Not-Homework Review

Identify the site of the first type error if *static* or *dynamic* scoping is used:

```
new a := 5;
if (a = 4) { write a > true; }
else { a := false; }
write not a;
```


Lifetime Management

We know about many examples of *heap-allocated* objects:

- Created by `new` or `malloc` in C++
- All `Objects` in Java
- *Everything* in languages with lexical frames (e.g. Scheme)

When and how should the memory for these be reclaimed (*freed*)?

Manual Garbage Collection

In some languages (e.g. C/C++), the programmer must specify when memory is de-allocated.

This is the *fastest* option, and can make use of the programmer's expert knowledge.

Dangers:

- **Dangling references**: Extant links to already-freed memory
- **Memory leaks**: Unused memory that is never freed.

(The programmer might not be such an expert after all!)

Automatic Garbage Collection

Goal: Reclaim memory for objects that are no longer *reachable*.

Reachability can be either direct or indirect:

- A name that is *in scope* refers to the object (direct)
- An data structure that is reachable contains a reference to the object (indirect)

This approach is relatively *conservative* (could miss some deallocations) but *safe* and usually effective.

Reference Counting

Each object contains an integer indicating how many references there are to it.

This count is updated continuously as the program proceeds.

When the count falls to zero, the memory for the object is deallocated.

Analysis of Reference Counting

This approach is used in filesystems and a few languages (notably PHP and Python).

Advantages:

- Memory is freed as soon as possible.
- Program execution never needs to halt.
- Over-counting is wasteful but not catastrophic.

Disadvantages:

- Additional code to run every time references are created, destroyed, moved, copied, etc.
- *Cycles* present a major challenge.

Mark and Sweep

Garbage collection is performed periodically, usually halting the program.

During garbage collection, the entire set of reachable references is *traversed*, starting from the names currently in scope (the *root set*).

Each object is *marked* when it is seen.
After the traversal, any unmarked objects are deallocated.

Analysis of Mark and Sweep

This is the most common GC technique in programming languages.

Advantages:

- Very aggressive strategy; there will be no un-referenced objects left in memory.
- Does not slow down normal execution. No effect whatsoever on programs with a small memory footprint.

Disadvantages:

- Potential to halt execution unpredictably — not suitable for real-time systems.
- *Undercounting* will cause dangling references.
- Deallocation is always delayed.

GC Tricks and Tweaks

- **Generational garbage collection**: Takes advantage of the observation that newer objects are more likely to be garbage.
- **Stop and Copy**: Like mark-and-sweep, but instead of marking reachable objects, copy them to another part of memory. Then free all of the old memory space at once.
- **Weak References**: Allow programmer to specify that some references should not keep an object alive by themselves. Example: keys in a hash table
- **Conservative GC**: Assume every integer is a pointer.
- Reference counting with **delayed cycle detection**
- **Incremental** mark and sweep

Class outcomes

You should know:

- Manual vs. Automatic Garbage Collection
- Reference counts vs. Mark-and-sweep
- What *reachability* is, and how it is determined
- The shortcomings of different methods for automatic garbage collection, and how they are mitigated in practice.

You should be able to:

- Show the reference counts of objects in a program trace.
- Perform a mark-and-sweep operation at any point during the execution of an example program.