CMPSCI 691ST Systems **Fall 2011**

Lecture 3

Lecturer: Emery Berger *Scribe: Dan Barowy*

3.1 Recursive Functions of Symbolic Expressions and Their Computation by Machine

This paper is about LISP, a language developed by John McCarthy and others at MIT. Unlike FORTRAN, which had the goal of making numerical computation easier (specifically, TRANSlating FORmulas into executable machine code) for scientists and mathematicians, LISP is based off of a mathematical logic, Church's (untyped) lambda calculus. The lambda calculus attempts to capture and formally express ideas that are computable in principle.

McCarthy invents a huge number of new concepts in this paper: reference-counting and mark-sweep garbage collection, lazy evaluation, macros, conditional compilation, lambdas, dynamic typing, recursive subroutines and the stack, "first-class" functions, and compound list-based data structures.

3.1.1 Garbage Collection

Two main strategies: mark-sweep and reference-counting. LISP called this "reclamation".

Mark-sweep works as follows: Starting at a root object (or many root objects), the algorithm follows pointers, marking each object encountered as it traverses the heap. After marking is finished, a "sweep" phase is started that sometimes manipulates the objects left over (e.g., compacting them into the space now no longer occupied by the unmarked objects), although LISP itself did not employ this strategy. LISP uses a "bump pointer" to divide "used" and "free" memory into two segments. When the dynamic memory subsystem is invoked, LISP "bumps" the pointer upward. LISP's "sweep" phase consists in shrinking the used area by moving the pointer backward if objects have been freed and there is space available. LISP's simple GC scheme means that fragmentation can effectively exhaust memory even if sufficient memory is available. This can be solved with compaction.

Reference-counting works differently, by counting the number of in-references to each object. When an object's reference count drops to zero, the collector knows that that object, and all objects referenced by that object, can be freed. McCarthy only mentions reference-counting parenthetically, and LISP does not use it. Reference-counting's major drawback is that it cannot deal with circular reference chains, as the reference count of self-referential objects will never drop to zero, and thus will never be "reclaimed" (i.e., the application will "leak" memory).

If the size of the "live set" is H and the size of the total memory is M, marking is $O(H)$ and sweeping is $O(M)$. Ideally, you want your garbage collector to be $H << M$. Modern mark-sweep collectors perform sweeping lazily (i.e., they do nothing to freed memory until said memory is reallocated). Modern collectors often combine bump pointers with "free lists", a list of free memory locations.

Figure 3.1: Heap before and after marking.

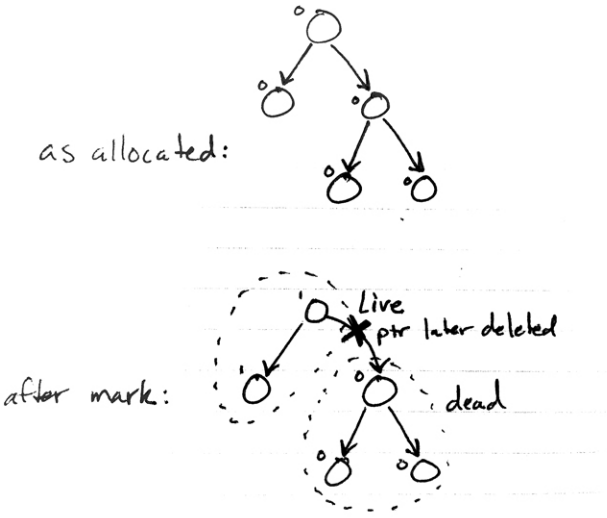
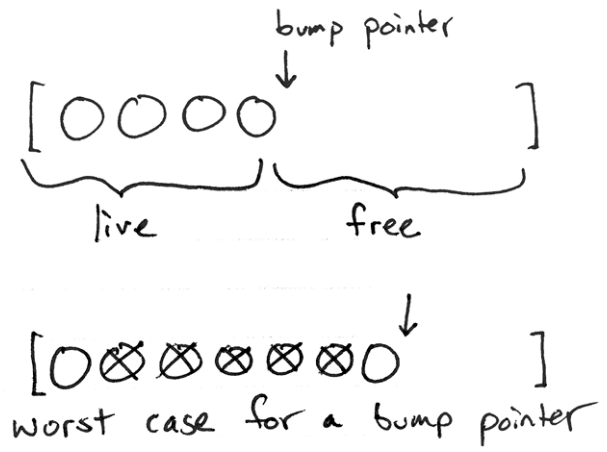


Figure 3.2: Bump-pointer allocator.



3.1.2 Lazy Evaluation

LISP employs a limited form of "lazy evaluation" strategy for propositional expressions. For example, if the first term in the expression $(x < y) \wedge (b = c)$ is not true, the second term, $(b = c)$ need not be evaluated. Later languages have extended this concept to apply to any computation.

