SYNTAX ANALYSIS

ROLE OF PARSER

Introduction:

Syntatic representation of Language is explained in

* Backus Naur Form (BNF)
* Context Free Grammar (CFG)

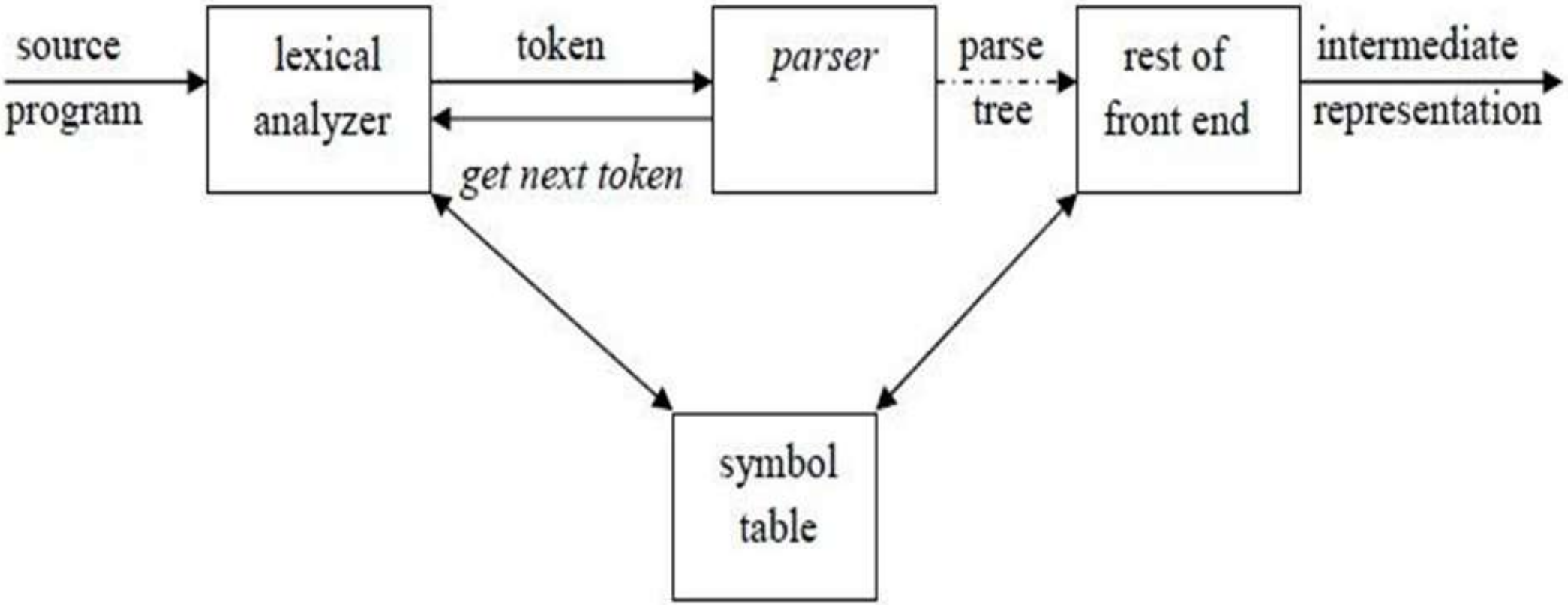
Usage of grammar in syntatic Language:

* Error identified easily
* Easy understanding
* Efficient parser construction

THE ROLE OF THE PARSER

- The parser obtains a string of tokens from the lexical analyzer, and verifies that the string of token names can be generated by the grammar for the source language.
- The parser will report any syntax errors in an intelligible fashion and recovers from commonly occurring errors to continue processing the remainder of the program.
- The parser constructs a parse tree and passes it to the rest of the compiler for further processing.

Position of parser in compiler model



Issues in the parser

* Parsing can't be done when redeclaration of variable can't be identified.
* Usage of variable before declaration can't be identified.
* Mismatch of data cannot be identified.

Parser

Accepts I/P as tokens and produces O/P as parse tree and produce syntax errors.

Types of Parser

* Top-down parser:
  - → predictive parser
  - → LL(1) parser
  - → Recursive descent parser
* Bottom-up parser
  - → shift reduce
  - → operator precedence
  - → LR parser
  - → SLR parser
  - → LALR parser
  - → CLR parser

ERROR HANDLING

Syntax Error Handling

→ Lexical error (eg: Misspelling of keyword)
→ Syntatic error (eg: Missing of semicolon)
→ Sumantic error (eg: Type mismatch)
→ Logical error (can't be identified by compilers)

* These errors should be identified and handled properly
* To recover the error within specific time, error recovery strategies are used.

- **The goals of the error handler in a parser**

1. Report the presence of errors clearly and accurately.
2. Recover from each error quickly enough to detect subsequent errors.
3. Add minimal overhead to the processing of correct programs.

- **Error-Recovery Strategies**
- Once an error is detected, how should the parser recover?

1. Panic-Mode Recovery
2. Phrase-Level Recovery
3. Error Productions
4. Global Correction

CONTEXT-FREE GRAMMARS

- Grammars were introduced to systematically describe the syntax of programming language constructs like expressions and statements.
- A context-free grammar (grammar for short) consists of terminals, nonterminals, a start symbol, and productions.

CONTEXT-FREE GRAMMARS (Contd..)

Notational Conventions

1) Terminals (a-z):

* Basic symbols to form a set of strings
* Terminals are also called as tokens
* Lowercase letters, special symbols (•, (•) + −), digits (0-9), Keywords (if, Else), Bold Face strings such as (if, id....).

CONTEXT-FREE GRAMMARS (Contd..)

2) Non-terminals

* A syntatic variable that denotes the set of strings.
* Made of terminals (or) non-terminals
* Uppercase letters (A-Z), lowercase italic letters (Expr)

CONTEXT-FREE GRAMMARS (Contd..)

3) Start Symbol:

One non-terminal in the grammar is selected as start symbol (or) distinguished symbol usually it is 'S'.

* Usually 'S' denotes the... start symbol.

4) Production

Otherwise called as rewriting rules in which terminals and non-terminals are combined to Form string

Eg: $A \rightarrow bcd$

(or)

$A ::= bcd$

CONTEXT-FREE GRAMMARS (Contd..)

Example for CFG

expr → expr op expr

expr → (expr)

expr → - expr

expr → id

op → +

op → -

op → *

op → /

op → ↑

9 terminals : id, +, −, *, /, ↑, (,), −

2 Non-terminals : expr, op

∴ Expr → Start symbol because it generates op and vice versa Not possible

CONTEXT-FREE GRAMMARS (Contd..)

eg: $A \rightarrow \alpha$

A: Non-Terminal

α: Terminal

Here A produces a 'α' which is a terminal

CONTEXT-FREE GRAMMARS (Contd..)

Grammar symbols.

* upper case letters (X, Y, Z)
* Lower case letters (u, v, z)
* Lower case greek letters (α, β, γ)
* $A \to \alpha_1, A \to \alpha_2, A \to \alpha_3 \ldots\ldots A \to \alpha_K$;

$\alpha_1 | \alpha_2 | \alpha_3 \ldots\ldots | \alpha_K \to$ Alternative for A

eg: $E \to EAE \mid (E) \mid -E \mid id$

$A \to + \mid - \mid / \mid * \mid \uparrow$

* Number of productions : 9

$E \to E+E$

$E \to E-E$

$E \to E/E$

$E \to E*E$

$E \to E \uparrow E$

$E \to (E)$

$E \to -E$

$E \to id$

CONTEXT-FREE GRAMMARS (Contd..)

Derivations, Reduction and Parse tree:

* **Derivations:**

Process of generating a valid string with a help of grammar is called derivations.

* **Reduction**

The process of Recognizing the valid string using grammar will be called as Reduction.

* **Parse tree:**

Graphical Representation of both derivation and Reduction.

CONTEXT-FREE GRAMMARS (Contd..)

(a) Derivations:

Derivation symbol is "$\Rightarrow$"

Eg:

$A \rightarrow \gamma$

$\alpha A \beta \Rightarrow \alpha \gamma \beta$

Here "$\alpha \gamma \beta$" is Derived From the production

$A \rightarrow \gamma$

$\overset{*}{\Rightarrow}$: Deriving zero (or) more times

$\overset{+}{\Rightarrow}$: Deriving one (or) more times

CONTEXT-FREE GRAMMARS (Contd..)

Types of Derivation

1* Left most derivation - Replacing the string from left handside

2* Right most derivation - Replacing the string from right hand side.

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

Derive: $-(id+id)$

$E \Rightarrow -E$

$\Rightarrow -(E)$

$\Rightarrow -(E+E)$

$\Rightarrow -(id+E)$

$\Rightarrow -(id+id)$

CONTEXT-FREE GRAMMARS (Contd..)

Example :

Leftmost

$E \Rightarrow -E$

$E \Rightarrow -(E)$

$E \Rightarrow -(E+E)$

$\Rightarrow -(id+E)$

$\Rightarrow -(id+id)$

Rightmost

$E \Rightarrow -E$

$\Rightarrow -(E)$

$\Rightarrow -(E+E)$

$\Rightarrow -(E+id)$

$\Rightarrow -(id+id)$

CONTEXT-FREE GRAMMARS (Contd..)

b) Reduction

Reduction symbol is '$\rightarrow$' Eg: $A \rightarrow \gamma$

Replacing γ by A is called Reduction

$\xrightarrow{*}$: Reducing 0 (or) more times

$\xrightarrow{+}$: Reducing 1 (or) more times

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

(i) check $-(id+id)$ valid or not:

$E \rightarrow -(id+id)$

$\rightarrow -(E+id)$

$\rightarrow -(E+E)$

$\rightarrow -(E)$

$\rightarrow -E$

$E \rightarrow -E$ valid string

$E \rightarrow -(id/id)$

$\rightarrow -(E/id)$

$E \rightarrow -(E/E) \rightarrow$ Not a valid string because reduction not possible

CONTEXT-FREE GRAMMARS (Contd..)

Types of Reduction

* Left-Most Reduction
* Right-Most Reduction

Left most

$E \rightarrow -(id+id)$

$E \rightarrow -(E+id)$

$E \rightarrow -(E+E)$

$E \rightarrow -(E)$

$E \rightarrow -E$

$E \rightarrow E$

Right most

$E \rightarrow -(id+id)$

$E \rightarrow -(id+E)$

$E \rightarrow -(E+E)$

$E \rightarrow -(E)$

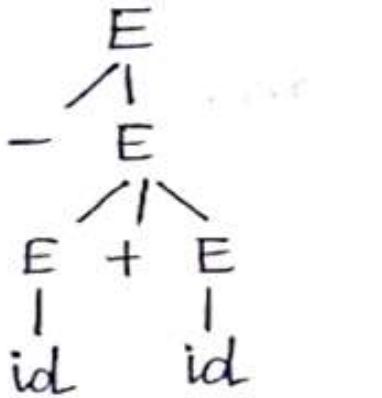
$E \rightarrow -E$

$E \rightarrow E$

Parse tree

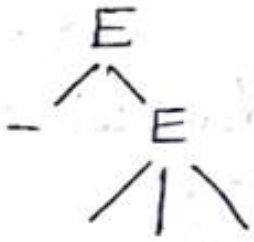
Eg: $E \rightarrow (E) | E*E | E+E | -E | id$; construct parse tree for,

(i) $-(id+id)$



valid

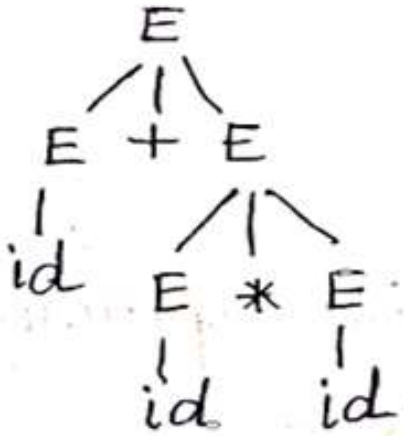
(ii) $-(id-id)$



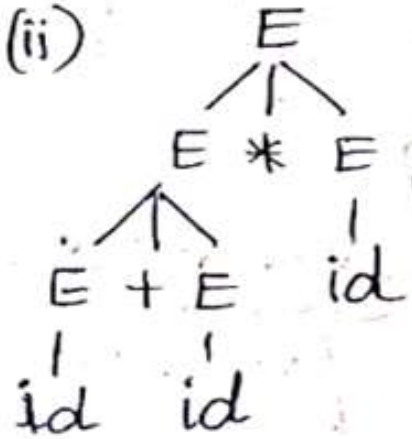
invalid

(ii) $id+id*id$ → construct parse tree

(i)



(ii)



CONTEXT-FREE GRAMMARS (Contd..)

Derivations

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

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

Ambiguity

If same statement produces two different parse tree is called Ambiguity.

Dangling else grammar:

Which parse tree we want is selected and others are rejected.

Here disambiguous rules can be used to eliminate ambiguity.

CS8602 COMPILER DESIGN

UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – **Writing a grammar** –Top Down Parsing – General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR(0) Item Construction of SLR Parsing Table -Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC.

WRITING A GRAMMAR

Writting a grammar

1) Regular Expression Vs CFG
2) Verifying the language genrated by a grammar
3) Eliminating Ambiguity
4) Elimination of left Recursion
5) Left Factoring

1) Regular Expressions Vs CFG

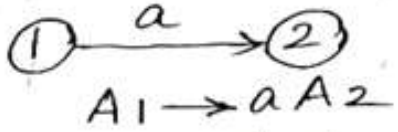
* The Algorithm for constructing the grammar from NFA

step1: For each state i of the NFA create a non-terminal Ai

begin

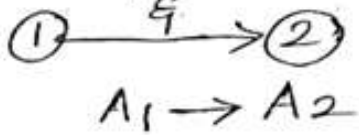
step 2: If the state 'i' has transition to state 'j' on symbol 'α' then introduce the production

$A_i \rightarrow \alpha A_j$



step3: if state 'i' goes to state 'j' on input 'ε' then introduce the production

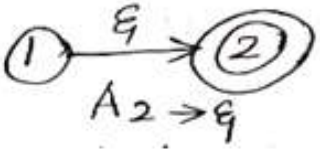
$A_i \rightarrow A_j$



end

step 4: if 'i' is an accepting state (final state), then introduce

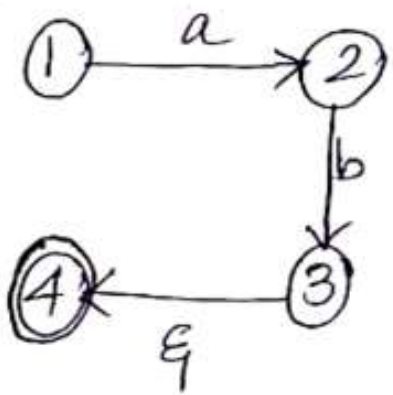
$A_i \rightarrow \varepsilon$



WRITING A GRAMMAR (Contd.)

step 5: if 'i' is the start state make Ai be the start symbol for grammar

eg:



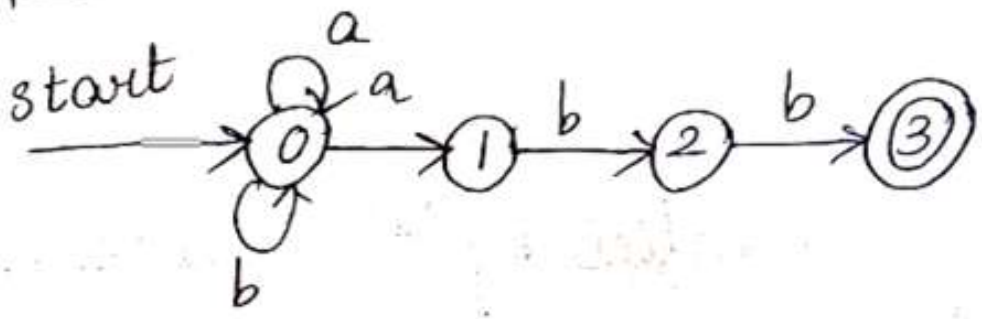
$A_1 \rightarrow aA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow A_4$

$A_4 \rightarrow \varepsilon$

Construct NFA for $(a/b)^*abb$:-



The productions are

$A_0 \rightarrow aA_1$

$A_0 \rightarrow aA_0$

$A_0 \rightarrow bA_0$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \epsilon$

(or)

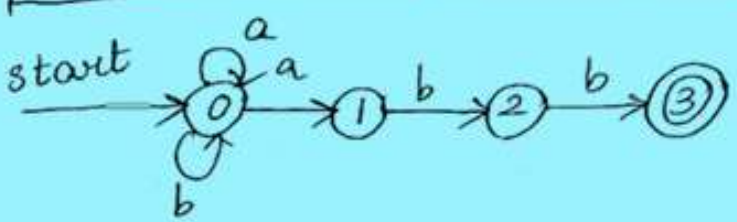
$A_0 \rightarrow aA_1 | aA_0 | bA_0$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \epsilon$

construct NFA for (a/b)* abb



The string "aababb" is obtained by using the grammar above

$A_0 \rightarrow aA_0$

$\rightarrow aaA_0$

$\rightarrow aabA_0$

$\rightarrow aabaA_1$

$\rightarrow aababA_2$

$\rightarrow aababbA_3$

$A_0 \rightarrow aababb$

WRITING A GRAMMAR (Contd.)

2) Verifying a language generated by a grammar:

steps followed:

→ We must show that every string by 'G' is in language 'L'

→ Every string is in language 'L' can be generated by grammar 'G'.

$\Rightarrow S \rightarrow (S)\,S \mid \epsilon$

condition : every string generated by 'S' is balanced

$S \rightarrow \epsilon$

$S \rightarrow (S)S$

WRITING A GRAMMAR (Contd.)

Here 's' can be considered as 'x' and 'y'

$S \rightarrow (x)y$

Every balanced string are derivable from 'x'.

Derive the string $(x)y$ and $((x)y)$

$S \Rightarrow (S)S$

$S \Rightarrow (x)y$

$S \Rightarrow (S)S$

$\Rightarrow ((S)S)S$

$\Rightarrow ((x)S)S$

$\Rightarrow ((x)y)S$

$\Rightarrow ((x)y)\varepsilon$

$\Rightarrow ((x)y)$

WRITING A GRAMMAR (Contd.)

