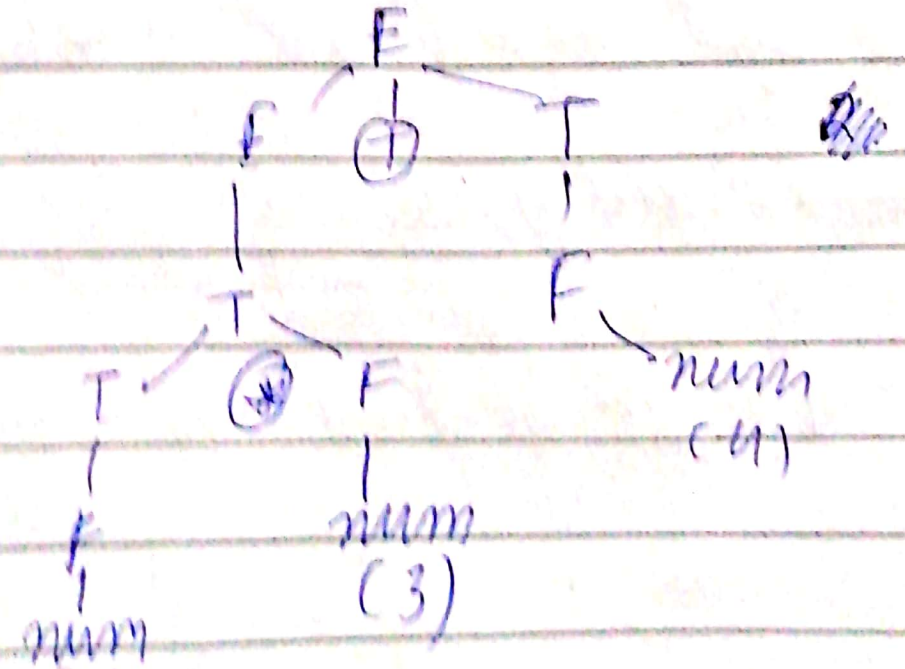
Ambiguous Grammar

$E \to E+E \mid E-E \mid E*E \mid E/E \mid num$

$E \to E+E \mid E-E \mid D \mid num$
$D \to D*D \mid D/D \mid num$



$E \to E+T \mid E-T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to num$



(Left associativity)

$E \rightarrow E + T \mid E - T \mid T$

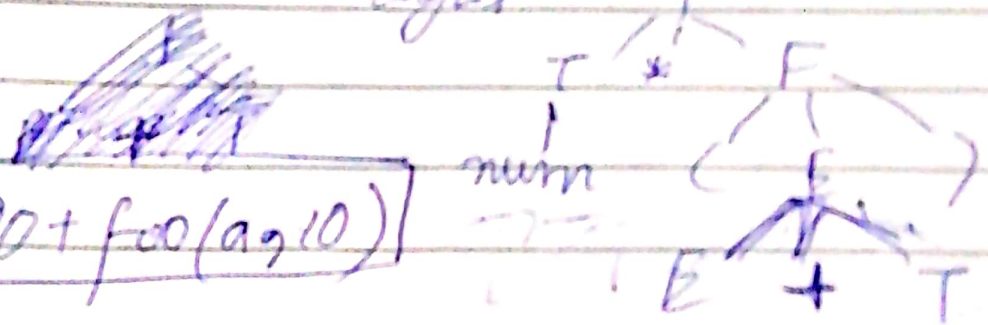
$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow num \mid (E) \mid id \mid id(AL)$

(a+3) * 4

a * (3+9)

args



x = 20 + foo(a, 10)

AL →

OAL → λ | AL

AL → AL, E | E