

An illustration featuring a central yellow circle with the text "KTUNOTES" in a black, hand-drawn font. The background is a solid blue color. Surrounding the central circle are several hands holding books. In the top left, a hand holds an open book with text. In the top right, a hand holds a closed yellow book. In the bottom left, a hand holds a red book. In the bottom right, a hand holds an open book. In the center bottom, two hands hold a yellow book. To the left of the central circle, there is a stylized graphic of a book with pages fanning out. To the right, there is a stack of books. The overall theme is education and learning.

KTUNOTES

WWW.KTUNOTES.IN

MODULE-2.

CFG

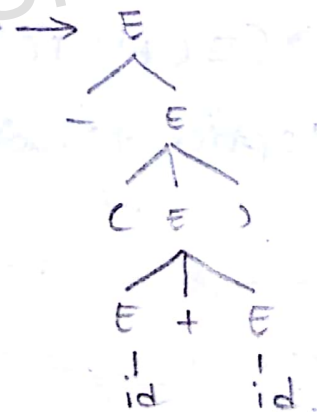
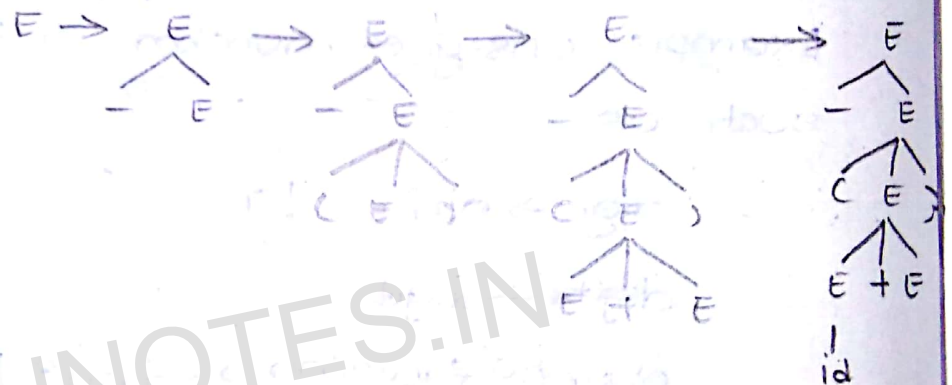
$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid id.$$

Derive the tree for $-(id+id)$

Use left most derivation

$$\exists \rightarrow \neg \exists$$
$$\rightarrow -(E)$$
$$\rightarrow -(E+E)$$
$$\rightarrow -(id + E)$$

→ - (cid + id)



Many programming language constructs have an inherently recursive structure (syntax) that can be defined by CFG

CFG consists of terminals, non-terminals, start symbol

production.

DERIVATION

A production is treated as rewriting rule in which non-terminal on left is replaced by the string on the right side of the production.

PARSE TREE

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding the replacement order.

AMBIGUITY

id + id * id.

LMD.

$$E \rightarrow E + E$$

$$\rightarrow id + E$$

$$\rightarrow id + E * E$$

$$\rightarrow id + id * E$$

$$\rightarrow id + id * id$$

$$E \rightarrow E * E$$

$$\rightarrow E + E * E$$

$$\rightarrow id + E * E$$

$$\rightarrow id + id * E$$

$$\rightarrow id + id * id$$

A grammar that produces more than one parse tree for the same string is said to be ambiguous i.e., an ambiguous grammar is one that produces more than one left most or more than one right most for the same string.

