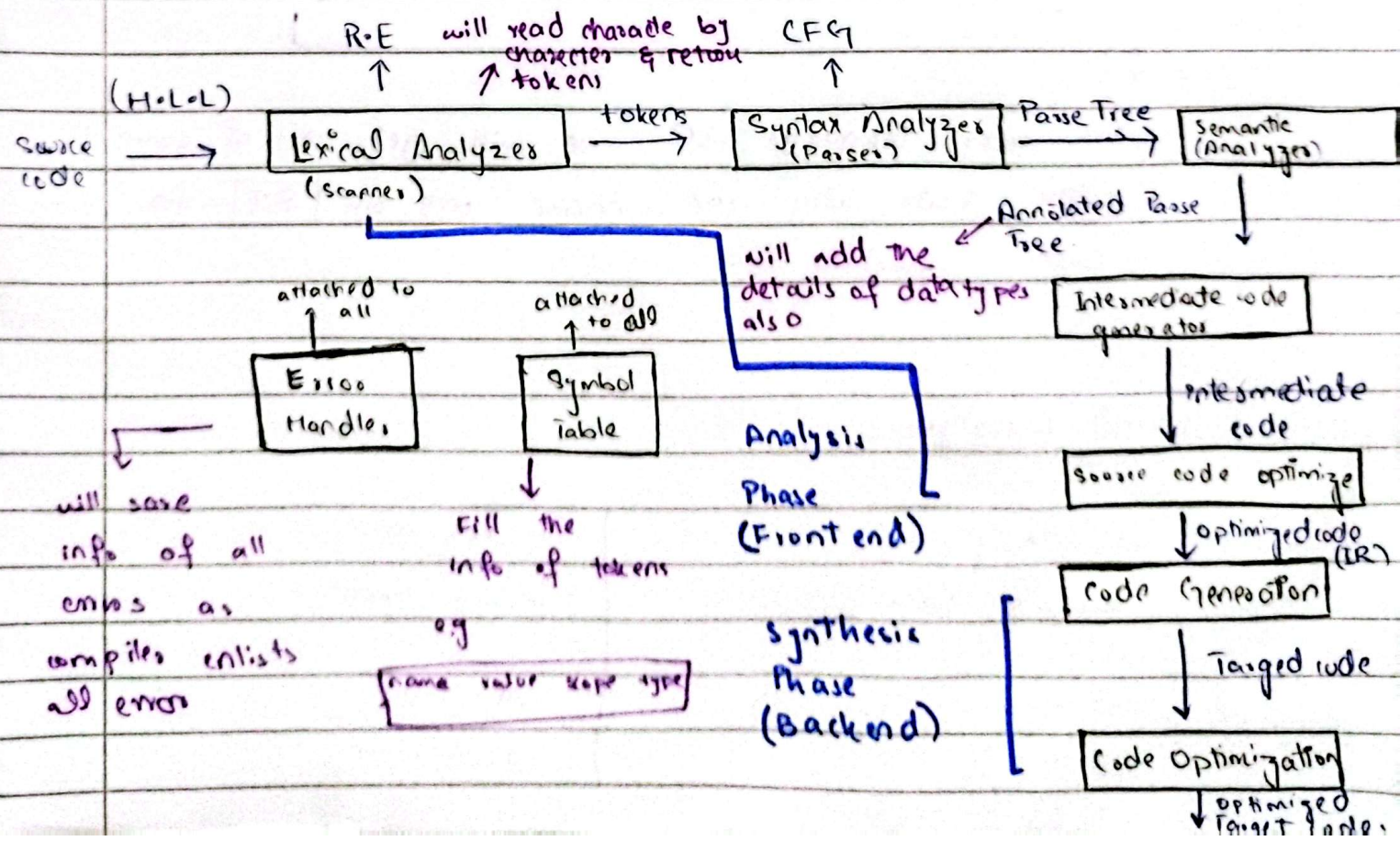
- Translator
  ↓
  Compiler
  - doesn't run other code (compile & run)
  - makes object code file & enlists the errors
    - ↳ usually assembly code
  - Faster
  ↓
  interpreter → easier debugging.

- Source Code → [Compiler] → Target Code (Assembly)
  ↓
  machine dependent

⇒ Phases in Compiler:



- Precedence
- Associativity

- two pass compiler

Source code → [front end | end] IR → [Backend] → target code

C++	[F.E]	[B.E]	intel
Java	[F.E]	[B.E]	Linux
C#	[F.E]	[B.E]	Mac
m components		n components	

m × n components → m × n compilers (Benefit of 2 pass compilation)

For every language IR we will get is of same syntax that's why we make only one [F.E] for each machine.

