USING VISUALIZATION TOOLS TO TEACH COMPILER DESIGN

Steven R. Vegdahl
Department of Electrical Engineering and Computer Science
University of Portland
5000 N. Willamette Blvd.
Portland, OR 97203
(503)943-7215
vegdahl@up.edu

1. ABSTRACT

A project-based compiler course presents several challenges to the student-implementor. In addition to the "book learning" about various compiler topics, a student must assimilate a large amount of information about the compiler's implementation. Furthermore he or she must be able understand each source-program construct at a number of different representation levels. Finally, the student must apply that knowledge during implementation and debugging of a compiler.

This paper describes a pair of packages that employ Java's graphical capabilities so that a program may be visualized at various stages of the compilation process. We argue that these tools are effective in helping students understand the transformation process from source program to machine code. We summarize our experience in using these tools in the context of a project-based compiler course. We also discuss other features of Java that make it well-suited for a student compiler project.

1.1 Keywords

Visualization, compiler, Java, abstract syntax tree.

2. INTRODUCTION

Students benefit in many ways from a project-based compiler course in their computer science curriculum. They:

- have the opportunity to integrate and apply knowledge from a variety areas of computer science, including automata theory, architecture, data structures and algorithms.

- exercise knowledge of a programming language on a "real world" project that involves a non-trivial amount of software engineering.
- are exposed to program-generation tools (e.g., a parser generator).
- exercise abstract thinking skills, as single conceptual object--the program--is transformed through several representations.
- gain confidence in understanding the compilation process, as they successfully implement a compiler.

Along with the above advantages, however, come difficulties. These include:

- Implementing anything close to a real programming language is difficult to do in a single term.
- Debugging must be done at two levels: on the compiler itself and on the generated program. Because compilers (and the programs they generate) generally make heavy use of address manipulation and pointer indirection (e.g., abstract syntax tree, flow graph, return address), illegal memory accesses are very common. This can further complicate debugging if the underlying implementation does not catch such accesses cleanly.
- If the programs being compiled are large, storage reclamation during compilation can be an issue, especially if significant optimization is performed.
- Visualizing a program across several representation levels (e.g., program text, abstract syntax tree, intermediate code, machine code) can be a challenging intellectual task.
- Getting even a "working knowledge" of the variants of a compiler's many data structures (e.g., "plus" node, "array-access" node) can be time-consuming.

Most students enter an introductory compiler class with only a general understanding of how a compiler works. Many have a fair amount of apprehension because compilers seem like such a mystery; they may have also heard from students in prior years that the compiler course is a difficult one.

A single academic term is not a huge amount of time for a student to digest the material and complete the implementation of a compiler for even a simple programming language. Tools that can be used to speed up the student's understanding are therefore attractive.

Recently, we had the opportunity to teach two introductory compiler courses, one each at the undergraduate and graduate level. The undergraduate course was a normal, full-semester course; the graduate course was a compressed, seven-week summer course. These two courses were our first experience using Java [2] as the implementation language. We chose Java for several reasons:

- A promising Java-specific compiler textbook [1] was available, along with accompanying scanner- and parser-generator tools, and "starter project".
- It is an object-oriented language, which we believe is well-matched to the compilation task.

- Previous experience grading student-compilers in type-unsafe languages--and their resulting core-dumps--made a type-safe language such as Java attractive.
- We wanted to experiment with using visual tools in teaching compilers that went beyond the animation of parsing algorithms [5,6,9]. Java's graphical features are well-suited to this.

We begin this paper by describing the visualization tools that we developed, and by discussing how they were used by our students. We then discuss other features of Java that would seem to make it particularly suitable for use as the implementation language for an introductory compiler course. Finally, we compare our work to that of others, and discuss directions we envision for future research.

3. TREE-VIEWER

Java's standard library [3] contains, among other things, a rich set of graphical and user-interface classes. This gives the potential to provide visualization tools to help the student understand the compilation process.

Experienced compiler writers routinely think about a program's many representations during the compilation process. As one gains experience with a compiler, one mentally moves back and forth among these representations with relatively little effort.

A student in an introductory compiler class, however, is not used to thinking about programs in this way. During the course of a single academic term, it is critical to develop an understanding of these representations--and the relationships among them--during the process of writing code to traverse and transform them.

