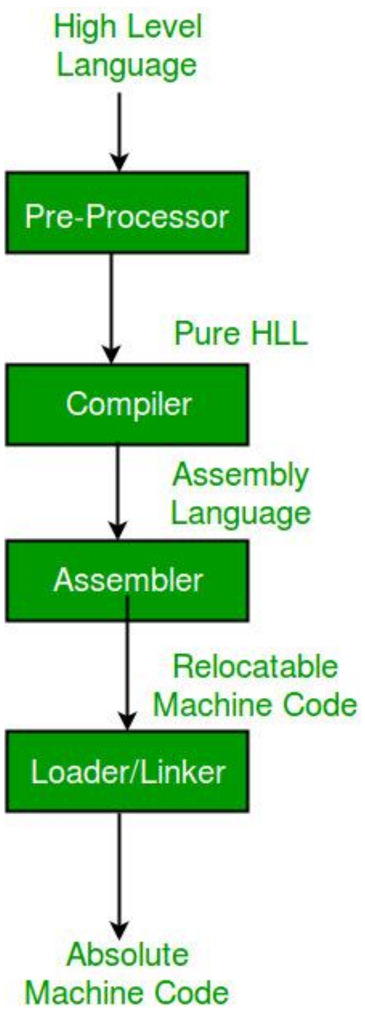
❑ Introduction to Compiler Design Course

Language Processing Systems

We know a computer is a logical assembly of Software and Hardware. The hardware knows a language, that is hard for us to grasp, consequently, we tend to write programs in a high-level language, that is much less complicated for us to comprehend and maintain in our thoughts. Now, these programs go through a series of transformations so that they can readily be used by machines. This is where language procedure systems come in handy.

- **High-Level Language:** If a program contains pre-processor directives such as #include or #define it is called HLL.
- **Pre-Processor:** The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing, etc.
- **Assembly Language:** It's neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler:** The output of the assembler is called an object file. It translates assembly language to machine code.
- **Compiler:** The compiler is an intelligent program as compared to an assembler. The compiler verifies all types of limits, ranges, errors, etc.
- **Interpreter:** An interpreter converts high-level language into low-level machine language, just like a compiler.
- **Relocatable Machine Code:** It can be loaded at any point and can be run.
- **Loader/Linker:** Loader/Linker converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message. Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.



❑ What Is a Compiler?

- ✓ A compiler is a translator that converts the high-level language into the machine language.
- ✓ High-level language is written by a developer and machine language can be understood by the processor.
- ✓ Compiler is used to show errors to the programmer.
- ✓ The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- ✓ When you execute a program which is written in HLL programming language then it executes into two parts.
- ✓ In the first part, the source program compiled and translated into the object program (low level language).
- ✓ In the second part, object program translated into the target program through the assembler.

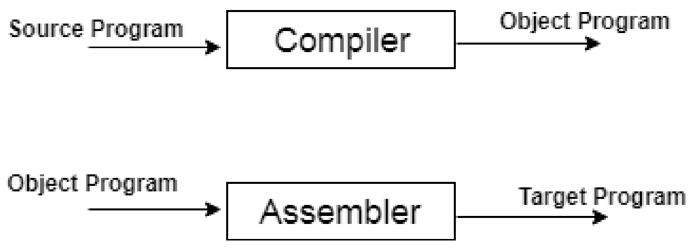
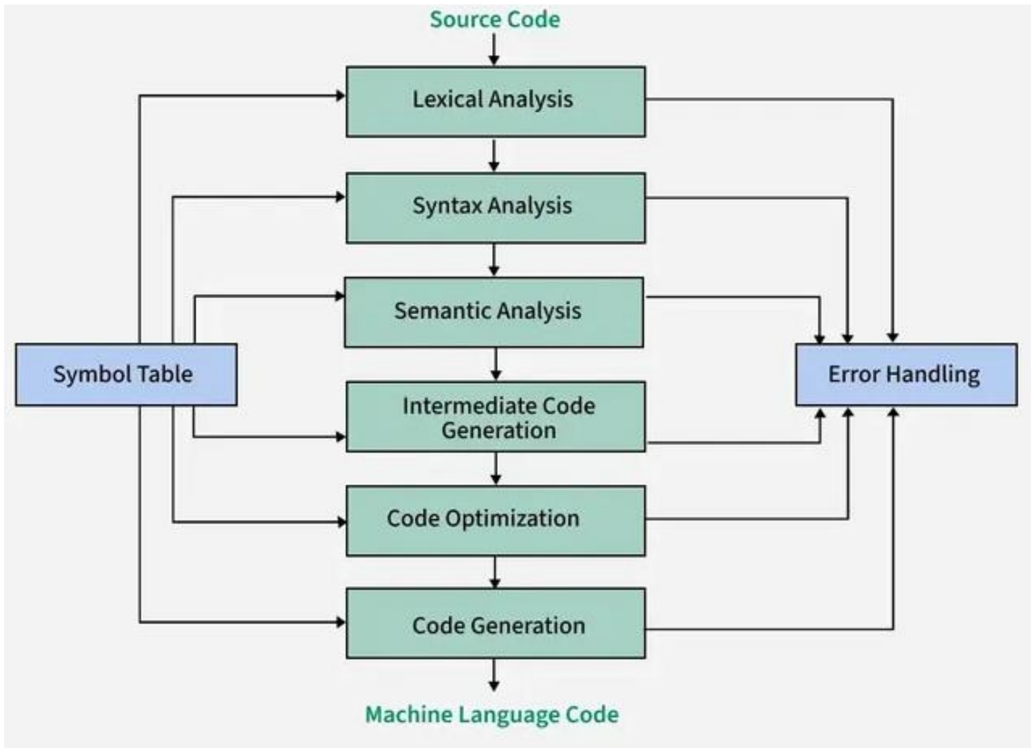


