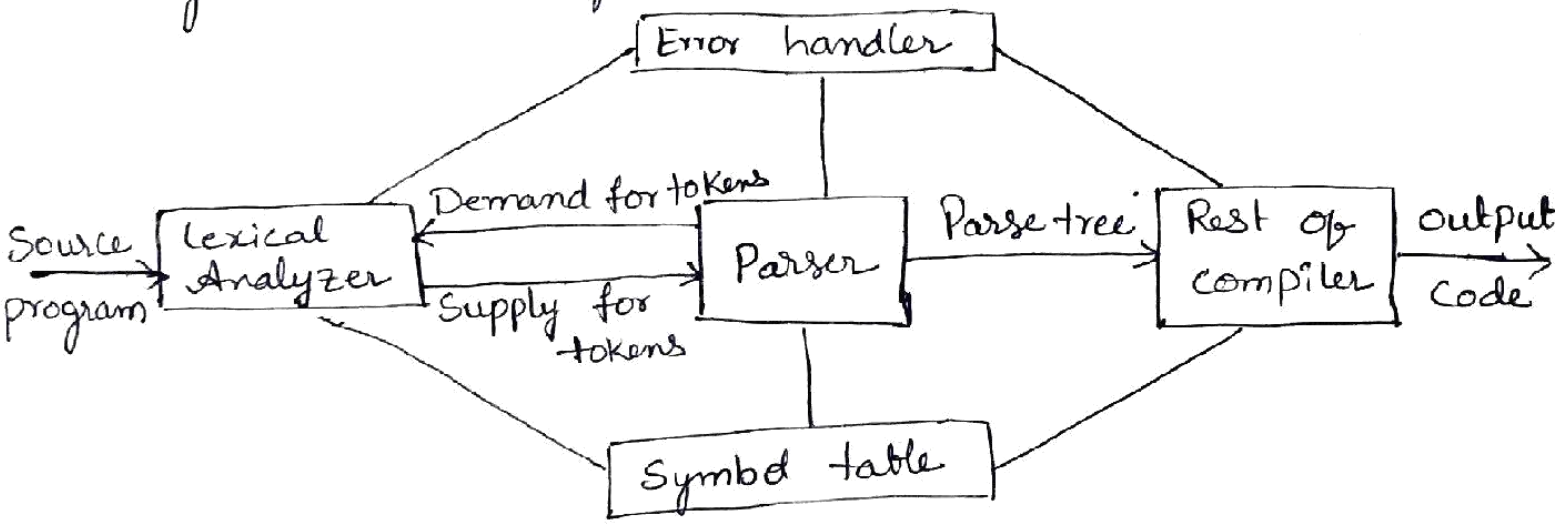
UNIT - 2

Syntax Analysis

Role of a parser:

The Syntax Analysis is the second phase in compilation. The syntax analyzer (parser) basically checks for the Syntax of the language.

Parser: A parsing or Syntax analysis is a process which takes the input string w and produces a parse tree and generate the syntactic errors



In the process of compilation the parser and lexical Analyzer work together. that means, when parser requires string of tokens it invokes lexical analyzer. the lexical analyzer supplies tokens to syntax analyzer (parser)

The parser collects sufficient number of tokens and build a parse tree parser also finds the Syntactical errors if any. It is also necessary that the parser should recover from commonly occuring errors so that remaining task of process the input can be continued.

Context free Grammar (CFG)

The context free grammar G is a collection of following things

V - Set of Non terminals
T - Set of terminals
S - Start symbol
P - production rules

G can be represented as $G = (V, T, P, S)$

The production rules are in the form

Non-Terminal $\longrightarrow (VUT)^*$

Rules to be followed while writing a CFG

1. A single non-terminal should be at L.H.S
2. The rule should be always in the form of LHS $\rightarrow$ RHS where R.H.S may be the combination of non-terminals and terminal symbols.
3. The NULL derivation can be specified as $NT \rightarrow \epsilon$
4. one of the non-terminals should be start symbol

