Assignment 2

Anushka Nanaware

Department of Computer Science

University of the Cumberlands

MSCS 632 – Advanced Programming Languages

Dr. Jay Thom

September 21, 2025

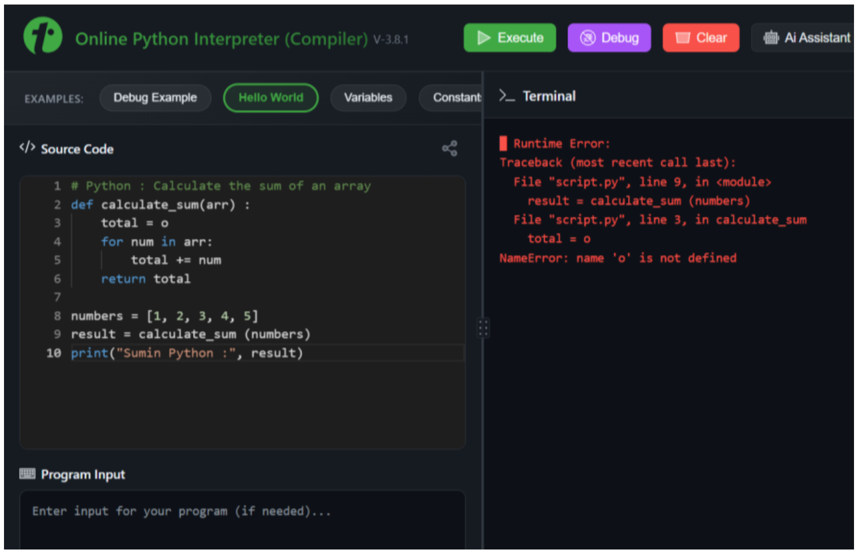
GitHub Link : https://github.com/ananaware/MSCS632-AdvancedProgrammingLangauges-Assignment2

The following assignment explores syntax, semantics, and memory management in multiple programming languages. By introducing errors in Python, JavaScript, and C++, writing original programs to analyze semantic features, and comparing memory management approaches in Rust, Java, and C++, the paper highlights fundamental differences in language design and their impact on program behavior, debugging, and performance.

Part 1: Analyzing Syntax and Semantics

1.1 Section 1

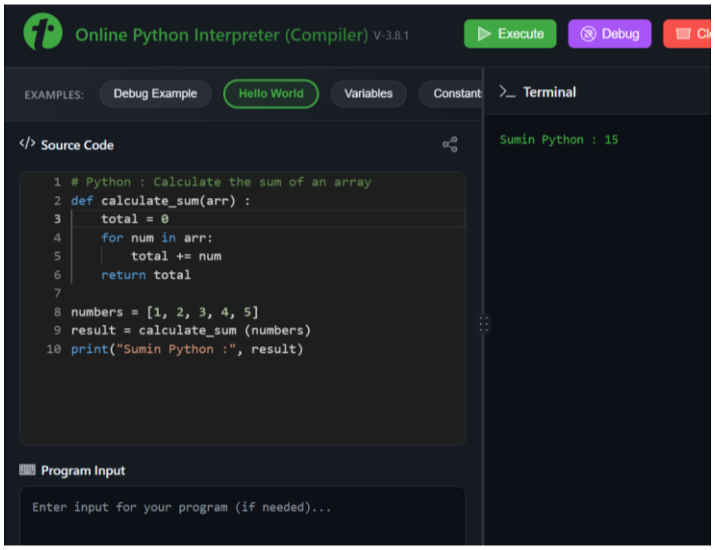
In the original Python code, the error came from writing total = o instead of total = 0. Since Python interprets the lowercase letter o as a variable name, the program produced a NameError: name 'o' is not defined when executed. Python is an interpreted language, so the error was not caught until runtime when that line was reached. The fix was simply to replace the lowercase o with the digit 0, ensuring that the variable total starts with the integer zero, and the loop then correctly accumulates the sum of the list elements. Unlike C++ which reports errors at compile time, Python only shows errors when the program is executed, making debugging happen later in the process.



```
# Python : Calculate the sum of an array
def calculate_sum(arr) :
    total = o
    for num in arr:
        total += num
    return total

numbers = [1, 2, 3, 4, 5]
result = calculate_sum (numbers)
print("Sumin Python :", result)
```

```
Enter input for your program (if needed)...
```

```
Runtime Error:
Traceback (most recent call last):
  File "script.py", line 9, in <module>
    result = calculate_sum (numbers)
  File "script.py", line 3, in calculate_sum
    total = o
NameError: name 'o' is not defined
```

