Semantic Parsing of Java I/O in Novice Programs for an Online Grading System

Abstract

Beginning programming students have access to sophisticated development tools that enable them to write syntactically correct code in a straightforward manner. However, code that compiles and runs can still execute poorly, or with unintended results. We present a tool, based on an open-source parser-generation product written in Java, that performs semantic analysis of novice Java code. Specifically, the present investigation concerns the semantics of Java output methods, particularly when they are enclosed within iterative structures in the language. The effort will be to guard against threats that such methods pose to system integrity and performance, intercepting them prior to runtime. The approach used here closely models the analysis a human reviewer would perform, given a printed copy of the code. The tool is an open-source product, like the parser generator, and is also written in Java. As such, it is written to be extensible. The tool will be integrated into a larger research project underway at Montclair State University which involves the development of an online grading system for students in beginning computer programming courses.

MONTCLAIR STATE UNIVERSITY

Semantic Parsing of Java I/O in Novice Programs for an Online Grading System

by

William Madden

A Master's Thesis Submitted to the Faculty of

Montclair State University

In Partial Fulfillment of the Requirements

For the Degree of

Master of Science

May 20, 2005

School Graduate School

Department Computer Science

Certified by:

_____
Dr. Jinan Jaber
Dean

_____
(date)

Thesis Committee:

_____
Dr. John Jenq
Thesis Sponsor

_____
Dr. Dorothy Deremer
Committee Member

_____
Dr. Edward Boyno
Committee Member

_____
Dr. Dorothy Deremer
Department Chair

SEMANTIC PARSING OF JAVA I/O IN NOVICE PROGRAMS

FOR AN ONLINE GRADING SYSTEM

by
WILLIAM MADDEN

A THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in
the Department of Computer Science in
the Graduate Program of
Montclair State University
May 2005

Acknowledgements

I would like to thank my wife, Carol Heinz, and my son, Kevin Madden, for their endless reserves of support and patient understanding while I worked late into too many evenings. I would like to thank my extended family, including my father, Ward Madden, my brother, Robert Madden, and my sister, Sara Madden and their respective families, for words of encouragement all along the way. Thanks especially go to Dr. John Jenq, my thesis sponsor who, all along, provided much-needed support and encouragement to see this project through to its conclusion, and who taught best by asking the right kinds of questions. My deep gratitude too, to Dr. Dorothy Deremer and Dr. Edward Boyno who, together with Dr. Jenq, served on the thesis committee for this work. Thanks to Dr. James Benham, the graduate student advisor, for suggesting that I consider doing original research in the first place and who provided kind words of encouragement throughout. A debt of gratitude goes out to the open-source community, and to Terence Parr in particular, for supporting the work that makes the antlr parser-generating program such a valuable tool. Thanks, too, go to my peers at Bergen Community College who were always curious to know how the research was coming along and who were always helpful with kind words and praise; and to the Bergen Community College administration who provided generous and much needed tuition assistance to make this possible. And, finally, thanks go to my fellow graduate and undergraduate students at Montclair, as well as my own students at Bergen Community College, who must find it amusing, to say the least, that I am both teacher and student. Humble thanks to all.

Table of Contents

Table of Figures

Chapter 1: Introduction

1.1: Overview

The study of languages is the province of linguistics. One of the notable researchers in the field, still active as of this writing, is Noam Chomsky, whose work has influenced not only the general field of linguistics itself, but has found fundamental application in computer science as well. His early description of a hierarchy of language grammars [4] has provided the basis for describing and formalizing most computer languages in use today. Most computer languages are described as implementations of context-free grammars (Type 2 in Chomsky's now-famous hierarchy). A widely-used tool for notating context-free grammars is Backus-Naur Form (BNF), developed in the late 1950s, immediately after Chomsky's seminal work. John Backus, who had already invented FORTRAN, the first widely-used high-level computer language, initially created what came to be known as BNF while working on the development of ALGOL, itself a very influential language. Peter Naur also contributed significantly to the development of BNF.

Inherent in the study of computer languages are notions such as *grammar*, *syntax*, and *semantics*. Loosely, *grammar* refers to the formal set of rules that results in the production of language elements and how they may relate to one another. *Syntax* refers to the sequencing of language elements to form language expressions. *Semantics* refers to the meaning of a language expression. Current development environments for computer languages provide complete implementations of the grammar of the language, a full set of tools for suggesting grammatical and syntactic elements to programmers, and fairly sophisticated tools for debugging grammatical and syntactic errors in programs

during the development process. All development environments also provide feedback to programmers during the compilation process, particularly with regard to syntactical errors. Thus programmers can fairly easily develop programs that compile successfully – that is, are grammatically and syntactically correct. However, the *semantic* structures they have created (that is, the *meaning* or intended purpose of the program) may still be found wanting. Most development environments provide only rudimentary feedback for runtime errors, and it is often the case that the error reported is not the error that caused the problem. This is often misleading to novice programmers who can spend a frustrating amount of time trying to apply the wrong corrections to a problem. Run-time errors are often the result of faults in the programmer's logic, or in the programmer's assumptions about possible run-time behaviors while developing the program.

A more sophisticated semantic analysis of source code begins with the ability to resolve code down to individual syntactic elements. Tools commonly used for this purpose are known as *lexical analyzers* (*lexers* for short, known more generally as language recognizers) and *parsers*. A *lexer* reads a stream of characters (source code in this case) and, acting as a finite state accepter, produces a set of *lexemes*, or acceptable strings of characters. These strings are then evaluated to produce a set of lexical tokens that can then be processed by a parser. *Parsers* are programs that, given a particular formal grammar (most often notated in BNF), analyze a stream of tokens and build a data structure, usually a tree or a graph that represents the grammatical relationships among the syntactic elements. This structure can become the basis for a semantic analysis of the code.

In practice, parsers are usually the result of parser generation programs. This is because a particular formal grammar is an arbitrary construct and often subject to change and elaboration. Each time the grammar changes, a new parser must be generated. Most so-called parsers today are actually parser generators and are dependent on accurate, current grammars. In the case of Java, which is still a relatively young language, the grammar is in a state of flux.

There are two basic approaches to parsing code: top-down and bottom-up [24]. Bottom-up parsers are typically used in the compilation process. They are characterized as LR parsers, meaning that source code is read from left to right and that syntactic elements are constructed or derived from the right-hand end of a particular input string. LR parsers are "Left-to-right, Right-hand derivation" parsers. Yacc, a common parser found on UNIX systems, is one example of an LR parser. Top-down (or LL) parsers also read input from left to right, but use left-hand derivation. The distinction is important because the two approaches produce syntactic elements in a different order from one another. This also has implications for parser performance and for the kinds of grammars for which the particular approach is suitable.

A particular type of LL parser is the *LL(k) recursive descent* parser. Such a parser "looks ahead" *k* tokens to make parsing decisions. The ability to examine tokens before processing them is necessary in practice because modern programming languages, such as Java, are not purely context-free. Recursive descent parsing is particularly suitable for the production of Abstract Syntax Trees, which are explained more fully below (section 3.2). The particular parser used in this investigation, antlr, is an *LL(k)* recursive descent parser.

1.2: Overall Project Description

This research is part of a larger project already underway at Montclair State University. An online system is being developed that will provide feedback to students in two introductory undergraduate programming courses, regarding their Java coding exercises. The purpose of this larger project is to create an expert system that analyzes the semantics of Java code and provides useful feedback to students as they develop their coding skills. The larger research effort is being directed by Dr. John Jenq, Computer Science Department, Montclair State University.

In developing expert systems, concerns arise regarding threats posed to system integrity by I/O operations, particularly as a result of poorly-written or even malicious code. For example, the author wrote a simple program that wrote "Hello, World" to the hard drive on a 2.4 GHz/Pentium 4 computer with 512 MB of RAM (a very typical student machine at the time of this writing). The only unusual feature of the program was that the write to disk occurred inside an infinite loop. In 6 seconds of execution the program created a 400 MB text file. The operating system attempted to buffer the file in memory; however, there was not enough available primary memory and it began using virtual memory. The result was so deleterious to performance that it was quicker to reboot the machine than to wait for it to stabilize itself.

1.3: Research Topic and Rationale

The present investigation concerns the semantics of Java output methods, particularly when they are enclosed within iterative structures in the language. The effort will be to guard against threats that such methods pose to system integrity and

performance, intercepting them prior to runtime, specifically to avoid scenarios such as the one described above.

