Types of Errors

- Static errors (detected before program runs) fall into 2 categories:

  – Lexical errors: detected by scanner – caused when tokens cannot be properly scanned.
    - Invalid character in program
    - Badly formed token
    - EOF in the middle of a token

  – Syntax errors: detected by parser – caused when a program does not follow the grammatical structure of the language
    - Expect a token and get a different one
    - Expect one of many non-terminals & don't get any
    - EOF in the middle of a production

  – Semantic errors: detected by semantic analyser – caused when a program does not follow the semantics of the language.

  Since programmers don't care about the difference, they are often simply all called syntax errors.

- Dynamic errors = runtime errors are detected when the program runs

- Examples: identify the type of error:
    - EOF reached before the end of a string →
    - Wrong sequences of types in list of function call parameters →
    - else does not match any if →
    - Invalid characters in program →
    - Missing ";" at the end of statement →
    - Type errors →
    - Name-matching problems (e.g function f() … end g; →
    - float x = 0.2Eb; →
    - Array index out of range →
    - Identifier used outside of its scope →
    - ; used to separate function parameters instead of , →
    - Division by 0 →

Components of Error Management

Error Prevention

Integrated Development Environments:

- Syntax directed editors can provide matching elements (e.g. closing ifs, loops, functions or intermediate delimiters such as *then*, *to,* etc.)
- Editor can also perform semantic error prevention: (e.g. enforcing function call definitions.)

Error Detection

Compiler/Interpreter will detect non-compliance & throw an exception

Error Reporting

Programmers (users) will want to know

- Where mistake happened
- What was expected
- What was found instead
- Why this is a mistake
- How it can be fixed
- Optionally: how program recovered from error.

Error Recovery

1) Compilers: when static errors are detected
   - Try to recover in order to detect as many errors as possible
   - Stop after a fixed number of errors
     - Because: error recovery could introduce new phantom errors which are not necessarily in the original program.
   - Do not generate code
2) Interpreters: consist of a parse-eval loop
   - Evaluation errors: stop evaluation and parse next structure
   - Parsing errors: try to recover to continue parsing
     - Must be able to detect where to restart, i.e. what should be thrown out
3) Integrated Development Environments:
   - Propose solutions to user and have them confirm change.
   - Debuggers can also support user-led investigation of and recovery from run-time errors by letting users step through program and try possible changes.

Parsing Error Recovery Strategies:

- Purpose: recover enough to be able to continue detecting more errors without introducing more errors in the process.
- All strategies are heuristics: solutions are not perfect, and not guaranteed to be correct but usually produce reasonable results.
- Strategy depends on type of error & where it occurs in the grammar.
  - Shallow error recoveries deal with input stream of tokens only without touching production stack
  - Deep error recoveries also pop production stack.

Shallow error recovery strategies:

- Obvious typos: Replace found token by expected token
  - E.g. f(a , b ; c) – replace by ,

  Note: token not really replaced. Instead alternate production is listed as acceptable but tagged as an error.

- Missing token: Insert missing token or placeholder
  - E.g. if (condition) ...
  - E.g. a = a + b

  Note: token not really inserted. Instead pop token from production stack. (it is considered a shallow recovery because it can be thought of as working with the stream of tokens)

- Panic Mode: Throw out all tokens until a good one is found from a set of synchronizing tokens.
  - E.g. things go wrong in the middle of parsing RHS of an assignment:
    - ASSIGN → identifier = SUM ;
    - SUM → identifier SUMMAND
    - SUMMAND → + identifier SUMMAND | ε
  - Parse a = b*c*d;
  - After reading b, trying to parse SUMMAND ; is next token on stack

  →Throw out all tokens until you reach ;

  Some heuristics to pick set of synchronizing tokens for non-terminal A:
    - Skip to an element of follow(A) & throw A out.
    - Skip to an element of first(A) & try to reparse A

<u>Deep error recovery strategies:</u>

- Pop stack until a reasonable production is found
  - E.g. Find an if even though previous statement is not finished
  → throw out what you were doing & start fresh new statement.

  How to find reasonable non-terminal on stack?
  Find a non-terminal A s.t. token ∈ First(A)

- Combine both
  - E.g. things go wrong in the middle of parsing RHS of an assignment: identifier = RHS ;
  - RHS complicated expression, gets confused & not working
  - Know that there is no ; in expressions
  - Pop stack until ; is on top
  - Throw out tokens until reach ;
  - Continue from there.