The first project-task in our compiler course uses parser- and scanner-generation tools to build an abstract syntax tree (AST). Definitions for all the AST classes are supplied to the student. (These definitions are based on those of Appel [1], with some modifications.) Because the AST is traversed/transformed in subsequent tasks, it is important that students have a firm understanding of the various AST nodes and how they relate to the original source program.

To help the students gain intuition (and comfort) with the AST, we created a tree-viewing library. By invoking the compiler with the appropriate command-line switch, students can view the AST created by the instructor's (allegedly correct) version of the compiler or the one produced by their compiler. The visual representation of the tree appears in a Java window. Students can interact with the AST in the following ways:

- Use scroll bars to move between various parts of the AST.
- "Visually prune" the tree. Clicking on a node causes all of its subnodes to be hidden; clicking again causes them to reappear. This allows a student to see, for example, top-level declarations in single windows.
- Dragging a box around a node causes all nodes that are linked to it and all nodes that link to it to be highlighted. For example, if you drag a box around a variable definition, nodes corresponding to all the *uses* of that variable in the program will be highlighted. During

debugging of the name-binding phase of the compiler, a student who has not properly filled in links between *definitions* and *uses* of a variable will not see all the appropriate nodes highlighted.

Figure 1 shows the AST for the following Tiger [1] program, which defines a recursive factorial function:

```
let
  function fact(n:int):int =
    if n <= 0 then 1
 else n * fact(n-1)
  var val := 8
in
  printint(fact(val))
end
```

Figure 2 shows the same AST, but with some of the subtrees hidden.

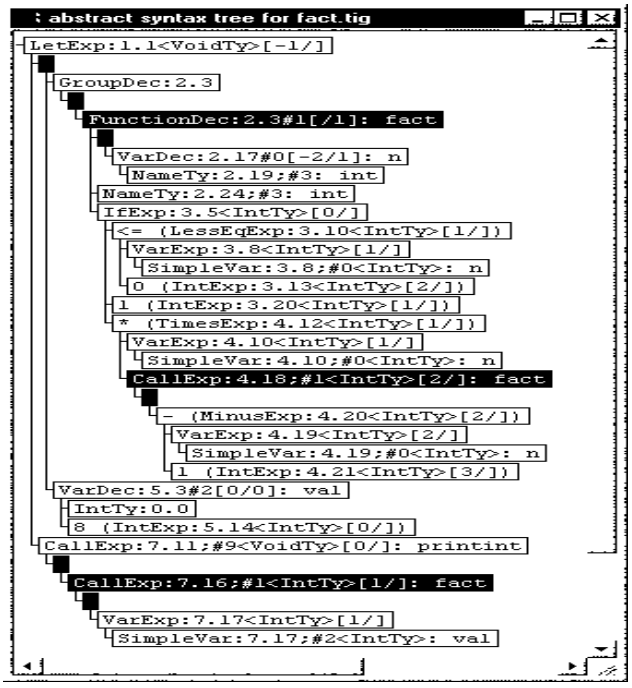

Figure 1: AST for fact.tig displayed in a window

The code for viewing and interacting with the abstract syntax tree actually knows nothing about the AST. Rather, it is a generic tree-viewing package that uses Java *interfaces* to allow any class with a hierarchical structure to be viewed. To satisfy the TreeDisplayable interface, a class must implement a protocol that lets the TreeDisplay package know how the objects are related to one another:

- Who are my children?
- Which of my children are (conceptually) lists, rather than single objects?
- What textual information do I display inside my node?
- What links do I have to objects that are not my children (e.g., a link from a *use* of an identifier to its *definition*)?

Any class that implements the TreeDisplayable interface can be displayed using the TreeDisplay package. For the AST, this means defining a handful of methods in our two root-classes (Absyn and AbsynList), and using inheritance to override the definitions in appropriate places in the

class hierarchy--a straightforward task. We supply the students with appropriate method definitions so that they do not have to worry about display-related issues.

Students use the tree-display window for two purposes:

- They run the instructor's version of the compiler to gain intuition about the relationship between the AST and the original source program by "playing with the tree".
- They use it to debug the various phases of their own compiler. When they run their own compiler, they examine the tree to see if the correct tree has been built, and if it has been properly annotated.