An underlying assumption is that the code to be examined (referred to as novice code) has already been compiled and that the development environment has reported no compilation errors, that is, the code is assumed to be syntactically correct. Today, most programming students have access to sophisticated development environments that do a lot of the syntax-checking as the program is being written and, in fact, provide drop-down lists of suggested keywords, classes, methods and arguments as the student types out the code. For this reason, it is becoming a relatively straightforward process for novices to be able to produce syntactically correct code. However, much in the same way that correct spelling of all the words in an English sentence does not ensure a grammatically correct sentence and, in turn, a grammatically correct sentence does not ensure semantic correctness (an intended meaning), so too with source code. A novice can write correct code that will execute, but it may execute poorly or with unintended results. A semantic analysis of the code can yield important feedback to both students and instructors prior to run-time and forestall errors or potentially harmful run-time behaviors.

Chapter 2: Literature Review

2.1: Background and Related Work

The author identified four areas of related research in the literature: semantic analysis of Java, handling malicious code, intelligent tutoring systems, and common errors in novice code.

The broadest of these research areas, semantic analysis, finds wide application: compiler generation, software testing, software quality assurance, program analysis and verification, exception handling, class analysis, software engineering tool development, code optimization, and constraint-based program analysis, to name a few. Much of the research in semantic analysis focuses on the technique of performing static analysis of code, that is, analyzing the semantic structure of source code without executing it. That is the approach used here.

2.2: Literature Review

The literature regarding semantic analysis revealed several alternative approaches to the development of automated grading systems that may be useful for the development of the overall project. A number of projects describe visualization tools that help novices to more easily discern structural and semantic relationships within their code [2] [3] [14]; however, these do not provide specific feedback mechanisms concerning specific semantic issues in code. Huynh [10], Fabry [7] employ variations of a pattern-matching approach; that is, source code is compared to a predefined 'template' program to identify syntactic and/or semantic differences. This may be a very worthwhile approach, especially with first-semester programming projects. The approach used in this

investigation is more open-ended and does not rely on modeling ideal code solutions. A particularly interesting approach [17] [19], is to use the structured data capabilities of XML in the semantic parsing process. One reason for this is to be able to take advantage of the rich array of tools available for XML that provide powerful ways to analyze semantic structures.

Much of the interest in identifying and handling malicious code is fueled by the vigorous interest in security issues these days, although there is an offshoot of this research that deals specifically with the phenomenon of run-time termination on mobile devices. This research often describes particular tools and or techniques. Rabek [20] describes a tool that performs a static semantic analysis of code, identifying Win32 API system calls. The tool then monitors the system calls made at runtime, verifying that they conform to the calls determined by the semantic analysis. The first half of this approach (the semantic analysis) is similar in principle to the approach used here, though it is not language specific and it is expressly designed to work only on Windows platforms. The approach used here also does not do any runtime monitoring – all analysis and feedback is provided during the static analysis.

Research in intelligent tutoring systems or agents (the term *de jour*) extends back at least into the 1960s, with a host of approaches and techniques that have evolved and changed as technology has grown and access to computers and the internet has grown to near ubiquity, at least in industrialized countries. A spin-off of this research has delved into the teaching and learning of programming languages, particularly regarding automated tools that can help teachers evaluate code written by novice programmers. Six specifically address the teaching of Java [23] [22] [5] [6] [15] [8].

Sykes [22] describes a prototype for a system that will involve a small subset of the Java language. This may be an approach worth investigating, again, especially for first-semester student programming projects. It remains to be seen, once the prototype is expanded to include the whole language how closely it will resemble the author's own efforts. The present investigation seeks to include the whole language from the outset.

DePasquale [5] also employs an approach that works with a subset of the full language; it is interesting to note that he does not find significant differences in performance between students who begin by learning subsets of the language in simplified development environments compared to students who learn from the outset using the full language in complete development environments. He does note that students using the simplified environments were more satisfied with their experience than students in the traditional environment.

Truong [23] begins by identifying a number of common semantic errors made by novice programmers (this was done through his own extensive literature review and then confirmed empirically on his own campus). He employs software metrics and structural metrics to analyze and evaluate code semantics, based on a comparison of semantic parse trees of student code with model code. The code assignments are "fill-the-gap" in nature, i.e., a skeleton structure is supplied and the student must complete what are called "well-formed gaps" in the code so the result will compile correctly. This approach seems particularly well-suited for first semester students and might be worthy of investigation for use within the larger research project at the University.

Truong employs metrics that quantitatively measure how far a novice's code departs from predefined stylistic guidelines or from samples of model solution code. An

interesting side note here is that Badros [1] provides a very nice set of examples that implement these kinds of metrics using XML-wrapped java source code.

The approach that Truong and Badros employ, providing a 'template' to which the novice's efforts are compared, are more recent implementations of what appears to be a 'pattern-matching' approach that has been used most often historically. A number of earlier efforts were investigated, more to understand the evolution of the tools, than to gain insight into particular approaches. An approach used by Jackson [11] seems representative: a tool that uses yacc and lex to compare student code to a 'correct' solution. Huynh and Fabry, noted earlier, seem also to have refined and elaborated on this historical approach.

Kumar [15] has created a number of tutors for use with either Java or C++ that address specific topics in beginning programming (the specific reference given here deals with expression evaluation). The tutor produces snippets of code that students must then evaluate. In more extended cases, the evaluation is done interactively in a step-wise fashion, essentially, tracing the execution logic of the code.

Hristova [8] has developed a tool, named Expresso, that specifically concerns itself with providing pre-runtime feedback for a set of common Java programming errors. Part of her research, like Truong, produced a list of common syntax, semantic and logic errors in Java. It appears that most, if not all, of the errors listed are actually caught by most current compilers. Part of the reason for the development of her tool was to help students interpret often cryptic compiler error message that can often actually be misleading in terms of the necessary corrections.

Etheridge [6] provides a very interesting tool called CMeRun that pre-processes source code, providing a snap-shot of the condition of variables and various kinds of output as each line of code executes. It provides something akin to the step-through execution that most debuggers produce within current IDEs. It allows students to step through loops providing the opportunity to break out at each iteration.

There are also other tools that provide visualization tools and various other techniques for helping novice programmers better understand what they are doing. Kumar already provides a concise summary of these approaches, including Hristova and Etheridge in his summary.

Like Hristova and Etheridge, and in contrast to Kumar, Truong and Sykes, the author's approach is designed to work with non-prescribed projects. The present approach works with any Java code and does not expect the code to match any pre-existing or closed solutions. This approach seeks to determine runtime results prior to runtime and prevent syntactically correct but troublesome code from executing. In this regard, the concerns in the present investigation are somewhat closer to Reiss [21] although his interest is in developing CASE tools for large-scale enterprise projects. Reiss describes a process whereby semantic parsing is largely employed to maintain a symbol table and in which the symbol table is continuously updated during incremental parsing, effectively generating a model of runtime behavior. This will become an important feature in future extensions of the current investigation. Lapierre [16] also describes a commercial tool developed at Bell Labs Canada, designed for large-scale projects, that builds an Abstract Semantics Graph as its central tool for analyzing source code. Since it is proprietary, details regarding its implementation are not available.

Finally, there is a significant body of research that identifies common novice errors in syntax and logic. Truong and Hristova both investigated a number of common programming issues specifically related to Java, drawing upon their own experiences and generating surveys of their peers, as well as reviewing the literature. Their work will be very useful in determining future directions for this research. As far as the author knows, at this writing, there have not been any investigations into the semantic issues involved in generating I/O from within iterative structures in Java.

Moreover, the author's approach appears somewhat unique in that it attempts to model the behavior of a human reviewer. A human reviewer could, for instance, read the following code: *for (x=0; x<10; x *=1);* and can model its runtime behavior without actually compiling and running the code. After performing the virtual run, the human reviewer can predict runtime performance and provide feedback to the novice programmer, pointing out, in the example used here, that while the loop will compile and execute, it is infinite. That is essentially the approach used here. The author has previously reported preliminary findings at an earlier stage in this research [18].

Chapter 3: Methodology and Implementation

3.1: Methodology

There are a host of open-source tools and resources readily available that provide a foundation for doing semantic analysis of Java code. The present effort seeks to build an open-source tool, based on existing tools and resources, that can anticipate runtime errors, saving both students and instructors time and effort, and helping to maintain system integrity.

Complete semantic analysis in any language is extremely difficult and, ultimately, open to much interpretation and debate. Therefore, a number of important assumptions and constraints were imposed in the present case in order to define a meaningful scope for this research and to provide an extensible basis for future research:

(1) The tools described in this investigation work with Sun Microsystems' official Java 1.3.1 specification.

(2) The tools are designed to examine Java source code produced by students in beginning computer science courses. Such code will hereinafter be referred to as 'novice code' or 'novice programs'.

(3) It is assumed that the novice code under examination has first been successfully compiled, that is to say, the code is syntactically correct.