Fig: Execution process of source program in Compiler

❑ What Are the Phases of a Compiler?

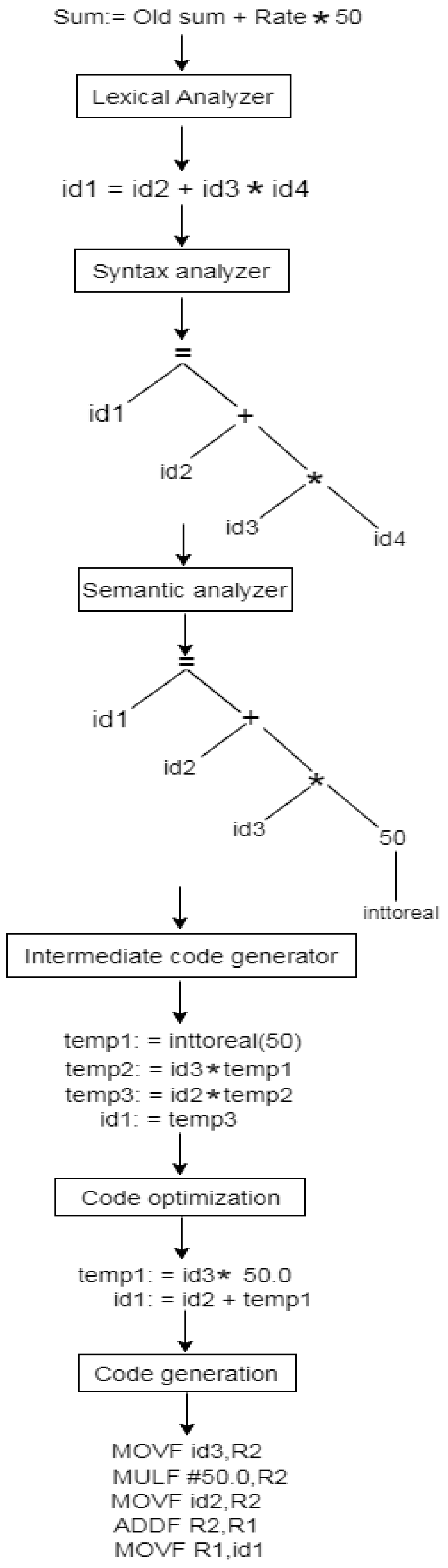
A compiler works in multiple stages, each performing a specific task to transform source code into machine code. Here are the six main phases of a compiler:

- Lexical Analysis– Breaks the source code into meaningful tokens.
- Syntax Analysis (Parsing)– Checks the structure of the code according to grammar rules.
- Semantic Analysis– Ensures logical correctness and detects type errors.
- Intermediate Code Generation– Converts code into an intermediate representation (IR).
- Code Optimization– Improves efficiency by refining the IR.
- Code Generation– Translates the optimized IR into final machine code.



