Project Requirements:

1. The MiniJava compiler shall not read from or write to any data related to garbage collection.
    a. "Data related to garbage collection" is defined as any data that resides within an instance of the HeapBlock structure.
    b. Only the MiniJava runtime shall directly modify this data.
2. The MiniJava garbage collector system shall not require the user to modify valid MiniJava programs in any way in order for the program to be supported.
    a. Valid, compilable MiniJava programs shall be interoperable as-is with the garbage collector system.
    b. Invalid MiniJava programs will not be supported by the garbage collector, as they cannot be compiled.
3. Mark-sweep garbage collection shall be implemented as part of the MiniJava runtime.
4. Mark-sweep garbage collection should be implemented through the following three C functions: "runtime_ms_mark", "runtime_ms_sweep", and "runtime_ms_recurse".
5. Reference counting garbage collection shall be implemented as part of the MiniJava runtime.
6. Reference counting garbage collection should be implemented through the following two C functions: "runtime_ref_inc", and "runtime_ref_dec".
7. The MiniJava compiler shall not call down to the mark-sweep garbage collection functions.
    a. The "mark-sweep garbage collection functions" shall be defined as the C functions in the MiniJava runtime which are prefixed by "runtime_ms_".
8. Only the mark-sweep garbage collection functions in the MiniJava runtime shall call the "runtime_ms_recurse" function.
9. The reference counting garbage collection functions in the MiniJava runtime shall not call down to the mark-sweep garbage collection functions.
    a. The "reference counting garbage collection functions" shall be defined as the C functions in the MiniJava runtime which are prefixed by "runtime_ref_".
10. The mark-sweep garbage collection functions in the MiniJava runtime shall not call down to the reference counting garbage collection functions.
11. Only the MiniJava compiler shall call the reference counting garbage collection functions in the MiniJava runtime.
    a. These "calls" shall be generated as assembly code in the compiled program.
    b. These calls shall be generated only during the codegen phase of compilation.
12. Only the memory allocation functions in the MiniJava runtime shall call the mark-sweep garbage collection functions.
    a. The function "runtime_ms_recurse" shall be the only exception to this rule, as it is a function for internal GC use only.
    b. These calls shall occur under the memory restraints specified in requirement 29.
13. All memory allocation of non-zero size performed through the MiniJava runtime shall allocate additional space to store the heap block header.

    a. The additional space shall be the size of the HeapBlock C structure declared in the runtime, rounded up to the nearest 4 bytes to maintain word alignment.
    b. Given a request to allocate a block of size zero, the MiniJava runtime shall not attempt to allocate any memory, and shall return a null pointer.
14. The "runtime_free" function in the MiniJava runtime shall free the memory associated with a heap block.
    a. The process of freeing the specified memory shall only involve passing the heap block to the "free" function in the C standard library
15. The MiniJava runtime shall initialize the "tag" field of all new HeapBlock instances with the value HEAP_BLOCK_TAG.
    a. Given a memory allocation failure (runtime_alloc_object returns NULL), the runtime shall not attempt to write to the pointer.
    b. HEAP_BLOCK_TAG shall be defined within the source file "runtime.h", and shall fit within the 32-bits of the "tag" field.
16. The MiniJava runtime shall initialize the "next " field of all new HeapBlock instances with the value NULL.
    a. Given a memory allocation failure (runtime_alloc_object returns NULL), the runtime shall not attempt to write to the pointer.
    b. The runtime shall not touch the "next" field again until the specified heap block will be linked to the newest created heap block.
17. The head pointer of the heap allocation list shall be NULL until the first HeapBlock is created.
    a. It may be NULL again in the future, in the event that all heap allocated objects are collected.
    b. It shall only be managed by the MiniJava runtime.
18. The tail pointer of the heap allocation list shall be NULL until the first HeapBlock is created.
    a. It may be NULL again in the future, in the event that all heap allocated objects are collected.
    b. It shall only be managed by the MiniJava runtime.
19. If the head pointer of the heap allocation list is non-NULL, the tail pointer of the heap allocation list shall also be non-NULL, and vice versa.
    a. In the event that all heap allocated objects are freed through the garbage collector, both the head and tail pointers should be NULL.
20. The MiniJava runtime shall initialize the "size" field of all new HeapBlock instances with the value corresponding to the size passed to runtime_alloc_object and/or runtime_alloc_array.
    a. Given a size of zero, the runtime shall not write to the "size" field and instead throw a runtime error, as this situation should never occur.
    b. A "runtime error" is defined logging any error message to the console, and terminating the execution of the program through the "exit" function.
21. The MiniJava runtime shall initialize the "marked" field of all new HeapBlock instances with the value FALSE (0).
    a. The runtime shall not write to the "marked" field again until the mark-sweep function "runtime_ms_mark" is called.

