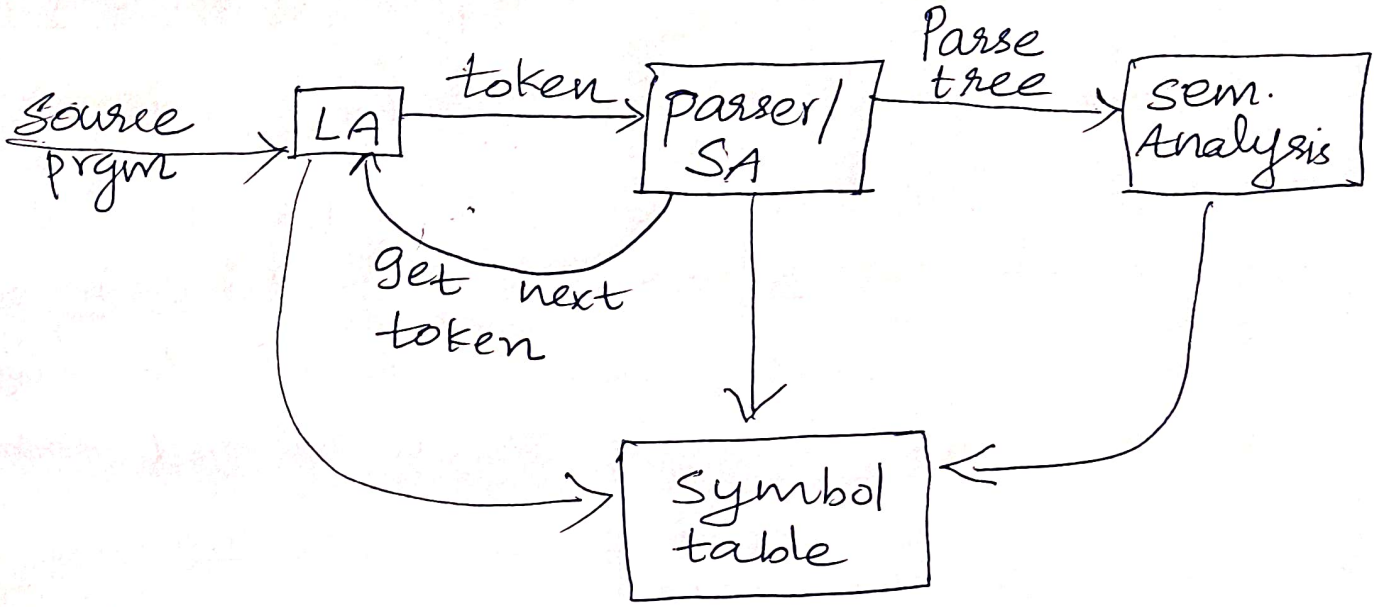
(2) Syntax Analysis / Parsing



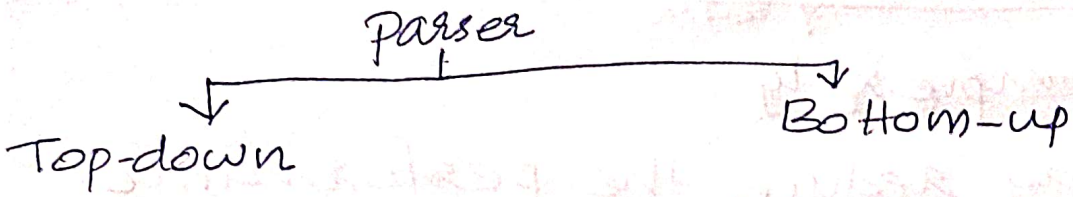
- It is the 2nd phase of compilation process.
- primary goal is to verify the syntactical correctness of the source code.

- i/p : tokens
  o/p : parsetree / Abstract syntaxtree.

- During this phase, the parser will check whether the input string is belong to the language generated by the Context Free Grammar (CFG)

If the syntax is correct, it will move forward; otherwise error.

- Two types:



~~Features of syntax Analysis:~~

- Features of syntax Analysis
  - → syntax trees
  - → CFG
  - → Top-down and Bottom-up
  - → Error-detection.
  - → Basic optimization.

- The PDA is used to design the parser.

- Advantages of SA
  - → Structural Validation
    (to check if the source code follows the grammatical rules of the prgming lang, which helps to detect and report errors. in the source code)
  - → Improved code generation
  - → Easier semantic analysis.

Disadvantages

→ Complexity

→ May reduce the performance.

→ Limited error recovery.

→ Inability to handle all languages.