⇒ **Lexical Analyzer:**

int x, y = 10;

making tokens

int ⇒ key (keyword)
x ⇒ id
, ⇒ comma operator

int x, y = 10

making tokens

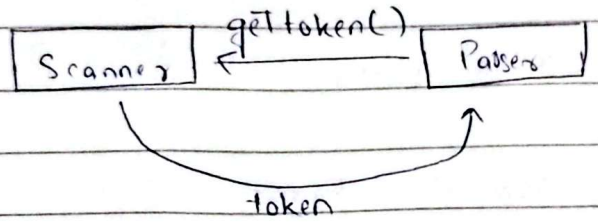
int x ⇒ id
, ⇒ comma

y => id	y => id
= => assign op	= => assin
10 => int literal	10 => num
; => semicolon	; => semi

- int $ y ; → lexical Error
- 59 = x → syntax Error
- int x, y = 59; → semantic Error

1) Lexical Analyzer :-

1 → Tokenization
2 → Removal of white spaces
3 → Lexical error handling
4 → Removing comments



set of character
↑

- Lexeme if it matches with a pattern = R.E then it will form Token otherwise lexical error.

→ int literal → $(0-9)^+$ or digit + or digit* − ε

→ for identifier ⟹ letter (Letter + digit)*

- (letter + digit)* letter (letter + digit)* letter (letter + digit)* letter (letter + digit)*

a(a+b)

3.2 e 10

- 3.2 , .2 , 3.

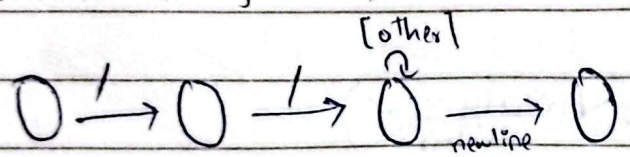
$(digit)^* \cdot digit^+ + (digit)^+ \cdot (digit)^*$

3.2 e 10
3.2 E+10
3.2 E-10
3.2 e+10

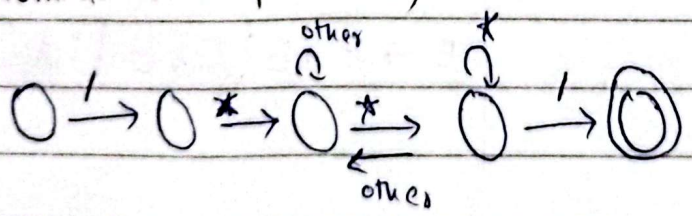
- $((e + E)("+" + "-" + "\lambda")\ digit^+ + \lambda)$

character & string literal → ?

- comment (single line)



- comment (multiple line)



- meta symbols

  - "+" + -

- digit → [0 - 9]
  num → (digit* . digit+) + (digit+ . digit*)
  signed_num → (+ / -) ? num

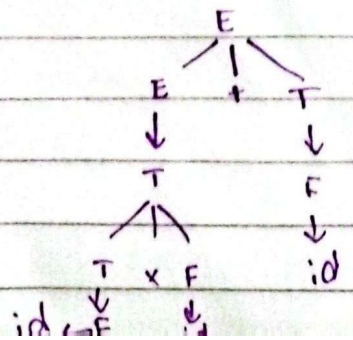
- Lookaheads → read the next characters

Parser :- (Syntax Analyzer)

Terminal = Tokens

A- E → E+T | T
T → T*F | F
F → id

id * id + id



E → E+E | E*E | id

- ambiguity
- due to precedence is disturbed.

- Jitne precedence high ho grammar me uska rule utna niche hoga.

Relational Expression: (== , <= , < , > , >= , !=)

R-E → A.E r-op A.E
r-op → == | <= | < | > | >= | !=

Logical Operator: (&& , || , !)

L.E → R.E L.op R.E | U.L.op R.E
L.op → && | ||
U.L.op → !

Assignment Operator:

These all are L to right associative

Assignment operators are right to left.

