ECE251: Programming Languages & Translators Fall 2010

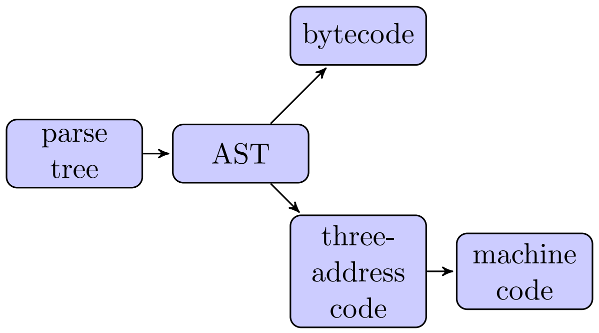
Lecture 14 — October 12, 2010

Patrick Lam *version 1*

Abstract Syntax Trees

I'm going to present some material on software design, specifically as it applies to compilers. The ECE elective ECE452 contains a lot more material on the more general topic of software design.

Abstract syntax trees are a high-level representation of an input program which closely resemble the source code; usually, it is possible to reconstruct the source code from the AST. As I've said before, we create the AST by dropping redundant information from the parse trees.



Compilers can use ASTs to reason about (and transform) the program structure, as it is provided by the programmers. In particular, most of the requirements that programming languages impose can be verified at the AST level.

What are some examples of requirements that Java imposes?

By the way, page 176 of the textbook describes most of what we did with ASTs in Lab 1.

ASTs for Object-Oriented languages. We've seen ASTs for expressions in Lab 1. Going to a full object-oriented language, what are the different classes in an AST?

What about something like PHP?

Visitor Pattern

The Visitor Pattern is one of the famous Design Patterns [GHJV95]. One way of thinking about it is as an object-oriented switch statement which centralizes the code for the different cases in one class. The `eval` method from Lab 1 was Visitor-inspired, although not strictly a Visitor. See also:

`http://sourcemaking.com/design_patterns/visitor`

Let's look at an example. First, add an `accept()` method to each of the classes you'd like to visit.

```
  interface ASTNode {
    public void accept(Visitor v);
  }
```

This means that you should be able to visit any class which properly implements `TreeNode`.

Any implementation of `accept()` must call `v.visit(this)`:

```
  class ExprNode {
    public void accept(Visitor v) {
      v.visit(this);
    }
  }
}
```

Now, you can visit the nodes using a `Visitor`.

```
  interface Visitor {
    public void visit(ExprNode n);
    public void visit(StmtNode n);
  }
```

A concrete implementation of this Visitor can, for instance, carry out type-checking, or can check that variables are defined before being used.

```
  class DefBeforeUseVisitor implements Visitor {
    List<Local> definedLocals = new LinkedList();
    public void visit(ExprNode n) {
      // check that all locals used are in definedLocals
    }
    public void visit(StmtNode n) {
      // add newly-defined locals to definedLocals
    }
  }
```

Note that the Visitor can store local state; in this case, it stores the list of defined variables[1].

Visitor variants can rewrite the tree by returning a new tree piecewise.

References

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[1]You actually need to do something more complicated in practice: variables must be defined on all paths.