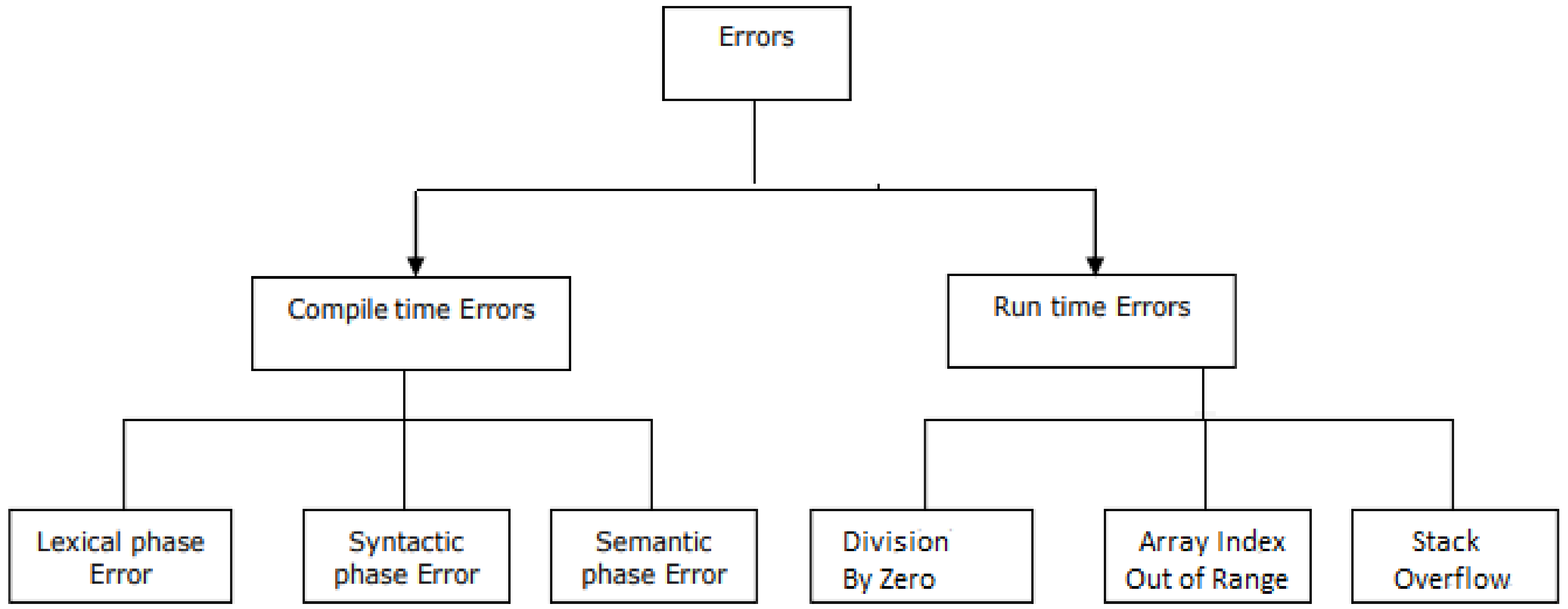
Error Handler

Error

- Error detection, reporting and recovery are the main functionality of error handler module.
- Error is generally detected in lexical analysis, syntax analysis and semantic analysis phase.

Error Classification

- Compilation Error: are detected during compilation of program.
- Dynamic/Run time error: are detected during execution of program.



Types of Compiler Based on Error Handling

- Simple Compiler: stop its all activities after detection of error.
- Complex Compiler: try to repair the error and if succeed then continue to normal processing
- Sophisticated Compiler: try to correct error by making guess according to user intention.

Properties of Good Error Message

- Easy to understand
- Error message must localize the problem, not general enough. E.g. in place of "**missing declaration**" use "**x is not declared in the function**".
- Locate the line number at which error has occurred
- In the form of source program
- Should not be redundant, i.e. same message should not be produced again

Lexical Phase Errors

- Lexical phase errors are mainly generated because next token to be read in source program is misspelled.
- User have declared digit at the beginning of identifier
- Integer constant out of bounds
- Identifier name is too long, more than 32 characters
- wilhe (a<b), LA will consider this as valid token and treat it as function in place of loop

Recovery from Lexical Phase Errors

- Panic mode: In LA, a situation arises when it is unable to detect the token then it either delete or skip the successive characters from the remaining input until the LA finds another token,
- Minimum distance matching: Spelling of token is corrected by choosing token which is closest to it alphabetically.

Syntactic Analysis Phase Errors

- Error detected during syntax analysis phase are called syntax phase errors
- These errors occurs when stream of tokens not follow the grammatical rules of programming language. e.g.
- Missing ; at the end
- Missing right parenthesis (a+(b*c)
- Colon in place of semicolon i=1:
- Whenever a parser don’t find any legal move corresponding to top stack symbol and current input symbol, then it will generate syntactic or syntax error.

Error Recovery Techniques from Syntactic Analysis Phase Errors

- Panic mode recovery: This recovery technique is used in all the parsers to recover from syntactic error. Parser discard the input symbols until a synchronizing token (e.g. ;) or end is encountered. Parser deletes stack entries until it finds an entry for which it can continue parsing.
- Minimum Distance Correction: Parser performs the least number of insertion, deletion and symbol modifications necessary to change incorrect program to correct program.
- Minimum Hamming Distance: If program P has K errors, then minimum number of modifications done to remove the K errors from the program P is called minimum hamming distance of that program.