A.E
[redacted] → id assignment-op exp; |
id assignment-op A.E

A-O → += , = , -= , *=

- **IF**

IF → if (COND) STMT | if (COND) STMT else STMT

COND → 0 | 1

Back Tracking issue

IF → if (COND) STMT ELSE_PART

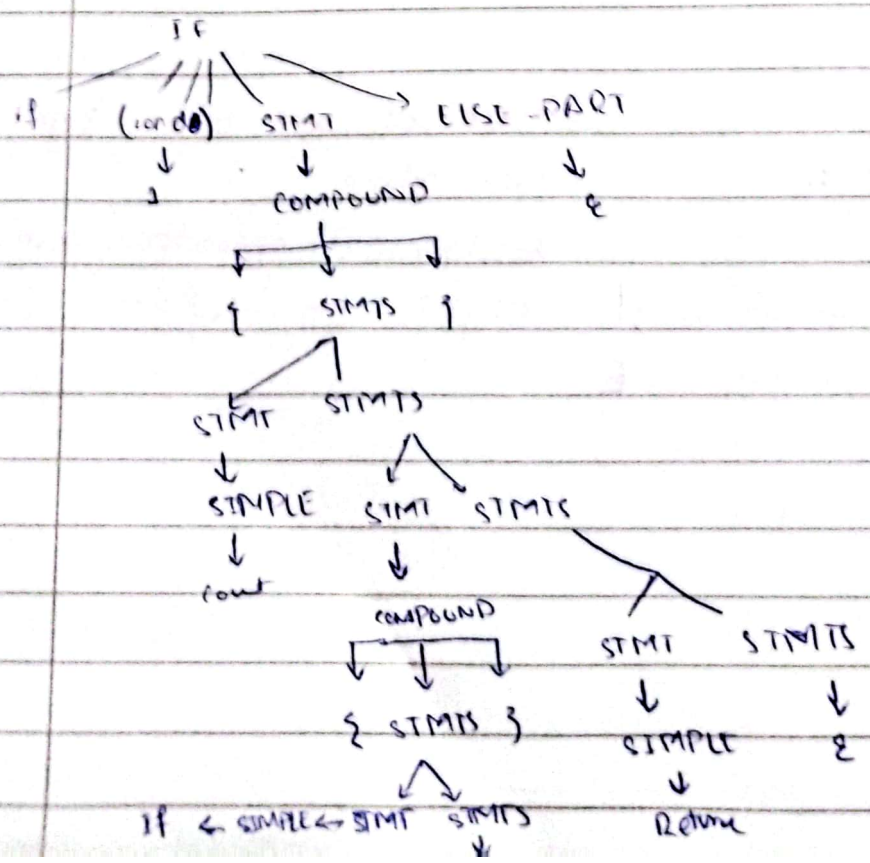
ELSE_PART → ∂ | else STMT

STMT → SIMPLE | COMPOUND

SIMPLE → WHILE | FOR | ASSIGN | IF | FUNCTION_CALL - - - - - - -

COMPOUND → { STMTS }

STMTS → ε | STMT STMTS



```
{(n)
{ return
  ?:{ }
  return;
}
```

- WHILE → while (cond) STMT
- DO_WHILE → do COMPOUND while (cond);
- FOR → for (stmt; cond; exp)
  ↓
  will have to define
  STMT_SEQ
- FUNC_CALL → id (P_LIST)
  P_LIST → ε | PARAMS
  PARAMS → PARAMS , EXP | EXP
- FUNC_DEF → FN_HDR FNBODY
  FNBODY → COMPOUND
  FN_HDR → RET_TYPE id (PARAM_LIST)
  RET_TYPE → void | int | float .....
  PARAM_LIST → ~~DECLARE~~ PARAMS9 | ε
  D_TYPE → int | float | double | id ...
  PARAMS9 → D_TYPE id, PARAMS9 | D_TYPE id

  Header → rettype id (PARAMS)
  {
  Body →
  }

→ learn how to add default parameters
→ Note default parameters are on right

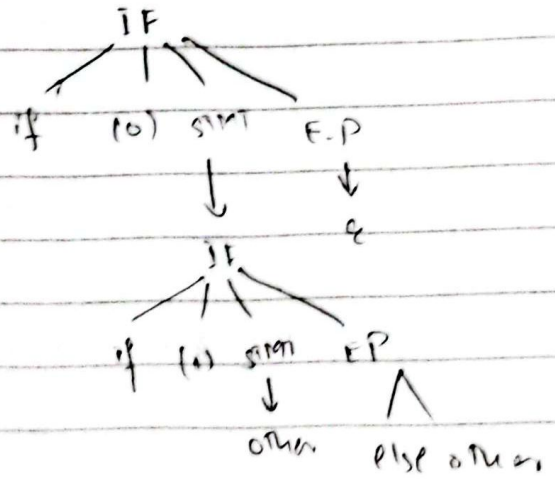
void display (int a, int b, int c=0, int d=5)