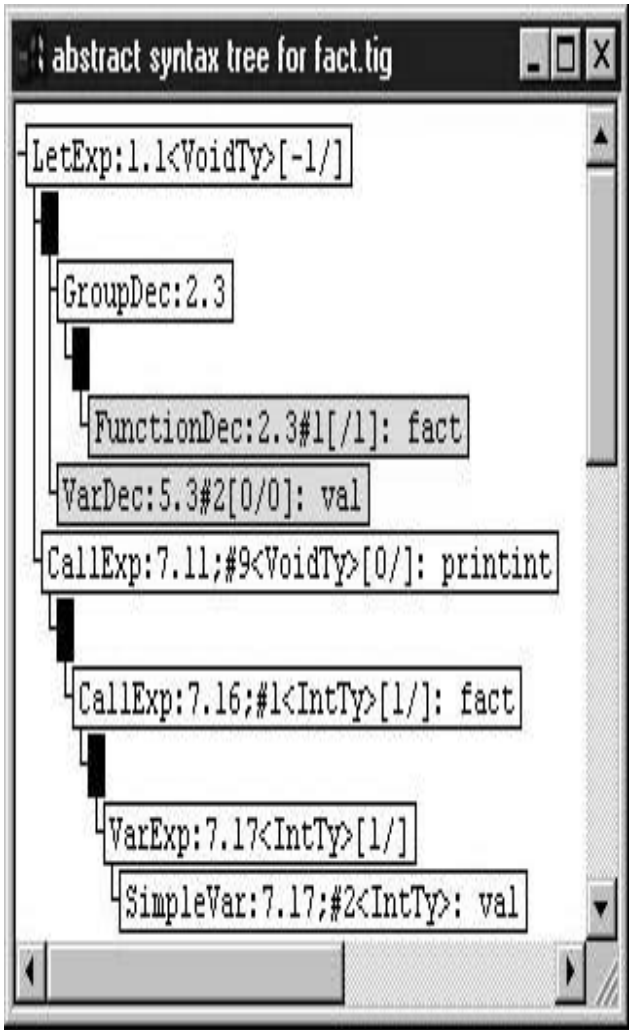

Figure 2: AST for fact.tig displayed, with some subtrees suppressed

The information displayed in the AST depends on which phase being debugged. When the AST is first created, an unannotated AST is shown, in which most nodes have no information other than their node's class (e.g., "function-call expression"). As compilation progresses, the information displayed about a node increases. For example, after type-checking each node will display its (Tiger) type (e.g., "int"). Similarly, the code-generation phase causes each node that represents a variable to display the machine location where it resides.

Our students have used the AST-display feature heavily. It was not uncommon to walk into the computer lab and see most of the monitors displaying one or more ASTs. Student comments about the tree-viewer ranged from "very helpful" to "absolutely vital" in understanding and debugging their programs.

4. ANNOTATING DEBUGGER

Central to understanding the compilation process is the understanding of how the execution of a program on the hardware relates to the original program. Depending on the constructs in the language, some of these may be difficult for a student to pick up. Compiler writers define *invariants* (e.g., regarding stack layout, function-call conventions, schemes for

handling uplevel variable references) that all functions are expected to enforce, and that all functions may assume. It is not enough for a student understand these invariants individually. Understanding the execution of compiler-generated code requires that one gain intuition about how the invariants work together.

To aid students in understanding the execution of a compiled program on a machine, we implemented an *annotating debugger*, which allows memory locations in the machine to be dynamically labeled as the program executes. This was done by modifying the assembler so that an instruction could be optionally followed by an *annotation directive*--a string that describes the data that the instruction computed. For example, the instruction:

```
mul 10
```

on our target architecture (which multiplies the contents of the top of the stack by 10), could be annotated with the string 'xyz' by writing:

```
mul 10 'xyz'
```

During the execution of the debugger, any time an annotated instruction is executed, the destination of the instruction is labeled with the instruction's annotation. Since our target-machine is a stack architecture, so all annotations are to locations on the stack. In the above example, whenever the annotated multiplication instruction, the location containing the result is labeled with the string 'xyz'.

We extended our version of the compiler so that it emitted annotation directives for variables, parameters, return addresses, access links and return-results. Students could then compile a program using our (allegedly correct) version of the compiler and run it using the debugger. As they stepped through the program they would, of course, see the program?s state change. In addition, locations in memory corresponding to variables, parameters, return addresses, or access links would be labeled.

Running the annotating debugger gave students better intuition about the relationship between the source code and the execution on the actual machine. Relevant locations on the stack were marked, so that stack-frame boundaries and the locations of program variables were evident. Because each assembly-language instruction was commented with the AST node and source code location to which it corresponds, students could also relate the assembly-language execution back to the source code or AST node from which it was generated.

