05.03.21

Introduction

A compiler is a program that can read a program in one language - the source language and translate it into an equivalent program in another language - the target language.

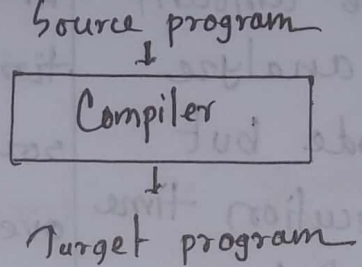


Fig: Compiler

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

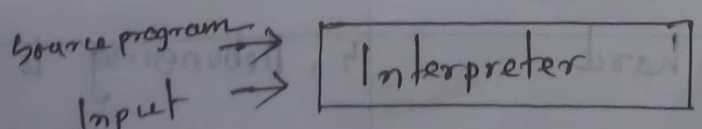


Fig: Interpreter

02

Compiler vs Interpreter

Compiler	Interpreter
1. Scans the whole program and translates it as a whole into machine code.	1. Translates one statement at a time.
2. It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.	2. It takes less amount time to analyze the source code but the overall execution time is slower.
3. Generates intermediate object code ~~with~~ which further requires linking, hence requires more memory.	3. No intermediate object code is generated, hence memories are efficient.
4. It generates the error message only after scanning the whole program.	4. Continuously translate the program until first error is found.
5. Debugging is hard	5. Debugging is easy
6. C, C++	6. Python, Ruby, MatLab

03

7. High speed	7. Low speed
8. Compilers are larger in size	8. Smaller in size
9. Error detection is difficult	9. Error detection is easier

#Types of a compiler

① Single pass compiler
② Multipass compiler

⇒ Single pass compiler : Single pass compiler that processes the source code only once.

⇒ Multipass compiler : It is a type of compiler that processes the source code multiple times to convert the source code to multiple code.

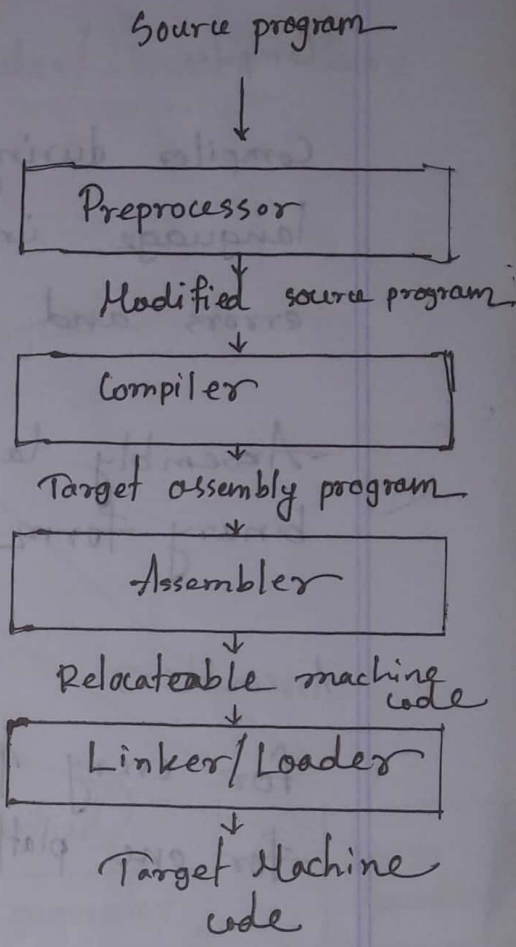
Two major parts of a compiler

- Analysis phase :- It breaks up the source code into small parts and creates an intermediate representation of the soure phase.

Synthesis phase :- Output of the ~~int~~ analysis part acts as input here, and generates the target output.

05

Language processing system



High level language programs are fed into a series of tools and os components to get the desired code that can be used ~~by the~~ by the machine. This is called language processing system.

Preprocessor

Source program inserted into preprocessor. Preprocessor modifies the source program. It removes all the preprocessor directories, and add the respective files.

Example: #include<stdio.h> ignore ~~those~~ the header file.

It will do macro expansion, operator conversion.

a--; ⇒ a = a-1;

Preprocessor

Compiler

Compiler during compilation converts pure high level language into assembly language. It also gives errors and warnings.

Assembly language is a low-level. It is not in a binary form.

Assemblers

for every platform we have a assembler. A assembler for one platform will not work for another platform.

Assembler code $\xrightarrow{\text{converted}}$ Executable machine code

Code can be loaded anywhere in the memory.

Loader & Linker

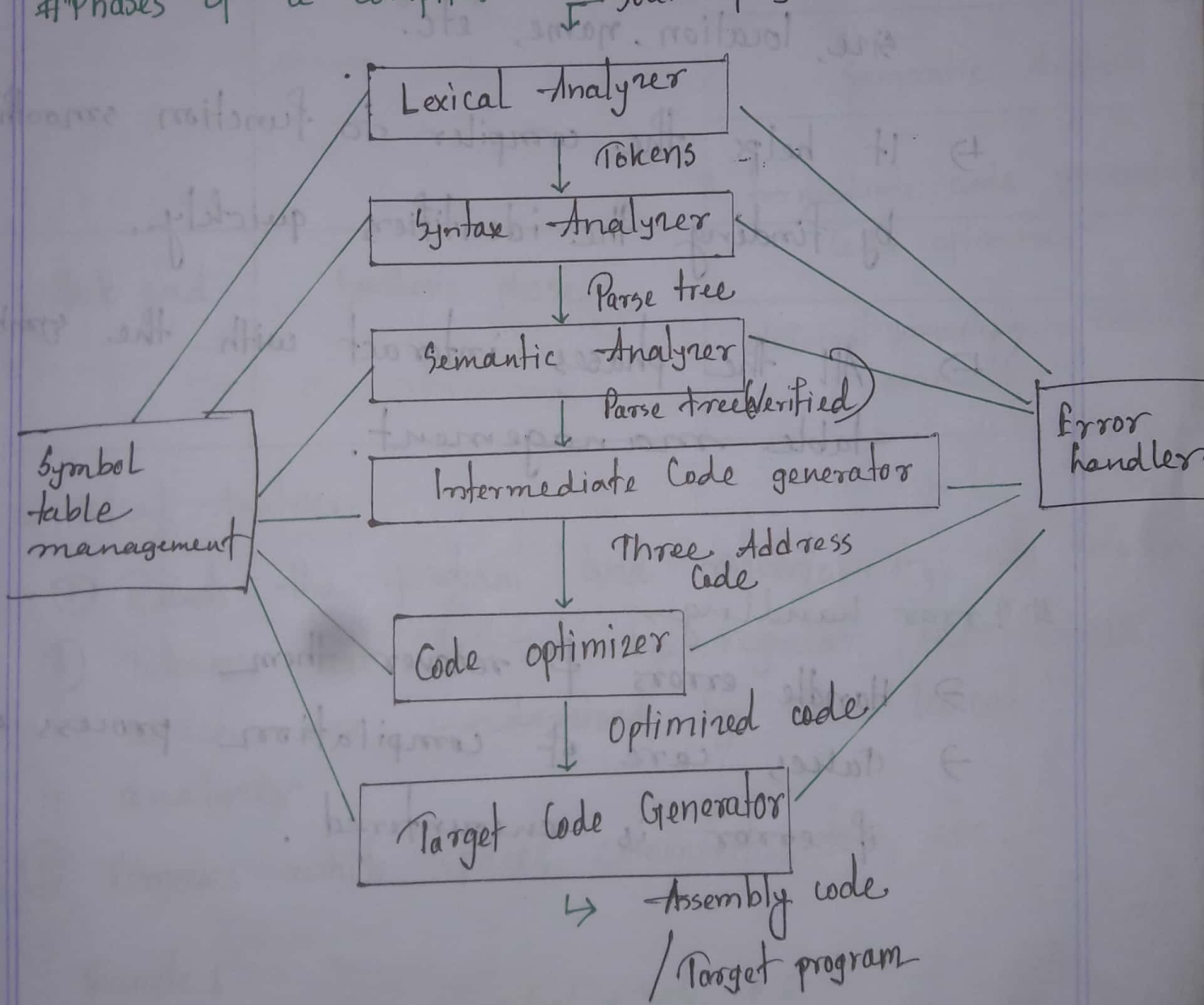
A linker links different object files into a single file/executable file. and

Loader loads that executable file in the memory and executes it.

07

Preprocessor, assembler and loader/linker are cousins of a compiler.

#Phases of a compiler



#Symbol table management

↳ A data structure which is used by the compiler to store all the information related to identifiers. Example: types, scope, size, location, name, etc.

↳ It helps the compiler to function smoothly by finding the identifiers quickly.

↳ All the phases interact with the symbol table management.

Error handling

⇒ Handle errors & recover them

→ Takes care of compilation process even if error is encountered.

09

Part	Major phases	Phases
front end	Analysis phase	Lexical Analysis
		Syntax Analysis
		Semantic Analysis
		Intermediate code generator
Back end	Synthesis phase	Code optimizer
		Target ~~machine~~ code generator

Lexical Analyzer

① Reads the program and converts it into tokens
② Tokens are defined by regular expressions which are understood by the lexical analyzer.
③ Removes white spaces, comments, tabs, etc.

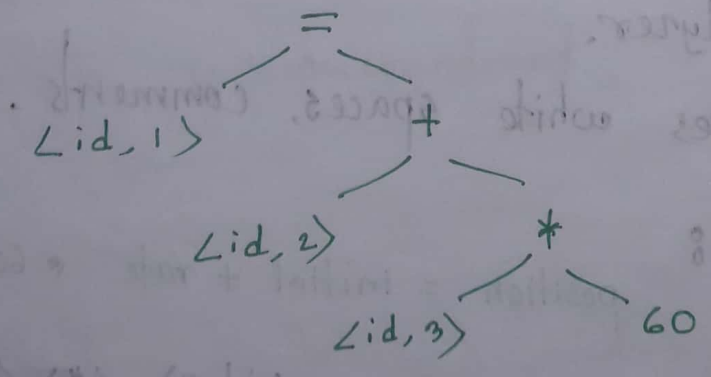
Example :

position = initial + rate * 60

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

Syntax Analyzer

1. Constructs the parse tree.
2. Takes the tokens one by one, and uses CFG to construct the parse tree.
3. Using the productions of CFG we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
4. Syntax errors can be detected by it if the input is not accurate to the grammar given.
5. So, it takes input as tokens and checks their syntactic correctness.

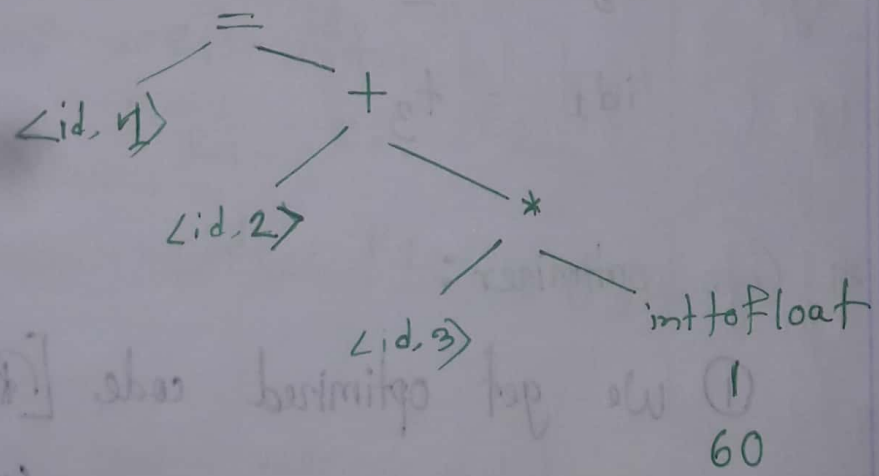


Semantic Analyzer

① Verifies ~~the meaningfull~~ whether ~~pa~~ the parse tree is meaningful or not.

② It uses the parse tree and info in the symbol table to checks the source program for semantic consistency with the language definition.

Type checking, definition of the variables used, label checking, flow control, etc.



Intermediate Code Generator

① Generates the intermediate code.
⇒ 3 address code

② This code converts into machine language using last two phases of the platform.

$t_1 = \text{inttofloat}(60)$

$t_2 = id_3 * t_1$

$t_3 = id_2 + t_2$

$id_1 = t_3$

Code optiminer:

① We get optimized code [Removing unnecessary code]

② Code optimization phase attempts to improve the intermediate code so that it runs faster and works efficiently.

$$t_1 = id_3 * 60.0$$
$$id_1 = id_2 + t_1$$

Target Code Generator

① Final phase of the compiler which generates the target code.

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

Grammar and Context Free Grammar

In normal human languages, a grammar is a set of rules that used for proper conversation with each other. So, in the same way for writing computer languages there is a mathematical model of grammar which is used to write the languages in the correct way.

Noam Chomsky gave a mathematical model of grammar for writing computer languages effectively.

