Lecture 18 — October 18, 2010

Patrick Lam *version 2*

Interpreting Bytecodes, a.k.a. Virtual Machines

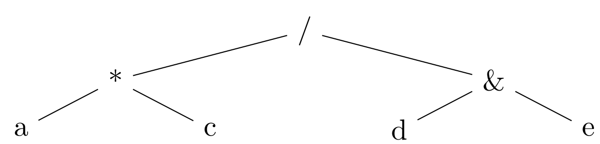
Bytecode interpretation trades off compiler complexity for interpreter complexity: the compiler processes the program at greater length, thus enabling a much more direct virtual machine implementation. Later on, we'll talk about just-in-time compilers, where the interpreter actually contains a native code compilation engine. JIT engines, though, are optional; you can always directly interpret the code by simulating the instruction set architecture of the target machine.

What are some examples of bytecode interpreters you know about?

Three-Address Code vs. Stack Code. Most bytecode interpreters are stack-based (like old RPN calculators[1].) Stack code is compact, which is good for transporting code across the network. Internal compiler intermediate representations are usually three-address code, where each statement contains at most one operation and, hence, two operands (in most cases).

Interpreter inputs and state. Implementing a virtual machine is implementing a processor in software. Examples of virtual machine instructions are loads, stores, and arithmetic instructions; note that by this point, we've compiled the AST's expression trees into lower-level instructions.

AST



Stack Code

```
push a
push c
mul
push d
push e
and
div
```

Three-Address Code

```
$t1 = a * c
$t2 = d & e
r = $t1 / $t2
```

(There are more instructions in the stack code, but they're shorter to encode.)

The Java virtual machine also includes object-oriented instructions, like virtual invoke instructions, implementing dynamic dispatch, and `instanceof` instructions, implementing dynamic type tests.

[1] Reverse Polish Notation is where you enter the operands and then the operation, e.g. `2 3 +`. You can get RPN by doing a postfix traversal of the expression tree. People using RPN calculators are far faster at keying in computations than with the usual infix-notation, and they make fewer mistakes.

Let's compare VM state to AST interpreter state:

AST	VM
locals	registers or stack
heap	heap
PC and implicit call stack	PC and explicit call stack

Unlike the interpreter's implicit call stack, a virtual machine will often maintain an explicit data structure, which keeps return addresses and function parameters. Registers would include:

- general purpose registers;
- a stack pointer `sp` pointing to the top of the stack;
- a frame pointer `fp` pointing to the current stack frame (where the local variables live); and
- a program counter `pc` pointing to the current instruction.

The machine state also includes condition codes, or flags (e.g. bits indicating the results of comparisons).

To execute an instruction, the interpreter:

- reads and decodes the instruction at the program counter;
- executes the instruction, changing the state of the machine (registers, stack, condition codes);
- updates the `pc` by incrementing it; jumping to a new address; or pushing the current address on the stack and jumping to a new address (function call).

Java VM code might look like this:

```
pc = code.start;
while(true)
  {  npc = pc + instruction_length(code[pc]);
     switch (opcode(code[pc]))
       {  case ILOAD_1: push(local[1]);
                        break;
          case ILOAD:   push(local[code[pc+1]]);
                        break;
          case ISTORE:  t = pop();
                        local[code[pc+1]] = t;
                        break;
          case IADD:    t1 = pop();  t2 = pop();
                        push(t1 + t2);
                        break;
          case IFEQ:    t = pop();
                        if (t == 0) npc = code[pc+1];
                        break;
          ...
       }
     pc = npc;
  }
```

Quadratic formula example

Here is the Java code for the quadratic formula example seen in class today.

```
public class Quadratic {

    public static void main(String[] args) {
        double x = positiveRoot(1, 3, -4);
        System.out.println(x);
        double x3 = positiveRoot3AC(1, 3, -4);
        System.out.println(x3);
    }

    static double positiveRoot(int a, int b, int c) {
        return ((-1 * b) + Math.sqrt(b*b - 4*a*c)) / (2*a);
    }

    static double positiveRoot3AC(int a, int b, int c) {
        int negB = -1 * b;
        int b2 = b*b;
        int ac = a * c;
        int fourac = 4 * ac;
        int discriminant = b2 - fourac;
        double sqrt = Math.sqrt(discriminant);
        double numerator = negB + sqrt;
        int denominator = 2 * a;
        double x = numerator / denominator;
        return x;
    }
}
```

and the corresponding Java bytecode:

```
Compiled from "Quadratic.java"
public class Quadratic extends java.lang.Object{
public Quadratic();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return

public static void main(java.lang.String[]);
  Code:
   0: iconst_1
   1: iconst_3
   2: bipush -4
   4: invokestatic #2; //Method positiveRoot:(III)D
   7: dstore_1
   8: getstatic #3; //Field java/lang/System.out:Ljava/io/PrintStream;
   11: dload_1
   12: invokevirtual #4; //Method java/io/PrintStream.println:(D)V
   15: iconst_1
   16: iconst_3
   17: bipush -4
   19: invokestatic #5; //Method positiveRoot3AC:(III)D
   22: dstore_3
   23: getstatic #3; //Field java/lang/System.out:Ljava/io/PrintStream;
   26: dload_3
   27: invokevirtual #4; //Method java/io/PrintStream.println:(D)V
   30: return

static double positiveRoot(int, int, int);
  Code:
   0: iconst_m1           // push -1
```

```
   1: iload_1             // push local #1 (b)
   2: imul                // integer multiply: -b
   3: i2d                 // int to double
   4: iload_1             // push local #1 (b)
   5: iload_1             // push local #1 (b)
   6: imul                // integer multiply: b*b
   7: iconst_4            // push 4
   8: iload_0             // push local #0 (a)
   9: imul                // integer multiply: 4*a
   10: iload_2             // push local #2 (c)
   11: imul                // integer multiply: 4a*c
   12: isub                // integer subtract: b2 - 4ac
   13: i2d                 // integer to double
   14: invokestatic #6; //Method java/lang/Math.sqrt:(D)D
   17: dadd                // double add:  -b + sqrt
   18: iconst_2            // push 2
   19: iload_0             // push local #0 (a)
   20: imul                // integer multiply: 2*a
   21: i2d                 // integer to double
   22: ddiv                // double division: numerator / denominator
   23: dreturn             // return x

static double positiveRoot3AC(int, int, int);
  Code:
   0: iconst_m1
   1: iload_1
   2: imul
   3: istore_3
   4: iload_1
   5: iload_1
   6: imul
   7: istore 4
   9: iload_0
   10: iload_2
   11: imul
   12: istore 5
   14: iconst_4
   15: iload 5
   17: imul
   18: istore 6
   20: iload 4
   22: iload 6
   24: isub
   25: istore 7
   27: iload 7
   29: i2d
   30: invokestatic #6; //Method java/lang/Math.sqrt:(D)D
   33: dstore 8
   35: iload_3
   36: i2d
   37: dload 8
   39: dadd
   40: dstore 10
   42: iconst_2
   43: iload_0
   44: imul
   45: istore 12
   47: dload 10
   49: iload 12
   51: i2d
   52: ddiv
   53: dstore 13
   55: dload 13
   57: dreturn
}
```