Eliminating Ambiguity

Consider the grammar:

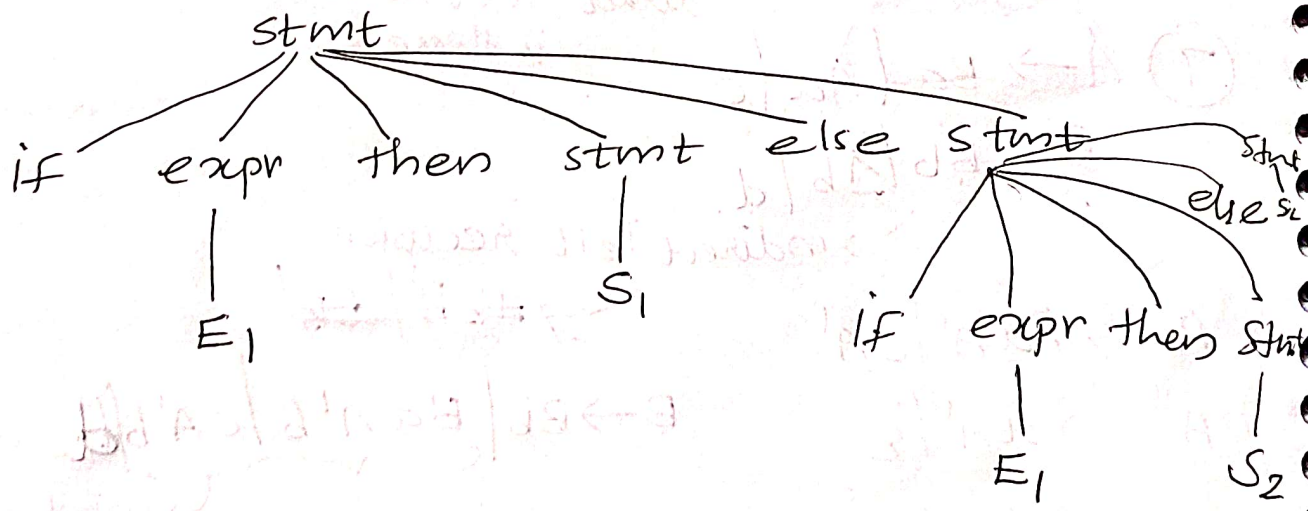
stmt → if expr then stmt /
if expr then stmt else stmt /
other.

(other → any other statement.)

Create parse tree for

if E_1 then S_1 else if E_2 then S_2 else S_3

[for expr → add E_1 and E_2
for stmt → add S_1 and S_2]

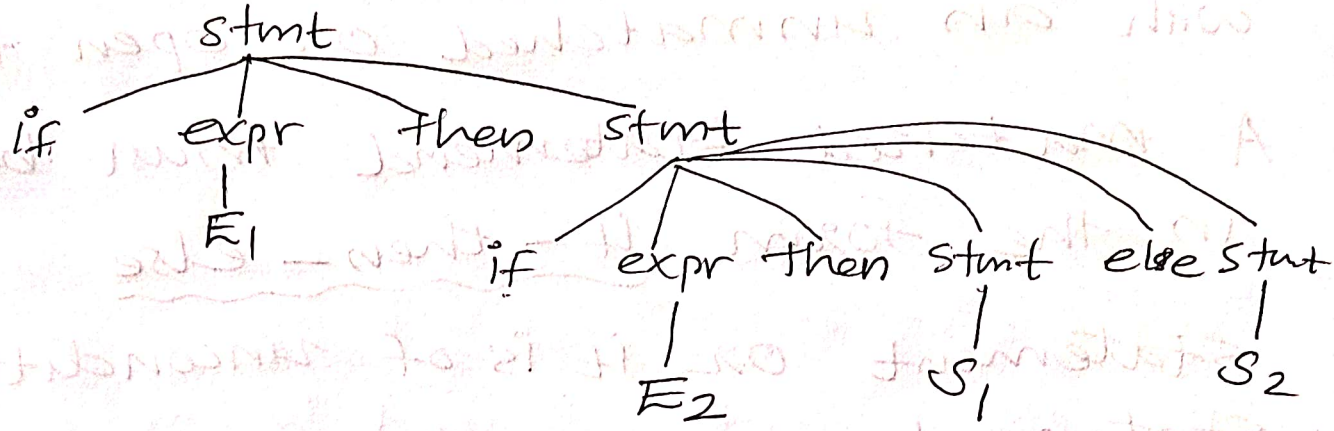


It is ambigous, because same string can be derive using two parse tree.

