10

ERROR HANDLING

10.1 ERROR

Error : Programmers are human beings who generally make programming mistakes. The incorrect program written by human being contains error. It is very important to handle these errors to perform normal execution of program.

- Error Handling is part of compilation Process.
- Error detection and its reporting is important function of compiler.

10.1.1 Classification of Errors

Broadly, errors can be classified as :–

1. **Compile Time Errors :** Errors which are detected during compilation time.
2. **Run-Time errors (Dynamic Errors) :** Errors which are detected during execution of program.

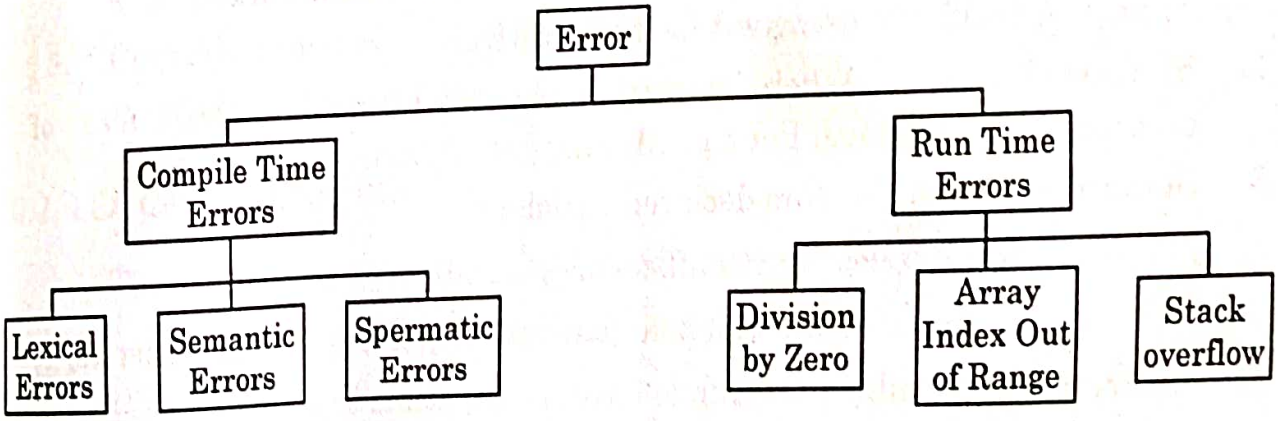


Fig 10.1 Classification of Errors

10.1.2 Types of Compiler on basis of Error Handling

1. **Simple Compiler :** After detection of error, it stop its all activities other than lexical & syntax analysis.
2. **Complex Compiler :** After detection of error, it repairs the error & then perform normal processing.
3. **Sophisticated Compiler :** After detection of error, it correct errors by making guess according to user intention.

10.1.3 Properties of Good Error Diagnostics

A good error handler should handle error messages as :–

1. Error message should be **easy to understand** by the user.
2. Error message should be specific and should **localize the problem**. For e.g., error message should be like "*x* is not declared in function" instead of just "missing declaration".
3. Error messages **should not be redundant** *i.e.* same message should not be produced again and again.
4. Error messages should be **in form of source program** rather than in some internal representation.
5. Error message should **locate** the line number at which **error** has occurred.

10.1.4 Errors detected at each phase

Errors occurred mainly at all phases of compiler.

1. **Lexical Phase :** When character sequences do not result in token.
   For *e.g.* identifier starting with a digit "5BE".
2. **Syntax Phase :** When structure of statements violates rules of programming language. For *e.g.* Missing paranthesis.
3. **Semantic Phase :** → Non declared variables
   → Type conflicts of operands
   → Incorrect procedure calls (*e.g.* wrong no. of parameters)
4. **Intermediate Code :** Operands have incompatible types.
5. **Code Optimization :** Some statements never reached *i.e.* Dead code.
   For *e.g.* Procedure which are never called.
6. **Code Generation :** → Too large constants to fit into registers.
   → Run out of memory

7. **Symbol Table Management :** Multiple declarations of a variable

When any of the phase detects error, it calls the error handler or corrector, which generates a diagnostic message.

