ECE251: Programming Languages & Translators Fall 2010

Lecture 19 — October 21, 2010

Patrick Lam *version 1*

Other uses of ASTs

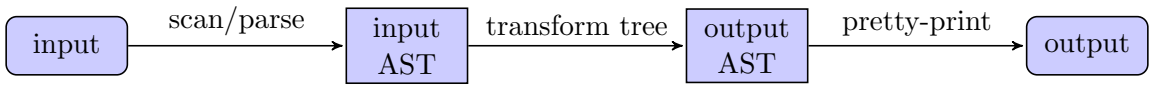
We've seen that one possible use of ASTs is for interpretation (although this is fairly slow). Two other uses of ASTs are for generating source code and three-address code or stack code. It is also possible to generate naïve native code from the AST, but we won't talk about that.

Source-to-source transformation.

Going back to source code might seem like a lot of effort for nothing; didn't we already start with source code? However, you might actually want to generate source for a number of reasons.

- You can generate source code in a different language; many early C++ compilers did this, and indeed, C is often used as a "portable assembly language". This way, you can write your compiler for your small domain-specific language and then execute it without incurring backend-related overhead.
- You can instrument the program to collect data about its executions. This can be for profiling and performance tuning or for verifying program properties at runtime.
- You can extract information from the program; `javadoc` is an example of a source-to-source compiler from Java to HTML. The `XMLVisitor` you're using in Lab 2 enables the use of XSLT on SQL programs, for instance.

I recommend that you examine the `XMLVisitor` code for an example of how to write a source-to-source transformer. In most cases, emitting source code from the AST is quite straightforward; you just need to write a bunch of `print` statements. If you want to translate to a different language, you want to have the two kinds of AST and then execute a tree transformation to go from one AST to the other. Then print the output AST.



Generating three-address code.

Last time, we mentioned three-address code. The two main steps in generating three-address code are: 1) flattening expressions; and 2) converting the syntax *tree* into a control-flow *graph*. To do this, the compiler traverses the AST and emits code into a list of three-address code instructions. It can usually emit code an instruction at a time.

Flattening expressions. We saw expression trees and three-address code last time. It is not hard to translate either way, but it is slightly easier to go from the expression tree to three-address code. Traverse the expression tree. At each expression with a subexpression as an operand:

- generate code for evaluating the subexpression, storing the result in a temp variable; then,
- replace the subexpression with the temporary variable in the initial expression.

Generating a control-flow graph. We also have to convert structured loops and branches into conditionals and jumps. For instance:

```
for (i = 0; i < 4; i++) {
  print(i);
}
```

```
    i = 0;
    $t0 = i < 4;
    $t1 = !$t0;
l0:
    if $t1 then goto l1;
    print(i);
    i = i + 1;
    goto l0;
l1:
    ...
```

We can use templates to do this transformation: when converting a `for` statement, the compiler would emit code for the head, then the condition, then the loop body, and include the appropriate `goto`s and labels.