1. (i) Evaluated left-to-right

For Sum1:

‘(i / 2)’ is evaluated first ‘20 / 2 = 10’

Then ‘mystery(&i)’ is called:

‘&i’ is pointer to ‘i’, so ‘mystery’ modifices the value of ‘i’, making it ‘i=25’.

‘3 \* (\*k) – 1’ where ‘\*k = 25’, so ‘3 \* 25 – 1 = 74’.

Finally ’10 + 74 = 84’

For Sum2:

‘mystery(&j)’ is called first:

‘&j is a pointer to ‘j’, so ‘mytery’ modifies the values of ‘j’, making it ‘j = 25’

‘3 \* (\*k) – 1’ where ‘\*k = 25’, so ‘3 \* 25 - 1 = 74’.

Then ‘(j / 2)’ is evaluated ’25 / 2 = 12’ (integer division).

Finally, ’74 + 12 = 86’.

(ii) Evaluated right-to-left

For Sum1:

‘mystery(&i)’ is called first:

‘&i’ is pointer to ‘i’, so ‘mystery’ modifies the value of ‘i’, making it ‘i = 25’.

‘3 \* (\*k) – 1’ where ‘\*k = 25’, so ‘3 \* 25 – 1 = 74’.

Then (i / 2) is evaluated ’25 / 2 = 12’ (integer division)

Finally ’12 + 74 = 86’

For Sum2:

‘(j / 2)’ is evaluated first ’20 / 2 = 10’.

Then ‘mystery(&j)’ is called:

‘&j’ is a pointer to ‘j’, so ‘mystery’ modifies the values of ‘j’, making it ‘j = 25’.

‘3 \* ( \*k) – 1’ where ‘\*k = 25’, so ‘3 \* 25 – 1 = 74’.

Finally ’74 + 10 = 84’

Results:

Left To Right

Sum1 = 84

Sum2 = 86

Right to Left

Sum1 = 86

Sum2 = 84

1. When this code(AsitPatel\_mystery.cpp) is run it will print the values of ‘sum1’ and ‘sum2’

For ‘Sum1’:

* The expression ‘(i / 2)’ evaluates to ‘10’.
* The ‘mystery(&i)’ call modifies ‘i’ to ‘25’ and return ‘74’
* So ‘sum1 = 10 + 74 = 84’

For ‘Sum2’:

* The ‘mystery(&j)’ call modifies ‘j’ to ‘25’ and return ‘74’.
* The expression ‘(j / 2)’ evaluates to ‘12’.
* So, ‘sum2 = 74 + 12 = 86’.

So, the values of ‘sum1’ and ‘sum2’ are ’84’ and ‘86’, respectively. These results are consistent with what we expected based on the analysis performed earlier. The function ‘mystery’ modifies the values of the variable passed to it by reference (‘i’ and ‘j’ in this case) and then returns a value based on the modified value.

1. In this code (AsitPatel\_mystery.java), the ‘mysery’ method accepts an integer ‘k’, adds 5 to it, then multiplies the result by 3 and subtracts 1 before returning the final value.

For ‘Sum1’

* The expression ‘(i / 2)’ evaluates to ‘10’
* Then, ‘mystery(i)’ is called:
  + Inside ‘mystery’, ‘i’ is passed by value, so the value of ‘i’ within the method remains unaffected.
  + ‘k’ is a local variable, and modifying it won’t affect the value of ‘i’ outside the method.
  + ‘k += 5’ increases the value of ‘k’ to ‘25’.
  + ‘3 \* k – 1’ where ‘k = 25’, so ‘3 \* 25 -1 = 74’
  + Finally ’10 + 74 = 84’

For ‘Sum2’:

* ‘mystery(j)’ is called first:
  + Similar to above, ‘j’ is passed by value, so the value of ‘j’ within the method remains unaffected.
  + ‘k += 5’ increases the value of ‘k’ to ‘25’.
  + ‘3 \* k – 1’ where ‘k = 25’ , so ‘ 3 \* 25 – 1 = 74’.
* Then, ‘(j / 2) ‘ is evaluated ’20 / 2 = 10’.
* Finally, ’74 + 10 = 84’.

Both ‘sum1’ and ‘sum2’ end up being ‘84’, which might seem counterintuitive given that ‘sum1’ involves modifying ‘i’ within the ‘mystery’ method. However, this behavior occurs because ‘i’ passed by values to the ‘mystery’ method, meaning any changes made to ‘k’ inside the method do not affect the value of ‘i’ outside the method. Therefore, ‘i’ remains ‘20’ throughout the execution of the ‘main’ method. The same applies to ‘j’, As a result, both ‘sum1’ and ‘sum2’ are calculated based on the original values of ‘i’ and ‘j’, resulting in the same output.

