11

Memory Management

Exercises

11.1 C functions `malloc` and `free` have the following prototypes:

```
void *malloc(size_t size);
void free(void *ptr);
```

The first allocates at least `size` bytes of memory and returns a pointer to the first byte of that block. If such a block is not available, the `null` pointer is returned.

The second deallocates that block of memory referenced by `ptr` that had been allocated by `malloc`.

These functions are thus similar to the functions *new* and *delete*, except that they exclude the idea of marking a memory block *unused* or *undef*. In C, unused blocks of memory are not so marked and the "undefined" value has no predefined meaning.

11.2 Perl provides reference-based garbage collection, which is built into the language. So unreachable memory blocks with a non-zero reference count will not normally get freed. However, when a Perl thread exits, a more complete mark-and-sweep garbage collection is performed, so that everything allocated by that thread gets destroyed.

Memory leaks, widows, and orphans in Perl are just as likely as they are in C and C++.

11.3 Here are the major ones:

- In a declaration, the value of `n` is negative or zero.
- In a reference, the value of `n` is outside the subscript range for `A`.

11.4 Using the abstract syntax in Fig. 2.14, Type Rule 6.2 should be expanded to read:

> All declared variables and arrays must have unique names. All declared arrays must have an `int size` whose value is positive.

Type rule 6.4 part 2a should be changed to read:

> The target Variable or ArrayRef is declared. An ArrayRef must have an `index` whose type is `int` and whose value is between 0 and the array's declared `size - 1`.

Type rule 6.5 part 2 should be expanded to read:

> A Variable or ArrayRef is valid if its `id` appears in the type map. The index of an ArrayRef must have type `int` and value between 0 and the array's declared `size - 1`.

11.5 Meaming Rule 11.1 would be incorporated into Meaning Rule 10.1 part 1, to cover cases where f's `local` is an array declaration. Meaning Rule 11.2 would be incorporated into Meaning Rule 10.1 part 2, to cover cases where c's `arg` is an array reference.

11.6 Meaning Rule 11.1 would be changed to add the entire array to the stack, alongside its dope vector, rather than the heap. The remaining meaning rules for arrays are not affected by this change. The overall impact on the program is that the stack size would increase and the heap requirements would be smaller.

11.7 While the stack and heap space requirements would change, as noted above, no appreciable impact on the program's run-time performance would occur. That is, array arguments can still be passed by reference, whether they are originally allocated in the stack or in the heap. However, the dynamic allocation and addressing of arrays in the stack is a far more complex problem than this question suggests.

11.8 To answer this question, students will need to be given the sources for Clite extended, which are available only at the instructor's web site.

11.9 Here is a sketch of the algorithm, assuming that the heap's address range is $\{h, \ldots, n\}$ and that each node has three addressable units of memory: a reference count (`rc`), a value (`value`), and a pointer to another node in the heap (`next`). The function `delete` is called whenever a pointer deletion occurs in an application program. The method `N = M` is called wheneever a pointer assignment occurs, the method `rcInit` is called when the application program begins, and the method `new` is called wnenever a `p = new node;` statement is executed. Here are their definitions:

```
free(N):   N.next = free_list;
           free_list = N;
           free_list.value = undef;
           free_list.rc = 0;
delete(N): N.rc = N.rc - 1;
           if (n.rc == 0) {
              delete(N.next);
              free(N);
           }
N = M:     delete(N);
           M.rc = M.rc + 1;
           N = M;
rcInit:    free_list = n;
           free_list.next = null;
           for (int i = n-1; i >= h; i--)
              free(i);
new(p):    if (free_list != null) {
              p = free_list;
              p.rc = 1;
              p.next = null;
              free_list = free_list.next;
           }
           else abort; // the heap is full
```