22. The MiniJava runtime shall initialize the “ref” field of all new HeapBlock instances with the value 0.
23. The “runtime_ref_inc” function in the MiniJava runtime shall increment a heap block’s reference count by exactly one.
    a. Given a NULL pointer to a heap block, the function shall perform no operations.
24. The “runtime_ref_dec” function in the MiniJava runtime shall decrement a heap block’s reference count by exactly one.
    a. Given a NULL pointer to a heap block, the function shall perform no operations.
    b. Given a heap block with a reference count of zero after the decrement operation, “runtime_ref_dec” shall call “runtime_free” to free memory allocated for said heap block.
25. When a heap block is used as an rvalue in an assignment statement, the MiniJava compiler shall insert exactly one function call to “runtime_ref_inc”, passing in the specified heap block.
26. When the value of a pointer/reference type variable in MiniJava code is overwritten via an assignment statement, the MiniJava compiler shall insert exactly one function call to “runtime_ref_inc” using the rvalue’s heap block, and exactly one function call to “runtime_ref_dec” using the lvalue’s heap block, in that order.
27. When compiling a statement in MiniJava code that involves both increment and decrement operations, the runtime function call(s) corresponding to the increment(s) shall be processed and generated first before the runtime function call(s) corresponding to the decrement(s).
28. When a pointer/reference type variable in MiniJava code goes out of scope, the compiler shall generate a function call to “runtime_ref_dec” if the specified variable is a valid heap block pointer.
    a. A potential heap block pointer shall not be considered valid if it resides on the program stack.
    b. A potential heap block pointer shall not be considered valid if it resides outside of the memory area in which the program has read/write access to.
    c. A potential heap block pointer shall not be considered valid if the 32-bit value it points to does not equal HEAP_BLOCK_TAG.
    d. A potential heap block pointer shall not be considered valid if the block’s next pointer is itself not a valid heap block pointer and is not NULL.
    e. If the variable does not contain a valid heap block pointer, the MiniJava compiler shall not emit any function calls to the MiniJava runtime, and the MiniJava runtime shall not perform any operation regarding the lifetime/scope of the specified local variable.
29. The MiniJava runtime shall only run the marking process when less than 20% of the system’s memory is available for allocation.
    a. “Running” the marking process shall be defined as calling the “runtime_ms_mark” function.
    b. The amount of available system memory should be queried using the “sysconf” function.
30. The mark-sweep garbage collector shall distinguish valid heap block pointers from other data types by first verifying that the immediate value is valid when interpreted as an

address, and then verifying that the 32-bit value at that address is equal to HEAP_BLOCK_TAG.
    a. “Valid heap block pointers” are defined in requirement 28.

