Assignment 4: Garbage Collection (GC)

Ali Zartash

This assignment involved the implementation of a garbage collection mechanism for the MITScript VM from assignment 3. The assignment 3 VM while operating creates a lot of MITScript objects but never deletes them which results in memory leaks. This assignment fixes that.

1 Design

The GC mechanism I opted for was the very simple Mark-and-Sweep method. This was because the commonly used Reference Counting mechanism is prone to reference cycles resulting in memory leaks which this mechanism is immune to. There exist other faster mechanism as well based on generational garbage collection techniques, but the Mark-and-Sweep is much more simpler to implement and debug. A key decision I made was to not collect functions or the constants used within the functions. These are features of the bytecode and I wanted to keep them separate from the objects created by the VM interpreter. Thus all the functions and the constants in their metadata are stored on the C++ heap and are deleted after the execution of the program is over (they do not even count towards the memory used by the program). But for the other objects used by the program, we use mark and sweep.

The mark and sweep mechanism works by first identifying a root-set of objects that are needed by the program (this is just the stack of stack-frames). These objects are marked (as needed). Then the pointers held by these are also marked and their dependencies are also marked recursively. At the end all unmarked objects are deleted. To allow the marking to take place, I made all the object types (Booleans, frames, closures, records, etc) extend an Abstract Collectable type. The follows method in these collectable then marks all the dependencies of the object recursively.

To keep track of all the marked/unmarked objects, I implemented a heap object (the starter code for which was given by 6.S081 staff). Whenever some object is allocated on this heap, the heap also keeps track of it in a linked list. The linked list is doubly linked to allow deletion in the middle. When garbage collection is triggered the GC iterates over this linked list and deletes and removes all unmarked objects freeing the memory.

One important decision was also when to trigger garbage collection. After each instruction the usage is checked and is triggered whenever the GC is over 77% full. This allows it to be infrequent, thus speeding the VM up but also keeps memory usage under control.