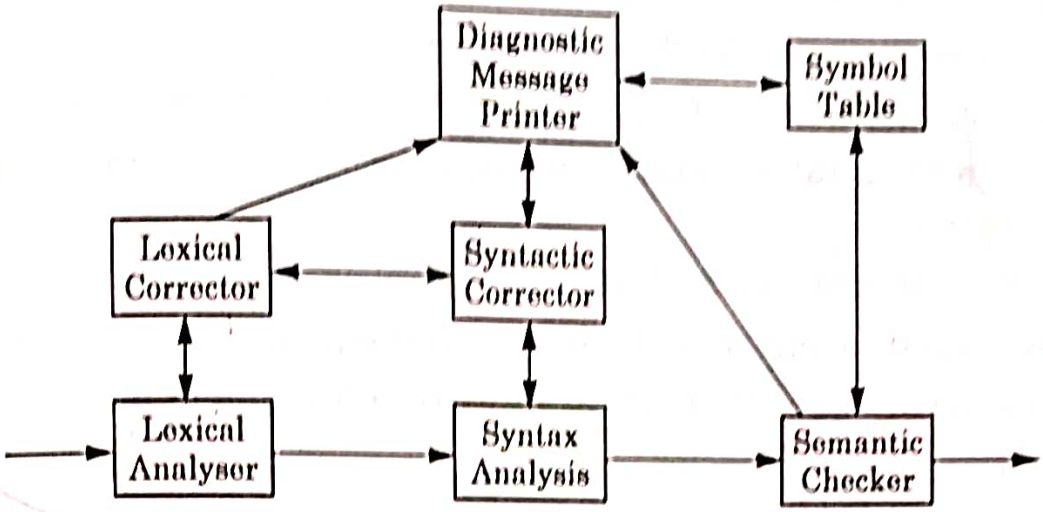


Fig 10.2 : Error Detection at each phase

10.1.5 Sources of Error in a Program

1. **Design Specification errors :** Program can be of poor or wrong design specification.
2. **Coding errors :** These errors are introduced by using improper programming language constructs by programmer.
3. **Algorithmic errors :** Algorithms design can be incorrect.
4. **Keypunching or Transcription errors :** Occurs when program is wrongly typed onto card or into file.
5. **Compiler errors :** Wrong Guess of a sophisticated compiler can insert errors into a program instead of correcting it.

10.2 LEXICAL PHASE ERRORS

- These are errors which are generated by lexical analyzer phase.
- Lexical Analyzer phase breaks the program into tokens *i.e.* it recognizes the tokens, in program.
- When the construct is recognized as not a token. It leads to lexical error.
- **Lexical Phase Errors :** If after some processing, the Lexical analyzer discovers that no prefix of the remaining input belongs to any token class, then lexical errors are generated.

→ Lexical phase errors are mainly generated because next token to be read in source program is misspelled.

Examples of Lexical Phase Errors :

→ In C ++, Java, digit cannot appear at beginning of identifier. For *e.g.* '0a'.

→ Integers constants out of bounds (0 to 32767).

→ Identifier names that are too long (max. length 32 characters).

For *e.g.* In FORTRAN, identifier whose length is more than 7 characters is a lexical error.

→ Text strings that are too long (max length 256 characters).

→ **Lexical errors occur mainly due to spelling mistakes. Some of main spelling errors in a program are due to following conditions :**

1. An extraneous character
2. An Incorrect character
3. A missing character
4. Two adjacent transposed characters

→ There are not many errors that can be caught at Lexical level.

→ Lexical Analyzer has no idea how tokens are grouped.

For *e.g.* Lexical Analyzer will not generate any error in following statement :

int a double } *b* = ;

As Lexical analyzer main task is to recognize tokens int, a, double, b, = .

It cannot detect their wrong grouping.

→ **Another example :**

Consider a statement in C program **wilhe (*a* < *b*).**

Here we have wrong spelling of keyword **while**. It is not possible for Lexical analyzer to tell whether wilhe is function, identifier or a keyword **while**. Lexical analyzer will assume it as valid identifier.

10.2.1 Recovery from Lexical Phase Errors

1. **Panic Mode Recovery :** It is an error recovery strategy. When a situation arises in Lexical analysis where it is unable to proceed. We can delete or skip successive characters from remaining input, until the Lexical analyzer finds another token. But this create difficulties for remaining phases.

2. **Minimum Distance Matching**
   - → As, Lexical error occur due to misspelling of tokens.
   - → The spelling of a token is corrected by choosing token which is closest to it alphabetically.

For *e.g.*, If we expect next token to be keyword "IF". But by Misspelling, we have written "FI". So, FI will be converted to IF by minimum distance matching.