The annotating debugger is laid out as a set of buttons to control execution (run, single-step, etc.), together with three panes that show program state. The left pane is used to display objects that are on the heap. The middle pane shows the contents of the stack. The right pane shows the executable code; each instruction contains a check-box that contains a breakpoint-toggle. All three panes are scrollable.

Figure 3 shows the annotating debugger running the recursive factorial program listed in Section 3. It shows the state of execution just after the call fact(3) has completed. The stack-frames are delimited by the return-address ("RA for...") and access-link ("AL for...") pair.

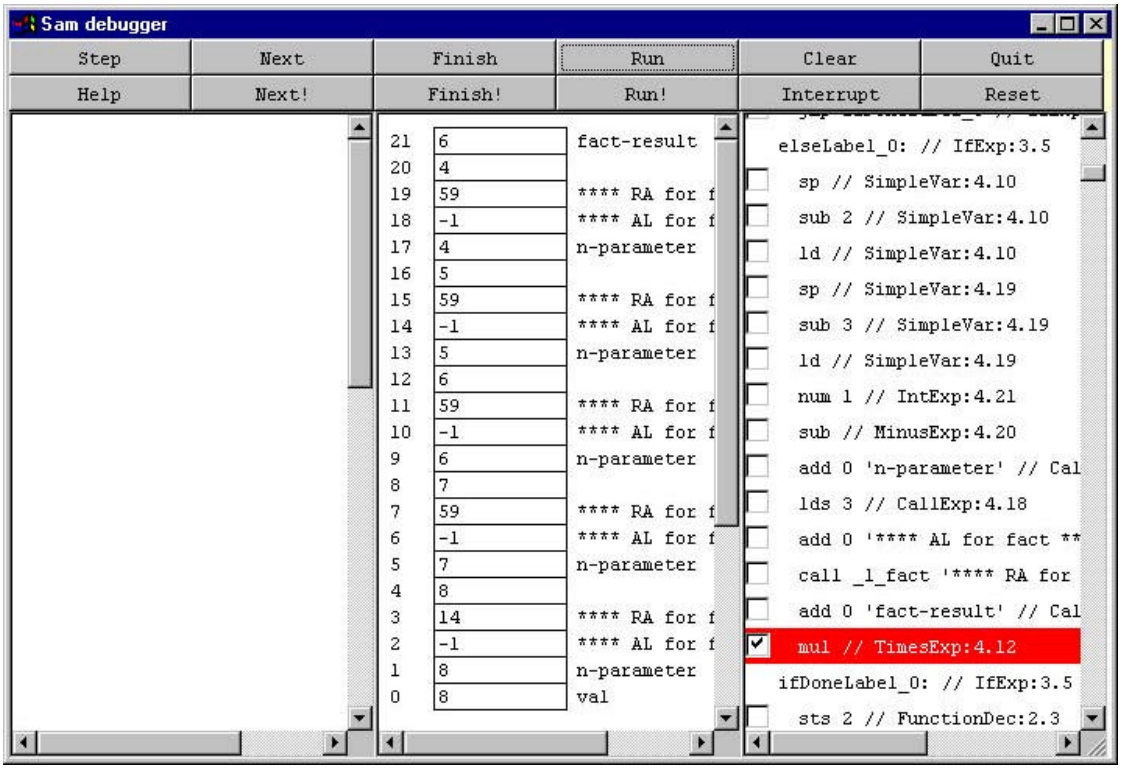

Figure 3

Parameters and variables are also labeled. (The left pane is not used because no heap allocation is done in this program.)

We queried our students after the course about their use of the annotating debugger. Many ran the debugger on code produced by our version of the compiler, and found it to be very helpful in gaining intuition about the how the executable code related to the source program. Some students also found it useful during the debugging of their compilers. This was particularly true for students who extended their compiler by adding optimizations. One student recalled four bugs that he does not believe he would have found had the annotating debugger not been available. Many students, however, admitted that their primary method of debugging their compiler back-ends was by "diff"-ing the assembly language output of their compiler with that of ours.

5. OTHER JAVA FEATURES

The Java programming language has a number of other features that we believe make it well-suited for use in a project-based compiler course. This section elaborates on some of these.

5.1 Object-Oriented Design

Java is very much an object-oriented programming language [2]. The first thing done when writing a Java program is to define a class. A Java program consists largely of applying operations to objects; inheritance pervades the language.

