SYNTAX ANALYSIS:= (ROLE OF SYNTAX ANALYZER):=

→ Syntax Analyzer is also called as the parser, the parser obtains a stream of tokens from the lexical Analyzer

→ And, Verifies the tokens syntactically, if the tokens are syntactically correct then it will generate a parse tree.

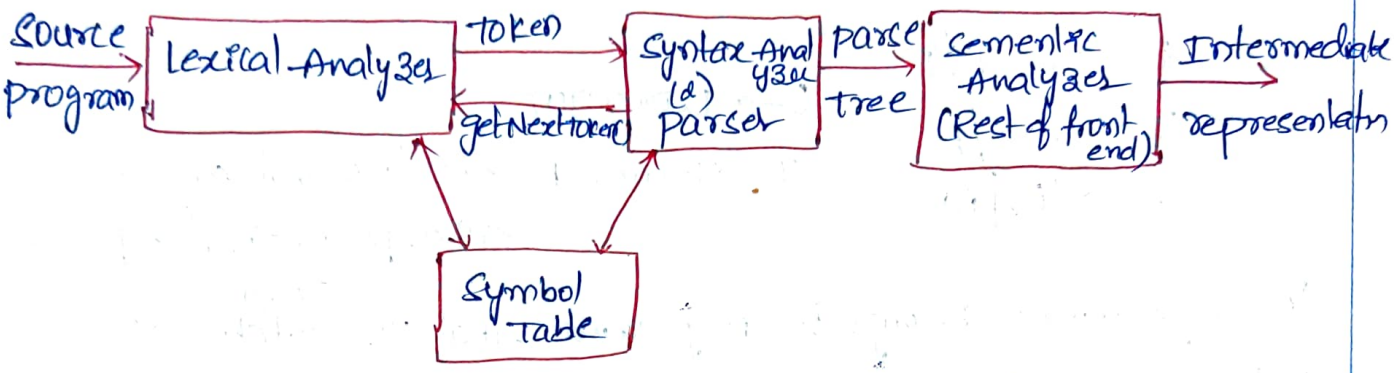


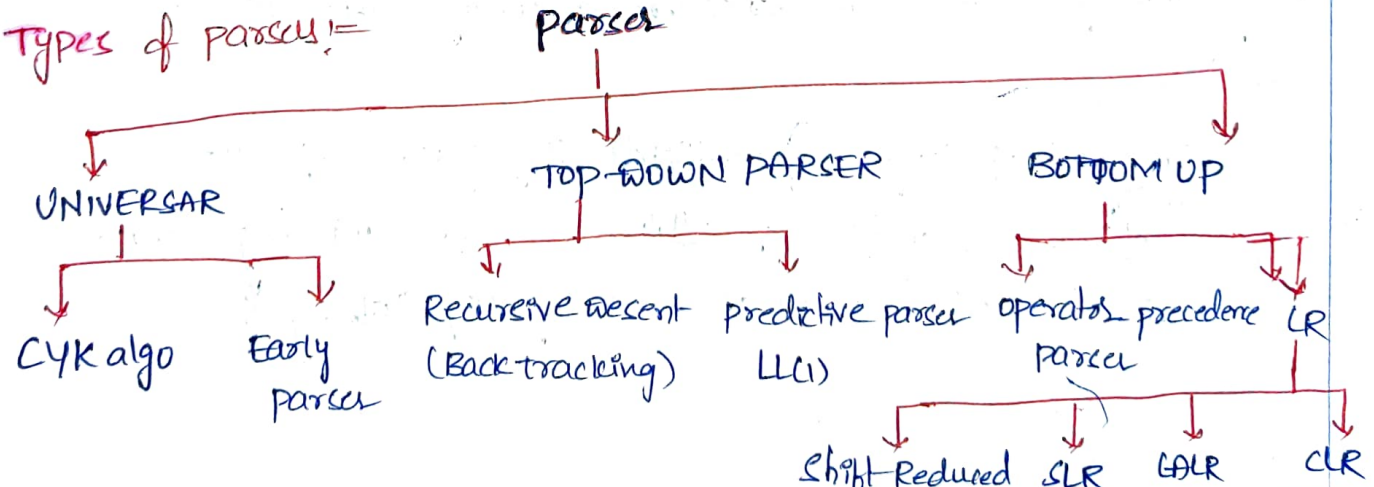
Fig:= position of parser in compiler model.

→ The main function of parser is to check the syntactic structure of i/p code, if the given i/p code is out of syntactic errors then it will generate a parse tree.

→ if i/p code is consists of syntax errors, then the it reports an errors to user, Now, the user will correct it is responsibility of the user to correct those errors.

→ Symbol Table communicate with both Lexical Analyzer & Syntax Analyzer for the token information.

Types of parser:=



→ In top-down parser construct the parse tree from top (root) to the bottom (leaves)

→ Bottom-up parser construct the parse tree from leaves to root (starts from i/p symbol and reduced to starting state)

Representative Grammars:=

Associativity and precedence are captured in the following grammar

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

E → Expression

T → Terms, F → Factors that can be either parathesized expression (or) id

These grammars belongs to LR grammars that are suitable for bottom-up parsing. This grammar cannot be used for top-down parsing

The following non-left-recursive grammar will be used for top-down parsing

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

Syntax Error Handling:=

Programming errors can occure at many different levels.

lexical Errors:

→ Include misspelling of identifiers, keywords, & operators

Eg:- identifies elipsesize instead of ellipsesize and missing quotes around text intended as a string. (eg: "String");

Syntax Errors:- Misplaced semicolon(;), extra & missing braces {, }.

Eg :- appearance of case statement without an enclosing switch.

Semantic Errors:- Eg:- (a+(b*) → missing parenthesis.

→Type mismatch b/w operators and operands.

Eg:- ① 2+a[i] → type mismatch.

