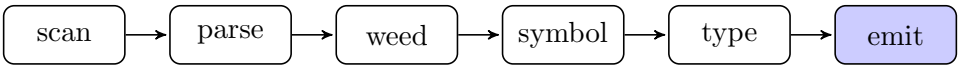
ECE251: Programming Languages & Translators Fall 2010

Lecture 17 — October 19, 2010

Patrick Lam *version 1*

Compilation: Interpreters, Virtual Machines, and Native Code

Recall again the general compiler picture:



We have seen scanning, parsing, weeding, and the symbol table. This week, we'll talk about emitting and executing code. We will defer type checking until after the midterm.

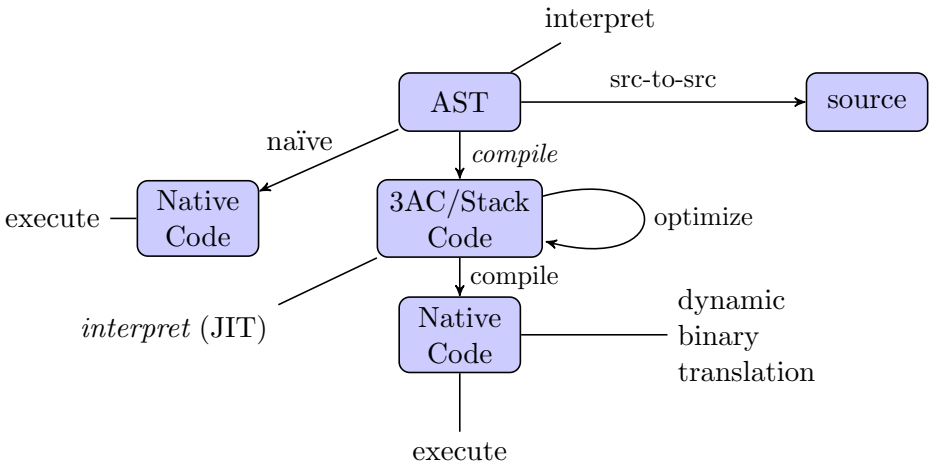
After this course, I expect you to be able to construct:

- a translator from ASTs to three-address code; and,
- a bytecode interpreter.

We will also survey:

- source-to-source transformers;
- naively generating native code;
- interpreting ASTs;
- just-in-time compilers and dynamic binary translation.

The below diagram shows how everything fits in.



Interpretation

We first revisit interpretation, which we surveyed briefly in Lecture 6 in preparation for Lab 1. Recall that an interpreter executes code by handling one statement or instruction at a time. Interpreters generally implement some sort of *virtual machine*.

What to interpret. The three main possibilities for interpreters are to get the code: 1) directly from parser actions (Lab 1); 2) from the abstract syntax tree; or 3) from bytecodes.

Interpreting directly from parser actions saves some implementation effort, but has severe disadvantages. In particular, you can't effectively handle loops, method calls or forward declarations, since the parser only makes one pass through the code. We focus on the other two code sources.

Interpreting an Abstract Syntax Tree

In this course, we've seen how to construct abstract syntax trees from the source code. One thing you can do with the AST is to interpret it. This tends to be less efficient than the alternatives, since the interpreter needs to do a lot of work at runtime to execute the program. Compilers typically do this work ahead-of-time.

Interpreter inputs and state. An AST interpreter has a high-level program representation—ASTs typically contain almost all of the program source. Such an interpreter therefore usually tracks source-level state. Namely, it would store:

- local variables (for the current method and its callers; in a symbol table);
- the heap (objects pointed-to by local variables); and
- the program counter (a pointer to an AST node) and call stack.

The call stack may be implicit in the interpreter's implementation.

We'd usually implement AST interpreters as tree walkers; see below for how. Executing a declaration (e.g. of classes or methods) simply adds a new name to the symbol table. The core of an AST interpreter (or any interpreter) is a big switch statement, which looks like this:

```
public void execute(Stmt n) {
  switch (n.type) { // maybe a Visitor pattern in practice
    case AssignStmt:
      AssignStmt as = (AssignStmt) n;
      Value v = eval(as.rhs, state); state.putVar(lhs, v);
      break;
    case BlockStmt:
      BlockStmt bs = (BlockStmt) n;
      for (Stmt b : bs.contents) {
          execute(b);
```

```
      }
      break;
    case IfStmt:
      IfStmt is = (IfStmt) n;
      if (eval(is.cond, state) == Eval.TRUE)
          execute(b.trueCase);
      else
          execute(b.falseCase);
      break;
    case ForStmt:
      ForStmt fs = (ForStmt) n;
      execute (fs.head);
      while (eval(fs.cond, state) == Eval.TRUE)
          execute(fs.body);
      break;
    case InvokeStmt:
      BlockStmt body = /* resolve target */;
      execute(body);
      break;
  }
}
```

Note that we execute the program by following its structure. Deviating from the structure, as for instance required by `goto` statements, is quite difficult.