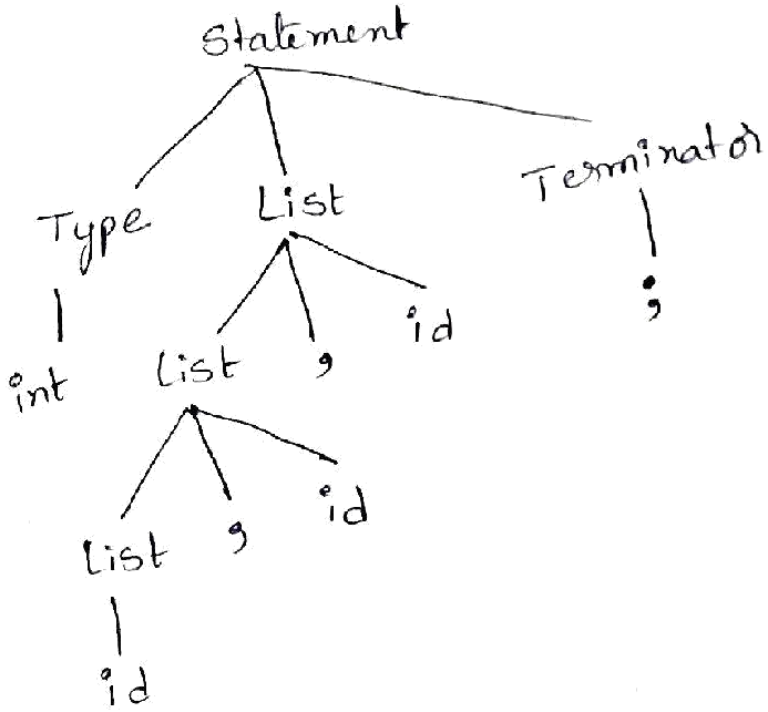
The Non-terminal symbol occurs at the left hand side. These are the symbols which need to be expanded The terminal symbols are nothing but the tokens used in the language.

for example.

Statement $\rightarrow$ Type List Terminator
Type $\rightarrow$ int | float
List $\rightarrow$ List, id
List $\rightarrow$ id
Terminator $\rightarrow$;

(2)

Parse tree for the string int id, id, id;



Derivation and parse trees

Derivation from S means generation of string w from S

For constructing derivation two things are important

1. choice of Non-terminal from several others
2. choice of rule from production rules for corresponding Non-terminal

The Derivation tree is a parse tree which can be constructed by following properties.

1. The root has label S
2. Every vertex can be derived from $(V \cup T \cup \epsilon)$
3. if there exists a vertex A with children $R_1, R_2, \cdots R_n$ then there should be production $A \rightarrow R_1 R_2 \cdots R_n$
4. The leaf nodes are from set T and interior nodes are from set V.

Example

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow a|b$

Left most derivation & parse tree for input $a+b*a+b$

$E \rightarrow \underline{E}+E$

$\underline{E}+E+E$

$a+\underline{E}+E$

$a+\underline{E}*E+E$

$a+b*\underline{E}+E$

$a+b*a+\underline{E}$

$a+b*a+b$

Right most derivation:

$E \rightarrow E*\underline{E}$

$E*\underline{E}+E$

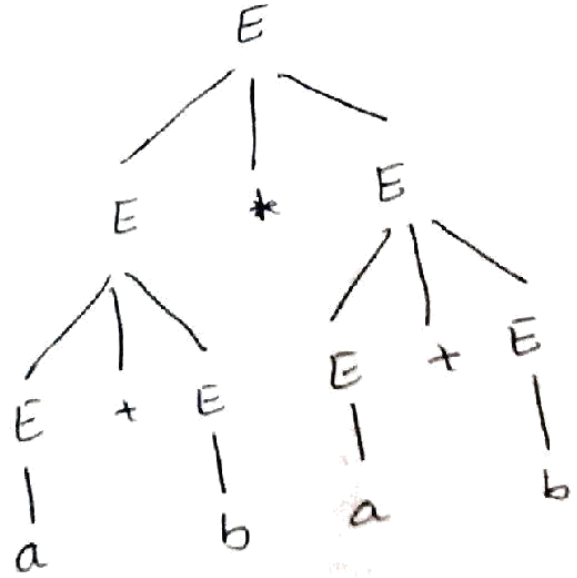
$E*E+b$

$E*a+b$

$E+E*a+b$

$E+b*a+b$

$a+b*a+b$



Ambiguous Grammar:-

A Grammar G is said to be ambiguous if it generates more than one parse tree for sentence of Language $L(G)$