② The return of a value in Java method with result type void.

logical Errors:-

→ errors can be anything from incorrect reasoning on the part of programming.

Eg:- In "c" - the assignment operator instead of comparision operator.

if (c=A) ✗ if (c==A) ✓

Errors Handles in parser:-

- should report the presence of errors clearly and accurately.
- should recovery from each error quickly enough to detect the subsequent errors.
- should not slow down the processing of remaining program

Errors Recovery Strategies:-

(i) Panic Mode Recovery:-

Eg:- int a, b {
printf(" ");}

→The parser discards i/p symbols at a time until one of a designated set of synchronizing tokens is found.

→The synchronizing tokens are delimeters ";" or }

→It is simple to implement & and does not goto a loop

(ii) phrase level Recovery:-

→ The parser peaform local correction on remaining i/p when the error is discovered.

→ The parser replaces the prefix of the remaining i/p by some strings that allows the parser to carry on its execution

Eg:= replace a (,) comma by semicolon (;), delete an extra semicolon, insert a missing semicolon (;)

① print(" "), → printf(" ");

Disadvantage:= Error correction is difficult when actual errors occure before the point of detection

(iii) Error production.

→ common Errors that can be encountered, we can augment the grammas for the language with productions that generate erroneous constructs.

→ use a new grammar for the parser.

(iv) Global correction:=

→ The aim is to make some changes while converting incorrect i/p string to a valid string

• Given an Incorrect i/p x & Grammar G find a parse tree for a related string w (using the given Grammar) such that no of changes (insertion/deletion) required to transform x to w is minimum

• Too costly to implement

(2) CONTEXT FREE GRAMMAR:=

Context Free Grammars consists of 4-tuples

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

- V → Set of variables (or) Non-terminals
- T → Set of terminals.
- P → Set of production Rules
- S → Start symbol.

(i) Non-terminal :=

→ Denotes set of strings.

- Uppercase letters early in alphabet A, B, C, ...
- S → Start symbol
- E, T, F (Expression – E, Term – T, Factor – F)

(ii) Terminal :=

→ Terminals are the basic symbols from which strings are formed

- Token-name is also called as terminal
- The following are terminal symbols.
  - (a) lowercase letters in the alphabets such as a, b, c.
  - (b) operator symbols such as +, *, ...
  - (c) punctuations ("{", "}", "(", ")", ",", ...) & digits 0, 1, ... 9

(iii) Production := consists of

(a) A Non terminal called head or leftside of the production. This production defines some of the symbols denoted by the head

(b) body or rightside → consisting of zero or more terminals and Non terminals.

* uppercase letters late in alphabet X, Y, Z represent Grammar symbols either to terminals or Non-terminals.

→ α, β, γ ... represent string of grammar symbol

→ A set of productions $A \to \alpha_1$, $A \to \alpha_2$ → $A \alpha K$ also written as $A \to \alpha_1 | \alpha_2 | \dots | \alpha K$

Product: $A \to \alpha$ (head ↑)

A ↓ Non-Terminal; α ↓ either terminal & Nonterminal (VUT)*

Eg:= Consides the grammar

$E \to E+E \quad E \to E*E \quad E \to I \quad I \to a$

$CFG = (\{E, I\}, \{+, *, a\}, P, E)$

(V, T, P, S)

→ ~~derive~~

Derivations:= Derivation is process of applying a sequence of production rules inorder to derive a string.

→ deriving an i/p string from the start symbol of Grammar in one or more steps by replacing the head of the production by its body of the production.

There are two types of Derivation.

(i) Left most Derivation (LMD)

(ii) Right most Derivation (RMD)

Left Most Derivation:= In each step we have to expand left-most Non-terminal by one of its production body.

Eg:= $E \to E+E | E*E | -E | (E) | id$

Derive a string id+id*id

LMD:=

$E \overset{LMD}{\Rightarrow} \underline{E}+E \quad (E \to E+E)$

$E \overset{lm}{\Rightarrow} id+\underline{E} \quad (E \to id)$

$\overset{lm}{\Rightarrow} id+\underline{E}*E \quad (E \to E*E)$

$\overset{lm}{\Rightarrow} id+id*\underline{E} \quad (E \to id)$

$\overset{lm}{\Rightarrow} id+id*id \quad (E \to id)$

Right most Derivation:=

→ Here, we have to expand the Right-most non-terminal by

RMD:=

$E \overset{rm}{\Rightarrow} \underline{E}+E \quad (E \to E*E)$

$\overset{rm}{\Rightarrow} E+E*\underline{E} \quad (E \to E*E)$

$\overset{rm}{\Rightarrow} E+\underline{E}*id \quad (E \to id)$

$\overset{rm}{\Rightarrow} \underline{E}+id*id \quad (E \to id)$

$\Rightarrow id+id*id \quad (E \to id)$

Parse tree:-

→ A Graphical representation of derivation is called parse tree.

There types of nodes in parse tree

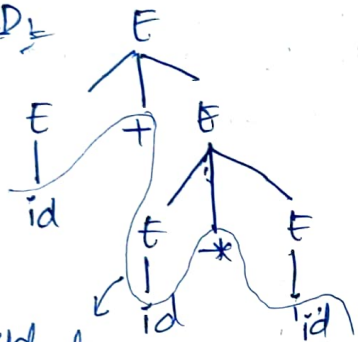
(i) Interior node → are Non terminals

(ii) children node → terminals

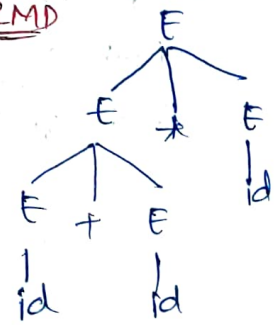
→ In parse tree the root node must be start symbol

construct parse tree for i/p string W= id+id * id

LMD:-



RMD



Yield of parse tree

Ambiguity:-

→ If a grammar (CFG G=(V,T,P,S)) produces more than one parse tree, then that grammar is called as ambiguous.