Neff [8] argues that object oriented approach is well-suited to the task of compiler writing. Our experience--both in industry and in academics--would confirm this. For example, an abstract syntax tree (AST) typically includes several dozen kinds of nodes (e.g., plus-expression, if-statement, type-declaration, integer literal). These can be naturally organized into a hierarchy of concrete and abstract classes.

Such a class-organization structure has the classic object-oriented advantage of allowing a particular behavior to be defined *once*, at the appropriate level of the AST hierarchy. For example, in Tiger [1], all binary *arithmetic* operators share the same type-checking rules, requiring both operands to be integer. The rules for binary *comparison* operators differ because some of them allow strings, arrays and record-objects as operands. It is therefore natural to specify type-checking behavior at the abstract level of *arithmetic* and *comparison* operators. During code generation, however, each concrete class has different behavior (emitting an "add" instruction rather than "sub", for example). It is therefore appropriate to specify code generation behavior in the concrete classes. Finally, the *tree-display* operation treats *all* binary operators identically: display the root; then recursively display the left and right subnodes. Tree-display behavior is therefore specified in the (abstract) "binary expression" class.

Another minor advantage of the object-oriented approach is that it makes it very easy to append a comment to each emitted instruction that tells what source-code construct produced it. In our project, each compiler-internal object contains information about the position in the source code to which it corresponds. All such objects inherit an "emit" operation--which is supplied to the students--that appends a source-position comment to each emitted instruction, as is shown in the right pane of Figure 3. This allows students--with no effort on their part--to know the mapping from each instruction to the source code construct from which it originated..

Although the intent of using an object-oriented design was to give students a better methodology for implementing their compiler projects, the reverse was also true: students learned a lot about object-oriented programming. The compiler project was among the largest examples of objected-oriented software that many of them had ever seen. The project afforded many opportunities to see--and to apply--their knowledge of object-oriented concepts such as inheritance, behavior-factoring, and protected class members. After the course, one student remarked, "Now I really understand the power of object-oriented programming."

Readers familiar with Appel's compiler text [1] will note that the implementation he presents is not object-oriented, but rather is procedural. Indeed, the design we give our students diverges from that of Appel when we begin semantic analysis. Our students--whose primary language is normally Java or C++--find this design to be quite natural. We would argue that the object-oriented approach is the most appropriate to use when Java is the implementation language.

5.2 Portability

The designers of Java tout portability as one of Java's greatest assets [2]. Our experience in that regard was generally positive. We were able to move fluidly between home and office environments (i.e., between Windows and Unix) without thinking about it.

Students were also able develop their compilers on a variety of environments, and were able to run their compilers, as well as the tree-viewer, simulator, and debugger. When grading student assignments, we were able to use Unix shell-scripts to compile and run the students' compilers without regard to the systems on which they were developed.

We encountered several relatively minor portability-related problems, however:

- An early version of our tree-viewer failed to use Java's FontMetrics class, and thereby made non-portable assumptions about font sizes. The result was that on some Java implementations, information about the AST nodes was printed in a font that was larger than the boxes that were supposed to enclose them.
- The format we chose for one of the auxiliary debugging-information files made references to character-positions in its own file. Our initial implementation did not account for the fact that "newline" is represented by a two-character sequence on some systems.
- One student FTP'd her programs using a version of FTP that truncated filenames. When we received these, we needed to run a shell-script that reconstructed the original filenames. Java would not otherwise compile these because it requires each public class to reside in a file whose name is the same as the class.

5.3 Garbage Collection

Many phases of the compilation process dynamically allocate and discard memory objects. When an implementation language does not support garbage collection, the student compiler-designer--or more often the student's instructor--is faced with the decision of whether to:

- Ignore any attempt at help-memory reclamation. This has the advantage of simplifying the project, but runs the risk of running out of memory when a large program is compiled.
- Attempt to reclaim unused memory. If done correctly, this can avoid the problem of exhausting memory due to leakage. However, it complicates the implementation, and runs the risk of premature deallocation and its consequent bugs.

While Java is certainly not the only language whose implementations support garbage collection, it is the first such language that we have used in a compiler course. In previous compiler courses (which used Pascal, C, or C++), we strongly discouraged students from even thinking about memory reclamation because we didn't want them distracted from the main task. This, of course, meant that large source programs might not compile.