Type	Grammar Accepted	Languages Accepted	Automation
Type-0	Unrestricted grammar	Recursively Enumerable language	Turing machine
Type-1	Context sensitive grammar	Context sensitive language	Linear bounded Automation
Type-2	Context free grammar	Context free languages	Pushdown automata
Type-3	Regular grammar	Regular language	finite state Automation

15 Regular grammar

A grammar 'G' can be formally described using 4 tuples as $G = (V, T, S, P)$ where,

V = Set of variables or Non terminal symbols

T = Set of terminals

S = Start symbol

P = Production rule

A production rule has the form $\alpha \to \beta$ where α and β are strings on $V \cup T$ and atleast one symbol of α belongs to V

Example : $G = (\{S, A, B\}, \{a, b\}, S, \{S \to AB, A \to a, B \to b\})$

Here

$V = S, A, B$

$T = a, b$

$S = S$

$P = S \to AB, A \to a, B \to b$

$S \to AB$

$S \Rightarrow aB$

$S \Rightarrow \underline{ab}$

Types of a regular grammar :

Regular grammars are two types as follows :

1. Right linear grammar
2. Left linear grammar

Right linear grammar : A grammar is said to be right linear if all the non-terminals are in the right side.

$$\left.\begin{array}{l} S \to xB \\ S \to x \end{array}\right\} \quad \begin{array}{l} V \to T^*V \text{ or} \\ V \to T^* \end{array}$$

Left linear grammar : A grammar is said to be left linear if all the non-terminals are in the left side.

$$\left.\begin{array}{l} S \to Bx \\ S \to x \end{array}\right\} \quad \begin{array}{l} V \to VT^* \text{ or} \\ V \to T^* \end{array}$$

17

A grammar is regular, if it is either ~~regular~~ right linear or left linear. This means, all productions in the grammar have to be completely left linear or completely right linear but not mixed left-linear and right linear.

Linear grammar : Grammars, in which each rule is in right linear or left linear form, i.e right linear rules and left linear rules can be mixed, is called linear.

$$S \rightarrow aX$$
$$X \rightarrow Sb$$
$$S \Rightarrow \epsilon$$

It is a linear grammar but neither left linear nor right linear.

Context Free Grammar

A context free grammar (CFG) consisting of a finite set of grammar rules where it has 4 tuples.

$G = (V, T, P, S)$

V = Non terminals / Variables

T = Terminals

P = Production rules

S = Starting ~~state~~ symbol

Production rule P will have the relation, $A \rightarrow a$, where $a = (V \cup T)^*$ and $A \in V$

That means, only nonterminals are allowed on the left side.

19

$G_1 = \{ (S, A), (a, b), S, (S \Rightarrow aAb, A \Rightarrow aAb | \epsilon) \}$

$V = (S, A)$

$T = (a, b)$

$S = S.$

$P = S \Rightarrow aAb, A \Rightarrow aAb | \epsilon$

Context free grammar is used for syntax analysis.

Derivation and Parse tree

#Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

⇒ Deciding the non-terminal which is to replaced.

⇒ Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

⇒ Left most derivation

⇒ Right most derivation

21

⇒ Left most derivation :- If the sentential form of an input is scanned from left to right, it is called left most derivation.

⇒ Right most derivation : If we scan and replace the input with production rules, from right to left, it is known as right most derivation.

Example : Production rules :

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

String : id + id * id

Left most	Right most
$E \Rightarrow E * E$	$E \Rightarrow E + E$
$E \Rightarrow E + E * E$	$E \Rightarrow E + E * E$
~~$E \Rightarrow id + id * id$~~	$E \Rightarrow E + E * id$
$E \Rightarrow id + E * E$	$E \Rightarrow E + id * id$
$E \Rightarrow id + id * E$	$E \Rightarrow id + id * id$
$E \Rightarrow id + id * id$	

22

#Parse tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

Grammar :

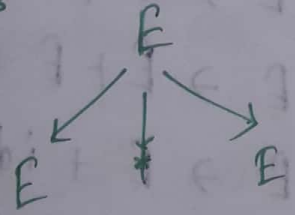
$E \rightarrow E + E$

$E \rightarrow E * E$

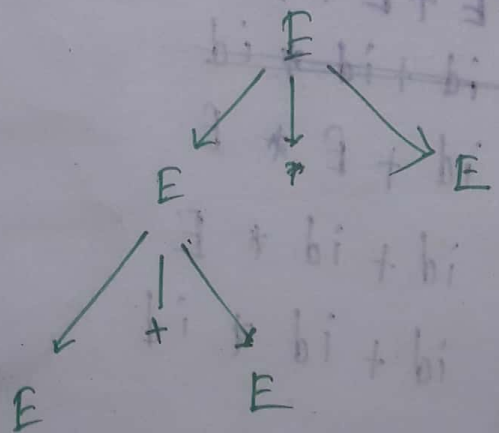
$E \rightarrow id$

String : id + id * id

Step 1 :

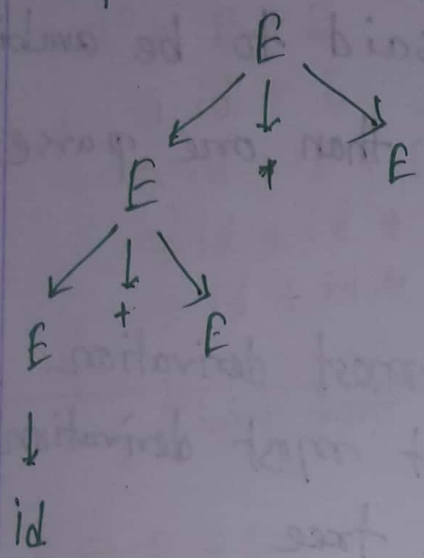


Step 2 :

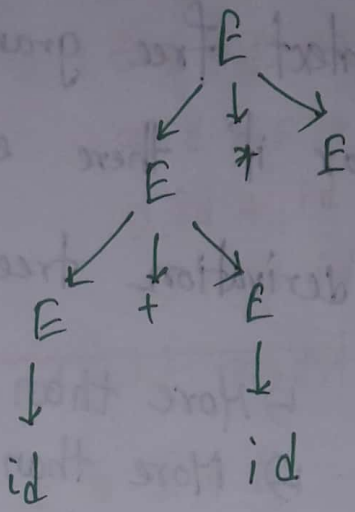


23

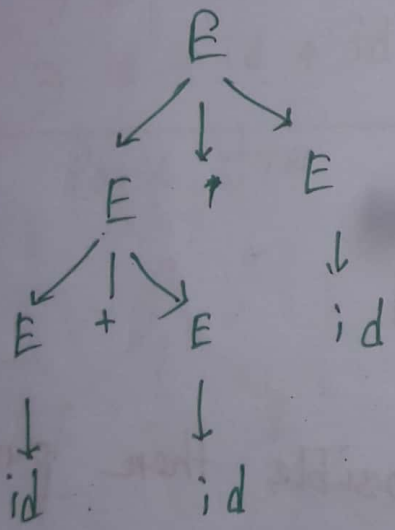
Step 3:



Step 4:



Step 5:



Properties of a parse tree

1. All leaf nodes are terminal.
2. All interior nodes are non-terminals
3. Start from the root node, which is the starting symbol.

Ambiguous Grammar

A context free grammar is said to be ambiguous grammar if there exists more than one parse tree / derivation tree.

- ↳ More than one left most derivation
- ↳ More than one right most derivation
- ↳ More than one parse tree

Grammar: $E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

String: id + id * id

If we have two parse trees possible then parser will get confused about which one to generate.

25

Left most derivation

$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

Left most derivation

$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$

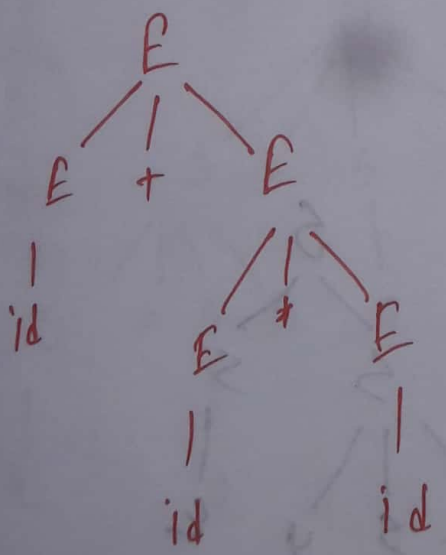
Right most derivation

$E \Rightarrow E + E$
$\Rightarrow E + E * E$
$\Rightarrow E + E * id$
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

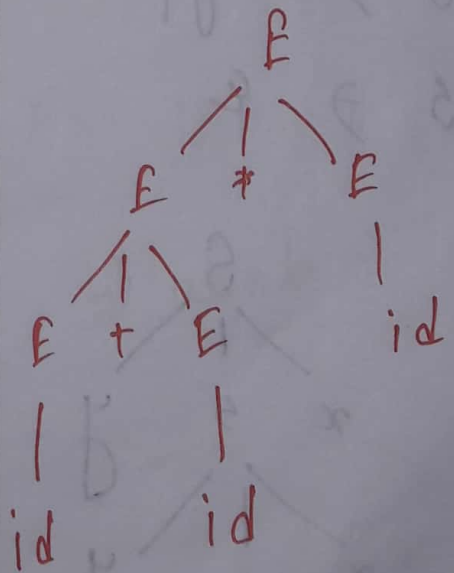
Right most derivation

$E \Rightarrow E * E$
$\Rightarrow E * id$
$\Rightarrow E + E * id$
$\Rightarrow E + id * id$
$\Rightarrow id + id * id$

Parse tree

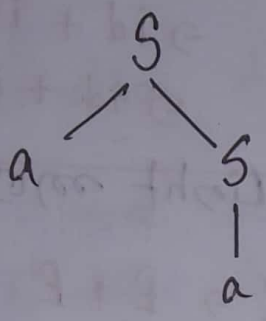


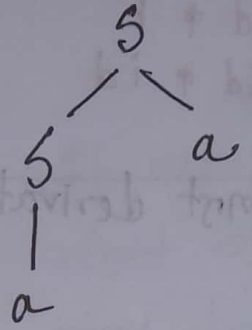
Parse tree



Check the grammar is ambiguous or not

(i) $S \to aS \mid Sa \mid a$ string $\Rightarrow aa$

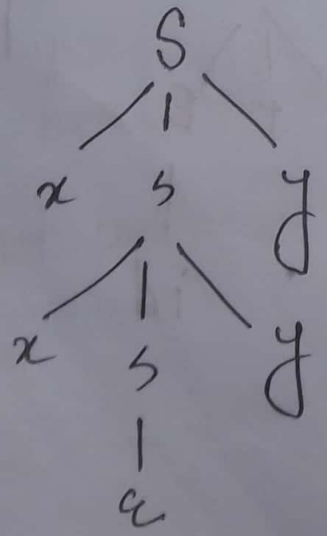


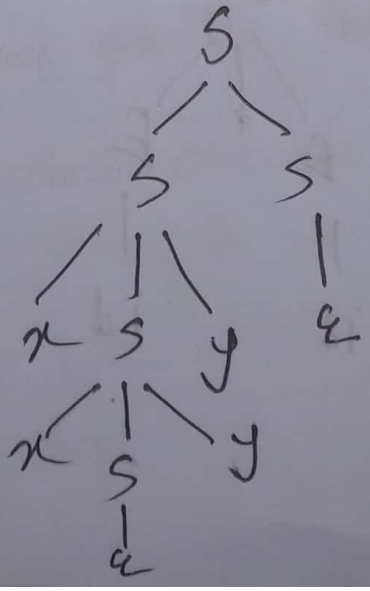


We have got two parse tree. So, this particular grammar is ambiguous.

(ii) $S \to xSy \mid SS$

$S \Rightarrow \varepsilon$ string $\Rightarrow xxyy$

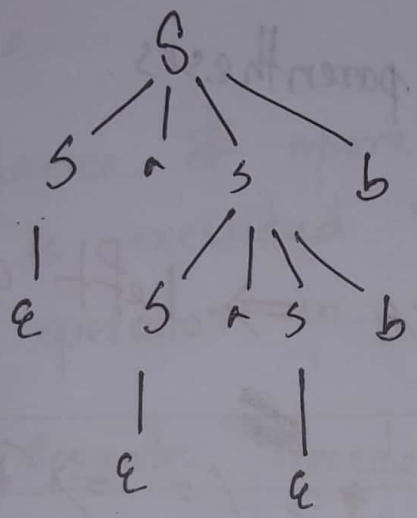




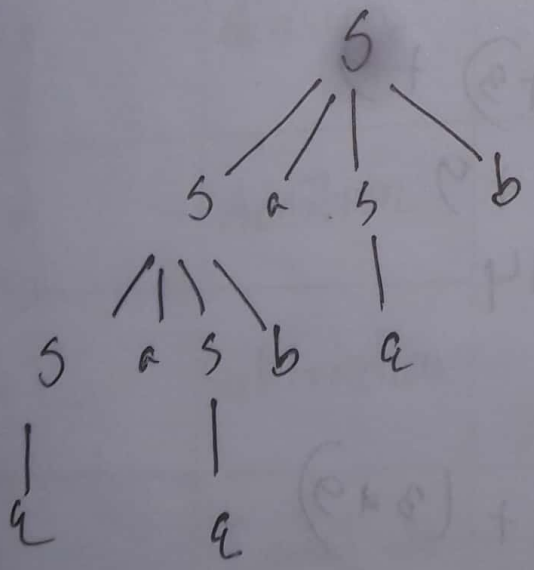
27.

Ambiguous grammar.

(iii) $S \rightarrow SaSb \mid \varepsilon$



$\Rightarrow aabb$



$\Rightarrow abab$

Unambiguous grammar

Associativity

The associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses.

+, -, *, /, % ⇒ Left associative

=, !, +=, -=, *=, /= ⇒ Right associa

2+3+9 = (2+3)+9
= 5+9
= 14

2+3*9 = 2+(3*9)
= 2+27
= 29

Here, + and * both ~~have the left~~ are left associative. So, we will consider the precedence of operator to decide which operator will tode progress first.