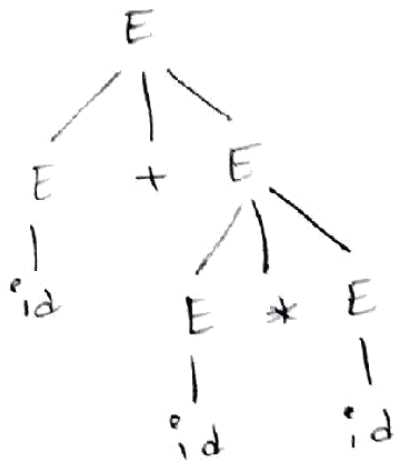
Example

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

for input id + id * id



writing a grammar:

It describes how to divide a work between a lexical analyzer and a parser, converting ambiguous to unambiguous grammar, eliminating left recursion and left factoring.

eliminating Ambiguity

Ambiguous grammar can be converted to unambiguous grammar based on precedence and associativity.

for example

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

In order to convert ambiguous grammar to unambiguous grammar consider precedence & associativity

First consider the production that has least precedence.

$E \rightarrow E + E$

$\Downarrow$

$E \rightarrow E + T$

$E \rightarrow T$

Next consider 2nd highest precedence

$$E \rightarrow E * E$$

$$\Downarrow$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

Next 3rd highest precedence

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

$$\Downarrow$$

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

finally the unambiguous grammar is

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

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

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

Eliminating Left Recursion

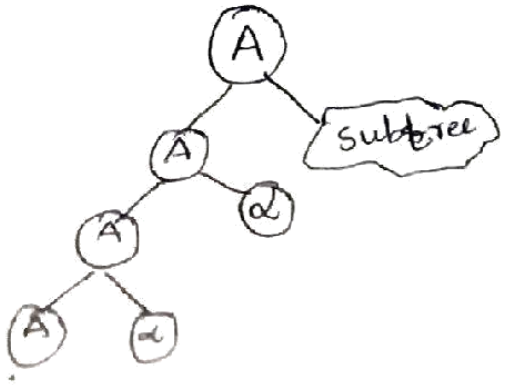
The left Recursive grammar is a grammar as given below

$$A \rightarrow A\alpha.$$

A can be a Non terminal

α denotes some input string.

Thus Expansion of A causes further expansion of A only and due to generation of A, $A\alpha$, $A\alpha\alpha$, $A\alpha\alpha\alpha$, ...



(4) To eliminate left recursion we need to modify the grammar

let G be a CFG having production rule with left Recursion

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

then we eliminate left recursion by re-writing the production rule as

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

a new symbol A' is introduced.

for example.

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

$E \rightarrow E+T \mid T$ ⇒ $E \rightarrow TE'$, $E' \rightarrow +TE' \mid \epsilon$ — $A = E, \alpha = +T, \beta = T$

$T \rightarrow T*F \mid F$ ⇒ $T \rightarrow FT'$, $T' \rightarrow *FT' \mid \epsilon$ — $A = T, \alpha = *F, \beta = F$

$F \rightarrow (E) \mid id$ = No left recursion is this production

after eliminating left Recursion the grammar is

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

Left factoring

Left factoring is used when it is not clear that which of the two alternatives is used to expand the Non-terminals By left factoring we may be able to re-write the production in which the decision can be deferred until enough of the input is seen to make the right choice.

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

after eliminating left factoring

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

for example

$$S \rightarrow \underbrace{iEtS}_{\alpha}\underbrace{}_{\beta_1} \mid \underbrace{iEtS}_{\alpha}eS \mid \underbrace{a}_{\beta_2}\underbrace{}_{\beta_3}$$
$$E \rightarrow b$$

