Assignment 2: Syntax, Semantics, and Memory Management

Haeri Kyoung

University of the Cumberlands

MSCS-632-A01 – Advanced Programming Languages

Professor Jay Thom

Aug 30, 2025

Part 1: Analyzing Syntax and Semantics

Section 1: Syntax Errors

For this section, I modified simple programs in Python, JavaScript, and C++ to introduce syntax mistakes on purpose. The goal was to observe how each language reports errors.

Python (py_sum.py)

- I left out a colon in the function definition and used o instead of 0.
- The interpreter immediately stopped with a SyntaxError: expected ':'.
- Once the colon was fixed, it reached runtime and reported NameError: name 'o' is not defined.
- This shows that Python halts execution at the very first parse problem and only exposes runtime issues afterward.

JavaScript (js_sum.js)

- I intentionally broke the function call identifier with a space (calculate Sum) and left out a closing brace.
- Running the code in Node.js gave SyntaxError: Unexpected identifier at the bad call.
- After fixing that, the engine then complained about the missing brace with SyntaxError: Unexpected end of input.
- JavaScript's parser is strict, and like Python, it reports only the first blocking error at a time.

C++ (cpp_sum.cpp)

- I replaced 0 with o and left out a semicolon after a return statement.
- Compiling with g++ reported two errors:

    - 'o' was not declared in this scope
    - expected ';' before '}' token
- Unlike Python or JavaScript, C++ tends to produce multiple messages at once, and one small mistake can cause cascading errors until the compiler recovers.

Comparison

Python and JavaScript stop at the first fatal parse issue and give short SyntaxError messages. C++ compilers are more verbose, often printing multiple related errors with caret pointers.

Section 2: Scopes, Closures, and Semantics

Here I wrote small closure-based programs in Python, JavaScript, and C++ to analyze scoping and typing.

Python (py_closure.py)

- Functions inside a loop capture names by reference, so without adjustments, all closures would use the final loop variable.
- By setting k=k as a default argument, I pinned the value for each closure.
- The result was correct: 10 11 12.

JavaScript (js_closure.js)

- With let, each iteration of the loop creates a new block scope, so closures capture the correct value.
- The output was also 10 11 12.
- If I had used var, all closures would have shared one scope and produced 12 12 12.

C++ (cpp_closure.cpp)

- I used lambdas with capture by value [k].

- This produced the expected 10 11 12.
- If captured by reference [&k], all lambdas would have seen the same final value.

Key Semantic Differences

1. Typing – Python and JavaScript are dynamically typed, catching type errors only at run-time. C++ is statically typed and enforces checks at compile time.
2. Closures – Python requires workarounds like default arguments to avoid late binding, while JavaScript with let and C++ with explicit capture behave more predictably.
3. Performance – C++ compiles ahead of time and optimizes aggressively. Python and JavaScript rely on interpreters or JITs, which trade some performance for flexibility.

Part 2: Memory Management

Section 3: Dynamic Allocation and Freeing Memory

I created short programs in Rust, Java, and C++ to show how each handles memory.

Rust (rs_memory.rs)

- Rust uses ownership and borrowing to enforce safety.
- In my example, a vector and a boxed integer are created, and ownership is transferred.
- When I tried to reuse a moved value, the compiler refused to compile, preventing a dangling pointer at runtime.
- Memory is freed automatically when variables go out of scope.

Java (JavaMemory.java)

- Objects are allocated on the heap with new.
- When I nulled out the list reference and requested System.gc(), the garbage collector reclaimed space.

- Java relies on garbage collection, which simplifies memory safety but makes the exact time of freeing unpredictable.

C++ (cpp_memory.cpp)

- First I allocated an array with new and had to remember to call delete[]. Forgetting this would leak memory.
- Then I rewrote the code using unique_ptr. This freed memory automatically at scope exit, showing the advantage of RAII (Resource Acquisition Is Initialization).
- C++ provides full control, but mistakes like mismatched new[]/delete can cause leaks or corruption.