For *e.g.* If we have written .EG instead of .GE for **Greater than Equal to**, Then .EG will be matched with all keywords. The Result will be .GE.

10.3 SYNTACTIC PHASE ERRORS

- → Errors detected during Syntax Analysis phase are called Syntax or Syntactic phase errors.
- → These errors occur when stream of tokens not follow the grammatical rules of programming language.
- → Syntactic errors can be detectable by Lexical or syntactic phase of compiler.

Examples of Syntactic Errors :-

1. Missing ; at end of statement
2. Missing Right Parenthesis
   $(a + (b * c)$
3. Misspelled Keyword
   BEGIN
4. Colon in place of Semicolon
   I = 1 :

- → Every Syntax Analyzer or Parser *i.e.* Shift-Reduce, operator Precedence, LL and LR Parser contain 2 components.
  - (*a*) Stack
  - (*b*) Input String

If parser do not have any legal move corresponding to top stack symbol and current input symbol, then it will generate syntactic error or syntax error.

10.3.1 Error Recovery Techniques

Following are various techniques for error recovery of syntactic phase errors :-

1. Panic Mode Recovery
2. Minimum Distance Recovery

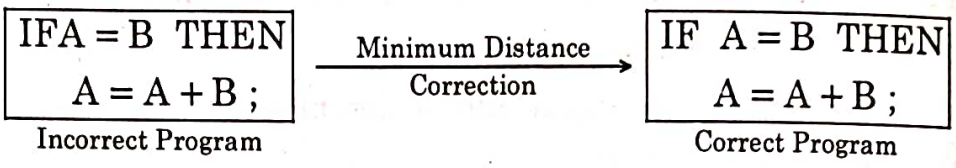
1. **Panic Mode Recovery :** Panic Mode Recovery is used in all the Parsers *i.e.* Operator-Precedence, LL and LR Parsers to recover from syntactic errors.

In Panic Mode, parser discards or skip input symbols until a synchronizing token *i.e.* ; or end is encountered. Parser then deletes stack entries until it finds an entry such that it can continue Parsing.

2. **Minimum Distance Correction :** It performs the least number of insertions, deletions and symbol modifications necessary to change incorrect program to correct or Legal Program.

Minimum Hamming Distance : If program P has K errors. So minimum number of error transformations that will change program P into correct one has length K.

For example :



Incorrect Program — Correct Program

Here, Minimum distance = 1, since *a* blank character of length 1 is added between IF and A, to make it correct.

Now we will discuss error recovery in each Parser :-

1. Error Recovery in Operator Precedence Parsing
2. Error Recovery in LL (1) Parsing
3. Error Recovery in LR Parsing

10.3.2 Error Recovery in Operator Precedence Parsing

Following are 2 main sources of syntactic errors in operator precedence Parsing :

1. Reduction Errors
2. Shift-Reduce Errors

1. **Reduction Errors :** These errors occur when handle has been found, but there is no production with this handle as a right side.

Consider a production

$S \rightarrow a\ B\ b\ c$

Consider a stack having *abc* at top of stack

If $<abc>$ is Handle found during parsing which can be Reduced. So, *abc* will be popped off stack with no surrounding non-terminals.

a
b
c

But instead of *abc*, we have *a B b c* on R.H.S. of Production. So, we cannot reduce *abc* to S.

Handling Reduction Errors : A diagnostic message can be generated because *abc* do not match with *a* B *bc*.

"Missing B on Line".

∴ B is missing on line *abc*.

Other Reduction Errors and their Handling :

(*a*) If any operator + or * is reduced, it checks operands on both sides of operator.

For *e.g.* If production is E → E + E and Handle to be reduced is <·E + ·>.

It will be handled by generating diagnostic message.

"Missing Expression"

(*b*) If two id's appear together to be reduced with no operator between them. Then message will be

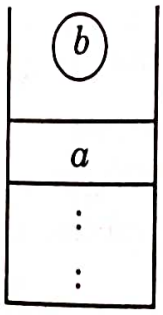
"Expressions not connected by operator"

(*c*) If ( ) appears to be reduced instead of (E). Then following message is generated.

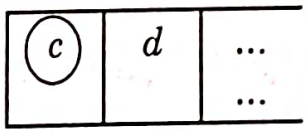
"Null Expression between Parenthesis"

2. **Shift-Reduce Errors :** These errors occur if no precedence relation exists between terminal on top of stack and current input symbol. There will be no shift or reduce if no relation exists between them.

