ECE251: Programming Languages & Translators Fall 2010

Lecture 9 — September 30, 2010

Patrick Lam *version 1*

Inheritance and Polymorphism

I didn't realize that no one had taught you about some fundamental object-oriented concepts. Computer Engineers need to know about these things, and it is only on your curriculum as part of ECE355 (Winter 2012), but you really need to know this earlier than that.

Java and C# are known as *object-oriented* languages. So, you create objects and call *methods* on these objects. For instance,

```
Value v = new AtomValue("5");
v.map(new Transformer(Expr.TransformerKind.PLUS, new AtomValue("2")));
```

Types. The basic idea here is that, in object-oriented languages, different objects may have different *types*. For instance, `v` is declared to have type `Value`. However, there are three different kinds of `Value` objects, namely `AtomValue`, `ListValue` and `TreeValue`.

In fact, `Value` is an *interface*. That is, it declares that all `Value` objects must implement certain methods, like `map` and `compare`. Classes may implement interfaces, and `AtomValue`, `ListValue` and `TreeValue` do implement the `Value` interface.

Inheritance. Although this isn't required in this assignment, you will eventually need to understand the concept of *inheritance*. We've seen that a class may implement an interface. It may also extend another class. For instance, we might declare a class for binary operators, which all share some methods, and a specific class for the addition operator.

```
interface Expr {
    public Value eval(Interp i);
}

class BinopExpr implements Expr {
    Expr left, right;
    public Value eval(Interp i) {
        throw OperationNotSupportedException();
    }
}

class PlusExpr extends BinopExpr {
    public Value eval(Interp i) {
```

```
            return left.eval(i) + right.eval(i);
        }
    }
```

(`BinopExpr` should actually be an *abstract* class.)

Polymorphism. We've seen that we have a variable `v` of "declared" type `Value`, which actually contains objects with "actual" type `AtomValue`, `ListValue` or `TreeValue`. This is known as *polymorphism*; variables may contain objects with any actual type conforming to the declared type.

You can put an `AtomValue` object into a `Value`, but not a `Value` into an `AtomValue`. Nor can you put a `ListValue` into a variable with declared type `AtomValue`.

Type casting. Sometimes you know that you have an `AtomValue`, but the declared type is actually `Value`. You might then want to cast. For instance,

```
if (v instanceof AtomValue) {
    AtomValue av = (AtomValue) v;
}
```

We will discuss this in greater detail in a month or two, when we talk about type checking. I think that this is enough to get you started on understanding classes for the assignment.

Ambiguity

While we put up the parse tree that you'd expect from the above expression, nothing prevents the parser from generating the following parse tree:

The expression grammar I put up before is *ambiguous*: it admits more than one valid parse tree. We don't want ambiguity, so we'll (later) see how we can modify grammars to remove ambiguity. Here is a disambiguated expression grammar.

$$
\begin{array}{lcl}
E & := & E \texttt{ '+' } T \mid E \texttt{ '-' } T \mid T \\
T & := & T \texttt{ '*' } F \mid T \texttt{ '/' } F \mid F \\
F & := & \texttt{'(' } E \texttt{ ')'} \mid \mathsf{LIT}
\end{array}
$$

Example: Building Parse Trees

Let's study an example of actually building a parse tree from a sequence of tokens. (Example 2.24 of the textbook is a second example.) Consider the *expression* nonterminal in the language defined here,

http://en.wikipedia.org/wiki/Recursive_descent_parser

Note that it is slightly different from the one declared above. We summarize the grammar as follows:

$$\begin{aligned} E &:= [\texttt{'+'} \mid \texttt{'-'}]\ T\ [(\texttt{'+'} \mid \texttt{'-'})\ T]^* \\ T &:= F\ [\texttt{'*'} \mid \texttt{'/'}\ F]^* \\ F &:= \texttt{'('}\ E\ \texttt{')'} \mid \mathsf{LIT} \end{aligned}$$

The following recursive descent parser parses this grammar:

```
void factor(void) {
    if (accept(lit)) {
        ;
    } else if (accept(lparen)) {
        expression();
        expect(rparen);
    } else {
        error("factor: syntax error");
        getsym();
    }
}

void term(void) {
    factor();
    while (sym == times || sym == slash) {
        getsym();
        factor();
    }
}

void expression(void) {
    if (sym == plus || sym == minus)
        getsym();
    term();
    while (sym == plus || sym == minus) {
        getsym();
        term();
    }
}
```

We can then construct a parse tree for

$$5 + 2/48 - 93,$$

by starting with the E production.