(4) For the purposes of this investigation, the tools perform a semantic analysis of novice code with the following constraints:

   a. All current Java (1.4.2) output classes and methods are included in the investigation. Input classes and methods are not part of the present investigation.

b. The tools perform semantic analysis of output classes and methods only when they occur within iterative structures (specifically, *for*, *while* and *do* loops). Invocations of output classes and methods that execute singly in-line are not investigated.

c. Only numeric loops are considered. That is, initial conditions for entrance into a loop, exit conditions from the loop and the loop iterator are assumed to be numeric expressions or assignments. Loop controls using other means (boolean expressions, character or string expressions, pointers and so on) may be the subject of future investigations.

d. Only loop control identifiers that have no dependencies are considered in the present investigation. Typically, it is assumed that loop control identifiers are declared and initial values assigned early in the code with no further modification before their use in the loop. More complex or subtle manipulation of loop control identifiers is not within the scope of the present investigation.

e. Only simple 'in-line' iterative structures are considered. Nested loops, embedded method calls, recursive structures, and so on, may be the subject of future investigations.

3.2: Implementation

This research began with an investigation of the Java language specification itself. A BNF grammar for version 1.3.1 was chosen because it was the most recent available for the language at the time the core of this research was conducted. As of March, 2005 there is now a BNF grammar available for the most recent version of the language

(version 1.4.2). It is anticipated that, at most, only minor changes will be needed to enable the author's tools to work with the newer grammar. In fact the author has used the 1.3.1 grammar with the complete set of 1.4.2 output classes and methods with no ill effect.

An examination of the official language specification on Sun Microsystems' web site [12] revealed that Java 1.4.2 contains 149 output classes. Because of polymorphism, these classes share 57 output methods. Java I/O classes are listed in Appendix A. Java output methods are listed in Appendix B. Appendix C lists all Java output classes, together with their associated methods.

Next, a language parsing engine was needed. A number of open-source products are available for this purpose: antlr, yacc, Bison, Semantic, lex, and flex among others. See Appendix G for links to more information concerning a number of these alternatives. There are also commercial products available that the author did not choose to investigate.

Antlr (ANother Tool for Language Recognition) [9] was chosen for the present investigation. It is the only parser that has a complete open-source implementation in Java. This was important since the author's own tools are also built using Java. It is also an extremely fully-featured product and is constantly growing and developing thanks to its large community of users and developers (approximately 45,000). It has also been ported to UNIX, Linux and Windows platforms. The antlr web site provides complete BNF grammars to a number of languages besides Java including C++, C, Ada, HTML, Python, Oracle SQL, and C#. The source code for antlr is currently written in Java, but there is a C++ version as well.

Antlr contains a parser generator; that is, it is implemented by first supplying a grammar in BNF notation. It is straightforward one-time process to build the actual parsing tools. Once this is done for any arbitrary language, antlr can parse source code in that language and produce a data structure known as an Abstract Syntax Tree (AST), to represent the source code. Antlr also contains a rich set of classes and methods for working with ASTs, token streams, tokens and lexemes.

The reader may wish to refer to *Figure 1* for the discussion in the next four paragraphs. It shows a simple Java program and the AST that it produces.

An Abstract Syntax Tree is a recursive tree structure and is the result of recursive descent LL(k) parsing. LL(k) parsing is particularly suitable for Java grammar which is not purely context-free. Recursive descent parsing is helpful because the resulting structure closely resembles the structure of the code that produced the AST.

Figure 1: An antlr-produced Abstract Syntax Tree (AST) and the source code that produced the tree.

An AST for a standalone complete Java program is structured as follows: The AST root has a FirstChild: the CLASS_DEF node. The CLASS-DEF node is composed of a FirstChild and four siblings: an internal MODIFIERS node, three leaf nodes (the Class name, an EXTENDS clause and an IMPLEMENTS clause) and an internal OBJBLOCK node. The last of these, the OBJBLOCK, contains at least one internal METHOD_DEF node (main) and could contain Class-level declarations and other METHOD_DEF nodes as well. The METHOD_DEF node contains four internal nodes: MODIFIERS, TYPE, PARAMETERS, and SLIST. The last of these, SLIST, finally contains the code body.

Any internal node has a FirstChild node at least. There may be other child nodes; however, since antlr is a recursive descent parser (described below) these are only accessible in a serial fashion as getNextSibling methods relative to the FirstChild node and are not directly accessible from the parent node. Many, but not all, internal nodes are binary; that is, they have a FirstChild and a NextSibling, or in more standard parlance, a LeftChild and a RightChild. However, there are a number of internal nodes (chiefly SLIST and ELIST nodes) that may have any number of child nodes (or, more precisely, a FirstChild node and several NextSibling nodes). A leaf node, by definition, has no child nodes, but may have NextSibling nodes. These NextSiblings may be either internal nodes or leaf nodes. There are no methods built into antlr that permit backtracking (except as the inevitable result of ascending out of a recursion), so the concept of a parent node is absent.

Antlr implements recursive descent parsing, which means, among other things, that 'walking' an AST involves traveling forward-only through source code in a recursive

depth-first manner. The algorithm uses a *getFirstChild* method to locate FirstChild nodes first. Then, if there are no more depth-first children, it uses a *getNextSibling* method to identify any sibling nodes, ascending out of the recursion along the way.

The general pseudocode for a recursive descent is shown in *Figure 2*.

```
1  boolean searchSubtree(AST tree, String searchItem) {
2    AST thisNode, child
3    boolean itemFound = false
4    thisNode = tree
5    while thisNode is not null {
6      child = thisNode.getFirstChild
7      if child is not null
8        itemFound = searchSubtree(child, searchItem)
9      if thisNode contains searchItem
10       itemFound = true
11     thisNode = thisNode.getNextSibling
12   }
13   return itemFound;
14 }
```

Figure 2

Line 8 contains the recursive statement and lines 6-8, contained in the while loop, illustrate the depth-first nature of the recursion. Line 11, also contained in the while loop, illustrates the serial nature of the 'walk' among sibling nodes. To process the last of several siblings, each of the previous siblings must first be processed in order before moving to the last sibling.

EXPR nodes are of particular importance. As simple as the code behind the example shown in Figure 1 is, its AST contains 5 EXPR nodes, used variously to parse a simple assignment statement, to express the exit condition for a while loop, to hold a println method call, to hold a literal string, and to contain the iterator expression. Because EXPR nodes have such varied uses, their internal structure (and nesting)

requires careful attention. One phase of this research addressed the development of Java code to evaluate EXPR nodes.

The basic technique in this investigation was to develop a software tool, called *IOScan*, that calls antlr methods to build an AST, walks the tree searching for iterative structures such as *for*, *while* and *do*, and then locates any Java output methods possibly executing within those structures. Any loops containing output methods are examined to see whether or not they are well-formed. In this context, an expression such as: *for (x=1; x<=10; x++)*, is well-formed, whereas: *for (x=1; x<=10; x--)*, is mal-formed. For a complete list of iterative structures (both well-formed and mal-formed) tested in this investigation see Appendix E. The complete source listing for *IOScan.java* is available in Appendix F.

For performance reasons, the search for output methods is conducted by comparing nodes to a hashed list of Java's 57 output methods. Iterative keywords *for*, *while* and *do* were also hashed, though it is doubtful such an effort was really necessary, at least in regard to performance. (At one point the author had hypothesized the need for a larger list of keywords, but the scope of this research effort is restricted to just *for*, *while* and *do*).

The *IOScan* logic addresses conditions where loop-control identifiers are numeric and have no dependencies. In this case, *for* loops in novice code are generally very tractable. The parenthetical expression following the keyword *for* defines entry, exit and iterator expressions for the loop-control identifier and these are easily located in a recursive descent search. *While* and *do* loops are not so straightforward. The only structure immediately tied to the keywords *while* or *do* is the exit condition of the loop-

control identifier and, even then, the identifier's type is not known. It is necessary, because of the recursive descent nature of the parser, to do separate 'walks' through the AST to locate the declaration and initial condition of the loop control identifier, and the iterator expression.

A further problem to be solved involves the evaluation of loop-control identifier expressions. A great deal of time was spent investigating the development of a Runtime Identifier Model (RIM). The purpose of the RIM is twofold. It will provide:

(1) a symbol table which stores identifier names, along with their addresses, types and values (or expression addresses), and

(2) an expression evaluator which determines, through a modeling of run-time behavior, whether a particular identifier can be evaluated (especially if it is a loop-control identifier) and, if so, what it's value is at any given point in the code.

The author was able to develop symbol table structures easily enough, but developing an extensible expression evaluator proved to be very time consuming and needs to be the subject of future research. There are 163 possible tokenizable syntactic structures (including keywords, operators and symbols) that make up Java 1.4.2 (see Appendix D). A great many of these can find their way into EXPR nodes. An expression evaluator would need code for each token, to be able to evaluate it in context, essentially requiring something of the complexity of an interpreter for the entire language.

The development of a RIM, with a workable expression evaluator, is an important next step because such a tool could accommodate more subtle, complex and real-world uses of loop-control identifiers, particularly as novice programmers become more skilled.