(i) more than one LMD (different)

(ii) more than one RMD (different).

Eg $E \to E+E \mid E*E \mid (E) \mid id$ string $w = id+id*id$

LMD1:-

$E \underset{lm}{\Rightarrow} E+E$

$\underset{lm}{\Rightarrow} id+E$

$\underset{lm}{\Rightarrow} id+E*E$

$\underset{lm}{\Rightarrow} id+id*E$

$\underset{lm}{\Rightarrow} id+id*id.$

RMD1:- $E \underset{rm}{\Rightarrow} E+E$

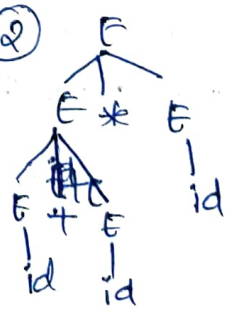
$\underset{rm}{\Rightarrow} E+E*E$

$\underset{rm}{\Rightarrow} E+E*id$

$\underset{rm}{\Rightarrow} E+id*id$

$\underset{rm}{\Rightarrow} id+id*id$

① 

② 

LMD2:- $E \Rightarrow E*E$

$\Rightarrow E+E*E$

$\Rightarrow id+E*E$

$\Rightarrow id+id*E$

$\Rightarrow id+id*id$

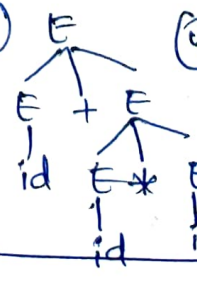
RMD2:- $E \underset{rm}{\Rightarrow} E*E$

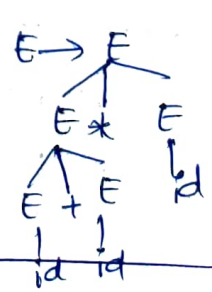
$\underset{rm}{\Rightarrow} E*id$

$\underset{rm}{\Rightarrow} E+E*id$

$\underset{rm}{\Rightarrow} E+id*id$

$\underset{rm}{\Rightarrow} id+id*id$

③ 

④ 

→ For the above string we got 2 LMD & RMD and 2 parse trees so, the given grammar is ambiguous

→ Top down parser can't handle ambiguous grammars, so we need to convert this grammar into unambiguous grammar.

*While converting ambiguous to unambiguous:=

→ The grammar should follow Associativity, and precedence rules

*, /, +, − → left Associatin (when an operand has operand on both sides, the operand should associate with left side operator)

/, * +, − → precedence order
1 2 3 4

② WRITING A GRAMMAR:=

→ Grammar is used to describe the syntax of programming language.

lexical versus syntactic Analysis:=

- seperating syntactic structure into smaller and manageable components of lexical and non-lexical parts.
- lexical rules are simple to understand than grammar.
- We need notations to describe the grammars.
- RE are used to Realtime Examples to make them easy to understand.
- Grammars are used for describing nested structure such as balanced parenthesis, matching begin-end, if-then-else.

Eliminating Ambiguity:=

Properties of CFG

1. Ambiguous
2. Left Recursive
3. Left Factoring.

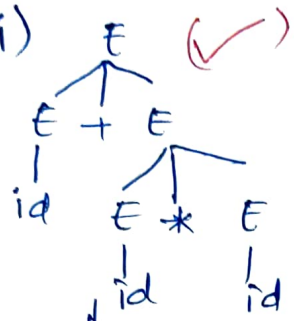
Elimination of ambiguity:-

Consider CFG:-

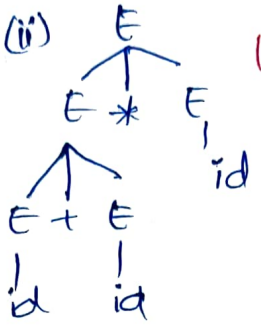
Eg:- $E \rightarrow E+E \mid E*E \mid (E) \mid id$

$w = id+id*id$ (There are 2 parse trees for string)

(i) (✓)



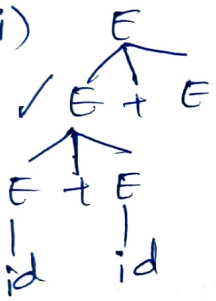
(ii) (X)



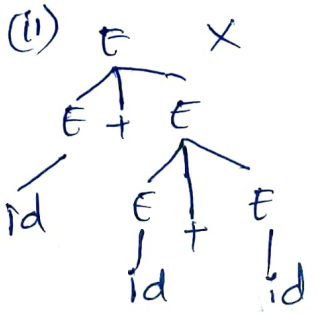
↳ This is the valid parse tree, because the "*" operator is appeared at bottom of parse, so it is evaluated first.

Eg2:- $w = id+id+id$

(i)



(ii)



→ Two factors that are needed to be taken

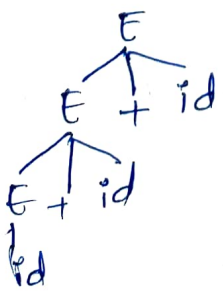
(1) precedence

(2) Left Associativity → To ensure left associativity, we need to convert the grammar into left recursive

→ left recursive $E \rightarrow E+E$ (the leftmost symbol of RHS is equal to the LHS)

- In left recursive, the parse tree can be grown on left side only

$E \rightarrow E+id \mid id$ $\quad id+id+id \Rightarrow (id+id)+id$ (Left Associative)



left associativity can be achieved

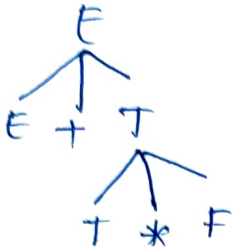
$(id+id)+id$

↓

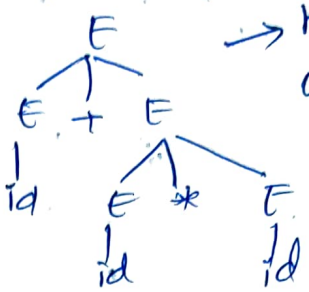
This expression is evaluated first, so that it is valid parse tree.