31. The mark-sweep garbage collector shall recurse through all roots of the object graph when beginning the marking process.
    a. “Recurse” shall be defined as calling the “runtime_ms_recurse” function, specifying the corresponding heap block, and the value FALSE (to signify root).
    b. Recursion should be performed as a depth first search.
    c. The “roots” of the object graph shall be defined as areas of data storage which are separate from the heap and accessible via no levels of indirection, such as: local variables on the stack and registers in the CPU.
    d. The “object graph” shall be defined as the net of references between objects allocated throughout program execution.
    e. There shall always be a nonzero amount of roots at any given program state: at the very least, the registers must be inspected.
32. For all roots of the object graph, the mark-sweep garbage collector shall recurse through all valid heap block pointers that are discovered as children of the roots.
    a. “Recurse” shall be defined as calling the “runtime_ms_recurse” function, specifying the corresponding heap block, and the value TRUE (to signify non-root).
    b. “Valid heap block pointers” are defined in requirement 28.
    c. The search for valid heap block pointers shall not begin at offset 0 into the heap block. Instead, it shall begin at the end of the HeapBlock structure.
33. During the marking process, the mark-sweep garbage collector shall set the “marked” bit of every HeapBlock structure it discovers.
    a. The “marking” process shall be defined as the depth first search through the object graph.
    b. The mark bit in the heap block shall be set to TRUE (1).
34. If cyclic garbage exists in the heap, it shall not be accessible through the object graph.
    a. “Cyclic garbage” shall be defined as a cyclic reference between garbage objects.
    b. Because it shall not be accessible through the object graph, it shall not be marked during the marking process.
35. If a heap block is unmarked, its reference count shall not affect its eligibility to be freed by the mark-sweep garbage collector.
    a. “Unmarked” blocks shall be defined as blocks that were created before the last time the marking process (“runtime_ms_mark” function) ran, and have their “marked” field set to FALSE (0).
    b. If a heap block is not reachable from the object graph, and the specified heap block has a non zero reference count, it can be asserted that the reference count must be either from itself or other objects that are also garbage. Therefore, the significance of the reference can be ignored during the sweep process.
36. Heap blocks which are not marked, and are included in the MiniJava runtime’s linked list of heap blocks shall be freed during the sweep process of mark-sweep garbage collection.

   a. The “sweep” process shall be defined as the process of iterating through the MiniJava runtime’s linked list of heap blocks, and freeing memory associated with non-marked blocks.

37. The sweep process of mark-sweep garbage collection shall remove from the MiniJava runtime’s linked list of heap blocks only the heap blocks that had their corresponding memory freed.
38. Under no circumstances shall the MiniJava runtime’s linked list of heap blocks be broken after the sweep process of mark-sweep garbage collection completes.
    a. A “broken” list shall be defined as a linked list where one or more nodes has their “next” pointer set to NULL, when the node is not actually the tail element in the list.

Project Test Plan:

Overview:

The objective of this test plan is to ensure that the implementation aligns with the expected behavior stated by the requirements.

Scope:

The test plan focuses to validate that:

1. The garbage collector functions in the MiniJava runtime.
2. MiniJava's compiler's interaction with the garbage collector functions.
3. The HeapBlock data structure is initialized and managed.
4. Errors are handled and the system boundary conditions are met.

Environment:

- Compiler Version: MiniJava
- Runtime Environment: Linux/Windows

Tests:

1.1 General Rules for the MiniJava Compiler and Runtime

Purpose: Ensure the compiler and runtime adhere to operational rules.

Methods: Automated testing.

Steps:

1. Run a series of valid and invalid MiniJava programs.
2. Check for expected behaviors, such as generating proper assembly code.

1.2 Definitions and Interpretations

Purpose: Verify that terms align with their definitions.

Methods: Manual review of code and documentation.

Steps:

1. Read through code, comments, and documentation to ensure that they adhere to the definitions.

1.3 Garbage Collection Mechanisms

Purpose: Validate that garbage collection works efficiently and properly.

Methods: Stress testing.

Steps:

1. Design test cases that create memory leaks if garbage collection fails.
2. Create cyclical references.
3. Attempt to allocate more resources than available.
4. Monitor memory usage to ensure memory is being freed.
5. Check for memory corruption or fragmentation after multiple garbage collection cycles.

1.4 Function Specifics and Requirements

Purpose: Confirm that specific functions behave as required in the SRS.

Methods: Unit testing.

Steps:

1. Isolate each function as possible.
2. Test multiple inputs, including edge cases.
3. Verify the outputs against expected results.
4. Have multiple people run tests, to ensure more edge cases are met.

1.5 Memory Allocation and Management

Purpose: Guarantee that memory allocation, deallocation, and management are handled correctly.

Methods: Dynamic analysis and profiling.

Steps:

1. Monitor memory allocations during runtime using a memory profiler or debugger.
2. Track dangling pointers.
3. Measure memory overhead to ensure it is within acceptable limits.

1.6 Error Handling and Exceptions

Purpose: Ensure that errors are caught and handled without causing system interrupts or failures.

Methods: Fault injection.