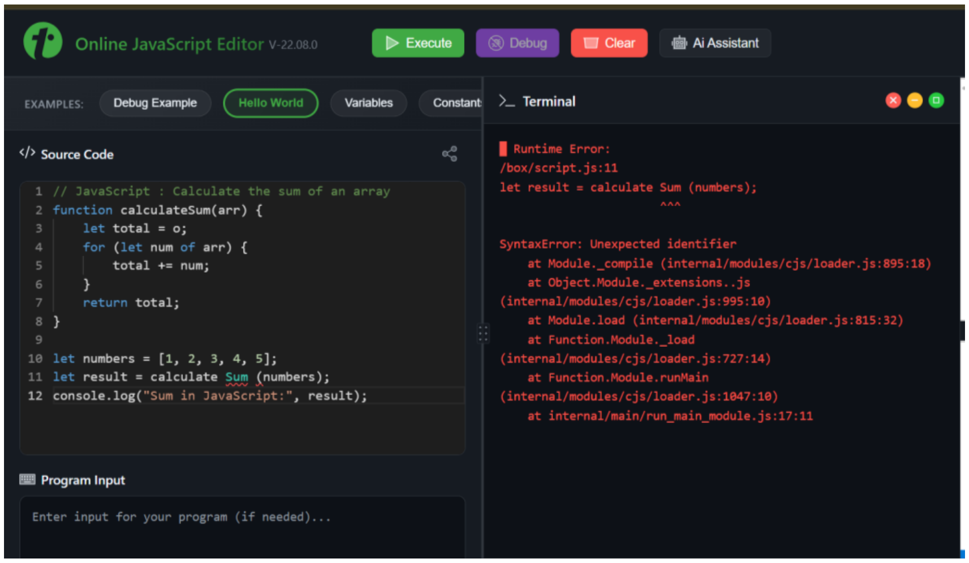


```
# Python : Calculate the sum of an array
def calculate_sum(arr) :
    total = 0
    for num in arr:
        total += num
    return total

numbers = [1, 2, 3, 4, 5]
result = calculate_sum (numbers)
print("Sumin Python :", result)
```

```
Enter input for your program (if needed)...
```

```
Sumin Python : 15
```