$E \rightarrow E+T$ (precedence)

$T \rightarrow T*F$



id+(id*id)



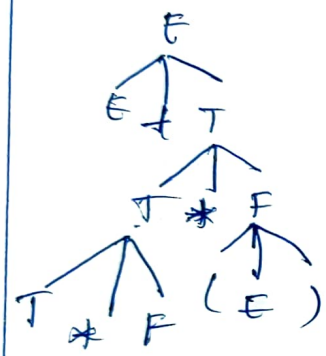
→ highest precedence operation is executing first.

The Final unambiguous Grammar

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$



Dangling else:- In a program if there are more than one if-stmts then else part is matched with wrong if stmt, that will lead to wrong results, that is called dangling else.

Eg: if (expr1) P_1 → T

if (expr2) → T

S_1

P_2 ← else

S_2

other stmt

(i) if P_1 is True Then it will goto P_2 and

- check the condi and condition, if it also true then it will execute S_1
- if and condition is false it will execute else part i.e S_2

(ii) if P_1 → is false it has to execute the other statement

(iii) But here if P_1 - False then it is executing else part i.e statem2 it is matching with wrong else.

- Consides the following dangling else grammas.

stmt → if Expr then stmt

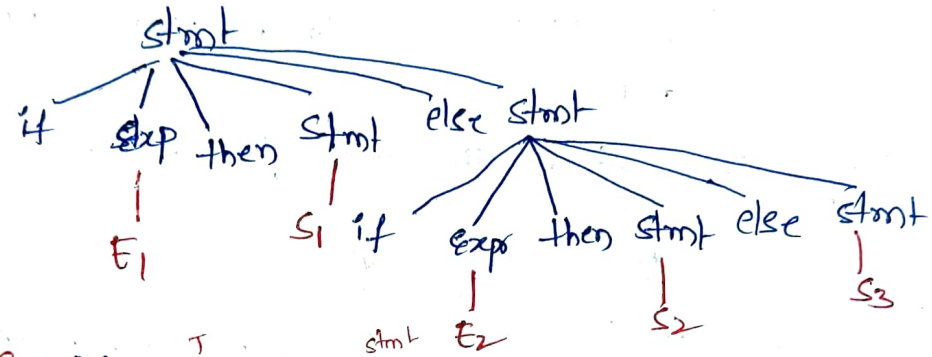
| if expr then stmt else stmt

| other

- The compound conditional stmt for the above grammas is

if E_1 then S_1 [else if E_2 then S_2 else S_3]

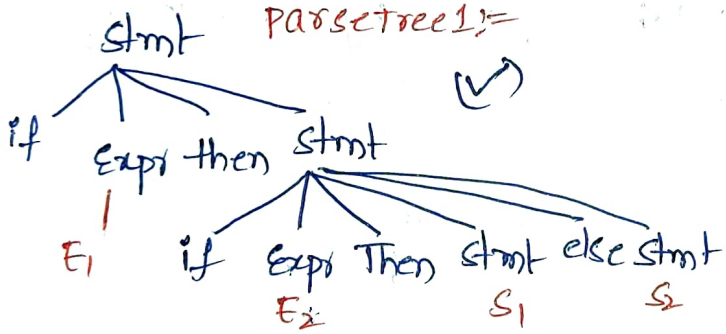
- The parse for this stmt



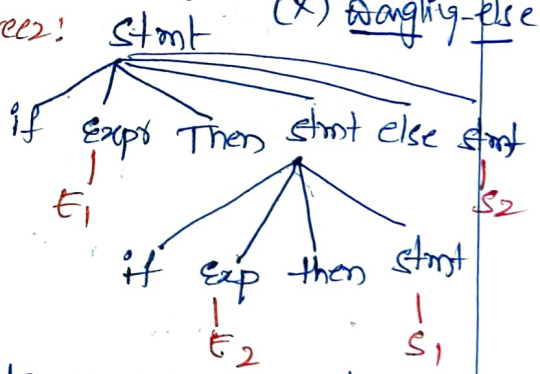
Eg i/p string = if E_1 then [if E_2 then S_1 else S_2]

there exist two parse trees for the given string.

parsetree1:= (✓)



parsetree2: (X) dangling-else



→ In these two parse trees, in programming language, the 1st parse tree is considered because, match the else statement with closest then. open stmt: if E_1 then [if E_2 then S_1 else S_2] → matched stmt

→ so, The given Grammas is ambiguous Grammas.

The unambiguous grammar will be.

stmt → matched stmt | open stmt

matched stmt → if Expr then matched stmt else matched stmt | other

open stmt → if Expr then stmt

| if expr then matched stmt else open stmt.

matched stmt → perfect if stmt

open stmt → simple if

(ii) Elimination of left Recursion:-

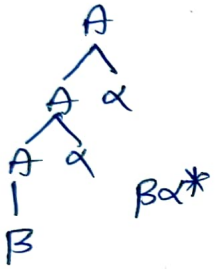
→ A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.

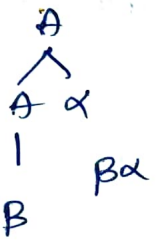
→ Top-down parsing methods cannot handle this grammar, so we need to eliminate left recursion.

$A \overset{+}{\Rightarrow} A\alpha / \beta$ (left recursive) $\alpha, \beta \in (V \cup T)^*$

[if LHS is equal to the leftmost Non-terminal of RHS]

The string derived from the above grammar.





string(w) = $\beta\alpha^*$
(any production should not contain * symbol)

Production $A \rightarrow A\alpha/\beta$ could be replaced by the following non-left-recursive productions.