3) Eliminating Ambiguity

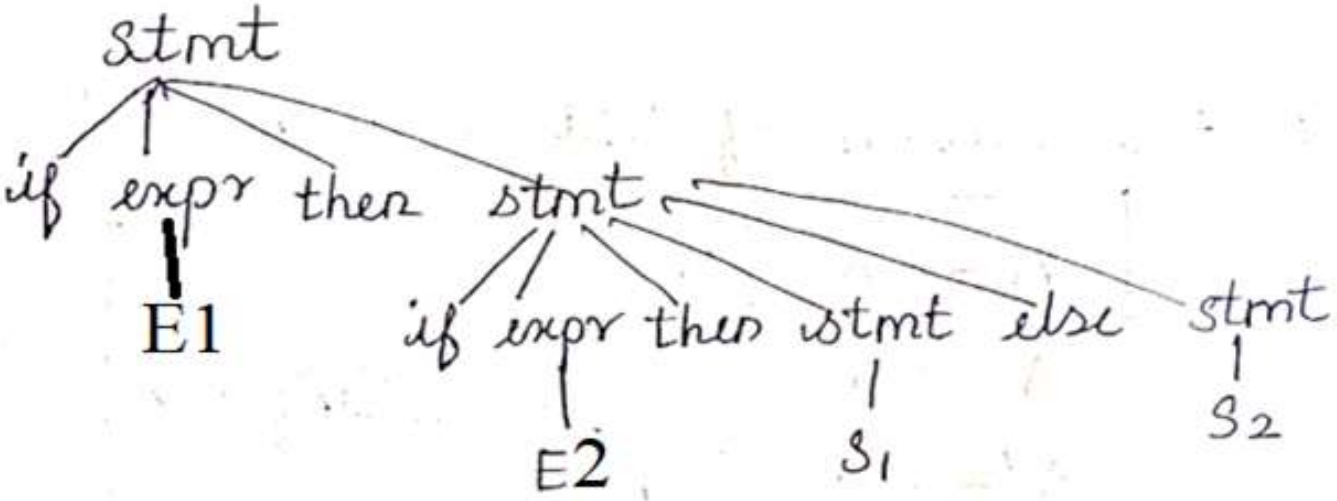
→ consider dangling – else grammar

Production:

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

if E_1 then if E_2 then S_1 else S_2, generate the parse tree for this.

TREE 1:



WRITING A GRAMMAR (Contd.)

Production:

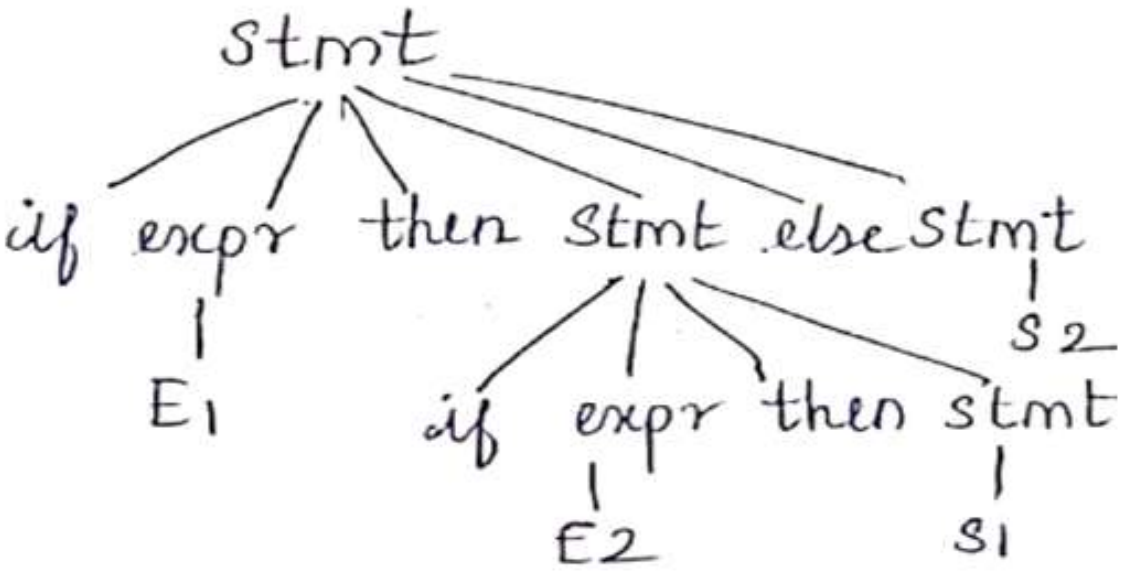
stmt → if expr then stmt

| if expr then stmt else stmt

| other.

if E_1 then if E_2 then S_1 else S_2, generate the parse tree for this.

TREE 2:



WRITING A GRAMMAR (Contd.)

Productions :

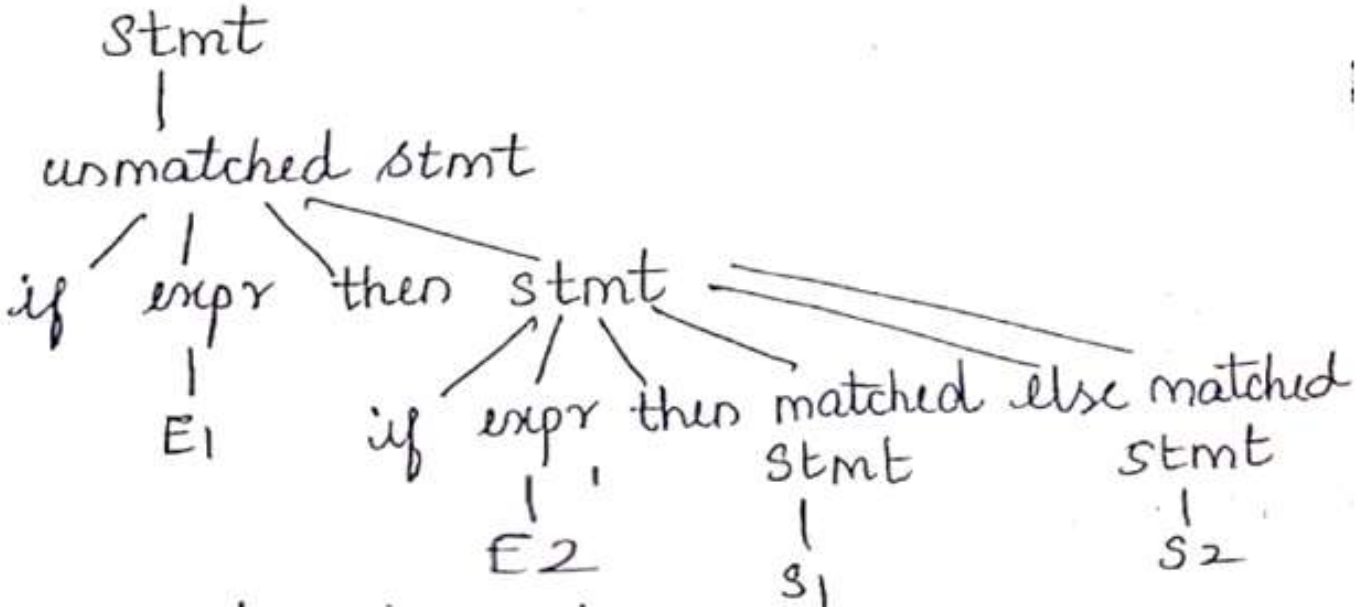
stmt → matched stmt / unmatched stmt

Matched stmt → if expr then Matched stmt else Matched stmt | other

Unmatched stmt → if expr then stmt .

PARSE TREE

if E_1 then if E_2 then S_1 then else S_2



WRITING A GRAMMAR (Contd.)

4) Elimination of left Recursion:

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

$A \rightarrow \beta A'$ } rewritting as

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

$E \rightarrow E+T \mid T$

Here $E \rightarrow E+T$

$E + T + T$

$E+T + T + T$

⋮

[leads to left recursion]

goes on so called as left recursion because replacing the 'E' by the left side.

WRITING A GRAMMAR (Contd.)

Rules for eccuring left recursion:

When non-terminal in both sides (same letter)

If any statement has the left recursion

$A \rightarrow A\alpha | \beta$; then the left recursion can be eliminated by rewritting the grammar as,

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

Example: Remove the Left Recursion for the given grammar

The grammar $E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id$

The grammar $E \rightarrow E+T|T$ is rewritting as,

1&) $\underset{A}{E} \rightarrow \underset{A}{E}\underset{\alpha}{+T}/\underset{\beta}{T}$

Rewritten

(i) $A \rightarrow \beta A'$

$F \rightarrow TE'$

(ii) $A' \rightarrow \alpha A' | \varepsilon$

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

WRITING A GRAMMAR (Contd.)

$A \to A\alpha \mid \beta$
$A \to \beta A'$
$A' \to \alpha A' \mid \varepsilon$
} rewritting as

2*) $T \to T*F \mid F$
Rewritten.
(i) $A \to \beta A'$
$T \to FT'$
(ii) $A' \to \alpha A' \mid \varepsilon$
$T' \to *FT' \mid \varepsilon$

3) $F \to (E) \mid id$
This production has no left recursion so no need to rewrite the production

The rewritten grammar after elimination of left recursion is :

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

WRITING A GRAMMAR (Contd.)

In general, we can eliminate immediate left recursions if there are productions of the form

$$A \to A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \beta_3 | \ldots | \beta_n$$

for eliminating the left recursion transform to

$$A \to \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$$
$$A' \to \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \varepsilon$$

when A is left recursive non-terminal, α is a sequence of non-terminal which is not ε, β is sequence of non-terminal that doesn't start A (No left recursion in stmt)

WRITING A GRAMMAR (Contd.)

$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \beta_3 | \ldots | \beta_n$

Rewrite as,

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

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \epsilon$

Eg: $S \rightarrow Aa | b$

$A \rightarrow Ac | Sd | \epsilon$

$S \rightarrow Aa | b$, Here there is no left recursion

$A \rightarrow Ac | Sd | \epsilon$

This is rewritten as,

$A \rightarrow SdA' | \epsilon A'$

$A' \rightarrow cA' | \epsilon$

Hence the rewritten grammar for the statement after elimination of left recursion is

$S \rightarrow Aa | b$

$A \rightarrow SdA' | \epsilon A'$

$A' \rightarrow cA' | \epsilon$

Algorithm for eliminating left recursion

Input: Grammar G with no cycles or ε-productions

Output: An equivalent grammar with no left recursion

Method: Apply the algorithm, for dangling else grammar..

Note that the resulting non-left recursive grammar may have ε-productions

1) Arrange the non-terminals in some order $A_1, A_2, \ldots A_n$
2) For i := 1 to n do
   Begin
   for j := 1 to i-1 do
   Begin
   Replace each production of form $A \to A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m$ $\beta_1 | \beta_2 | \ldots | \beta_n$ ... the rewrite productions are

$$A \to \beta_1 A' | \beta_2 A' | \beta_3 A' | \ldots | \beta_n A'$$

$$A' \to \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \varepsilon$$

End
Eliminate the immediate left recursion among the Ai productions
End

WRITING A GRAMMAR (Contd.)

5) Left factoring

→ Left factoring is a grammar transforming that a grammar suitable for predictive parsing (Top-down parsing).

→ In top-down parsing, when it is not clear which is of two alternative productions to use to expand a non-terminal A, defer the decision till we have enough I/P.

→ If there are productions of the form

$A \to \alpha\beta_1 \mid \alpha\beta_2$, then rewrite the stmt-for left-factoring as follows.

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

WRITING A GRAMMAR (Contd.)

consider the grammar $S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

$i \rightarrow if \quad E \rightarrow exp \quad t \rightarrow then \quad S \rightarrow Stmt, \; e \rightarrow else$

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

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \varepsilon \mid eS$

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

Rewritten as,

$A \rightarrow \alpha A'$

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

$E \rightarrow b$

After left factoring,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \varepsilon \mid eS$

$E \rightarrow b$

for "ibta"

$S \rightarrow iEtSS'$

$\rightarrow ibtSS' \; [E \rightarrow b]$

$\rightarrow ibtaS' \; [S \rightarrow a]$

$\rightarrow ibta\varepsilon \; [S' \rightarrow \varepsilon]$

$\rightarrow ibta.$

CS8602 COMPILER DESIGN

UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar –**Top Down Parsing** – General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table -Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC.

PARSING

Parsing is the process of analyzing a continuous stream of i/p in order to determine its grammatical structure with respect to a given formal grammar.

The task of the parser is essential to determine if & how the i/p can be derived from the start symbol within the rules of the formal grammar.

The types of parsing are :-

(i) Top down parsing

(ii) Bottom - up parsing

PARSING (Contd.)

i) Top down parsing :-

A parser can start with the start symbol & try to transform it to the i/p (i.e) the parser starts from the largest elements & breakes down into incrementally smaller units.

Eg : LL Parser.

ii) Bottom - up parsing :-

A parser can start with the i/p & attempt to rewrite it to the start symbol

Eg. LR Parser.

PARSING (Contd.)

Parser :- A parser for Grammar G1 is a program that takes a string w as i/p and Produces o/p either a parse tree for w, if w is a sentence of Grammar G1 or an error message indicating that w is not a sentence of G1.

Parsing :- process / methodology to generate parse tree.

Parser :- A tool for performing parsing.

PARSING - Top Down Parsing (Contd.)

1) Top down parsing :-

Top down parsing can be viewed as an attempt to find the left most Derivation for an i/p string. It can also be viewed as attempt to construct parse tree for the i/p starting from the root & creating the nodes of parse tree in pre-order.

PARSING - Top Down Parsing (Contd.)

Example: Consider the grammar

S→cAd

A→ab/a

and the input string **w=cad**

Solution:

Construct the **parse tree** to track the input string **"cad"**

PARSING - Top Down Parsing (Contd.)

Given grammar

S→cAd

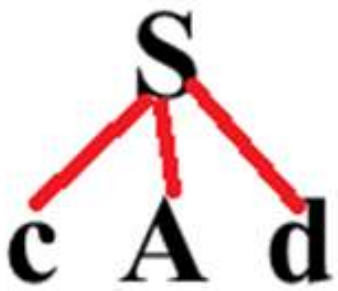
A→ab/a

given input string **w=cad**

Construct the **parse tree** to track the input string **"cad"**

Step1: Initially create the Parse tree with the starting symbol **'S'** . The input pointer points to c (ie) the 1st symbol of w. Expand the tree with the production of **S.**

Parse tree



PARSING - Top Down Parsing (Contd.)

Given grammar

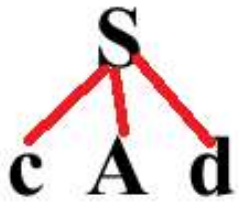
S→cAd

A→ab/a

given input string **w=cad**

Step1: Initially create the Parse tree with the starting symbol **'S'.** The input pointer points to c (ie) the 1st symbol of w. Expand the tree with the production of **S.**

Parse tree



Input string	Prediction	Left Most Derivation	Comment
↑cad	S→cAd	→cAd	I/p ptr match with left most symbol of derivation advance the ptr

PARSING - Top Down Parsing (Contd.)

Given grammar

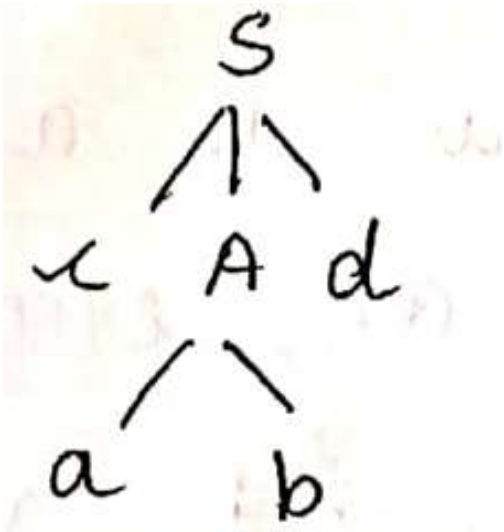
S→cAd

A→ab/a

given input string **w=cad**

Step2: The left most leaf c matches the 1st symbol of w. advance the pointer to the 2nd symbol of w, 'a' & consider A in tree which is the next leaf. Expand A with the 1st alternative ab.

Parse tree



cad ↑	A→ab	→cabd	I/p ptr match with second symbol of derivation Advance the ptr

PARSING - Top Down Parsing (Contd.)

Given grammar

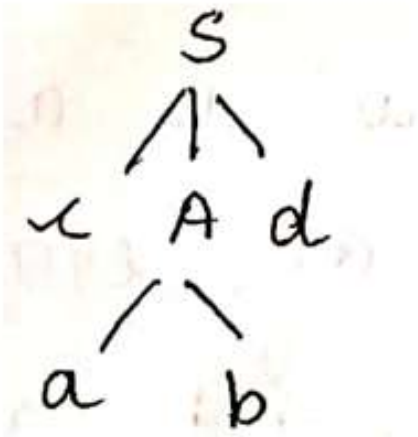
S→cAd

A→ab/a

given input string **w=cad**

Step3: The 2nd symbol 'a' of w matches with 2nd leaf of tree so advance the ptr to the 3rd symbol of w, 'd' but the 3rd leaf of tree is 'b' which is not matched with the i/p symbol 'd'. Hence discard the chosen production & reset the ptr to 2nd position. This is called as **Backtracking.**

Parse tree



cad ↑		→ cabd	I/p ptr doesn't match with 3rd symbol, hence the previous prediction is wrong. Discard it reset the ptr (Back tracking)

PARSING - Top Down Parsing (Contd.)