Error Recovery in Operator Precedence Parsing

- Two main source of syntactic error in operator precedence parsing are:
    - Reduction error
    - Shift reduce error

Error Recovery in Operator Precedence Parsing

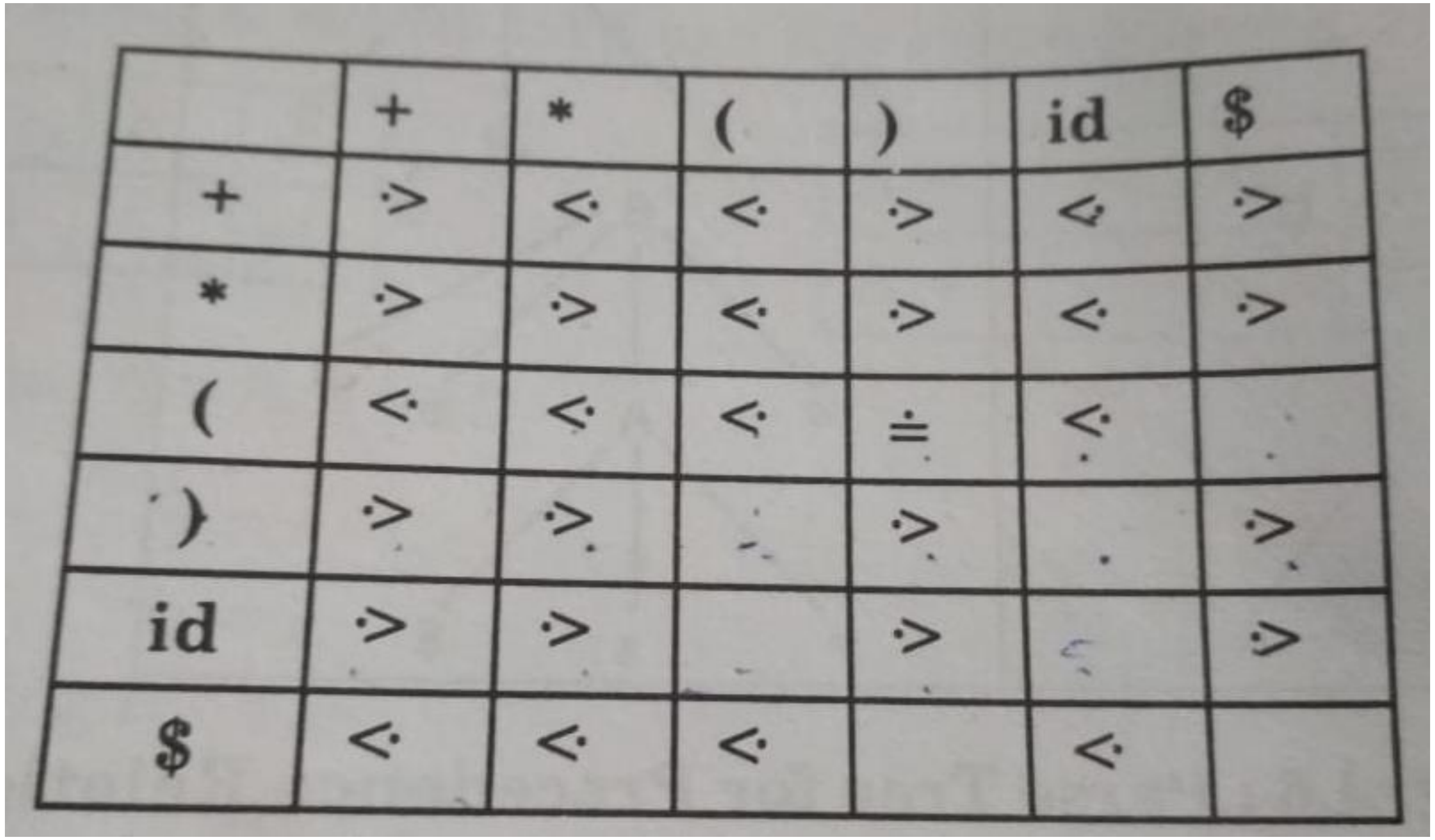
- Reduction error: These errors occurs when handle has been found but there is no production with this handle as a right side of production. E.g.
  - S->aBbc is a production rule and “abc” is at the top of the stack, a diagnostic message can be generated because abc don’t match with aBbc, “Missing B on line”
  - If any operator + or * is reduced, it checks operands on both sides of operator. If it don’t found then error message will be “**Missing expression**”
  - If two id’s appear together to be reduced with no operator between them then message will be “**Expressions not connected by operator**”
  - If () appears to be reduced instead of (E), then message will be “**Null expression between parenthesis**”

Error Recovery in Operator Precedence Parsing

- Shift reduce error: These errors occurs if no precedence relation exists between terminal on top of the stack and current input symbol.
- Let b is the symbol on top of the stack and let a is the symbol below b in the stack. Let c is the current input symbol and d is the next input symbol. If there is no precedence between b and c then this error can be handled as follows:
  - If a<. c then delete b from stack
  - If b<.d then delete c from input string
  - If there is symbol e such that b<.e<.c then insert e in front of c in input string.
  - In other method of handling shift reduce error, we can insert error recovery routines in the blank entries of the table.

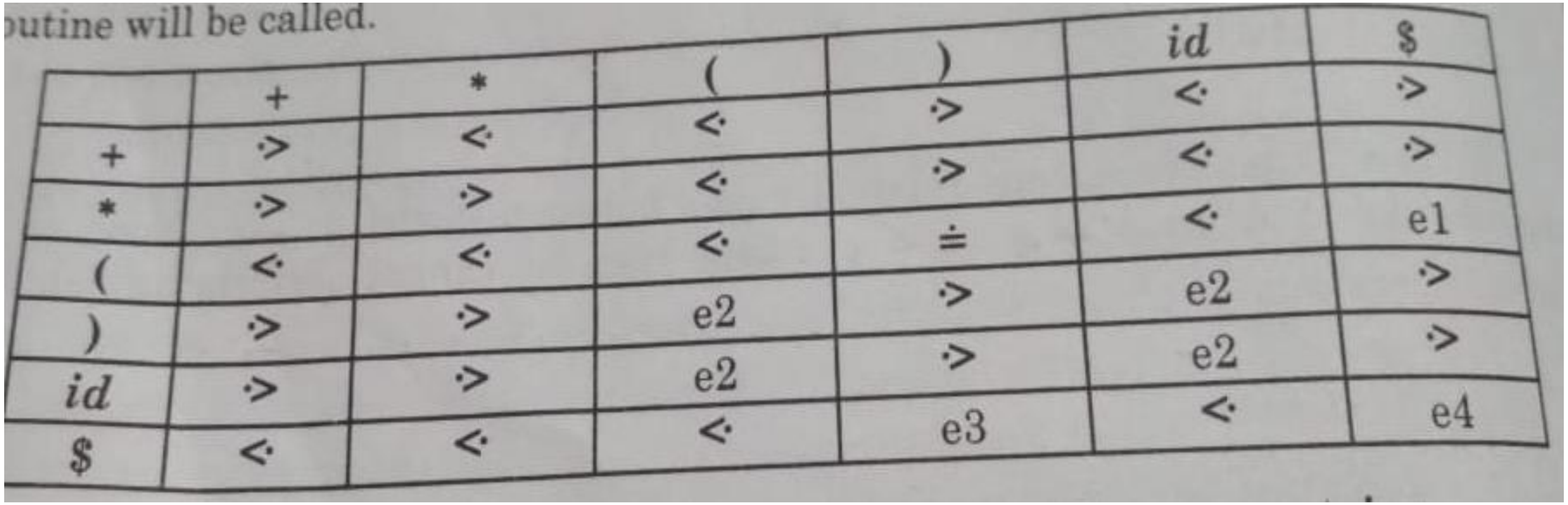
Error Recovery in Operator Precedence Parsing

