CS 24: Introduction to Computing Systems Fall 202X

Extra Credit 01: GC (due ~~Monday,~~ Thursday November 16 @ 11:30pm)

In this extra credit, you will write two algorithms to support garbage collection in a Python-like interpreter. This project will be graded all-or-nothing based on the gitlab tests. Full credit will add +~~3~~1% to your final grade. All extra credit is applied after the cutoffs are already set.

So far, we have seen how Java is run, compiled a small version of BASIC to machine code, and implemented a memory allocator for a low-level language. This time, we continue our exploration of systems-level programming by writing several garbage collection algorithms for a Python-like language called `subpython`.

`subpython` Interpreter

We have provided you with an interpreter capable of evaluating Python-like expressions in a Read-Eval-Print Loop (REPL). You can build it with the provided `Makefile`, and then give it a try:

```
compute-cpu2>./subpython
Subpython [CS24 Fall 2019]
Using a memory size of 1024 bytes.
>>> 2 + 3
5
>>> "hello"
"hello"
>>> a = 5
>>> b = 9
>>> a + b
14
>>> lst = [1, "goodbye", 3]
>>> lst[1]
"goodbye"
>>>
```

This interpreter is only *Python-like*. It has a syntax similar to Python, but it only supports very limited functionality:

- Supported data types are integers, strings, lists and dictionaries along with the values None, True and False.
- Basic comparisons and arithmetic are supported.
- The only available control-flow structures are if and while statements.
- Global variables are supported, and can be assigned to multiple times.
- Global variables, list elements and dictionary entries can be deleted with the `del` command, e.g. "`del a`".
- Dictionary keys may only be `None`, `True`, `False`, integers, or strings.
- Both list elements and dictionary entries may be the target of an assignment. New dictionary entries may be created by assigning to a key that is not in the dictionary. Existing ones may be updated using a similar mechanism.
- Lists cannot be extended.

This is sufficient to generate very interesting graphs of values within the interpreter. For example, you can do fun things like this:

```
>>> lst = [1, "b", 3]
>>> lst[2] = lst
>>> lst
[1, "b", [1, "b", [1, "b", [1, "b", [..., ..., ...]]]]]
```

Finally, the interpreter provides these functions:

- `quit()` exits the interpreter. (You can also exit the interpreter with Ctrl-D, etc.)
- `mem()` prints the number of used bytes of memory
- `gc()` manually invokes the garbage collector.
- `len(v)` returns the length of a list, string or dictionary
- `print(v,...)` will print the passed in value(s)

Caution: Lots of Code!

In most previous projects, the code we have written has been either isolated from the code you're implementing or small enough that you could understand all of it. **For this project, neither of these is true, and it is an intentional, learning-related decision.**

The `subpython` Allocator

When the `subpython` interpreter starts, it allocates a chunk of memory called the "memory pool" which can be used throughout the program. To manage the memory pool, the interpreter uses an explicit list allocator similar to what you probably implemented last week. If the pool runs out of memory, `subpython` reports an out-of-memory error.

Initially, allocations are placed sequentially at the start of the pool. When an allocated value becomes free, it is added to a singly-linked explicit free list. There is a special `value_t` type `VAL_FREE` for values in the free list. Later allocations are served from the free list using a best-fit approach. The allocator performs splitting, but not coalescing, since the garbage collector (which you will implement later) coalesces free space automatically.

As discussed in lecture, we can use *references* instead of pointers to allow us to eventually move the contents of memory around and compact the pool. Just by introducing this one level of indirection, the allocator can implement some very sophisticated functionality.

In our system, a "reference" is simply an index into the `ref_table` maintained by the allocator (in `refs.c`); this table records the actual address of each value. Global variables (managed in `eval.c`) store a `reference_t` to the value associated with the name. Lists and dictionaries use `reference_t`s to record the keys and values they hold. If code needs to look up the actual value from a reference, it must use the `deref(reference_t)` function provided by `refs.c` to get back a pointer to the `value_t`.

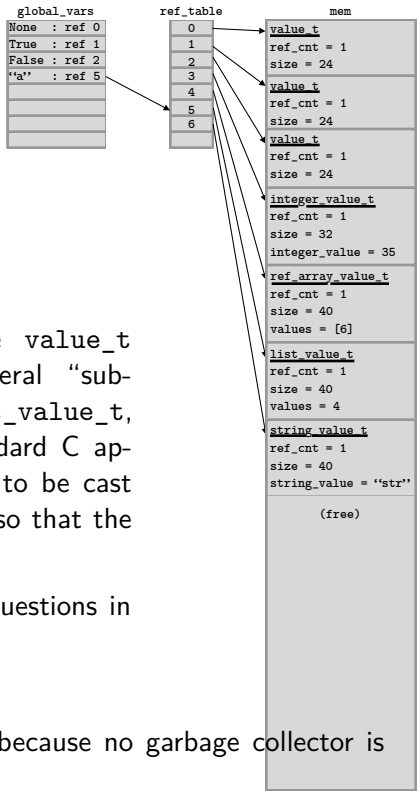
(Note that `-1` is a special value for references; it is a "null" reference that doesn't refer to anything. The file `types.h` defines a symbol `NULL_REF` that should be used within the code.)