LEFT FACTORING

: is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

Eg: Consider the CFG

$\text{Stmt} \rightarrow \text{if Expr then Stmt}$

$\text{Stmt} \rightarrow \text{if Expr then Stmt Else Stmt}$

$\text{Stmt} \rightarrow a$

$\text{Expr} \rightarrow b$

Consider general form of grammar

$A \rightarrow \alpha B_1 \mid \alpha B_2$

It can be left factored as

$A \rightarrow \alpha A'$

$A' \rightarrow B_1 \mid B_2$

$S \rightarrow iEtS$

$S \rightarrow iEtSES$

$S \rightarrow a$

$E \rightarrow b$

After left factoring

$S \rightarrow iEtSS'$

$S' \rightarrow \epsilon \mid eS$

$S \rightarrow a$

$E \rightarrow b$

If there are more productions in a CFG which is non-deterministic we can write general form as given below.

$$A \rightarrow \alpha B_1 \mid \alpha B_2 \mid \dots \mid \alpha B_n \mid \gamma$$

where γ represents an alternative that does not begin with α .

$$A \rightarrow \alpha A'$$

$$A' \rightarrow B_1 \mid B_2 \mid \dots \mid B_n$$

$$A \rightarrow \gamma$$

It can be written as

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow B_1 \mid B_2 \mid \dots \mid B_n$$

ELIMINATION OF LEFT FACTORING RECURSION

A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α .

Top-down parsing method cannot handle left recursive grammar so a transformation that eliminates left recursion is needed.

The production of the form $A \rightarrow A\alpha \mid \beta$ can be replaced by the non-left recursive productions as:

$$A \rightarrow \beta A'$$

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

Consider the grammar

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

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

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

$$\begin{aligned} A \rightarrow A\alpha \mid B \\ E \rightarrow E + T \mid T \Rightarrow \end{aligned} \quad \begin{aligned} A \rightarrow \beta A' \\ E \rightarrow T E' \\ A' \rightarrow \alpha A' \mid \epsilon \\ E' \rightarrow T E' \mid \epsilon \end{aligned}$$

$$\begin{aligned} T \rightarrow T * F \mid F \Rightarrow T \rightarrow F T' \\ T' \rightarrow * F T' \mid \epsilon \end{aligned}$$

$$F \rightarrow (E) \mid id \Rightarrow F \rightarrow (E) \mid id$$

More generally it can be written as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid B_1 \mid B_2 \mid \dots \mid B_n$$

The left recursion can be eliminated as:

$$A \rightarrow B_1 A' \mid B_2 A' \mid \dots \mid B_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

FIRST AND FOLLOW

		FIRST	FOLLOW
$S \rightarrow ABCD$	S	{a, b, c}	{ \$ }
$A \rightarrow a \mid \epsilon$	A	{a, ϵ }	{ b, c }
$B \rightarrow b \mid \epsilon$	B	{b, ϵ }	{ c }
$C \rightarrow c$	C	{c}	{ d, \$ }
$D \rightarrow d \mid \epsilon$	D	{d, ϵ }	{ \$ }

2.2.18

Q Write first and follow of all non-terminals of given CFG.