Given grammar

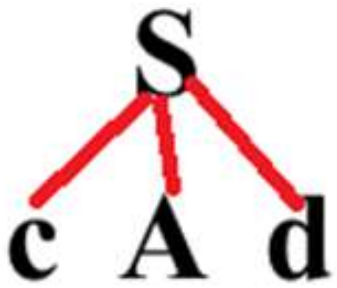
S→cAd

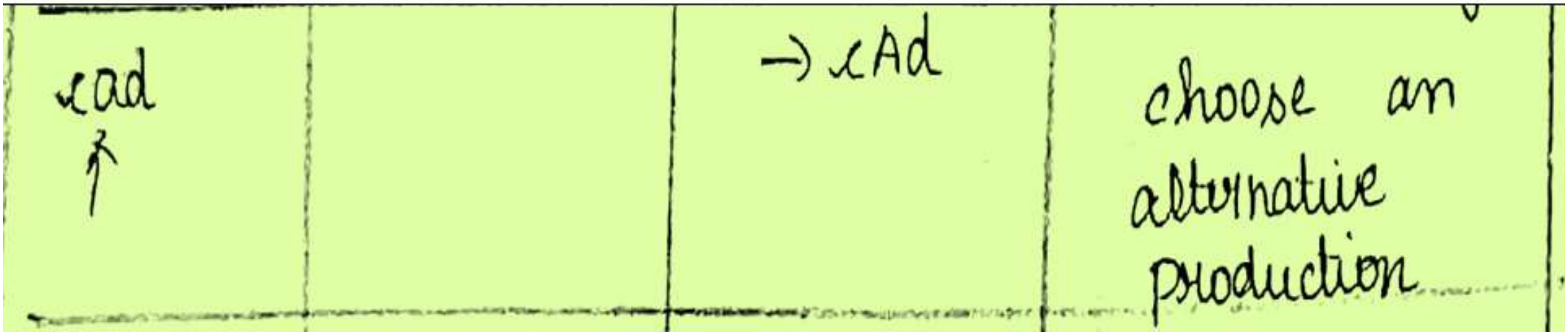
A→ab/a

given input string **w=cad**

Step3: The 2nd symbol 'a' of w matches with 2nd leaf of tree so advance the ptr to the 3rd symbol of w, 'd' but the 3rd leaf of tree is 'b' which is not matched with the i/p symbol 'd'. Hence discard the chosen production & reset the ptr to 2nd position. This is called as **Backtracking.**

Parse tree





PARSING - Top Down Parsing (Contd.)

Given grammar

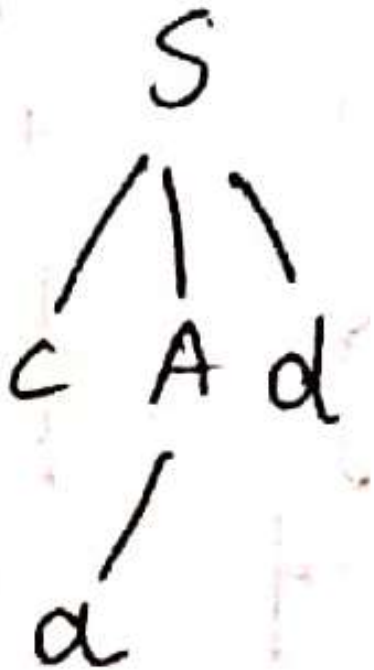
S→cAd

A→ab/a

given input string **w=cad**

Step4: Now try the 2nd alternative for A. the 2nd leaf 'a' matches with i/p symbol of w. hence advance the ptr to 3rd position 'd' & which gets matched with 3rd leaf of tree, thus we can produce a parse tree for w.

Parse tree



cad ↑	A→a	→ cad	I/P ptr match with 2nd symbol & advance ptr

PARSING - Top Down Parsing (Contd.)

Given grammar

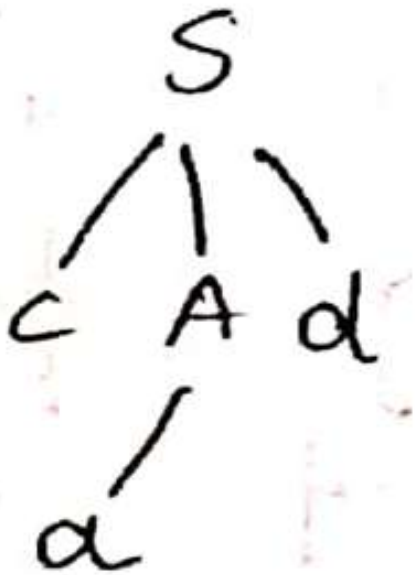
S→cAd

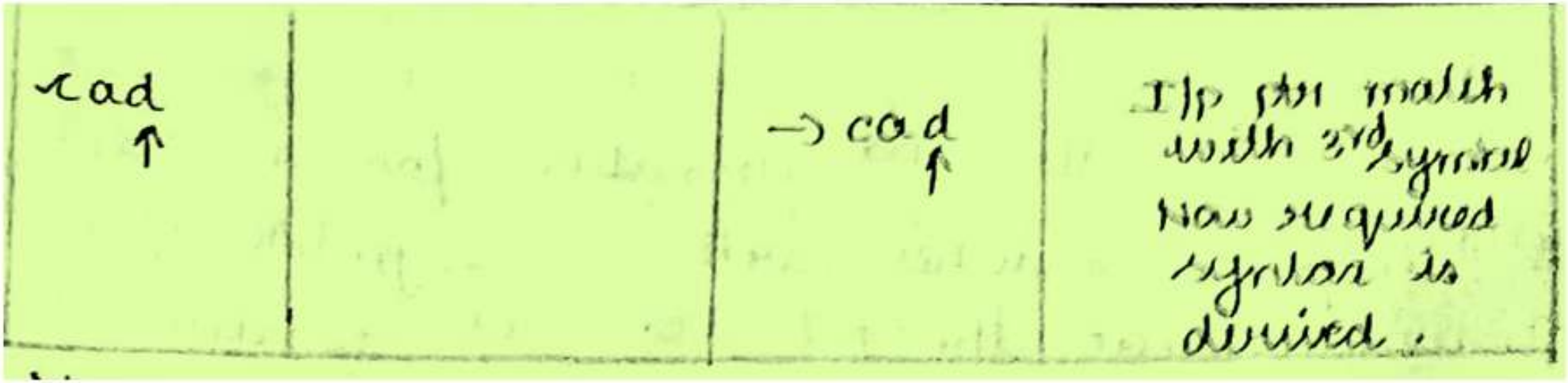
A→ab/a

given input string **w=cad**

Step4: Now try the 2nd alternative for A. the 2nd leaf 'a' matches with i/p symbol of w. hence advance the ptr to 3rd position 'd' & which gets matched with 3rd leaf of tree, thus we can produce a parse tree for w.

Parse tree





PARSING - Top Down Parsing (Contd.)

There are several difficulties with TDP while writing the procedure.

(i) left recursive grammar can cause a TDP to go into an infinite loop. When writing a procedure to use top-down parsing we must eliminate all left recursion from the grammar.

Eg:- Consider the grammar

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

Eg:- Consider the grammar

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

TDP writes the procedure for E, T, & F for 3 procedures:

```
Procedure F()          Procedure E()          procedure T()
begin                  begin                  begin
. .                    E()                    T()
. .                    . .                    . .
. .                    . .                    . .
end                    end                    end
```

Here the procedure E() infinitely calls E() (recursively) & produce an infinite loop similarly the procedure T() calls T() recursively & produce a Infinite loop.

PARSING - Top Down Parsing (Contd.)

- There are TDP with no backtracking such as

1. **Recursive descent parser**
2. **Predictive parser**
3. **Rejection of valid string**
4. **Error reporting:-** When failure is reported in TDF we have very little idea the error actually occurs.

(iii) Rejection of valid string:- Due to backtracking we may reject some valid sentences, for eg in a grammar

S → cAd
A → ab/a we use 'a' and 'ab' order of alternative of A.

CS8602 COMPILER DESIGN

UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar –Top Down Parsing – **General Strategies Recursive Descent Parser** Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table -Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC.

1) Recursive descent parsing :-

A parser that uses a set of recursive procedures to recognize its i/p with no backtracking is called recursive descent parser.

i) It is easy to write.

ii) Efficient if written in a language that implements procedure calls effectively.

Need for Recursive descent Parsing :-

(i) Elimination of left Recursion

(ii) left factoring.

PARSING - Top Down Parsing (Contd.)

Recursive descent:

```
void A()
{
choosen A Production, where A → x1, x2, ..... xk
for i=1 to k
{
if xi is a non-terminal
call procedure xi()
else
  if (xi == current i/p symbol 'A')
then
Advance the i/p to the next symbol.
else
  error
}
}
```

Eg:- Consider the grammar that recognize arithmetic expressions:-

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

This grammar contains left recursion & which should be eliminated

After eliminating the left recursion the grammar becomes.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Rules for acciving left recursion:

When non-terminal in both sides (same letter)

If any statemint has the left recursion

$A \rightarrow A\alpha | \beta$; then the left recursion can be eliminated by rewritting the grammar as,

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

The grammar $E \rightarrow E+T|T$ is rewritting as,

1 &) $\underset{\bar{A}}{E} \rightarrow \underset{\bar{A}}{E}\underset{\alpha}{+T}/\underset{\beta}{T}$

Rewritten

(i) $A \rightarrow \beta A'$

$F \rightarrow TE'$

(ii) $A' \rightarrow \alpha A' | \varepsilon$

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

Example: Remove the Left Recursion for the given grammar

The grammar $E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id$

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

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$ } rewritting as

2*) $T \rightarrow T*F \mid F$

Rewritten.

(i) $A \rightarrow \beta A'$

$T \rightarrow FT'$

(ii) $A' \rightarrow \alpha A' \mid \varepsilon$

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

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

This production has no left recursion so no need to rewrite the production

The rewritten grammar after elimination of left recursion is :

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

The procedure for grammar is given as follows:

```
procedure E()
begin
T()
E Prime()
end;
```

E -> TE'
E' -> +TE' / ε
T -> FT'
T' -> *FT' / ε
F -> (E) / id

```
Procedure EPrime()
if i/p symbol = '+' then
begin
ADVANCE()
T()
EPrime()
end;
```

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

```
Procedure T()
begin
F()
TPrime()
end;
```

E → TE'
E' → +TE' / ε
T → FT'
T' → *FT' / ε
F → (E) / id

```
Procedure TPrime ()
if   i/p symbol = '*' then
begin
ADVANCE ()
 FC)
TPrime ()
end;
```

```
Procedure F()
if input symbol = 'id' then
ADVANCE ()
else if input symbol = '(' then
begin
 ADVANCE()
 E()
if input symbol = ')' then
ADVANCE ();
else
  ERROR
 end;
```

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

```
Procedure E()
begin
T()
Eprime()
end;

procedure Eprime()
If i/p symbol = '+' then
begin
ADVANCE()
T()
EPRIME()
end;
```

```
procedure T()
begin
F()
Tprime()
end();

procedure Tprime()
If i/p symbol = '*' then
begin
ADVANCE()
F()
TPRIME()
end;
```

```
procedure F()
If i/p symbol = 'id' then
ADVANCE()
elseIf i/p symbol = '(' then
begin
 ADVANCE()
 E()
 If i/p symbol = ')' then
 ADVANCE();
 else
  ERROR
end;
```

Example 1:

Consider the input string **id + id * id** and parse the string using recursive descent parsing.

```
Procedure E()
begin
T()
Eprime()
end;

procedure Eprime()
If i/p symbol = '+' then
begin
ADVANCE()
T()
EPRIME()
end;
```

```
procedure T()
begin
F()
Tprime()
end();

procedure Tprime()
If i/p symbol = '*' then
begin
ADVANCE()
F()
TPRIME()
end;
```

```
procedure F()
If i/p symbol = 'id' then
ADVANCE()
elseIf i/p symbol = '(' then
begin
 ADVANCE()
 E()
 If i/p symbol = ')' then
 ADVANCE();
 else
   ERROR
end;
```

```
Procedure E()
begin
T()
Eprime()
end;

procedure Eprime()
If i/p symbol = '+' then
begin
ADVANCE()
```

```
procedure T()
begin
F()
Tprime()
end();

procedure Tprime()
If i/p symbol = '*' then
begin
ADVANCE()
```

```
procedure F()
If i/p symbol = 'id' then
ADVANCE()
else If i/p symbol = '(' then
begin
ADVANCE()
E()
If i/p symbol = ')' then
```

Production	Input String	Comments
1. E()	id + id * id ↑	Pointer pointing to id
2. T()	id + id * id ↑	Pointer pointing to id
3. F()	id + id * id ↑	Pointer pointing to id
4. ADVANCE()	id + id * id ↑	Advance the ptr pointing to '+' return to T() and called TPRIME()

```
Procedure E()
begin
T()
Eprime()
end;

procedure Eprime()
If i/p symbol = '+' then
begin
ADVANCE()
T()
EPRIME()
end;
```

```
procedure T()
begin
F()
Tprime()
end();

procedure Tprime()
If i/p symbol = '*' then
begin
ADVANCE()
F()
TPRIME()
end;
```

```
procedure F()
If i/p symbol = 'id' then
ADVANCE()
elseIf i/p symbol = '(' then
begin
  ADVANCE()
  E()
  If i/p symbol = ')' then
  ADVANCE();
  else
    ERROR
end;
```

5. TPRIME()	id + id * id ↑	Pointer Pointing to '+'
6. EPRIME()	id + id * id ↑	Pointer Pointing to '+'
7. ADVANCE()	id + id * id ↑	Advance the ptr pointing to id.

```
Procedure E()
begin
T()
Eprime()
end;

procedure Eprime()
If i/p symbol = '+' then
begin
ADVANCE()
T()
EPRIME()
end;
```

```
procedure T()
begin
F()
Tprime()
end();

procedure Tprime()
If i/p symbol = '*' then
begin
ADVANCE()
F()
TPRIME()
end;
```

```
procedure F()
If i/p symbol = 'id' then
ADVANCE()
elseIf i/p symbol = '(' then
begin
ADVANCE()
E()
If i/p symbol = ')' then
ADVANCE();
else
  ERROR
end;
```

8. T()	id + id * id ↑	Pointer pointing id
9. F()	id + id * id ↑	Pointer pointing id
10. ADVANCE()	id + id * id ↑	Advance the pointer, and the pointer pointing the '*'

```
Procedure E()
begin
T()
Eprime()
end;

procedure Eprime()
If i/p symbol = '+' then
begin
ADVANCE()
T()
EPRIME()
end;
```

```
procedure T()
begin
F()
Tprime()
end();

procedure Tprime()
If i/p symbol = '*' then
begin
ADVANCE()
F()
TPRIME()
end;
```

```
procedure F()
If i/p symbol = 'id' then
ADVANCE()
elseIf i/p symbol = '(' then
begin
  ADVANCE()
  E()
  If i/p symbol = ')' then
  ADVANCE();
  else
    ERROR
end;
```

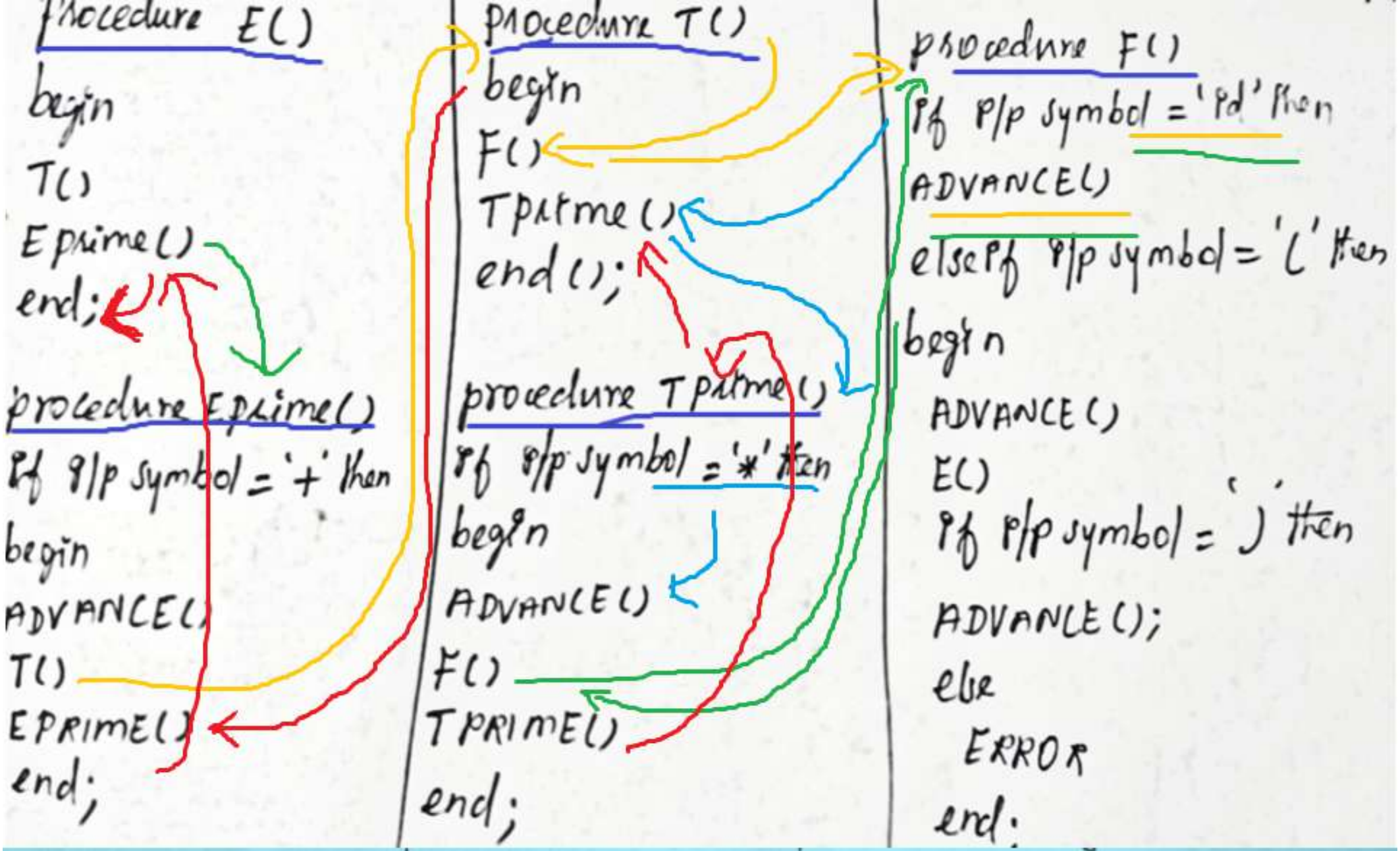
11. TPRIME()	i d + id * i d ↑	Pointer pointing to '*'
12. ADVANCE()	id + id * id ↑	Advance the ptr, and the ptr pointing to the id.

```
Procedure E()
begin
T()
Eprime()
end;

procedure Eprime()
If i/p symbol = '+' then
begin
ADVANCE()
T()
EPRIME()
end;
```

```
procedure T()
begin
F()
Tprime()
end();

procedure Tprime()
If i/p symbol = '*' then
begin
ADVANCE()
F()
TPRIME()
end;
```

```
procedure F()
If i/p symbol = 'id' then
ADVANCE()
elseIf i/p symbol = '(' then
begin
  ADVANCE()
  E()
  If i/p symbol = ')' then
  ADVANCE();
  else
    ERROR
end.
```

13. F()	id + id * id ↑	Pointer pointing to id
14. ADVANCE()	id + id * id ↑	Advance the pointer, and the ptr pointing to end of string.