- E->E+T|T
- T->T*F|F
- F->(E)|id
    - First of F=(,id
    - Last of F=),id
    - First of T=(,id,*
    - Last of T=),id,*
    - First of E=(,id,*,+
    - Last of E=),id,*,+



	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

Error Recovery in Operator Precedence Parsing



outine will be called.

	+	*	(	)	*id*	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	e1
)	·>	·>	e2	·>	e2	·>
id	·>	·>	e2	·>	e2	·>
$	<·	<·	<·	e3	<·	e4

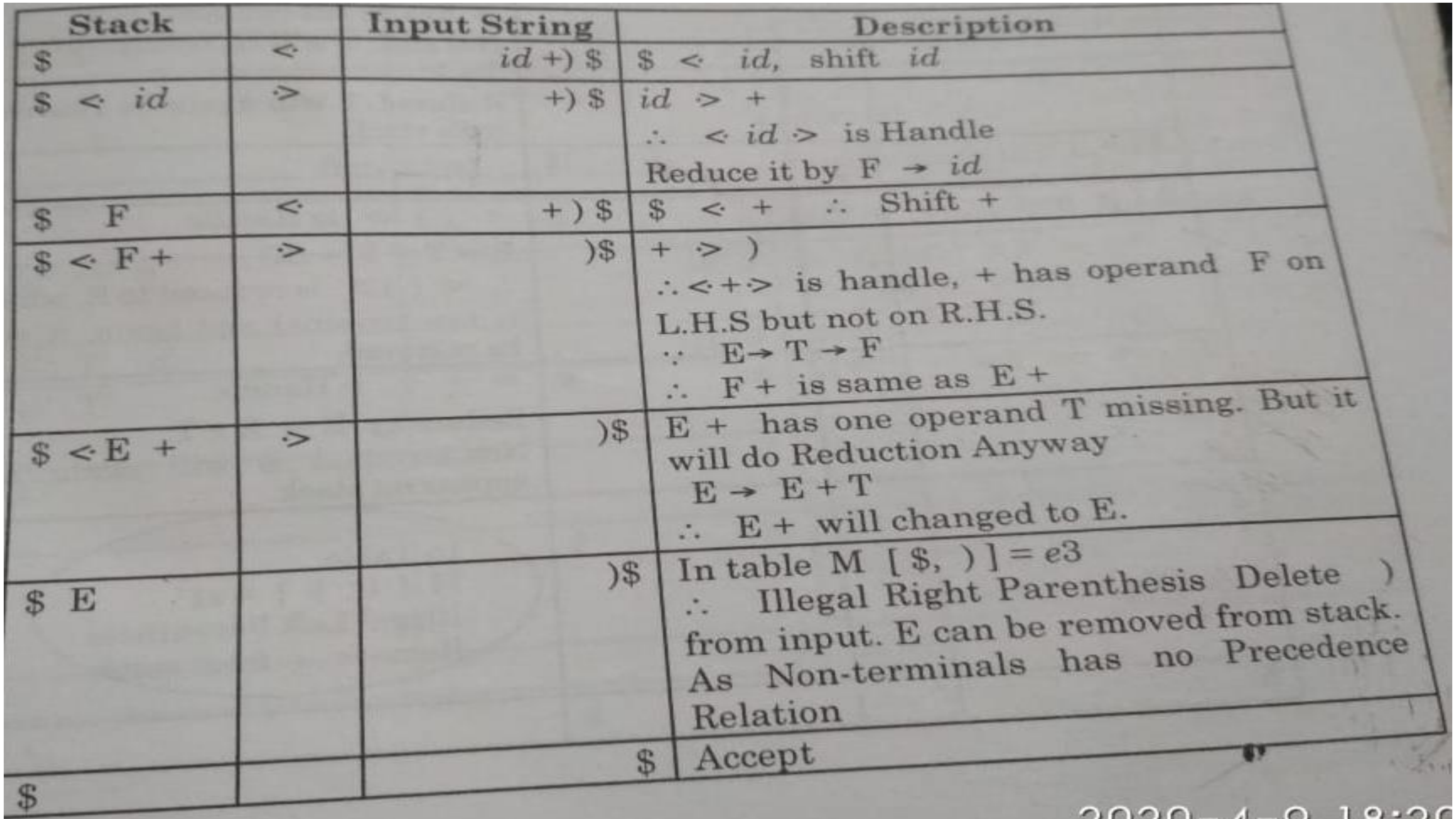
e1: occurs when expression ends with left parenthesis: Diagnostic message: "**Illegal left parenthesis**", Recovery: **remove (from stack**

e2: occurs when we have expression like id(,   idid,   )id,   )(i.e. operator is missing between symbols, Diagnostic message: "**Missing operator**", Recovery: **insert operator between symbols**

e3: occurs when expression begins with right parenthesis , Diagnostic message: "**Illegal right parenthesis**", Recovery: **Delete) from input**

e4: occurs when there is no input, Diagnostic message: "**Missing expression**", recovery **insert id onto input**