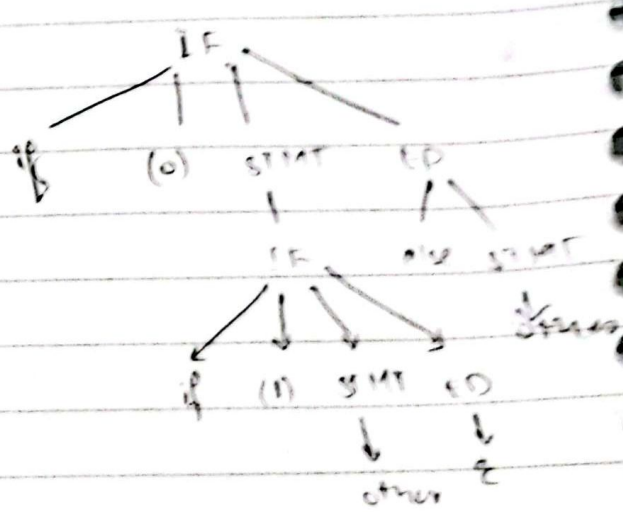
- **Dangling else problem:-**

IF → if (cond) STMT ELSE_PART
ELSE_PART → ε | else STMT
STMT → other | IF
COND → 0|1

$\boxed{\text{if (0) if (1) other else other}}$

ambiguous





- In python indentation helps. C++ resolved this by assigning else to nearest if.

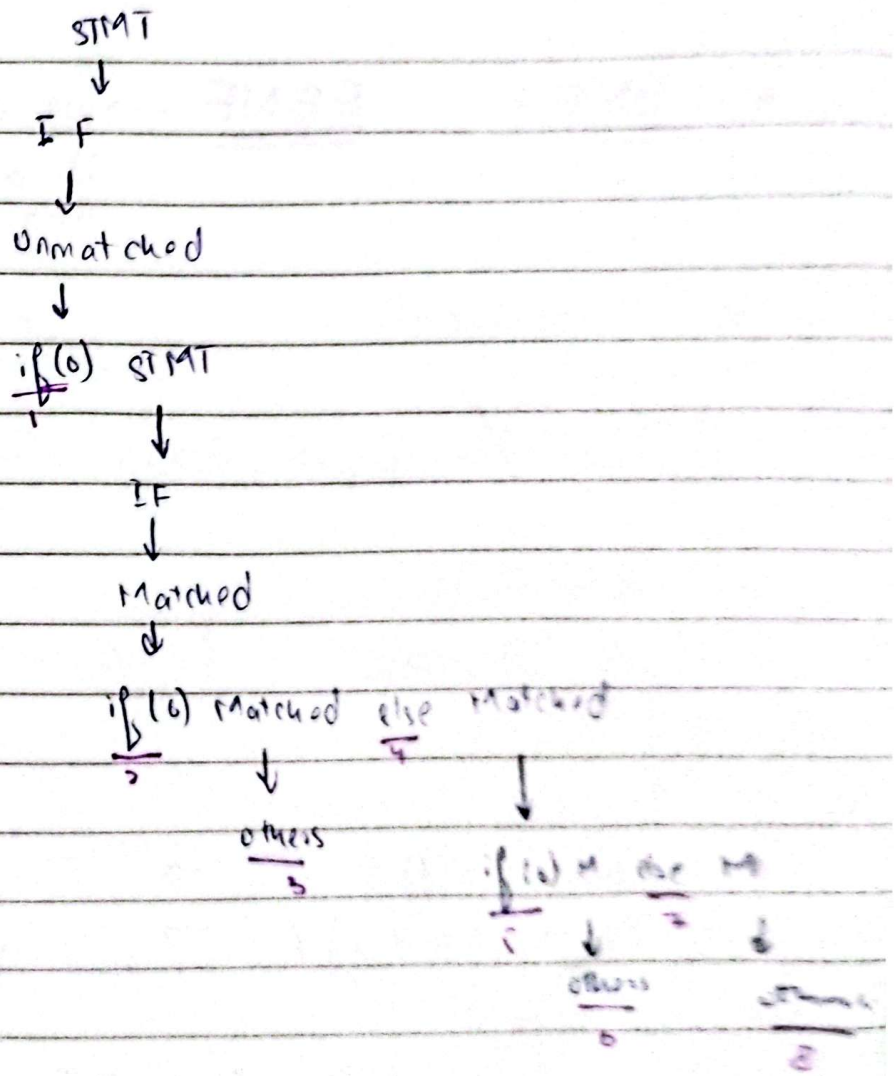
STMT → IF | others

IF → Matched | Unmatched

Matched → if (c) Matched else Matched | others

Unmatched → if (c) STMT | if (c) Matched else Unmatched

if (c) if (c) others else if (c) others else others

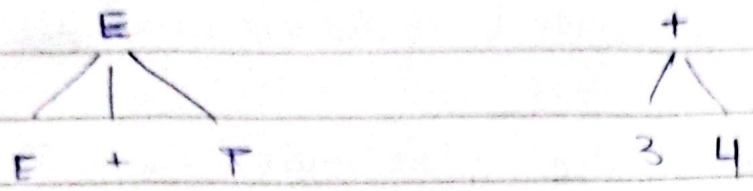


Papers be like

⇒ else should be attached with fastest if

⇒ .