Currently, *IOScan* examines the following kinds of expressions (in the examples supplied below *n* can be any valid numeric value, depending on identifier type).

Loop-control identifier initial condition:

Declarations of loop-control identifiers without assignment: *int x;*

Declarations of loop-control identifiers with assignment: *int x = n;*

Simple assignment statements following earlier declarations: *x = n;*

Loop-control identifier exit condition:

Standard boolean comparisons: *x < n; x > n; x = = n; x <= n; x >= n; x != n;*

Loop-control iterator. Expressions in any of the following forms:

*x++; x--; x += n; x -= n; x *= n; x /= n; x = x + n; x = x – n; x = x * n; x = x / n;*

Chapter 4: Results and Conclusions

4.1: Results

In its present state of development, *IOScan* can evaluate simple in-line (un-nested) *for*, *while* and *do* loops using numeric loop-control identifiers that have no dependencies. Despite its rudimentary scope, there are still quite a number of unusual (but entirely possible) novice programs that can be written, which will compile and run, but which are still mal-formed and will perform either unpredictably, poorly, not at all, or will loop indefinitely.

Given the restrictions on expressions noted above, the following kinds of loop behaviors are possible (all examples are illustrated using *for*-loops for brevity, but *while* and *do* loops can be written that will behave identically).

(1) Well-formed loops. These are loops where the relationships among the initial condition, the exit condition and the iterator are such that the loop will execute a finite and predictable number of times. Example: *for (x=1; x<10; x++)*

(2) Infinite loops. Examples: *for (x=1; x<10; x += 0)* and *for (x= -10; x<1; x /= 2)*

(3) Loops where the iterator heads in the wrong direction. The following example is actually an infinite loop because, in Sun's implementation of Java, int-types are 2s-complement values and, instead of going out of range, will increment or decrement 'around to the other side' of their permissible range. Example: *for (x=1; x<10; x--)*

(4) Loops where the exit condition is the inverse of what it should be. This occurs when the novice programmer confuses > and < symbols. Example: *for (x=1; x>10; x++)*

(5) Loops where the iterator converges to 0 from either side (positive or negative). This may be intentional in some cases. Examples: *for (x=1; x<10; x *=0.5)* probably not intentional; for *(x= -10; x< -1; x *= 0.5)* perhaps intentional.

(6) Loops where the iterator converges to 0 alternating between positive and negative values. This is likely not done intentionally. Example: *for (x=1; x<10; x *= -0.5)*

(7) Loops where the iterator alternately diverges toward negative and positive infinity with each iteration. This is likely unintended. Example: *for (x=1; x<10; x /= -0.5)*

(8) Loops that converge to 0 in 1 iteration. This is a special case and likely not an intentional effort. Example: *for (x=1; x<10; x *= 0)*

(9) Loops that produce an undefined condition. This is also a special case and likely not an intentional effort. Example: *for (x=1; x<10; x /= 0)*

(10) Loops that use != as the boolean operator in an exit condition. This may be intentional or otherwise. Example: *for (x=1; x != 10; x = x + 2)*

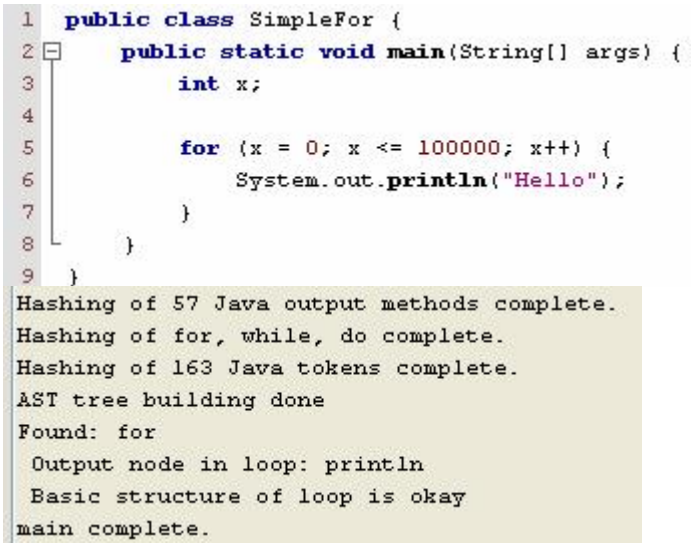
(11) Loops where initial and exit conditions are the same. This is not likely an intentional effort. Example: *for (x=1; x <= 1; x++)*

(12) Loops that iterate between the same negative and positive values. Not likely an intentional effort. Example: *for (x=1; x<10; x /= -1)*

(13) Loops that iterate excessively. This might be intentional or otherwise. Currently *IOScan* reports any loop governed by an arithmetically increasing or decreasing iterator that executes more than 1 million times. This is an arbitrary choice and can easily be made adjustable. Example: *for (x= -1; x > -10000000; x -= 2)*

IOScan distinguishes among all the behaviors listed above. There can, of course, be ambiguous scenarios. For instance, it is quite possible in example (3) above that the novice programmer meant to write *for (x=1; x >- 10; x--)*, so that the error is not in the iterator expression, but in the exit condition expression. *IOScan* only seeks to distinguish between loops that execute in a well-behaved fashion (a finite, predictable number of times for numeric loop-control identifiers), and those that do not, or those that behave in very unconventional ways.

Sample runs for several novice programs are presented below (*Figures 3 – 8*). For a complete listing of feedback provided by *IOScan*, see the *feedback( )* method in the source code listing in Appendix F.



```
1  public class SimpleFor {
2      public static void main(String[] args) {
3          int x;
4
5          for (x = 0; x <= 100000; x++) {
6              System.out.println("Hello");
7          }
8      }
9  }
Hashing of 57 Java output methods complete.
Hashing of for, while, do complete.
Hashing of 163 Java tokens complete.
AST tree building done
Found: for
 Output node in loop: println
 Basic structure of loop is okay
main complete.
```

Figure 3: Properly formed for loop with println output method embedded

```
1  public class SimpleFor {
2      public static void main(String[] args) {
3          int x;
4
5          for (x = 0; x <= 10000000; x--) {
6              System.out.println("Hello");
7          }
8      }
9  }
```

```
Hashing of 57 Java output methods complete.
Hashing of for, while, do complete.
Hashing of 163 Java tokens complete.
AST tree building done
Found: for
 Output node in loop: println
 Loop iterator headed in wrong direction!
 This output will execute more than 1 million times!
main complete.
```

Figure 4: Mal-formed loop; println embedded; iterator should be ++; excessive iterations

```
1  public class SimpleWhile {
2      public static void main(String[] args) {
3          int x;
4          x = 1;
5          while (x <= 100000) {
6              System.out.print("Hello");
7              x *= -1.5;
8          }
9      }
10 }
```

```
Hashing of 57 Java output methods complete.
Hashing of for, while, do complete.
Hashing of 163 Java tokens complete.
AST tree building done
Found: while
 Output node in loop: print
 Iterator alternates between negative and positive values as it increases
  in absolute value.  Be sure the relation between initial and exit conditions
  is appropriate for this situation, or choose a simpler way to iterate from the
  start to the finish of the loop.
main complete.
```

Figure 5: Mal-formed while loop; print method embedded; unusual iterator behavior

```
1  public class SimpleWhile {
2      public static void main(String[] args) {
3          double x;
4          x = 1;
5          while (x <= 100000) {
6              System.out.print("Hello");
7              x = x / 1.5;
8          }
9      }
10 }
```

```
Hashing of 57 Java output methods complete.
Hashing of for, while, do complete.
Hashing of 163 Java tokens complete.
AST tree building done
Found: while
 Output node in loop: print
 This iterator converges toward 0 from either the positive direction or the
  negative direction.  Make sure the relation of the initial and exit
  conditions of the loop are appropriate for this situation.
main complete.
```

Figure 6: Mal-formed while loop; print method embedded; unusual iterator behavior

```
1  public class SimpleWhile {
2      public static void main(String[] args) {
3          double x;
4          x = 1;
5          while (x <= 100) {
6              System.out.println("Hello");
7              x = x - -1.5;
8          }
9      }
10 }
```

```
Hashing of 57 Java output methods complete.
Hashing of for, while, do complete.
Hashing of 163 Java tokens complete.
AST tree building done
Found: while
 Output node in loop: println
 Basic structure of loop is okay
main complete.
```

Figure 7: An unusual iterator that actually works

```
1  public class SimpleWhile {
2      public static void main(String[] args) {
3          double x;
4          x = 1;
5          while (x >= 100) {
6              System.out.println("Hello");
7              x = x - -1.5;
8          }
9      }
10 }
```

```
Hashing of 57 Java output methods complete.
Hashing of for, while, do complete.
Hashing of 163 Java tokens complete.
AST tree building done
Found: while
 Output node in loop: println
 Exit condition operator should be inverse - caused loop termination without execution of loop body
main complete.
```

Figure 8: same as Figure 5 except exit condition is inverted

4.2: Conclusions

The particular concern of this investigation was to be able to produce a tool that can examine novice code and detect possible semantic issues resulting in the execution of code that could compromise host system integrity. The particular issue was the detection of I/O methods executing within iterative structures in Java. *IOScan* is able to do that, at least for iterative structures in which the loop control identifier has no dependencies, is initially declared and/or defined with simple assignment statements, the exit condition is a simple single Boolean comparison, and where the iterator is a simple single increment, decrement, multiplier or divisor statement. The program carries out an automated semantic analysis of Java programs and is built to be extensible. Its *modus operandus* is to emulate the behavior of a human reviewer, carrying out the same kind of pre-run-time analysis and predictive modeling of run-time behavior that a human reviewer carries out. Writing code that can automatically evaluate a wide range of novice programs and

provide useful feedback for a range of semantic issues in novice code is a large task. The present effort represents a rudimentary first effort in that direction.