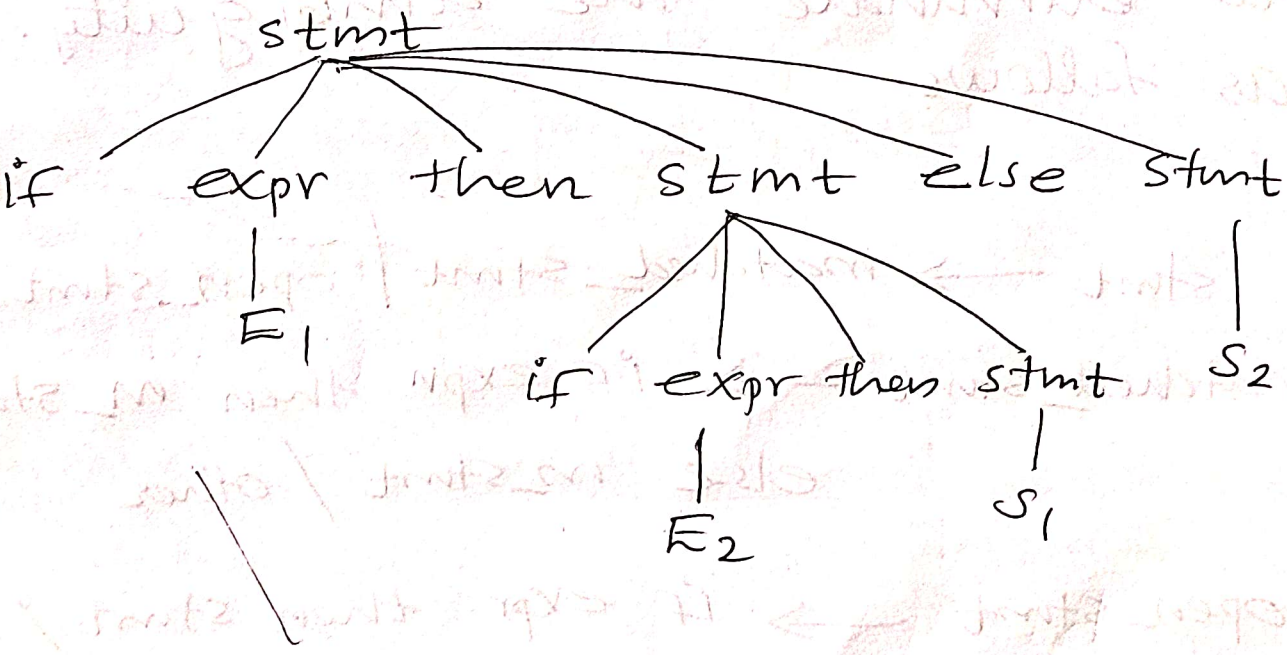
Eg: For the string

if E_1 then if E_2 then S_1 else S_2.

(i)



(ii)



So the above grammar is ambigous.

So the grammar can be call as 'dangling- else grammar'.

must be in the form:
if-then-else

ie, a statement appearing b/w a then and an else must be 'matched';
ie, interior statement must not end with an unmatched or open then.

A matched statemend must be in the form if-then-else statement or it is of unconditional statement.

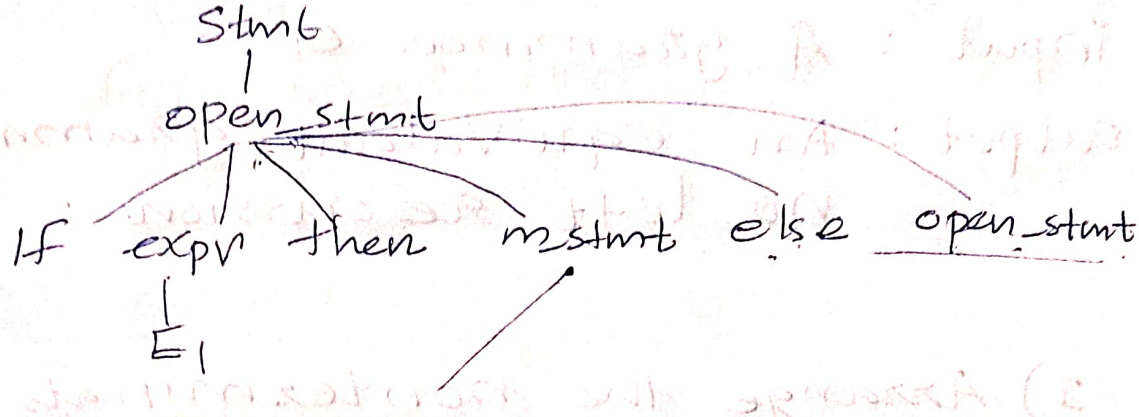
So we can rewrite the grammar to eliminate the ambiguity. as follows.