15. TPRIME()	id + id * id ↑	Comments no change.
16. EPRIME()	id + id * id ↑	comments halt.

Workout

- **Example 2:**

Consider the input string **id*id** and parse the string using recursive descent parsing.

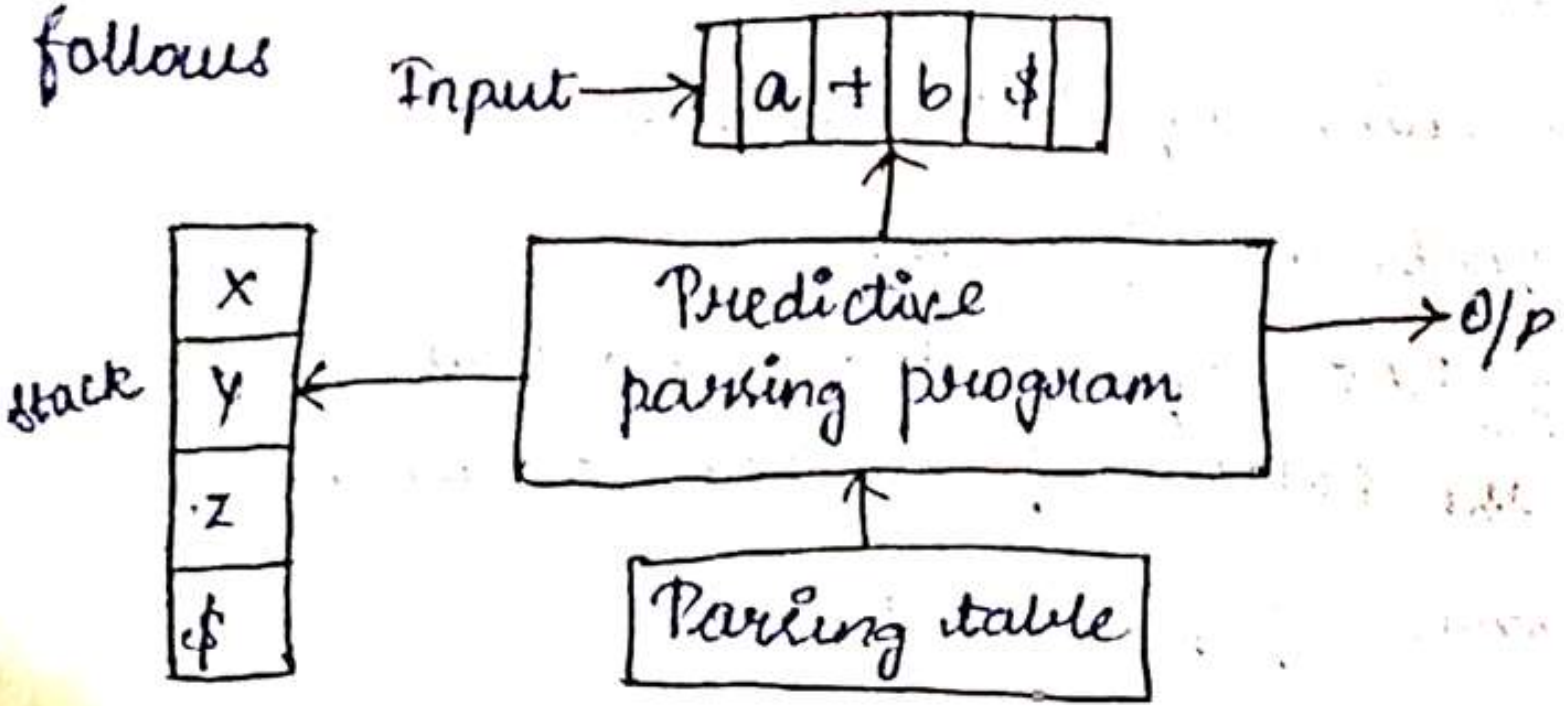
CS8602 COMPILER DESIGN

UNIT II SYNTAX ANALYSIS

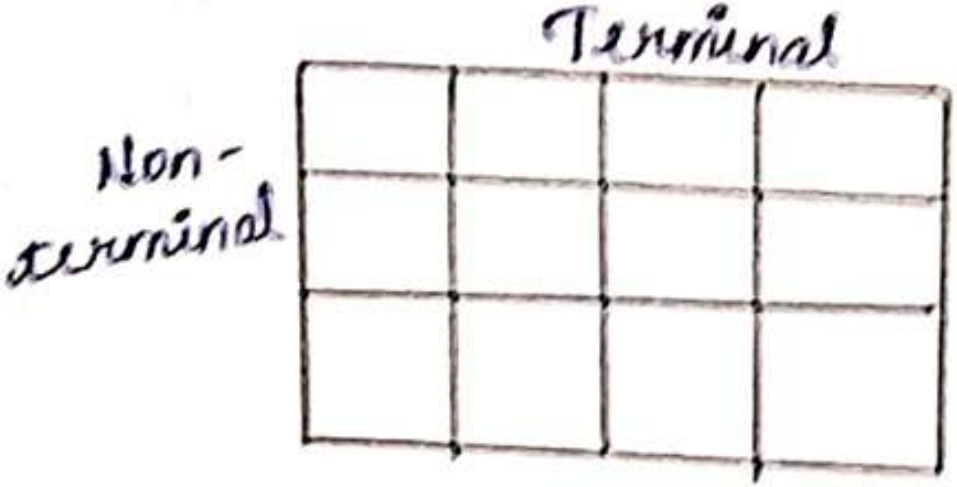
Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar –Top Down Parsing – General Strategies Recursive Descent Parser **Predictive Parser**- LL(1) Parser-Shift Reduce Parser-LR Parser- LR(0)Item Construction of SLR Parsing Table -Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC.

2. Predictive Parser (or) Non Recursive Predictive Parser (or) Table Driven Predictive parser

Predictive Parser: A predictive parser is an effective way of implementing recursive descent parsing (Top-down parsing without backtracking). By handling the stack of activation records. The model of a predictive parser is given as follows



Parsing table: It is a two-dimensional array M[A,a] where 'A' is a non-terminal & 'a' is terminal or '$'



Implementation: Predictive parser is implemented using a stack. Initially, the parser is in configuration.

Stack	Input
$S	w$

2. Predictive Parser (Cond.)

Step 1: Elimination of left recursion

Step 2: left factoring if needed

Step 3: Finding FIRST of all non-terminals.

Step 4: Find the FOLLOW of all non-terminals.

Step 5: Construction of parsing table.

Step 6: Parsing using parsing table.

Step 7: Accept or reject the string.

1. Consider the grammer $E \to E+T / T$, $T \to T*F / F$, $F \to (E) / id$

Parse the i/p
i) id + id * id,
ii) id + (id is a valid string according to the grammer using the predictive parser algorithm.

Step 1: Elimination of left recursion

$E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$

Step 2: Left factoring is not needed.

Step3 : Finding FIRST of all non-terminals

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

FIRST(E) = {(, id}

FIRST(E') = {+, ε}

FIRST(T) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

Step4 : Finding FOLLOW of all non-terminals :

HINT : Include '$' for the start symbol of grammer in & 'ε' should not be included at all.

→ Right side of the production to be considered.

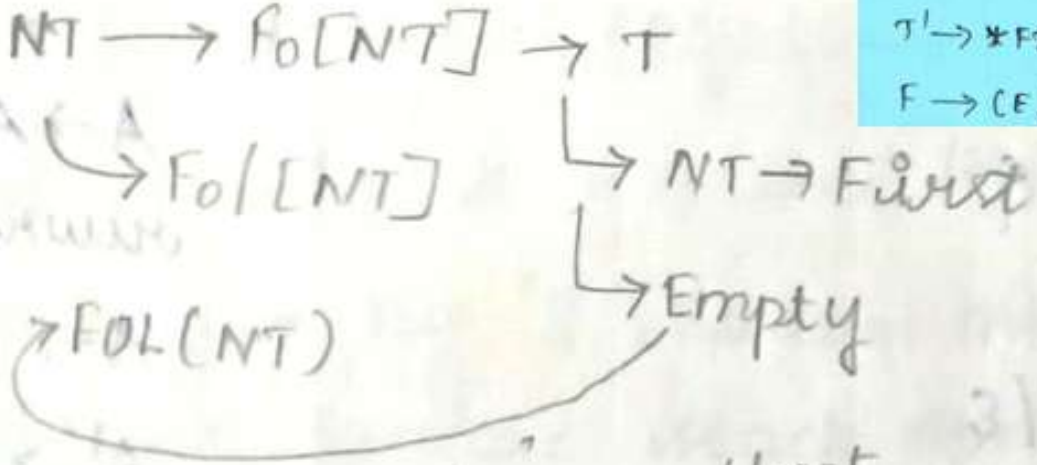
The next position of NT where FOLLOW is to be identified by 3 ways

1. The next position could a terminal, if it is so then consider the terminal for the FOLLOW position of NT.

2. If the next position is a NT, then find the FIRST of that NT (immediate FOLLOW) & also consider the FOLLOW of NT that is present at the left-hand side of the production.

3. If the next position is empty, then consider the FOLLOW of NT that is present at the left hand side of the production

$E \to TE'$	FIRST(E) = {(, id}
$E' \to +TE' / \varepsilon$	FIRST(E') = {+, ε}
$T \to FT'$	FIRST(T) = {(, id}
$T' \to *FT' / \varepsilon$	FIRST(T') = {*, ε}
$F \to (E) / id$	FIRST(F) = {(, id}



FOLLOW(E) = {), $} ← start symbol

FOLLOW(E') = {), $}

FOLLOW(T) = {+,), $}

FOLLOW(T') = {+,), $}

FOLLOW(F) = {*, +,), $}

2. Predictive Parser (Cond.)

Step 5: Constructing Parsing Table

	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Note:

For ε- productions, FOLLOW of NT should be considered

$E \rightarrow TE'$ — FIRST(E) = {(, id}

$E' \rightarrow +TE' / \varepsilon$ — FIRST(E') = {+, ε}

$T \rightarrow FT'$ — FIRST(T) = {(, id}

$T' \rightarrow *FT' / \varepsilon$ — FIRST(T') = {*, ε}

$F \rightarrow (E) / id$ — FIRST(F) = {(, id}

Step 5: Constructing Parsing Table

M

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'						
T						
T'						
F						

1) $E \rightarrow TE'$

FIRST(E) = FIRST(TE') = {(, id}

M[E, (] = $E \rightarrow TE'$

M[E, id] = $E \rightarrow TE'$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

$FIRST(E) = \{(, id\}$

$FIRST(E') = \{+, \varepsilon\}$

$FIRST(T) = \{(, id\}$

$FIRST(T') = \{*, \varepsilon\}$

$FIRST(F) = \{(, id\}$

Step 5: Constructing Parsing Table

M

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'				
T						
T'						
F						

2) $E' \rightarrow +TE'$

$FIRST(E') = FIRST(+TE') = \{+\}$

$M[E', +] = E' \rightarrow +TE'$

FOLLOW(E) = {), $}

FOLLOW(E') = {), $}

FOLLOW(T) = {+,), $}

FOLLOW(T') = {+,), $}

FOLLOW(F) = {*, +,), $}

E → TE'

E' → +TE' / ε

T → FT'

T' → *FT' / ε

F → (E) / id

FIRST(E) = {(, id}

FIRST(E') = {+, ε}

FIRST(T) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

Step 5: Constructing Parsing Table

M

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T						
T'						
F						

3) E' → ε

FOLLOW[E'] = {), $}

M[E',)] = E' → ε

M[E', $] = E' → ε

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)/id$

$FIRST(E) = \{(, id\}$

$FIRST(E') = \{+, \varepsilon\}$

$FIRST(T) = \{(, id\}$

$FIRST(T') = \{*, \varepsilon\}$

$FIRST(F) = \{(, id\}$

Step 5: Constructing Parsing Table

M

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'						
F						

4) $T \rightarrow FT'$

$FIRST(T) = FIRST(FT') = \{(, id\}$

$M[T, (] = T \rightarrow FT'$

$M[T, id] = T \rightarrow FT'$

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)/id$

FIRST(E) = {(, id}

FIRST(E') = {+, ε}

FIRST(T) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

Step 5: Constructing Parsing Table

M

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'			T'→*FT'			
F						

5) $T' \rightarrow *FT'$

FIRST [T'] = FIRST [*FT'] = {*}

M[T', *] = $T' \rightarrow *FT'$

FOLLOW(E) = {), $}

FOLLOW(E') = {), $}

FOLLOW(T) = {+,), $}

FOLLOW(T') = {+,), $}

FOLLOW(F) = {*, +,), $}

E → TE'

E' → +TE'/ε

T → FT'

T' → *FT'/ε

F → (E)/id

FIRST(E) = {(, id}

FIRST(E') = {+, ε}

FIRST(T) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

Step 5: Constructing Parsing Table

M

	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						

b) T' → ε

FOLLOW[T'] = {+,), $}

M[T', +] = T' → ε

M[T',)] = T' → ε

M[T', $] = T' → ε

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)/id$

FIRST(E) = {(, id}

FIRST(E') = {+, ε}

FIRST(T) = {(, id}

FIRST(T') = {*, ε}

FIRST(F) = {(, id}

Step 5: Constructing Parsing Table

M

	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→(E)		

7) $F \rightarrow (E)$

FIRST(F) = FIRST((E)) = {(}

M[F, (] = $F \rightarrow (E)$

$E \rightarrow TE'$ — FIRST(E) = {(, id}

$E' \rightarrow +TE'/\varepsilon$ — FIRST(E') = {+, ε}

$T \rightarrow FT'$ — FIRST(T) = {(, id}

$T' \rightarrow *FT'/\varepsilon$ — FIRST(T') = {*, ε}

$F \rightarrow (E)/id$ — FIRST(F) = {(, id}

Step 5: Constructing Parsing Table

M

	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)		

8) $F \rightarrow id$

FIRST [F] = FIRST [id] = {id}

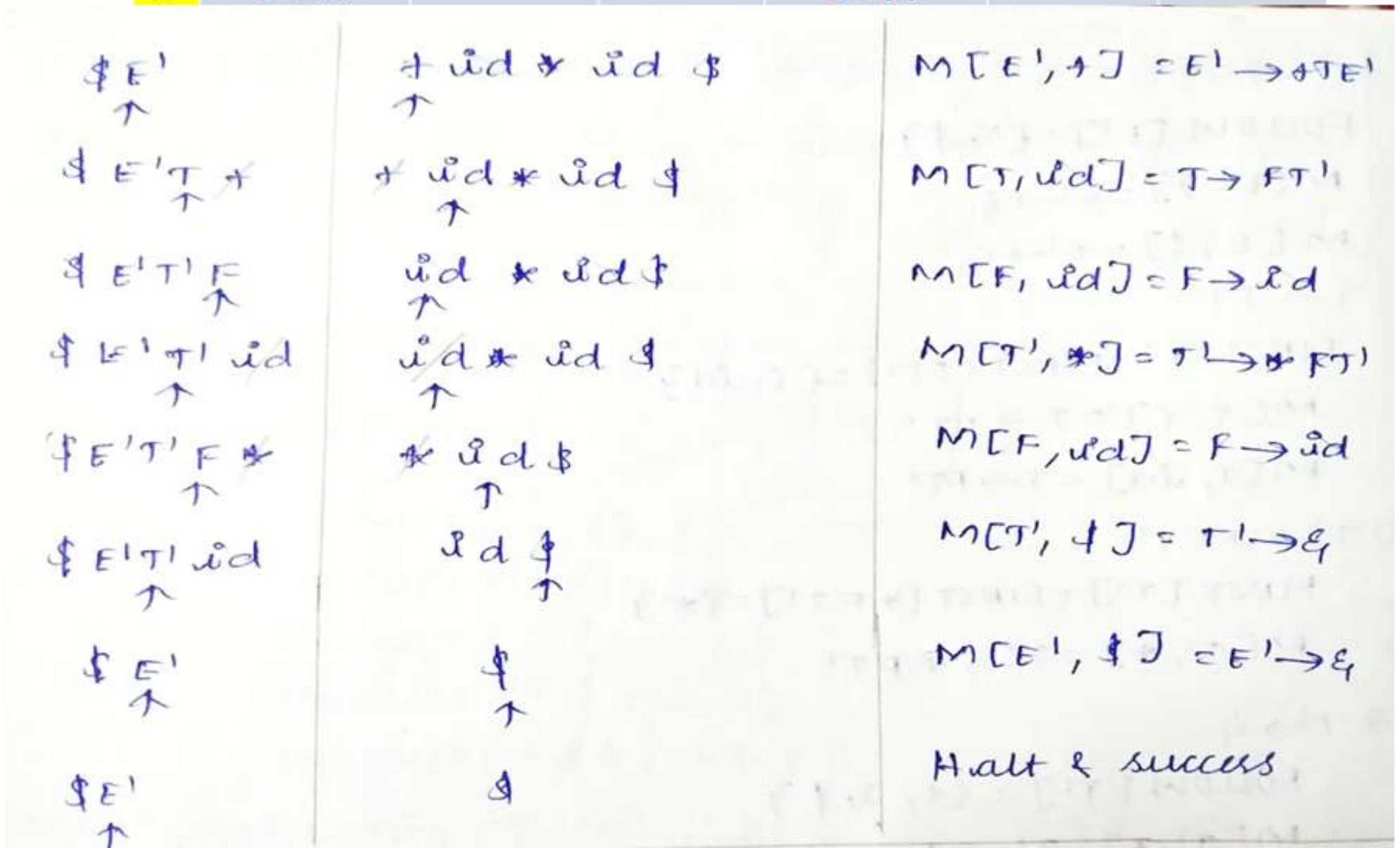
M[F, id] = $F \rightarrow id$

	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)		