Error Recovery in Operator Precedence Parsing



Stack		Input String	Description
$	<·	*id* +) $	$ <· *id*, shift *id*
$ <· *id*	·>	+) $	*id* ·> + ∴ <· *id* ·> is Handle Reduce it by F → *id*
$ F	<·	+) $	$ <· + ∴ Shift +
$ <· F +	·>	)$	+ ·>) ∴ <·+·> is handle, + has operand F on L.H.S but not on R.H.S. ∵ E→ T → F ∴ F + is same as E +
$ <·E +	·>	)$	E + has one operand T missing. But it will do Reduction Anyway E → E + T ∴ E + will changed to E.
$ E		)$	In table M [$,)] = *e*3 ∴ Illegal Right Parenthesis Delete) from input. E can be removed from stack. As Non-terminals has no Precedence Relation
$		$	Accept

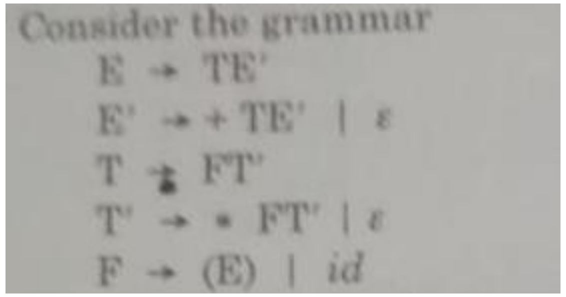
(*id* + (* *id*)

Stack		Input String	Description
$	<·	(*id* + (* *id*) $	$ <· (, Shift (
$ (	<·	*id* + (* *id*) $	(<· *id*, Shift *id*
$ (<· *id*	·>	+ (* *id*)$	<· *id* ·> is Handle Reduce it by F → *id*
$ (F	<·	+ (* *id*)$	(<· +, shift +
$ (F +	<·	(* *id*)$	+ <· (, shift (
$ (F + (	<·	* *id* $	(<· *, shift *
$ (F + (*	<·	*id* > $	* <· *id*, shift *id*
$ (F + (* <· *id*	·>	)$	<· *id* ·> is Handle, id ·>) Reduce by F → *id*
$ (F + (<· * F	·>	) $	Remove Non-Terminal F. As, there is no Precedence Relation between Non-terminals. ∴ <· * F ·> can be written as <· * ·>
$ (+ (<· *	·>	)$	<· * ·> is Handle. * has no left operand But still, it will be reduced by T → T * F Reduced T will again be removed from stack.
$ (+ (	≐	)$	(≐) ∴ shift)
$ (+ <· ( )	·>	$	<· ( ) ·> is Handle E → T → F → (E) ∴ <· ( ) ·> is reduced to E, which is non-terminal and again, it will be removed.
$ (<· +	·>	$	<· + ·> is Handle. Reduce by E → E + T Non-terminal E will again not appear on stack.
$ (		$	In Table M [(, $] = *e*1 Illegal Left Parentheis Remove (from stack
$		$	

Error Recovery in Predictive/LL Parsing

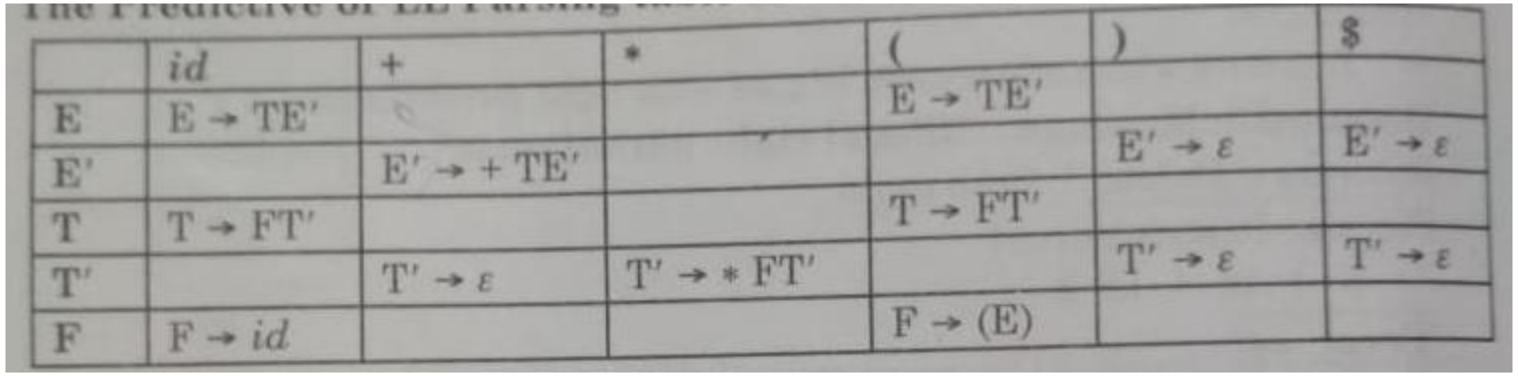
- In this parser, error situation arises when top stack symbol does not match with next input character or parsing table entry M[X,a] is blank
- To recover from error:
  - Panic Mode: On finding error, symbol from input string and non terminals from top of the stack are deleted until symbol on top of the stack matches with current input symbol.
  - Fill blank entries with error routines.