#Precedence

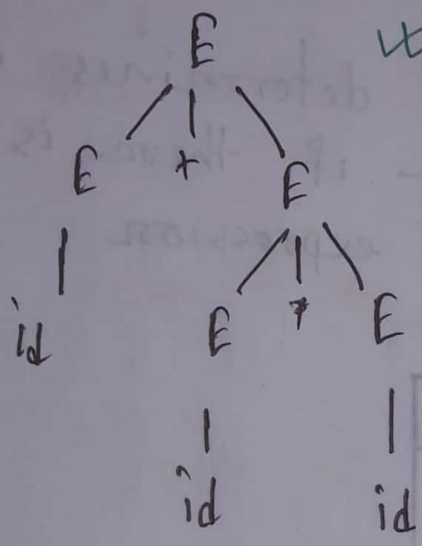
The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

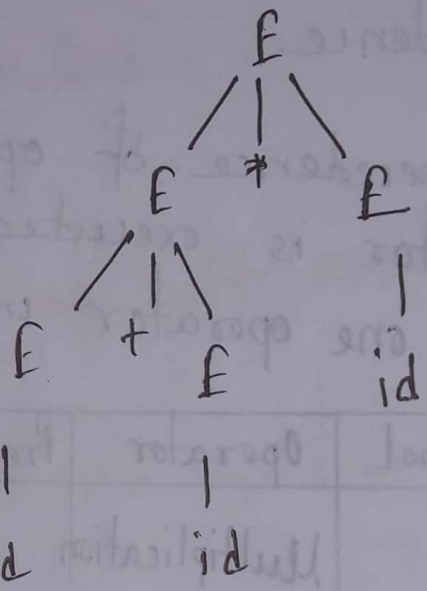
Symbol	Operator	Precedence
*	Multiplication	1
/	division	2
+	Addition	3
-	Subtraction	4
=	Equal to	5

Ambiguous grammar to unambiguous grammar.

$E \Rightarrow E + E \mid E * E \mid id$ (id + id * id)

This is an ambiguous grammar. because





id + (id * id)

1 + (3 * 4)

= 1 + 12 = 13

(id + id) * id

(1 + 3) * 4

= 4 * 4

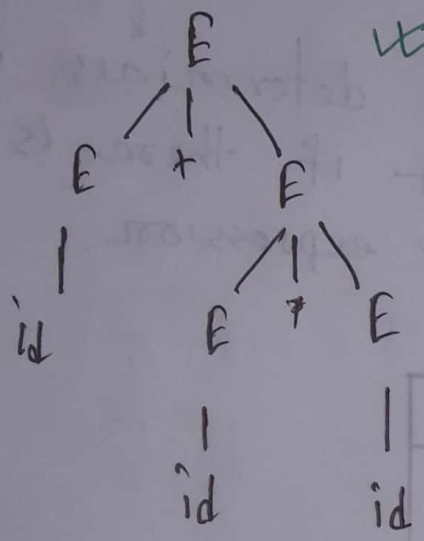
= 16

Compiler becomes confused when it interac with ambiguous grammar. So, it is necessary

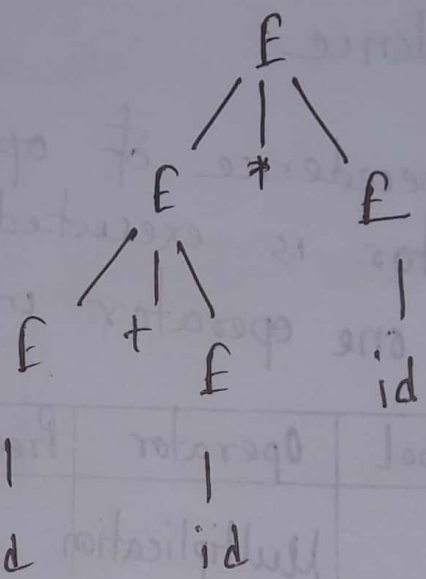
Ambiguous grammar to unambiguous grammar.

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

This is an ambiguous grammar because



✓



✗

id + (id * id)

1 + (3 * 4)

= 1 + 12 = 13

(id + id) * id

(1 + 3) * 4

= 4 * 4

= 16

Compiler becomes confused when it interacts with ambiguous grammar. So, it is necessary

for us to improve the grammar and remove ambiguity.

Depending on two criterias we convert an ambiguous grammar into an unambiguous grammar.

- Associativity
- Precedence

We will introduce recursion for solving associativity problem. Left recursion for left associative and right recursion for right associative.

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

Left Recursion

If the variable (RHS) is same as the leftmost variable then we say that particular grammar has left recursion.

$A \rightarrow A\alpha | \beta$

Two types:

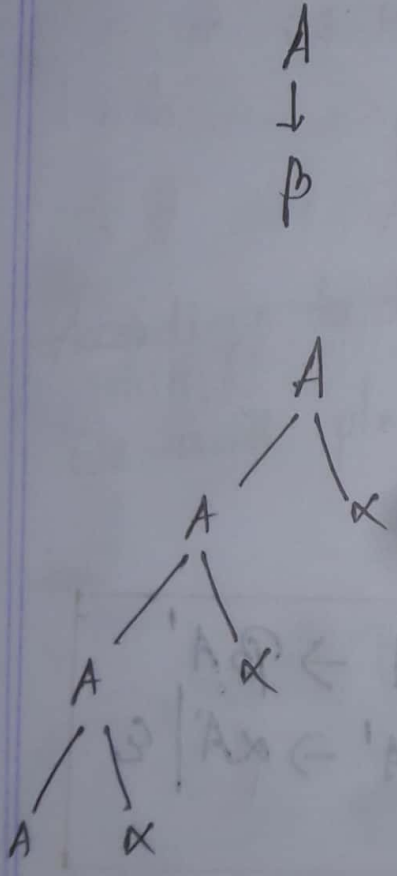
1. Direct left recursion $[A \rightarrow Ai]$
2. Indirect left recursion $[X \rightarrow Ab \quad A \rightarrow Xt]$

Top-down parser can not accept left recursion that's why we have to remove it. But we have to keep in mind that after removing the left recursion the grammar should be the one as it was before.

Recursion two types :

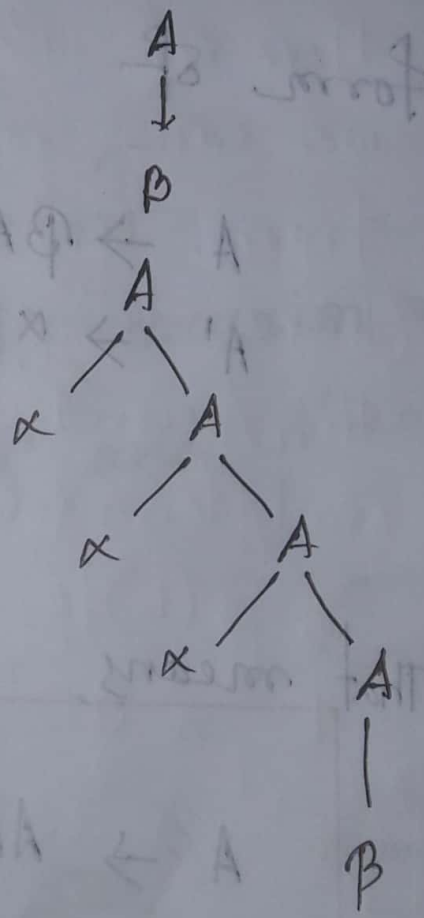
1. Left recursion
2. Right recursion

Left recursion	Right recursion
$A \rightarrow A\alpha \mid \beta$	$A \rightarrow \alpha A \mid \beta$




```
f()
{
f();
others;
}
```

Infinite loop




```
f()
{
}
other;
f();
}
```

Conversion of left recursion to right recursion :

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

This grammar is a left recursive grammar. To convert it we will write it in the form of

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

That means,

$$A \rightarrow A\alpha \mid \beta \equiv \begin{matrix} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{matrix}$$

Example: 01

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

Sol'n: This is a left recursive grammar. We will compare it with $A \rightarrow A\alpha \mid \beta$

So we can see,

$$\underset{A}{E} \rightarrow \underset{A}{E} \; \underset{\alpha}{+ \; T} \mid \underset{\beta}{T}$$

According to the rule of left recursion removing we will place, $A \rightarrow \beta A'$ and $A' \rightarrow \alpha A' \mid \epsilon$

~~in~~

$$E \rightarrow TE'$$

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

(Ans)

Example: 02

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

$$S \rightarrow (L) \mid x$$
$$L \rightarrow L, S \mid S$$

Sol^n:

Here in the first line of the grammar we can see that the production is free from left recursion. So, we don't have to apply anything there. But in the second line left recursion is available and we have to remove it.

$$S \rightarrow (L) \mid x$$
$$L \rightarrow SL'$$
$$L' \rightarrow , SL' \mid \varepsilon$$

Example 03

$S \rightarrow S051 \mid 01$

Soln

$S \rightarrow 01S'$

$S' \rightarrow 051S' \mid \epsilon$

Example 04

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

$B \rightarrow Be \mid b$

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

$A \rightarrow \beta A'$

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

Soln

$\underset{A}{A} \rightarrow \underset{A}{A}\underset{\alpha}{Bd} \mid \underset{A}{A}\underset{\alpha}{a} \mid \underset{\beta}{a}$

So, $A \rightarrow aA'$

$A' \rightarrow BdA' \mid aA' \mid \epsilon$

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

Again,

$$\underset{A}{B} \rightarrow \underset{A}{Be} \mid \underset{\beta}{b}$$
$$\alpha$$

$$B \rightarrow bB'$$
$$B' \rightarrow eB' \mid \varepsilon$$

So,

$$A \rightarrow aA'$$
$$A' \rightarrow BdA' \mid aA' \mid \varepsilon$$
$$B \rightarrow bB'$$
$$B' \rightarrow eB' \mid \varepsilon$$

(A)

Example 05

$E \rightarrow E + T \mid T$

$T \rightarrow T \times F \mid F$

$f \rightarrow id$

Sol^n

$E \rightarrow TE'$

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

$T \rightarrow fT'$

$T' \rightarrow \times fT' \mid \epsilon$

$f \rightarrow id$

(A)

Left Factoring

If RHS of more than one production starts with the same symbol, then such a grammar is called as grammar with common prefixes.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

⇒ Such grammar creates a problematic situation for top-down parsers

⇒ Top down parsers can not decide which production must be chosen to parse the string in hand.

To remove this confusion we use left factoring.

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for topdown parsers.

⇒ We make one production for each common prefixes.
⇒ The common prefix may be a terminal or a nonterminal or a combination of both.
⇒ Rest of the derivation is added by new productions.

The grammar we obtained after the process is called as left factored grammar.

Examples:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

Soln

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$$

01

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Soln:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow \varepsilon \mid eS$$
$$E \rightarrow b$$

Q2 $A \to aAB \mid aBc \mid aAc$

Sol^n

$A \to aA'$

$A' \to AB \mid Bc \mid Ac$

Again, there is a common prefixes.

$A \to aA'$

$A' \to Bc \mid AA''$

$A'' \to B \mid c$

Q3 $S \to bSSaaS \mid bSSaSb \mid bSb \mid a$

Sol^n

$S \to bSS' \mid a$

$S' \to SaaS \mid SaSb \mid b$

$S \to bSS' \mid a$

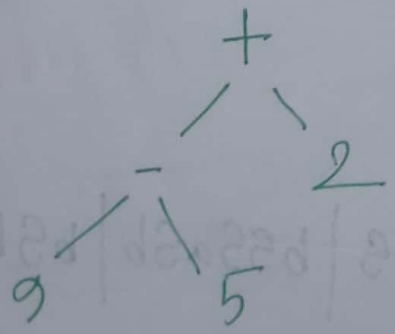
$S' \to \cancel{SaaS} \mid SaS'' \mid b$

$S'' \to aS \mid Sb$

#Abstract Syntax: A useful starting point for designing a translator is a data structure called an abstract syntax tree.

In an abstract syntax tree for an expression, each interior node represents an operator; the children of the node represents the operand of the operator.

What are the significant parts of the expression?



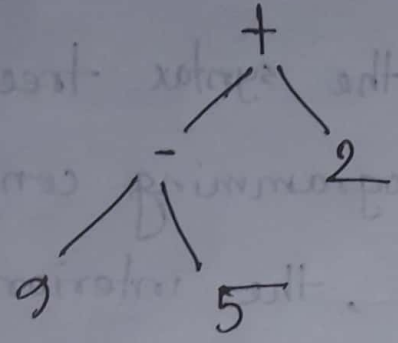
A sum expression has its two operand expressions as its significant parts.

Concrete Syntax ~~Abstract syntax trees~~ or simply syntax trees, resemble parse trees to an extent. However in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent nonterminals.

Many nonterminals of a grammar represent programming constructs, but others are helpers of one sort or another such as those representing terms, factors, or other variations of expressions.

In the syntax tree these helpers typically are not needed and are hence dropped. To emphasize the contrast, a parse tree is sometimes called a concrete syntax tree, and the underlying grammar is called a concrete syntax for the language.

What does the expression look like?



The same sum expression can look in different ways:

~~2+3~~ infix	2+3
prefix	(+ 2 3)
postfix	(2 3 +)
JVM	bipush 2 bipush 3 iadd
English	the sum of 2 and 3

#Removal of white space and comments //

The expression translator sees every character in the input, so extraneous characters, such as blanks, will cause it to fail.

Most languages allow arbitary amounts of white space to appear between tokens. Comments are likewise ignored during parsing, so they may also be treated as white space.

If white space is eliminated by the lexical analyzer, the parser will never have to consider it.

```
i=0;
while(string[i] != '\0')
{   if(string[i] == whitespace or tab or newline)
    i++;
}
```

#Reading Ahead: A lexical analyzer may need to read some characters before it can decide on the token to be turned to the parser. For example, a lexical analyzer for C or Java must read ahead after it sees the character >. If the next character is =, then > is part of the character sequence >=, the lexeme for the token for the greater than or equal to operator. Otherwise > itself forms the greater than operator, and the lexical analyzer has read one character too many.

