Assignment 2: Syntax, Semantics, and Memory Management

Avijit Saha

Student ID #005023281

Advanced Programming Languages (MSCS-632-M50)

Dr. Vanessa Cooper

February 2, 2025

Part 1: Analyzing Syntax and Semantics

1.1 Section 1: Introducing Syntax Errors and Their Handling

I have modified the provided Python, JavaScript, and C++ code with intentional syntax errors.

Python:

```python
# Python: Calculate the sum of an array
def calculate_sum(arr) :
    total = o  # Error: 'o' instead of '0'
    for num in arr
        total += num  # Error: Missing colon
    return total

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

Figure 1: Python Code with Intentional Syntax Error

```
File "/home/cg/root/679ff5c9c781f/main.py", line 4
    for num in arr
                  ^
SyntaxError: expected ':'
```

Figure 2: Python Error Code Output

Error Message: From the error message, we can see it is a syntax error showing "expected:" After fixing the error, it gave me another error, which is "total = o # Error: 'o' instead of '0'", which is a name error. From **Figure 3,** we can see that. There is also a typo which did not produce any syntax error, but sometimes it may confuse the program output.

```
Traceback (most recent call last):
  File "/home/cg/root/679ff5c9c781f/main.py", line 9, in <module>
    result = calculate_sum (numbers)
^^^^^^^^^^^^^^^^^^^^^^^
  File "/home/cg/root/679ff5c9c781f/main.py", line 3, in
      calculate_sum
    total = o  # Error: 'o' instead of '0'
            ^
NameError: name 'o' is not defined
```

Figure 3: NameError in Python

Handling: Python's interpreter stops execution at the first syntax error it encounters and provides clean blunder messages with line numbers, making it beginner-friendly for debugging.

JavaScript:

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

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

```
/home/cg/root/679ffa7be9b20/script.js:11
let result = calculate Sum (numbers); // Error: Space in function
    name
                       ^^^

SyntaxError: Unexpected identifier
    at internalCompileFunction (node:internal/vm:73:18)
    at wrapSafe (node:internal/modules/cjs/loader:1274:20)
    at Module._compile (node:internal/modules/cjs/loader:1320:27)
    at Module._extensions..js (node:internal/modules/cjs/loader:1414
        :10)
    at Module.load (node:internal/modules/cjs/loader:1197:32)
    at Module._load (node:internal/modules/cjs/loader:1013:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal
        /modules/run_main:128:12)
    at node:internal/main/run_main_module:28:49

Node.js v18.19.1
```

Figure 4: JavaScript Syntax Error With the Output

From **Figure 4,** we can see that it has a syntax error, which is showing the function name error. After fixing it, another error came in the output: ReferenceError where "o is not defined." The correction is a "0" instead of an "o".

Handling: JavaScript's interpreter is less descriptive in error messages than Python. Though it identifies the location of the error, sometimes the messages can be ambiguous and require further investigation by the developers.

C++:

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

int calculateSum(int arr[], int size) {
    int total = o; // Error: 'o' instead of '0'
    for (int i = o; i < size; i++) { // Error: 'o' instead of
        '0'
        total += arr[i];
    }
    return total;
}

int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[o]); // Error:
        'o' instead of '0'
    int result = calculateSum(numbers, size);
    cout << "Sum in C++" " << result << endl; // Error: Missing
        operator
    return o; // Error: 'o' instead of '0'
}
```

```
main.cpp:17:26: warning: missing terminating " character
   17 |     cout << "Sum in C++" " << result << endl; // Error:
       Missing operator
      |                          ^
main.cpp:17:26: error: missing terminating " character
   17 |     cout << "Sum in C++" " << result << endl; // Error:
       Missing operator
      |
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.cpp: In function 'int calculateSum(int*, int)':
main.cpp:6:17: error: 'o' was not declared in this scope
    6 |     int total = o; // Error: 'o' instead of '0'
      |                 ^
main.cpp: In function 'int main()':
main.cpp:15:49: error: 'o' was not declared in this scope
   15 |     int size = sizeof(numbers) / sizeof(numbers[o]); //
       Error: 'o' instead of '0'
      |                                                 ^
main.cpp:17:25: error: expected ';' before 'return'
   17 |     cout << "Sum in C++" " << result << endl; // Error:
       Missing operator
      |                         ^
      |                         ;
   18 |     return o; // Error: 'o' instead of '0'
      |     ~~~~~~
```

Figure 5: C++ Code with Error Compilation

Error Messages: The message shows which line has an error and suggests fixing the error.