❑ Working of Compiler Phases with Example

Example - 1: where sum, old sum, rate is a float variable

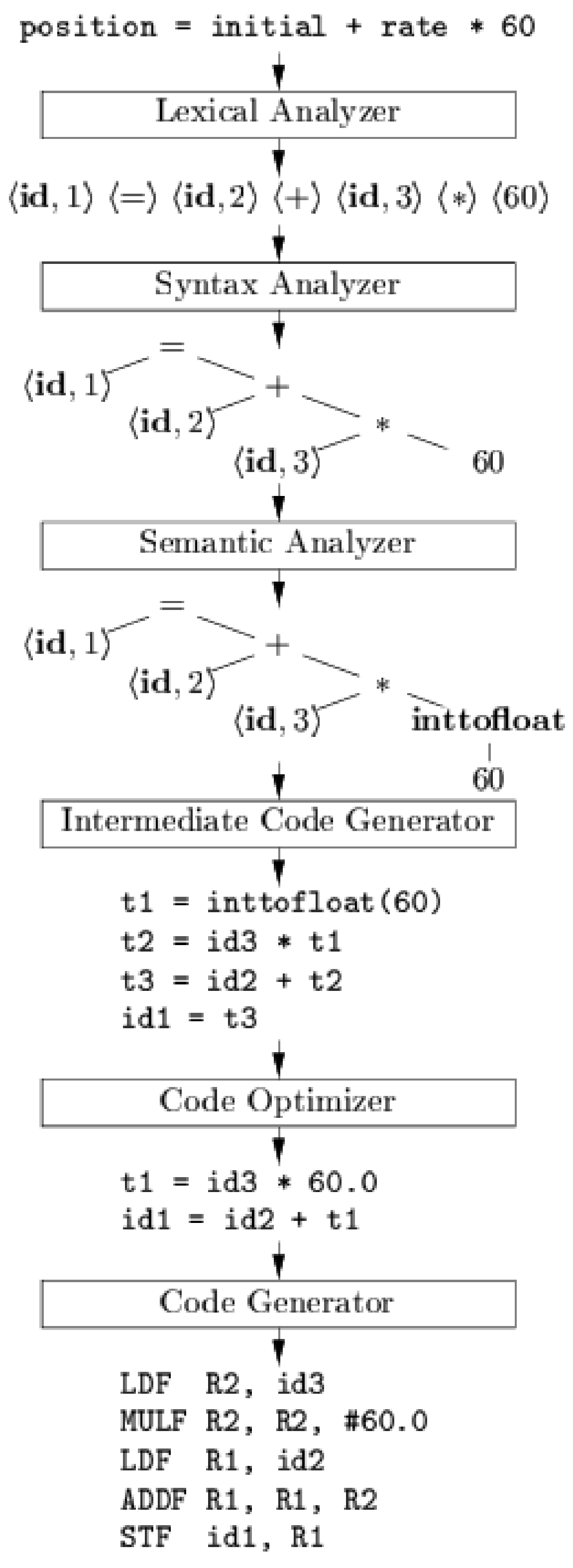


```
temp1: = inttoreal(50)
temp2: = id3∗temp1
temp3: = id2∗temp2
   id1: = temp3
```

```
temp1: = id3∗  50.0
   id1: = id2 + temp1
```

```
MOVF id3,R2
MULF #50.0,R2
MOVF id2,R2
ADDF R2,R1
MOVF R1,id1
```

