CS245 Lab 5: An Object-Oriented AST

Due Wednesday of the last (14th) week of classes (7 Dec 2011); handed out the 13th Monday.

For this project, you will need to obtain the version of the translator from the lectures — check out the “OOP” project to get it. It contains a complete parser, a nearly-complete set of AST classes (Let_Node is missing), and a partial set of Translate functions (which do not currently have any mechanism for passing around AST data or translating Let nodes or Variable Use nodes). Your job is to complete the translator by implementing let; for the basic version, you need only implement the semantics of Scheme’s let*; for the advanced version, you should allow the semantics of let or letrec.

1. Create the class Let_Node, to represent the “let” operation; for now, just make the Translate function return the string “("Let node not translated", 0)” (you’ll need to use “ in the string to get the quotation marks into the string — see Dec_Node::Translate for an example of this).

2. Modify your let node to detect errors involving declaration of the same name more than once in a single let (you may need to add new methods to some of the classes). You should use exceptions to process this error (and any other errors you detect in this lab), rather than simply calling upon exit. You may either catch the exception in your main function, or (better) try to catch it at some other point that allows you to continue processing some of the program. However, your translator should never stop due to an uncaught exception.

3. Implement a better Translate function, to allow proper translations of Let Nodes and Variable Use Nodes.

   For the “basic credit” level of this lab, you should pick the semantics of either Scheme’s let*, let, or letrec (let* is probably the easiest). Give a note in a comment stating which semantics you use, and also provide a test suite that demonstrates that you have, in fact, provided the semantics you claim to provide. (This test suite can just be a set of a few comments, each of which shows an example input with the output that would be expected from let*, let, and letrec.)

   Note that you may, if you wish, change the parameters to the Translate functions, but you must make them all take the same set of parameters.

Submit what you have at this point — you have reached the “basic credit” level; before the end of the semester, do the “advanced” level (below) for either this lab or the Advanced Functional Programming lab.

4. Implement the semantics for all three flavors of let from Scheme. You should add two new keywords (e.g., let* and letrec, or other words of your choosing), so that all three versions can be mixed in one program (as in Scheme); you will need to have several classes, and should think about (and use) proper object-oriented design techniques to express them in C++.

   Once again, provide some examples showing that your system works as claimed.

If you wish, you may build this project in a language other than C++, as long as you create a set of AST node classes that are related via inheritance in the same way as the C++ classes we did in class. Let us know if you would like to do this.
Remember to submit your work via “Team- >Commit” when you’re done. You may also want to commit as you get each part working.

Note that the object-oriented paradigm, and related paradigms such as aspect-oriented programming (which we’ll discuss again in CMSC 350: Complier Design) raise many potential thesis topics in both language design (e.g., which paradigm is “better”, and in what ways?) and language implementation (e.g., how is dynamic dispatch implemented in various systems (such as CLOS and Cecil), and how does implementation of dispatch constrain other language features?).