CMPSCI 630 Systems **Spring 2022**

Lecture 05

Lecturer: Emery Berger *Scribes: Maliha Islam, Nicolas van Kempen*

5.1 LISP

LISP (LISt Processor) is a family of programming languages that follow a fully parenthesized prefix notation. As the name suggests, it makes extensive use of lists. We continued the discussion on LISP, touching upon Common LISP - a more modern and prominent descent of LISP. More specifically, we discussed cons cell and the most commonly associated functions car and cdr.

A cons cell is simply a pair of objects. It is the building block of more complicated data structures (e.g., lists). Typically, the first element of a cons cell points to a value and the second element points to the rest of the list (e.g, another cons cell). However, there is no strict rule here and the two elements of a cons cell can point to anything. A cons cell is created by the function *cons*, and elements in the pair are extracted using the functions *car* and *cdr*. *car* returns the first element, while *cdr* returns the second. The name of these functions actually come from the hardware: car was named after the address register, cdr after the decrement register. For example:

(car (cons “a” 4))
⇒ “a”
(cdr (cons “a” 4))
⇒ 4

5.2 Scope

Scope generally refers to valid name binding, i.e., the parts of a program where a name refers to a certain entity or variable. Since the same name can refer to different entities at different times, programs usually take great care to understand what a particular name refers to at a given point in the code. Scope is of two types- *lexical* and *dynamic*.

Lexical scope is used in most programming languages due to being relatively straightforward. Simply put, lexical scope means name bindings are resolved wrt to the region of code where the name was defined. The immediate surrounding scope is first checked, then the enclosing one, and so on until either the variable is found or a variable not found error is generated. When using a compiled language, this is easily done at compile time. With an interpreted language, the interpreter has to keep track of variable definitions at each scope level. Lexical scoping is dependent on the program text, and does not require the runtime call stack to resolve name bindings. On the other hand, dynamic scope is always determined wrt to the execution context. Entities or variables can be referenced from anywhere, regardless of where they are defined. For example, if a name’s scope is within a certain function, then its scope is the time-period during which the function executes. Since functions can be called from anywhere in the code, resolving such bindings is far more complicated. To resolve name bindings, each name typically has a global stack of bindings. Introducing a variable name pushes a binding on to the stack and popped off when the program control leaves the scope. The top of the stack always determines what the name currently refers to.

Lexical scope is far more popular than dynamic scoping due to a couple of reasons. First, lexical scope is easier to determine since local naming structure can be understood during compilation time. Dynamic scope cannot be determined in such a fashion. With dynamic scope, errors always happen at runtime. Second, lexical scope follows a "locality of reasoning" (Emery Berger special term), i.e., it allows us to reason about entities in isolation. This makes it much easier to make modular code and reason about it. In a dynamic scope, determining the context of a variable is much more involved from a human perspective.

Going back to LISP, it was supposed to have lexical scoping as McCarthy designed it. However, by mistake, it had dynamic scoping. One can understand how this mistake can be made in the context of interpreted languages, where it makes sense to just walk the stack for variable name resolution looking for a value, instead of searching for that variable's definition. This was "fixed" with Common Lisp, where lexical scope became the standard.

5.3 Garbage Collection

Garbage collection (GC) is essentially automatic memory management- reclaiming memory allocated by a program, that is no longer in use. GC takes the burden away from programmers to manually allocate/free unreferenced memory. We learned about two types of garbage collection algorithm— *reference counting* and *mark-sweep*. The original McCarthy LISP paper invents both. They are still in use today. To cite a few languages for each: Python, Swift, *C++* (smart pointers) for reference counting, Python, Java for mark-sweep.

In reference counting, the garbage collector maintains a count of the number of pointers to each object in memory. This count is incremented or decremented as necessary when a reference to the object is created or destroyed. When no pointer are left to an object, i.e., it is unreachable, that memory is reclaimed. Reference counting is a simple algorithm and unused memory is reclaimed almost immediately. However, if the number of pointers to an object increases, the number of bits needed to store that number increases. This leads to higher memory usage compared to mark-sweep discussed below. It also cannot handle objects organized in a cyclic structure (linked list is a big offender), which makes it an *incomplete* garbage collection algorithm, since it cannot guarantee all memory will eventually be reclaimed. It can also be inefficient as reference counting needs to work every time a pointer is assigned/modified.

Mark-Sweep consists of a mark phase followed up by a sweep phase. Compared to a reference counting GC, mark-sweep only requires one "mark" bit per object. In the mark phase, the collector starts from all the roots (objects, variables, etc.), pointers on the stack and registers, and traces what other objects are reachable. It marks all the reachable objects until there is no unexplored path. In the sweep phase, all unmarked memories are reclaimed as they are considered to be unreachable by all other objects in the program. This algorithm is run intermittently and do not need to run after every pointer operation. This algorithm can lead to fragmentation as it sweeps whatever memory is unreachable at a given time. This is typically not a huge problem in practice.

A couple optimizations are possible: first, the mark bit value's meaning is usually swapped every mark phase, which saves the need to reset all mark bits, since they were all set to a particular value during the previous mark phase. Second, the sweep phase can be delayed / run in parallel, since we know those objects are not reachable by the user code ("mutator" in mark-sweep GC vocabulary). This allows shorter GC "pauses".