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

Eg:- $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$ (left recursive grammar)

So, we need to eliminate left-recursion from the above grammar

(i) $E \rightarrow E+T/T$

A A α β

$A = E$

$\alpha = +T$

$\beta = T$

$E \rightarrow TE'$

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

(ii) $T \to T*F \mid F$
$A \quad A\alpha \mid \beta$

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

(iii) $F \to (E) \mid id$ (Non left recursive)

After eliminating the left recursion from the grammar

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id \mid \varepsilon$

* Elimination of left recursion for multiple production.

$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid \beta_1 \mid \beta_2 \cdots$

$A \to \beta_1 A' \mid \beta_2 A'$
$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon \cdots$

Eg1:- $Expr \to Expr+Expr \mid Expr*Expr \mid id$
$A \quad A \quad \alpha_1 \quad A \quad \alpha_2 \quad \beta_1$

$Expr \to idExpr'$
$Expr' \to +ExprExpr' \mid *exprexpr' \mid \varepsilon$

Eg2:- $S \to Sx \mid Ssb \mid xs \mid a$
$A \quad A\alpha_1 \; A\alpha_2 \quad \beta_1 \quad \beta_2$

$S \to xSs' \mid as'$
$S' \to xs' \mid sbs' \mid \varepsilon$

Eg3:- $S \to Aa \mid b$

$A \to Ac \mid Sd \mid f$
$S \to Aa \mid b \to$ NO left recursion
$A \to Ac \mid Sd \mid f$

(1) Elimination of left factoring:-

→ A grammar contains production rule in the form

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \gamma_1 \mid \gamma_2$$

↓

Then that grammar contains left factoring.

→ TDP parsers can't handle ~~left factoring~~ the grammar contains left-factoring.

→ we can eliminate the left-factoring by replacing with the following production.

$$A \to \alpha A' \mid \gamma_1 \mid \gamma_2$$
$$A' \to \beta_1 \mid \beta_2$$

Eg1:- $S \to iEts \mid iEtses \mid a$ (it contains left factoring)
$\alpha \quad\quad \alpha \quad \beta_2 \quad \gamma_1$

$E \to b$

Here, A=S, α=iEts β_1=ε β_2=es γ_1=a

$S \to iEtsS' \mid a$

$S' \to \varepsilon \mid es$

Eg2:- $A \to aAB \mid aA \mid a$
$\alpha \quad \beta_1 \quad \alpha \quad \alpha$

$B \to bB \mid b$

$A \to aAB \mid aA \mid a$
$\alpha \quad \beta_1 \quad \alpha \quad \beta_2 \quad \alpha \quad \beta_3$

$A \to aA'$

$A' \to AB \mid A \mid \varepsilon$

Eg3:- $B \to bB \mid b$
$A \quad \alpha \quad \beta_1 \quad \beta_2$

$B \to bB'$

$B' \to B \mid \varepsilon$

Eg4:- $X \to X+X \mid X*X \mid D$
$A \quad \alpha \quad \beta_1 \quad \alpha \quad \beta_2 \quad \gamma_1$

$X \to XX' \mid D$

$X' \to +X \mid *X$

Eg5:- $E \to T+E \mid T$

$T \to int \mid int*T \mid (E)$

$E \to T+E \mid T$
$A \quad \alpha \quad \beta_1 \quad \alpha \quad \beta_2 = \varepsilon$

$E \to TE'$

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

$T \to \underset{\alpha}{int} \mid \underset{\alpha}{int} \underset{\beta_1}{*} \underset{\beta_2}{T} \mid \underset{\gamma_1}{(E)}$

$T \to intT' \mid (E)$

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

Top-Down parsing :-

→ In Top-down parsing, The parse tree is constructed from root node to child node.

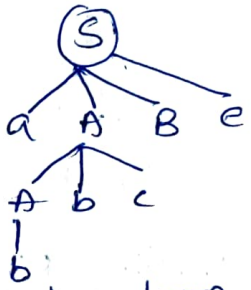
→ Top-down parser uses left-most derivation to derive the input string from the grammar.

→ TDP is starting from start symbol of grammar and reaching the i/p string

Eg :- Grammar is, derive the i/p string w=abbcde.

$S \to aABe$
$A \to Abc/b$
$B \to d$

LMD :- $S \to aABe$
$\to aAbcBe$
$\to abbcBe$
$\to abbcde$



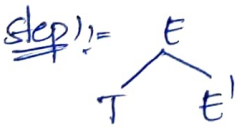
→ TDP constructed for the grammar if it is free from
- ambiguity
- left recursion.

→ The problem with TDP parser is if we have more than one alternative for production, which alternative we should use. $\{E \to E+T/T, T \to T*F/F, F \to (E)/id.\}$ ↳ ambiguous, left recursion

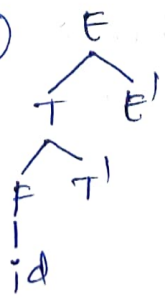
Eg :- $E \to TE'$ → Unambiguous grammar
$E' \to +TE'$
$T \to FT'$
$T' \to *FT'/\varepsilon$
$F \to (E)/id$