With Java, storage reclamation is automatic. As a result, one student was able to write a fairly substantial database program in Tiger, and to compile it with no problems.

6. RELATED WORK

6.1 Parser Visualization

A number of efforts have been made in the area of visualizing parsers. Resler and Deaver [9] produced a system for visualizing LL(1) parsers that simultaneously highlights grammar productions and source code during parsing, but apparently does not display the abstract syntax tree as it is built.

The system of Khuri and Sugono[6] show animations of both an LL(1) and SLR(1) parser. Their system displays parse trees, *not* abstract syntax trees. It therefore appears to be useful as an aid in understanding LL- and LR-parsing algorithms, but does not give any intuition about how the parsing phase is connected with later phases of the compiler.

Kaplan and Shoup [5] have developed a Java-based tool for LALR(1) parser visualization that is an extension of the CUP parser generator [4]. In addition to showing the state of the parse stack and the reduction applied at each step, it allows the user to examine portions of the abstract syntax tree as the parse occurs. Their depiction of the AST is similar to ours.

6.2 Visualization of Later Compiler Phases

The system of Korn[7] is the more ambitious than those above in that it accommodates later phases of the compiler. His system is a general data-structure visualization system. Though it was not designed specifically for compiler visualization, Korn has used it to display the AST in two different graphical forms. It also allows the user to edit the AST graphically. He also reports that students have gained better intuition about the AST and compilation process. It is unclear to us from its description whether cross-tree links are handled. It appears that his system could be augmented to display annotation information.

7. FUTURE WORK

Understanding and implementing a compiler is a significant effort for most students. Visualization tools that allow a student to see and "touch" a program at many stages of the representation are, in our experience, very useful.Like Kaplan and Shoup [5], we envision a tool that allows the program to be visualized at each step of the compilation process.

Kaplan and Shoup have provided an implementation of the first compilation step--parsing. Our work would extend theirs by allowing the visualization of subsequent compilation steps. Because their system is "pluggable", it would appear to be straightforward replace their view with ours, thereby allowing further compiler phases to be handled.

The two visualization tools that we have implemented so far have been well-received by students, yet they give the student only a set of "snapshots" into the compiler?s transformation process. We would envision a suite of visualization tools that would both allow the student to

view (and interact with) the representations individually but, in addition, would allow the student to put the various representations side by side, so that their connection is evident. For example, the student might highlight the source code, and have the corresponding code in the intermediate-code and machine-code windows become highlighted. In the present implementation, this information is available only in that our "viewable" representations happen to be annotated with source-code-position information.

We would like to extend this further by integrating these visualizations with the program's execution. When the debugger is used to run the source program we would like to show the program executing at several stages of compilation, allowing the user to interact with (e.g., view the flow of execution, set breakpoints) each of the level of program representation.

8. REFERENCES

[1] Appel, A. *Modern Compiler Implementation in Java*. Cambridge University Press (1998).

[2] Arnold, K. and Gosling, J. *The Java Programming Language*. Addison-Wesley (1997)

[3] Chan, P. and Lee, R. *The Java Class Libraries : Java.Applet, Java.Awt, Java.Beans (Vol 2)*. Addison-Wesley (1997).

[4] Hudson, S.E., Flannery, F., Ananian, C.S.,Wang, D., and Appel, A.W. *Cup Parser Generator for Java*. (March 1998)

[5] Kaplan, A. and Shoup, D. CUPV--A Visualization Tool for Generated Parsers. *SIGCSE Bulletin 32.1* (March 2000), 11-15. *Proc. SIGCSE Technical Symposium on Computer Science Education*.

[6] Khuri, S. and Sugono, Y. Animating Parsing Algorithms. SIGCSE Bulletin 30.1 (March 1998), 232-236. *Proc. SIGCSE Technical Symposium on Computer Science Education*.

[7] Korn, J.L., *Abstraction and Visualization in Graphical Debuggers*. Ph.D. Dissertation, Princeton University (November 1999).

[8] Neff, Norman. OO Design in Compiling an OO Language. SIGCSE *Bulletin 31.1* (March 1999), 326-330. *Proc. SIGCSE Technical Symposium on Computer Science Education*.

[9] Resler, R.D. and Deaver, D.M. VCOCO: A Visualisation Tool for Teaching Compilers. *SIGCSE Bulletin 30.3* (September 1998), 199-202. *Proc. Conference on the Integrating Technology into Computer Science Education*.