Common Programming Errors

1. Lexical errors include misspelling of identifiers, keywords, etc.
2. Syntactic errors include misspaced semicolon, extra or missing braces
3. Semantic errors include type mismatch.
4. Logical errors can be anything by a programmar.

Tokens

A token is a sequence of characters that can be treated as a unit / single logical entity.

Typical tokens are : keywords, identifiers, seperators, etc.

```
int a = 15;
```

5 tokens here.

keywords : for, if, int, while, etc

identifiers : variable name, function name, etc.

operators : +, -, *, =, etc

Seperator : , ; etc

Constants : Any numbers

Lexeme

A sequence of input characters that comprises a single token is called a lexeme.

"int" "a", "=", "18"

Pattern

Pattern is a rule describing all those lexemes that can represent a particular token in a source program.

These are some predefined rules for every lexeme to be identified as a valid token these rules are defined by grammar rules, by means of pattern.

Examples:

```
int main()
{
    int b;
}
```

⇒ Lexical analyzer read int and finds it to be valid and accepts as token.

⇒ read main and finds it to be a valid function name and accepts as token.

⇒ Similarly (,), {, int, b, ;, }

total tokens: 9

53

⇒ int main ()

{

// 2 variables declared below

int a, b;

a = 10;

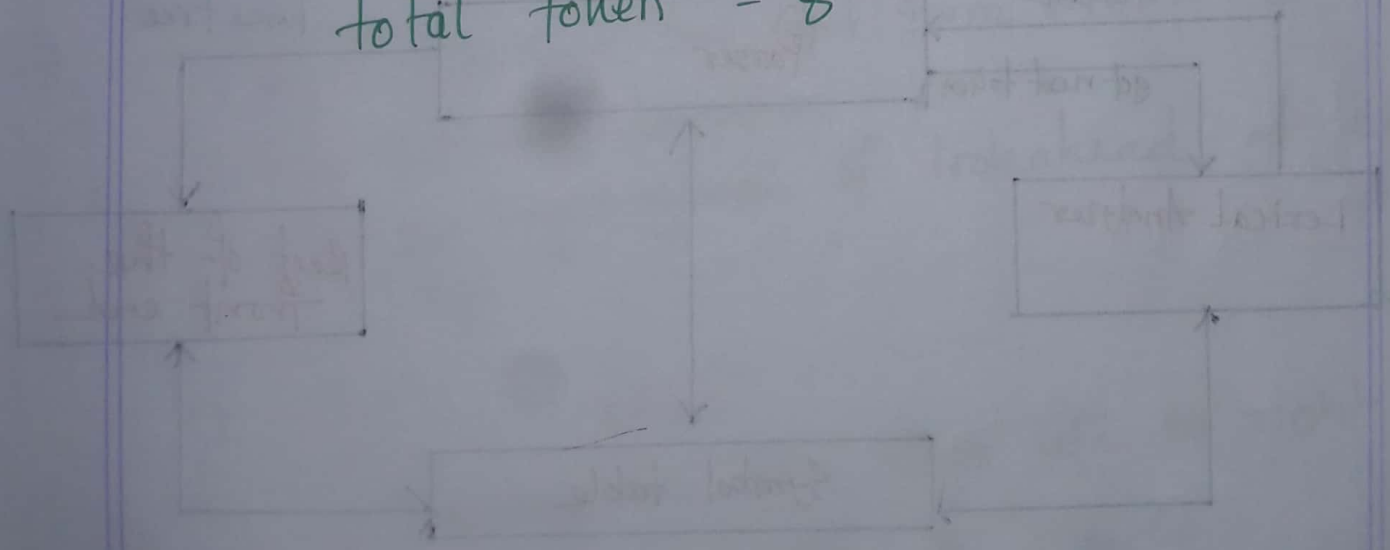
return 0;

total tokens = 18

}

⇒ printf ("%d", &j);

total token = 8

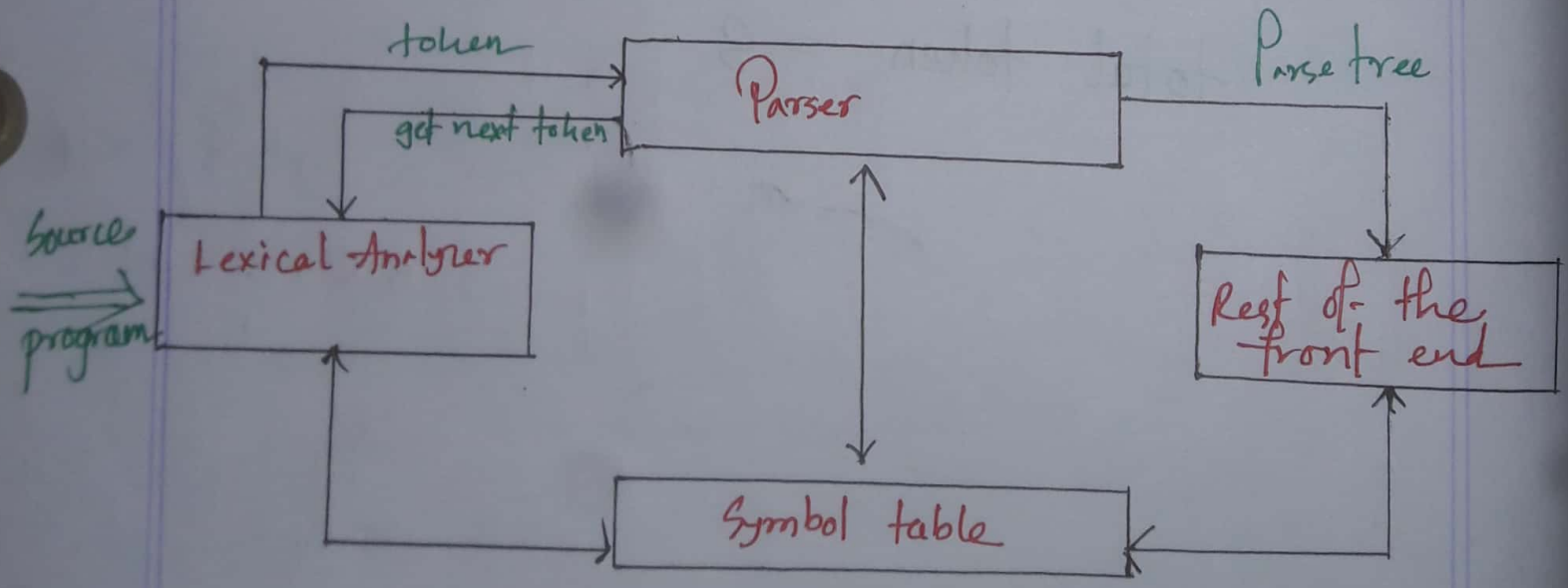


Syntax Analysis

⇒ The parser obtains a string of tokens from the lexical analyzer.

⇒ Verifies that the string can be the grammar for the source language.

⇒ Detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.



Some important topics required to understand parsing properly.

1. Context free grammar
2. Ambiguity
3. Left Recursion
4. Left factoring

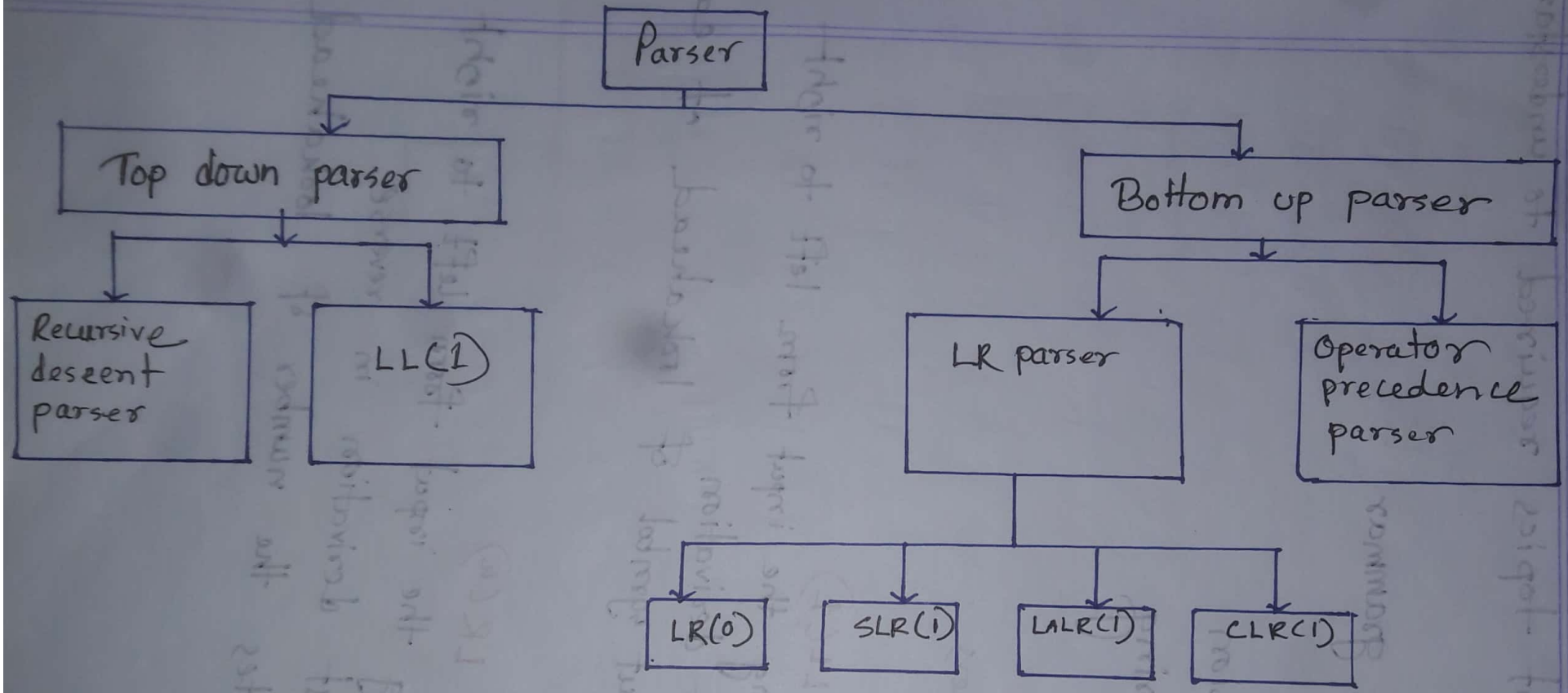
Types of Parsers

LL(1)

L ⇒ Scanning the input from left to right
L ⇒ Leftmost derivation
1 ⇒ One input symbol of lookahead at each step

LR(k)

L ⇒ Scanning the input from left to right
R ⇒ Rightmost derivation in reverse
k ⇒ K denotes the number of lookahead



57

SLR(1) ⇒ Simple LR parser

LALR(1) ⇒ Look-Ahead LR parser

CLR(1) ⇒ Canonical LR parser

Top down parser

Top down parsing can be viewed as the the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder. Equivalently, top down parsing can be viewed as finding a leftmost derivation for an input string. To design a top-down parser, a grammar must not be in a left recursive form and also free from common prefixes problem.

Design a top down parser for the following grammar where the input is id + id * id

$E \rightarrow TE'$

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

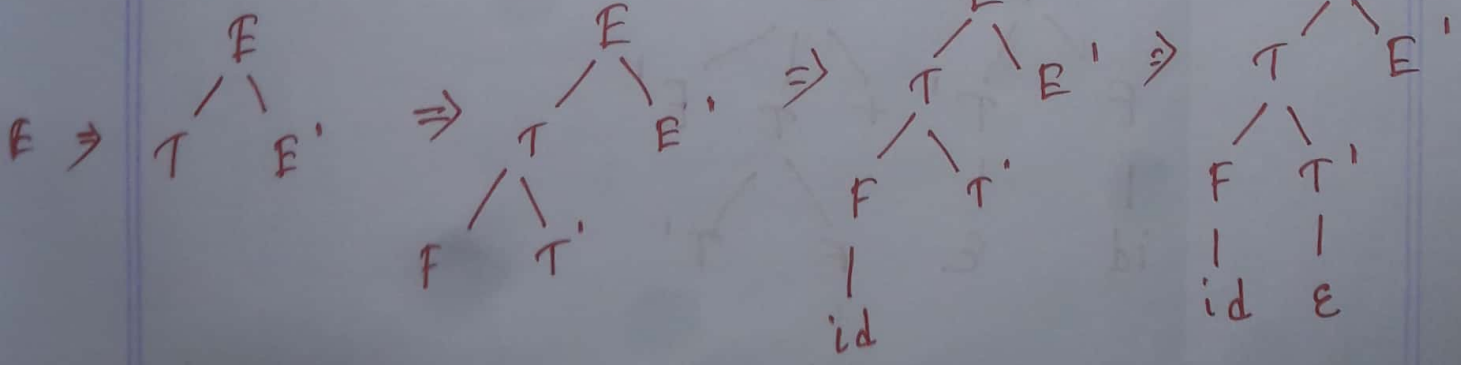
$T \rightarrow FT'$

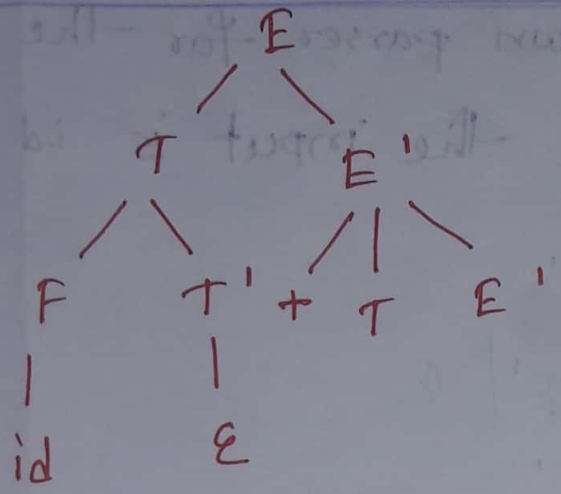
$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

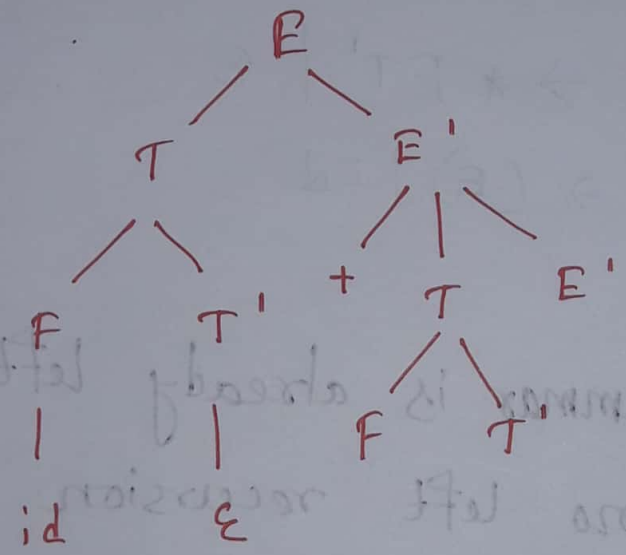
Soln :

The grammar is already left factored. And no left recursion available here. So, we can apply top-down parsing here.