Example - 2: : where position, initial, rate is a float variable

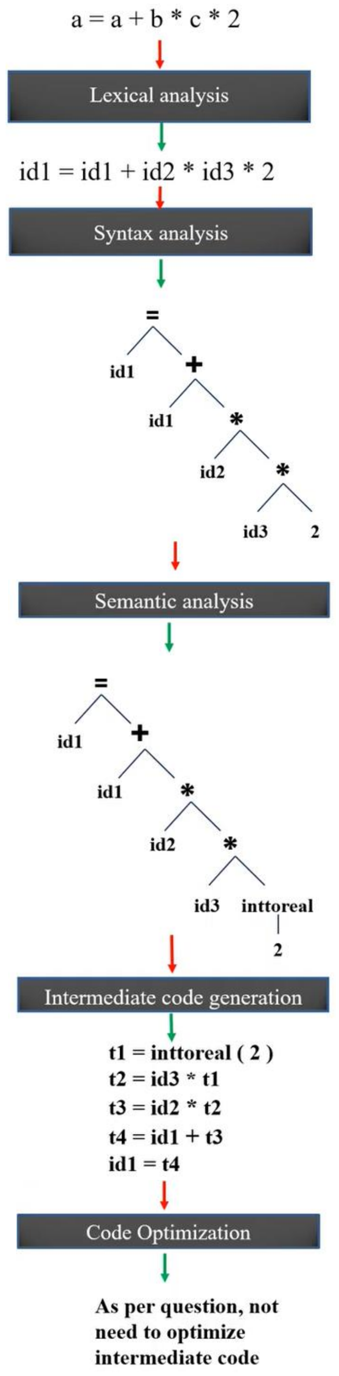


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Example -3: where a,b,c is a float variable



```
t1 = inttoreal ( 2 )
t2 = id3 * t1
t3 = id2 * t2
t4 = id1 + t3
id1 = t4
```

As per question, not need to optimize intermediate code

<u>Code Generation</u>

```
MOVF R1, #2
MOVF R2, id3
MULF R2, R1
MOVF R3, id2
MULF R3, R2
MOVF R4, id1
ADDF R4, R3
MOVF id1, R4
```

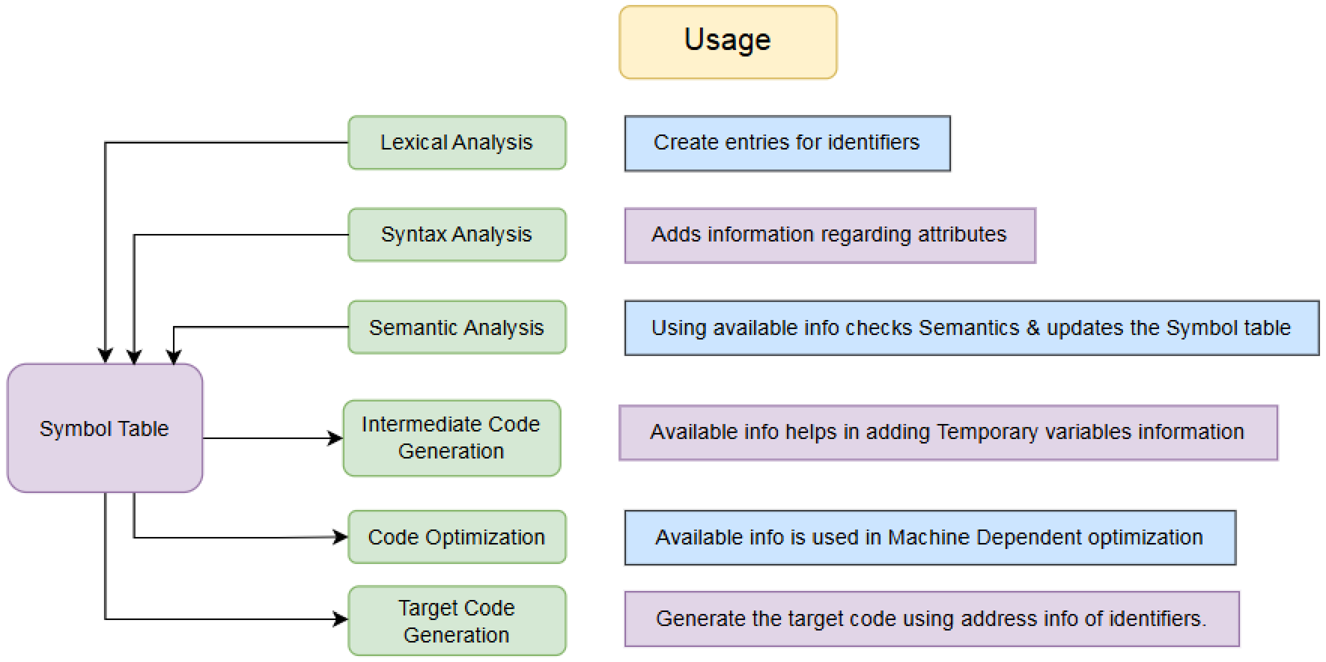
Example-4:
x = a+b*50, **where a,b,x is a float variable**
Solution: **Solve By yourself!**

Example – 5:
data = data + total/50,
where data, total is a float variable
Solution: **Solve By yourself!**

Questions for Exam: -
1. what are the phases of language processor?
2. what are the phases of compiler?
3. write down the Working of Compiler Phases for this following example.