For *e.g.* Consider following configuration.



Stack



Input String

If there is no precedence relation exists between *b* and *c* *i.e.* None of the relation like $b \lessdot c$, $b \gtrdot c$, $b \doteq c$ exists, then we cannot perform any shift or Reduce operation.

Handling Shift-Reduce Errors : If there is no precedence relation between b and c. But

1. If $a \lessdot c$, then delete or pop b from stack to compare a with c.

2. If $b \lessdot d$, then delete c from input string to compare b with d.

3. If there is symbol e such that $b \leq e \leq c$ then, insert e in front of c in input string.

The other method of Handling Shift-Reduce errors is by inserting error recovery routines (e_i) into blank entries in table.

Consider following operator precedence table with error entries.

When no precedence relation exists between two symbols, error recovery routine will be called.

	+	*	(	)	*id*	$
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	e1
)	$\gtrdot$	$\gtrdot$	e2	$\gtrdot$	e2	$\gtrdot$
id	$\gtrdot$	$\gtrdot$	e2	$\gtrdot$	e2	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	e3	$\lessdot$	e4

Fig. 10.3 : Operator Precedence Table with error entries

Here, *e1*, *e2*, *e3*, *e4* represent error entries

Error e1

- It occurs when expression ends with left parenthesis.
- **Diagnostic Message : "Illegal Left Parenthesis".**
- Recovery : Remove (from stack

For example : $ (*id* + *id* ($

Error e2

- It occurs when we have expressions like
  - *id* (
  - *id id*
  -) *id*
  -) (

} Common thing between these is missing operator between these symbols

Diagnostic Message : "Missing Operator".

Recovery : Insert + onto the input symbols

For *e.g.* In string $ *id* *id* $

On inserting +, it will become $ *id* + *id* $

Error e3

- It occurs when expression begins with right parenthesis.

Diagnostic Message : "Illegal right parenthesis".

Recovery : Delete) from input

For *e.g.* In string

$) *id* + *id* $

Error e4

- It occurs when there is no input.
- **Diagnostic Message : "Missing Expression".**
- Recovery : Insert *id* onto input

For *e.g.* $$

Things we have to Remember before doing error correcting operator Precedence Parsing :

1. Non Terminals (*e.g.* E, T, F) do not participate in precedence relations.
2. If top of stack is non-terminal, then terminal below top of stack will be compared with current input symbol.
3. Precedence Relation exists only between terminal symbols. So, Non-Terminals can be removed.

   i.e. $\lessdot$ E + $\gtrdot$ can be written as $\lessdot$ + $\gtrdot$

   or

   $\lessdot$ * F $\gtrdot$ can be written as $\lessdot$ * $\gtrdot$

10.3.3 Error Recovery in LL Parsing

Predictive Parser or LL Parsers gets into an error when :–

(*a*) The terminal symbol on top of stack does not match with next input character.

(*b*) When Parsing table entry M [X, *a*] is empty or blank where,
X = Non-terminal on top of stack
a = next input symbol

Error Recovery Methods in LL Parsing :

1. **Panic Mode →** When an error is encountered, symbols from the input string and non-terminals from top of the stack are deleted until the symbol on top of stack matches with current input symbol.
2. **Fill blank entries with error routines *i.e.* *e*1, *e*2, *e*3 etc.**

Consider the grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

The Predictive or LL Parsing table for the above grammar is

	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)		

Fig. 10.4 LL Parsing Table

The corresponding LL Parsing table with error entries are :

	id	+	*	(	)	$
E	E → TE′	*e*1	*e*1	E → TE′	*e*1	*e*1
E′	E′ → ε	E′ → + TE′	E′ → ε	E′ → ε	E′ → ε	E′ → ε
T	T → FT′	*e*1	*e*1	T → FT′	*e*1	*e*1
T′	T′ → ε	T′ → ε	T′ → * FT′	T′ → ε	T′ → ε	T′ → ε
F	F → *id*	*e*1	*e*1	F → (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

Fig. 10.5 LL Parsing Table with error entries

- **pop :** Pop entries in the table will be executed when the symbol at top of stack is same as current input symbol.
- $E' \rightarrow \varepsilon$ and $T' \rightarrow \varepsilon$ are added in some blank entries to postpone some error detection.