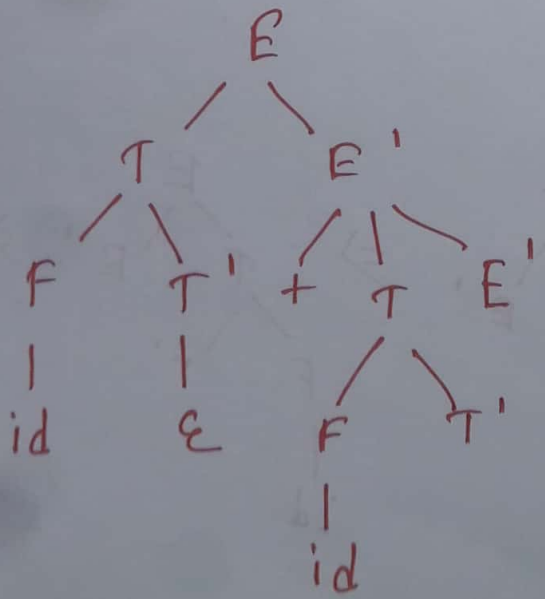




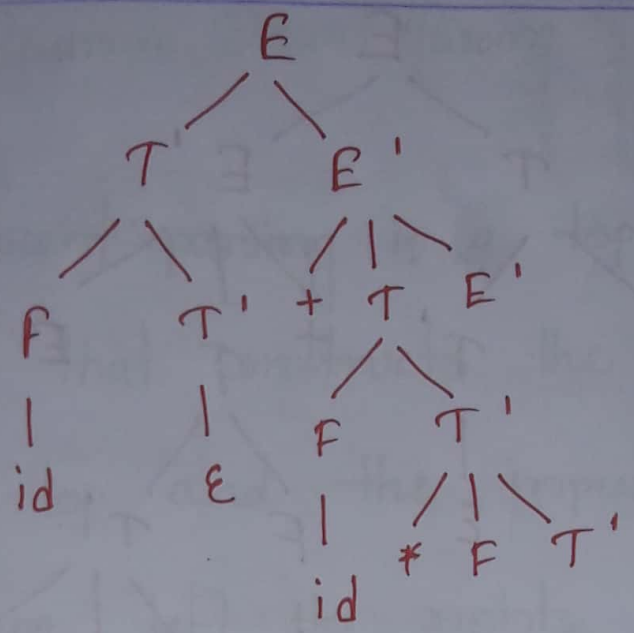
c)

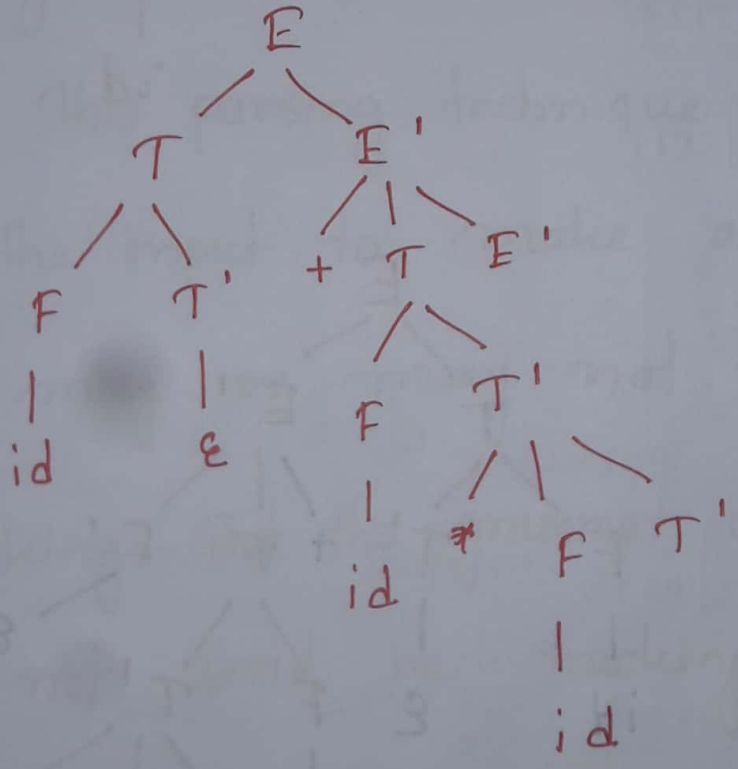


c)

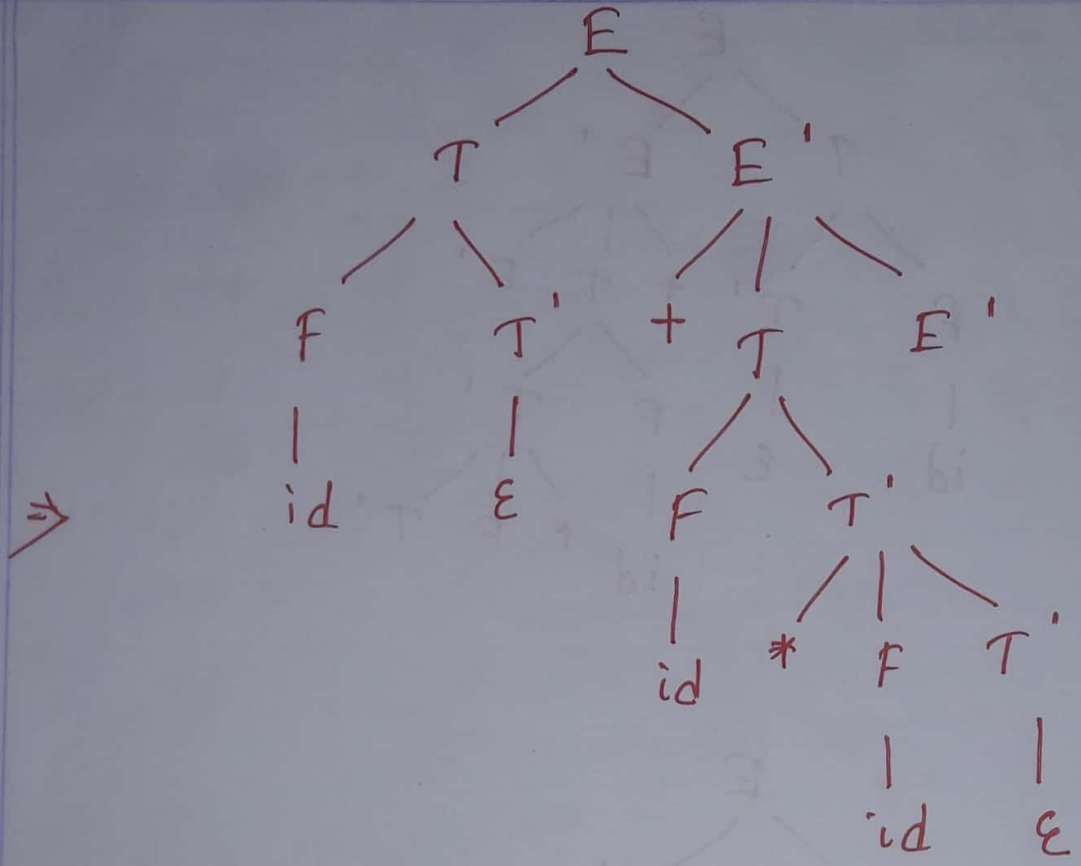


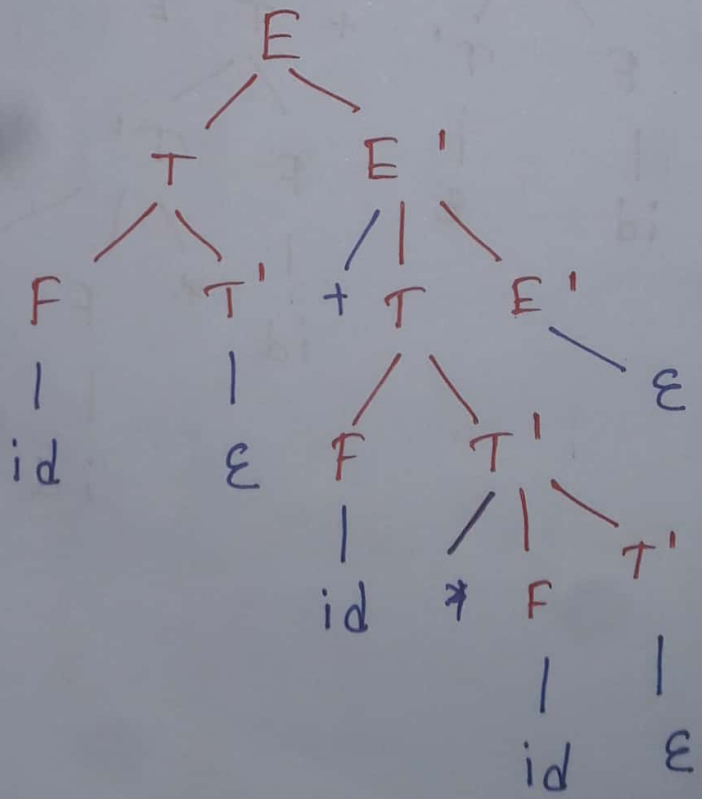
61





62





Recursive Descent Parsing

Recursive descent parsing is a top-down parsing technique that constructs the parse tree from the top and the input is read from from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it can not avoid back-tracking (if not left factored).

Recursive Descent Parsing

Recursive descent parsing is a top-down parsing technique that constructs the parse tree from the top and the input is read from from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it can not avoid back-tracking (if not left-factored).

Predictive Parsing

A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing. This parsing technique is regarded as it uses context free grammar which is recursive in nature.

Backtracking

Top down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched).

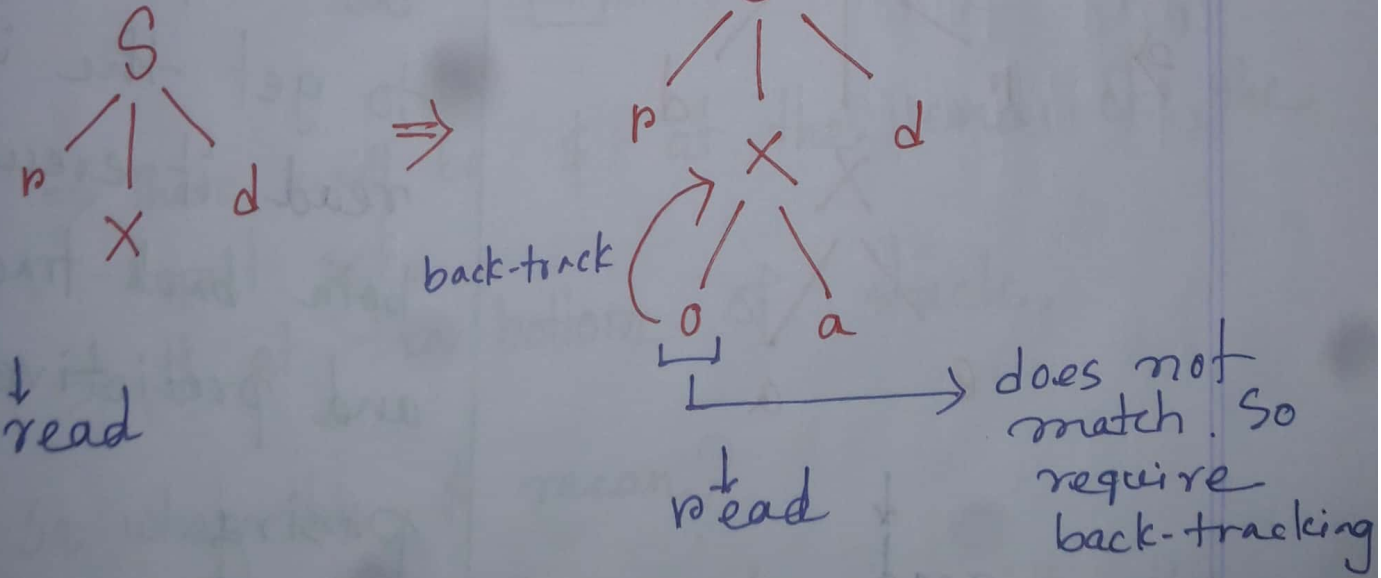
To understand this, take the following example of CFG.