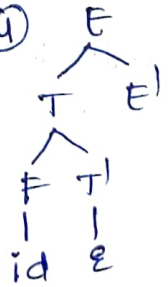
↳ Derive an i/p string W = id+id*id

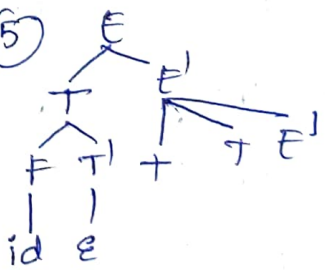
Step 1 :-

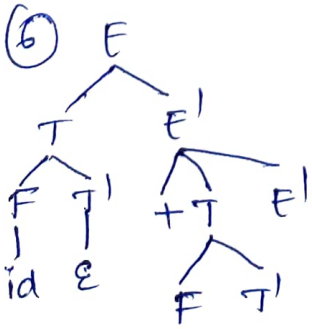


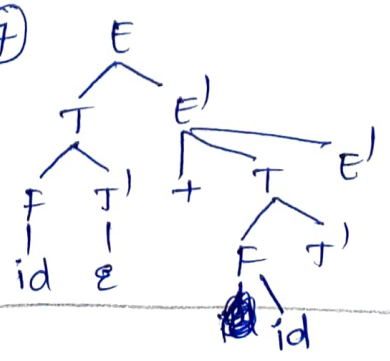
② 

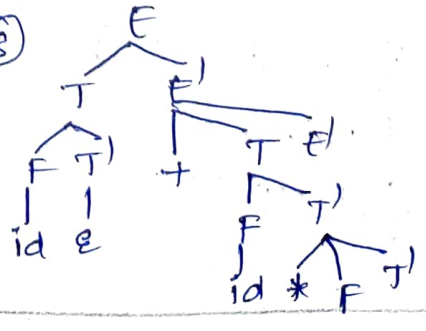
③ 

④ 

⑤ 

⑥ 

⑦ 

⑧ 

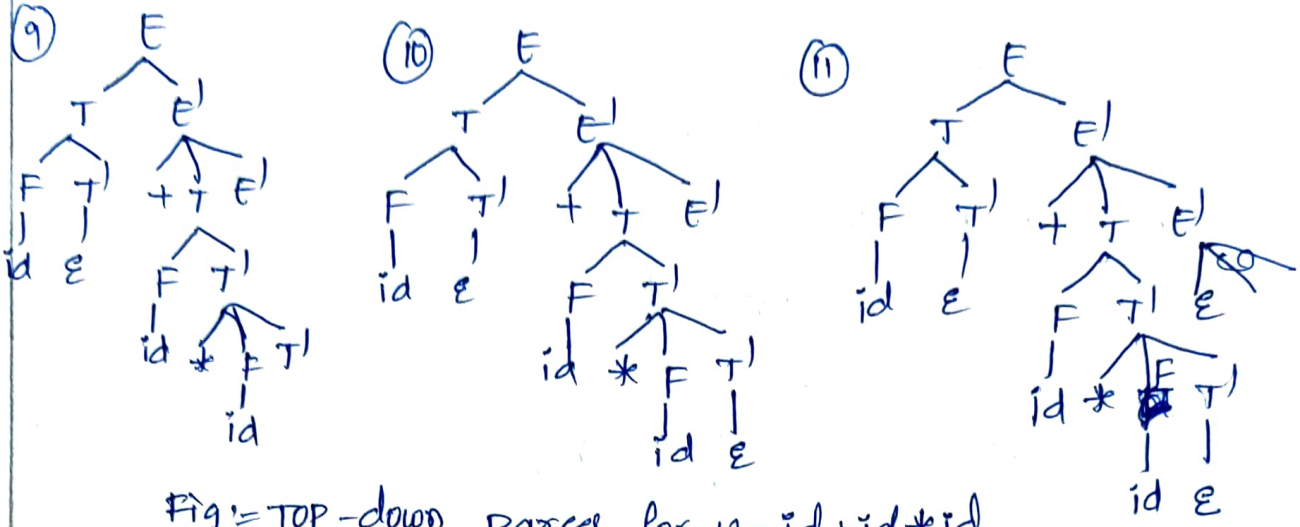


Fig:- TOP-down parses for w = id+id*id

(i) Recursive-Descent parsing:- (with back tracking)

→ The construction of parse tree starts from root & proceed to childnode

→ Recursion:- A function which is called by itself.

steps for constructing Recursive Descent parser:-

(i) If the i/p is a non-terminal, then call corresponding function

(ii) If the i/p is terminal, then compare terminal with i/p symbol, if both are same, the increment input pointer

(iii) If a Non-terminal contains more than one production then all the production code should be written in the corresponding function.

Eg 1:- $E \rightarrow iE' \quad E' \rightarrow +iE' \mid \varepsilon$

(In the given grammar we have 2-Nonterminals E, E', so define two functions, same as C-lang functions)

```
E()
{
 if (input == 'i')
 input++;
 EPRIME();
}
```

```
EPRIME();
{
   if (input == '+')
   {
     if (inp input++;
   if (input == 'i')
    input++;
    EPRIM();
    }
  elee
  return;
}
```

i/p string: i+i$

i/p string

Eg2:- Grammar $E \to TE'$ $E' \to +TE' / \varepsilon$
$T \to FT'$ $T' \to *FT' / \varepsilon$ $F \to (E) / id$

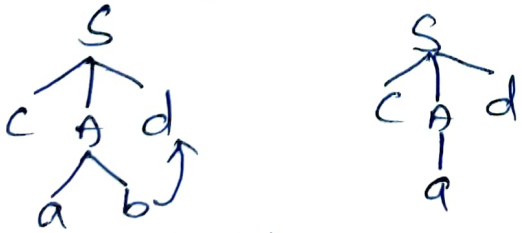
```
E()
{
 T();
 EPRIME();
}
EPRIME();
{
 if (input == '+')
 {input++;
  T();
  EPRIME();
 }else
  return;
}
```

```
T()
{
  F();
  TPRIME();
}
TPRIME()
{
  if (input == '*')
  {
    input++
    F();
    TPRIME();
  }
  else
    return;
}
F()
{
 if (input == '(')
 {input++;
   E();
 if (input == ')')
    input++;
 }
else if (input == 'id')
  input++;
}
```

