ECE251: Programming Languages & Translators Fall 2010

Lecture 21 — November 9, 2010

Patrick Lam *version 1*

Let's look back at what we've seen so far and where we're going.

- scanning (NFAs, regular expressions);
- parsing (CFGs, recursive-descent, ANTLR) and ASTs;
- symbols: scoping; dynamic and static scoping; building a symbol table;
- execution: interpreters and virtual machine implementations; interpreting ASTs, bytecodes, and three-address codes; and
- backend issues: source to source transformations, code generation, register allocation, code optimization, runtime issues.

Remaining topics:

- wrapping up code generation today with JIT compilers and binary translation;
- types and type checking: you will be able to write a simple type checker;
- programming paradigms: functional, scripting, logic languages; Scala;
- domain-specific languages.

There are going to be exam questions on all of these topics, and the lab will include a type checker.

Code Generation

Let's reiterate what we saw already. Three main outcomes of code generation:

- source-to-source translation;
- stack-based code (e.g. Java Virtual Machine);
- native code, directly executable (generate assembler, then follow steps as in ECE222).

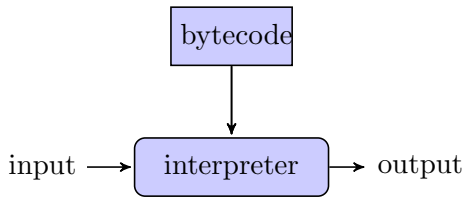
Steps in native code generation: given a parse tree,

- convert parse tree to an AST;
- convert AST to there-address code (see Lecture 19): flatten expressions, convert tree to control-flow graph;
- convert three-address code to assembler (see Figure 14.6 of textbook): allocate registers, stack locations for variables and static objects; also note object layouts from L21.

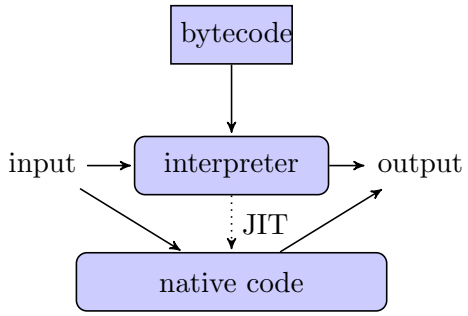
I haven't said much about register allocation; it's not that useful for you.

JIT Compilation

So far we've talked about compilers and interpreters as if they were totally separate. But that's not what happens in many systems today: Java, JavaScript, .NET. These systems have Just-In-Time (JIT) compilers. Recall that interpreters look like this:



A Just-In-Time compiler looks more like this:



Tradeoffs. Running native code is slow. But running the compiler takes time which you might better spend executing code. So you have to figure out which code is going to be executed a lot in the future. The best way of doing that is to figure out which code has executed a lot in the past.

Typical solution. The interpreter in this system would typically have counters; when, say, a method runs too often, the interpreter calls the JIT compiler to compile the bytecode into native code. When it runs even more often, the JIT might compile the native code into optimized native code. There are a lot of implementation details.

Advantages of JITs. One of the difficulties in program optimization is that you don't know anything about the inputs. At run-time, though, you have the inputs, and you can specialize the program given the inputs. Because of this, just-in-time compilation has the potential to produce surprisingly fast code: sometimes even faster than traditional compilers.

Dynamic Binary Translation

One application of just-in-time compilation technology is *dynamic binary translation*, where you JIT, say, x86 code to x86 code. Why? This permits you to instrument the code. For instance, Valgrind will make your life much better when writing C or C++ code, because it can spot memory errors for you.