The left factored grammar is

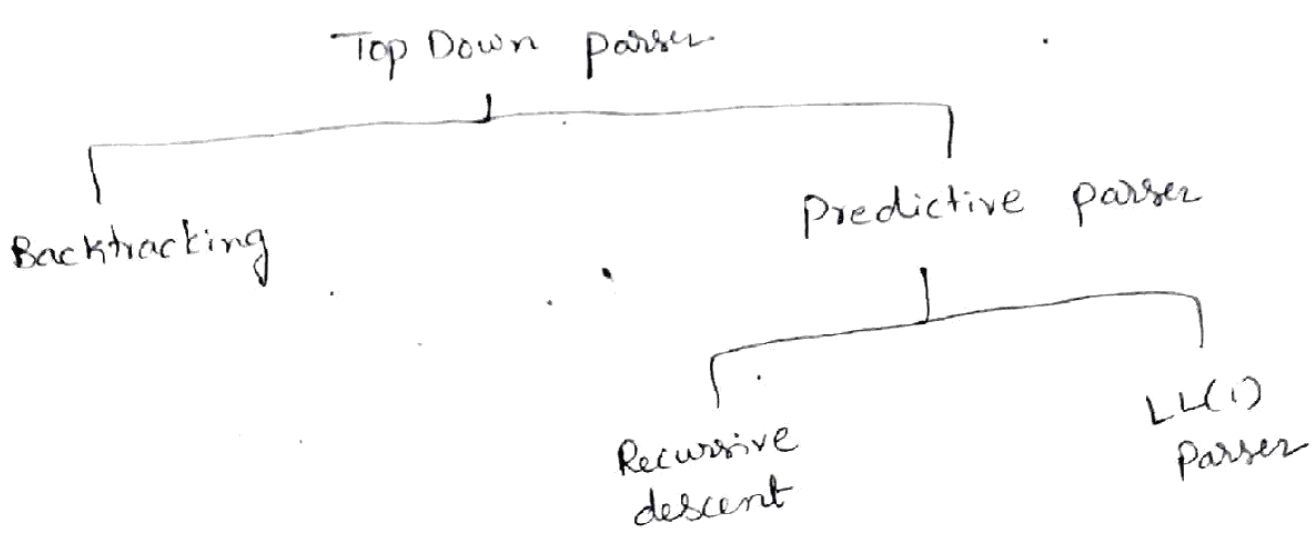
$$S \rightarrow iEtSS'$$
$$S' \rightarrow eS \mid \epsilon$$
$$E \rightarrow b$$

Top Down parsing

There are two types of top down parsing techniques.

1. Backtracking
2. predictive parsing

(5)



1. Backtracking

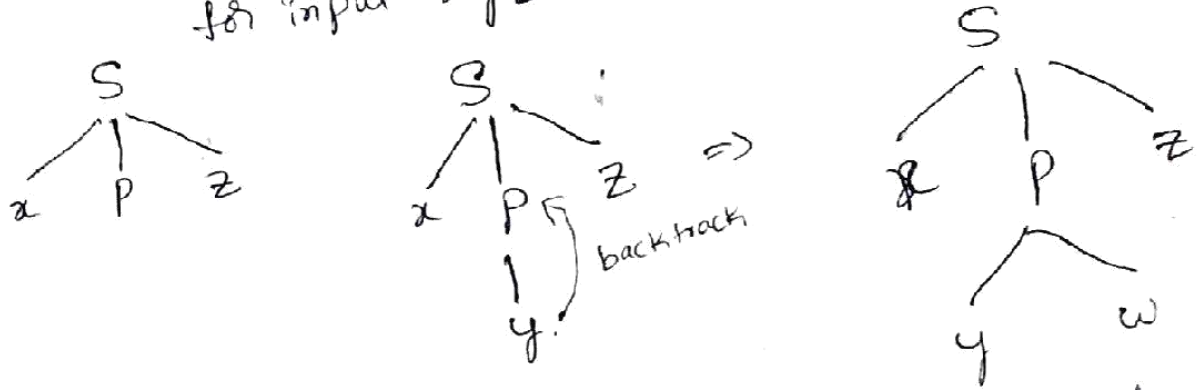
Back tracking is a technique in which for expansion of Non-terminal Symbol we choose one alternative and if some mismatch occurs then we try another alternative if any

for example.

$S \rightarrow xPz$

$P \rightarrow yw \mid y$

Then for input xyzwz



If for Non-terminal there are multiple production rules beginning with the same input Symbol then to get the correct derivation we need to try all these alternatives.

Recursive Descent parser

A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent (RD) parser. In this type of parser the CFG is used to build the recursive routines. The R.H.S of the production rule is directly converted to a program.

Basic steps for construction of RD parser

The R.H.S of the rule is directly converted into program code symbol by symbol

1. If the input symbol is non-terminal then a call to the procedure corresponding to the Non-terminal is made
2. If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol
3. If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.
4. The parser should be activated by a procedure corresponding to the start symbol

Let us consider a grammar.