Eg3:- $S \to CAd$

$A \to ab / a$

w = i/p string w = cad



Backtracking

(If None of the product for is satisfying the condition then we should check on another possibility for "S")

```
input++
EPRIME();
}
EPRIME()
{
 if (input == '+')
  { input++;
   if (input=='i')
    input++;
   EPRIME();
  }else
   return;
 }
```

i/p: i+i$

i/p = id+id

Eg: 2

E → TE'

E' → +TE' / ε

T → FT'

T' → *FT' / ε

F → (E) / id

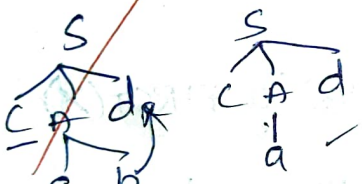
```
E()
{
 T()
 EPRIME();
}
EPIRIME()
{
 if(input=='+')
  input++
```

```
T()
EPRIME();
}
else
return;
}
T()
{
 F()
 TPRIMEC();
}
TPRIME()
{
 if(input=='*')
 {
  input++;
  F()
  TPRIME();
 }
 else
  return;
}
```

```
F()
}
if(input=='(')
{ input++;
  E();
if (input==')')
 input++;
}
else if
(input=='id')
 input++;
}
```

Eg S → cAd

A → ab / a w = cad



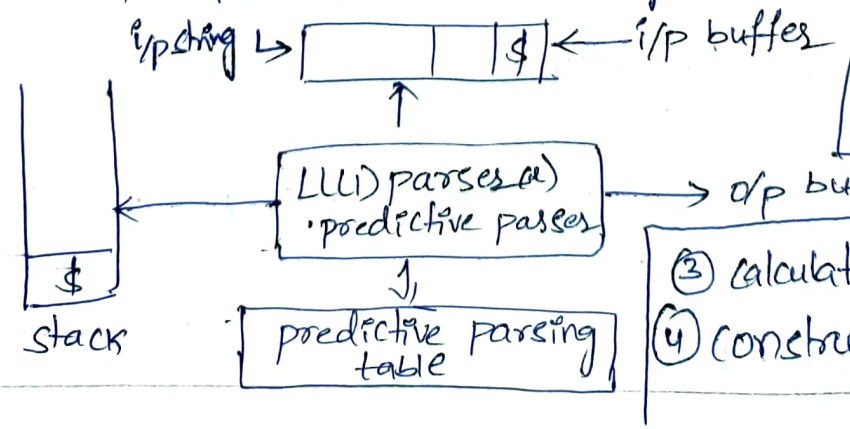
Backtrack

(∵ if Non of the product for A is satisfying the condition, then we should check another possibility for S)

(ii) predictive parsing: (or) LL(1) & Non recursive parser:-

LL(1) - means

- It scans the input from left to Right by using leftmost derivation.
- LL(1) → 1 — look ahead — at a time we are reading & checking at a time



Steps for constructing LL(1) parser:-

1. Elimination of left recursion
2. Elimination of left factoring
3. Calculation of FIRST() & FOLLOW()
4. Construction of parsing table.

LL(1) parser → parsing algorithm

LL(1) parsing table – Data structure which is constructed from the LL(1) grammar

Stack := Data structure used to store the grammar symbols

- Always the bottom of the stack is $

→ The end of i/p buffer is $

→ after $, the starting symbol of i/p is pushed to the stack.

To construct the parsing table we should compute two functions. ① FIRST() ② FOLLOW()

FIRST(X) :=

- if X is a terminal then FIRST(X) = {X}
- if X is a Non terminal & $X \to Y_1 Y_2 Y_3 \cdots Y_n$ is a production for X.

FIRST(X) = FIRST(Y_1) &

if FIRST(Y_1) contains ε add FIRST(Y_2) to FIRST(X)

if all FIRST($Y_1 Y_2 \cdots Y_n$) contains ε then add ε to FIRST(X)

FOLLOW(B) :=

→ Always the follow() of starting symbol is $.

→ $A \to \alpha B \beta$

FOLLOW(B) = FIRST(β) &

if FIRST(β) contain ε or $A \to \alpha B$ then add Follow(A) to Follow(B)

Eg :- Given Grammar is

$E \to E+T / T$ – left recursion

$T \to T*F / F$ – left recursion

$F \to (F) / id$

} Eliminat left recursion & left factoring from these 2 productions

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

④ FIRST (E) = FIRST (T) = FIRST (F) = { (, id)

- FIRST (E') = { +, ε }
- FIRST (T) = FIRST (F) = { (, id }
- FIRST (T') = { *, ε }
- FIRST (F) = { (, id }

→ FOLLOW (E) = ~~{), $ }~~, ~~Follow~~

$F \rightarrow (E) / id$

↓

follow(E) = { $, FIRST ( ) } = { $,) } ?

⑤ FOLLOW (E') = { Follow (E) + Follow (E') }

$E \rightarrow TE'$ = { $,) }

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

Follow (T) = ~~E → TE' Follow(E')~~

$E \rightarrow TE'$ Follow(T) = FIRST (E') ∉ Follow (E)

$E' \rightarrow +TE' / \varepsilon$ = { +, $,) }

Follow (T') ⇒ [Follow (T) + Follow (T')]

~~T → FT'~~ ⇒ { +, $,) }

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

Follow (F) = Follow (T') + FIRST (T) + Follow (T)

$T \rightarrow FT'$ = { *, +, $,) }

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

(LL(1))

predictive parsing table :-

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

$W = id+id$

Stack	Input	Action
$E\$$	$id+id\$$	$E \to TE'$
$TE'\$$	$id+id\$$	$T \to FT'$
$FT'E'\$$	$id+id\$$	$F \to id$
$idT'E'\$$	$id+id\$$	
$T'E'\$$	$+id\$$	$T' \to \varepsilon$
$\varepsilon E'\$$	$+id\$$	
$E'\$$	$+id\$$	$E' \to +TE'$
$+TE'\$$	$+id\$$	
$TE'\$$	$id\$$	$T \to FT'$
$FT'E'\$$	$id\$$	$F \to id$
$idT'E'\$$	$id\$$	$T' \to \varepsilon$
$\varepsilon E'\$$	$\$$	$E' \to \varepsilon$
$\$$	$\$$	accepted