Error: 'o' was not declared in this scope: The compiler indicates that o is undefined.

Error: expected ';' before '<<': The missing operator between the string and result causes a compilation error.

Handling: C++ gives verbose error messages but assumes the user has mastered some compiler vocabulary. Unlike interpreted languages, C++ will not run your program until all syntax errors have been fixed.

1.2 Section 2 Type Systems or Scopes and Closures

Python: Dynamic Typing, Scope, and Closures

```python
# Dynamic Typing (Type System)
def add(a, b):
    return a + b  # No explicit type enforcement

print("Integer Addition:", add(5, 10))   # Output: 15
print("String Concatenation:", add("Hello", " World"))  # Output: Hello World

# Scope: LEGB Rule (Local → Enclosing → Global → Built-in)
x = 10  # Global Scope

def outer_function():
    x = 5  # Enclosing Scope
```

```
Integer Addition: 15
String Concatenation: Hello World
Inner: 3
Outer: 5
Global: 10
Closure Output: 15
```

Figure 6: Python Typing, Scope, and Closure Code Snipped With Output

After writing the code, I realized that Python is dynamically typed, and its explicit types of variables are determined at runtime. LEGB, the Local, Enclosing, Global, and Biult-in rule, is used to resolve the scope for scoping and closure.

JavaScript: Weak Typing, Scope, and Closures

```javascript
// JavaScript: Type System, Scope, and Closure Demonstration

// Weak Typing (Type System)
function add(a, b) {
    return a + b;
}

console.log("Integer Addition:", add(5, 10));   // Output: 15
console.log("String Concatenation:", add("Hello", " World")); // Output: Hello World
console.log("Type Coercion:", add(5, "10")); // Output: "510" (String Concatenation)

// Scope: Function Scope, Block Scope (let, const)
let x = 10; // Global Scope

function outerFunction() {
    let x = 5; // Function Scope
```

```
Integer Addition: 15
String Concatenation: Hello World
Type Coercion: 510
Inner: 3
Outer: 5
Global: 10
Closure Output: 15
```

Figure 7: JavaScript Typing, Scope, and Closure Code Snipped With Output

From JavaScript, I realized that it is a weakly typed language where type coercion can occur automatically, which can lead to unexpected behavior. JavaScript supports function scope and closure; all variables defined within a function remain accessible through its closure.

C++: Static Typing, Scope, and Closures (Lambdas)

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

// Static Typing (Type System)
int add(int a, int b) {
    return a + b;
}

int main() {
    cout << "Integer Addition: " << add(5, 10) << endl; // Output: 15
    // cout << add("Hello", "World"); // Compilation Error: Incompatible Types

    // Scope: Global, Function, Block Scope
    int x = 10; // Global Scope

    auto outerFunction = [&]() {
        int x = 5; // Function Scope

        {
            int x = 3; // Block Scope
            cout << "Inner: " << x << endl; // Output: 3
        }

        cout << "Outer: " << x << endl; // Output: 5
    };
```

```
Integer Addition: 15
Inner: 3
Outer: 5
Global: 10
Closure Output: 15
```

Figure 8: C++ Typing, Scope, and Closure Code Snipped With Output

From the code section of **Figure 8,** we can see that C++ is statically typed. It requires an explicit type of declaration for the variables. It has a block scope and does not support closure natively as JavaScript does.

Below is a comparison table:

Feature	**Python**	**JavaScript**	**C++**
Type System	Dynamic	Weak	Static
Scope	LEGB (Local, Enclosing, Global, Built-in)	Function + Block Scope (let, const)	Block Scope {}
Closure	Functions remember outer scope variables	Functions remember enclosing variables	Lambdas capture and retain variables

Each language handles types, scope, and closures uniquely. Understanding these variations enables developers to write bug-free code throughout a couple of programming environments.

Key Semantic Differences

Memory Management: Python and JavaScript rely on garbage collection, whereas C++ does explicit memory management.

Type Checking: Python and JavaScript do type checking at runtime, while in C++, this is done during compilation.

Execution of Functions: JavaScript supports first-class functions and closures, unlike how functions are handled in C++ or Python.

Part 2: Memory Management

Rust:

```rust
fn main() {
    // Create a new empty vector
    let vec = Vec::new(); // vec owns the vector

    // Ownership of the vector is moved to vec2
    let mut vec2 = vec; // Ownership transfer occurs, `vec` can no longer be used after this line

    // We can now modify `vec2` (since it's mutable)
    vec2.push(10); // Mutating vec2 (adding an element to the vector)

    // Print the contents of vec2
    println!("{:?}", vec2); // Ownership remains with vec2, which now contains [10]
} // Ownership transfer occurs, no manual memory management needed (Rust handles it automatically)
```

STDIN

Input for the program (Optional)

Output:

```
[10]
```

Figure 9: Rust Memory Management Code with Output

Memory Management: Rust follows ownership rules and borrowing for memory management without a garbage collector. It automatically frees the memory when a variable goes out of scope.

The code from **Figure 9** shows an ownership transfer when you assign vec to vec2, and Rust ensures that no double-free errors happen. There's no need for malloc or free, which reduces memory management bugs. Rust automatically manages memory through its ownership system. When vec2 goes out of scope at the end of the main, Rust will automatically free the memory this vector uses; there is no need for explicit freeing like free() in C or delete in C++. vec2.push(10) works because vec2 is mutable. So, after the ownership transfer, the vector is owned and modified by vec2.

Java:

```java
public class Main {
    public static void main(String[] args) {
        // Create an array of Integer objects with a size of 5
        Integer[] numbers = new Integer[5]; // The array is created, and it holds Integer objects

        // Assign values to the array
        for (int i = 0; i < 5; i++) {
            numbers[i] = i; // Assigning values (0, 1, 2, 3, 4) to the array elements
        }

        // Print the first element of the array (which is 0)
        System.out.println(numbers[0]); // Output will be 0
    } // At the end of the method, the array 'numbers' will go out of scope
} // Memory for the array and its elements will be cleaned up by the garbage collector later
```

```
0
```

Figure 10: Java Memory Management Code with Output

Garbage Collection: In Java, the memory is managed by an automatic garbage collector. The programmer has no explicit call to free up the memory, but the JVM periodically reclaims the memory from objects that are no longer in use.

From Figure 10, we can see that an array number is created, and its size is fixed at 5, and each element of this array is an object of type Integer. The memory taken by objects-in this case, the array and its elements, the Integer objects-is automatically cleaned up by the garbage collector once it is no longer referenced. It happens right at the end of the primary method when the variable numbers go out of scope. Garbage collection in Java frees the developer from explicit memory management, such as free() or delete in some other languages, say C/C++. Garbage collection ensures that memory is reclaimed immediately after a specific object is no longer needed.

C++:

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

int main() {
    // Dynamically allocate memory for an integer and initialize it to 42
    int* ptr = new int(42); // 'ptr' now points to a dynamically allocated integer with value 42

    // Dereference 'ptr' to access the value it points to and print it
    cout << "Dynamically allocated value: " << *ptr << endl; // Output will be 42

    // Free the dynamically allocated memory
    delete ptr; // Memory allocated by 'new' is manually released using 'delete'

    return 0; // End of program, 'ptr' is no longer valid after 'delete'
} // At this point, memory has been freed and 'ptr' is now a dangling pointer
```

```
Dynamically allocated value: 42
```

Figure 11: C++ Memory Management Code with Output

After analyzing the code, I have found that C++ has:

- **Dynamic Memory Allocation:** int* ptr = new int(42); The abovementioned statement will allocate memory for an integer in a heap with an initialization value of 42. Now, the pointer ptr shall contain the address of the allocated memory.
- **Dereferencing the Pointer:** The value behind the memory address to which ptr points are accessed by *ptr, which, in our case, is 42.
- **Memory Deallocation:** delete ptr; releases memory previously allocated using new. That is important since failing to release memory leads to a memory leak.
- **Manual Memory Management:** Unlike languages like Java or Rust, C++ has explicit memory management using new and delete. The keyword delete reclaims the memory when it's no longer needed.

