CSC 3201 Compiler Construction

Department of Computer Science

SZABIST (Islamabad Campus)

Week 7 (Lecture 1)

Types of Errors

- Compile-Time Error
- Run-Time Errors
  - Lexical Errors
  - Syntactic Errors
  - Semantic Errors
  - Logical Errors

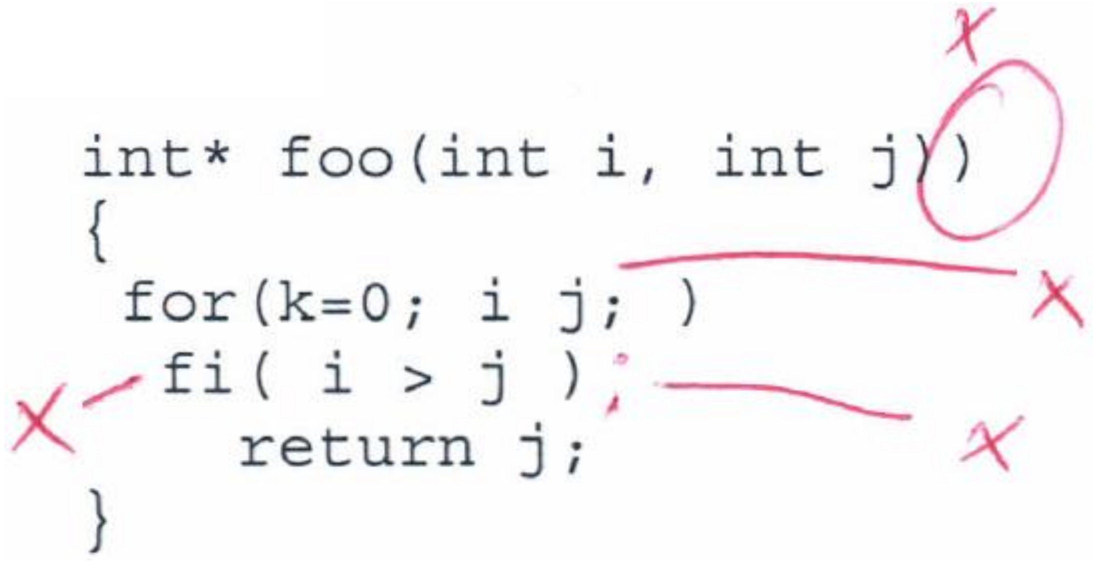
Lexical Errors

- A character sequence which is not possible to scan into any valid token.
- Rejected by the lexer.
- Generally results due to the failure of token recognition as per defined rules.
- Important facts:
  - Misspelling of identifiers, operators and keyword.
  - Appearance of illegal characters.
  - Exceeding length of identifier or numeric constants.

Syntax Errors

- Appear during syntax analysis.
- Typical Errors:
  - Errors in structure.
  - Missing operators.
  - Unbalanced/Missing parenthesis.

Syntax Errors

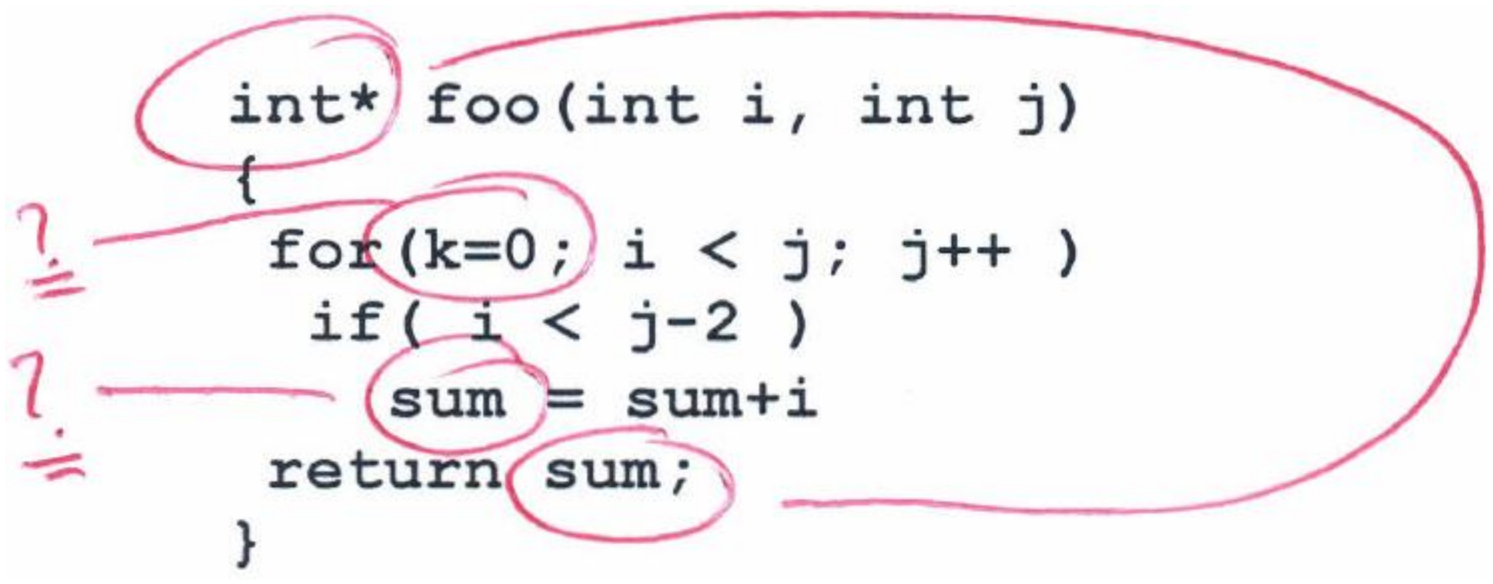


```
int* foo(int i, int j))
{
 for(k=0; i j; )
  fi( i > j );
    return j;
}
```