Step 6: Parsing using parsing table

Stack	I/p string	Comment
$E	id + id * id $	M[E, id] = E→TE'
$E'T	id + id * id $	M[T, id] = T→FT'
$E'T'F	id + id * id $	M[F, id] = F→id
$E'T'id	id + id * id $	M[T', +] = T'→ε

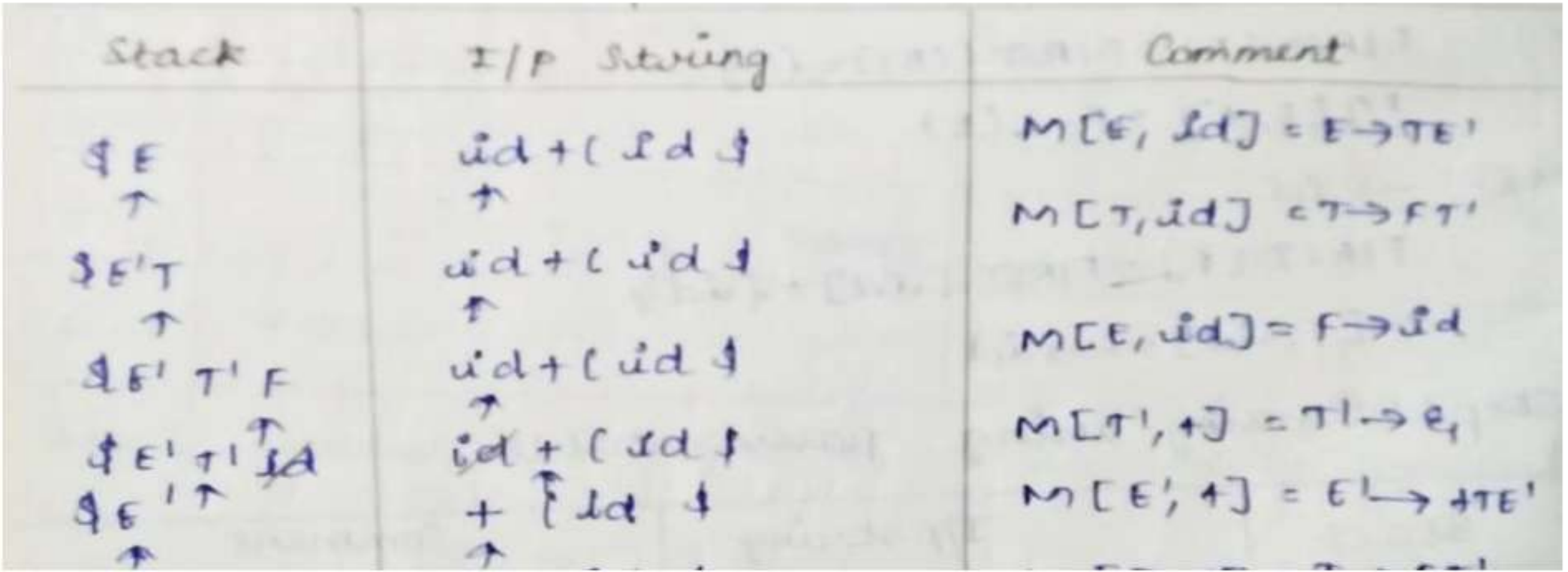
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→ id			F→(E)		



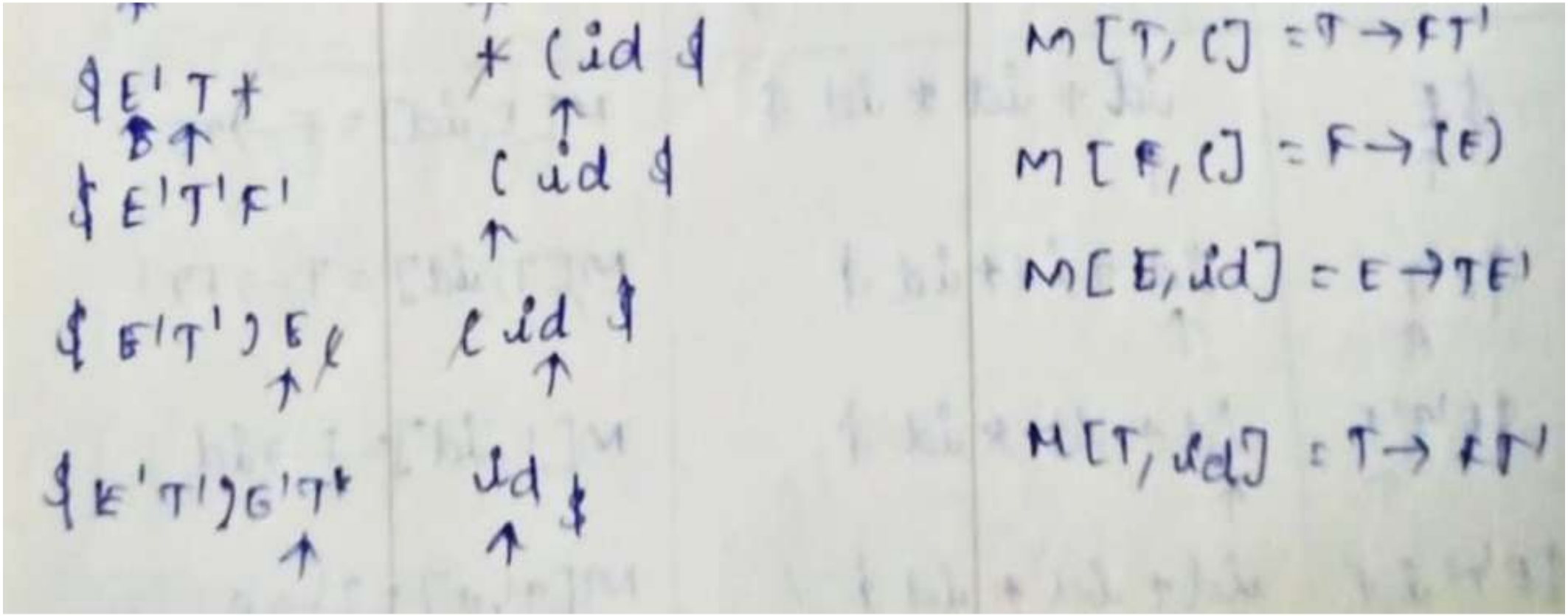
Step 7: Therefore the string **i) id + id*id** a valid string for the given grammar.

	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)		

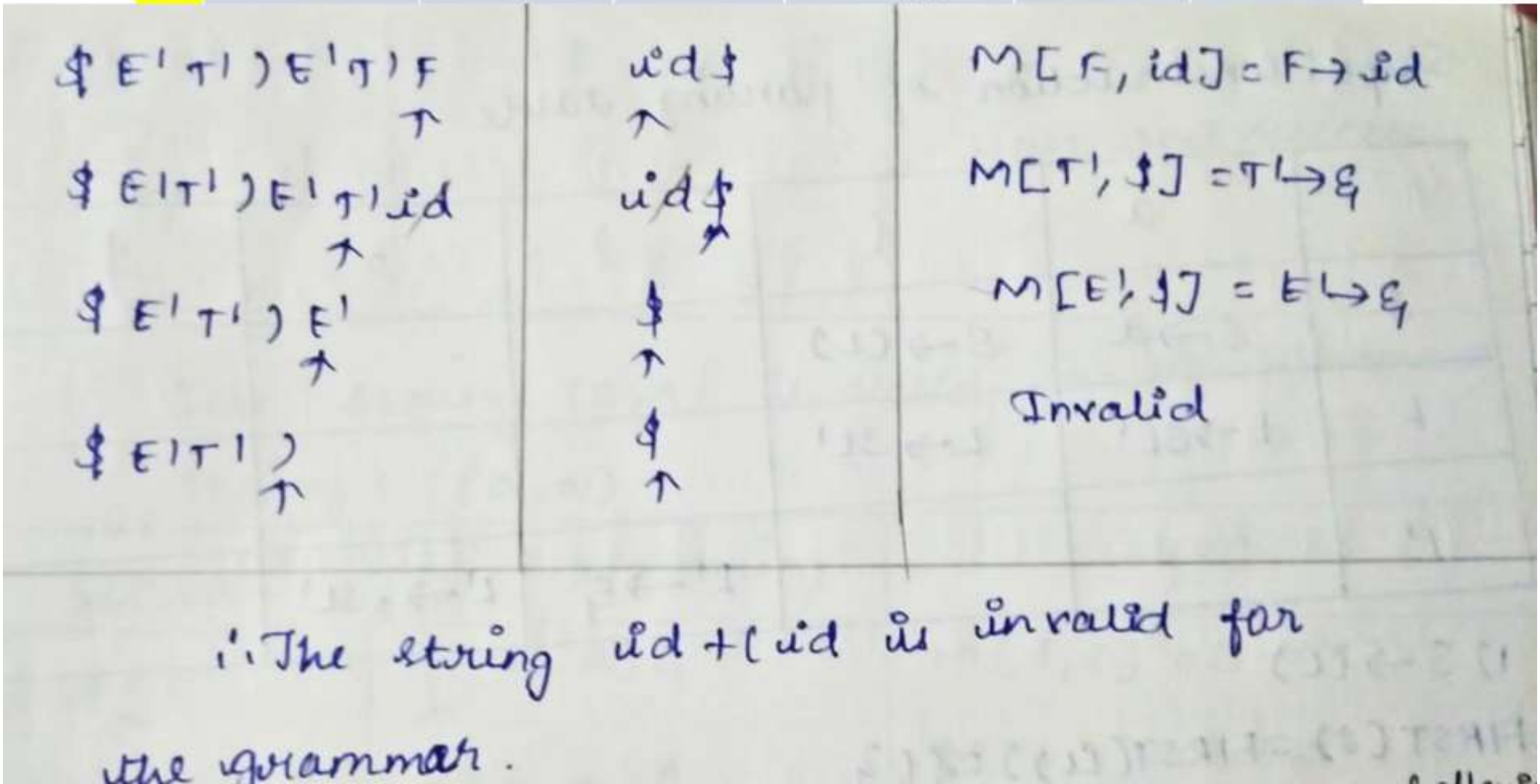
Parsing the 2nd string ii) id + (id



	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)		



	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)		



CS8602 COMPILER DESIGN

UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar –Top Down Parsing – General Strategies Recursive Descent Parser Predictive Parser- **LL(1) Parser**-Shift Reduce Parser-LR Parser- LR(0)Item Construction of SLR Parsing Table -Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC.

Parser

Accepts I/P as tokens and produces O/P as parse tree and produce syntax errors.

Types of Parser

* Top-down parser:
  - → predictive parser
  - → LL(1) parser
  - → Recursive descent parser
* Bottom-up parser
  - → shift reduce
  - → operator precedence
  - → LR parser
  - → SLR parser
  - → LALR parser
  - → CLR parser

LL(1) Parser

LL(1) : using one i/p symbol of look ahead at each step to make actions producing left most derivation scanning the i/p from left to right

The parsing table construction alg can be applied to any grammar G to produce a parsing table M. However for some grammar, M may have some entries that are multiply defined.

For eg, if the grammer G is left recursive or ambiguous then M will have atleast one multiply defined entry.

LL(1) Parser (Contd.)

Consider the grammar $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Step 1 : Elimination of left recursion

This left recursion cannot be considered at all since $A \rightarrow A\alpha \mid \beta$ is the production of or left recursion to be considered, we conclude no to go for left recursion elimination.

$S \rightarrow iEtS / \underbrace{iEtSeS}_{\alpha} / \underbrace{a}_{\beta_2}$

$E \rightarrow b$

Step 2 : Left factoring

$S \rightarrow iEtSS' / a$

$S' \rightarrow \epsilon / eS$

$A \rightarrow \alpha\beta_1 / \alpha\beta_2$

rewrite as

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2$

$S \rightarrow iEtSS' / a$

$S' \rightarrow \epsilon / eS$

$E \rightarrow b$

Step 3 : Finding FIRST of all non-terminals.

FIRST (S) = { i, a }

FIRST (S') = { ε, e }

FIRST (E) = { b }

$S \rightarrow iEtSS'/a$

$S' \rightarrow \epsilon/eS$

$E \rightarrow b$

FIRST (S) = {i, a}

FIRST (S') = {ε, e}

FIRST (E) = {b}

step 4:- Finding FOLLOW of all non-terminals

FOLLOW (S) = {e, FOLLOW(S), FOLLOW(S'), $} =

{e, $}

FOLLOW (S') = {FOLLOW(S)} = {e, $}

FOLLOW (E) = {t}

$S \rightarrow iEtSS'/a$

$S' \rightarrow \varepsilon / eS$

$E \rightarrow b$

FIRST (S) = {i, a}

FIRST (S') = {ε, e}

FIRST (E) = {b}

	i	a	e	b	t	$
S	S→iEtSS'					
S'						
E						

Step 5:-

1) $S \rightarrow iEtSS'$

FIRST (S) = FIRST (iEtSS') = {i}

M[S, i] = $S \rightarrow iEtSS'$

$S \rightarrow iEtSS'/a$

$S' \rightarrow \varepsilon / eS$

$E \rightarrow b$

FIRST (S) = {i, a}

FIRST (S') = {ε, e}

FIRST (E) = {b}

	i	a	e	b	t	$
S	S→iEtSS'	S→a				
S'						
E						

2) $S \rightarrow a$

FIRST (S) = {a}

M [S, a] = $S \rightarrow a$

$S \rightarrow iEtSS'/a$

$S' \rightarrow \varepsilon/eS$

$E \rightarrow b$

FOLLOW (S') = {FOLLOW(S)} = { e, $ }

	i	a	e	b	t	$
S	S→iEtSS'	S→a				
S'			S'→ε			S'→ε
E						

3) $S' \rightarrow \varepsilon$

FOLLOW (S') = { e, $ }

M [S', e] = $S' \rightarrow \varepsilon$

M [S', $] = $S' \rightarrow \varepsilon$

$S \rightarrow iEtSS' / a$

$S' \rightarrow \varepsilon / eS$

$E \rightarrow b$

FIRST (S) = {i, a}

FIRST (S') = {ε, e}

FIRST (E) = {b}

	i	a	e	b	t	$
S	S→iEtSS'	S→a				
S'			S'→ε S'→eS			S'→ε
E						

4) S' →es

FIRST (S')={e}

M[S',e]=S'→eS

$S \rightarrow iEtSS'/a$

$S' \rightarrow \varepsilon/eS$

$E \rightarrow b$

FIRST (S) = { i, a }

FIRST (S') = { ε, e }

FIRST (E) = { b }

	i	a	e	b	t	$
S	S→iEtSS'	S→a				
S'			S'→ε S'→eS			S'→ε
E				E→b		

5) $E \rightarrow b$

FIRST (E) = {b}

M [E, b] = $E \rightarrow b$

LL(1) Parser (Contd.)

	i	a	e	b	t	$
S	S→iEtSS'	S→a				
S'			S'→ε S'→eS			S'→ε
E				E→b		

A grammar whose parsing table has no multiple defined entries is said to be LL(1) grammar.

CS8602 COMPILER DESIGN

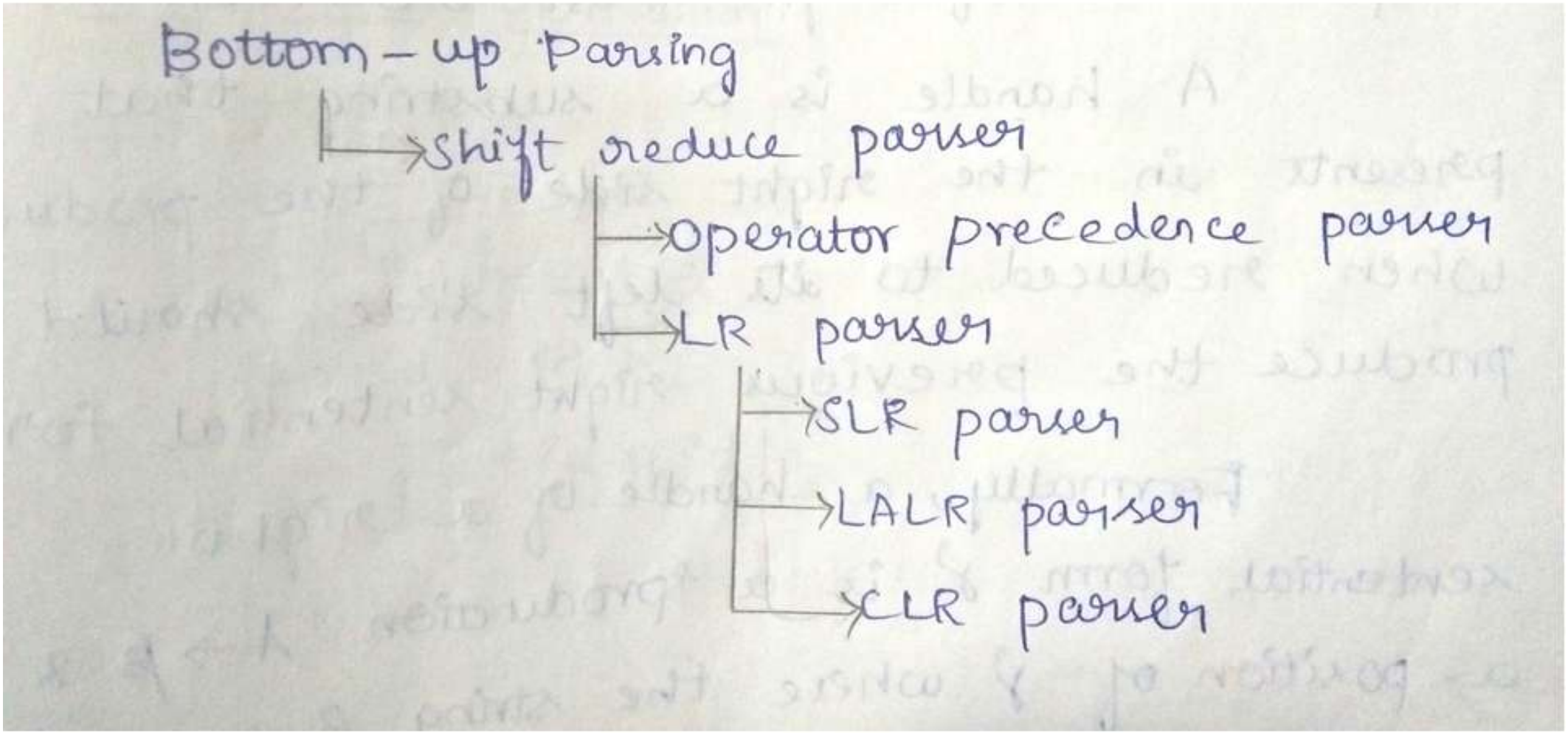
UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar –Top Down Parsing – General Strategies Recursive Descent Parser Predictive Parser- LL(1) Parser **-Shift Reduce Parser-** LR Parser- LR(0)Item Construction of SLR Parsing Table -Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC.