Steps:

1. Intentionally introduce faults.

2. Intentionally execute failures.
3. Ensure the system captures these inputs, then properly responds to them.

1.7 Specific Memory Operations and Processes

Purpose: Confirm that particular memory operations work as intended.

Methods: Unit testing.

Steps:

1. Focus on operations that directly interact with memory.
2. Track memory addresses and values before and after these operations.
3. Validate the operations result in expected memory locations.

Project Design Document:

The majority of the code for the garbage collection implementation will be part of the MiniJava C runtime. This is primarily done to avoid code bloat by abstracting GC operations as much as possible in the generated assembly code. (i.e the compiler will call these functions instead of performing the reference count operations itself inline.)

Because the runtime is written in C, it is not object-oriented, and there are no classes; however, this document will explain its functions in relevant detail.

During the code generation phase, the compiler will insert calls to the relevant runtime functions based on what is happening in the code (assignment, allocation, etc.). When an object and/or array is allocated by MiniJava's **new** operator, the compiler will first insert a call to either **runtime_alloc_object** or **runtime_alloc_array** to allocate the required memory.

We will primarily rely on reference counting garbage collection, while having mark-sweep garbage collection as a backup to catch cyclic garbage (as reference counting alone cannot free cyclic garbage). In order to support both of these methods, all memory blocks will be resized to include a header structure, to store additional information relevant to the garbage collection process:

```c
// 32-bit "tag" to identify heap blocks from all other memory
#define HEAP_BLOCK_TAG 'HBLK'

typedef struct HeapBlock {
    // Block identification
    u32 tag;                    // at 0x0

    // Next block in list of runtime allocations
    struct HeapBlock* next; // at 0x4
    // Size of this allocation
    size_t size;                // at 0x8

    // Mark bit (for mark-sweep GC)
    s32 marked : 1;             // at 0xC
    // Reference count (for reference count GC)
    s32 ref : 31;               // at 0xC

    // Block user data begins at offset 0x10 . . .
} HeapBlock;
```

Heap blocks contain a "tag" value, which is used during the mark-sweep process to determine whether any given 32-bit value is:

1. A valid pointer

2. A pointer to a heap allocation

Because C encodes pointers in the same way as other integral types, we must use heuristics to determine whether any given 32-bit value is a reference to another heap block. This is required for the marking phase of mark-sweep garbage collection, because we must traverse the object graph in order to determine which heap allocations are still reachable.

It can be safely assumed that a given 32-bit value **X** is a valid reference to another heap block if it is a valid pointer (one within the memory region which the program has access to), and if the 32-bits at that address are the expected heap block tag ('HBLK' in ASCII). This value is set by the MiniJava runtime at the point of allocation. Assuming the pointer is to a heap block, it is only valid if the pointer destination's next block is also valid, or is NULL.

Heap blocks also contain a pointer to the next heap allocation. This is set by the MiniJava runtime at the point of allocation. The MiniJava runtime also holds the head and tail elements of this linked list. The linked list is also used for mark-sweep garbage collection: specifically, the sweep process. After marking is complete, we simply iterate over all heap allocations using this list, and free the unmarked blocks. This is necessary because we need a way for the MiniJava runtime to reach blocks that are unreachable by the user program (garbage).

The next element in the heap block structure holds the size of the block. In addition to the tag system, our pointer heuristics also must not run past the end of the heap block, or they will read data unrelated to the specified block. In order to know where the block ends, we must keep track of the block's size. This value is also set by the MiniJava runtime at the point of allocation.

The heap block's reference count takes up 31 of the 32 bits at offset 0xC (12) bytes into the structure. This is to save space by having the mark flag only take up one bit, as opposed to one byte. The reference count is managed by the MiniJava runtime as the program executes, and determines whether a given heap block can be safely freed.

The **runtime_alloc** family of functions shall initialize the reference count to zero. This is because in a statement such as ”**Object o = new Object();**”, the reference count increment should occur in the assignment statement, not the call to **new**/the constructor.

The MiniJava runtime will be extended to include at least two additional functions for reference count manipulation: **runtime_ref_inc** (to increment a block's reference counter), and **runtime_ref_dec** (to decrement a block's reference counter).