$E \rightarrow num\ T$

$T \rightarrow * num\ T \mid \epsilon$

Q

```
Procedure E
{
   if lookahead = num then
   {
      match(num);
      T;
   }
   else
      error;
   if lookahead = $
   {
      declare success;
   }
   else
      error;
}

Procedure T
{
   if lookahead = '*'
   {
      match('*')
      if lookahead = 'num'
      {
         match(num);
         T;
      }
      else
         error;
   }
   else NULL
   ~~Combinate~~
}
```

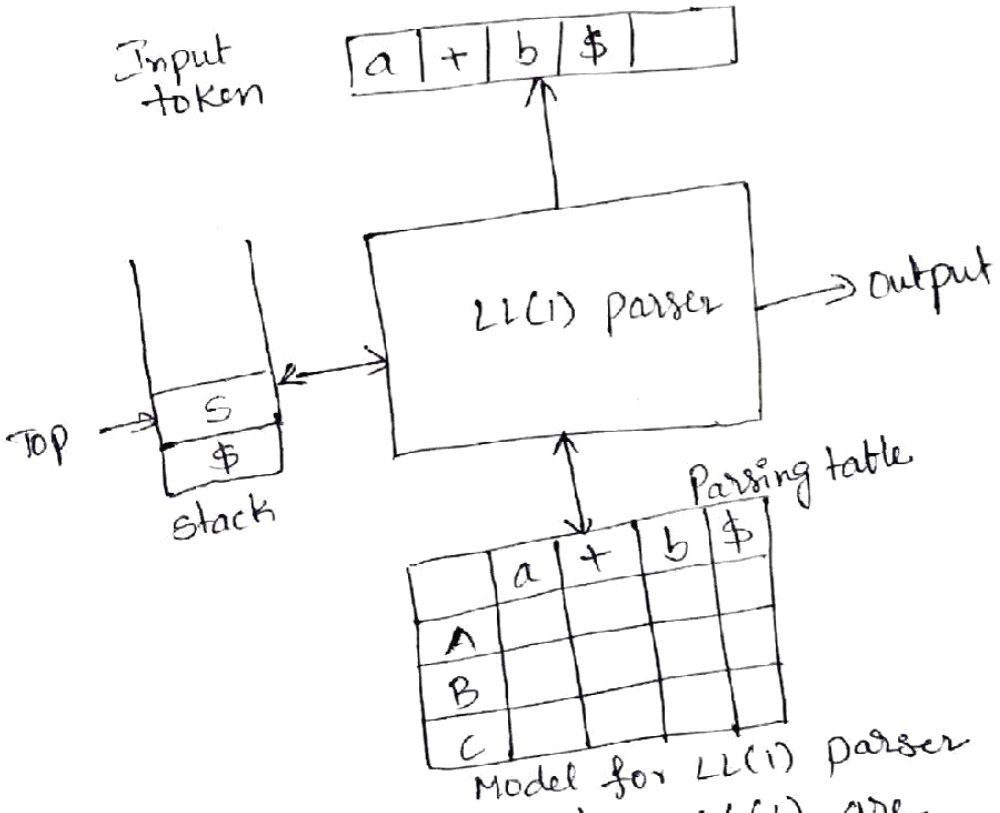
```
Procedure match(token t)
{
   if lookahead = t
   lookahead = next_token;
   else
      error;
}

Procedure error
{
   print("Error!!!");
}
```

LL(1) Grammars :

This top-down parsing algorithm is of non-recursive type For LL(1). The first L means the input is scanned from left to right. The second L means it uses left most derivation for input string. And the number 1 in the input symbol means it uses only one input Symbol (lookahead) to predict parsing process

block diagram for LL(1)



Model for LL(1) parser

The data structures used by LL(1) are

(i) Input buffer
(ii) Stack
(iii) Parsing table

Input buffer is used to store the input tokens The stack is used to hold the left sentential form. The symbols in R.H.S of rule are pushed into the

Stack in reverse order. ie from right to left. The table is basically a two dimensional array. The table has row for Non-terminal and column for terminals The table can be represented as M[A,a] where A is Non terminal and a is current input symbol

A grammar that is obtained after eliminating left recursion & left factoring then the resultant grammar is suitable for predictive parser

Construction of predictive LL(1) parser

1. Computation of FIRST & FOLLOW functions
2. Construct Predictive parsing table using FIRST & FOLLOW
3. Parse the input string using table