Error Recovery in Predictive/LL Parsing

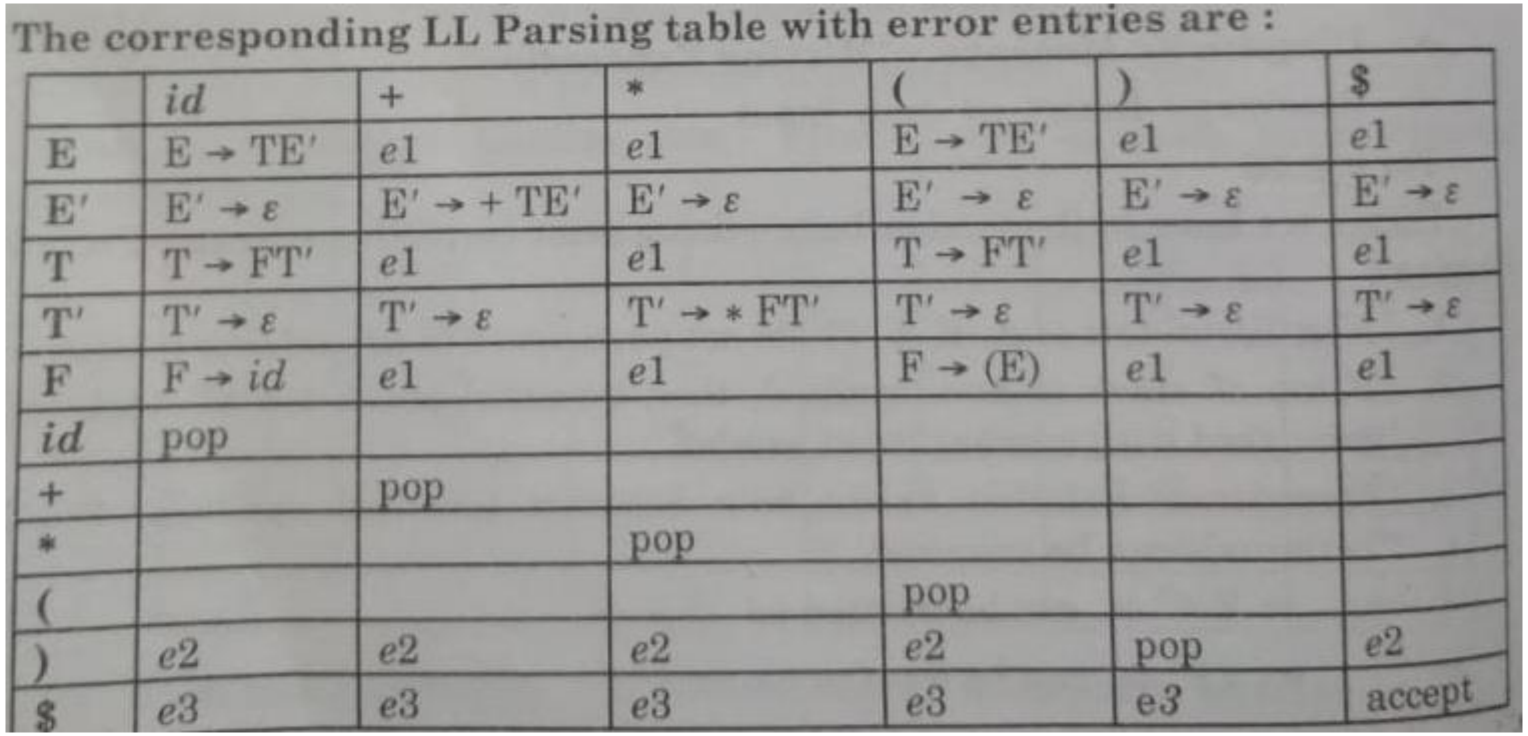


Consider the grammar

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



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



The corresponding LL Parsing table with error entries are :

	id	+	*	(	)	$
E	$E \rightarrow TE'$	*e*1	*e*1	$E \rightarrow TE'$	*e*1	*e*1
E'	$E' \rightarrow \varepsilon$	$E' \rightarrow +TE'$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	*e*1	*e*1	$T \rightarrow FT'$	*e*1	*e*1
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$	*e*1	*e*1	$F \rightarrow (E)$	*e*1	*e*1
id	pop					
+		pop				
*			pop			
(				pop		
)	*e*2	*e*2	*e*2	*e*2	pop	*e*2
$	*e*3	*e*3	*e*3	*e*3	*e*3	accept

Error Recovery in Predictive/LL Parsing

- Pop: entries in the table will be executed when the symbol at the top of stack is same as current input symbol.
- E’->ε and T’->ε are added in some blank entries to postpone some error detection
- e1 occur when input string begins with operator, “Missing operand” will be the diagnostic message and recovery from this error can be done by pushing id into input.
- e2 occur when top of the stack contain right parenthesis but input string does not contain it, “Missing right parenthesis will be the diagnostic message ” and it can be recovered from this error by popping right parenthesis from stack.
- e3 occur when stack is empty but still there is input symbol left in the input string, “unexpected input symbol” will be the error message and it can recover from this by removing remaining symbols from the input string.

Ans. (*i*) *id* +)