Semantic Errors

- Detected during semantic analysis.
- Typical Errors:
  - Incompatible types of operands.
  - Undeclared variables.

Semantic Errors



```
int* foo(int i, int j)
{
  for(k=0; i < j; j++ )
    if( i < j-2 )
      sum = sum+i
  return sum;
}
```

Logical Errors

- A bug in a program that causes it to operate incorrectly, but not to terminate abnormally (or crash).
- Produces unintended or undesired output or other behavior, although it may not immediately be recognized as such.
- Logic errors occur in both compiled and interpreted languages.

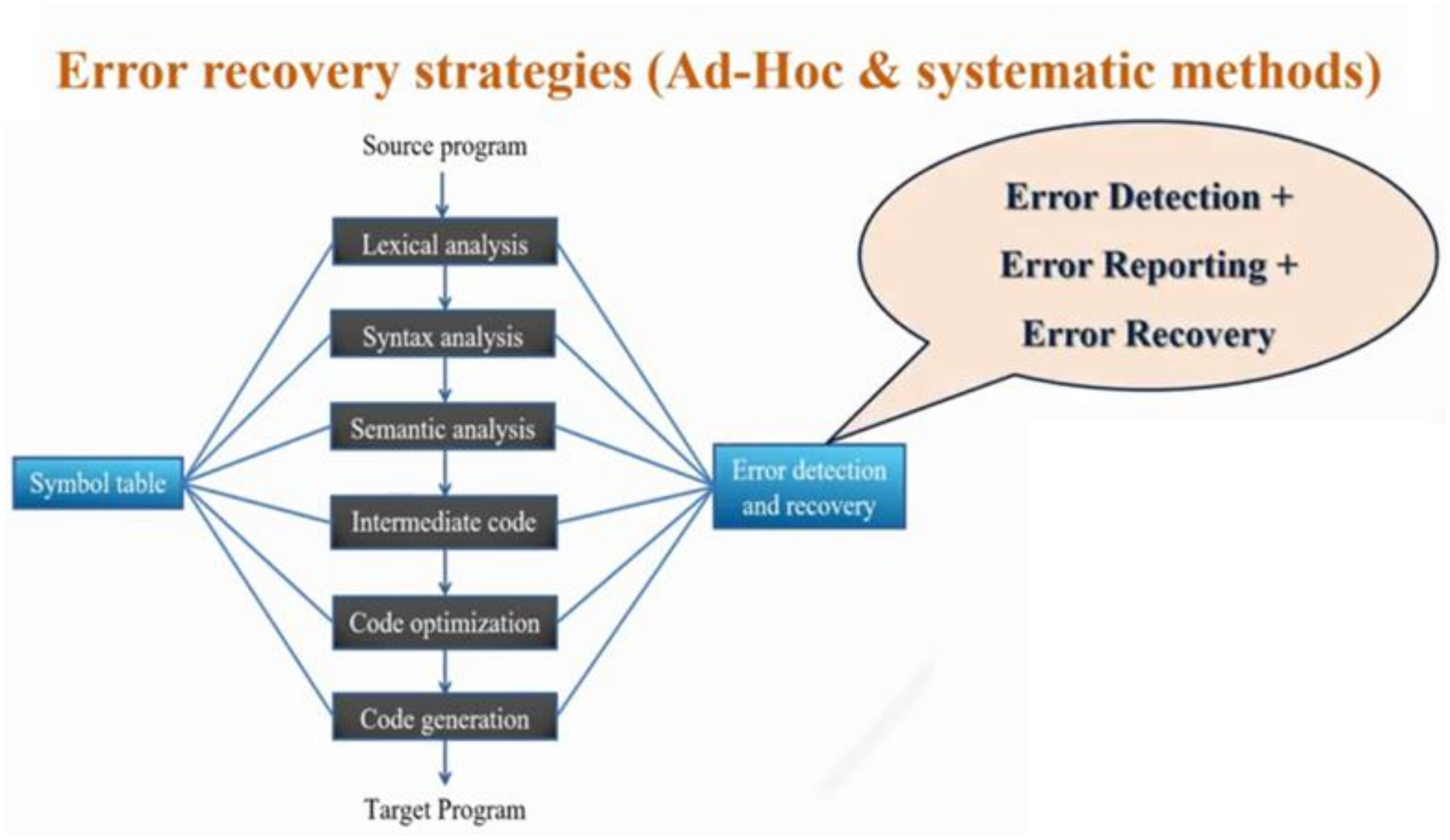
Logical Errors

- Unlike a program with a syntax error, a program with a logic error is a valid program in the language, though it does not behave as intended.
- Often the only clue to the existence of logic errors is the production of wrong solutions, though static analysis may sometimes spot them.
- Example: a+b*c instead of (a+b)*c.

Error-Recovery Strategies

- Goals:
  - Report the presence of errors clearly and accurately.
  - Recover from each error quickly enough to detect subsequent errors.
  - Add minimal overhead to the processing of correct programs.

Error-Recovery Strategies



Error-Recovery Strategies

- No strategy has proven itself universally acceptable, a few methods have broad applicability.
- Simplest Approach.
- Panic-mode Recovery.
- Phrase-level Recovery.
- Error Production.
- Global Correction.

Simplest Approach

- The parser quits with an informative error message when it detects the first error.
- Additional errors are often uncovered if the parser can restore itself to a state where processing of the input can continue with reasonable hopes that the further processing will provide meaningful diagnostic information.
- If errors pile up, the compiler gives up after exceeding some error limit.

Panic-mode Recovery

- Discards input symbols one at a time until a designated synchronizing tokens is found.
- The compiler designer must select the synchronizing tokens appropriate for the source language.
- Often skips a considerable amount of input without checking it for additional errors, the advantage is of simplicity.
- Guaranteed not to go into an infinite loop.

Panic-mode Recovery

- Example:
  - `int a, 7wk, num, #34;`
  - `int a,, num,;`

`

Phrase-level Recovery

- Performs local correction on the remaining input.
  - May replace a prefix of the remaining input by some string that allows the parser to continue.
  - Typical local corrections
    - Replace a comma by a semicolon.
    - Delete an extraneous semicolon.
    - Insert a missing semicolon.
- The choice of local correction is left to the compiler designer.

Phrase-level Recovery

- We must be careful to choose replacements that do not lead to infinite loops, as would be the case, for example, if we always inserted something on the input ahead of the current input symbol.
- Has been used in several error-repairing compilers, as it can correct any input string.
- Major drawback: Difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Phrase-level Recovery

- Example:
  - `int a,, num,;`
  - `int a, num;`

`

Error Productions

- Production rules for common errors.
- By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs.
- A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing.

Error Productions

- The parser can generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.
- Example: abcd cannot be derived

$$S \rightarrow A$$
$$A \rightarrow aA \mid bA \mid a \mid b$$
$$B \rightarrow cd$$

Augmented Grammar:

$$E \rightarrow SB$$
$$S \rightarrow A$$
$$A \rightarrow aA \mid bA \mid a \mid b$$
$$B \rightarrow cd$$

Global Correction

- Compiler should make as few changes as possible in processing an incorrect input string.
- There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.

Global Correction

- The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.
- When an erroneous input X is fed, it creates a parse tree for some closest error-free statement Y.
- May allow the parser to make minimal changes in the source code.

Global Correction

- Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible.
- Due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Global Correction

- A closest correct program may not be what the programmer had in mind. Nevertheless, the notion of least-cost correction provides a yardstick for evaluating error-recovery techniques, and has been used for finding optimal replacement strings for phrase-level recovery.