FIRST :-

1. if the terminal symbol a $X \rightarrow a\beta$ then FIRST(X) = {a}
2. if there is a rule $X \rightarrow \epsilon$ then FIRST(X) = {ε}
3. For the rule $A \rightarrow X_1 | X_2$ then FIRST(A) = FIRST(X_1) ∪ FIRST(X_2)

FOLLOW :

1. For the start symbol S place $ in Follow(S)
2. If there is a production $A \rightarrow \alpha B\beta$ then everything in FIRST(β) without ε is to be placed in Follow(B)
3. If there is a production $A \rightarrow \alpha B\beta$ & $A \rightarrow \alpha B$ and FIRST(β) = {ε} then FOLLOW(A) = FOLLOW(B) that means everything in FOLLOW(A) is in FOLLOW(B)

Algorithm for parsing table:

For the rule $A \to \alpha$ of Grammar G

1. For each a in FIRST(α) create entry $M[A,a] = A \to \alpha$ where a is terminal
2. For ϵ in FIRST(α) create entry $M[A,b] = A \to \alpha$ where b is the symbol from Follow(A)
3. All remaining entries in the table M are marked as ERROR

Example

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

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

FIRST ①

$E \to TE'$
$\to FT'$ ($\because T \to FT'$)
$\to (E)T' \mid idT'$ ($\because F \to (E) \mid id$)

FIRST(E) = {(, id}

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

FIRST(E') = {+, ϵ}

$T \to FT'$
$\to (E)T' \mid idT'$

FIRST(T) = {(, id}

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

FIRST(T') = {*, ϵ}

$F \to (E) \mid id$

FIRST(F) = {(, id}

FOLLOW ②

1st rule

Starting non terminal is E

Follow(E) = {$}

2nd rule

$E \to TE'$ ($\alpha = \epsilon$, $B = T$, $\beta = E'$)

Follow(T) = FIRST(E') − {ϵ}
= {+, ϵ} − {ϵ}

Follow(T) = {+}

$E \to TE'$ ($\alpha = T$, $B = E'$, $\beta = \epsilon$)

Follow(E') = FIRST(ϵ) − {ϵ}
= ϕ

$E' \rightarrow +TE'$
$\alpha\ B\ \beta$

$FOLLOW(T) = FIRST(E') - \{\epsilon\}$
$= \{+, \epsilon\} - \{\epsilon\}$
$= \{+\}$

$E' \rightarrow +TE'$ $\quad \alpha = +T,\ B = E',\ \beta = \epsilon$

$Follow(E') = FIRST(\epsilon) - \epsilon$
$= \phi$

$T \rightarrow FT'$
$\alpha\ B\ \beta$

$Follow(F) = FIRST(T') - \{\epsilon\}$
$= \{*, \epsilon\} - \{\epsilon\}$
$Follow(F) = \{*\}$

$T \rightarrow FT'$ $\quad \alpha = F,\ B = T',\ \beta = \epsilon$

$Follow(T') = FIRST(\epsilon) - \epsilon$
$= \phi$

$T' \rightarrow *FT'$
$\alpha\ B\ \beta$

$Follow(F) = FIRST(T') - \epsilon$
$= \{*, \epsilon\} - \{\epsilon\}$

$Follow(F) = \{*\}$

$T' \rightarrow *FT'$ $\quad \alpha = *F,\ B = T',\ \beta = \epsilon$

$Follow(T') = FIRST(\epsilon) - \epsilon$
$= \phi$

$F \rightarrow (E)$
$\alpha\ B\ \beta$

$Follow(E) = FIRST(\beta)) - \epsilon$
$= \{)\}$

$F \rightarrow id$ ✗ there is NO nonterminal

$E \rightarrow TE'$ $\quad \alpha = \epsilon,\ B = T,\ \beta = E'$
$\alpha\ B\ \beta$

$FIRST(E') = \{+, \epsilon\}$
$Follow(T) = Follow(E)$
$Follow(T) = \{\$,)\}$

$E \rightarrow TE'$ $\quad \alpha = T,\ B = E',\ \beta = \epsilon$
$\alpha\ B\ \beta$

$FIRST(\beta) = FIRST(\epsilon)$
$= \{\epsilon\}$
$Follow(E') = Follow(E)$
$= \{), \$\}$