stmt → matched_stmt | open_stmt

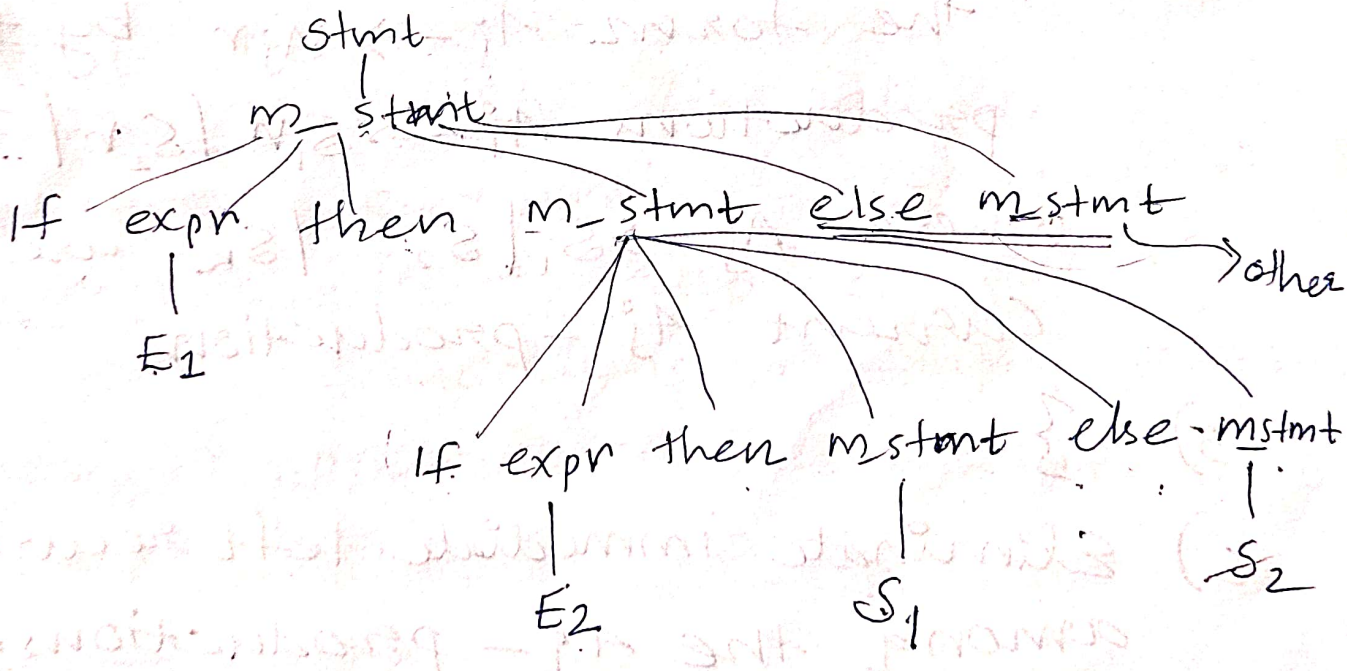
matched_stmt → if expr then m_stmt
else m_stmt / other

open_stmt → if expr then stmt /
if expr then m_stmt else
open_stmt.

The above is an unambigous grammar for if-then-else statements.



If E_1 then if E_2 then S_1 else S_2



Algorithm for eliminating left recursion:

input : A grammar G

output : An equivalent grammar with no left recursion

1) Arrange the nonterminals in some order $A_1, A_2 \ldots, A_n$.
2) for (each i from 1 to n) {
3) for (each j from 1 to i-1) {
4) replace each production of the form $A_i \rightarrow A_j r$ by the productions $A_i \rightarrow \delta_1 r \mid \delta_2 r \mid \ldots \delta_k r$, where $A_j \rightarrow \delta_1 \mid \delta_2 \ldots \mid \delta_k$ are all current A_j - productions.
5) }
6) Eliminate immediate left recursion among the A_q - productions.
7) }

Removal of Left Recursion

Recursion
- Normal : $S \rightarrow ASB$
- Left : $S \rightarrow Sa$
- Right : $S \rightarrow aS$

Removal:- If it is in the form $A \rightarrow A\alpha / \beta$

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

Examples

⇒ Direct left recursion

(1) $S \rightarrow Sa / b$

$S \rightarrow bS'$
$S' \rightarrow aS' / \varepsilon$

(2) $S \rightarrow Sa / d$

$S \rightarrow dS'$
$S' \rightarrow aS' / \varepsilon$

(3) $A \rightarrow Abcd / e$

$A \rightarrow eA'$
$A' \rightarrow bcdA' / \varepsilon$

(4) $A \rightarrow ABd / Aa / a$

$B \rightarrow Be / b$

$A \rightarrow aA'$

$A' \rightarrow BdA' / aA' / \varepsilon$

$B \rightarrow bB'$

$B' \rightarrow eB' / \varepsilon$

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

$T \rightarrow T*F / F$

$F \rightarrow id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id$

⇒ Indirect left recursion

(6)

$S \longrightarrow A\alpha / \beta$

$A \longrightarrow Sd$

Replace S with $A\alpha/\beta$

$S \longrightarrow A\alpha / \beta$

$A \longrightarrow A\alpha d / \beta d$

Now, remove left recursion of A as usual method.

$S \longrightarrow A\alpha / \beta$

$A \longrightarrow \beta d A'$

$A' \longrightarrow \alpha d A' / \varepsilon$

(7) $A \longrightarrow Ba / Aa / c$ → direct left recursion is there.

