CMPSCI 630 Systems **Fall 2015**

Lecture 2

Lecturer: Emery Berger *Scribe: Bobby Powers*

2.1 Lisp

Lisp (sometimes all caps for LISt Processing) was developed almost immediately after Fortran, and is an implementation of Church's lambda calculus. This is in stark contrast to Fortran, whose syntax is loosely based on math but contains little formalism. With Lisp, syntax and parsing go away due to its prefix notation and parentheses (which is good or bad depending on your perspective). Programming Language people love the syntax, as there is no syntax. It was very ahead of its time with respect to code modifying code.

Lisp has several primitives, like lists:

```
(1 2 3 x)
```

Lists can contain items of any type, including other lists. There is no need for all the items in a particular list to be of the same type.

And atoms:

```
1 x foo-bar
```

Example Lisp function:

```
(defun foo (a)
    (let ((bar 0))
        (+ bar a)))
```

This function takes a single parameter (`a`) creates a local variable (`bar`) bound to the value 0, and returns the result of adding `a` and `bar`. Lisp doesn't have explicit return statements, the value of the last statement executed in a function is used as the result.

2.2 Compilation

Compilers take source-code as input, lex and parse it to produce an internal representation of structure, performs optimizations, and then emits assembly or machine code.

The compiler's internal representation of structure can take several forms, including control flow graphs (CFGs), dependency graphs, and abstract syntax trees.

Control flow graphs track the paths that execution can proceed in a given function, file, library or program. Sequences of instructions that execute serially are called a basic block. Every time an `if` statement is encountered (or other branching construct, like `switch` in C and `cond` in Lisp), the current basic block ends, as control could transfer in one of several directions, depending on the runtime value of a variable.

Dependency graphs track the relationships between variables:

```
a = 1
b = 2                a      b
c = a + b            | \  /  \
d = a + c            |  c     e
e = b + 2             \ |
f = 2                   d
return e
```

In this example, dependency flow in the graph on the right flows down (`e` depends on `b`, not the other way around). Once you have a dependency graph, you can do analyses such as dead code elimination. The function in the example above returns the value `e`. Because `e` depends only on `b`, all of the lines calculating `a`, `c`, `d` and `f` can be removed without changing the behavior of the function.

2.2.1 Intermediate Representation

The intermediate representation is the compiler's internal representation of program structure. In Lisp, this has a 1:1 correspondence with the text of the source-code. In most other languages, the internal representation is often known as an Abstract Syntax Tree (AST). An AST is created by parsing the program source:

```
int foo(int a) {
    int bar;
    int baz = 2;
    for (bar = 0; bar < a; bar++) {
        baz = baz * a;
    }
    return baz;
}
```

The AST for the above function would, at its root, have a single `FUNCTION` node. This node has several attributes, like name (`''foo''`), return-type (`int`), args (`int a`), and a block. The block is a list of statements that make up the body of the function, and each statement might be a node that has children (like the `for` loop above).

2.3 Recursion

Recursion was considered controversial, and was snuck into Algol 60 by Dijkstra. It was a very natural concept for math oriented people, but less so for others.

2.3.1 Peano arithmetic

Peano arithmetic is a recursive definition of all math. With two primitives, `0` and `plusone`, you can define all the positive integers, addition, subtraction, multiplication and division. The example for the first few integers is:

```
0 - 0
1 - (plusone 0)
2 - (plusone (plusone 0))
```

2.4 Scope

Scoping refers to how variable references are resolved. There are 2 types of scoping - dynamic and lexical. Lexical scoping is what we are used to today, and what was included in Algol 60 (along with Fortran). Lisp screwed up and created dynamic scope. In dynamic scoping, the existence or non-existence of a variable in a function depends on where it was called from:

```
void f(...) {
    ...
    a;
    ...
}
```

In the example above, in lexical scoping (like used in Java), `a` must be declared either in the function body, as a function parameter, or in the containing scope (class or file). With dynamic scope, `a` would be looked up at runtime, and may or may not exist depending on where the function was called from.

2.5 Typing

Lisp is a language that doesn't feature static types, similar to Python and JavaScript. This is known as dynamic typing. Static types associate types with variables, and allow compile-time type checking. In contrast - dynamic typing associates types with *values*, not variables. One way to implement this is known as boxing - it associates a type tag with each value. If you have dynamic typing, type checking is undecidable (essentially the halting problem).

2.6 Garbage Collection

Lisp invented Garbage Collection. Reclaiming memory is not part of the λ calculus, and would have added significant clutter to lisp programs. The goal of garbage collection is to reclaim memory that was allocated but is (provably) no longer in use. If nobody has a pointer to a variable, than the space for that variable can be reclaimed. The roots of a program consist of the global variables, and any variables referenced from the stack or registers of all threads of execution.

2.6.1 Mark/Sweep

The mark/sweep algorithm is a basic method of garbage collection, and still widely implemented today. It relies on the idea of being able to associate a shade (white or black) with each variable - this typically requires a bit per word. The algorithm for Lisp is as follows:

- Set everything to 0/white.
- Start from the roots
  - Set each variable to 1/black
  - Recurse, following each var's `car` and `cdr`
- At this point, everything still marked 0/white is safe to reclaim

This is a sequential, whole heap algorithm, and has the property that it "stops the world". This means that while it is running, application code (also known as "the mutator") is not running. There are other classes of GC algorithms, such as parallel GC, concurrent GC, and incremental GC, but these are almost impossible to implement correctly. Eliot Moss invented a parallel, concurrent GC for Java, the MOS collector, but Sun couldn't get the implementation right and switched away from it.

2.6.2 Reference Counting

Reference counting (refcounting) is an alternative to garbage collection that amortizes the cost of reclamation, with some significant drawbacks. The CPython implementation is a major system that uses refcounting, as does Ruby and Objective-C. By keeping track of the number of references to an object as a counter on that object itself, the object can be freed immediately when the last reference is removed. This requires modifying this reference count each time the variable is stored or goes out of scope.

A major problem with refcounting is cycles. If we have 2 variables, A and B:

```
A <---> B
```

their reference counts will be non-zero even if they are unreachable from the rest of the program, which constitutes a memory leak.

2.7 The Stack and the Heap

Language implementations, like C, make a distinction between 2 classes of dynamically allocated memory. The heap is a large region where objects whose lifetimes are unknown are allocated from. This is conceptually simple, but there is significant overhead in this management.

The stack, on the other hand, is very fast. Objects are allocated on the stack by simply manipulating a pointer (the frame pointer), and this pointer is often restored automatically by the processor when leaving a function.