Chapter 5: Future Work

Development of a successful runtime identifier model would be an important next step in extending the capability of *IOScan*. An important feature of such a model would be for it to be able to evaluate EXPR nodes (expressions) in an AST. The model suggested by Reiss (mentioned earlier [21]) could provide a useful starting point for development in this direction. An important issue to address is to be able to recognize identifier scoping and external dependencies such as method calls or user input at runtime. A next logical step would be to extend the runtime identifier model so that expressions involving non-numeric loop-control identifiers could be evaluated. Other directions could include the evaluation of nested loops and recursive structures.

There are also other semantic issues in novice code apart from the dangers of iterative output. Truong [23:319] mentions a number of these: improperly structured switch blocks, hard-coding of literals, poor scoping of identifiers, and unused identifiers. These could also be fruitful avenues for exploration and extension of the current research. Hristova [8] also describes a number of specific syntactic, semantic and logical errors throughout her article.

For the larger project of which this investigation is a part, it remains to be seen how the present research can be integrated into the system as it evolves. In addition to the approach described in this investigation, some of the approaches outlined in the literature review are worthy of investigation, particularly for first-semester students. Two approaches especially seem appropriate in this regard: starting novices with a simpler subset of the language making use of a relatively simple and uncluttered development environment, and providing projects of a "fill-the-gap-with-well-formed-code" nature.

There is no reason that either of these approaches could not be integrated with the author's own approach as the system is refined and developed.

A number of more recent approaches to the problem of providing intelligent tutoring systems rely on web-based delivery and XML-based parsing tools that have been developed. These approaches represent quite a different approach to providing novice programmers with useful feedback than what has been considered here and should be examined. Finally, more recent approaches are incorporating visualization tools for providing intuitive feedback to novices, taking advantage of the great strides that have been made in multimedia development in recent years. These too would be worthy of investigation.

References

[1] Badros, Greg J. (2000): JavaML: a markup language for Java source code. *Computer Networks,* **33**(1-6): 159-177.

[2] Callaghan, Michael and Heiko Hirschmüller (1998): 3-D visualisation of design patterns and Java programs in computer science education. *ACM SIGCSE Bulletin, Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, **30**(3): 37-40.

[3] Campbell, Alistair E. R., Geoffrey L. Catto, Eric E. Hansen (2003): Language-independent interactive data visualization. *ACM SIGCSE Bulletin , Proceedings of the 34th SIGCSE technical symposium on Computer science education*, **35**(1): 215-219.

[4] Chomsky, Noam (1956): Three models for the description of language. *IRE Transactions on Information Theory*, **2**(1956): 113-124.

[5] DePasquale, Peter, John A. N. Lee, Manuel A. Pérez-Quiñones (2004): Evaluation of subsetting programming language elements in a novice's programming environment. *ACM SIGCSE Bulletin, Proceedings of the 35th SIGCSE technical symposium on Computer science education*, **36**(1): 260-264.

[6] Etheredge, Jim (2004): CMeRun: program logic debugging courseware for CS1/CS2 students. *ACM SIGCSE Bulletin, Proceedings of the 35th SIGCSE technical symposium on Computer science education*, **36**(1): 22-25.

[7] Fabry, Johan, Tom Mens (2003): Language-independent detection of object-oriented design patterns. *Computer Languages, Systems & Structures*, **30**(2004): 21-33.

[8] Hristova, Maria, Ananya Misra, Megan Rutter and Rebecca Mercuri (2003): Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin , Proceedings of the 34th SIGCSE technical symposium on Computer science education*, **35**(1): 153-156.

[9] http://www.antlr.org. This is the antlr web site. The open-source product can be freely downloaded. There are complete grammars available for several languages and there are a great number of tutorials and forums available for users and developers of the product.

[10] Huynh, Quoc Hung Le (2005): Comparing Java programs: syntactic and semantic contextual differences. *Thesis, University of Oslo*, January, 2005. Available online at http://wo.uio.no/as/WebObjects/theses.woa/wa/these?WORKID=24301 (4/09/2005).

[11] Jackson, David and Michelle Usher (1997): Grading student programs using ASSYST. *ACM SIGCSE Bulletin, Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, **29**(1): 335-339.

[12] Java 1.4.2 language specification: http://java.sun.com/j2se/1.4.2/docs/api/

[13] Java BNF grammars are available at: http://www.antlr.org/grammar/list

[14] Kaplan, Alan and Denise Shoup (2000). CUPV-a visualization tool for generated parsers. *ACM SIGCSE Bulletin, Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, **32**(1): 11-15.

[15] Kumar, Amruth N. (2005): Results from the evaluation of the effectiveness of an online tutor on expression evaluation. *ACM SIGCSE Bulletin , Proceedings of the 36th SIGCSE technical symposium on Computer science education*, **37**(1): 216-220.

[16] Lapierre, Sebastien, Bruno Lague, Charles Leduc (2001): Datrix™ Source Code Model and its Interchange Format: Lessons Learned and Considerations for Future Work. *ACM SIGSOFT Software Engineering Notes*, **26**(1): 53-56.

[17] Lee, Dongwon, Wesley W. Chu (2001): CPI:Constraints-Preserving Inlining algorithm for mapping XML DTD to relational schema. *Data & Knowledge Engineering,* **39**(2001): 3-25.

[18] Madden, Bill and John Jenq (2004): Semantic Parsing of Java I/O Programming for an On Line System. *Proceedings of The Eighth World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando, Florida, **1**:76.

[19] Palaste, Joona (2001). Java XML Parsing Specification. Online article available at http://www.cs.helsinki.fi/u/campa/teaching/j2me/papers/AJS.pdf (04/09/2005), pp97-109.

[20] Rabek, Jesse C., Roger I. Khazan, Scott M. Lewandowski, Robert K. Cunningham (2003): Defensive technology: Detection of injected, dynamically generated, and obfuscated malicious code. *Proceedings of the 2003 ACM workshop on Rapid Malcode, Washington,DC, USA*. October 27, 2003: 76-82.

[21] Reiss, Steven P. (1999): The Desert Environment. *ACM Transactions on Software Engineering and Methodology*, **8**(4): 297-342.

[22] Sykes, Edward R. (2003): An Intelligent Tutoring System Prototype for Learning to Program Java. *The 3rd IEEE International Conference on Advanced Learning Technologies*, Athens, Greece, p485.

[23] Truong, Nghi, Paul Roe, Peter Bancroft (2004): Static Analysis of Students' Java Programs**.** *Proceedings of the sixth conference on Australian computing education,* Dunedin, New Zealand, **30**: 317-325.

[24] Wikipedia (http://en.wikipedia.org/wiki/Main_Page). A number of the more technical details in this and the next paragraph are drawn from various articles on the Wikipedia web site. See Appendix G for a list of search terms used in Wikipedia.

Appendix A: Java 1.4.2 Input/Output Classes

In the electronic version of this document all the keywords below are hyperlinked to the documentation for Java Version 1.4.2 on Sun's web site. *Italicized* entries are interfaces; regular type entries are classes.