$B \longrightarrow Bb / Ab / d$ → indirect left recursion

$A \longrightarrow BaA' / cA'$

$A' \longrightarrow aA' / \varepsilon$

~~$B \longrightarrow AbB' / dB'$~~

$B \longrightarrow Bb / BaA'b / cA'b / d$ → prod. of A

$B \longrightarrow cA'bB' / dB'$

$B' \longrightarrow bB' / aA'bB' / \varepsilon$

II. Left Factoring

It is a grammar transformation that's useful for producing a grammar suitable for predictive, or top-down parsing.

Algorithm

Input: A grammar G.
Output: An equivalent left-factored grammar.

Method:

For each non-terminal A, find the longest prefix α common to two or more of its alternatives.
if $\alpha \neq \varepsilon$, ie, there's a non-trivial common prefix — replace all of the A— productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 \ldots \alpha\beta_n | \gamma$ where γ represents all alternatives that do not begin with α, by

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$$

Here, A' is a new non-terminal.

Repeat.

Normally, left factoring is applying for grammar with common prefixes.

Eg: $A \longrightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$

Here, α is common.

- These kind of grammar creates a problem to topdown parsers.
- The parser cannot decide which production must be choosen to parse the string in hand.

$$A \longrightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$$

Removal of left factoring can be done as;

$$A \longrightarrow \alpha A'$$
$$A' \longrightarrow \beta_1 | \beta_2 | \beta_3$$

Examples:

1) $X \longrightarrow abcx / abx$

$$X \longrightarrow abx'$$
$$x' \longrightarrow cx / x$$

2) $S \longrightarrow iEtS / iEtSeS / a$
$E \longrightarrow b$

$$S \longrightarrow iEtSS' / a$$
$$S' \longrightarrow eS / \epsilon$$
$$E \longrightarrow b$$

3) $A \longrightarrow aAB / aBC / aAC$

Ans:- $A \longrightarrow aA'$ ⟶ Step 1

$A' \longrightarrow AB / BC / AC$

$A \longrightarrow aA'$ ⟶ step 2

$A' \longrightarrow AA'' / BC$

$A'' \longrightarrow B / C$

4) $S \to aSSbS / aSaSb / abb / b$

Step1

$S \to aS' / b$

$S' \to SSbS / SaSb / bb$

Step2

$S \to aS' / b$

~~$S' \to SSbS$~~

$S' \to SS'' / bb$

$S'' \to SbS / aSb$

5) $S \to a / ab / abc / abcd$

Step1

$S \to aS'$

$S' \to b / bc / bcd / \varepsilon$

Step2

$S \to aS'$

$S' \to bS'' / \varepsilon$

$S'' \to c / cd / \varepsilon$

Step3

$S \to aS'$

$S' \to bS'' / \varepsilon$

~~$S'' \to c / cd / \varepsilon$~~

$S'' \to cS''' / \varepsilon$

$S''' \to d / \varepsilon$

Now, Can Stop!

Parse tree and Abstract Syntax tree

Consider the grammar,

exp ⟶ exp op exp / (exp) / number

op ⟶ + / − / *

The derivation for the string 'number + number' become

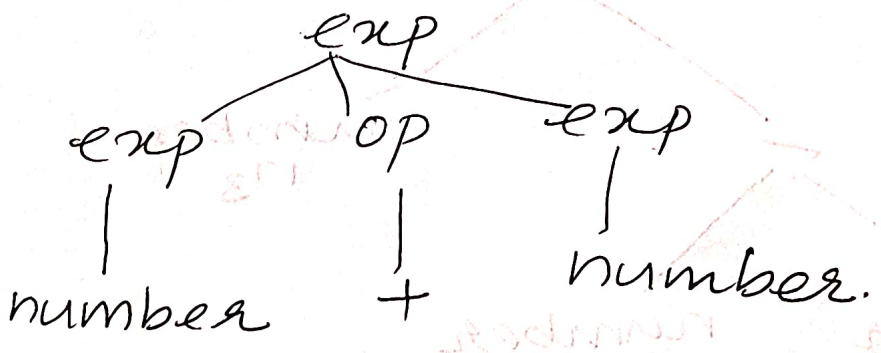
exp ⇒ exp op exp

⇒ number op exp

⇒ number + exp

⇒ number + number.

corresponding parse tree,

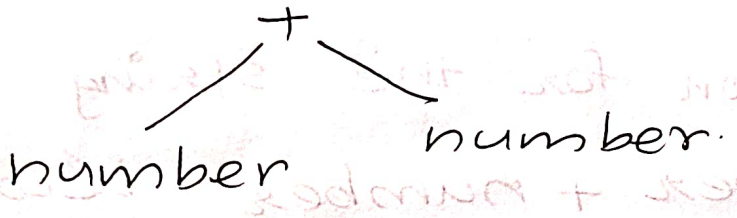


We can apply pre-order numbering (LMD) or post order numbering (RMD). for deriving the same string.