$E' \rightarrow +TE'$
$\alpha\ B\ \beta$

$FIRST(E') = \{+, \epsilon\}$
$Follow(T) = Follow(E')$
$= \{), \$\}$

$E' \rightarrow +TE'\epsilon$ $\quad \alpha = +T,\ B = E',\ \beta = \epsilon$
$\alpha\ B\ \beta$

$FIRST(\beta) = FIRST(\epsilon) = \epsilon$
$Follow(E') = Follow(E')$
$= \{), \$\}$

$T \rightarrow FT'$ $\quad \alpha = \epsilon,\ B = F,\ \beta = T'$
$\alpha\ B\ \beta$

$FIRST(\beta) = FIRST(T')$
$= \{*, \epsilon\}$
$Follow(F) = Follow(T)$
$= \{+,), \$\}$

$T \rightarrow FT'\epsilon$ $\quad \alpha = F,\ B = T',\ \beta = \epsilon$
$\alpha\ B\ \beta$

$FIRST(\beta) = FIRST(\epsilon) = \epsilon$
$Follow(T') = Follow(T)$
$= \{+,), \$\}$

$T' \rightarrow *FT'$ $\quad \alpha = *$
$\alpha \; B \; \beta$ $\quad B = F$
$\beta = T'$

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

Follow(F) = Follow(T')
$= \{+,), \$\}$

$T' \rightarrow *FT'\epsilon$ $\quad \alpha = *F$
$\alpha \; B \; \beta$ $\quad B = T'$
$\beta = \epsilon$

FIRST(β) = FIRST$(\epsilon) = \epsilon$

Follow(T') = Follow(T')
$= \{+,), \$\}$

$F \rightarrow (E)$ $\quad \alpha = ($
$\alpha \; B \; \beta$ $\quad B = E$
$\beta =)$

FIRST(β) = FIRST$())$

there is NO $\{\epsilon\}$ element in FIRST(β) then 3rd rule is not satisfied for this production

$F \rightarrow id$ ✗

there is NO Non terminal in this production then we can not map $A \rightarrow \alpha B \beta$.

The FIRST & FOLLOW values are as follows.

	FIRST	Follow
E	$\{(, id\}$	$\{), \$\}$
E'	$\{+, \epsilon\}$	$\{), \$\}$
T	$\{(, id\}$	$\{+,), \$\}$
T'	$\{*, \epsilon\}$	$\{+,), \$\}$
F	$\{(, id\}$	$\{*, +,), \$\}$

Parsing table

Now we will fill up the entries in the table by using algorithm. For that consider each rule & production one by one

$E \rightarrow TE'$ $\quad A = E$
$A \rightarrow \alpha$ $\quad \alpha = TE'$

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

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

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

$E' \rightarrow +TE'$
$A \rightarrow \alpha$

FIRST(α) = FIRST$(+TE')$
$= \{+\}$

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

$E' \rightarrow \epsilon$

$A \rightarrow \alpha$

$\alpha = \epsilon$

Follow(E') = {), \$}

$M[E',)] = E' \rightarrow \epsilon$

$M[E', \$] = E' \rightarrow \epsilon$

$T \rightarrow FT'$

$A \rightarrow \alpha$

FIRST(FT') = FIRST(F)
= {(, id}

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

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

$T \rightarrow *FT'$

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

$T \rightarrow \epsilon$

Follow(T) = {+,), \$}

$M[T', +] = T' \rightarrow \epsilon$

$M[T',)] = T' \rightarrow \epsilon$

$M[T', \$] = T' \rightarrow \epsilon$

$F \rightarrow (E)$

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

$F \rightarrow id$

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

	id	+	*	(	)	$
E	$E \rightarrow TE'$	Error	Error	$E \rightarrow TE'$	Error	Error
E'	Error	$E' \rightarrow +TE'$	Error	Error	$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	Error	Error	$T' \rightarrow FT'$	Error	Error
T'	Error	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	Error	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	Error	Error	$F \rightarrow (E)$	Error	Error

Now the input string id + id * id $ can be parsed using above table

The stack will contain start symbol E, in the input buffer input string is placed.