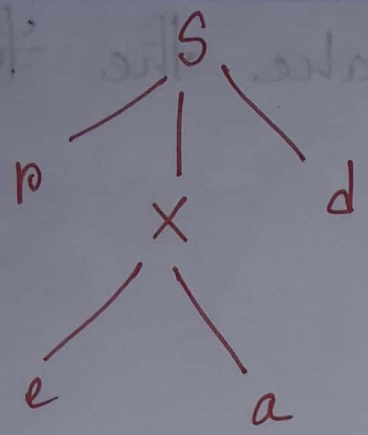
$S \rightarrow rXd \mid r2d$

$X \rightarrow oa \mid ea$

$2 \rightarrow ai$

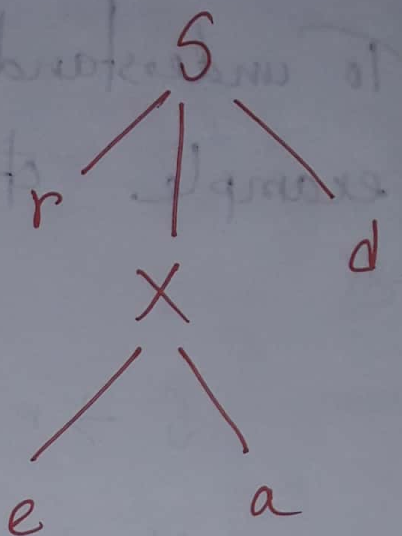
Consider input = read





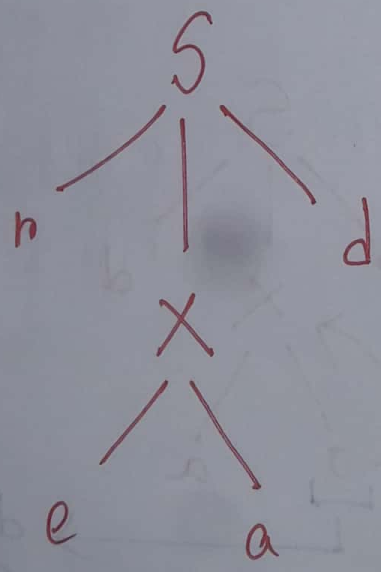
read

⇒



read

2)



read

So, the parse tree to get the input read, it execute both back-tracking and predictive parsing.

LL(1) Parser

L ⇒ Left to right

L ⇒ Left most derivation

(1) ⇒ No. of lookahead
how many symbol to see to make a desision

input set : 

abcd

LL(1) parser : 

⇒ stack

⇒ There will be $ at the end of the input set.
⇒ $ at the bottom of stack.

So, what this $ mean?

It means we are done with the input set also with the stack. Then we can generate a parsing table.

To generate the parsing table we will learn a function called

FIRST & FOLLOW.

FIRST (A) ⇒ is a set of terminal (s) that begin after derivation from A.

FOLLOW (A) ⇒ is the terminal which could follow a variable in the process of derivation.

Rules to derive ~~FOLLOW~~ FIRST :

Rule 1 :- for a production rule $X \to \epsilon$,

$$FIRST(X) = \{\epsilon\}$$

Rule 2 :- for any terminal symbol $X \to a$

$$FIRST(X) = \{a\}$$

Rule 3 : for any production rule $X \to Y_1 Y_2 Y_3$

FIRST (X)

⇒ If $\epsilon \notin$ FIRST (Y_1), then FIRST (X) = FIRST (Y_1)

⇒ If $\epsilon \in$ FIRST (Y_1), then FIRST (X) =

$$\{FIRST(Y_1) - \epsilon\} \cup FIRST(Y_2 Y_3)$$

FIRST $(Y_2 Y_3)$

⇒ If $\epsilon \notin$ FIRST (Y_2), then FIRST $(Y_2 Y_3)$ = FIRST (Y_2)

⇒ If $\epsilon \in$ FIRST (Y_2), then FIRST $(Y_2 Y_3)$ =

$$\{FIRST(Y_2) - \epsilon\} \cup FIRST(Y_3)$$

Similarly, we can make expansion for any production rule.

$$X \Rightarrow Y_1 Y_2 Y_3 \cdots Y_n$$

Rules to derive FOLLOW

Rule 01 :- for the start symbol S, place $ in Follow(S).

Rule 02 :- for any production rule, $A \Rightarrow \alpha B$

$$FOLLOW(B) = FOLLOW(A)$$

Rule 03 :- for any production rule $A \Rightarrow \alpha B\beta$.

⇒ If $\epsilon \notin FIRST(\beta)$, then $FOLLOW(B) = FIRST(\beta)$

⇒ If $\epsilon \in FIRST(\beta)$, then $FOLLOW(B) =$

$$\{FIRST(\beta) - \epsilon\} \cup FOLLOW(A)$$ [if β is the last element]

$\{FIRST(\beta) - \epsilon\} \cup$ replace with ϵ and generate the terminals [if β is not the last element]

Important notes:

1. ε may appear in the first function of a non-terminal.

2. ε will never appear in the follow function of a non-terminal.

3. Before calculating the first and follow functions eliminate left recursion from the grammar, if present.

4. We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Example 01 :

$S \rightarrow aBDh$

$B \rightarrow cC$

$C \rightarrow bC \mid \varepsilon$

$D \rightarrow EF$

$E \rightarrow g \mid \varepsilon$

$F \rightarrow f \mid \varepsilon$

Soln

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

$FIRST(B) = \{c\}$

$FIRST(C) = \{b, \varepsilon\}$

$FIRST(D) = \{FIRST(E) - \varepsilon\} \cup FIRST(F)\}$

$= \{g, f, \varepsilon\}$

$FIRST(E) = \{g, \varepsilon\}$

$FIRST(F) = \{f, \varepsilon\}$

FOLLOW (S) = {$}

FOLLOW (B) = FIRST (f)

= {g, f, ε} Here we got an empty. So replace.

= {g, f, h}

FOLLOW (D) = {h}

FOLLOW (C) = FOLLOW (B)

= {g, f, h}

FOLLOW (E) = {FIRST (F) - ε} ∪ FOLLOW (D)

= {f, h}

FOLLOW (F) = FOLLOW (D)

= {h}

Problem 02

$S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow b$

$C \rightarrow g$

Sol?: The grammar is left recursive. So we have to eliminate it.

$S \rightarrow A$

$A \rightarrow aBA'$

$A' \rightarrow dA' \mid \epsilon$

$B \rightarrow b$

$C \rightarrow g$

FIRST(S) = FIRST(A)
= {a}

FIRST(A) = {^}

FIRST(B) = {b}

FIRST(A') = {d, ε}

FIRST(C) = {g}

FOLLOW

FOLLOW(S) = {$}

FOLLOW(A) = FOLLOW(S)
= {$}

FOLLOW(A') = FOLLOW(A)
= {$}

FOLLOW(B) = {FIRST(A' - ε)} ∪ {FOLLOW(A)}
= {d, $}

FOLLOW(C) = NA.

Problem 03

$$S \rightarrow (L) \mid a$$

$$L \rightarrow SL'$$

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

Soln

$$FIRST(S) = \{ (, a \}$$

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

$$FIRST(L') = \{ , , \varepsilon \}$$

FOLLOW

$$FOLLOW(S) = \{ \$ \} \cup FIRST(L')$$

Here, $FIRST(L')$ contains ε so,

$$= \{ \$ \} \cup \{ FIRST(L') - \varepsilon \} \cup \{ FOLLOW(L) \}$$

$$= \{\$, ,,)\}$$

$$\text{FOLLOW}(L) = \{)\}$$

$$\text{FOLLOW}(L') = \text{FOLLOW}(L)$$
$$= \{)\}$$

Problem 03:

$$S \rightarrow AAb \mid BbBa$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow \varepsilon$$

Soln:-

$$\text{FIRST}(S) = \text{FIRST}(A) \cup \text{FIRST}(B)$$

But both FIRST(A) and FIRST(B) contain ε. So,

$$= \{\text{FIRST}(A - \varepsilon)\} \cup \{a\} \cup \{\text{FIRST}(B - \varepsilon)\} \cup \{b\}$$

$$= \{a, b\}$$

FIRST (A) = ε

FIRST (B) = ε

FOLLOW

FOLLOW (S) = {$}

FOLLOW (A) = ~~FIRST~~ {a, b}

FOLLOW (B) = {a, b}

(A)

LL(1) Parsing Table

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

Soln: First of all, we will find the FIRST & FOLLOW of this grammar.

Productions		FIRST	FOLLOW
$E \rightarrow TE'$	E	{id, (}	{$,)}
$E' \rightarrow +TE' \mid \varepsilon$	E'	{+, ε}	{$,)}
$T \rightarrow FT'$	T	{id, (}	{$,), +}
$T' \rightarrow *FT' \mid \varepsilon$	T'	{*, ε}	{$,), +}
$F \rightarrow id \mid (E)$	F	{id, (}	{$,), +, *}

	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→id			F→(E)		

If the grammar has left recursion eliminate it.

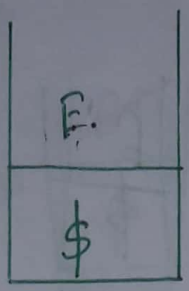
If the grammar has nondeterminism, apply left factoring to make it deterministic. Then we can make LL(1) parsing table. But not guaranteed that we can generate parse tree.

**If same cell has two productions then LL(1) parse tree impossible

8)

Input : id + id $

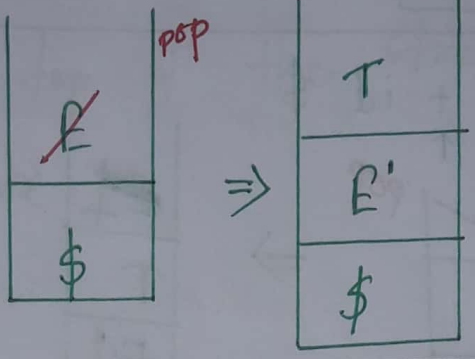
Stack :

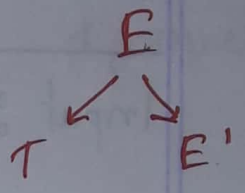


E

Input : id + id $

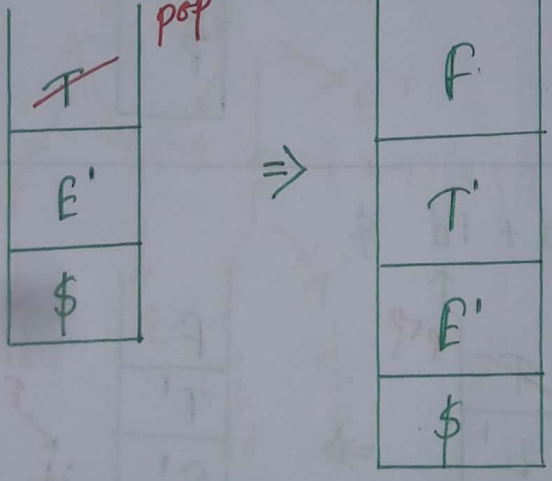
Stack :

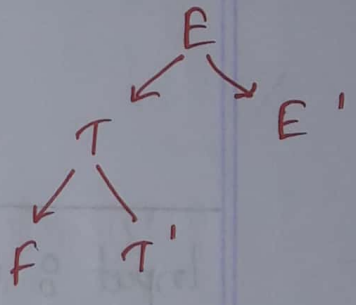




Input : id + id $

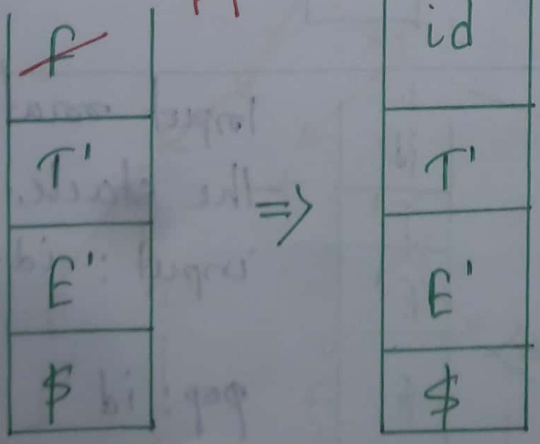
Stack :





Input : id + id $

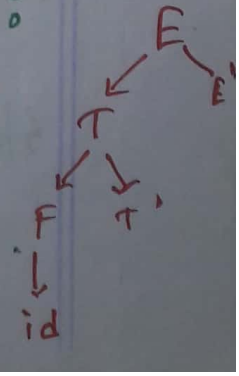
Stack :



top of the stack & input matches. So, increment the pointer of the input :

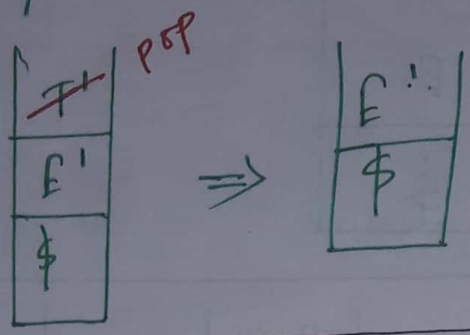
id + id $

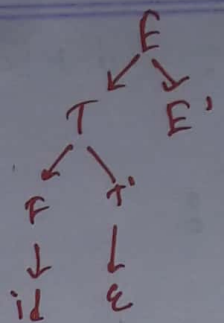
And pop id.