❑ Role of Symbol Table in Compiler Phases

The symbol table acts as a bridge between the analysis and synthesis phases of the compiler. It collects information during the analysis phases and utilizes it during the synthesis phases to generate efficient code, ultimately enhancing compile-time performance.
It is used by various phases of the compiler as follows:-


Usage
Lexical Analysis
Create entries for identifiers
Syntax Analysis
Adds information regarding attributes
Semantic Analysis
Using available info checks Semantics & updates the Symbol table
Symbol Table
Intermediate Code Generation
Available info helps in adding Temporary variables information
Code Optimization
Available info is used in Machine Dependent optimization
Target Code Generation
Generate the target code using address info of identifiers.

Example of symbol table: -

1. int count
2. char ch[] = “cse”;
3. double matrix[2][2]={{1,1}, {2,2} }

Name	Type	Size	Dimension	Line of declaration	Line of usage	Memory address
count	int	4	0	-	-	-
ch	char	3	1	-	-	-
matrix	double	4	2	-	-	-

➢ **Imagine a program that includes a series of mathematical expressions, such as:**

A variable distance representing the distance traveled.
A constant pi representing the value of Pi.
A function calculateArea that computes the area of a circle.

Name	**Type**	**scope**	**Memory address**	**value**	**Additional Info**
distance	variable	Global	0x1000	Uninitialized	Data type: float
pi	constant	Global	0x1004	3.14159	Data type: float, read-only
calculateArea	function	Global	0x1008	N/A	Return type: float
radius	parameter	Local	0x2000	0x1000	Data type: float

Operations of Symbol table

The basic operations defined on a symbol table include

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Data Structures in Symbol Table

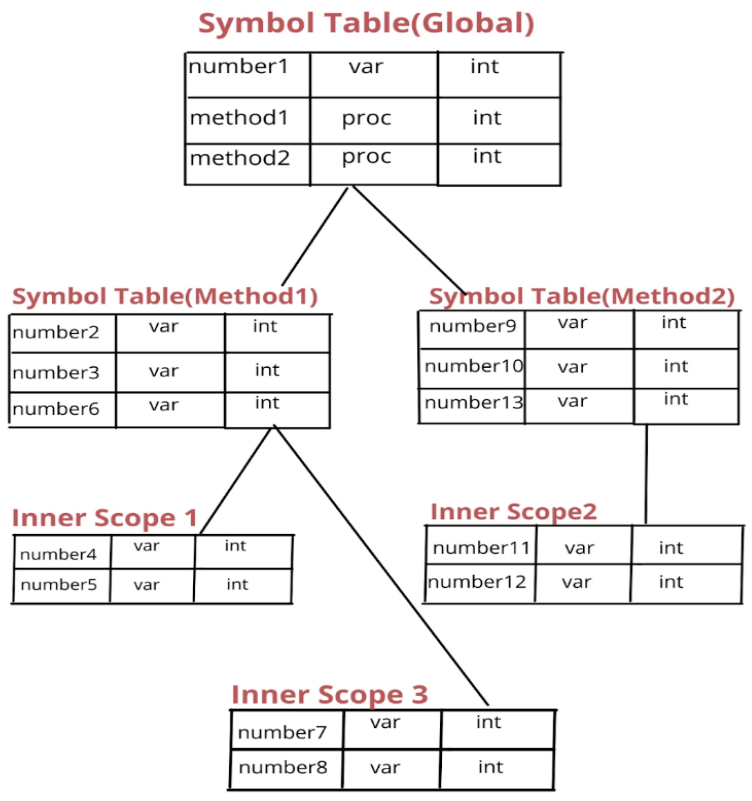
Data Structures used for the implementation of symbol tables are-

1.Binary Search Tree
2.Hash Tables
3.Linear search

A compiler contains two types of symbol tables: **global** and **scope** symbols tables. All the procedures and the scope symbol table can access the global symbol table while the scope symbol tables are created for each scope in the program. The scope of a name can be determined by arranging the table in the hierarchy structure as shown below:

```
int number1;
int method1()
{
  int number2;
  int number3;
  {
    int number4;
    int number5;
  }
  int number6;
  {
   int number7;
   int number8;
   }
 }
int method2()
{
  int number9;
  int number10;
  {
    int number11;
    int number12;
  }
  int number13;
}
```