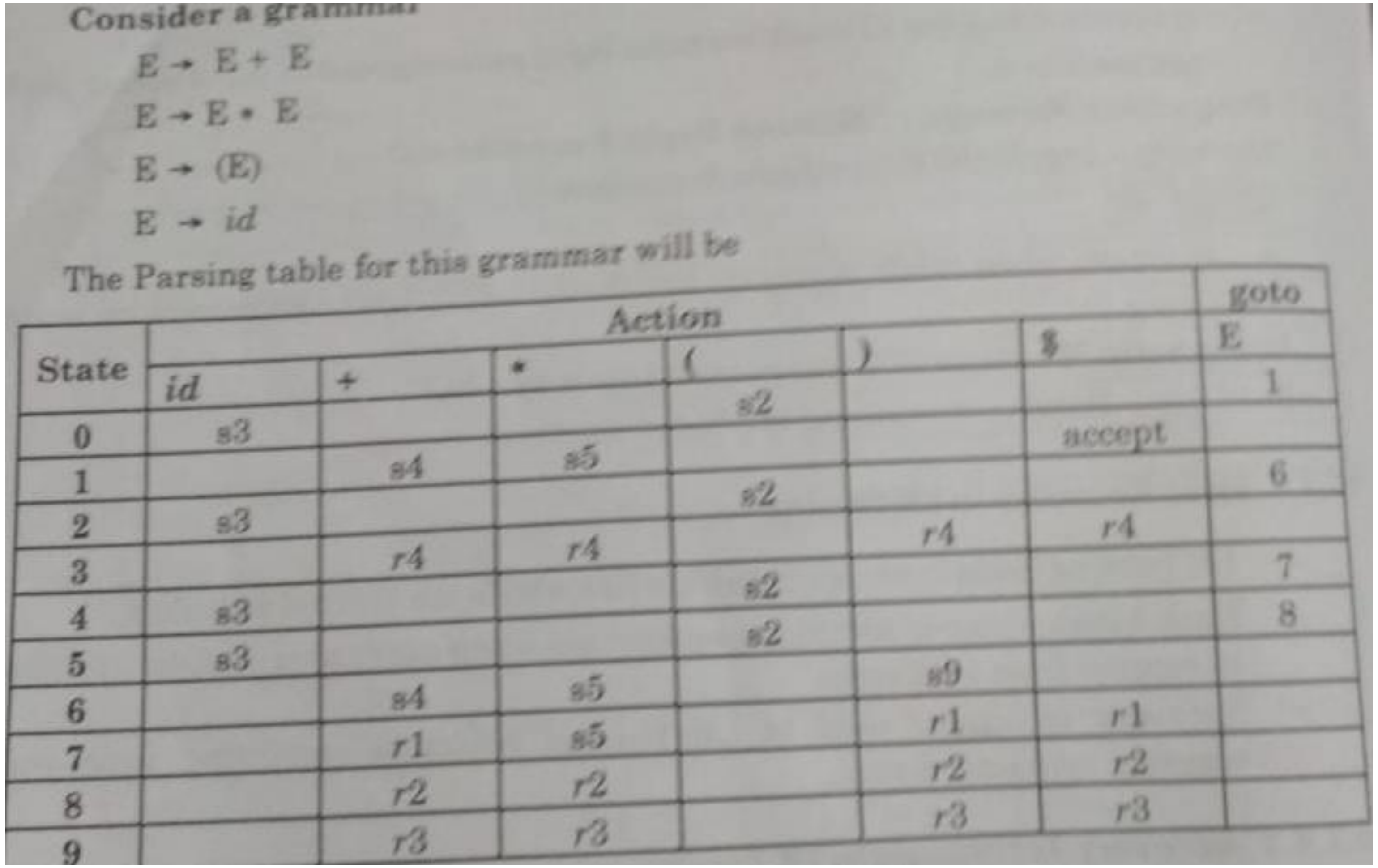
Stack	Input String	Description
$ E	*id* +)$	M [E, *id*] = E → TE′ ∴ Insert E′ T onto stack
$ E′ T	*id* +)$	M [T, *id*] = T → FT′ Pop T . Push T′ F onto stack
$ E′ T′ F	*id* +) $	M [F, *id*] = F → *id* Pop F . Push d
$ E′ T′ *id*	*id* +) $	Top Stack Symbol = current input symbol (*id*) ∴ Remove *id* from both
$ E′ T′	+) $	M [T′, +] = T′ → ε ∴ Pop T′ . Push ε
$ E′	+) $	M [E′, +] = E′ → + TE′ Pop E′ . Push E′T + onto stack
$ E′ T +	+) $	Remove + from both
$ E′ T	) $	M [T,)] = *e*1 ∴ Missing operand Error ∴ Insert *id* onto input
$ E′ T	*id*) $	M [T, *id*] = T → FT′ Pop T. Push T′F onto stack
$ E′ T′ F	*id*) $	M [F, *id*] = F → *id* ∴ Pop F. Push *id*.
$ E′ T′ *id*	*id*) $	Remove *id* from both
$ E′ T′	) $	M [T′,)] = T′ → ε ∴ Pop T′ . Push ε
$ E′	)$	M [E′,)] = E′ → ε ∴ Pop E′. Push ε.
$	)$	M [$,)] = *e*3 ∴ Unexpecte Right Parenthesis. ∴ Remove input symbol)
$	$	accept

(ii) (*id* + (* *id*)

Stack	Input String	Description
$ E	(*id* + (* *id*) $	M [E, ( ] = E → TE′ ∴ Pop E, Push E′ T (reverse)
$ E′ T	(*id* + (* *id*) $	M [T, ( ] = T → FT′ Pop T, Push T′ F
$ E′ T′ F	(*id* + (* *id*) $	M [F, ( ] = F → (E) Pop F, Push) E (
$ E′ T′) E (	(*id* + (* *id*) $	Remove (from both
$ E′ T′) E	*id* + (* *id*) $	M [E, *id*] = E → TE′ Pop E, Push E′ T
$ E′ T′) E′ T	*id* + (* *id*) $	M [T, *id*] = T → FT′ Pop T, Push T′ F
$ E′ T′) E′ T′ F	*id* + (* *id*) $	M [F, *id*] = F → *id* Pop F, Push *id*
$ E′ T′) E′ T′ *id*	*id* + (* *id*)$	Remove *id* from both
$ E′ T′) E′ T′	+ (* *id*)$	M [T′, +] = T′ → ε Pop T′, Push ε.
$ E′ T′) E′	+ (* *id*)$	M [E′, +] = E′ → + TE′ Pop E′ Push E′ T +
$ E′ T′) E′ T +	+ (* *id*)$	Remove + from both
$ E′ T′) E′ T	(* *id*)$	M [T, ( ] = T → FT′ Pop T, Push T′ F
$ E′ T′) E′ T′ F	(* *id*)$	M [F, ( ] = F → (E) Pop F, Push) E (.
$ E′ T′) E′ T′) E (	(* *id*)$	Remove (from both of them
$ E′ T′) E′ T′) E	* *id* $	M [E, *] = *e*1 ∴ Missing operand

Stack	Input String	Description
		Pop E, Push E′ T
$ E′ T′) E′ T′) E′ T	*id* * *id* $	M [T, *id*] = T → FT′ Pop T, Push T′ F
$ E′ T′) E′ T′) E′ T′ F	*id* * *id*, $	M [F, *id*] = F → *id* Pop F, Push *id*
$ E′ T′) E′ T′) E′ T′ *id*	*id* * *id* $	Remove *id* from both
$ E′ T′) E′ T′) E′ T′	* id$	M [T′, *] = T′ → * F T′ Pop T′, Push T′ F *
$ E′ T′) E′ T′) E′ T′ F *	* *id* $	Remove *
$ E′ T′) E′ T′) E′ T′ F	*id* $	M [F, *id*] = F → *id* Pop F, Push *id*
$ E′ T′) E′ T′) E′ T′ *id*	*id* $	Remove *id*
$ E′ T′) E′ T′) E′ T′	$	M [T′, $] = T′ → ε Pop T′, Push ε
$ E′ T′) E′ T′) E′	$	M [E′, $] = E′ → ε ∴ Pop E′, Push ε
$ E′ T′) E′ T′)	$	M [ ), $] = *e*2 ∴ Missing Right Parenthes Pop)
$ E′ T′) E′ T′	$	M [T′ $] = T′ → ε
$ E′ T′) E′	$	M [E′, $] = E′ → ε
$ E′ T′)	$	M [ ), $] = *e*2 Remove)
$ E′ T′	$	M [T′, $] = T′ → ε
	$	M [E′, $] = E′ → ε
$ E′		