Various errors which are generated and their recovery method is given below :

Error e1

- It occurs when input string begins with any operator + or *,) and when end of input occurs with $.

Diagnostic Message : "Missing Operand".

Recovery : Push id onto input

Error e2

- It occurs when top of stack contains right parenthesis but input string does not contain it.

Diagnostic Message : "Missing Right Parenthesis"

Recovery : Pop Right Parenthesis from stack

Error e3

- It occurs when stack is empty, but there is still input symbol left on input string.

Diagnostic Message : "Unexpected input symbol".

Recovery : Remove remaining symbols from input string.

10.3.4 Error Recovery in LR Parsing

- LR Parsing table contains blank entries which are treated as errors.
- Each blank entry is examined and proper error correcting routine is called to recover from that error.
- Recovery technique used in LR Parsing refers to recovering from very common syntactic errors.

10.3.4.1 Recovery Techniques in LR Parsing

1. **Exhaustive Method :** Examine each error entry in parsing table and deciding which syntactic error in a program is related to it.
2. **Panic Mode :** Scan down the stack until a state with goto on a particular non-terminal A is found. Zero or more input symbols can then be discarded until a symbol is found which can follow A. Parser then stacks the state goto [s, A] and resumes normal processing.

3. **Error Recovery in Yacc :** Yacc refers to Yet Another Compiler Compiler. It generates LALR (1) parser. It has provided some features for error recovery in the form of error productions. Some extra rules known as error production may be added to the grammar like $A \rightarrow \alpha$ **error** β to take care of the erroneous inputs. Here, **error** is a predefined terminal symbol and α, β are any strings. When the parser encounters an error, it tries to use a relevant error-production by popping symbols from its stack until it finds the topmost state on its stack that contains an item of the form $A \rightarrow \alpha \cdot$ error β. The parser then resumes with the topmost state on its stack and shifts a fictitious token **error** onto the stack and reaches to state $A \rightarrow \alpha$ **error** $\cdot \beta$.

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		*r*4	*r*4		*r*4	*r*4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		*r*1	s5		*r*1	*r*1	
8		*r*2	*r*2		*r*2	*r*2	
9		*r*3	*r*3		*r*3	*r*3	

Fig. 10.6 : LR Parsing Table

Fill each blank entry in Action field with a pointer to error routine that will take appropriate action. The action can be insertion or deletion of symbols from stack or alteration or transposition of input symbols.

Rules :

1. If there are entries for r_j in some particular state then fill up all blank entries r_j only.

2. If there are shift entries only in some particular state then do not replace blank entries. Keep them as it is.
3. Change some blank entries by particular reduction rules. This change means we are postponing the error detection until one or more reductions are done.

LR parsing table with error 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*	

Fig. 10.7 : LR Parsing table with error entries.

Various Error Messages or Error routines are :–

Error *e1*

- It occurs in states 0, 2, 4, 5.
- It indicates operands should appear before operator.
- **Diagnostic Message – "Missing Operand"**
- **Recovery –** Push *id* onto stack and then push state 3.

∵ goto ({ 0, 2, 4, 5 }, *id*) = 3

Error *e2*

- It occurs in states 0, 1, 2, 4, 5.
- It indicates right parenthesis.
- **Diagnostic Message – "Unbalanced right Parenthesis".**
- **Recovery –** Remove) from stack.

Error *e3*

- It occurs for states 1 and 6.

→ It expect operator or *id* but right parenthesis appears.

→ **Diagnostic Message – "Missing operator"**

→ **Recovery** – Push + onto stack & then push state 4.

Error *e*4

→ It occurs at state 6.

→ It expects right parenthesis but end of string occurs.

→ **Diagnostic Message – "Missing Right Parenthesis".**

→ **Recovery** – Push) onto stack & then push state 9.

10.4 SEMANTIC ERRORS

→ Semantic Errors are those errors which are detected by Semantic Analysis phase.

Types of Semantic Errors :

1. Undeclared or Mis declared Names

For *e.g.* consider int Arr [10] ;

But, if programmer writes it as **Ar [i]** in some part of program. Then, this misspelling will generate an error because no declaration is found corresponding to Ar in Symbol Table.

So, use of identifier or Array that was never declared will cause semantic error.

Recovery : Recovery from undeclared name is done by making an entry for the name Ar into symbol table with appropriate attributes. A flag in symbol table entry is set to indicate that entry was made in response to semantic error rather than declaration.