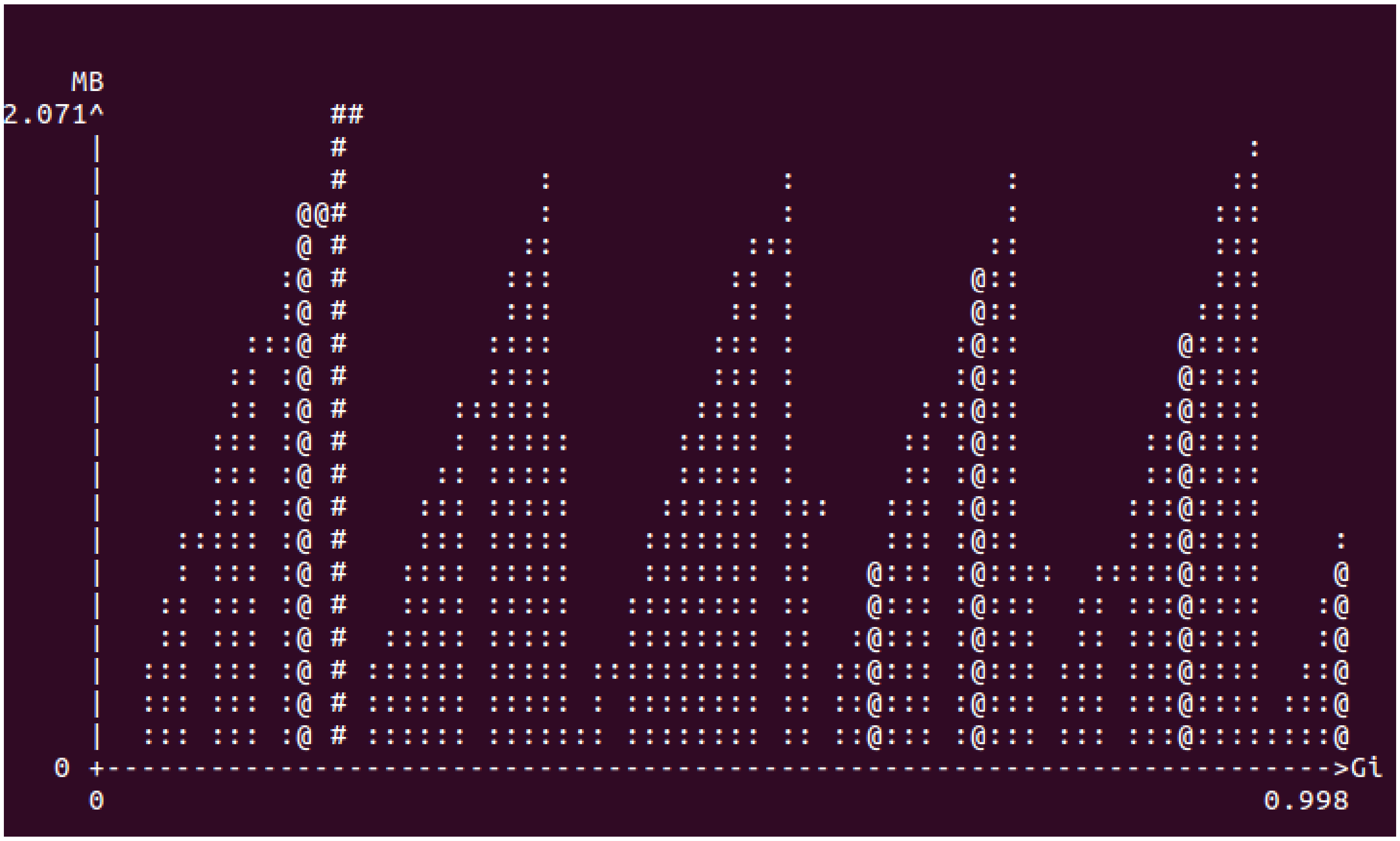
2 Extras

Usage:

```
mitscript    (file_path)    -s | -sp | b     [-mem N]
```

The other options/flags are described in the previous Assignment 3.
The mem flag tells the compiler to limit memory to N megabytes.

The VM sill leaks about 200+16000 bytes but the source of these is the flex/bison yyalloc calls as opposed to the VM itself. The VM cleans up everything else on exit. With regards to GC which is relevant during the program, here is a massif screenshot:



```
    MB
2.071^             ##
     |             #                                                    :
     |             #           :             :            :            ::
     |           @@#           :             :            :           :::
     |           @ #          ::           :::           ::           :::
     |          :@ #         :::          :: :          @::           :::
     |          :@ #         :::          :: :          @::          ::::
     |        :::@ #        ::::         ::: :         :@::         @::::
     |       :: :@ #        ::::         ::: :         :@::         @::::
     |       :: :@ #      ::::::        :::: :       :::@::        :@::::
     |      ::: :@ #      : :::::      ::::: :      :: :@::       ::@::::
     |      ::: :@ #     :: :::::      ::::: :      :: :@::       ::@::::
     |      ::: :@ #    ::: :::::     :::::: :::   ::: :@::      :::@::::
     |    ::::: :@ #    ::: :::::    ::::::: ::    ::: :@::      :::@::::    :
     |    : ::: :@ #   :::: :::::    ::::::: ::   @::: :@::::  :::::@::::    @
     |   :: ::: :@ #   :::: :::::   :::::::: ::   @::: :@:::  :: :::@::::   :@
     |   :: ::: :@ #  ::::: :::::   :::::::: ::  :@::: :@:::  :: :::@::::   :@
     |  ::: ::: :@ # :::::: ::::: :::::::::: :: ::@::: :@::: ::: :::@::::  ::@
     |  ::: ::: :@ # :::::: ::::: : :::::::: :: ::@::: :@::: ::: :::@:::: :::@
     |  ::: ::: :@ # :::::: ::::::: :::::::: :: ::@::: :@::: ::: :::@::::::::@
   0 +----------------------------------------------------------------------->Gi
     0                                                                   0.998
```

This is with a 4MB limit and I trigger collection at less than 50% in this screenshot – the actual GC is triggered at 77%. This is for a simple while loop with assignments that results in creation of many Integer and Boolean objects – which the GC destroys.

3 DIFFICULTIES

This was a relatively easier assignment compared to assignment 3, perhaps because I decided to go ahead with mark and sweep as opposed to the more advanced collectors. I also had to fix all the memory leaks for the previous modules like the compiler and the AST. But upon exit the VM has freed almost all memory (there are some leaks in Flex/Bison yyalloc) but none related to the VM.

4 CONTRIBUTIONS

This version of Assignment 4 has been done entirely by me (Ali Zartash). My teammates worked on a separate implementation – this is because our group is a group of 4 and I felt like I can learn more by implementing the whole VM and GC by myself – the base VM implementation used for the GC was also entirely my own. This is also a good example of N-version programming so when we are optimizing for performance we can pick and choose the better/faster parts from either implementation for Assignment 5.