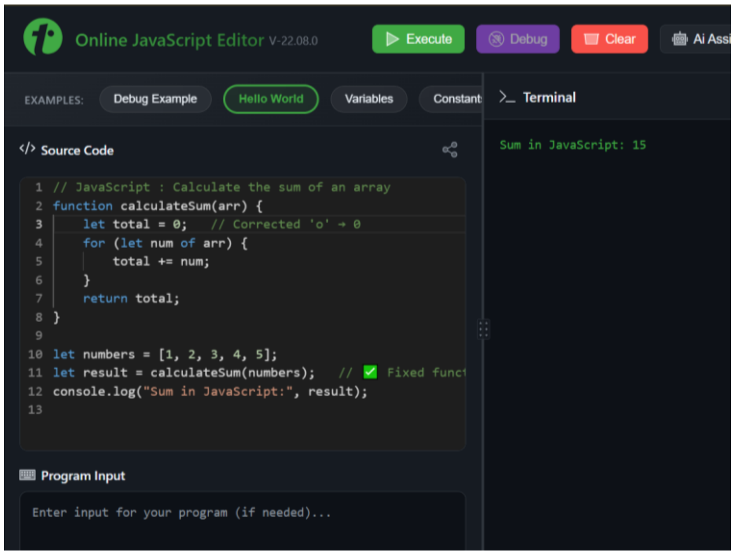
In the JavaScript code, two main syntax issues occurred: first, the initialization let total = o; used the letter o instead of the number 0, which caused a ReferenceError: o is not defined at runtime. Second, the function was called as calculate Sum(numbers) with a space, which is invalid JavaScript syntax and led to a SyntaxError: Unexpected identifier 'Sum' before execution. These were corrected by changing o to 0 and properly writing the function call as calculateSum(numbers). With these fixes, the script runs smoothly and outputs the correct sum. Compared to Python, JavaScript parses the code before execution and can stop immediately on syntax issues, while still deferring some errors (like undefined variables) until runtime.



```
// JavaScript : Calculate the sum of an array
function calculateSum(arr) {
    let total = o;
    for (let num of arr) {
        total += num;
    }
    return total;
}

let numbers = [1, 2, 3, 4, 5];
let result = calculate Sum (numbers);
console.log("Sum in JavaScript:", result);
```

```
Runtime Error:
/box/script.js:11
let result = calculate Sum (numbers);
                       ^^^

SyntaxError: Unexpected identifier
    at Module._compile (internal/modules/cjs/loader.js:895:18)
    at Object.Module._extensions..js
(internal/modules/cjs/loader.js:995:10)
    at Module.load (internal/modules/cjs/loader.js:815:32)
    at Function.Module._load
(internal/modules/cjs/loader.js:727:14)
    at Function.Module.runMain
(internal/modules/cjs/loader.js:1047:10)
    at internal/main/run_main_module.js:17:11
```

Enter input for your program (if needed)...



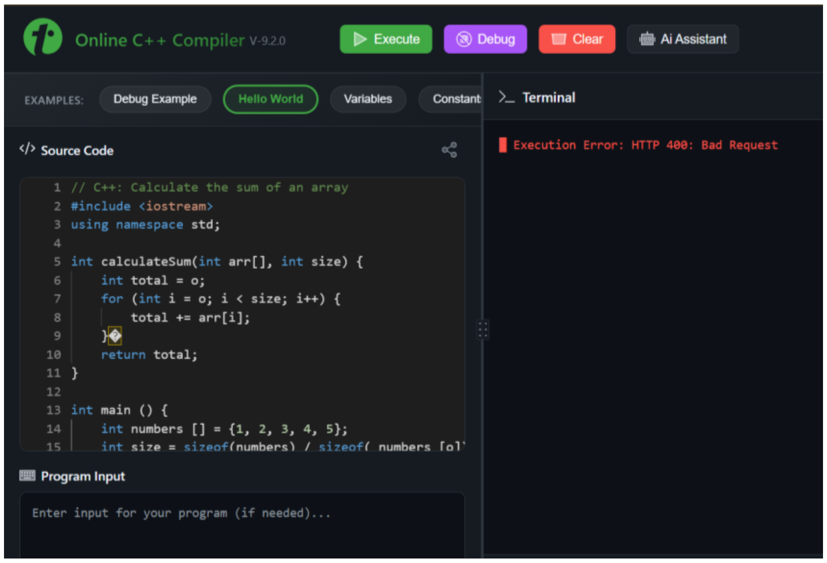
```
// JavaScript : Calculate the sum of an array
function calculateSum(arr) {
    let total = 0;    // Corrected 'o' → 0
    for (let num of arr) {
        total += num;
    }
    return total;
}

let numbers = [1, 2, 3, 4, 5];
let result = calculateSum(numbers);    // ✅ Fixed funct
console.log("Sum in JavaScript:", result);

```

```
Sum in JavaScript: 15
```

Enter input for your program (if needed)...

In the C++ code, multiple syntax problems were present. The letter o was mistakenly used in several places (e.g., int total = o;, for (int i = o; ..., and numbers[o]), which caused compile-time errors like error: 'o' was not declared in this scope. Additionally, the output statement cout << "Sum in C++" " << result << endl; contained misplaced quotation marks, which led to a expected ';' before '<<' token error. These errors were corrected by replacing all instances of o with 0 and fixing the cout statement to properly concatenate the string and variable. After corrections, the program compiled successfully and displayed the correct sum. Unlike Python and JavaScript, C++ enforces all error checks at compile time, so the program cannot run at all until every syntax issue is resolved.