Input : id + id $

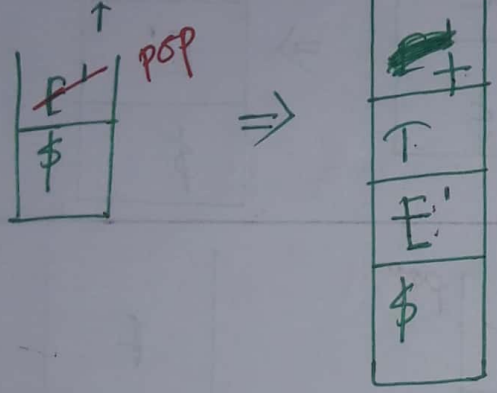
Stack :





Input : id + id $

Stack :

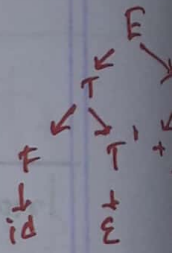


input matches top of the stack.

So, input:

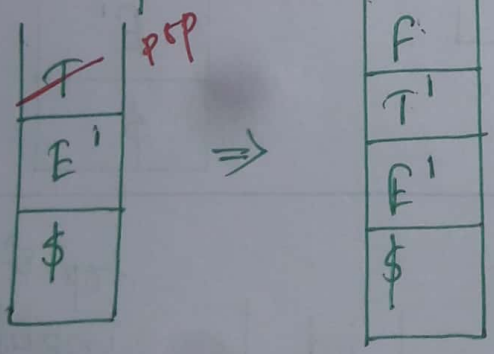
id + id $

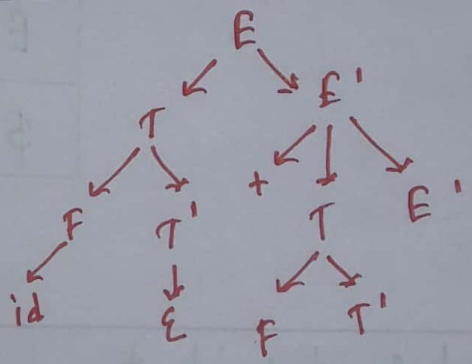
pop +



Input : id + id $

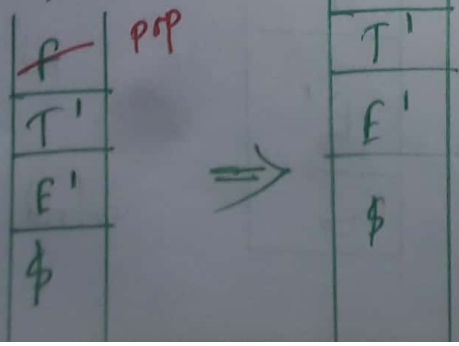
Stack :





Input : id + id $

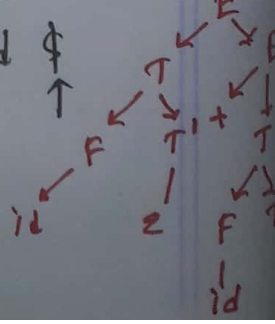
Stack :



Input matches top of the stack. So,

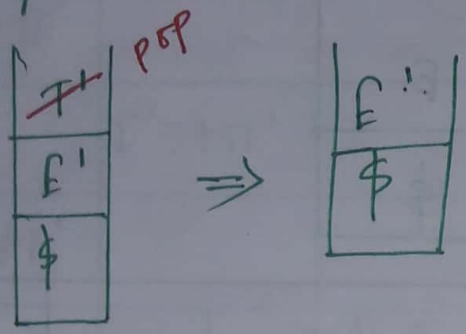
input : id + id $

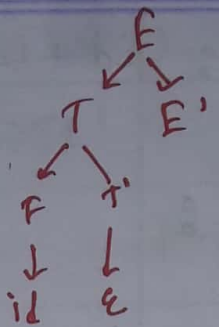
pop : id



Input : id + id $

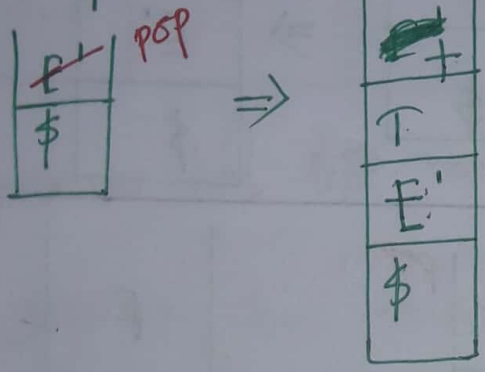
Stack :





Input : id + id $

Stack :

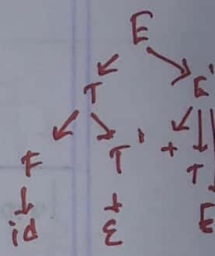


input matches top of the stack.

So, input:

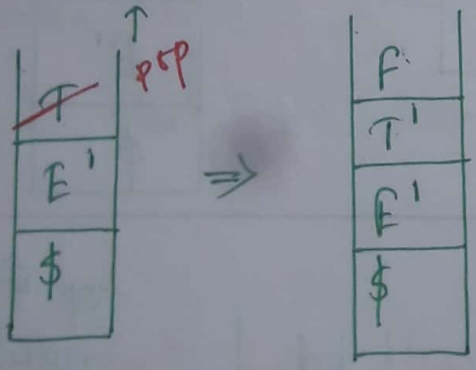
id + id $

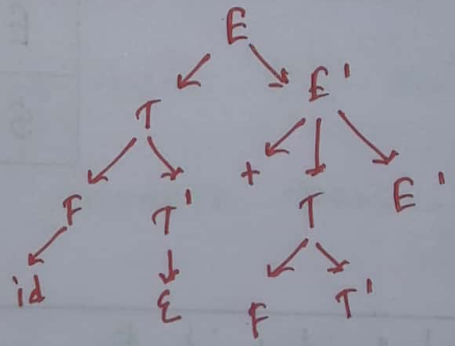
pop +



Input : id + id $

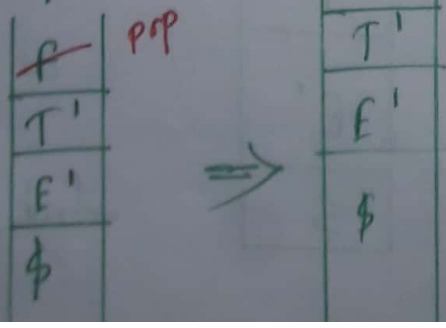
Stack :





Input : id + id $

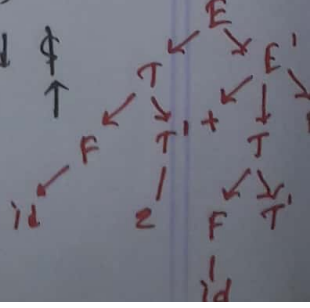
Stack :



Input matches top of the stack. So,

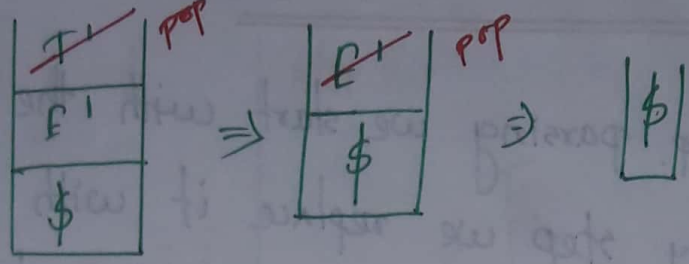
input : id + id $

pop : id

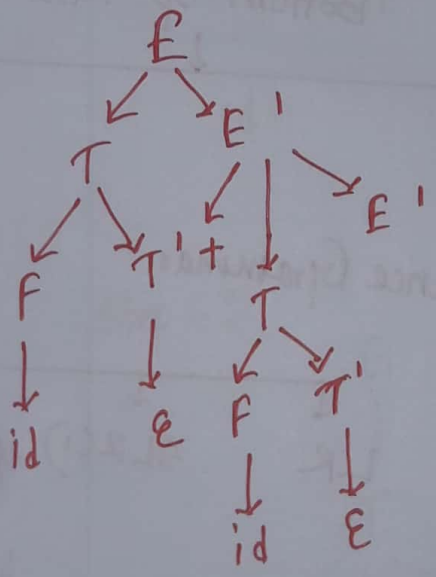


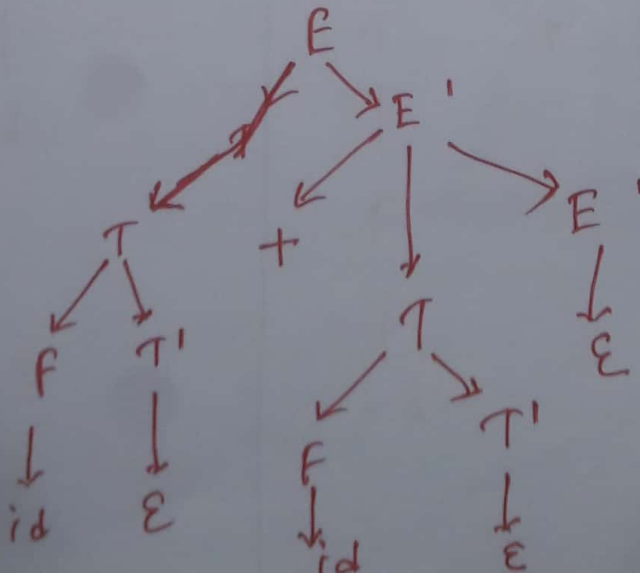
Input : id + id $
↑

Stack :



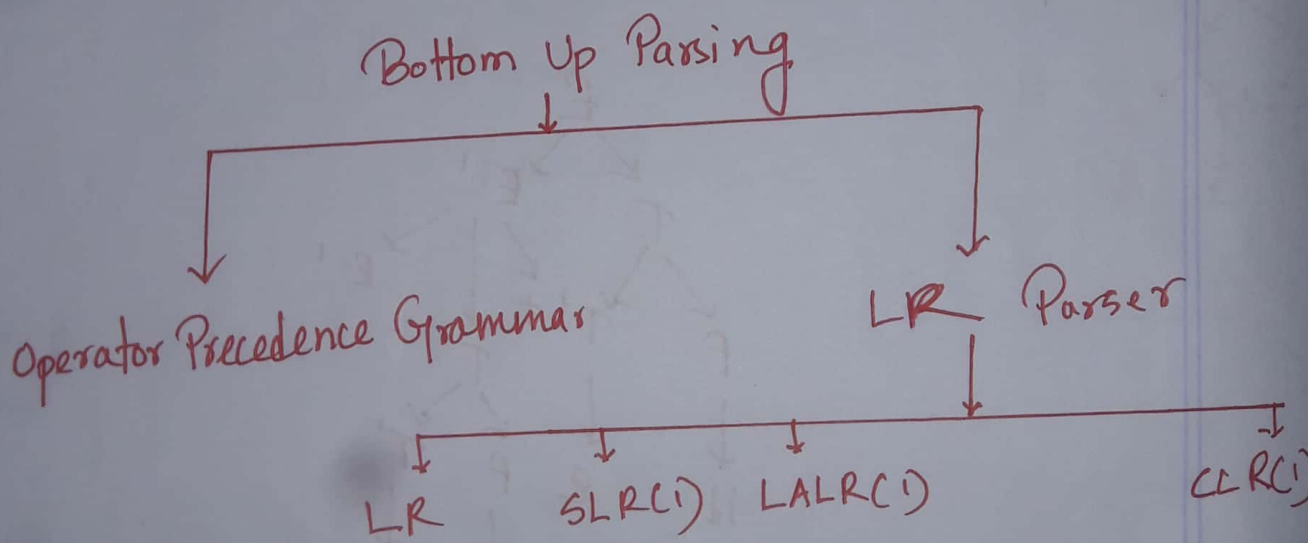
Input matches top of the string. $ meets $. So, terminate.





Bottom Up Parsing

In bottom up parsing we start with the string symbol and step by step we replace it with its production. Ultimately we end up with the starting state. It is also known as Shift Reducing Parser.



65

Input string: id * id

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

step 1: id * id

step 2: F * id

id

step 3: T * id

F

id

step 4: T * F

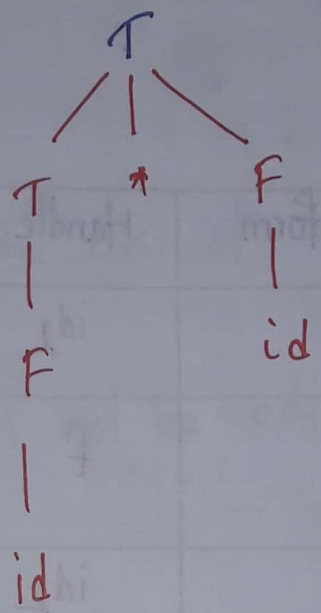
T

F

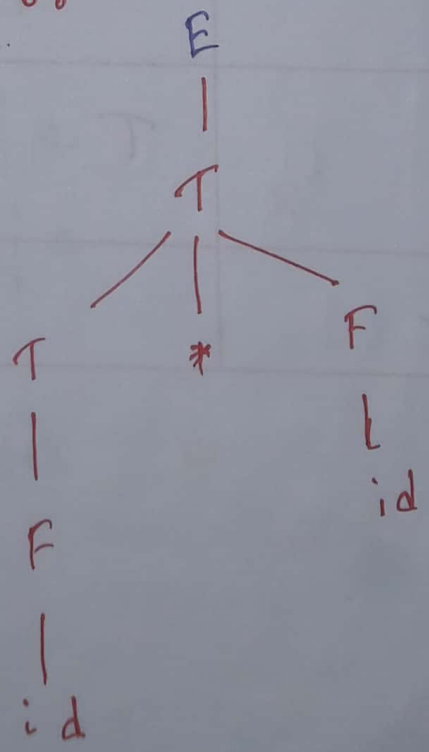
id

id

step 5:



step 6:



Handle Pruning

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

$id_1 * id_2$

Right sentential form	Handle	Reducing Production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$T \to T * F$
T	T	$E \to T$
E		

Operator Grammar

Operator grammar is used to define mathematical operator.