The compiler will insert calls to **runtime_ref_inc** in situations where a given block **X**'s reference count must be incremented, such as:

- When block **X**'s address is used as an assignment rvalue

The compiler will also insert calls to **runtime_ref_dec** in situations where a given block **X**'s reference count must be decremented, such as:

- When a MiniJava variable **Y** (reference type) who points to block **X** is overwritten to point instead to block **Z**. Reference count operations should occur in the following order:
    1. **runtime_ref_inc(Z)**
    2. **runtime_ref_dec(X)**
- When a MiniJava variable **Y** (reference type) who points to block **X** goes out of scope. The variable no longer exists and thus cannot be used to reach block **X**.

In more complex statements which involve both incrementing and decrementing of the same block's reference count, the compiler must ensure that the incrementing occurs first wherever possible. Without this ordering, the following sequence of events can happen, which lead to an erroneous free operation:

1. A given MiniJava variable **Y** (reference type) points to a given block **X**. We assume for the purposes of this demonstration that Block **X** is only referenced by **Y**, so **X->ref** is one.
2. The following statement occurs in the code: **Y = X;**
3. If decrementation occurs first, **runtime_ref_dec(X)** will drop **X->ref** to zero, where it will either be individually freed or added to a zero count list (implementation dependent). This causes an erroneous free operation, because **X->ref** must still be incremented via its assignment to **Y.** This makes **Y** a dangling pointer, and the following **runtime_ref_inc(X)** call will corrupt memory.

The MiniJava runtime does not yet have a function to free memory. This will be added under the name **runtime_free**. The **runtime_alloc** family of functions calls down to the C standard library's **malloc** function to allocate memory blocks, so **runtime_free** will instead call **free** in the standard library.

To minimize overhead in the runtime system, we will not currently maintain a "zero count list" of allocations: instead, when a block's reference count is deemed unreachable (reference count of zero), it will be individually freed by the runtime system. This is likely slower than maintaining a zero count list; however, it will reduce both the memory footprint and complexity of the runtime system. In the future we will conduct more tests and determine whether it is worth pursuing optimizations such as a zero count list to minimize calls **runtime_free**.

A significant weakness of reference counting garbage collection is cyclic references. A cyclic reference is a recursive reference (i.e. an object which references itself). These pose a fundamental problem for reference counting because a cyclic reference must still contribute to an object's reference count. Therefore, given an object **X** containing a cyclic reference, **X->ref** must always be at least one (no matter how many levels of indirection exist). Without a way to detect cyclic references, this results in a deadlock where object **X** can never be freed by the garbage collector.

For example, consider a doubly-linked list structure. Each node in a doubly-linked list must point to the previous node in the list, and the next node in the list. In such a list where a given node **X** precedes a given node **Y**, if the list is not broken we can assert that **X->next == Y** and **Y->prev == X**. This results in a cyclic problem: **X** cannot be freed until its reference from **Y** is gone, but this reference cannot be removed until **Y** is freed. **Y** cannot be freed either because of its reference in **X**. If both **X** and **Y** are unreachable outside of their cyclic dependency, this creates a deadlock known as "cyclic garbage", where reference counting alone can never free this memory.

To combat cyclic garbage, we have decided to utilize mark-sweep garbage collection as a secondary method. Reference counting will do most of the work, so the mark-sweep garbage collection will only run when the runtime attempts to allocate memory and **less than 20%** of the system memory is available. We plan to measure the amount of free memory on the system using the **sysconf** function of the POSIX C standard library (unistd.h)

The mark-sweep system involves traversing the object graph and marking (by setting the mark bit) all objects that were reached through the object graph. The graph traversal begins at the "roots", or areas of data storage which are separate from the heap and accessible via no levels of indirection, such as: local variables on the stack and registers in the CPU. MiniJava does not support the **static** storage qualifier (with the exception of the program entrypoint), so this set of roots will not contain any "global" variables.

After marking objects, the sweep process iterates over all allocated heap blocks through the runtime's linked list, and frees the blocks that are not marked (unreachable).