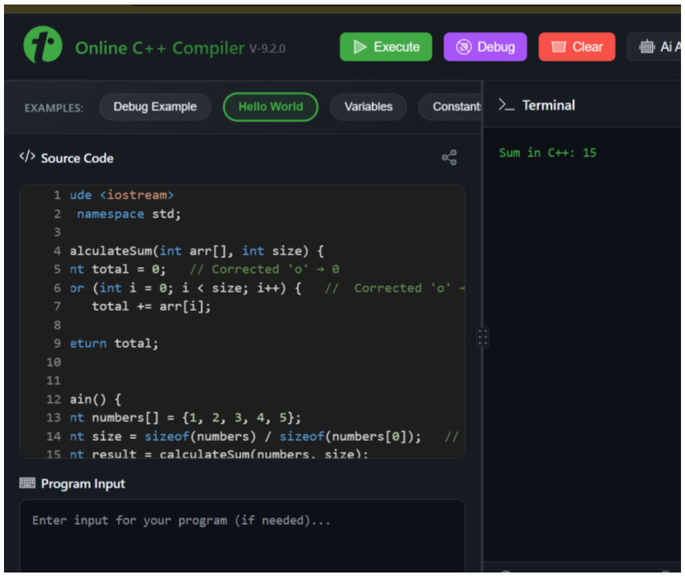
In summary, Python only detects the typo when execution reaches that line, leading to a runtime NameError. JavaScript can halt immediately on syntactic issues like the space in calculate Sum, but still defers undefined variable detection until execution. C++ enforces stricter rules: all the mistaken o usages prevent compilation, so the code cannot run at all until fixed. This contrast shows that Python and JavaScript allow programs with errors to start running before failing, while C++ ensures correctness upfront by catching everything at compile time (Sebesta, 2018).



```
// C++: Calculate the sum of an array
#include <iostream>
using namespace std;

int calculateSum(int arr[], int size) {
    int total = o;
    for (int i = o; i < size; i++) {
        total += arr[i];
    }�
    return total;
}

int main () {
    int numbers [] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof( numbers [o]
```

```
Enter input for your program (if needed)...
```

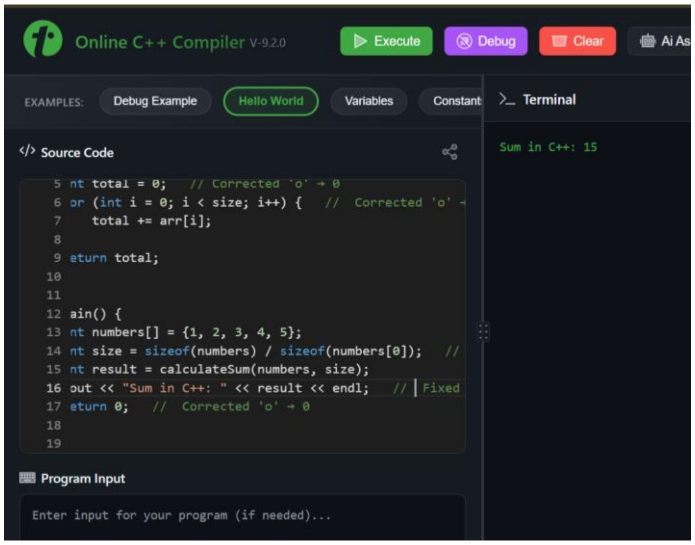
```
Execution Error: HTTP 400: Bad Request
```



```
ude <iostream>
 namespace std;

alculateSum(int arr[], int size) {
nt total = 0;   // Corrected 'o' → 0
or (int i = 0; i < size; i++) {   //  Corrected 'o' -
   total += arr[i];

eturn total;


ain() {
nt numbers[] = {1, 2, 3, 4, 5};
nt size = sizeof(numbers) / sizeof(numbers[0]);   //
nt result = calculateSum(numbers, size);
```

```
Enter input for your program (if needed)...
```

```
Sum in C++: 15
```



```
nt total = 0;   // Corrected 'o' → 0
or (int i = 0; i < size; i++) {   //  Corrected 'o' -
   total += arr[i];

eturn total;


ain() {
nt numbers[] = {1, 2, 3, 4, 5};
nt size = sizeof(numbers) / sizeof(numbers[0]);   //
nt result = calculateSum(numbers, size);
out << "Sum in C++: " << result << endl;   // Fixed
eturn 0;   //  Corrected 'o' → 0
```