Input Classes/Interfaces

AudioFileReader
AudioInputStream
BufferedInputStream
BufferedReader
ByteArrayInputStream
CharArrayReader
CipherInputStream
DataInputStream
DataInputStream
DigestInputStream
FileCacheImageInputStream
FileImageInputStream
FileInputStream
FileReader
FilterInputStream
FilterReader
ImageInputStream
ImageInputStreamImpl
ImageInputStreamSpi
ImageReader
ImageReaderSpi
ImageReaderWriterSpi
ImageReadParam
InputStream
InputStream
InputStream
InputStreamReader
JarInputStream
MemoryCacheImageInputStream
MidiFileReader
ObjectInput
ObjectInputStream
ObjectInputStream.GetField
ObjectInputValidation
PipedInputStream
PipedReader
Reader
StringBufferInputStream
StringReader
WriteAbortedException
ZipInputStream

Output Classes/Interfaces

AudioFileWriter
BufferedOutputStream
BufferedWriter
ByteArrayOutputStream
CharArrayWriter
CipherOutputStream
DataOutput
DataOutputStream
DataOutputStream
DigestOutputStream
FileCacheImageOutputStream
FileImageOutputStream
FileOutputStream
FileWriter
FilterOutputStream
FilterWriter
ImageOutputStream
ImageOutputStreamImpl
ImageOutputStreamSpi
ImageWriteParam –no output
ImageWriter
ImageWriterSpi
JarOutputStream
MemoryCacheImageOutputStream
MidiFileWriter
ObjectOutput
ObjectOutputStream
ObjectOutputStream.PutField
OutputStream
OutputStreamWriter

Appendix A (continued)

PipedOutputStream
PipedWriter
PrintStream
PrintWriter
StringWriter
Writer
ZipOutputStream

Other I/O-related Classes/Interfaces

EOFException
File
FileChannel
FileChannel.MapMode
FileChooserUI
FileDescriptor
FileDialog
FileFilter
FileHandler
FileFilter
FileLock
FileLockInterruptionException
FilenameFilter
FileNameMap
FileNotFoundException
FilePermission
FileSystemView
FileView
IIOByteBuffer
IIOException
IIOImage
IIOInvalidTreeException
IIOMetadata
IIOMetadataController
IIOMetadataFormat
IIOMetadataFormatImpl
IIOMetadataNode
IIOParam
IIOParamController
IIORegistry
IIOServiceProvider
IIOReadProgressListener
IIOReadUpdateListener
IIOReadWarningListener
IIOWriteProgressListener
IIOWriteWarningListener
ImageIO
InputMap
InputMapUIResource
InputContext
InputEvent
InputMethod
InputMethodContext
InputMethodDescriptor
InputMethodEvent
InputMethodHighlight
InputMethodListener
InputMethodRequests
InputSource
InputSubset
InputVerifier
IOException
JarFile
ObjectStreamClass
ObjectStreamConstants
ObjectStreamException
ObjectStreamField
ReadOnlyBufferException
StreamCorruptedException
StreamHandler
StreamPrintService
StreamPrintServiceFactory
StreamResult
StreamSource
StreamTokenizer
StringBuffer
StringTokenizer
WritableByteChannel
WritableRaster
WritableRenderedImage
ZipFile

Appendix B: List of Java 1.4.2 Output Methods

Below is a listing of all output methods associated with output classes in Java 1.4.2. Many methods are polymorphous, often associated with many or most Java output classes.

checkError
connect
createOutputStreamInstance
createWriterInstance
flush
flushBefore
flushBits
newLine
prepareInsertEmpty
prepareReplacePixels
prepareWriteEmpty
prepareWriteSequence
print
println
processImageProgress
processImageStarted
processThumbnailComplete
processThumbnailProgress
processThumbnailStarted
put
putField
putNextEntry
replaceImageMetadata
replaceObject
replacePixels
replacePixels
replaceStreamMetadata
reset
seek
setOutput
write
writeBit
writeBits
writeBoolean
writeByte
writeBytes
writeChar
writeChars
writeClassDescriptor
writeDouble
writeDoubles
writeFields
writeFloat
writeFloats
writeInt
writeInts
writeInsert
writeLong
writeLongs
writeObjectOverride
writeShort
writeShorts
writeStreamHeader
writeToSequence
writeTo
writeUnshared
writeUTF

Appendix C: Java 1.4.2 Output Classes and Associated Output Methods

AudioFileWriter( )
write(AudioInputStream stream, AudioFileFormat.Type fileType, File out)
write(AudioInputStream stream, AudioFileFormat.Type fileType, OutputStream out)

BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)
flush()
write(byte[] b, int off, int len)
write(int b)

BufferedWriter(Writer out)
BufferedWriter(Writer out, int sz)
flush()
newLine()
write(char[] cbuf, int off, int len)
write(int c)
write(String s, int off, int len)

ByteArrayOutputStream()
ByteArrayOutputStream(int size)
reset()
write(byte[] b, int off, int len)
write(int b)
writeTo(OutputStream out)

CharArrayWriter()
CharArrayWriter(int initialSize)
flush()
reset()
write(char[] c, int off, int len)
write(int c)
writeTo(Writer out)

CipherOutputStream(OutputStream os)
CipherOutputStream(OutputStream os, Cipher c)
flush()
write(byte[] b)
write(byte[] b, int off, int len)
write(int b)

DataOutputStream(OutputStream out)
flush()
write(byte[] b, int off, int len)
write(int b)

Appendix C (continued)

writeBoolean(boolean v)
writeByte(int v)
writeBytes(String s)
writeChar(int v)
writeChars(String s)
writeDouble(double v
writeFloat(float v)
writeInt(int v)
writeLong(long v)
writeShort(int v)
writeUTF(String str)

DigestOutputStream(OutputStream stream, MessageDigest digest)
write(byte[] b, int off, int len)
write(int b)

FileCacheImageOutputStream(OutputStream stream, File cacheDir)
flushBefore(long pos)
seek(long pos)
write(byte[] b, int off, int len)
write(int b)

FileImageOutputStream(File f)
FileImageOutputStream(RandomAccessFile raf)
seek(long pos)
write(byte[] b, int off, int len)
write(int b)

FileOutputStream(File file)
FileOutputStream(File file, boolean append)
FileOutputStream(FileDescriptor fdObj)

FileOutputStream(String name)
FileOutputStream(String name, boolean append)
write(byte[] b)
write(byte[] b, int off, int len)
write(int b)

FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(FileDescriptor fd)
FileWriter(String fileName)
FileWriter(String fileName, boolean append)

Appendix C (continued)

FilterOutputStream(OutputStream out)
flush()
write(byte[] b)
write(byte[] b, int off, int len)
write(int b)

FilterWriter(Writer out)
flush()
write(char[] cbuf, int off, int len)
write(int c)
write(String str, int off, int len)

ImageOutputStreamImpl()
flushBits()
write(byte[] b)
write(byte[] b, int off, int len)
write(int b)
writeBit(int bit)
writeBits(long bits, int numBits)
writeBoolean(boolean v)
writeByte(int v)
writeBytes(String s)
writeChar(int v)
writeChars(char[] c, int off, int len)
writeChars(String s)
writeDouble(double v)
writeDoubles(double[] d, int off, int len)
writeFloat(float v)
writeFloats(float[] f, int off, int len)
writeInt(int v)
writeInts(int[] i, int off, int len)
writeLong(long v)
writeLongs(long[] l, int off, int len)
writeShort(int v)
writeShorts(short[] s, int off, int len)
writeUTF(String s)

ImageOutputStreamSpi()
ImageOutputStreamSpi(String vendorName, String version, Class outputClass)
createOutputStreamInstance(Object output)
createOutputStreamInstance(Object output, boolean useCache, File cacheDir)

ImageWriter(ImageWriterSpi originatingProvider)

Appendix C (continued)

prepareInsertEmpty(int imageIndex, ImageTypeSpecifier imageType, int width, int height, IIOMetadata imageMetadata, List thumbnails, ImageWriteParam param)
prepareReplacePixels(int imageIndex, Rectangle region)
prepareWriteEmpty(IIOMetadata streamMetadata, ImageTypeSpecifier imageType, int width, int height, IIOMetadata imageMetadata, List thumbnails, ImageWriteParam param)
prepareWriteSequence(IIOMetadata streamMetadata)
processImageProgress(float percentageDone)
processImageStarted(int imageIndex)
processThumbnailComplete()
processThumbnailProgress(float percentageDone)
processThumbnailStarted(int imageIndex, int thumbnailIndex)
replaceImageMetadata(int imageIndex, IIOMetadata imageMetadata)
replacePixels(Raster raster, ImageWriteParam param)
replacePixels(RenderedImage image, ImageWriteParam param)
replaceStreamMetadata(IIOMetadata streamMetadata)
setOutput(Object output)
write(IIOImage image)
write(IIOMetadata streamMetadata, IIOImage image, ImageWriteParam param)
write(RenderedImage image)
writeInsert(int imageIndex, IIOImage image, ImageWriteParam param)
writeToSequence(IIOImage image, ImageWriteParam param)

ImageWriterSpi()
ImageWriterSpi(String vendorName, String version, String[] names, String[] suffixes, String[] MIMETypes, String writerClassName, Class[] outputTypes, String[] readerSpiNames, boolean supportsStandardStreamMetadataFormat, String nativeStreamMetadataFormatName, String nativeStreamMetadataFormatClassName, String[] extraStreamMetadataFormatNames, String[] extraStreamMetadataFormatClassNames, boolean supportsStandardImageMetadataFormat, String nativeImageMetadataFormatName, String nativeImageMetadataFormatClassName, String[] extraImageMetadataFormatNames, String[] extraImageMetadataFormatClassNames)
createWriterInstance()
createWriterInstance(Object extension)