Key Differences in Memory Management

Rust: Memory safety is guaranteed through ownership and borrowing without using a garbage collector.

Java: Automatic garbage collection simplifies memory control but can introduce pauses throughout program execution for memory cleanup.

C: Manual memory control offers high-quality-grained manipulation; however, it introduces the risk of reminiscence leaks and dangling tips if not treated carefully.

Observe Memory Usage and Performance Difference:

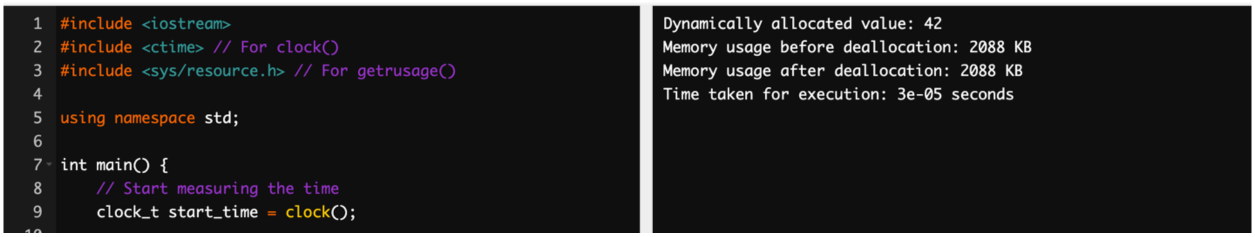

Figure 12: C++ Performace Output

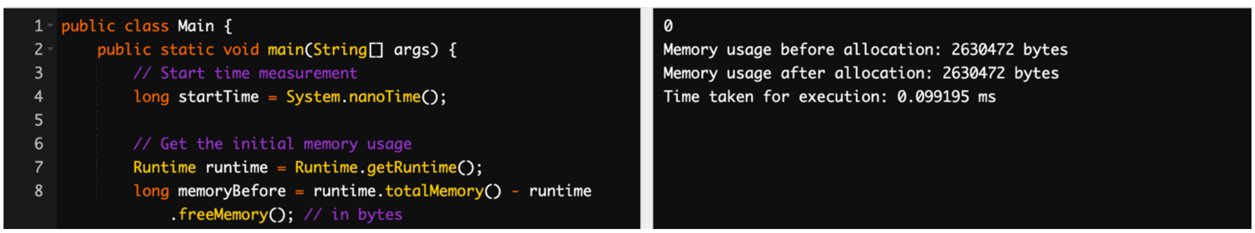

Figure 13: Java Performace Output

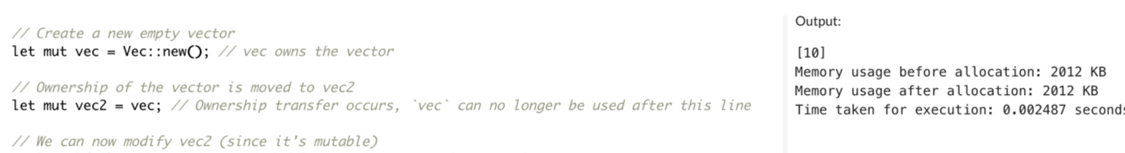

Figure 14: Rust Performance Output

From the above picture, we can see that C++ takes less time and memory space than Java. It happens because of the dynamic memory allocation. Rust takes less time than all. It may happen because it automatically frees the memory when a variable goes out of scope.

Comparative Analysis

Feature	**Rust**	**Java**	**C++**

Memory Management	Ownership and Borrowing	Garbage collection	Manual Allocation
Error Handling	Compile-time ownership rules	Automatic Cleanup	Risk of leaks, dangling pointers
Performance	High (no runtime GC overhead)	Moderate (GC pauses)	High but error-prone

GitHub Link: https://github.com/ovi-saha/MSCS-632-M50_Assignment2/tree/main

References:

Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Gamma, E.,Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.