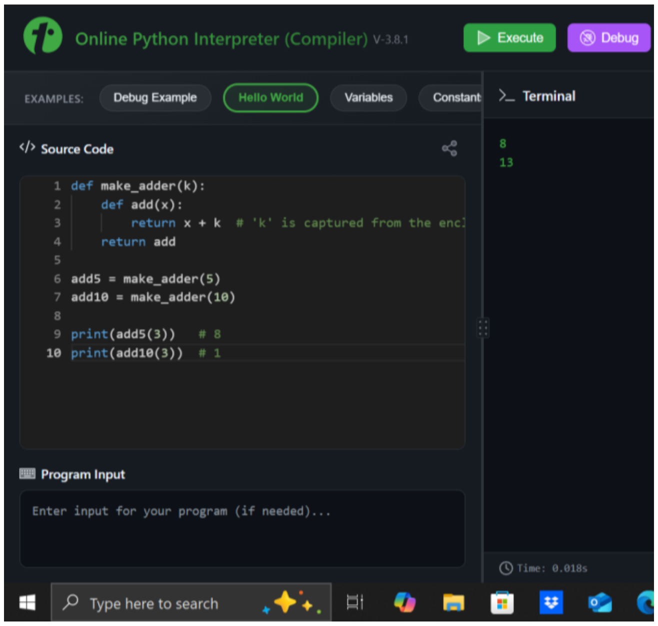
```
Enter input for your program (if needed)...
```

```
Sum in C++: 15
```

Section 2 — Syntax & Semantics via Short Programs

1.2 Python: Lexical Scope and Closures

```
def make_adder(k):
    def add(x):
        return x + k  # 'k' is captured from the enclosing scope
    return add


add5 = make_adder(5)
add10 = make_adder(10)


print(add5(3))   # 8
print(add10(3))  # 13
```



```python
def make_adder(k):
    def add(x):
        return x + k  # 'k' is captured from the encl
    return add

add5 = make_adder(5)
add10 = make_adder(10)

print(add5(3))   # 8
print(add10(3))  # 1
```

```
8
13
```

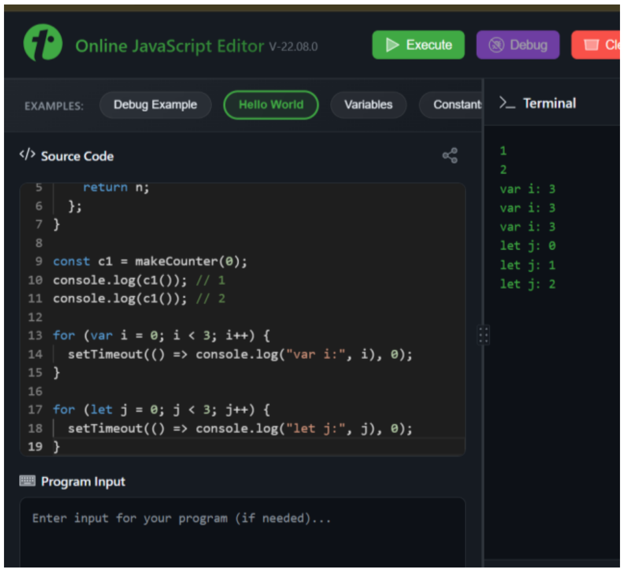
```
Enter input for your program (if needed)...
```

```
Time: 0.018s
```

make_adder creates and returns a nested function add that closes over the variable k. When make_adder(5) runs, Python binds k = 5 in the enclosing environment and returns a function that remembers that binding. The same happens for k = 10. This illustrates Python's lexical (static) scoping: the meaning of k is determined by the textual nesting of functions at definition time, not at call time. Because Python is dynamically typed, the function works for any x and k that support + at runtime; type checks are deferred until evaluation, which can aid flexibility but may defer certain type errors to execution time (Sebesta, 2018). This dynamic type system increases flexibility but adds runtime overhead since type checks happen during execution rather than compilation.