JarOutputStream(OutputStream out)
JarOutputStream(OutputStream out, Manifest man)
putNextEntry(ZipEntry ze)

MemoryCacheImageOutputStream(OutputStream stream)
flushBefore(long pos)
write(byte[] b, int off, int len)
write(int b)

Appendix C (continued)

MidiFileWriter()
write(Sequence in, int fileType, File out)
write(Sequence in, int fileType, OutputStream out)

ObjectOutputStream()
ObjectOutputStream(OutputStream out)
flush()
putFields()
replaceObject(Object obj)
write(byte[] buf)
write(byte[] buf, int off, int len)
write(int val)
writeBoolean(boolean val)
writeByte(int val)
writeBytes(String str)
writeChar(int val)
writeChars(String str)
writeClassDescriptor(ObjectStreamClass desc)
writeDouble(double val)
writeFields()
writeFloat(float val)
writeInt(int val)
writeLong(long val)
writeObject(Object obj)
writeObjectOverride(Object obj)
writeShort(int val)
writeStreamHeader()
writeUnshared(Object obj)
writeUTF(String str)

ObjectOutputStream.PutField()
put(String name, boolean val)
put(String name, byte val)
put(String name, char val)
put(String name, double val)
put(String name, float val)
put(String name, int val)
put(String name, long val)
put(String name, Object val)
put(String name, short val)
write(ObjectOutput out)

OutputStream()
flush()

Appendix C (continued)

write(byte[] b)
write(byte[] b, int off, int len)
write(int b)

OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, Charset cs)
OutputStreamWriter(OutputStream out, CharsetEncoder enc)
OutputStreamWriter(OutputStream out, String charsetName)
flush()
write(char[] cbuf, int off, int len)
write(int c)
write(String str, int off, int len)

PipedOutputStream()
PipedOutputStream(PipedInputStream snk)
connect(PipedInputStream snk)
flush()
write(byte[] b, int off, int len)
write(int b)

PipedWriter()
PipedWriter(PipedReader snk)
connect(PipedReader snk)
flush()
write(char[] cbuf, int off, int len)
write(int c)

PrintStream(OutputStream out)
PrintStream(OutputStream out, boolean autoFlush)
PrintStream(OutputStream out, boolean autoFlush, String encoding)
checkError()
flush()
print(boolean b)
print(char c)
print(char[] s)
print(double d)
print(float f)
print(int i)
print(long l)
print(Object obj)
print(String s)
println()
println(boolean x)
println(char x)

Appendix C (continued)

println(char[] x)
println(double x)
println(float x)
println(int x)
println(long x)
println(Object x)
println(String x)
write(byte[] buf, int off, int len)
write(int b)

PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
checkError()
flush()
print(boolean b)
print(char c)
print(char[] s)
print(double d)
print(float f)
print(int i)
print(long l)
print(Object obj)
print(String s)
println()
println(boolean x)
println(char x)
println(char[] x)
println(double x)
println(float x)
println(int x)
println(long x)
println(Object x)
println(String x)
write(char[] buf)
write(char[] buf, int off, int len)
write(int c)
write(String s)
write(String s, int off, int len)

StringWriter()
StringWriter(int initialSize)
flush()

Appendix C (continued)

write(char[] buf, int off, int len)
write(int c)
write(String s)
write(String s, int off, int len)

Writer()
Writer(Object lock)
flush()
write(char[] buf)
write(char[] buf, int off, int len)
write(int c)
write(String s)
write(String s, int off, int len)

ZipOutputStream(OutputStream out)
putNextEntry(ZipEntry e)
write(byte[] b, int off, int len)

Appendix D: List of Java 1.4.2 tokens

1. ABSTRACT
2. ARRAY_DECLARATOR
3. ARRAY_INIT
4. ASSIGN
5. BAND
6. BAND_ASSIGN
7. BLOCK
8. BNOT
9. BOR
10. BOR_ASSIGN
11. BSR
12. BSR_ASSIGN
13. BXOR
14. BXOR_ASSIGN
15. CASE_GROUP
16. CHAR_LITERAL
17. CLASS_DEF
18. COLON
19. COMMA
20. CTOR_CALL
21. CTOR_DEF
22. DEC
23. DIV
24. DIV_ASSIGN
25. DOT
26. ELIST
27. EMPTY_STAT
28. EQUAL
29. ESC
30. EXPONENT
31. EXPR
32. EXTENDS_CLAUSE
33. FINAL
34. FLOAT_SUFFIX
35. FOR_CONDITION
36. FOR_INIT
37. FOR_ITERATOR
38. GE
39. GT
40. HEX_DIGIT
41. IDENT
42. IMPLEMENTS_CLAUSE
43. IMPORT
44. INC
45. INDEX_OP
46. INSTANCE_INIT
47. INTERFACE_DEF
48. LABELED_STAT
49. LAND
50. LBRACK
51. LCURLY
52. LE
53. LNOT
54. LOR
55. LPAREN
56. LT
57. METHOD_CALL
58. METHOD_DEF
59. MINUS
60. MINUS_ASSIGN
61. ML_COMMENT
62. MOD
63. MOD_ASSIGN
64. MODIFIERS
65. NOT_EQUAL
66. NUM_DOUBLE
67. NUM_FLOAT
68. NUM_INT
69. NUM_LONG
70. OBJBLOCK
71. PACKAGE_DEF
72. PARAMETER_DEF
73. PARAMETERS
74. PLUS
75. PLUS_ASSIGN
76. POST_DEC
77. POST_INC
78. QUESTION
79. RBRACK
80. RCURLY
81. RPAREN
82. SEMI
83. SL
84. SL_ASSIGN

Appendix D (continued)

85. SL_COMMENT
86. SLIST
87. SR
88. SR_ASSIGN
89. STAR
90. STAR_ASSIGN
91. STATIC_INIT
92. STRICTFP
93. STRING_LITERAL
94. SUPER_CTOR_CALL
95. TYPE
96. TYPECAST
97. UNARY_MINUS
98. UNARY_PLUS
99. VARIABLE_DEF
100. VOCAB
101. WS
102. abstract
103. assert
104. boolean
105. break
106. byte
107. case
108. catch
109. char
110. class
111. continue
112. default
113. do
114. double
115. else
116. extends
117. false
118. final
119. finally
120. float
121. for
122. if
123. implements
124. import
125. instanceof
126. int
127. interface
128. long
129. native
130. new,
131. null
132. package
133. private
134. protected
135. public
136. return
137. short
138. static
139. strictfp
140. super
141. switch
142. symchronized
143. this
144. threadsafe
145. throw
146. throws
147. transient
148. true
149. try
150. void
151. volatile
152. while
153. byvalue
154. cast
155. const
156. future
157. generic
158. goto
159. inner
160. operator
161. outer
162. rest
163. var

Appendix E: *for*, *while* and *do* structures tested with *IOScan*

1. ```
   vartype x;
   …
   for (x = n; x bop n; iter) { … }
   ```

2. ```
   for (vartype x = n; x bop n; iter) { … }
   ```

3. ```
   vartype x;
   …
   x = n;
   …
   while (x bop n) { … iter … }
   ```

4. ```
   vartype x = n;
   …
   while (x bop n) { … iter …}
   ```

5. ```
   vartype x;
   …
   x = n;
   …
   do { … iter … } while (x bop n)
   ```

6. ```
   vartype x = n;
   …
   do { … iter … } while (x bop n)
   ```

Explanation:

iter	can take any of the following forms: `x++ | x-- | ++x | --x | x op= n | x = x op n`
op	are standard arithmetic operators: +, -, *, /
bop	are standard Boolean operators: <, >, = =, <=, >=, !=
vartype	can be any legal Java primitive numeric type
n	can be any permissible numeric value, depending on `vartype`
…	represents additional in-line code