This allows the allocator to change the actual address of values without breaking any other part of the program. As long as the `ref_table` entries are updated properly, a `value_t` can be moved within the memory pool without any problem.

You can make `subpython` print out the current state of its memory pool by giving it the `-d` argument. If you do this and feed in the above statements, you will see exactly the above results. Additionally, you can specify a different pool size with the argument `-m` size.

To help you understand how everything works together, here is a diagram of what the allocator will have in memory after running the following two commands:

```
>>> 35
35
>>> a = ["str"]
```



Task 0: The `value_t` Structs

All values in the interpreter are represented by versions of the `value_t` struct (in `types.h`). You will notice that `value_t` has several "sub-types," `integer_value_t`, `string_value_t`, `list_value_t`, `dict_value_t`, `ref_array_value_t`, and `free_value_t`. These types use the standard C approach of having the same leading members; this allows a `value_t*` to be cast to the appropriate sub-type (see the `type` member in the `base` field), so that the specific members can be accessed.

In order to familiarize yourself with these types, answer the following questions in the `answers.md` file in your repository.

Task 1: Reference Counting

In its present form, the interpreter will eventually run out of memory because no garbage collector is implemented.

Notably, the `ref_count` variables look...wrong. This is because you have to implement the `incref` and `decref` functions which get called by the `subpython` evaluator every time a reference is added or removed, respectively.

Your task is to implement these reference counting functions so values that no longer have any references are reclaimed by the memory allocator. Each `value_t` (stored in the memory pool) has a `ref_count` member that should be increased or decreased accordingly. If the `ref_count` is 0, then no other value refers to this one; so, it is definitely garbage and should be collected.

Garbage values must to be marked as free and added to the free list using the `mm_free` function. The corresponding entry in the `ref_table` also needs to be set to `NULL` so the reference can be reused. When a value becomes garbage, it decrements the reference count of each value it is referencing as well. For example, in the following subpython code, `del x` causes both the list and the 1 to be freed:

```
1 x = [1] # reference graph: x -> VAL_LIST -> VAL_REF_ARRAY -> 1
2 del x # VAL_LIST becomes garbage, which makes
3       # VAL_REF_ARRAY garbage, which makes 1 garbage
```

Q0: Implement reference counting in the file `refs.c`, between the "`//// REFERENCE COUNTING ////`" comment and the "`//// END REFERENCE COUNTING ////`" comment. Once you've finished, `make test1` should run successfully.

Task 2: Fixing Dictionaries

The subpython evaluator already makes calls to `incref` and `decref` in most of the places where a reference to a value is added or removed. However, these calls are missing in the implementations of dict operations. Your task is to add all the necessary `incref` and `decref` calls to the following functions in `eval_dict.c`:

- `dict_bool`: called to convert a dict to a bool (e.g. in `not not {1: 2}`)
- `dict_len`: called to get the size of a dict (e.g. in `len({1: 2, 3: 4})`)

- `dict_subscr_get`: called to get the value corresponding to a key (e.g. in `d = {1: 2}; d[1]`)
- `dict_subscr_set`: called to set the value corresponding to a key (e.g. in `d = {1: 2}; d[1] = 3`)
- `dict_subscr_del`: called to delete a mapping from a dict (e.g. in `d = {1: 2}; del d[1]`)

Some of these functions may not need to change reference counts. The corresponding functions for lists (in `eval_list.c`) should serve as a guide for where the reference counting calls are needed.

Q1: Fix reference counting in the file `eval_dict.c` by adding calls to `incref` and `decref` in the right places. Once you've finished, `make test2` should run successfully.

Task 3: Stop and Copy

Now that you've implemented reference counting, you should have a sense for why subpython (and real Python) use it as their default mechanism for identifying garbage. It frees garbage as soon as possible and it is very fast because it only checks if a value has become garbage when its reference count is decremented. However, it can't recognize cycles of garbage values referencing each other. To solve this problem, both subpython and Python rely on periodic garbage collection to collect cycles of garbage.
Your task is to implement the stop-and-copy algorithm for the subpython allocator. This requires splitting the memory pool into two halves:

- The "from-space", where allocations are performed
- The "to-space", which is (currently) all garbage

When `collect_garbage` is called, it swaps the from-space with the to-space. Any non-garbage values that were in the from-space must be moved to the to-space. Non-garbage (or "reachable") values are defined as values that are referenced directly or indirectly by global variables.
Some implementation notes:

- Points will be deducted for using more than a constant amount of additional memory for stop-and-copy. It is possible to implement it without using any additional memory.
- The function `is_pool_address` can be used to determine if an address is inside a memory pool. It is documented in `mm.c`.
- The function `foreach_global` can be used to iterate over the global variables. It is documented in `eval.c`.
- To avoid fragmentation, the copied values must be "compacted", i.e. placed contiguously at the start of the to-space
- The `ref_table` needs to be updated to point to the relocated values. As in task 1, garbage references also need to be removed from the `ref_table`.
- The free list and reference counts may need to be updated
- Once you've finished, make should run successfully.

Q2: Implement stop-and-copy in the file `refs.c`, between the "`//// GARBAGE COLLECTOR ////`" comment and the "`//// END GARBAGE COLLECTOR ////`" comment. Once you've finished, `make` should run successfully.