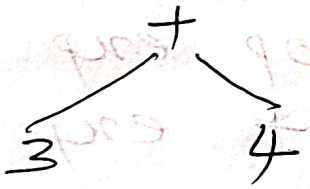
Abstract syntax tree

For number + number,

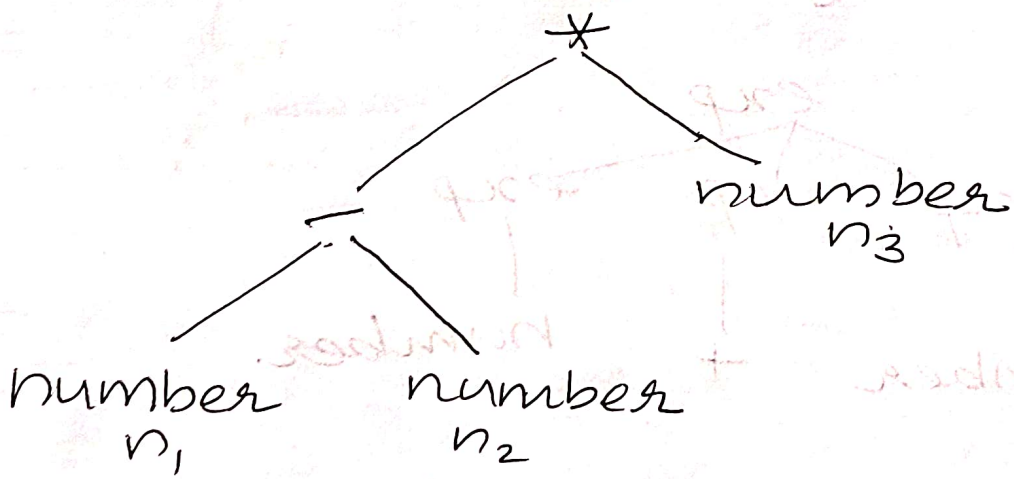
Abstract ST become;



For 3+4,



For (number − number) * number,



- A parser will go through all the steps represented by a passe tree, but will usually only construct an abstract syntax tree.

- They contain all the information needed for translation, in a more efficient form. than parse tree.

Syntax Error Handling

Common programming errors can occur at many different levels.

1) Lexical errors

→ Misspelling of identifiers, keywords or operators.

→ Missing quotes around text intended as a string.

2) Syntactic errors

→ misplaced semicolons or extra/missing braces.

3) Semantic errors

→ Type mismatches b/w operators & operands,

4) Logical errors

→ Anything from incorrect reasoning on the part of the programmer, to the use in a C program of the assignment operator '=' instead of the comparison operator '=='.

→ && , &
→ ||

we've parsing methods, which allows syntactic errors to be detected very efficiently.

(Methods such as LL and LR can detect errors as soon as possible)

The error handler in a parser has goals such as:

→ Report the presence of errors clearly and accurately. (line, column)

→ Recover from each error quickly enough to detect subsequent errors.

→ Add minimal overhead to the processing of correct programs.

Error-Recovery strategies

Once an error is detected, how should the parser recover?

The simplest method is for the parser to quit with an error. msg when it detects the first error.

There're different error recovery strategies:

1) Panic-mode recovery

On discovering an error, the parser discards i/p symbols one at a time until one of a designated set of 'synchronizing tokens' is found.

→ semicolon, } etc.

(to guarantee not to go into infinite loop).

2) Phrase-level recovery

A parser may perform local correction on the remaining input by some string that allows the parser to continue.

A typical local correction is
— to replace a comma by a semicolon.
— delete an extraneous semicolon
— insert a missing semicolon.

- The choice of the local correction is left to the compiler designer.
- phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. But it's difficult to correct.

3) Error Productions

Find from where these errors coming, rewrite or replace those production rules of that particular grammar.

4) Global correction.

Too costly.

Top-Down Parsing

- Top-down parsing can be viewed as 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.
- It uses leftmost derivation to construct a parse-tree.
- Construction of parse tree starts at the root and proceeds towards the leaves.
- Efficient parsers can be constructed more easily by hand using top-down methods.
- The main problem in TDP is to determine the production that has to be applied for a non-terminal (say A).

  Once we choose A-production, the parsing process consists of matching terminal symbols in the production body with the input string.
- Applicable to small class of languages.

(1) Recursive – Descent Parsing

→ It is a general parsing technique, but not widely used as it is not efficient (C slower).

→ It involves backtracking.

→ So more recursive, ie; it may require ~~several~~ repeated scans over the input.