Two properties :

1. ε can not be in the RHS of any production.

2. Two nonterminals can not be adjacent to each other.

Examples :

1. $E \rightarrow E+E \mid E*E \mid id$ ✓

2. $X \rightarrow ABX \mid a$ ✗
   $B \rightarrow bXb \mid a$

But $X \rightarrow AbXbX \mid a \mid AaX$ ✓

Operator Precedence Relation

3 types of relation.

1. $a \gtrdot b$ [a has higher precedence than the terminal b]

2. $a \lessdot b$ [b has higher precedence than the terminal a]

3. $a \doteq b$ [a and b both have same precedence]

Operator Precedence Table

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

	id	+	*	$
id	–	$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	–

string: id + id * id

⇒ Here we will use a stack.
⇒ We will add $ at the bottom of the stack as well as at the end of the input string.
⇒ We will compare the value of stack (top) with the input (cursor)
⇒ Stack values are the values from column (left)
⇒ Input values are the values from row (top)

Comparison:

⇒ If the values of stack are smaller than input string then,

* push the value into the stack
* increment the cursor of input string

⇒ If the values of stack are greater than input string then

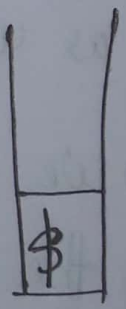
* pop from the stack
* reduce the tree

Step 1:

id + id * id $

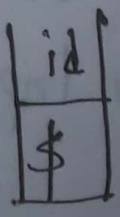
↑

Compare ($, id) = $ smaller so push the id into the stack & increment the lookahead/cursor



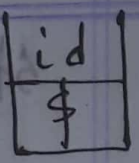
id + id * id $

↑

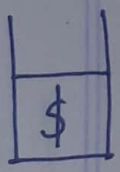


$E \rightarrow E+E \mid E*E \mid id$

Step 0: id + id * id $



Compare (id, +) = id greater so, pop id from the stack and reduce the grammar



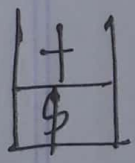
E

|

id + id * id $

Step 3:

Compare ($, +) = $ smaller so push + and increment the cursor



E

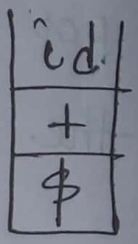
|

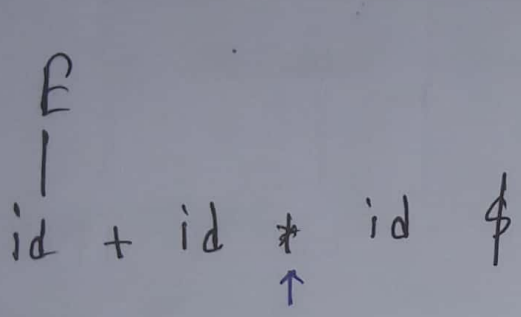
id + id * id $

9✓

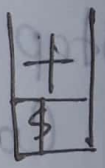
E ⇒ E+E | E*E | id

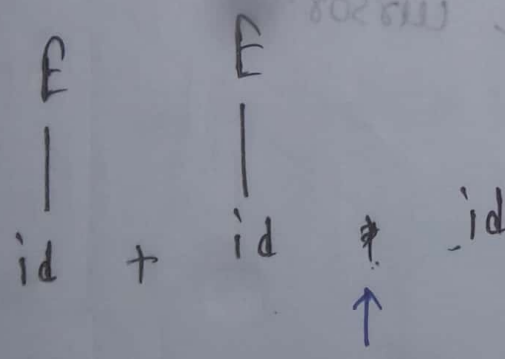
Step 4: Compare (+, id) = + smaller so push id into the stack & increment the lookahead



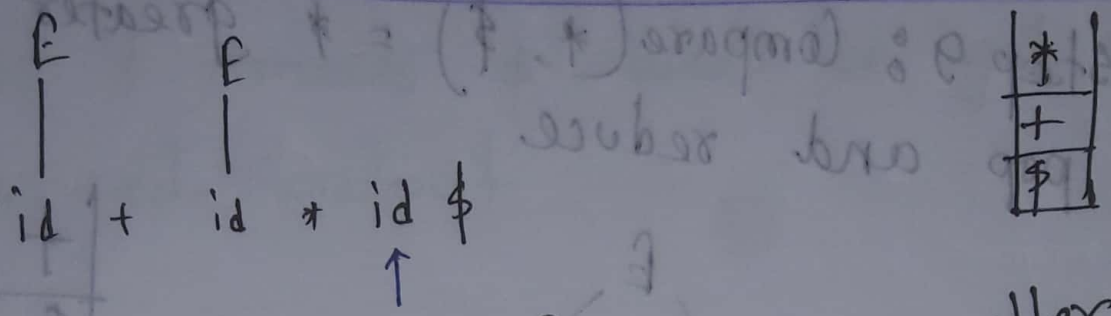


Step 5: Compare (id, *) = id greater so pop id and reduce the grammar

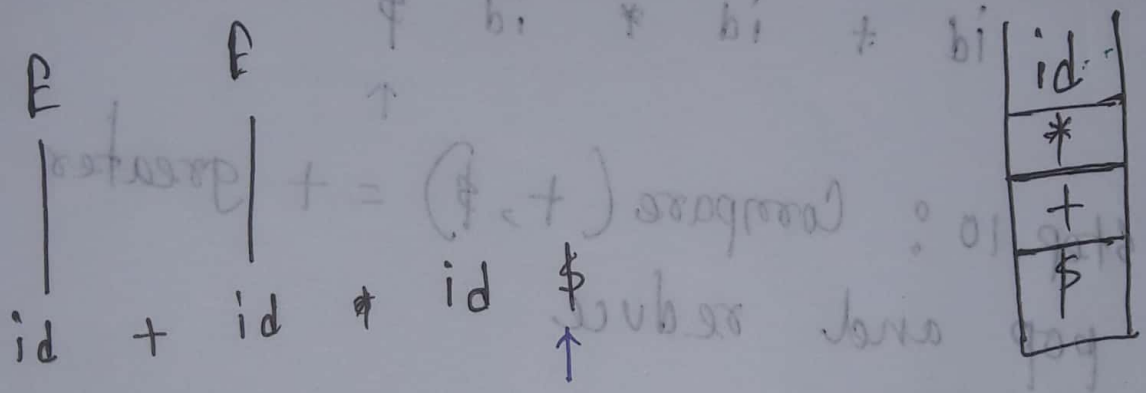




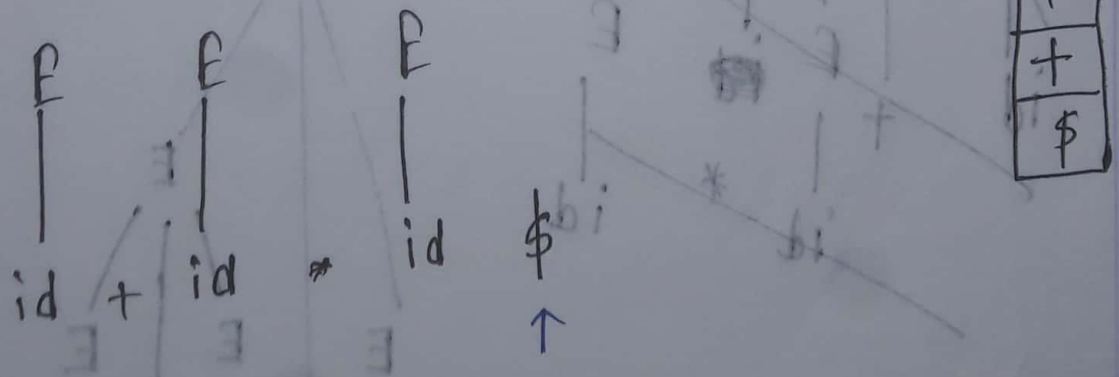
Step 6: Compare (+, *) = + smaller so push * and increment the cursor.



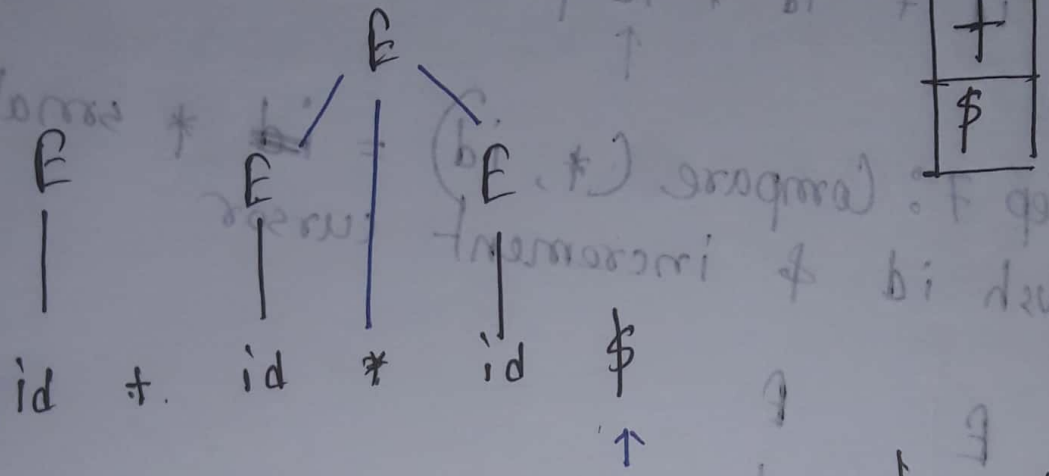
Step 7: Compare (*, id) = * smaller so push id & increment cursor



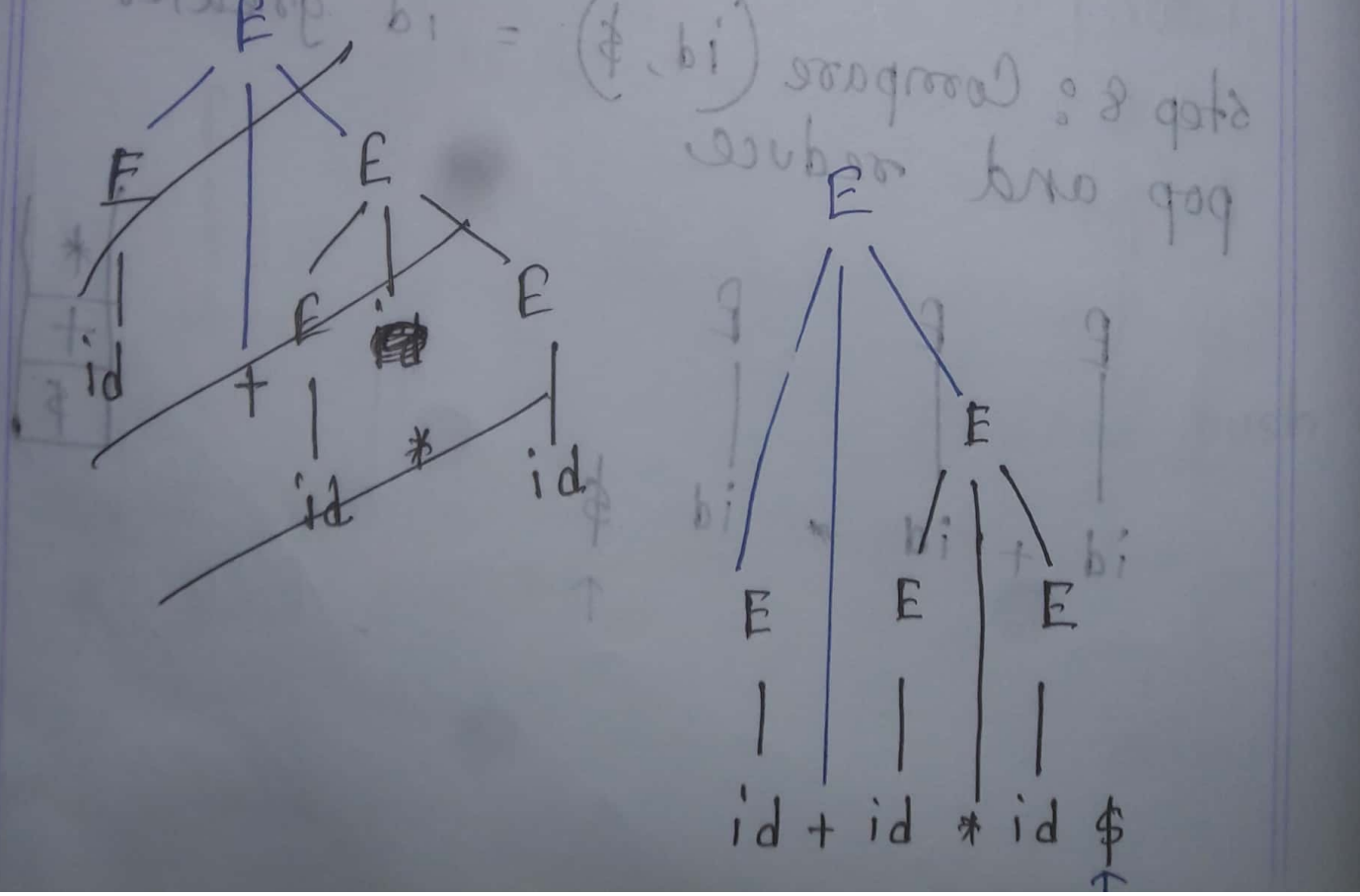
Step 8: Compare (id, $) = id greater so pop and reduce



Step 9: Compare (*, $) = * greater so pop and reduce



Step 10: Compare (+, $) = + greater so pop and reduce



So, $ meets $ and we got our desired output.