ECE251: Programming Languages & Translators Fall 2010

Lecture 6 — September 23, 2010

Patrick Lam *version 1*

Interlude: Interpreters and Domain-Specific Languages

Before we discuss parsing, let's talk about interpreters and DSLs, which are both relevant to lab 1.

Interpreters

What are examples of interpreters?

Main characteristic. An interpreter reads a program as input and runs the program; contrast this to a traditional compiler, which produces a binary.

Implementing an interpreter is like implementing a compiler; however, you need to carry out the program's instructions rather than emit code. (Note that modern interpreters can translate to native code just-in-time.) As you visit the AST[1], execute the instructions.

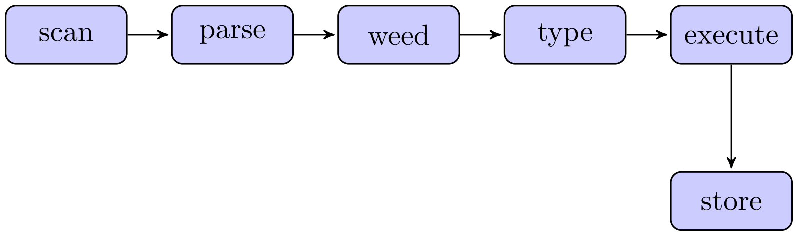


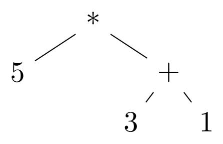
Figure 1: Parts of an interpreter.

Figure 1 presents one possible way to write an interpreter. Compare it to the parts of a compiler from Lecture 1. Note in particular the store, where the interpreter puts values of variables. We also will need scope information for these variables; we'll talk about that later. For now, use a `HashMap` or `Hashtable`.

Executing statements. The key action in an interpreter is executing the program. At an assignment statement `lhs = exp`, you need to evaluate the right-hand side and put the resulting value into the store, filed under `lhs`. (i.e. `store.put(lhs, eval(exp))`). You might need to read variables `v`; just query the store for them. (i.e. `store.get(v)`).

[1]You can also skip AST generation, limiting what you can execute, basically to straight-line code.

Evaluating expressions. You have an AST. You need a value. What to do? Walk the AST. We'll use arithmetic expressions as an example.



We start walking the AST at its root, *.

- eval(*, 5, 3+1): Need to eval 5 and 3+1, then multiply them.
  - eval(5): 5
  - eval(3+1): eval(3)+eval(1) = 4
- We then multiply $5 * 4 = 20$.

Writing Interpreter Code. One of the key concepts in this course is the separation of the program that you're compiling (which is data) from its inputs (which is also data). I'm going to present a partial AST hierarchy along with some classes implementing values.

The internal representation of the program is as an AST, which you could implement as follows:

```
interface Expr {
  Value eval();
}

class PlusExpr implements Expr {
  Expr left, right;

  public PlusExpr(Expr left, Expr right) {
      this.left = left; this.right = right;
  }

  public Value eval() {
    IntValue lv = (IntValue)left.eval(),
             rv = (IntValue)right.eval();
    return new IntValue(lv.intVal + rv.intVal);
  }
}

class IntLiteral implements Expr {
  String text;
  public Value eval() {
    return new IntValue(Integer.parseInt(text));
  }
}
```

This code manipulates `IntValue` objects, which we could implement as follows:

```
interface Value {}
class IntValue {
  public int intVal;
  public IntValue(int intVal) { this.intVal = intVal; }
}
```

Domain-Specific Languages: Internal vs. External

See `http://martinfowler.com/bliki/DomainSpecificLanguage.html` for lots of information of DSLs.

Recall that a DSL solves some specific problem, e.g. dealing with text, laying out graphics, performing scientific computations.

Usually the DSL isn't good at everything; you often need to combine it with a more general language. e.g. the program Asymptote draws graphs, but you usually want to combine these graphs with text from LaTeXor Word.

We can classify DSLs as internal or external.

Internal DSLs. These are essentially stylized libraries which make a host language do something interesting. You don't need parsers for such a language. The downside to these DSLs is that they need to conform to the host language.

Here is an example of a computer specification DSL, thanks to Martin Fowler:

```
computer()
  .processor()
    .cores(2).i386()
  .disk()
    .size(150)
  .disk()
    .size(75).speed(7200).sata()
  .end()
```

This code is written in the "fluent interface" style. An example of a real DSL written in this way is easyMock, which is a library to help write mock objects for testing Java programs.

```
expect(mockedDependency.getPriceBySku(SKU))
           .andReturn(new BigDecimal(100));
```

I'll leave it up to you to figure out how people implement such libraries.

External DSLs. We'll see how to build these DSLs in class; they are more heavyweight as they require parsers and other compiler infrastructure. They are usually stand-alone, but you probably will need to combine them with other tools to get useful results.