- **Syntax Trees :-**



- **BNF :-** **EBNF** → we remove left recursion
  it also has some issues that why not used widely.

* $A \rightarrow A\alpha \mid B$

  $B\alpha^*$

  $\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha$

* $A \rightarrow \alpha A \mid B$

  $\beta, \alpha\beta, \alpha\alpha\beta$ $\qquad \alpha^*\beta$

- $E \rightarrow E+T \mid T \quad \rightarrow \quad E \rightarrow T\{+T\}$

  $X \rightarrow Xa\,Yb \mid Y \quad \rightarrow \quad X \rightarrow Y\{aYb\}$

2 new things in EBNF

$\{\ldots\}$: Loop

$[\ldots]$: conditional

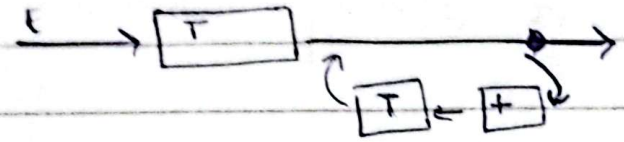
IF → if (COND) STMT | if (COND) STMT else STMT

→ conditional may or may not be

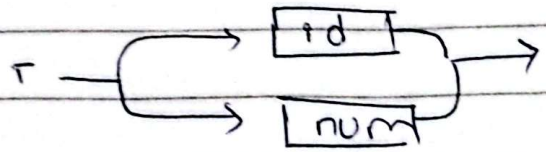
if (COND) STMT [else STMT]

- **Syntax Diagram:-**

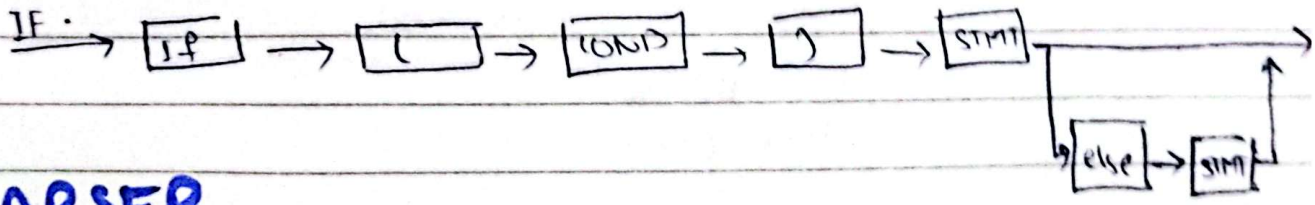
- L → T {+T}



- T → id | num



- if (cond) STMT [else STMT]



PARSER:-

- Top Down
  - Recursive Descent
  - Predicted Parser
    Table Driven
    LL (1) parser
- Bottom UP
  - Precedence Parser
    - Operator Precedence
  - LR Parser
    - SLR(1)
    - LALR (1)

Right most derivation

id + id

E → E + T

E + id

T + id

id + id

In bottom up we will go right most in reverse order.

- Left recursion & left factoring → missed 1 week

- **Recursive Decent Parsing**

∵ We match the terminals & call the non terminals

E → E+T | T

T → T * F | F

F → (E) | number

Make function for each nonterminal

```
F () {
    if (token == '(' )
        {
            Match ('(')
            E ();
            Match (')')
        }
```

→ if not we display syntax error & move to next line

```
E ()
{
E()
Match(+)
T()
```

∵ Now we will get stuck in infinite recursion so we remove left recursion

```
main()
{
    token = get_token();
}
```

- Removing left recursion

$E \to TE'$

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

```
E() {
    T();
    E'();
}
```

```
E'() {
    if (token == '+')
    {
        match('+');
        T();
        E'()
    }
    else
    return;
}
```

```
E()
{
    T()
    while (token == '+')
        match('+')
        T();
    endwhile
}
```

- $T \to FT'$

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

```
T() {
    if (token == '(')
    {
        match('(');
        E();
        match(')');
```

not possible due to back tracking

- IF → if (COND) STMT | if (COND) STMT else STMT
  IF (COND) STMT [Else STMT]

our option
i) left factor
ii) EBNF

E → T { + T } EBNF

```
E()
{ T()
  while (token == '+')
      Match('+')
      T();
  end while
}
```

- <u>Issues</u>:-