Error Recovery in LR Parsing



Consider a grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

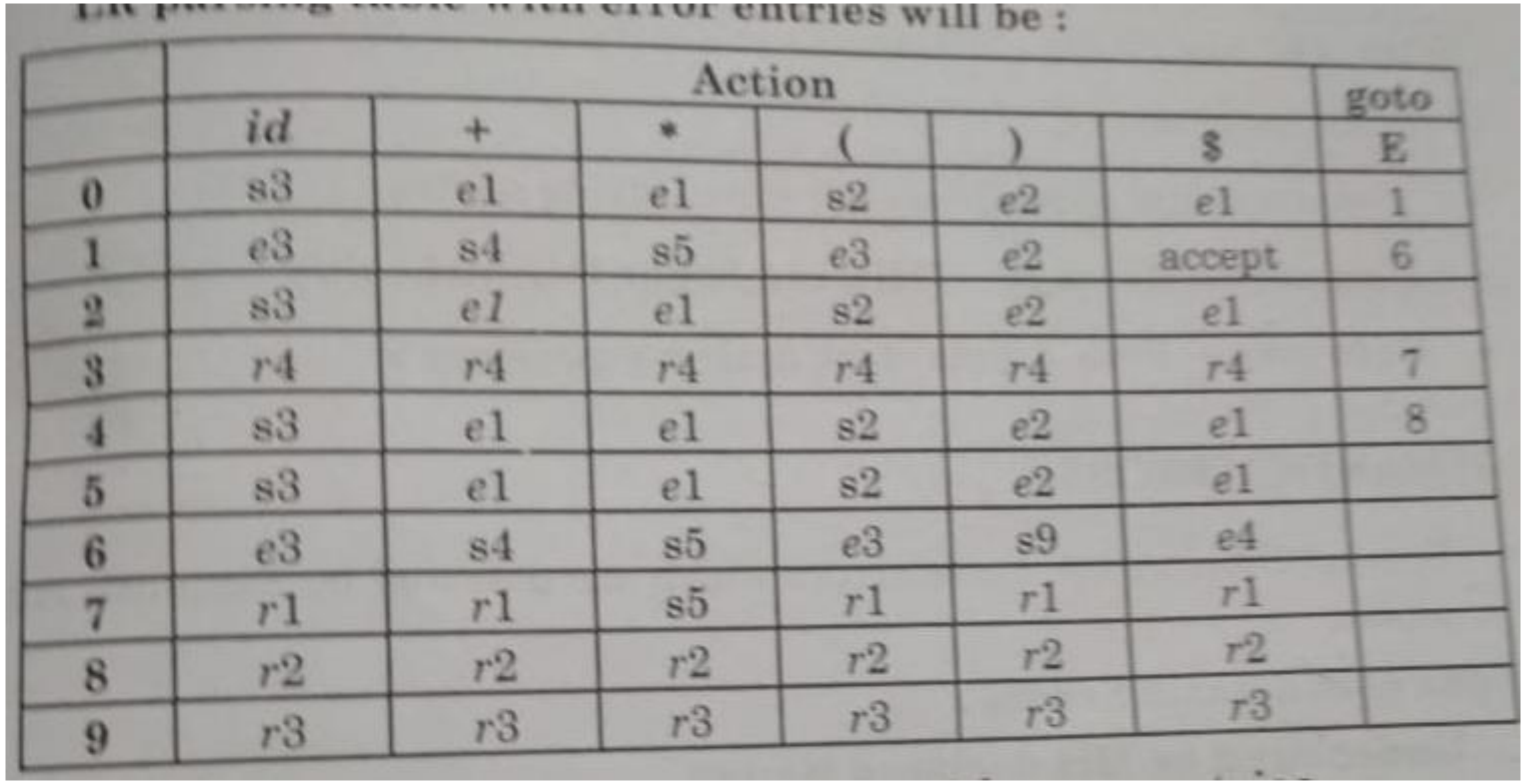
$E \rightarrow (E)$

$E \rightarrow id$

The Parsing table for this grammar will be

State	Action						goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			accept	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Error Recovery in LR Parsing