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

TOP DOWN PARSING

Parsers

Top Down

Bottom Up

Top Down parsing
with backtracking

Topdown parsing
without backtracking

Brute Force

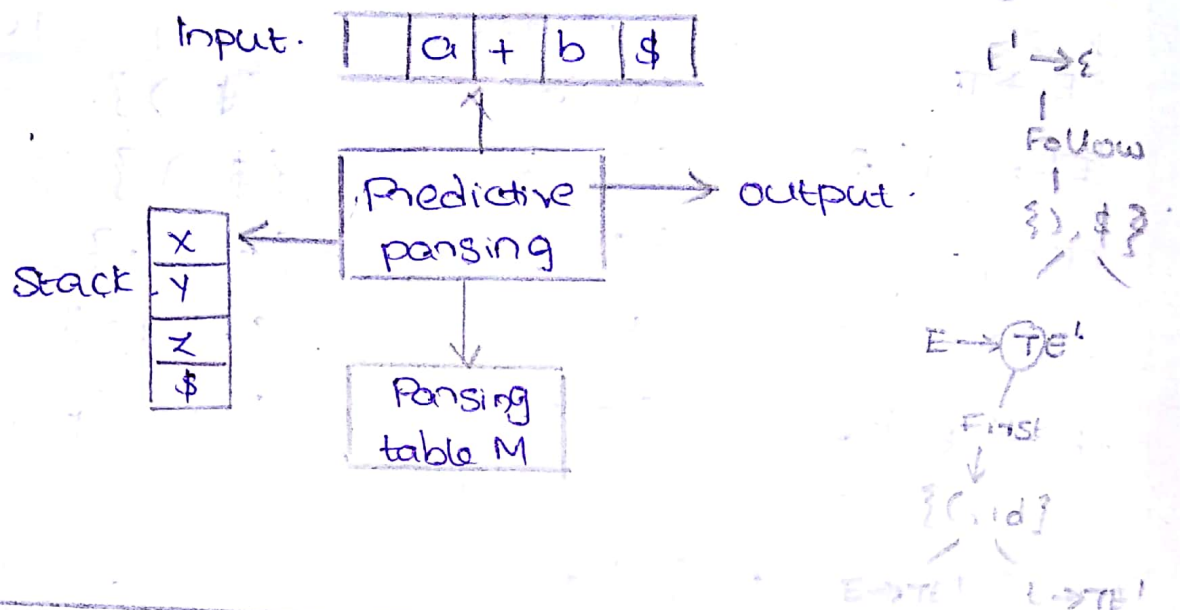
Recursive
descent
method

Non-recursive
descent method
(LLC)

PREDICTIVE

A special form of recursive descent parsing in which lookahead symbol unambiguously descends the procedure selected for each non-terminal.

MODEL OF NON-RECURSIVE PREDICTIVE PARSING



Non-terminal	input stack					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

$E \rightarrow TE'$
 $E' \rightarrow TE' | \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow KFT' | \epsilon$
 $F \rightarrow (E) | id$

The predictive parsing can be done on the input `id + id * id` as follows

Stack	input	output
$\$E$	\downarrow <code>id + id * id</code>	—
$\$E'T$	\downarrow <code>id + id * id</code>	$E \rightarrow TE'$
$\$E'T'F$	\downarrow <code>id + id * id</code>	$T \rightarrow FT'$

\$E'T'id	↓ id+id*id	F → id
\$E'T'	↓ +id*id	—
\$E'	↓ +id*id	T' → ε
\$E'T+	+id*id	E' → +TE'
\$E'T	id*id	—
\$E'T'F	↓ id*id	T → T*FT'
\$E'T'id	↓ id*id	F → id.
\$E'T'	*id	—
\$E'T'F*	↓ *id	T' → *FT'
\$E'T'F	id	—
\$E'T'id	id	F → id
\$E'T'	\$	—
\$E'	\$	T' → ε
\$	\$	E' → ε

Q.8.18

NON-RECURSIVE PREDICTIVE PARSING

It is possible to build a non-recursive predictive parser by maintaining a stack explicitly rather than implicitly via recursive calls. The main problem during predictive parsing is that of

determining a production to be applied for a non-terminal. The non-recursive parser looks up the production to be applied in a parsing table. A table driven predictive parser has an input

- (i) input buffer
- (ii) stack.
- (iii) parsing table.
- (iv) output stream.

LL(1): Every cell has single entry.

$$1. S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \epsilon$$

$$E \rightarrow b$$

Check whether given grammar is LL(1).

Step 1: Find first and follow of each non-terminal.

	First	Follow
$S \rightarrow iEtSS' a$	$\{i, a\}$	$\{\$, e\}$
$S' \rightarrow eS \epsilon$	$\{e, \epsilon\}$	$\{\$, e\}$
$E \rightarrow b$	$\{b\}$	$\{t\}$

Non-terminal

input stack.

	i	E	a	e	b	\$
S	$S \rightarrow iEtSS'$		$S \rightarrow iEtSS'$			
S'				$S' \rightarrow eS$		$S' \rightarrow eS \epsilon$
E					$E \rightarrow b$	