BOTTOM-UP PARSING

Bottom-up parsing constructs the parse tree for an input string beginning at leaves and going towards root. In other words, a bottom-up parser reduces string w to start symbol S of grammar G. A general type of bottom-up parser is shift reduce parser.

BOTTOM-UP PARSING



SHIFT REDUCE PARSING

Shift Reduce Parsing: It is a type of bottom-up parsing that constructs a parse tree for an input beginning at the leaves and working towards root. It performs reduction or right most derivation in reverse.

SHIFT REDUCE PARSING (Contd.)

1. Consider the grammar $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

The sentence to be recognised is "abbcde"

Solution:

→abbcde

→aAbcde (A→b)

→aAde (A→Abc)

→aABe (B→d)

→S (S→ aABe)

SHIFT REDUCE PARSING (Contd.)

Handle : A handle of a string is a substring that matches with the right side of the production and whose reduction to non-terminal on the left side of the production represents one step reverse of right most derivation.

SHIFT REDUCE PARSING (Contd.)

Consider the example, $E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to id$. and input string is $id_1 + id_2 * id_3$.

Right most Derivation:

$E \to E+E$

$\to E+E*E$ [$E \to E*E$]

$\to E+E*id$ [$E \to id$]

$\to E+id*id$ [$E \to id$]

$\to id+id*id$ [$E \to id$]

In this derivation, the underlined strings are handles

Consider the example, $E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$. and input string is $id_1 + id_2 * id_3$.

Input string	Handle	Reduction Production
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E * E$	$E * E$	$E \rightarrow E * E$
E		

SHIFT REDUCE PARSING (Contd.)

Stack Implementation Of Shift Reduce Parser:

We use a stack to implement shift reduce parser that hold grammar symbol. An input buffer is used to hold the string w. We use $ to mark the bottom of the stack and right end of the input.

SHIFT REDUCE PARSING (Contd.)

Step 1: The Initial configuration for the shift reduce parser is

stack	Input
$	w $

Step 2: The parser operates by shifting 0 or more input symbols onto the stack until a handle is on top of the stack.

Step 3: The parser reduces β to the left side of the appropriate production.

Step 4: The parser repeats step 2 and 3 until it gets the following configuration:

Stack	Input
$ S	$

SHIFT REDUCE PARSING (Contd.)

Step 5: If parser enters this configuration. it halts and announces successful completion of parsing, otherwise it announces error.

Eg: The sequence of actions performed by shift reduce parser for the input string id1+id2*id3 according to the grammar E → E+E / E*E / id is as follows.

$E \rightarrow E+E / E*E / id$ is as follows.

Stack	Input	Comment
$	id1+id2*id3$	shift 'id1'
$id1	+id2*id3$	E→id. Reduce 'id' to E
$E	+id2*id3$	shift '+'
$E+	id2*id3$	shift 'id2'
$E+id2	*id3$	E→id. Reduce 'id' to E
$E+E	*id3$	E→E+E. Reduce 'E+E' to E.
$E	*id3$	shift '*'
$E*	id3$	shift 'id3'
$E*id3	$	E→id. Reduce 'id' to E
$E*E	$	E→E*E. Reduce 'E*E' to E.
$E	$	Halt and success (Accept the i/p)

Conflicts during shift reduce parsing.

The general shift reduce technique

1. Perform shift action when there is no handle on the stack.
2. Perform shift action when there is no handle on TOS.

The problems of shift reduce parser

(i) Shift reduce conflict

(ii) Reduce reduce conflict

(i) Shift reduce conflict: What action to take incase both shift and reduce actions are valid.

(ii) Reduce reduce conflict: Which rule to use for reduction. if reduction is possible by more than one rule, these conflicts arises because of ambiguous grammar or the parsing method which is not powerful enough.

Shift - reduce conflict

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

Stack	Input	Comment
$	id+id*id$	shift 'id'
$id	+id*id$	Reduce E→id
$E	+id*id$	shift '+'
$E+	id*id$	shift 'id'
$E+id	*id$	Reduce E→id
$E+E	*id$	shift *
$E+E*	id$	shift 'id'
$E+E*id	$	Reduce E→id
$E+E*E	$	Reduce E→E*E
$E+E	$	Reduce E→E+E
$E	$	Accepted state

Stack	Input	Comment
$	id+id*id$	shift 'id'
$id	+id*id$	Reduce E→id
$E	+id*id$	shift '+'
$E+	id*id$	shift 'id'
$E+id	*id$	Reduce E→id
$E+E	*id$	Reduce E→E+E
$E	*id$	shift *
$E*	id$	shift 'id'
$E*id	$	Reduce E→id
$E*E	$	Reduce E→E*E
$E	$	Accepted state

Reduce - reduce conflict.

$M \rightarrow R+R \mid R+c \mid R$

$R \rightarrow c$

Input :- c+c

Stack	Input	Comment
$	c+c$	shift 'c'
$c	+c$	Reduce R→c
$R	+c$	shift '+'
$R+	c$	shift 'c'
$R+c	$	Reduce R→c
$R+R	$	Reduce M→R+R
$M	$	Accepted state

Stack	Input	Comment
$	c+c$	shift 'c'
$c	+c$	Reduce R→c
$R	+c$	shift '+'
$R+	c$	shift 'c'
$R+c	$	Reduce M→R+c
$M	$	Accepted state

Viable prefixes:

α is a viable prefix of the grammar if there is a string 'w' such that αw is a right sentential form.

The set of prefixes of right sentential form that appears on the stack of a shift reduce parser are called viable prefixes

As long as the parser has viable prefixes on the stack, no parser error has been seen.

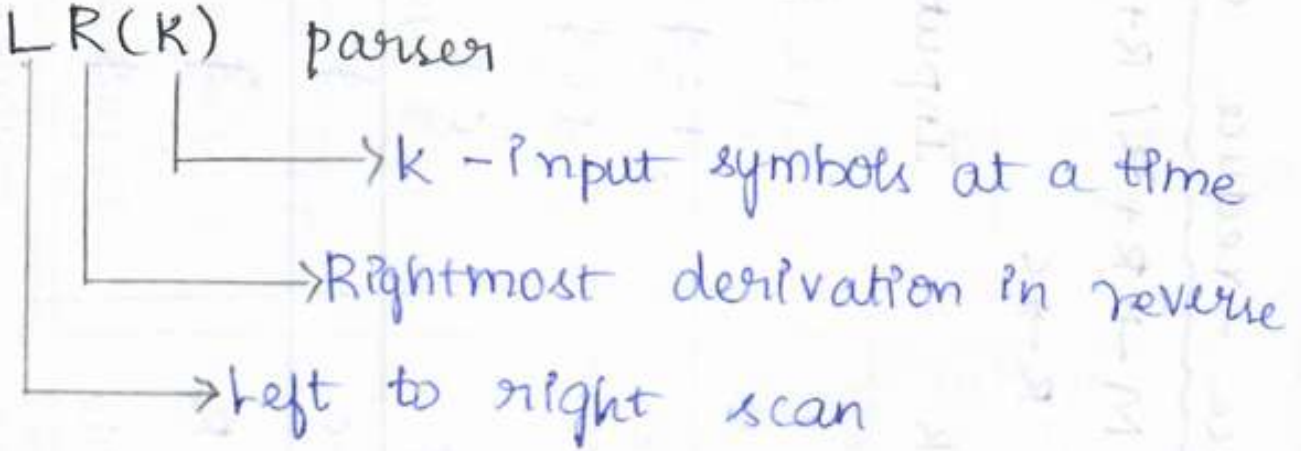
The set of viable prefixes is a regular language.

Construct an automation that accepts viable prefixes.

LR Parser:

LR parser is used for large set of CFGs.

Efficient Bottom-up syntatic analytic technique that can be used to parse a large class of CFG is called as LR(K) parser.



- If K = 1 (one input symbol at a time)
- If K is not given, one input symbol at a time (by default)

Advantages of LR parsing:

* It recognizes virtually all programming language constructs for the grammar.
* It is an efficient non-backtracking shift reduce parsing.
* It is used to construct large set of CFG.
* It is easy to identify, the syntatic error as error as possible.

Disadvantages of LR Parsing:

It is a tedious job to work in hard

It is very hard to construct LR parser for all large set of programming languages, so we need a tool for LR parsing generator which is YACC (Yet Another compiler - compiler)

Types of LR parser - 3 types.

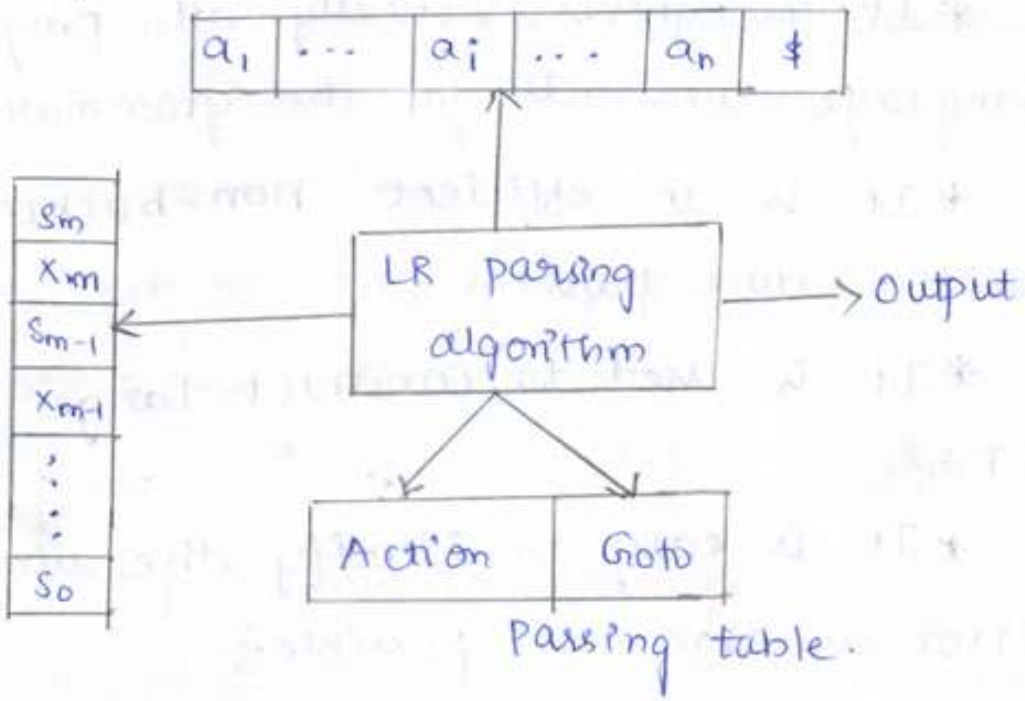
1. Simple LR (SLR)
2. LOOK Ahead LR (LALR)
3. canonical LR (CLR)

SLR - Easy to implement, least powerful, lower cost

LALR - Intermediate between SLR and CLR, easy to implement, moderate cost and moderate powerful.

CLR - Tedious to implement, most expensive, most powerful.

LR Parsing Algorithm:



Stack:

$S_0 X_1 , S_1 X_2 , S_2 X_3 , \ldots X_m S_m$ (S_m is on Top)

$S_0 , S_1 , S_2 \ldots S_m \Rightarrow$ state symbols

$X_1 , X_2 , X_3 \ldots X_m \Rightarrow$ Grammar symbols.

Parsing Table

State	Action	Goto
	Terminals	Non-terminals

Construction of SLR Parsing Table:

A simple LR or SLR parsing table is easy to construct.

A grammar for which SLR parser to be constructed is said to be SLR grammar.

2m LR(0) items: An item of a grammar is a production of G with a '.' at some position of the right side.

Eg: $A \rightarrow XYZ$

LR(0) items for this production can be written as; $A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

LR(0) items for $A \rightarrow \epsilon$ can be written as,

$A \rightarrow .$

Set of items: We group the items together into sets which gives rise to the states of SLR parser.

A collection of set of LR(0) items, we called it as canonical LR(0) items provides the basics for the construction of SLR parser.

For constructing SLR parsing table, 3 steps are followed.

(i) Augmented grammar

(ii) Closure operation

(iii) Goto operation.

Augmented Grammar:

If G is a grammar, S is the start symbol, then G' is the augmented grammar and S' is the start symbol of the augmented grammar then include the production $S' \rightarrow S$ in the given grammar for constructing augmented grammar.

Consider the grammar $E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Augmented Grammar, $E' \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

The start symbol is E, so include $E' \rightarrow E$ is the augmented grammar.

Closure Operation:

If 'I' is the set of items for a Grammar G then closure of I is constructed by 2 rules.

Rule 1: Every item in I is added to closure (I)

Rule 2: If $A \rightarrow \alpha \cdot B\beta$ is in closure (I) and $B \rightarrow \gamma$ is a production then add $B \rightarrow \cdot\gamma$ to I, if it is not already there. We apply the rule until more new items can be added to closure (I).

Eg: If $I = \{E' \to \cdot E\}$, then closure(I) can be written as,

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

Goto operation:

The function goto(I, X) where I is the set of items and X is the grammar symbol which is defined to be closure of the set of all items $A \to \alpha X \cdot \beta$ for $A \to \alpha \cdot X\beta$ is in X i.e if we have $A \to \alpha \cdot X\beta$, then goto() gives the closure $A \to \alpha X \cdot \beta$

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

Consider the example, $I: E \to .E$

$E \to .E+T$ Now

perform goto (I, E)

goto (I, E): $E \to E.$
$E \to E.+T$ } I_0

goto $(I_0, +)$: $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

Consider the state $I_0: E \to T.$

$I \to T.*F$ Perform

goto $(I_0, *)$:- $T \to T*.F$
$F \to .(E)$
$F \to .id$

Consider the grammar 1) $E \rightarrow E+T$

2) $E \rightarrow T$

3) $T \rightarrow T*F$

4) $T \rightarrow F$

5) $F \rightarrow (E)$

6) $F \rightarrow id$

Hint: Numbering the given grammar

1. Augmented grammar
2. LR(0) items
3. closure and goto
4. Transition table
5. Follow(NT)
6. Parsing table construction
7. Parsing

Step 1: Augmented Grammar.

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

Step 2: I_0: $E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

Step 3: Closure and goto

I_1: goto($\overset{\text{state}}{\downarrow}0$, E)

$E' \to E.$

$E \to E.+T$

I_2: goto(0, T)

$E \to T.$

$T \to T.*F$

I_3: goto(0, F)

$T \to F.$

I_4: goto(0, ()

$F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I_5: goto(0, id)

$F \to id.$

I_6: goto(1, +)

$E \to E+.T$

Step 4: Construct Transition Diagram

goto(0, E) = I_1

goto(0, T) = I_2

goto(0, F) = I_3

goto(0, () = I_4

goto(0, id) = I_5

goto(1, +) = I_6

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

I7: goto(2, *)
T → T * . F
F → . (E)
F → . id

I8: goto(4, E)
F → (E .)
E → E . + T

I2: goto(4, T)
E → T .
T → T . * F

I3: goto(4, F)
T → F .

I4: goto(4, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(4, id)
F → id .

I9: goto(6, T)
E → E + T .
T → T . * F

I3: goto(6, F)
T → F .

I4: goto(6, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(6, id)
F → id .

I10: goto(7, F)
T → T * F .

I4: goto(7, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(7, id)
F → id .

I11: goto(8,))
F → (E) .

