444 Compilers — Project #3— Interpreter

Project #3: AST Interpreter

Introduction

The objective of this assignment is to be able to "run" the AST as a program -- that is, to build an AST Interpreter. To do this, we also need to know if the various user-defined identifiers are used correctly. We obtain the AST, itself, from the Parser.

To handle the identifiers, we will build a tree of nested scopes (SCT), where each scope will contain a local symbol table (AKA symtab) to hold that scope's local identifiers.

Once the SCT is built, we can then perform a treewalk of the AST to run the AST executable statements.

.

Team

The team may contain from two to four members, and may be different from the prior project team. Pick a three-letter name for the team as described for project #2.

Obtaining the AST

The AST can either be input from the output file produced by the Parser, or (slightly more risky) the Parser code can be included directly into the Interpreter so that AST is already in memory. The former is more modular, but requires the ability to correctly read the Parser output -- which itself must contain enough information. The latter is less modular, but not unreasonable as the AST is already in memory and ready to go -- except for correctly hooking up the Parser into the Interpreter code.

.

Scope Tree

The SCT can be built from a treewalk of the AST. Three AST node visitation events are important, as discussed in lecture: 1) a node that opens a **new scope**, 2) a node that has a new identifier definition (variable/slot, function name, or class name), and 3) a node that uses an identifier(variable/slot, function name, or class name). Note that opening a new scope happens on pre-order visit, and closing that scope happens on post-order visit.

The root SCT node will hold the file global identifiers, so this root should be opened during setup prior to walking the AST.

AST-Scope Linkage

An AST identifier declaration or definition node should create a new symtab entry for that identifier in the current scope node, and that symtab entry should link back to the AST declaration or definition nod. An AST identifier usage node should be linked to the SCT node containing its symtab.

AST Evaluation

After the AST treewalk to create the SCT, the AST can be interpreted (its statements executed, or evaluated) via a second treewalk. Each visited node will have its own evaluation function, because each will handle its kid nodes differently. Therefore, you will have a single treewalk function that, conditioned on the kind of node, will call that node's corresponding evaluation function. (You can use either an OO style Node 2-level hierarchy with up-cast or a switch statement on the kind of node, but remember Rule #0.) That AST-node-eval fcn might recursively call back to the treewalk function to handle its kid sub-trees when, and only when, it wants them evaluated.

The different statements, as AST subtrees, are fairly obvious in how they should be evaluated.

Note that when using a variable identifier as an Rvalue, you will want to get its value out of its box. This is why there is a link from the variable identifier's AST usage node to its symtab entry from which you can find its box location. Similarly, if the variable is used as an Lvalue, but then you just need its box location.

Variable Values and Boxes

Because the project's language doesn't allow recursive calls, a call frame stack is not needed. Each variable can have its box as a slot directly in its symtab entry in its SCT node, or provide a reference to box which might be hosted elsewhere in your interpreter's heap memory. Note that as we have several data types.

Expression "Operator" Nodes and Their Values

Each operator node needs to compute its value from its kids. (Note, statements are not expressions and, hence, have no values.)

Arithmetic and relational operators are fairly obvious in their operation. The sequence of expressions for the print “operator” might use the comma terminal symbol as an operator, if that symbol was hoisted.

Semantic Phase

After the SCT phase, but either prior to, or during, the AST interpreter phase, you can perform semantic type analysis phase (another treewalk). For type checking, you can (synthetically) propagating type information from leaf nodes to the operators and checking if a mom node operator (e.g., the assignment op) gets compatible types from all its kids.

Similarly, you can perform a variety of optimizations (another treewalk), such as constant propagation, and strength reduction. These are bonus features that should be postponed until the basic interpreter is working.

Technical Debt

We emphasize working S/W (Rule #0). However, getting to working S/W fast often leaves technical debt. Technical debt will rapidly turn into a Bad Smell if left too long to fester. Therefore, as this is the last rapid delivery project, the technical debt must be paid, approximately in full: the source code for this project should be reasonably clean and well-documented.

Academic Rules

Correctly and properly attribute all third party material and references, lest points be taken off.

Readme File

Same as for project #2.

Submission

Same as for project #2.

Grading

Same as for project #2.