Q3} Ans

In compiler design, error handling is crucial phase that ensures a compiler can detect & manage errors. Primary objective is to identify, report & if possible continue with compilation.

Types of errors

1) Lexical errors. eg int 2x= 10 → variable name doesn't begin with number

2) Syntax error eg if (a>b{-})

3) Semantic error eg int x= "abc";

4) Runtime error. eg division by zero

5) Logical error. eg code correct, logic incorrect.

Phases of error handling

1) Lexical analyser: Replace/ skip invalid characters.

2) Syntax analysis Use parse tree modifications

3) Semantic analysis Logs type mismatch / undeclared variable

4) Intermediate/ code generation: Recover by generating placeholder or use default code

Recovery strategies

1) Panic mode: Skip tokens until ~~an~~ a synchronising token is found

This is simple & fast, but might skip too much. Eg. if ')' is missing in 'if(x>10', then skip till next ')' or ';'.

2) Phrase level: Replace / insert token to repair old ones.
eg replace 'fi' with "if"

3) Error productions: Extend grammar to catch errors.
eg: add rule $E \to E+)$ to detect extra ')'.

4) Global correction: Modify entire program to catch minute errors. This is most accurate, but quite expensive.

Error reporting

This is of two types:

1) Immediate: Shows line number, error type etc.

2) Multiple errors: helps to solve the root cause by reporting all the places that lead to the error.

Q4) Ans

A compiler convert high-level source code to machine code. This allows the program to actually run.

A compiler must ensure:

1) Correctness (follows defined CFG)
2) Efficiency (generates optimized code)
3) Speed (fast translation)
4) Error detection (compile-time errors)

~~Phases of compiler~~

~~Input~~

Phases of compiler

1) **Lexical Analysis**
1) Input : Source code
2) Output : Tokens.
Converts code into meaningful units
eg int x = 10 ⇒ int id(x) = num(10);

2) **Syntax Analysis** : Input : Tokens Output : Parse Tree
Checks for proper grammar using CFG.
eg if (x>y) {} ✓ if (x>y} {). ✗

3) ~~Sema~~ **Semantic Analysis** : ~~Input~~: parse tree O/p : Symbol tree
Checks for meaning - type compatibility, undeclared variable
eg : checks for errors in int x = "hello";

4) **Intermediate Code Generation** : I/p ⇒ Annotated Tree O/p : Intermediate Representation (IR)
Produce easy-to-optimize machine code

5) **Code Optimization** : I/p ⇒ IR O/p ⇒ Optimized IR.
Remove redundancies, improve performance.

6) **Code generation** : I/p Optimized IR O/p ⇒ Machine code
Generate machine code.

Here, there is first machine independent code generation, optimization, and after that machine dependent part.
Compiler also does symbol table management & error handling.