E is at top of the stack and input pointer is at id hence M[E, id] is referred this entry is $E \rightarrow TE'$

So, we will push E' first then T (Right hand side of the production is placed in reverse order in stack)

10

Input parsing

Stack	Input	parsing Action
$E	id+id*id$	E → TE'
$E'T	id+id*id$	T → FT'
$E'T'F	id+id*id$	F → id
$E'T'id	id+id*id$	pop id
$E'T'	+id*id$	T' → ϵ
$E'ϵ	+id*id$	
$E'	+id*id$	E' → +TE'
$E'T+	+id*id$	pop +
$E'T	id*id$	T → FT'
$E'T'F	id*id$	F → id
$E'T'id	id*id$	pop id
$E'T'	*id$	T' → *FT'
$E'T'F*	*id$	pop *
$E'T'F	id$	F → id
$E'T'id	id$	popid
$E'T'	$	T' → ϵ
$E'	$	E' → ϵ
$	$	string Accepted.

unit-2

* Strategies to Recover from Syntactic Errors :-

4 types of Recover strategies from Syntactic Errors.

No one strategy is universally acceptable but can be applied to a broad domain.

1. Panic mode :

* This strategy is used by most parsing methods.
* This is simple to implement.
* In this method on discovering error, the parser discards input symbol one at a time. this process is continued until one of a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as semicolon & end. These tokens indicate an end of input statement.
* In this mode a considerable amount of input is skipped with out checking it for additional errors.
* This method guarantees not to go in infinite loop.
* If there are less number of errors in the same statement. then this strategy is a best choice.

2. Phrase Level Recovery :

* In this method, on discovering error parser perform local correction on remaining input.
* It can replace a prefix of remaining input by some string. this actually helps the parser to continue its job.
* The local correction can be replacing comma by semicolon, deletion of extra semicolons & inserting missing semicolon, The type of local correction is decided by compiler designer
* while doing the replacement a care should be taken for not going in an infinite loop.

* This method is used in many error-repairing compilers.
* The draw back of this method is it finds difficult to handle the situations where the actual error has occured before the point of detection.

3. Error Production:

* If we have a knowledge of common errors that can be encountered then we can incorporate these errors by augmenting the grammar of the corresponding language with error productions that generate the erroneous constructs.
* If error production is used then during parsing, we can generate appropriate error message and parsing can be continued.
* This method is extremely difficult to maintain. Because if we change the grammar then it becomes necessary to change the corresponding error productions.

4. Global Production:

* A Compiler that makes very few changes in processing an incorrect input string.
* We expect less number of insertions, deletions and changes of tokens to recover from erroneous input
* Such methods increase time and space requirements at parsing time.
* Global production is thus simply a theoretical concept

3.1 Error Recovery in Predictive Parsing:

An error is detected during the predictive parsing when the terminal on top of the stack does not match the next input symbol, or when nonterminal A on top of the stack, a is the next input symbol, and parsing table entry M[A, a] is empty.

Panic mode recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens.

Following are some ways by which the synchronizing set can be choosen -

1. place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it likely that parsing can continue.
2. We might add keywords that begins statements to the synchronizing for the nonterminals generating expressions.
3. We can add symbols in FIRST(A) to the synchronizing set for non terminal A.

   Due to this it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.
4. If a terminal on top of stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted.
5. If a nonterminal can generate the empty string, then the production deriving ϵ can be used as a default. This may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of non terminals that have to be considered during error recovery.

Consider following parsing table

Nonterminals	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	Synch	Synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	Synch		$T \to FT'$	Synch	Synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	Synch	Synch	$F \to (E)$	Synch	Synch

* Synch indicates the synchronizing tokens obtained from FOLLOW set of non terminal.

The FOLLOW set for the given grammar is

$$FOLLOW(E) = FOLLOW(E') = \{), \$\}$$

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

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

* If parser looks up entry M[A, a] as blank then the input symbol a is skipped.
* If entry is synch then the non terminal on the top of the stack is popped.
* If token on the top of the stack does not match the input symbol then we pop the token from the stack.

~~Stack~~ ~~Inp~~

Example

Consider the input +id**id How to recover this input from the following productions.

$E \to TE'$

$E \to +TE'$

$E \to *FT'$

$T' \to *FT'$

$F \to (E)/id$

Stack	Input	Action
$E	+id**id$	skip +
$E	id**id$	
$E'T	id**id$	
$E'T'F	id**id$	
$E'T'id	id**id$	
$ET'	**id$	
$E'T'F*	**id$	
$E'T'F	*id$	Error. M[F, *] = Synch ∴ popF
$E'T'	*id$	
$E'T'F*	*id$	
$E'T'F	id$	
$E'T'id	id$	
$E'T'	$	
$E'	$	
$	$	