I6: goto(8, +)
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I7: goto(9, *)
T → T * . F
F → . (E)
F → . id

Step 4: Construct Transition Diagram

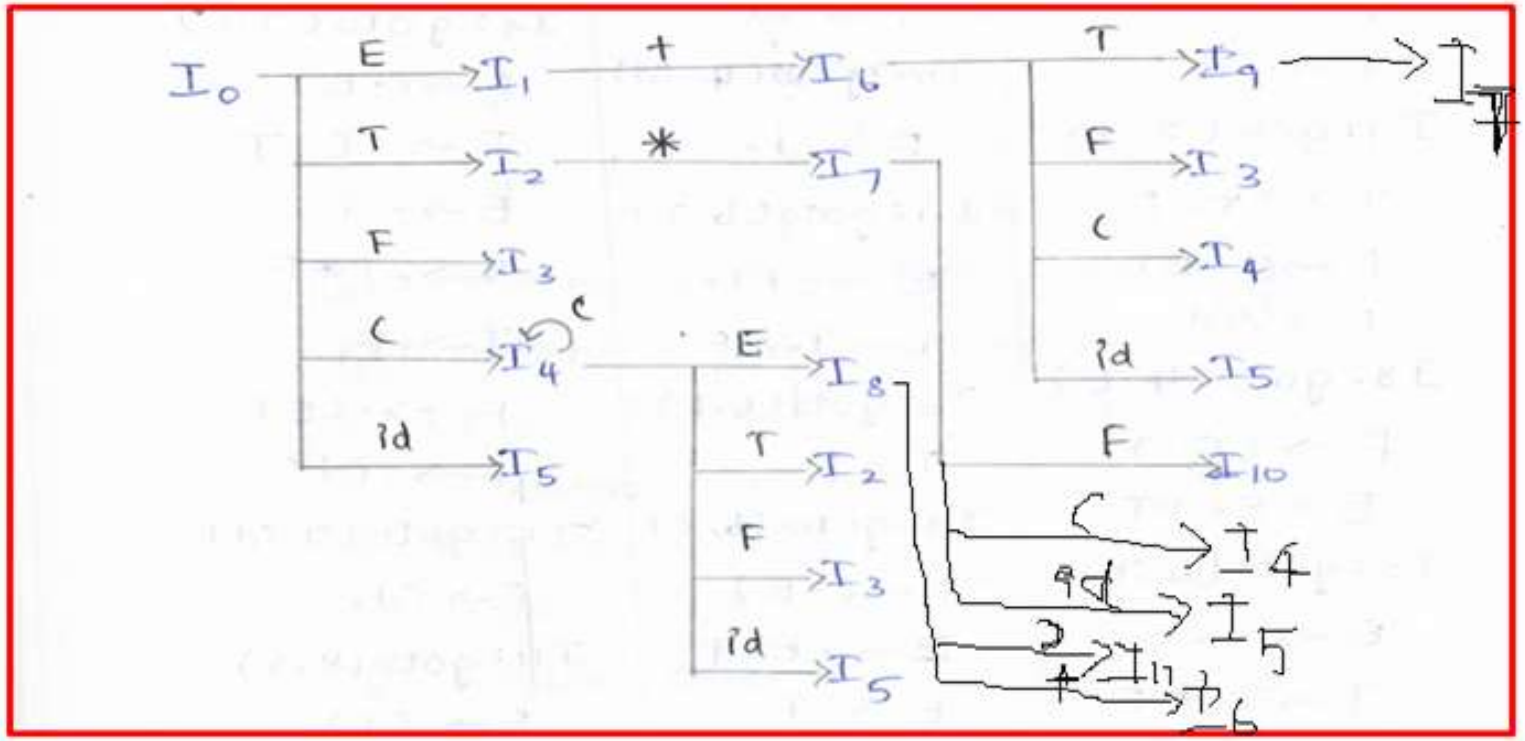
goto(2, *) = I7	goto(6, T) = I9
goto(4, E) = I8	goto(6, F) = I3
goto(4, T) = I2	goto(6, () = I4
goto(4, F) = I3	goto(6, id) = I5
goto(4, () = I4	goto(7, F) = I10
goto(4, id) = I5	goto(7, () = I4

goto(7, id) = I5	goto(8, +) = I6
goto(8,)) = I11	goto(9, *) = I7

Step 4: Construct Transition Diagram

goto(0, E) = I_1	goto(2, *) = I_7	goto(6, T) = I_9
goto(0, T) = I_2	goto(4, E) = I_8	goto(6, F) = I_3
goto(0, F) = I_3	goto(4, T) = I_2	goto(6, ( ) = I_4
goto(0, ( ) = I_4	goto(4, F) = I_3	goto(6, id) = I_5
goto(0, id) = I_5	goto(4, ( ) = I_4	goto(7, F) = I_{10}
goto(1, +) = I_6	goto(4, id) = I_5	goto(7, ( ) = I_4

goto(7, id) = I_5	goto(8, +) = I_6
goto(8,)) = I_{11}	goto(9, *) = I_7



$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Step 5: FOLLOW(E) = {+,), \$}

FOLLOW(T) = {+,), *, \$}

FOLLOW(F) = {*, +,), \$}

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1		S_6				Accept			
2	S_7	r_2		r_2		r_2			
3	r_4	r_4		r_4					
4			S_4		S_5		8	2	3
5	r_6	r_6		r_6		r_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8		S_6		S_{11}					
9	S_7	r_1		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Construction of SLR Parsing Algorithm

Constructing an SLR Parsing table

Input:

An augmented grammar G_1'

output:

The SLR Parsing table is function action and goto - for G_1'.

Method :

Step 1: Construct $C = \{I_0, I_1 \cdots I_n\}$ the collection of sets of LR (0) items for G_1'.

Step 2: state I is constructed from I_i.

The Parsing action for state i are determined as follows.

(a) if $[A \to \alpha . a\beta]$ is in I_i and Goto $[I_i, a] = I_j$ then set Action $[i, a]$ to "shift j". here 'a' must be a terminal

(b) If $[A \to \alpha .]$ is in I_i, then set Action $[i, a]$ to "reduce $A \to \alpha$" for all a is in FOLLOW(A). Here A maynot be in s'

(c) If $S' \to S$ is in I_i, then set Action $[i, \$]$ to "accept". If any conflicting actions results from the above rules, We say the grammar is not SLR(1) grammar. The algorithm fails to produce a parser in this case.

Step 6: Parsing Table:

I_0:	
goto(0,E)=I_1	goto(0,E)= 1
goto(0,T)=I_2	goto(0,T)= 2
goto(0,F)=I_3	goto(0,F)=3
goto(0,()=I_4	action(0,()=S_4
goto(0,id)=I_5	action(0,id)=S_5

I_2: FOLLOW(E)= {+,),$}	production no:
$E \to T.$	action(2,+)=r_1 (2)
goto(2,*)=I_7	action(2,))=r_2
	action(2,$)=$r_2$
	action(2,*)=S_7

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

FOLLOW(E) = {+,), $}
FOLLOW(T) = {+,), *, $}
FOLLOW(F) = {*, +,), $}

(a) if $[A \to \alpha . a\beta]$ is in I_i and Goto $[I_i, a] = I_j$ then set Action $[i, a]$ to "shift j". here 'a' must be a terminal

(b)

I_1:	
$E' \to E.$	action(1, $) = accept
goto(1, +) = I_6	action(1, +) = S_6

(c) If $S' \to S$ is in I_i, then set Action $[i, \$]$ to "accept". If any conflicting actions results from the above rules, we say the grammar is not SLR(1) grammar. The algorithm fails to produce a parser in this case.

Step 6: Parsing Table:

I_0:	
goto(0, E) = I_1	goto(0, E) = 1
goto(0, T) = I_2	goto(0, T) = 2
goto(0, F) = I_3	goto(0, F) = 3
goto(0, () = I_4	action(0, () = S_4
goto(0, id) = I_5	action(0, id) = S_5

I_2: FOLLOW(E) = {+,), $}	production no:
$E \to T.$	action(2, +) = r_2 (2)
goto(2, *) = I_7	action(2,)) = r_2
	action(2, $) = r_2
	action(2, *) = S_7

1) $E \to E+T$
2) $E \to T$
3) $T \to T*F$
4) $T \to F$
5) $F \to (E)$
6) $F \to id$

FOLLOW(E) = {+,), $}
FOLLOW(T) = {+,), *, $}
FOLLOW(F) = {*, +,), $}

I_0:	
goto(0,E)=I_1	goto(0,E)= 1
Goto(0,T)=I_2	goto(0,T)= 2
goto(0,F)=I_3	goto(0,F)=3
goto(0,()=I_4	action(0,()=S_4
goto(0,id)=I_5	action(0,id)=S_5

Step 3:- The goto transition
are constructed for all n
A using the rule.

If GOTO $(I_i, A) = I_j$, then GOTO $(i, A) = j$

Step 4:- All entries not defined by rules 2 and 3 are errors.

Step 5:- The initial state of parser is the one constructed from the set of items containing $S' \to .S$

Step 6: Parsing Table:

I_0:	
$goto(0,E) = I_1$	$goto(0,E) = 1$
$goto(0,T) = I_2$	$goto(0,T) = 2$
$goto(0,F) = I_3$	$goto(0,F) = 3$
$goto(0,() = I_4$	$action(0,() = S_4$
$goto(0,id) = I_5$	$action(0,id) = S_5$

I_1:	
$E' \to E.$	$action(1,\$) = accept$
$goto(1,+) = I_6$	$action(1,+) = S_6$

I_2: (FOLLOW(E) = {+,), $})	production no:
$E \to T.$	$action(2,+) = r_2$
$goto(2,*) = I_7$	$action(2,)) = r_2$
	$action(2,\$) = r_2$
	$action(2,*) = S_7$

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1		S_6				Accept			
2	S_7	r_2		r_2		r_2			

Step 3: Closure and goto

I1: goto(0, E)
E' → E.
E → E. + T

I2: goto(0, T)
E → T.
T → T. * F

I3: goto(0, F)
T → F.

I4: goto(0, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(0, id)
F → id.

I6: goto(1, +)
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I7: goto(2, *)
T → T * . F
F → . (E)
F → . id

I8: goto(4, E)
F → (E .)
E → E . + T

I9: goto(6, T)
E → E + T .
T → T . * F

I10: goto(7, F)
T → T * F .

goto(0, E) = I1	goto(2, *) = I7	goto(6, T) = I9
goto(0, T) = I2	goto(4, E) = I8	goto(6, F) = I3
goto(0, F) = I3	goto(4, T) = I2	goto(6, () = I4
goto(0, () = I4	goto(4, F) = I3	goto(6, id) = I5
goto(0, id) = I5	goto(4, () = I4	goto(7, F) = I10
goto(1, +) = I6	goto(4, id) = I5	goto(7, () = I4

goto(7, id) = I5	goto(8, +) = I6
goto(8,)) = I11	goto(9, *) = I7

1) E → E + T
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

FOLLOW(E) = { +,), $ }
FOLLOW(T) = { +,), *, $ }
FOLLOW(F) = { *, +,), $ }

I3: goto(6, F)
T → F.

I4: goto(6, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(6, id)
F → id.

I11: goto(8,))
F → (E).

I6: goto(8, +)
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I7: goto(9, *)
T → T * . F
F → . (E)
F → . id

I_3: → FOLLOW(T) = {*, +,), $}	
T → F.	action(3, *) = r_4 action(3, +) = r_4 action(3,)) = r_4 action(3, $) = r_4

I_4:	
goto(4, E) = I_8	goto(4, E) = 8
goto(4, T) = I_2	goto(4, T) = 2
goto(4, F) = I_3	goto(4, F) = 3
goto(4, () = I_4	action(4, () = S_4
goto(4, id) = I_5	action(4, id) = S_5

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1		S_6				Accept			
2	S_7	r_2		r_2		r_2			
3	r_4	r_4		r_4					
4			S_4		S_5		8	2	3

Step 3: Closure and goto

I1: goto(0, E)
E' → E.
E → E. + T

I2: goto(0, T)
E → T.
T → T. * F

I3: goto(0, F)
T → F.

I4: goto(0, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(0, id)
F → id.

I6: goto(1, +)
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I7: goto(2, *)
T → T * . F
F → . (E)
F → . id

I8: goto(4, E)
F → (E .)
E → E . + T

I9: goto(6, T)
E → E + T .
T → T . * F

I10: goto(7, F)
T → T * F .

goto(0, E) = I1	goto(2, *) = I7	goto(6, T) = I9
goto(0, T) = I2	goto(4, E) = I8	goto(6, F) = I3
goto(0, F) = I3	goto(4, T) = I2	goto(6, () = I4
goto(0, () = I4	goto(4, F) = I3	goto(6, id) = I5
goto(0, id) = I5	goto(4, () = I4	goto(7, F) = I10
goto(1, +) = I6	goto(4, id) = I5	goto(7, () = I4

goto(7, id) = I5	goto(8, +) = I6
goto(8,)) = I11	goto(9, *) = I7

1) E → E + T
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

FOLLOW(E) = { +,), $ }
FOLLOW(T) = { +,), *, $ }
FOLLOW(F) = { *, +,), $ }

I_5: FOLLOW(F) = {+, *,), $}	
F → id.	action(5, *) = r_6 action(5, +) = r_6 action(5,)) = r_6 action(5, $) = r_6

I_6:	
goto(6, T) = I_9	goto(6, T) = 9
goto(6, F) = I_3	goto(6, F) = 3
goto(6, () = I_7	action(6, () = S_4
goto(6, id) = I_5	action(6, id) = S_5

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1		S_6				Accept			
2	S_7	r_2		r_2		r_2			
3	r_4	r_4		r_4					
4			S_4		S_5		8	2	3
5	r_6	r_6		r_6		r_6			
6			S_4		S_5			9	3

Step 3: Closure and goto

goto

I1: goto(0, E)

E' → E.

E → E. + T

I2: goto(0, T)

E → T.

T → T. * F

I3: goto(0, F)

T → F.

I4: goto(0, ()

F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

I5: goto(0, id)

F → id.

I6: goto(1, +)

E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

I7: goto(2, *)

T → T * . F

F → . (E)

F → . id

I8: goto(4, E)

F → (E .)

E → E . + T

I9: goto(6, T)

E → E + T .

T → T . * F

I10: goto(7, F)

T → T * F .

goto(0, E) = I1	goto(2, *) = I7	goto(6, T) = I9
goto(0, T) = I2	goto(4, E) = I8	goto(6, F) = I3
goto(0, F) = I3	goto(4, T) = I2	goto(6, () = I4
goto(0, () = I4	goto(4, F) = I3	goto(6, id) = I5
goto(0, id) = I5	goto(4, () = I4	goto(7, F) = I10
goto(1, +) = I6	goto(4, id) = I5	goto(7, () = I4

goto(7, id) = I5	goto(8, +) = I6
goto(8,)) = I11	goto(9, *) = I7

1) E → E + T
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

FOLLOW(E) = { +,), $ }

FOLLOW(T) = { +,), *, $ }

FOLLOW(F) = { *, +,), $ }

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
5	r_6	r_6		r_6		r_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8		S_6		S_{11}					
9	S_7	r_1		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

I_7:	
goto(7, F) = I_{10}	goto(7, F) = 10
goto(7, ( ) = I_4	goto(7, ( ) = 4
goto(7, id) = I_5	action(7, id) = S_5

I_8:	
goto(8,)) = I_{11}	action(8,)) = S_{11}
goto(8, +) = I_6	action(8, +) = S_6

Step 3: Closure and goto

I1: goto(0, E)
E' → E.
E → E. + T

I2: goto(0, T)
E → T.
T → T. * F

I3: goto(0, F)
T → F.

I4: goto(0, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(0, id)
F → id.

I6: goto(1, +)
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I7: goto(2, *)
T → T * . F
F → . (E)
F → . id

I8: goto(4, E)
F → (E .)
E → E . + T

I9: goto(6, T)
E → E + T .
T → T . * F

I10: goto(7, F)
T → T * F .

goto(0, E) = I1	goto(2, *) = I7	goto(6, T) = I9
goto(0, T) = I2	goto(4, E) = I8	goto(6, F) = I3
goto(0, F) = I3	goto(4, T) = I2	goto(6, () = I4
goto(0, () = I4	goto(4, F) = I3	goto(6, id) = I5
goto(0, id) = I5	goto(4, () = I4	goto(7, F) = I10
goto(1, +) = I6	goto(4, id) = I5	goto(7, () = I4

goto(7, id) = I5	goto(8, +) = I6
goto(8,)) = I11	goto(9, *) = I7

1) E → E + T
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

FOLLOW(E) = { +,), $ }
FOLLOW(T) = { +,), *, $ }
FOLLOW(F) = { *, +,), $ }

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3

I_9: FOLLOW(E) = {+,), $}

$E \to E+T.$

goto(9, *) = I_7

action(9, +) = r_1

action(9,)) = r_1

action(9, $) = r_1

action(9, *) = S_7

I_{10}: FOLLOW(T) = {*, +,), $}

$T \to T*F.$

action(10, *) = r_3

action(10, +) = r_3

action(10,)) = r_3

action(10, $) = r_3

6			S_4		S_5			9	3
7			S_4		S_5				10
8		S_6		S_{11}					
9	S_7	r_1		r_1		r_1			
10	r_3	r_3		r_3		r_3			

Step 3: Closure and goto

I1: goto(0, E)
E' → E.
E → E. + T

I2: goto(0, T)
E → T.
T → T. * F

I3: goto(0, F)
T → F.

I4: goto(0, ()
F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

I5: goto(0, id)
F → id.

I6: goto(1, +)
E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

I7: goto(2, *)
T → T * . F
F → . (E)
F → . id

I8: goto(4, E)
F → (E .)
E → E . + T

I9: goto(6, T)
E → E + T .
T → T . * F

I10: goto(7, F)
T → T * F .

goto(0, E) = I1	goto(2, *) = I7	goto(6, T) = I9
goto(0, T) = I2	goto(4, E) = I8	goto(6, F) = I3
goto(0, F) = I3	goto(4, T) = I2	goto(6, () = I4
goto(0, () = I4	goto(4, F) = I3	goto(6, id) = I5
goto(0, id) = I5	goto(4, () = I4	goto(7, F) = I10
goto(1, +) = I6	goto(4, id) = I5	goto(7, () = I4

goto(7, id) = I5	goto(8, +) = I6
goto(8,)) = I11	goto(9, *) = I7

1) E → E + T
2) E → T
3) T → T * F
4) T → F
5) F → (E)
6) F → id

FOLLOW(E) = { +,), $ }
FOLLOW(T) = { +,), *, $ }
FOLLOW(F) = { *, +,), $ }

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1									
2									
3									
4									3
5									
6									3
7			S_4		S_5				10
8		S_6		S_{11}					
9	S_7	r_1		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

I_{11}: FOLLOW(F) = {*, +,), $}

$F \rightarrow (E).$

action(11, *) = r_5

action(11, +) = r_5

action(11,)) = r_5

action(11, $) = r_5

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1		S_6				Accept			
2	S_7	r_2		r_2		r_2			
3	r_4	r_4		r_4					
4			S_4		S_5		8	2	3
5	r_6	r_6		r_6		r_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8		S_6		S_{11}					
9	S_7	r_1		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

Example 1 2 2

Step 6: Tracing the inputs

Stack	Input	Comment
0	id + id * id $	Action [0, id] = S5
0 id 5	+ id * id $	Reduce F → id
0 F 3	+ id * id $	Reduce T → F

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S4		S5		1	2	3
1		S6				Accept			
2	S7	r2		r2		r2			
3	r4	r4		r4					
4			S4		S5		8	2	3
5	r6	r6		r6		r6			

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

Example 1 & 2

Step 6: Tracing the inputs

Stack	Input	Comment
0	id + id * id $	Action [0, id] = S5
0 id 5	+ id * id $	Reduce F → id
0 F 3	+ id * id $	Reduce T → F
0 T 2	+ id * id $	Reduce E → T
0 E 1	+ id * id $	Action [1, +] = S6
0 E 1 + 6	id * id $	Action [6, id] = S5
0 E 1 + 6 id 5	* id $	Reduce F → id
0 E 1 + 6 F 3	* id $	Reduce T → F
0 E 1 + 6 T 9	* id $	Action [9, *] = S7
0 E 1 + 6 T 9 * 7	id $	Action [7, id] = S5
0 E 1 + 6 T 9 * 7 id 5	$	Reduce F → id
0 E 1 + 6 T 9 * 7 10	$	Reduce T → T*F
0 E 1 + 6 T 9	$	Reduce E → E+T
0 E 1	$	Accept

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1		S_6				Accept			
2	S_7	r_2		r_2		r_2			
3	r_4	r_4		r_4					
4			S_4		S_5		8	2	3
5	r_6	r_6		r_6		r_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8		S_6		S_{11}					
9	S_7	r_1		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

ii) id + id * (id

Stack	Input	Comment
0	id + id * (id $	Action [0, id] = S5

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

State	Action						Goto		
	*	+	(	)	id	$	E	T	F
0			S4		S5		1	2	3
1		S6				Accept			
2	S7	r2		r2		r2			
3	r4	r4		r4					
4			S4		S5		8	2	3
5	r6	r6		r6		r6			

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

0id5	+id * (id $	Reduce F→id
0F3	+id * (id $	Reduce T→F
0T2	+id * (id $	Reduce E→T
0E1	+id * (id $	Action [1,+] = S6
0E1+6	id * (id $	Action [6, id] = S5
0E1+6id5	* (id $	Reduce F→id
0E1+6F3	* (id $	Reduce T→F
0E1+6T9	* (id $	Action [9,*] = S7
0E1+6T9*7	(id $	Action [7,(] = S4
0E1+6T9*7(4	id $	Action [4, id] = S5
0E1+6T9*7(4id5	$	Reduce F→id
0E1+6T9*7(4F3	$	Reduce T→F
0E1+6T9*7(4T2	$	Reduce E→T
0E1+6T9*7(4E8	$	Error

State	Action								
	*	+	(	)	id	$	E	T	F
0			S4		S5		1	2	3
1		S6				Accept			
2	S7	r2		r2		r2			
3	r4	r4		r4					
4			S4		S5		8	2	3
5	r6	r6		r6		r6			
6			S4		S5			9	3
7			S4		S5				10
8		S6		S11					
9	S7	r1		r1		r1			
10	r3	r3		r3		r3			
11	r5	r5		r5		r5			

CS8602 COMPILER DESIGN

UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar –Top Down Parsing – General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table -**Introduction to LALR Parser** – Error Handling and Recovery in Syntax Analyzer-YACC.