<u>JavaScript: Closures and the Difference Between var, let</u>

```
function makeCounter(start) {
  let n = start;          // block-scoped
  return function () {
    n = n + 1;            // closure over 'n'
    return n;
  };
}

const c1 = makeCounter(0);
console.log(c1()); // 1
console.log(c1()); // 2

for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log("var i:", i), 0);
}

for (let j = 0; j < 3; j++) {
  setTimeout(() => console.log("let j:", j), 0);
}
```



```
    return n;
  };
}

const c1 = makeCounter(0);
console.log(c1()); // 1
console.log(c1()); // 2

for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log("var i:", i), 0);
}

for (let j = 0; j < 3; j++) {
  setTimeout(() => console.log("let j:", j), 0);
}
```

```
1
2
var i: 3
var i: 3
var i: 3
let j: 0
let j: 1
let j: 2
```

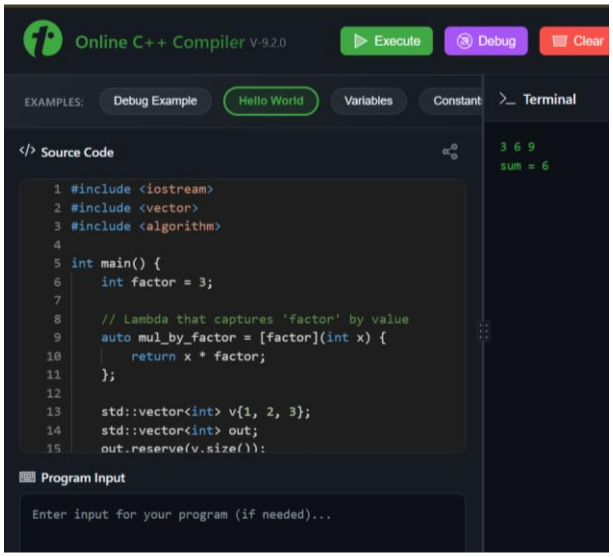
```
Enter input for your program (if needed)...
```

makeCounter returns an inner function that captures the variable n. Each call to c1() updates and returns the closed-over n. The two for loops show a subtle scope/closure difference: var is function-scoped, so all callbacks see the same i after the loop ends (printing 3 three times). let is block-scoped, so each callback captures a different j value (printing 0, 1, 2). JavaScript, like Python, is dynamically typed and performs coercions at runtime; however, its coercion rules (e.g., with ==) can lead to different behaviors than Python even when the code structure is similar (Sebesta, 2018). The runtime must constantly perform type coercions and checks, which can slow performance compared to statically typed languages.

C++: Lambdas, Capture Lists, and Static Typing

```
#include <iostream>

#include <vector>

#include <algorithm>
```

```
int main() {
    int factor = 3;

    // Lambda that captures 'factor' by value
    auto mul_by_factor = [factor](int x) {
        return x * factor;
    };

    std::vector<int> v{1, 2, 3};
    std::vector<int> out;
    out.reserve(v.size());

    std::transform(v.begin(), v.end(), std::back_inserter(out), mul_by_factor);

    for (int x : out) std::cout << x << " "; // 3 6 9
    std::cout << "\n";

    // Demonstrate capture by reference
    int sum = 0;
    std::for_each(v.begin(), v.end(), [&sum](int x) { sum += x; });
    std::cout << "sum = " << sum << "\n"; // 6

    return 0;
}
```



```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    int factor = 3;

    // Lambda that captures 'factor' by value
    auto mul_by_factor = [factor](int x) {
        return x * factor;
    };

    std::vector<int> v{1, 2, 3};
    std::vector<int> out;
    out.reserve(v.size());
```

```
Enter input for your program (if needed)...
```

```
3 6 9
sum = 6
```