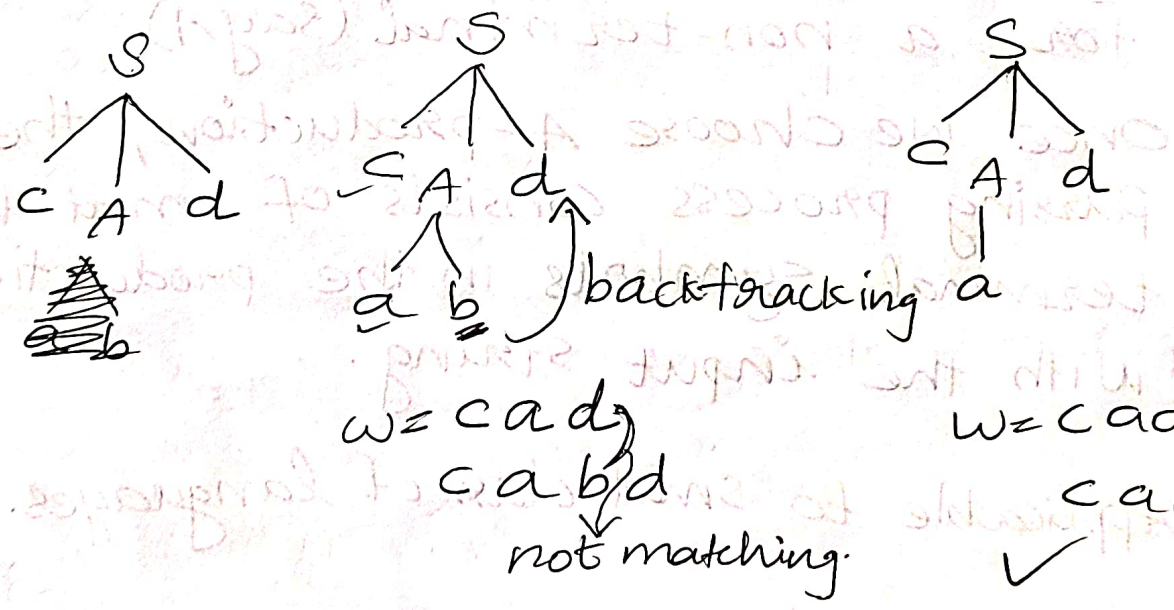
Example:-

Consider the grammar

$S \longrightarrow cAd$

$A \longrightarrow ab|a.$

Construct a parse tree for w=cad



(2) Predictive-parsing / LL(1)

Predictive-parsing or non-recursive descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1).

LL(1) → look ahead symbol.
→ for left most derivation.
↙ Scanning the input from left to right

'1' - for using one input symbol for lookahead at each step to make parsing action decisions.

→ The class of LL(1) grammars is rich enough to cover most programming constructs.

ie; No left-recursive or ambigous grammar can be LL(1).

→ Most popular and powerful grammar in top-down parsing.

Lookahead symbol:-
The next terminal that a compiler will try to match in the input.

I. Find FIRST() and FOLLOW().

II. Create Parsing Table / MTable.

In top-down parsing, FIRST() and FOLLOW() allow us to choose which production to apply, based on the next input symbol.

FIRST()

Rule 1: If x is a terminal, then

FIRST(x) = {x}.

Rule 2:

If $X \longrightarrow \varepsilon$ is a production, then add ε to FIRST(X).

Rule 3:

If X is a non-terminal and $X \longrightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1), ... FIRST(Y_{i-1}). ie; $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$

If ε is in FIRST(Y_j) for all $j = 1, 2 \ldots k$, then add ε to FIRST(X).

FOLLOW()

Rule 1:

Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right end-marker.

Rule 2:

If there is a production $A \rightarrow \alpha B\beta$, then everything in FIRST(β) except ε is in FOLLOW(B).

Rule 3:

If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$, where FIRST($\beta$) contains ε, then every thing in FOLLOW(A) is in FOLLOW(B).

Consider the grammar.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

$T' \rightarrow *FT / \varepsilon$

$F \rightarrow id / (E)$

Step 1: Eliminate left recursion and factoring

Step 2: Find FIRST() and FOLLOW().

	FIRST()	FOLLOW()
E	id, (	$,)
E'	+, ε	$,)
T	id, (	+, $,)
T'	*, ε	+, $,)
F	id, (	*, +, $,)

Step 3: LL(1) Table / Predictive parser table / M 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	$F \to id$			$F \to (E)$		

Construction of a predictive parsing table.

Algorithm

Input : Grammar G.

output : parsing table M.

Method

For each production $A \rightarrow \alpha$ of the grammar, do the following:

1) For each terminal a in FIRST(α), Add $A \rightarrow \alpha$ to M[A,a].

2) If ε is in FIRST(α), then for each terminal b, b in FOLLOW(A), add $A \rightarrow \alpha$ to M[A,b].

If ε is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A,\$] as well.

If after performing the above, there's no production at all in M[A,a], then set M[A,a] to error (an empty entry).