3.1.3 Scope

LISP employed a dynamic scope scheme, although this was not originally intended. Common LISP and Scheme (standardization efforts led by Guy Steele) switched to static scoping.

Dynamic scoping means that the meaning (i.e., the "binding") of a variable is determined at runtime. In order to determine the value of a variable, the runtime first examines the value of the variable in the current execution context. Failing to find a binding there, the runtime recursively ascends the stack, looking for a binding. LISP used an association list (a key-value data structure) to store these values in each scope.

LISP also allowed for closures in function expressions. Function parameters were "bound" to the execution context of the function body, however any remaining undefined variables were considered "free" and would be resolved using dynamic scoping rules.

3.1.4 Syntax and Semantics

LISP employed an unusual syntax for S-expressions ("symbolic expressions"), which was essentially parenthesized Polish notation. This made parsing easy at the expense of readability.

Furthermore, everything in LISP was an expression. Statements that only performed side-effects did not exist.

3.1.5 Digression: Locality

Locality is an important design consideration for software on modern architectures. Architectures around the time of LISP did not have complicated memory hierarchies as do current machines (which have many levels of caches), so locality was less of an issue.

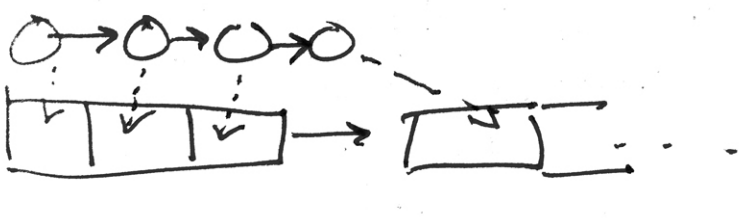
There are two kinds of locality: spatial and temporal.

Spatial locality, where a unit of data is in the machine's memory, is a property of the allocator.

Temporal locality, how a unit of data differs over time, is a property of the program.

Low spatial locality (related data are stored distantly) can have significant effects on program runtime, thus allocators strive to keep related data "close together"", hence in the cache together, justified by the observation that if a unit of data is accessed, its neighbor is often also accessed. Linked-lists can often be "vectorized" (and their locality improved) by moving their data into linked arrays. B-trees are a common data structure explicitly designed to achieve high locality.

Figure 3.3: List vectorization.



3.1.6 "First-class" Functions

Languages that allow the programmer to treat a function as both data and as executable code support so-called "first-class" functions. LISP converts S-expressions into a string-like representation and allows passing this data around in a program. The `eval` M-expression allows this string to be converted back into code and executed.

Note: Languages with an `eval` function are extremely hard to make secure, something of a non-issue in 1960.

C/C++ support a limited kind of function-passing, i.e., function pointers. Modern functional programming languages make passing functions as easy as passing objects, and some of them even type-check the passed functions.

3.1.7 Recursion

LISP strongly supported recursion. Recursive functions could be defined in terms of themselves, and at runtime, the system could call the same function any arbitrary number of times (subject to stack overflow). The relatively novel stack data structure (which FORTRAN lacked) allowed for this freedom. Recursion was considered to be a luxury.

3.1.8 Issues with LISP

1. Early versions of LISP could not represent circular lists.
2. Dynamic scoping was confusing to programmers.
3. Garbage collection was slow and had memory fragmentation issues.
4. LISP's syntax is hard to understand. "Somewhat readable" in McCarthy parlance.
5. `eval` requires dynamic typing, although it allows a much-loved PL feature: "bootstrapping", i.e., writing the compiler for the language in the language itself using the interpreter. Use of `eval` prevents an entire program from being compiled, as the values of the evaluated strings may not be known at compile-time.

3.1.9 Features of LISP Only Recently in Common Use

1. Garbage collection
2. Conditional compilation
3. Lambda expressions
4. Dynamic typing (untyped lambda calculus)

3.1.10 Digression: Why Java?

Why is Java so popular?

Java was designed with the intention of being embedded. Originally named "Oak". Designers solved common embedded portability, reliability, and security issues by defining an abstract "virtual machine" which interpreted a contrived bytecode. This VM had a stack-based architecture. Programs moved to different architectures did not need to be recompiled; only the native Java runtime was required. Java included a standard library that was truly standard across all supported platforms.

Sun realized that the system could be retargeted for the web browser and that applications could be distributed over the Internet. It had already solved many of the issues that needed to be solved around the time that the Internet was popularized. It has since become very popular on server platforms, where the startup cost is amortized over very long run times. The compiler has also been switched to just-in-time (JIT).

3.1.11 Why Not LISP?

With all of its advanced features, why didn't LISP catch on? It did not solve the problem at hand at the time: numerical computation. LISP was extremely inefficient at this task, even though programs could be expressed elegantly. FORTRAN was a better choice, and FORTRAN is still widely used by scientists and engineers despite its many now-archaic limitations.