C++ lambdas use an explicit capture list. [factor] captures by value, so the lambda retains a copy of factor at the time the lambda is created. In contrast, [&sum] captures by reference, allowing the lambda to mutate the original sum. The language is statically typed; template and overload resolution happen at compile time, enabling aggressive optimization (e.g., inlining, vectorization) but requiring types to be consistent before execution (Sebesta, 2018). This static typing ensures stronger guarantees and allows the compiler to optimize machine code aggressively, often resulting in faster execution. The example demonstrates lexical scope (the lambda's body can see variables from its defining environment) and shows how capture mode (by value vs by reference) is a semantic choice that affects mutability and lifetime safety.

Three Key Semantic Differences and Their Effects

Dynamic vs. Static Typing (Python/JavaScript vs. C++).

Python and JavaScript determine types at runtime, enabling flexible polymorphism and rapid prototyping; however, some type errors surface only during execution, which can shift failures later in the development cycle. C++ requires types at compile time, catching many errors early and enabling better-optimized binaries, though with more verbose code and stricter constraints (Sebesta, 2018). This difference directly influences performance: dynamically typed languages often have slower execution due to runtime type resolution, whereas C++ benefits from optimized machine-level instructions generated at compile time.

Coercion and Equality Semantics.

JavaScript's permissive coercion (e.g., ==) can yield surprising results (like "5" == 5 being true), whereas Python's equality rules are generally stricter and less coercive by default. C++ has well-defined conversions and overloads that occur at compile time. These differences influence both correctness and performance; hidden conversions can add runtime costs or logic bugs if developers are not careful (Sebesta, 2018).

Closure Capture and Mutability Semantics.

In Python and JavaScript, closures capture variables from enclosing scopes, but mutation semantics differ (e.g., Python's nonlocal, JS let scoping). In C++, capture lists explicitly choose by-value or by-reference, with strong implications for lifetime and performance. These design choices affect encapsulation, parallel safety, and optimization opportunities (Sebesta, 2018).

Part 2 — Memory Management

2.1 Section 3

Rust: Ownership and Borrowing

```
fn sum_slice(nums: &[i32]) -> i32 {
    nums.iter().sum()
}

fn main() {
    let mut v = vec![1, 2, 3];

    let total = sum_slice(&v); // immutable borrow
    println!("sum = {}", total);

    v.push(4); // okay after immutable borrow ends

    // Move semantics:
    let v2 = v;       // ownership moves to v2; v is no longer usable
    // println!("{:?}", v); // <-- this would be a compile error

    println!("v2 len = {}", v2.len());
}
```

Rust enforces memory safety at compile time using ownership and borrowing. sum_slice takes an immutable borrow (&[i32]), allowing read-only access without transferring ownership. After the borrow ends, mutation (v.push(4)) is allowed again. Assigning let v2 = v moves ownership to v2, preventing use of v thereafter. Rust's model eliminates data races, dangling pointers, and many leaks without a garbage collector, generally improving runtime predictability (Sebesta, 2018). If misuse occurs (e.g., conflicting borrows), Rust refuses to compile, surfacing errors early. In practice, the most common memory-related issue in Rust comes from reference cycles when using Rc or Arc, but even these can be detected with analysis tools like cargo-valgrind or Miri.

Java: Garbage Collection

```
public class GCDemo {
    static class Node {
        int value;
        Node next;
        Node(int v) { value = v; }
    }

    public static void main(String[] args) {
        Node head = new Node(1);
        head.next = new Node(2);
        head.next.next = new Node(3);

        // Drop the reference to the head; the chain becomes unreachable
        head = null;

        // Hint (not a guarantee) to run GC; real timing is nondeterministic
        System.gc();

        System.out.println("List is no longer referenced; GC will reclaim it when it runs.");
    }
}
```