2. Type Incompatibilities :

This error occurs, if operator is applied to an incompatible operand.

For *e.g.* $x = y * z$

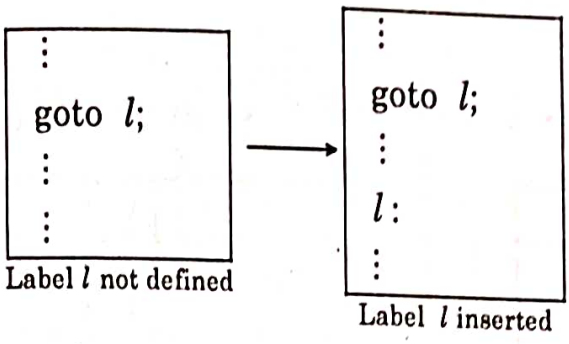
where y is of boolean type and x, z are integer types.

So, multiplication of different data types operands *i.e.* boolean with integer will cause semantic error.

Recovery : Type conversion (coercion) or Type Casting is Performed.

3. Missing Labels :

Sometimes, label table shows that some labels which have been referenced in the program are still not defined.



Recovery : To recover from such error, compiler has to find range in program which are accessible to those statements which have referenced the missing label and it has to select where missing label may be inserted.

4. **Incorrect Procedure Calls**

This error occurs when wrong number of parameters or wrong type of parameters are passed while calling procedure.

SOLVED PROBLEMS

Example 1. *Trace behaviour of operator precedence error correcting Parser on Input using figure 10.3.*

(a) id +)

*(b) (id + (* id)*

Ans. **(a)** ***id +)***

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

(b) (*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
$		$	

Example 2. *Trace the behaviour of LL error correcting Parsers on following inputs using figure 10.5.*

(i) $id +)$ *(ii)* $(id + (* id)$

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 ∴ Unexpected 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, *] = e1 ∴ Missing operand push *id* onto input
$ E′ T′) E′ T′) E	*id* * *id* $	M [E, *id*] = E → TE′

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 Parenthesis 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
$ E′	$	M [E′, $] = E′ → ε
$	$	accept

Example 3. *Trace the behaviour of LR error correcting Parser for following input strings :–*

(i) *id* +)

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

Ans. *(i) id +)*

Stack	Input String	Description
0	*id* +) $	M [0, *id*] = s3 ∴ Shift *id*, 3
0 *id* 3	+) $	M [3, +] = *r*4, Reduce by E → *id* goto (0, E) = 1
0E 1	+) $	M [1, +] = s4
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 E6 + 4 (2	* *id*) $	
0 (2 E 6 + 4 (2 *id* 3	* *id*) $	M [2, *] = *e*1 Push *id*, 3
0 (2E 6 + 4 (2 E 6	* *id*) $	
0 (2E 6 + 4 (2 E 6 * 5	*id*) $	
0 (2 E 6 + 4 (2E 6 * 5 *id* 3	) $	
0 (2 E 6 + 4 (2 E 6 * 5 E 8	) $	
0 (2 E 6 + 4 (2 E 6	) $	

Stack	Input String	
0 (2 E 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

KEY POINTS TO REMEMBER

- **Types of Error**
  (*a*) Compile Time (*b*) Run Time
- **Lexical Phase Errors** → These errors are detected by Lexical Analyzer Phase. If after some processing no prefix of remaining input belongs to any token class, then Lexical errors are generated.
- **Syntactic Phase Errors** → These errors occur when tokens do not follow grammatical rules of programming Language.
- **Errors Recovery Techniques**
  (*a*) Panic Mode Recovery
  (*b*) Minimum Distance Recovery
- **Semantic errors :**
  - Undeclared or Misdeclared names
  - Type Incompatibilities
  - Missing Labels
  - Incorrect Procedure calls

REVIEW QUESTIONS

Q 1. Explain in detail about :–
(*a*) Lexical Errors (*b*) Syntactic Errors (*c*) Semantic errors

Q 2. Explain the properties of good error Diagnostics ?

Q 3. Explain Error Recovery in LR Parsing ?

Q 4. Explain Error Recovery in operator Precedence Parsing ?

Q 5. Trace behaviour of operator Precedence, LR and LL error correcting Parsers for : * + *id*) + (*id* *

Q 6. Explain Minimum Distance Correction of Syntactic Errors.