Comparison

- Rust prevents misuse at compile time, rejecting invalid memory access before the code runs.
- Java delegates responsibility to the garbage collector, which prevents most leaks but cannot stop all retention issues.
- C++ gives direct control, rewarding careful programmers with high performance but punishing mistakes with leaks or crashes.

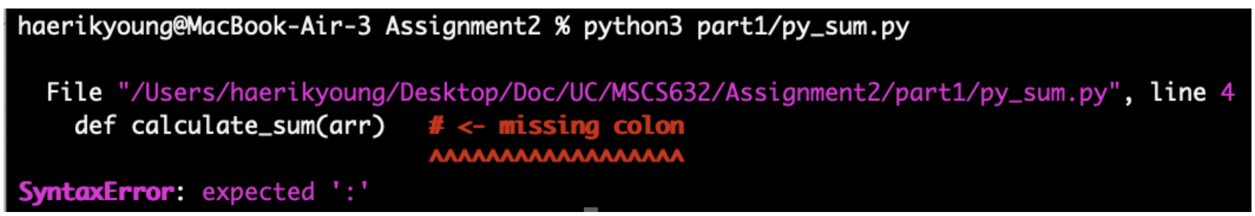
Conclusion

Across both parts of this assignment, I observed how programming languages differ in error handling, closures, typing, and memory management. Python and JavaScript provide concise error feedback and flexible closures but defer type checks until runtime. C++ is stricter, verbose in its diagnostics, and gives fine-grained control at the cost of potential mistakes. Rust, Java, and C++ illustrate three distinct philosophies for memory management: compile-time safety, automatic garbage collection, and manual control with optional RAII.

This exercise reinforced how language design choices directly shape developer experience, program safety, and performance. By comparing across paradigms, I gained a clearer understanding of why some languages trade speed for safety and why others demand more discipline from developers.

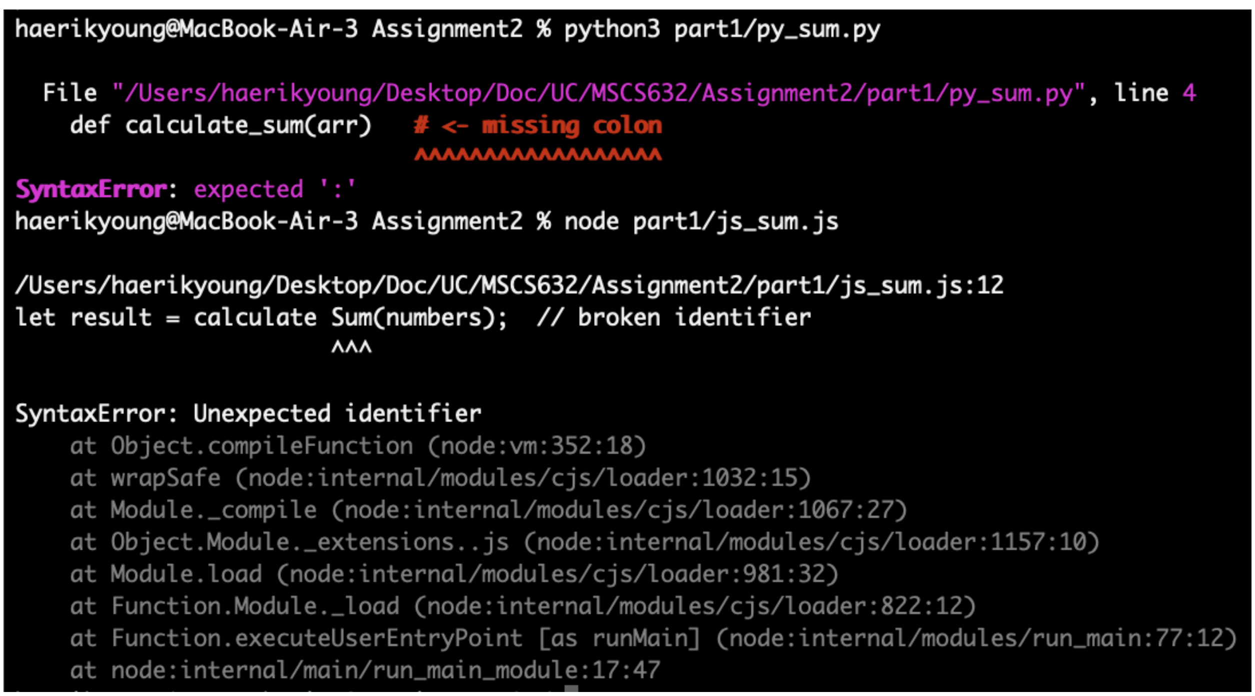
GitHub Repository: https://github.com/hkyoung38554/MSCS632_assignment2

