# Error Detection and Recovery

## Features of an Error Reporter

The **Error Reporter** is a crucial component of a compiler responsible for identifying, reporting, and helping recover from errors in the source code.

---

### Key Features

1. **Accuracy**
   - Should report the **correct type and location** of errors.
   - Helps the programmer quickly identify and fix the issue.
2. **Clarity**
   - Error messages should be **clear, concise, and user-friendly**.
   - Should avoid cryptic or overly technical terms.
3. **Consistency**
   - Should use a **uniform format** for reporting different kinds of errors.
   - Makes it easier for users to understand and locate issues.
4. **Multiple Error Reporting**
   - Should attempt to **report as many errors as possible** in a single pass, rather than stopping at the first one.
   - Helps in comprehensive debugging.
5. **Error Classification**
   - Should distinguish between different types of errors:
     - Lexical errors
     - Syntax errors
     - Semantic errors
     - Runtime errors
6. **Position Indication**
   - Should show **line numbers**, **column numbers**, or even highlight the **offending token**.
7. **Suggestive Messages**
   - May provide **possible corrections** or hints to resolve the error.
8. **Support for Recovery**
   - Should integrate with **error recovery mechanisms** (like panic mode or phrase-level recovery) to continue parsing after an error is detected.

# Error Recovery Strategies

When an error is detected during compilation, the compiler must recover and continue processing to detect further errors. This is essential for generating useful feedback to the programmer.

The following are **major error recovery strategies**:

## 1. Panic Mode Recovery

### Description:

- The parser **discards input tokens** until a synchronizing token (like `;`, `}`, or `end`) is found.
- It then resumes parsing from that point.

### Characteristics:

- Simple and fast.
- Guarantees that parsing continues.
- May **skip large parts** of the input and miss subsequent errors.

### Example:

```
int x = 10    // missing semicolon
printf("Value");
```

Parser skips tokens until it finds `;` or `printf`.

## 2. Phrase-Level Recovery

### Description:

- The parser performs **local corrections** to the input, such as:
- Inserting a missing token
- Deleting an extra token
- Replacing a token with another
- It tries to **repair** the error and continue parsing.

### Characteristics:

- More precise than panic mode.
- Might introduce **incorrect assumptions** about the program.

- Increases compiler complexity.

## 3. Error Productions

### Description:

- The grammar is **augmented with additional rules** that account for common errors.
- When such a rule is matched, the parser generates a specific error message.

### Characteristics:

- Helps in catching **frequent or known mistakes** early.
- Requires knowledge of typical errors.
- Increases grammar size and complexity.

### Example:

```
stmt → if ( expr ) stmt | error ) stmt
```

Catches missing opening parenthesis in `if` statements.

## 4. Global Correction

### Description:

- The compiler **analyzes the entire input** and makes **minimum changes** to transform it into a valid program.
- Uses algorithms to compute the smallest set of insertions, deletions, or replacements.

### Characteristics:

- Highly accurate but **computationally expensive**.
- Not commonly implemented in real-world compilers.
- More useful in theoretical models or teaching tools.

| Strategy | Description | Accuracy | Complexity | Used In Practice |
|---|---|---|---|---|
| Panic Mode | Skip tokens until synchronizing point | Low | Low | Yes |
| Phrase-Level | Local corrections (insert/delete) | Medium | Medium | Yes |

| Strategy | Description | Accuracy | Complexity | Used In Practice |
|---|---|---|---|---|
| Error Productions | Add rules for common errors | Medium | Medium | Sometimes |
| Global Correction | Minimal edits for valid input | High | High | Rarely |

# Lex vs Yacc

| Feature | **Lex** | **Yacc** |
|---|---|---|
| **Full Form** | Lexical Analyzer | Yet Another Compiler Compiler |
| **Purpose** | Used for **lexical analysis** (tokenizing) | Used for **syntax analysis** (parsing) |
| **Input** | Regular Expressions | Context-Free Grammar |
| **Output** | C code for lexical analyzer | C code for syntax parser |
| **Generates** | `yylex()` function | `yyparse()` function |
| **Used For** | Breaking source code into tokens | Checking the grammatical structure of tokens |
| **Input File Extension** | `.l` | `.y` |
| **Integration** | Works with Yacc (provides tokens) | Works with Lex (receives tokens) |
| **Token Handling** | Identifies tokens and returns token codes | Receives tokens and builds parse tree |
| **Grammar Support** | Regular expressions | Context-free grammar |
| **Tool Type** | Scanner Generator | Parser Generator |
| **Common Language** | C (output is in C code) | C (output is in C code) |

## How They Work Together

1. Lex reads input and matches patterns defined using regular expressions.
2. On finding a token, it returns it to Yacc.

3. **Yacc** uses these tokens to match grammar rules and build the parse tree or abstract syntax tree (AST).

---

## Example Workflow

```
[Lex source (.l)] → Lex → C code for scanner → Token stream
[Tokens] → Yacc parser (.y) → Yacc → C code for parser → Parse Tree
```