CLR

canonical LR: (CLR)

The action and goto functions from the set of LR(1) items are same as such of SLR, but with some i/p symbols.

CLR

1. Augmented grammar
2. LR(0) items
3. closure and goto
4. Transition table
5. Follow(NT)
6. Parsing table construction
7. Parsing

Canonical LR (CLR)

- Consider the grammar

S→CC
C→cC
C→d

Note: Number the given grammar
1) S→CC
2)C→cC
3) C→d

Step1: Augmented Grammar
S'→S
S→CC
C→cC
C→d

Step 2: LR(0) items

S'→.S,

S→.CC,

C→.cC,

C→.d,

Note: Need to include input symbols

Example: Augmented Grammar,

S'→S

S→CC

C→cC

C→d

A→ $\alpha . B\beta$,a

B→ . γ,b

↓

FIRST(βa)

To find goto and closure

S'→.S , $

S→.CC, ?

C→.cC, ?

C→.d, ?

Example:

S'→S

S→CC

C→cC

C→d

A→ $\alpha.B\beta$,a

B→.γ,b

↓

FIRST(βa)

To find goto and closure

S'→.S , $

S→.CC, ?(have to write FIRST(βa)

C→.cC, ?

C→.d, ?

To find goto and closure

S'→ .S , $

A→ $\alpha.B\beta$,a

Here

$\alpha\rightarrow\varepsilon$

B→S

$\beta\rightarrow\varepsilon$

a→ $

FIRST(βa)=>FIRST(ε, $)

=>$

S→.CC, $

C→.cC, ?

C→.d, ?

Example:

S'→S

S→CC

C→cC

C→d

$A \rightarrow \alpha . B\beta, a$

$B \rightarrow . \gamma, b$

↓

FIRST(βa)

To find goto and closure

S'→.S , $

S→.CC, $

C→.cC, ? (have to write FIRST(βa)

C→.d, ? (have to write FIRST(βa)

To find goto and closure

S'→ .S , $

$A \rightarrow \alpha . B\beta, a$

Here

$\alpha \rightarrow \varepsilon$

$B \rightarrow S$

$\beta \rightarrow \varepsilon$

a→ $

FIRST(βa)=>FIRST(ε, $)

=>$

S→ .C C,$

$A \rightarrow \alpha . B\beta, a$

Here

$\alpha \rightarrow \varepsilon$

$B \rightarrow C$

$\beta \rightarrow C$

a→ $

FIRST(βa)=>FIRST(C, $)

=> FIRST(C) ⇒ c/d

=> (c/d,$)

C→.cC, c/d

C→.d, c/d

Step 3 : Find the set of items based on the **goto** and **closure** operation

I_0

$A \rightarrow \alpha \cdot B\beta, a$

$S' \rightarrow \cdot S, \$$

$S \rightarrow \cdot CC, \$$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

I_1: Goto (0, S)

$S' \rightarrow S \cdot, \$$

I_2: Goto (0, C)

$S \rightarrow C \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

I_3: Goto (0, c)

$C \rightarrow c \cdot C, c/d$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

I_4: Goto (0, d)

$C \rightarrow d \cdot, c/d$

I_5: Goto (2, C)

$S \rightarrow CC \cdot, \$$

I_6: Goto (2, c)

$C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

$A \rightarrow \alpha . B\beta, a$

$B \rightarrow . \gamma, b$

↓

FIRST(βa)

I_1: Goto (2, d)

C → d · $

I_8: Goto (3, C)

C → cC · , c/d

I_3: Goto (3, c)

C → c · C, c/d

C → · cC, c/d

C → · d , c/d

I_4: Goto (3, d)

C → d · , c/d

I_9: Goto (6, C)

C → cC · , $

I_6: Goto (6, c)

C → c · C, $

C → · cC, $

C → · d , $

I_7: Goto (6, d)

C → d · , $

Goto (0, S) = I_1

Goto (0, C) = I_2

Goto (0, c) = I_3

Goto (0, d) = I_4

Goto (2, c) = I_5

Goto (2, c) = I_6

Goto (2, d) = I_7

Goto (3, C) = I_8

Goto (3, c) = I_3

Goto (3, d) = I_4

Goto (6, C) = I_9

Goto (6, c) = I_6

Goto (6, d) = I_7

$A \rightarrow \alpha . B\beta, a$

$B \rightarrow . \gamma, b$

↓

FIRST(βa)

Step 4: Draw Transition Diagram

Goto (0, S) = I_1

Goto (0, C) = I_2

Goto (0, c) = I_3

Goto (0, d) = I_4

Goto (2, c) = I_5

Goto (2, c) = I_6

Goto (2, d) = I_7

Goto (3, C) = I_8

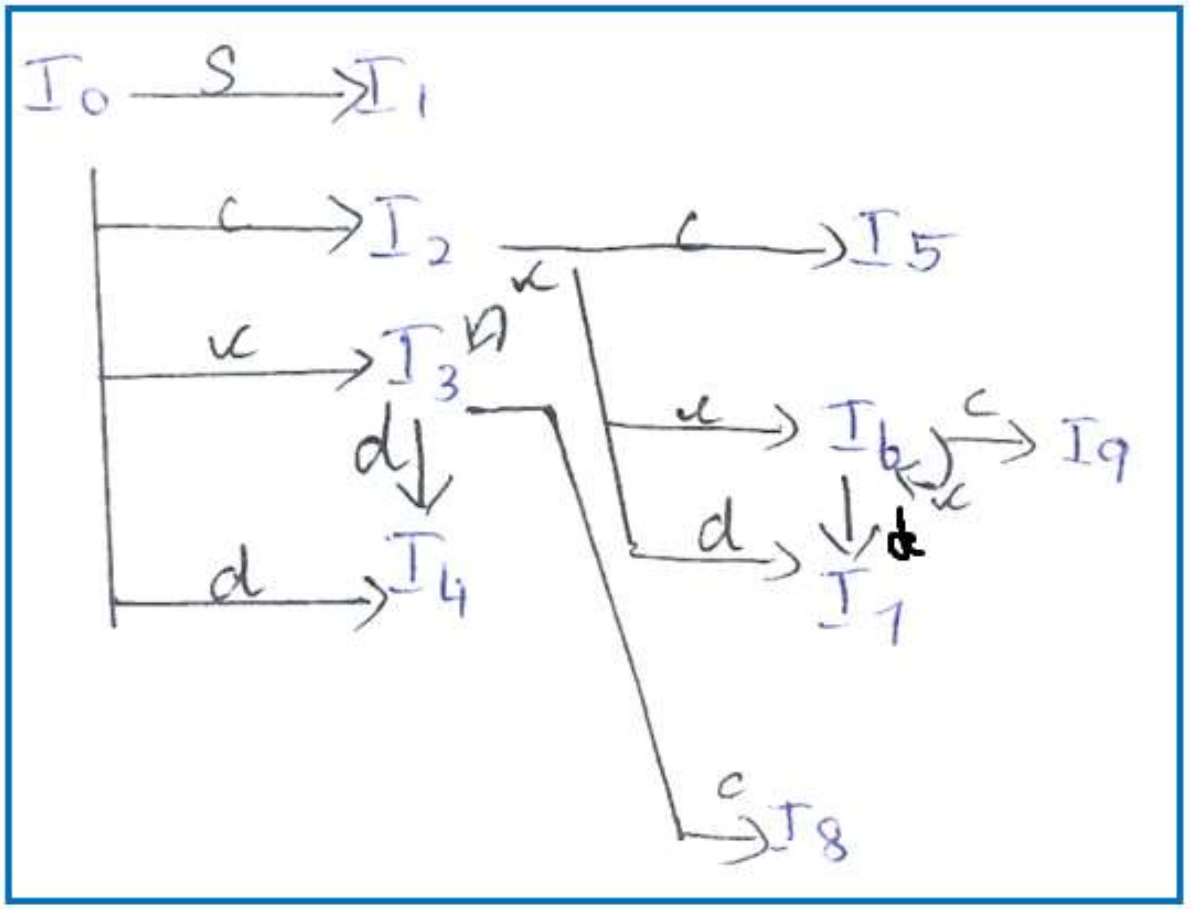
Goto (3, c) = I_3

Goto (3, d) = I_4

Goto (6, C) = I_9

Goto (6, c) = I_6

Goto (6, d) = I_7



I_0 Goto (0,S) = I_1 Goto (0,C) = I_2 Goto (0, c) = I_3 Goto (0, d) = I_4	Goto (0,S) = 1 Goto (0,C) = 2 ACTION (0, c) = S_3 ACTION (0, d) = S_4
I_1 $S' \rightarrow S.$, \$	ACTION [1, \$] = Acc
I_2 Goto (2,C) = I_5 Goto (2, c) = I_6 Goto (2, d) = I_7	Goto (2, C) = 5 ACTION [2, c) = S_6 ACTION (2, d) = S_7
I_3 Goto (3,C) = I_8 Goto (3, c) = I_3 Goto (3, d) = I_4	Goto (3, C) = 8 Goto (3, c) = S_3 Goto (3, d) = S_4
I_4 FOLLOW (C) => { c, d, \$} $C \rightarrow d.$, c / d	ACTION (4, c) = $\gamma 3$ ACTION (4, d) = $\gamma 3$
I_5 {\$} $S \rightarrow CC.$, \$	ACTION (5, \$) = $\gamma 1$

Step 5: Construct Parsing Table

States	Actions			Goto	
	c	d	\$	S	C
0	S_3	S_4		1	2
1			Acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	$\gamma 3$	$\gamma 3$			
5			$\gamma 1$		
6	S_6	S_7			9
7			$\gamma 3$		
8	$\gamma 2$	$\gamma 2$			
9			$\gamma 2$		

1) S → CC
2) C → cC
3) C → d

FOLLOW (S) = {\$}
FOLLOW (C) = {\$, c, d}

Step 5: Construct Parsing Table

I_6 Goto (6, C) = I_9 Goto (6, c) = I_6 Goto (6, d) = I_7	Goto (6, C) = 9 Goto (6, c) = S_6 Goto (6, d) = S_7
I_7: {c, d, \$} C -> d·, \$	ACTION (7, \$) = γ_3
I_8: {c, d, \$} C -> cC·, c/d	ACTION (8, C) = γ_2 ACTION (8, d) = γ_2
I_9: {c, d, \$} C -> cC·, \$	ACTION (9, \$) = γ_2

States	Actions			Goto	
	c	d	\$	S	C
0	S_3	S_4		1	2
1			ACC		
2	S_6	S_7			5
3	S_3	S_4			8
4	γ_3	γ_3			
5			γ_1		
6	S_6	S_7			9
7			γ_3		
8	γ_2	γ_2			
9			γ_2		

1) S→CC
2)C→cC
3) C→d

FOLLOW(S) = {\$}
FOLLOW(C) = {\$, c, d}

Step 6: Parsing the input string "dcd"

stack	i/p	comment
0	dcd $	Action (0, d) = S_4
0d4	cd $	Action (4, c) = r_3
0C2	cd $	r_3 C → d Action (2, c) = S_6
0C2c6	d$	S_6 ACTION (6, d) = S_7
0C2c6d7	$	Action (7, $) = r_3
00C6C9	$	r_3 C → d Action (9, $) = r_2
0C2C5	$	r_2 C → CC Action (5, $) = r_1
0S1	$	r_1 S → CC Action (1, $) = Acc
		Accept

1) S → CC
2) C → cC
3) C → d

states	Actions			Goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			Acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

LOOK AHEAD LR [LALR]

LALR

Hint :-

Solve the sum using canonical LR method, by seeing similarities of states, combine them together so that LALR Parsing table is constructed.

The LALR Parsing table is constructed as

(a) If I is the union of one or more sets of LR(1) items (i e) $J = \{ I_1 \cup I_2 \cup \ldots \cup I_K \}$ then the cores of GOTO (I, x), (i e) GOTO (I_1, x), GOTO (I_2, x) ... GOTO (I_K, x) are same since $\{I_1, I_2 \ldots I_K\}$ all have the same core

Let K be the union of all sets of Items having the same core as GOTO (I_1, x) then GOTO $(I, x) = K$.

Step 3 : Find the set of items based on the **goto** and **closure** operation

I_0

$A \rightarrow \alpha \cdot B\beta, a$

$S' \rightarrow \cdot S, \$$

$S \rightarrow \cdot CC, \$$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

I_1: Goto (0, S)

$S' \rightarrow S \cdot, \$$

I_2: Goto (0, C)

$S \rightarrow C \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

I_3: Goto (0, c)

$C \rightarrow c \cdot C, c/d$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

I_4: Goto (0, d)

$C \rightarrow d \cdot, c/d$

I_5: Goto (2, C)

$S \rightarrow CC \cdot, \$$

I_6: Goto (2, c)

$C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

CLR Parsing Table

Step 5: Construct LALR Parsing Table

states	Actions			Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

states	Action			Goto	
	c	d	$	S	C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
3 6	S36	S47			8 9
4 7	r3	r3	r3		
5			r1		
8 9	r2	r2	r2		

I_1: Goto (2, d)

$C \rightarrow d \cdot \$$

I_8: Goto (3, C)

$C \rightarrow xC \cdot, C/d$

I_3: Goto (3, x)

$C \rightarrow x \cdot C, C/d$

$C \rightarrow \cdot xC, C/d$

$C \rightarrow \cdot d, C/d$

I_4: Goto (3, d)

$C \rightarrow d \cdot, C/d$

I_9: Goto (6, C)

$C \rightarrow xC \cdot, \$$

I_6: Goto (6, x)

$C \rightarrow x \cdot C, \$$

$C \rightarrow \cdot xC, \$$

$C \rightarrow \cdot d, \$$

I_7: Goto (6, d)

$C \rightarrow d \cdot, \$$

CLR Parsing Table

states	Actions			Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Step 5: Construct LALR Parsing Table

states	Action			Goto	
	c	d	$	S	C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
3 6	S36	S47			8 9
4 7	r3	r3	r3		
5			r1		
8 9	r2	r2	r2		

I_1: Goto (2, d)

$C \rightarrow d \cdot \$$

I_8: Goto (3, C)

$C \rightarrow cC \cdot , c/d$

I_3: Goto (3, c)

$C \rightarrow c \cdot C, c/d$

$C \rightarrow \cdot cC, c/d$

$C \rightarrow \cdot d, c/d$

I_4: Goto (3, d)

$C \rightarrow d \cdot , c/d$

I_9: Goto (6, C)

$C \rightarrow cC \cdot , \$$

I_6: Goto (6, c)

$C \rightarrow c \cdot C, \$$

$C \rightarrow \cdot cC, \$$

$C \rightarrow \cdot d, \$$

I_7: Goto (6, d)

$C \rightarrow d \cdot , \$$

CLR Parsing Table

states	Actions			Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Step 5: Construct LALR Parsing Table

states	Action			Goto	
	c	d	$	S	C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
3 6	S36	S47			8 9
4 7	r3	r3	r3		
5			r1		
8 9	r2	r2	r2		

Step 6: Parsing the input string "dcd"

stack	i/p	comment
0	dcd $	A (0,d) : S47
0d47	cd $	ACTION (47, C) → r3 r3 C → d
0C2	cd$	ACTION [2, c] → S36 S36
0C2c36	d$	ACTION [36, d] → S47 S47
0C2C36d47	$	ACTION [47, $] → r3 r3 C → d
0C2c36C89	$	ACTION [89, $] → r2 r2 C → cC
0C2C5	$	ACTION [5, $] → r1 r1 S → CC
OS1	$	ACTION (1, $) → ACC ACC

1) S→CC
2)C→cC
3) C→d

states	Action			Goto	
	c	d	$	S	C
0	S36	S47		1	2
1			ACC		
2	S36	S47			5
3 6	S36	S47			89
4 7	r3	r3	r3		
5			r1		
8 9	r2	r2	r2		

CS8602 COMPILER DESIGN

UNIT II SYNTAX ANALYSIS

Role of Parser – Grammars – Error Handling – Context-free grammars – Writing a grammar –Top Down Parsing – General Strategies Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item Construction of SLR Parsing Table -Introduction to LALR Parser – Error Handling and Recovery in Syntax Analyzer-YACC.