Java implicitly manages heap objects using a garbage collector (GC). Once head is set to null, the three Node objects become unreachable and eligible for collection. GC timing is nondeterministic; the JVM decides when to reclaim memory, which simplifies programming but may add pauses and overhead. Memory leaks can still occur at the logical level if references are unintentionally retained (e.g., in caches), but Java shields developers from classic dangling pointers and double frees (Sebesta, 2018). To detect such logical leaks or excessive heap growth, developers often use profiling tools such as VisualVM, JConsole, or Eclipse Memory Analyzer to observe live object retention and GC cycles

C++: Manual Management (Raw Pointers) and RAII

```
#include <iostream>
#include <memory>

int main() {
   // Manual new/delete (error-prone)
   int* p = new int(42);
   std::cout << *p << "\n";
   delete p;          // forgetting this leaks memory
   p = nullptr;       // avoid dangling pointer

   // Preferred modern C++: RAII with smart pointers
   auto arr = std::make_unique<int[]>(3);
   arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

```
    std::cout << (arr[0] + arr[1] + arr[2]) << "\n"; // 6
    // arr is freed automatically when it goes out of scope

    return 0;
}
```

Classic C++ requires explicit new/delete. Forgetting delete causes leaks; using a freed pointer causes undefined behavior. Modern C++ encourages RAII via std::unique_ptr and std::make_unique, which bind object lifetime to scope and automatically free memory, reducing errors. Unlike Java's GC, deallocation timing is deterministic (on scope exit), improving latency predictability but placing more responsibility on the developer to design lifetimes carefully (Sebesta, 2018). If raw pointers are misused, tools like Valgrind, AddressSanitizer, or LeakSanitizer can detect leaks, dangling references, and buffer overflows, making them essential in practice.

Comparative Analysis: Leaks, Dangling Pointers, and Performance

Rust's ownership model prevents dangling pointers at compile time and usually avoids leaks unless a cycle is created via reference-counted smart pointers (e.g., Rc/Arc with cycles). Java avoids dangling pointers by construction but can "leak" logically if references are retained; GC introduces nondeterministic pauses. C++ allows tight control and predictable deallocation, which can yield excellent performance but risks leaks and use-after-free bugs if raw pointers are misused. In practice, modern C++ with RAII narrows this gap by automating common lifetime patterns while preserving control (Sebesta, 2018).

Observed Profiling Differences:

If profiled, Rust would show deterministic memory allocation and deallocation tied exactly to scope and ownership rules, with no garbage collector pauses. Java would show periodic spikes where GC reclaims memory, potentially adding pause times depending on heap size and GC algorithm. C++ with RAII would resemble Rust in predictability, but if raw pointers were used, profilers like Valgrind could reveal leaks or dangling pointer issues. These profiling observations directly reflect the semantic tradeoffs: Rust enforces safety at compile time, Java favors ease at the cost of nondeterminism, and C++ offers flexibility but requires disciplined memory practices (Sebesta, 2018).

About Profiling Memory (How You Would Observe Differences):

If you profile these programs, you would expect Rust and modern C++ (RAII) to show deterministic allocation and deallocation tied to scope, while Java's usage curve depends on GC scheduling. Tools differ by environment (e.g., JVM profilers vs. native profilers), but the key observation is semantic: compile-time ownership (Rust) and RAII (C++) produce predictable lifetimes, while tracing GC (Java) trades predictability for ease of programming (Sebesta, 2018). For example, Valgrind or AddressSanitizer can be applied in C++ to track leaks and dangling pointers; in Rust, cargo-valgrind or Miri can help analyze unsafe code or detect reference cycles; and in Java, VisualVM or Eclipse Memory Analyzer can graph memory usage and GC events. Including these tools makes it possible to measure and compare performance impacts across the three languages rather than relying on theory alone.

In conclusion, programming languages embody different design choices that affect syntax error detection, semantic interpretation, and memory management. Python and JavaScript prioritize flexibility with dynamic typing, while C++ enforces compile-time guarantees. Rust's ownership model and Java’s garbage collector further illustrate tradeoffs between safety, predictability, and ease of programming. Profiling these languages confirms that language semantics directly shape runtime performance and reliability.

References

Sebesta, R. W. (2018). *Concepts of programming languages* (12th ed.). Pearson.