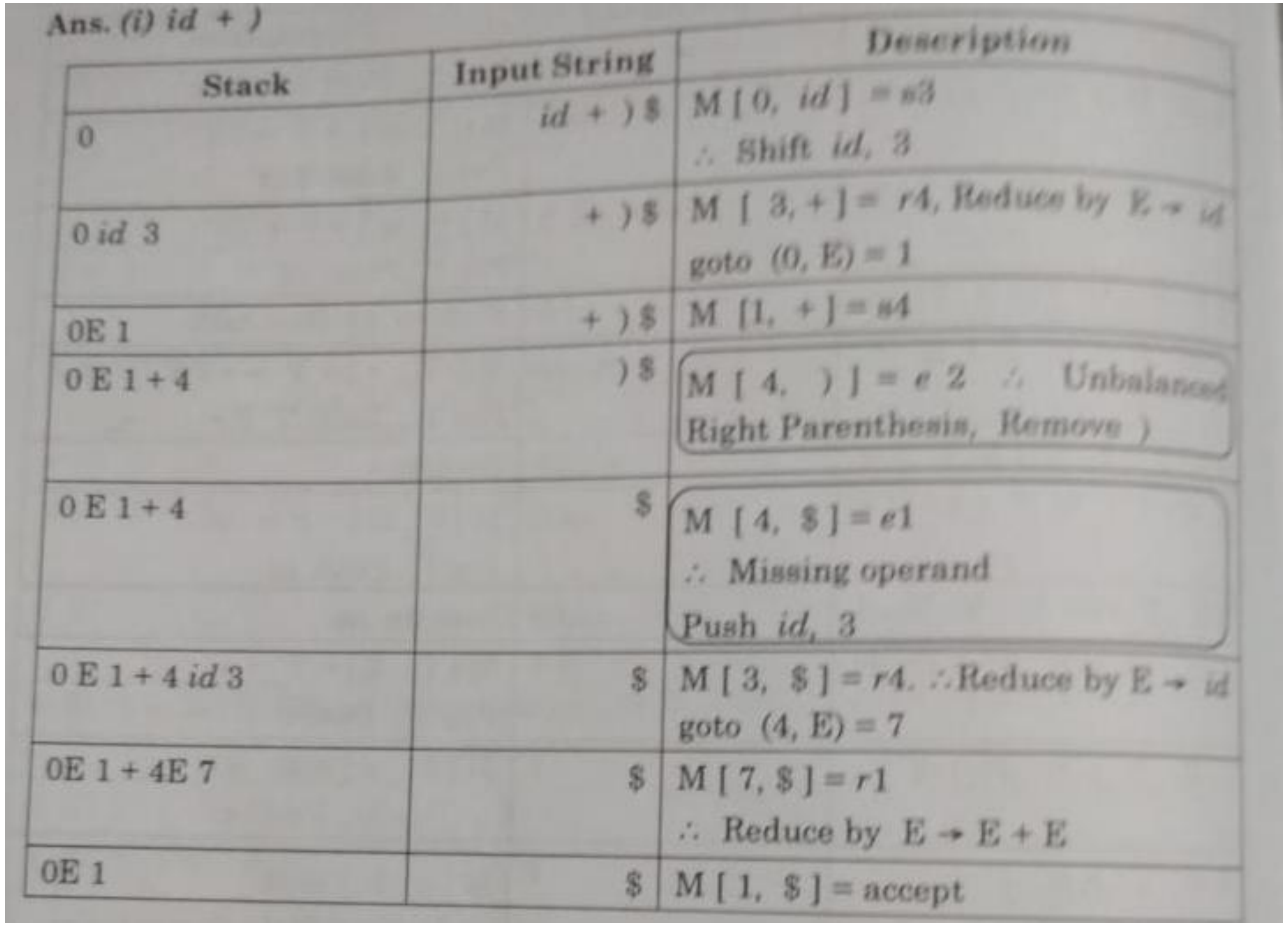
entries will be :

	Action						goto
	id	+	*	(	)	$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	accept	6
2	s3	e1	e1	s2	e2	e1	
3	r4	r4	r4	r4	r4	r4	7
4	s3	e1	e1	s2	e2	e1	8
5	s3	e1	e1	s2	e2	e1	
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Error Recovery in LR parsing

- If there are entries for r_j in some particular state then fil up all blank entries with r_j only.
- Error e1 occur when operand before operator is not available, diagnostic message is “Missing operand” and recovery method is push id onto stack and then push state 3. goto({0,2,4,5},id)=3
- Error e2 occurs in state when extra right parenthesis is there, diagnostic message is “Unbalanced right parenthesis” and recovery method is remove) from stack.
- Error e3 occurs when parser expect operator or id but right parenthesis appears, diagnostic message is “Missing operator” and recovery method is push + onto stack and then push state 4.
- Error e4 occurs when parser expects right parenthesis but end of string occurs, diagnostic message is “Missing right parenthesis” and recovery is by pushing) onto stack and then push state 9.

Error Recovery in LR Parsing



Ans. (i) *id* +)

Stack	Input String	Description
0	*id* +) $	M [0, *id*] = *s*3 ∴ Shift *id*, 3
0 *id* 3	+) $	M [3, +] = *r*4, Reduce by E → *id* goto (0, E) = 1
0E 1	+) $	M [1, +] = *s*4
0 E 1 + 4	) $	M [4,)] = *e* 2 ∴ Unbalanced Right Parenthesis, Remove)
0 E 1 + 4	$	M [4, $] = *e*1 ∴ Missing operand Push *id*, 3
0 E 1 + 4 *id* 3	$	M [3, $] = *r*4. ∴Reduce by E → *id* goto (4, E) = 7
0E 1 + 4E 7	$	M [7, $] = *r*1 ∴ Reduce by E → E + E
0E 1	$	M [1, $] = accept

(*ii*) (*id* + (* *id*)

Stack	Input String	
0	(*id* + (* *id*) $	
0 (2	*id* + (* *id*) $	
0 (2 *id* 3	+ (* *id*) $	
0 (2 E 6	+ (* id) $	
0 (2 E 6	+ (* id) $	
0 (2 E 6 + 4	+ (* id) $	
0 (2 E 6 + 4 (2	(* *id*) $	
0 (2 E 6 + 4 (2 *id* 3	* *id*) $	
	* *id*) $	M [2, *] = *e* 1 Push *id*, 3
0 (2E 6 + 4 (2 E 6		
0 (2E 6 + 4 (2E 6 * 5	* *id*) $	
0 (2 E 6 + 4 (2E 6 * 5 *id* 3	*id*) $	
0 (2 E 6 + 4 (2 E 6 * 5 E 8	) $	
0 (2 E 6 + 4 (2 E 6	) $	
	) $	
0 (2E 6 + 4 (2 E 6) 9	$	
0 (2 E 6 + 4 E 7	$	
0 (2 E 6	$	M [6, $] = *e* 4 Missing right Parenthesis, push)
0 (2 E 6) 9	$	
0 E 1	$	accept

Semantic Analysis Phase Errors

- Undeclared or mis-declared name: Use of identifier that was never declared will cause this kind of semantic error.

  Recovery from undeclared name is done by making an entry for that in the symbol table with a flag. Purpose of flag is to indicate that entry is done in response of semantic error rather than declaration.

- Type incompatibilities: This error occurs if operator is applied to an incompatible operand. x=y*z where x and z are integer type while y is Boolean type.

  Recovery is possible through type conversion or type casting.

- Missing Labels: Sometime label referenced in program are not defined.
- Incorrect Procedure call: This error occurs when wrong number or type of parameters are passed while calling procedure.