Screenshots:



```
haerikyoung@MacBook-Air-3 Assignment2 % python3 part1/py_sum.py

  File "/Users/haerikyoung/Desktop/Doc/UC/MSCS632/Assignment2/part1/py_sum.py", line 4
    def calculate_sum(arr)    # <- missing colon
                              ^^^^^^^^^^^^^^^^^^
SyntaxError: expected ':'
```



```
haerikyoung@MacBook-Air-3 Assignment2 % python3 part1/py_sum.py

  File "/Users/haerikyoung/Desktop/Doc/UC/MSCS632/Assignment2/part1/py_sum.py", line 4
    def calculate_sum(arr)    # <- missing colon
                              ^^^^^^^^^^^^^^^^^^
SyntaxError: expected ':'
haerikyoung@MacBook-Air-3 Assignment2 % node part1/js_sum.js

/Users/haerikyoung/Desktop/Doc/UC/MSCS632/Assignment2/part1/js_sum.js:12
let result = calculate Sum(numbers);  // broken identifier
                       ^^^

SyntaxError: Unexpected identifier
    at Object.compileFunction (node:vm:352:18)
    at wrapSafe (node:internal/modules/cjs/loader:1032:15)
    at Module._compile (node:internal/modules/cjs/loader:1067:27)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1157:10)
    at Module.load (node:internal/modules/cjs/loader:981:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47
```

```
haerikyoung@MacBook-Air-3 Assignment2 % /opt/homebrew/bin/g++-15 -std=c++17 part1/cpp_sum.cpp -o cpp_sum

part1/cpp_sum.cpp: In function 'int calculateSum(int*, int)':
part1/cpp_sum.cpp:7:17: error: 'o' was not declared in this scope
    7 |     int total = o;             // 'o' not declared
      |                 ^
part1/cpp_sum.cpp:11:17: error: expected ';' before '}' token
   11 |     return total               // <- missing semicolon
      |                 ^
      |                 ;
   12 | }
```

```
haerikyoung@MacBook-Air-3 Assignment2 % python3 part1/py_closure.py

10 11 12
```

```
haerikyoung@MacBook-Air-3 Assignment2 % node part1/js_closure.js

10 11 12
```

```
haerikyoung@MacBook-Air-3 Assignment2 % /opt/homebrew/bin/g++-15 -std=c++17 part1/cpp_closure.cpp -o cpp_closure && ./cpp_closure

10 11 12
```

```
haerikyoung@MacBook-Air-3 Assignment2 % rustc part2/rs_memory.rs -o rs_memory
./rs_memory

len=5 first=0
boxed=42
moved=42
```

```
haerikyoung@MacBook-Air-3 Assignment2 % javac part2/JavaMemory.java   # compiles to part2/JavaMemory.class
java -cp part2 JavaMemory     # run with classpath pointing to part2

error: invalid flag: #
Usage: javac <options> <source files>
use --help for a list of possible options
Done
```

```
haerikyoung@MacBook-Air-3 Assignment2 % /opt/homebrew/bin/g++-15 -std=c++17 part2/cpp_memory.cpp -o cpp_memory
./cpp_memory

ok
```