COSC 30403
Compiler Errors - Questions and Answers

Submitted to: Dr. Mei

Submitted by: Rahul Shrestha

22 January 2024

Exercise 1.1

Errors in a computer program can be classified according to when they are detected and, if detected at compile time, what part of the compiler detects them. Using C++, I give an example of each of the following.

Answers

(a) A lexical error, detected by the scanner

Answer: A lexical error occurs when invalid tokens are encountered during lexical analysis.
Example:

```
#include <iostream>
using namespace std;

int main() {

    printf("Dr.MeiIsReallyNice");$
    return 0;
}
```

Explanation: Since the statement ends with the illegal character $, this is a lexical error.

(b) A syntax error, detected by the parser

Answer: A syntax error occurs when the program structure violates the grammar of the language.
Example:

```
#include <iostream>
using namespace std;

int main() {
    cout << "HELLOOO THEREEEEEE" // Missing semicolon
    return 0;
}
```

Explanation: The statement is missing a semicolon (;) at the end.

(c) A static semantic error, detected by semantic analysis

Answer: A static semantic error is a logical fault that semantic analysis may detect during code compilation.
Example:

```
#include <iostream>
using namespace std;

int main() {
    string x = "Hello";
    int y = x + 5; // Static semantic error: invalid operation

    cout << y << endl;
    return 0;
}
```

Explanation: A string (`x`) cannot be added to an integer (`5`).

(d) A dynamic semantic error, detected by code generated by the compiler

Answer: A dynamic semantic error occurs during runtime when an invalid operation is performed.
Example:

```
#include <iostream>
using namespace std;

int main() {
    int x = 5;
    int y = 0;
    int result = x / y; // Dynamic semantic error: Division by zero

    cout << "Result: " << result << endl;

    return 0;
}
```

Explanation: Division by zero will raise a runtime error.

(e) An error that the compiler can neither catch nor easily generate code to catch

Answer: An error that cannot be caught by the compiler due to limitations in static analysis or the language definition.
Example:

```
#include <iostream>
using namespace std;

int main() {
    int a = 5, b = 10;
```

```
    // Intended to swap a and b, but logic is incorrect
    a = b + a; // a becomes 15
    b = a - b; // b becomes 5 (correct)
    a = a + b; // a becomes 20 (wrong!)

    cout << "a: " << a << ", b: " << b << endl; // Incorrect output

    return 0;
}
```

Explanation: This type of error, like incorrect swapping logic, occurs because the compiler cannot understand the programmer's intent, and the code is correct but produces unintended results.

Exercise 1.2

Consider again the Pascal tool set distributed by Niklaus Wirth (Example 1.15). After successfully building a machine language version of the Pascal compiler, one could in principle discard the P-code interpreter and the P-code version of the compiler. Why might one choose not to do so?

Answer: The P-code interpreter and the P-code version of the compiler should not be discarded for the following reasons:

- **Portability:** The P-code processor simplifies the operation of Pascal programs on multiple machines. The P-code is common, so we don't need to rebuild the machine-language translator for each new platform.
- **Development Flexibility:** If the machine-language compiler has not been ported or is not accessible, Pascal programs can still run via the P-code interpreter.
- **Testing and Debugging Made Easy:** Before converting to machine language, Pascal programs can be easily tested and solved in a controlled environment using the P-code interpreter.
- **Efficiency in the Initial Stages:** Porting the entire machine-language compiler is considerably more challenging than translating the P-code interpreter to an entirely different platform.

Conclusion: In the initial stages of coding or for architectures with limited support for machine code, retaining the P-code interpreter and compiler gives significant advantages in terms of flexibility, troubleshooting, and system adaptability.

Exercise 1.9

Why is it difficult to tell whether a program is correct? How do you go about finding bugs in your code? What kinds of bugs are revealed by testing? What kinds of bugs are not?

Answer:

(a) Why is it difficult to tell whether a program is correct?

Because requirements may be complicated, it can be challenging to figure out whether the program is correct:

- Programs are capable of handling a limitless amount of inputs, not all of which can be tested.
- Logical errors may not lead to obvious failures.
- Programs depend on outside components, which may have unexpected failures.

(b) How do you find bugs in your code?

To find bugs:

- Create test cases, even edge cases, and run them.
- To find errors that have been missed, do code reviews.
- To monitor the actions of programs during execution, add logs.

(c) What kinds of bugs are revealed by testing?

Testing can reveal:

- Runtime errors like crashes.
- Logic errors, such as incorrect calculations and infinite loops.
- Failures with inputs containing edge cases.

(d) What kinds of bugs are not revealed by testing?

Testing may miss:

- Errors in untested sections of the code.
- Bugs very specific to certain hardware or environments.
- Ignored logic errors that produce incorrect but believable outputs.