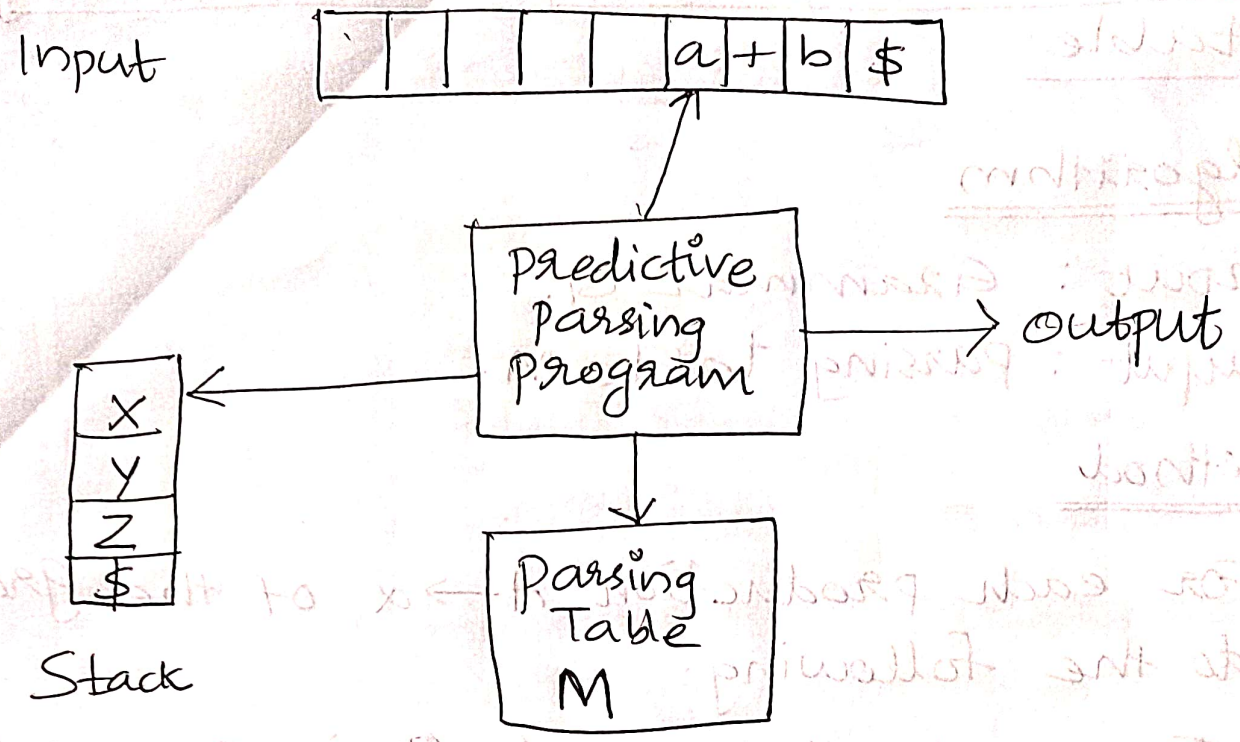


Fig: Model of a table driven predictive parser.

Example 2

$S \rightarrow (L) / a$
$L \rightarrow L,S / S$

Step 1: Elimination of left recursion

$S \rightarrow (L) / a$
$L \rightarrow SL'$
$L' \rightarrow ,SL' / \varepsilon$

Step 2: Elimination of left factoring.
Here, no common prefixes.

Step 3: Find FIRST() and FOLLOW().

FIRST(S) = {(, a}
FIRST(L) = {(, a}
FIRST(L') = {, , ε}

FOLLOW(S) = {, , $} (S is the starting symbol.)
FOLLOW(L) = {)}
FOLLOW(L') = {)}

LL(1) / parsing table.

	(	)	a	,	$
S	S → (L)		S → a		
L	L → SL'		L → SL'		
L'		L' → ε		L' → , SL'	

(a) $

Stack	Upstaing	Action
$S	(a) $	S → (L)
$)L(	↑	
$)L	(a) $	pop
$)L	a) $	L → SL'
$)L'S	a) $	S → a
$)L'a	a) $	pop
$)L'	) $	L' → ε
$)	) $	pop
$	$	

Accepted.

Example - 3

$S \longrightarrow aAB \mid bA \mid \varepsilon$

$A \longrightarrow aAb \mid \varepsilon$

$B \longrightarrow bB \mid \varepsilon$.

Construct parsing table for the given grammar. Also check whether 'aabb' is accepted or not.

	FIRST()	FOLLOW()
S	a, b, ε	{$}
A	a, ε	{b, $}
B	b, ε	{$}

	a	b	$
S	S → aAB	S → bA	S → ε
A	A → aAb	A → ε	A → ε
B	~~B → bB~~	B → bB	B → ε

Stack	Input string	Action
$S	aabb$	S → aAB
$BAa	aabb$	pop
$BA	abb$	A → aAb
$BbAa	abb$	pop
$BbA	bb$	A → ε
$Bb	bb$	pop
$B	b$	B → bB
$Bb	b$	pop
$B	$	B → ε
$	$	Accepted