1. In this code(AsitPatel\_mystery.go),

For ‘Sum1’:

* The expression ‘(i / 2)’ evaluates to ‘10’
* Then ‘mystery(&i)’ is called:
  + The address of ‘i’ is passed to the ‘mymstery’ function.
  + Inside ‘mystery’, the value pointed to by ‘k’ (which is the address of ‘i’) is incremented by 5, making ‘’i = 25’.
  + The return value of ‘mystery(&i)’ is ‘3 \* 25 – 1 = 74’.
* Finally, ‘sum1 = 10 + 74 = 84’

For ‘sum2’:

* ‘mystery(&j)’ is called first:
  + Similar to above, the address of ‘j’ is passed to the ‘mystery’ function.
  + Inside ‘mystery’, the value pointed to by ‘k’ (which is the address of ‘j’) is incremented by 5, making ‘j = 25’.
  + The return value of ‘mystery(&j)’ is ‘3 \* 25 -1 = 74’.
* Then, ‘(j / 2)’ is evaluated: ’25 / 2 = 12’
* Finally, ‘sum2 = 74 + 12 = 86’

# command-line-arguments

./Asit\_mystery.go:7:6: can inline mystery

./Asit\_mystery.go:20:27: inlining call to mystery

./Asit\_mystery.go:21:17: inlining call to mystery

./Asit\_mystery.go:23:12: inlining call to fmt.Printf

./Asit\_mystery.go:24:12: inlining call to fmt.Printf

./Asit\_mystery.go:7:14: k does not escape

./Asit\_mystery.go:23:12: ... argument does not escape

./Asit\_mystery.go:23:27: sum1 escapes to heap

./Asit\_mystery.go:24:12: ... argument does not escape

./Asit\_mystery.go:24:27: sum2 escapes to heap

The key point to understanding here is that when you pass the address of a variable to a function in Go (‘&i’ or ‘&j’), any modification made to the value at that address within the function is reflected outside the function as well. Therefore, both ‘i’ and ‘j’ are modified by the ‘mystery’ function, resulting in the same value of both ‘sum1’ and ‘sum2’.

1. In C, Java, and Go, the order of operator evaluation in assignment statements follows the same general rule: operators are evaluated from left to right. However, the behaviour of pointers and references, as well as the handling of function arguments, can lead to differences in how this order affects the overall behaviour of the program.

Let's compare these languages in terms of their order of operator evaluation and how it relates to the experimental results obtained in the provided questions:

C Programming Language:

In C, the order of operator evaluation in assignment statements is well-defined. Expressions are evaluated strictly from left to right. However, when function calls are involved, the order of evaluation can be impacted by the compiler's optimization settings. In the provided experiments, we observed the following:

In Question (b), the order of evaluation was straightforward, leading to the expected results.

In Question (c), the results varied based on the order of evaluation of function arguments, which can depend on the compiler and its optimization settings.

Design:

C is a low-level language with a minimal runtime and straightforward syntax. It provides direct access to memory through pointers, allowing for fine-grained control over memory management. However, this power comes with the risk of memory-related bugs such as segmentation faults and memory leaks.

Advantages:

* Direct memory manipulation allows for efficient code.
* Close-to-hardware access makes it suitable for system programming and embedded systems.
* Well-established and widely used in various domains.

Disadvantages:

* Prone to memory-related bugs and undefined behaviour.
* Lacks modern language features like garbage collection and object-oriented programming.

Java Programming Language:

In Java, the order of operator evaluation in assignment statements is also left-to-right, but with some differences due to its object-oriented nature and pass-by-value semantics. In the provided experiments, we saw that:

Java passes objects by reference-like semantics, but primitives are passed by value.

Modifications to objects passed as arguments are reflected outside the method.

The order of evaluation can impact the behaviour of expressions, especially when function calls are involved.

Design:

Java is designed to be platform-independent and highly object-oriented. It features automatic memory management through garbage collection, which reduces the risk of memory-related bugs. Pass-by-value semantics simplify memory management but can lead to confusion when dealing with mutable objects.

Advantages:

* Platform independence allows for a "write once, run anywhere" capability.
* Robust memory management with garbage collection reduces the risk of memory leaks.
* Object-oriented features promote code reuse and maintainability.

Disadvantages:

* Slower execution compared to low-level languages like C due to the JVM overhead.
* Lack of direct memory access limits its use for systems programming.

Go Programming Language:

In Go, the order of operator evaluation in assignment statements is similar to C and Java, but with its own set of rules and idioms. In the provided experiments, we observed:

Go passes all arguments by value, but it allows passing pointers for more efficient access to large data structures.

The language encourages simplicity and readability, which can lead to more predictable behaviour in expressions and assignments.

Design:

Go is designed for simplicity, efficiency, and readability. It features built-in concurrency support through goroutines and channels, making it suitable for scalable and concurrent applications. Go promotes a "less is more" philosophy, favouring simplicity over complexity.

Advantages:

* Built-in concurrency support simplifies concurrent programming.
* Static typing and strict syntax help catch errors at compile time.
* Efficient memory management through garbage collection.

Disadvantages:

* A lack of generics can lead to code duplication in some scenarios.
* Limited support for low-level memory manipulation compared to C.

Compiler Differences:

Each language has its compiler or interpreter, which may implement the language specifications differently. Compiler optimizations, target architecture, and runtime environment can all affect the behaviour of the program, including the order of operator evaluation. Therefore, the experimental results observed in each language can be influenced by compiler-specific optimizations and runtime behaviours.

In summary, while C, Java, and Go follow similar rules for the order of operator evaluation in assignment statements, differences in their language designs and compiler implementations can lead to variations in behaviour. Understanding these differences is crucial for writing reliable and predictable code in each language.