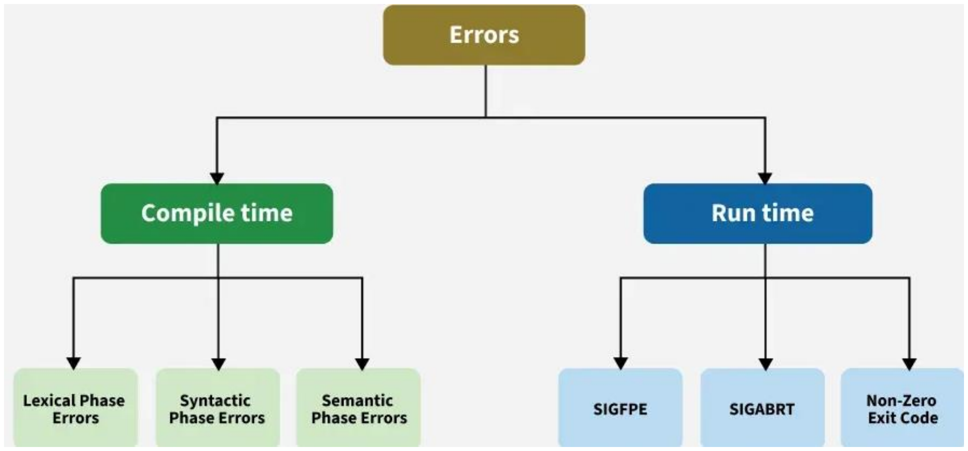
The above code can be arranged hierarchically in the following order:



❑ Error Detection and Recovery in Compiler

- ✓ Error detection and recovery are essential functions of a compiler to ensure that a program is correctly processed. **Error detection** refers to identifying mistakes in the source code, such as syntax, semantic, or logical errors. When an error is found, the compiler generates an error message to help the programmer fix it.
- ✓ **Error recovery** allows the compiler to handle errors gracefully without abruptly stopping the compilation process. It ensures that minor errors do not prevent the compiler from analyzing the rest of the program. Common recovery techniques include skipping incorrect parts, suggesting corrections, and continuing compilation.
- ✓ Effective error handling improves the debugging process, enhances code reliability, and helps developers write error-free programs efficiently.

➢ Classification of Errors

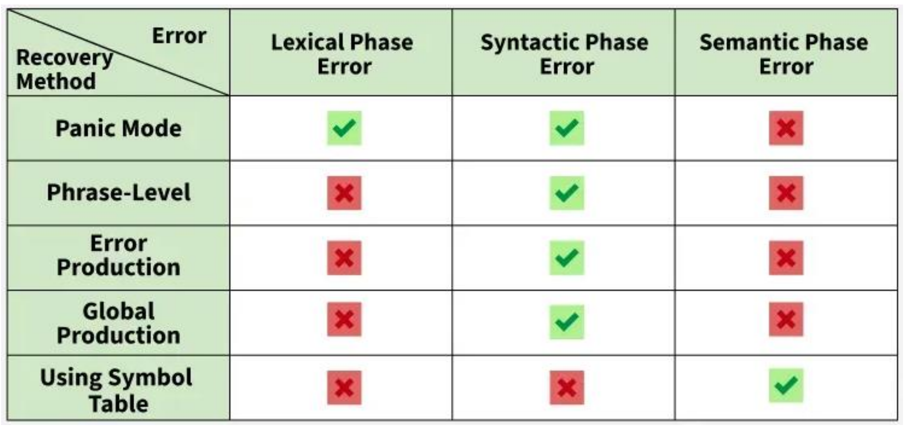


❑ Error Recovery

These methods help the compiler continue processing the code instead of stopping immediately.
Common recovery methods include:
Panic Mode Recovery – Skips erroneous code and resumes from the next valid statement.
Phase-Level Recovery – Replaces small incorrect code segments with valid ones.
Error Productions – Recognizes common errors and provides specific suggestions.
Global Correction – Makes multiple changes to fix errors optimally.

❑ Compile-time errors

Compile-time errors are of three types:-



Error / Recovery Method	Lexical Phase Error	Syntactic Phase Error	Semantic Phase Error
Panic Mode	✓	✓	✗
Phrase-Level	✗	✓	✗
Error Production	✗	✓	✗
Global Production	✗	✓	✗
Using Symbol Table	✗	✗	✓