The i/p is parsed

Eg1:- $S \to (L) | a$

$L \to L,S | S$

$W = (a)$

Eg2:- $S \to aAB | bA | \varepsilon$

$A \to aAb | \varepsilon$

$B \to bB | \varepsilon$

$W = 'aabb'$

$S \to iEtS | iEtSeS | a$

$E \to b$ (Non-LL(1))

Eg4: $S \to AaB | cbB | Ba$

$A \to da | BC$

$B \to g | \varepsilon$

$C \to h | \varepsilon$

Bottom-up parser :-

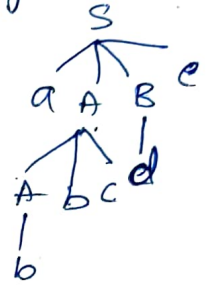
→ Bottom-up parser construct parse tree from child (bottom) node to root node (top).

→ Bottom up parser use RMD in Reverse order.

→ In Bottom-up parser the process is reducing i/p string w to start symbol of grammar

Eg :- S → aABe
A → Abc / b
B → d.

Input string w = abbcde



aAbcde (A → b)
aAde (A → Abc)
aABe (B → d)
S (S → aABe)

S → aABe
→ aAde
→ aAbcde
→ abbcde

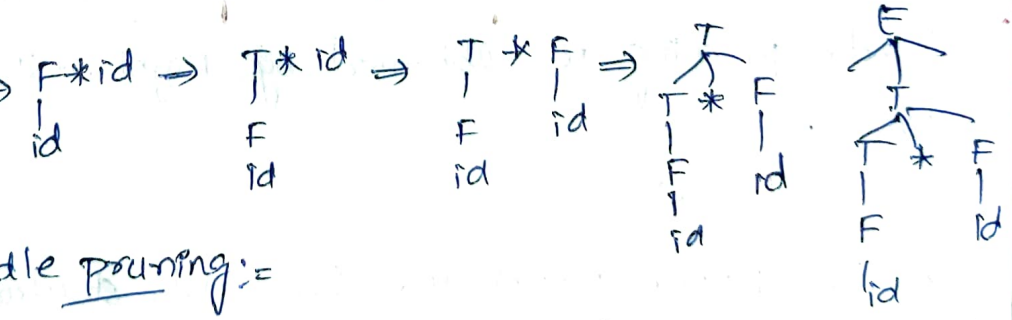
Eg 2 :- E → E+T / T
E → T*F / F
F → (E) / id

w = id*id

id*id (F → id)
F*id (T → id)
T*id (E → id)
T*F (F → T*F)
T (E → T)
E

→ The problem with Bottom-up parser is when to reduce the or to production.

Step : id*id ⇒ F*id ⇒ T*id ⇒ T*F ⇒ T



Handle & Handle pruning :-

Handle : Handle is a substring, which matches with right side of production

→ if Handle matches with the right side of production, then it is replaced with left hand side Non-terminal

Eg1:- $S \to aABe$ $\quad w = abbcde$
$A \to Abc / b$
$B \to d$

Right sentential form	Handle	Reducing production
abbcde	b	$A \to b$
aAbcde	Abc	$A \to Abc$
aAde	d	$B \to d$
aABe	aABe	$S \to aABe$
S		

Handles during parse of $id_1 * id_2$

Eg2:- $E \to E+T/T$ $\quad w = id_1 * id_2$
$T \to T*F/F$
$F \to (E)/id$

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

→ Handle pruning is obtained by Right most derivation in reverse order.

(1) Shift- Reduce parser :=

→ shift - Reduce parser use two data structures ① stack ② i/p buffer

① stack – is used to store the grammar symbol

② Input buffer – holds the i/p string to be parsed

Stack	Input string
$	w$

Actions of shift-Reduce parser:

1. shift — shift the next I/p symbol onto the top of the stack.
2. Reduce — If the top of stack is matched with Right side of production, then it is reduced to corresponding non-terminal
3. Accept — successful completion of parsing
4. Error — Discovers a syntax error and call error recovery methods

Eg :- E → E+T/T
T → T*F/F
F → (E) / id

w = id * id

Fig:- configuration of shift-Reduce parser on i/p id*id

Stack	Input Buffer	Action
$	id*id$	shift
$id	*id$	Reduce by F→id
$F	*id$	Reduce T→F
$T	*id$	shift
$T*	id$	shift
$T*id	$	Reduce F→id
$T*F	$	Reduce T→T*F
$T	$	Reduce E→T
$E	$	Accepted

conflicts during shift-Reduce parsing

① shift-Reduce conflict : cannot decide whether to shift or to reduce

② reduce-reduce conflict : if the two productions have same handle (substring).

Shift - Reduce

Eg2 :- S → (L) / a

L → L,S / S parse the string (a, (a,a)) using SR-Parser

Stack	I/P Buffer	parsing Action
$	(a, (a,a))$	shift
$(	a, (a,a))$	shift
$(a	, (a,a))$	Reduce S → a
$(S	, (a,a))$	L → S Reduce
$(L	, (a,a))$	shift
$(L,	(a,a))$	shift
$(L, (	a,a))$	shift
$(L, (a	,a))$	Reduce S → a
$(L, (S	,a))$	Reduce L → S
$(L, (L	,a))$	shift
$(L, (L,	a))$	shift
$(L, (L, a	))$	Reduce S → a
$(L, (L, S	))$	Reduce L → L,S
$(L, (L	))$	shift
$(L, (L)	)$	Reduce S → (L)
$(L, S	)$	Reduce L → L,S
$(L	)$	shift
$(L)	$	Reduce S → (L)
$S	$	Accept