Mark-sweep garbage collection will catch cyclic garbage because it will not be reachable through the object graph. The mark-sweep process will be implemented as part of the MiniJava C runtime (as with reference counting), into the following functions: **runtime_ms_mark**, **runtime_ms_sweep**, and **runtime_ms_recurse**. When the available memory drops low enough as specified earlier, the **runtime_alloc** family of functions will call both **runtime_ms_mark** and **runtime_ms_sweep**. **runtime_ms_recurse** is an internal function to simplify the mark-sweep system.

To demonstrate these algorithms, below is pseudocode representation of the mark-sweep garbage collection algorithm we plan to implement:

```
private runtime_ms_recurse(block: HeapBlock, mark: bool) {
    // Mark block
    block.marked = mark

    // Search for block references
    for word in block[sizeof(HeapBlock):]:
        if word is HeapBlock:
            runtime_ms_recurse(word, true)
}
```

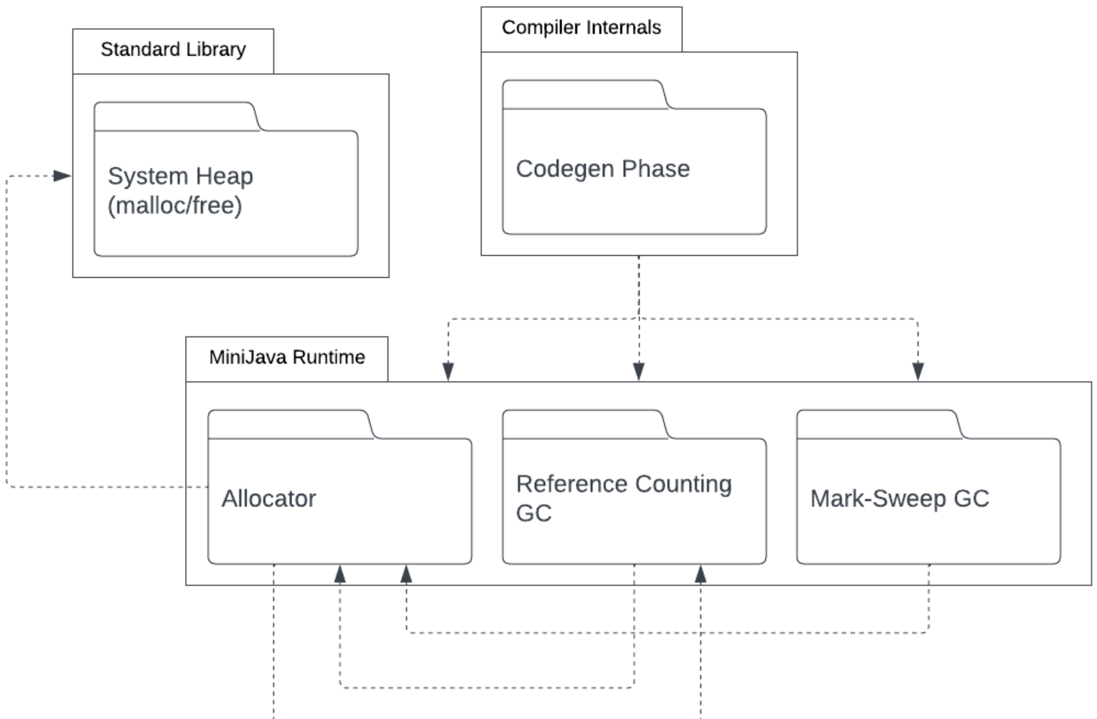
```
public runtime_ms_mark() {
    // Gather graph roots
    roots = [x for x in locals if x is HeapBlock]
          + [x for x in registers if x is HeapBlock]

    // Traverse roots
    for root in roots:
        runtime_ms_recurse(root, false)
}
```

```
public runtime_ms_sweep() {
    prev = null
    block = list_head

    // Traverse heap blocks
    while block != null:
        if not block.marked:
            if prev != null:
                // Repair linked list
                prev.next = block.next
            // Free unmarked blocks
            free(block)
        else:
            // Backup block
            prev = block
}
```

Below is the system architecture diagram for our overall garbage collection system:


Standard Library
System Heap (malloc/free)
Compiler Internals
Codegen Phase
MiniJava Runtime
Allocator
Reference Counting GC
Mark-Sweep GC