Appendix F: Source Code listing for *IOScan.java*

```
1  import java.io.*;
2  import java.util.Hashtable;
3  import antlr.collections.AST;
4  import antlr.collections.ASTEnumeration;
5  import antlr.collections.impl.*;
6  import antlr.debug.misc.*;
7  import antlr.*;
8  import java.awt.event.*;
9  import java.lang.Math.*;
10
11 public class IOScan {
12
13     //Class-level declaration
14     static boolean showTree = false;             //show tree in frame or not
15     static Hashtable outHT = new Hashtable();    //hash Java output methods
16     static Hashtable blockHT = new Hashtable();  //hash: while, for, do.
17     static Hashtable tokensHT = new Hashtable(); //hash: all AST tokens
18     static AST fullTree = null;                  //full AST tree
19
20     //main method - dispatches methods for building hashtables, main AST
21     // tree and locating iterated Java output methods
22     public static void main(String[] args) throws Exception {
23
24         //declarations/initialization
25         AST t = null;
26         Vector roots = new Vector(10);
27         ASTEnumeration e = null;
28
29         //build hash tables
30         hashOutputMethods();
31         hashJavaLoops();
32         hashTokens();
33
34         //build AST tree and walk it, looking for loops
35         t = buildAST(args);
36         fullTree = t;
37         if (t != null) {
38             e = findLoops(roots, t); //builds an enum of AST
39         }//if
40
41         System.out.println("main complete.");
42
43     }//main method
44
45     //recursive method - depth-first search through tree looking for loops
46     public static ASTEnumeration findLoops(Vector v, AST t) {
47
48         //declarations and initialization
49         AST thisNode, child;
50         thisNode = t;
51
52         while (thisNode != null) {
53             child = thisNode.getFirstChild();
54             if (child != null) {
55                 findLoops(v, child); //recursion occurs here
56             }//if
57             v.appendElement(thisNode);
58             checkLoops(thisNode, t); //if thisNode starts loop, look for I/O
59             thisNode = thisNode.getNextSibling();
60         }//while
```

Appendix F (continued)

```
61
62          return new ASTEnumerator(v);
63
64      }//findLoops
65
66      //check to see if thisNode starts a loop (for, while, do)
67      public static void checkLoops(AST node, AST t) {
68
69          //declarations and initialization
70          AST child = null, sibling = null, varType = null;
71          AST ident = null, assign = null, expr = null;
72          AST subtree = null;
73          ASTEnumeration st = null;
74          boolean foundVarDef = false;
75          String nodeText = node.getText();
76
77          //if node is start of a loop structure, look for output methods
78          if (blockHT.contains(nodeText)) {
79              System.out.println("Found: " + nodeText);
80
81              //check subtree for output methods
82              subtree = node.getFirstChild();   //must go inside of structure
83              st = findOutputNodes(subtree);
84              //if no output methods found, move on
85              if (!st.hasMoreNodes()) {
86                  System.out.println("  No output nodes found");
87              } else { //show output nodes, branch to process for, while & do
88                  while (st.hasMoreNodes()) {
89                      System.out.println(" Output node in loop: " + st.nextNode());
90                  }//while
91
92                  //if output method found, branch: process for-while-do loops
93                  if (nodeText.equals("for")) {
94                      processFor(node);
95                  //} else if (nodeText.equals("while")) {
96                  //    processWhileDo(node, t); //logic very similar for both
97                  } else {
98                      processWhileDo(node, t);
99                      //System.out.println("Error: for-while-do ONLY coded");
100                 }//if node was for, while or do
101             }//if there are output nodes inside this current loop
102         }//if the current node is a loop structure
103
104     }//checkLoops
105
106     //Process for loops: captures init, exit and iter conditions that define
107     // the execution of the for loop
108     public static void processFor(AST loopNode) {
109         //presently, we assume that for_loop init, exit (cond) and iterator
110         // have no dependencies.
111
112         //declarations and initialization
113         AST init, cond, iter;
114         AST elist, expr;
115         AST initOper, condOper, iterOper;
116         AST initIdent, initVal, tmp, condIdent, condVal, iterIdent, arithOper, iterVal;
117         arithOper = null;
118         iterVal = null;
119         String strCond = "";  //represents exit condition operator
```

Appendix F (continued)

```
120
121       //parse nodes in FOR_INIT, FOR_CONDITION and FOR_ITERATOR subtrees
122       init = loopNode.getFirstChild(); //walking default structure of
123       cond = init.getNextSibling();    //for-loop with no dependencies and
124       iter = cond.getNextSibling();    //simple assignment and boolean
125       tmp = init.getFirstChild();      //expressions for init/cond/iter
126       if (tmp.getText().equals("ELIST")) {
127            expr = tmp.getFirstChild();
128            initOper = expr.getFirstChild();
129            initIdent = initOper.getFirstChild();
130            initVal = initIdent.getNextSibling();
131       } else { //VAR_DEF is occurring inside for definition
132           tmp = tmp.getFirstChild();             //Modifiers
133           tmp = tmp.getNextSibling();            //Type
134           initIdent = tmp.getNextSibling();      //Ident name
135           initOper = initIdent.getNextSibling(); //init operator
136           tmp = initOper.getFirstChild();        //EXPR node
137           initVal = tmp.getFirstChild();  //init val of loop-control ident
138       }
139       expr = cond.getFirstChild();              //keep walking to locate
140       condOper = expr.getFirstChild();          //iter node
141       condIdent = condOper.getFirstChild();
142       condVal = condIdent.getNextSibling();
143       elist = iter.getFirstChild();
144       expr = elist.getFirstChild();
145       iterOper = expr.getFirstChild();
146       iterIdent = iterOper.getFirstChild();
147
148       //declarations for determining loop direction and behavior
149       int itDir = 0;  //+int moves right (on number line); -int moves left
150       double dblInit, dblCond, dblIter;  //converts AST nodes to numerics
151       dblInit = dblCond = dblIter = 0.0;
152
153       //convert value nodes to numerics - doubles are least restrictive
154       // next 7 lines convert the cond (exit_condition) node to a double
155       if (condVal.getText().equals("-")) {        //in case of unary minus
156           tmp = condVal.getFirstChild();
157           dblCond = -(Double.parseDouble(tmp.getText()));
158       } else if (condVal.getText().equals("+")) { //rare case: unary plus
159           tmp = condVal.getFirstChild();
160           dblCond = Double.parseDouble(tmp.getText());
161       } else dblCond = Double.parseDouble(condVal.getText());
162
163       //next 7 lines convert init_condition to a double
164        if (initVal.getText().equals("-")) {       //in case of unary minus
165           tmp = initVal.getFirstChild();
166           dblInit = -(Double.parseDouble(tmp.getText()));
167       } else if (initVal.getText().equals("+")) { //rare case: unary plus
168           tmp = initVal.getFirstChild();
169           dblInit = Double.parseDouble(tmp.getText());
170       } else dblInit = Double.parseDouble(initVal.getText());
171
172       //evaluate iter expression, result will be a double
173       String strOper = iterOper.getText(); //if iterOper=null report Error
174
175       if (strOper.equals("=")) {   //assumes expression form: x = x op n
176           //process x = x + 1
177           iterVal = iterOper.getFirstChild();
178           arithOper = iterVal.getNextSibling(); //this can be: +, -, *, /
179           iterVal = arithOper.getFirstChild();
```

Appendix F (continued)

```
180             iterVal = iterVal.getNextSibling();
181             dblIter = Double.parseDouble(iterVal.getText());
182         } else {
183             dblIter = getIter(iterOper);
184         }//if
185 
186         if (arithOper != null) {
187             strOper = arithOper.getText() + "=";
188         }
189 
190         //There are 9 possible outcomes for a direction of the iterator:
191         //see comments in iterDirect method signature (line 470 approx)
192         itDir = iterDirect(strOper, dblIter, dblInit);
193         strCond = condOper.getText();
194 
195         //We now have all the values needed for analysis of for loops:
196         //   dblInit = initial starting value of loop control identifier
197         //   dblIter and strOper = iterator value and operation
198         //   dblCond = exit value of loop control identifier
199         //Logic is similar to that in the while/do loop analysis.
200 
201         //display feedback analysis (line 530 approx)
202         feedBack(dblInit, dblIter, dblCond, itDir, strCond);
203 
204     }//processFor
205 
206     //Process while and do loops
207     public static void processWhileDo(AST loopNode, AST t) {
208 
209         //presently we assume a while or do loop employing a loop control
210         //variable that has no dependencies on its exit_condition.
211 
212         //declarations and initialization
213         int condType;
214         //ASTEnumeration e;
215         AST init, cond, iter;
216         AST sList, expr;
217         AST initOper, condOper, iterOper, arithOper;
218         AST initIdent, initVal, tmp, condIdent, condVal, iterIdent, iterVal;
219         iterVal = null;
220         arithOper = null;
221         String strCond;        //represents of exit condition operator
222 
223         //the following will be used to convert AST nodes to numerics so we
224         // can evaluate loop direction and behavior
225         int itDir = 0;
226         double dblInit, dblCond, dblIter;
227         dblInit =dblCond = dblIter = 0.0;
228 
229         //There are two major nodes inside while/do loops: EXPR and SLIST
230         // EXPR contains the exit condition for the loop
231         // SLIST contains the body of the loop, including the iterator
232         if (loopNode.getText().equals("while")) {
233             expr = loopNode.getFirstChild()    //while loop
234             sList = expr.getNextSibling();
235         } else {
236             sList = loopNode.getFirstChild();  //order of two major nodes is
237             expr = sList.getNextSibling();     //reversed in do-loop
238         }
239 
240         //walk the EXPR node: parenthetical exit condition following while/do
```

Appendix F (continued)

```
241       // e = evalTree(expr); - eventually we will have a general EXPRession
242       // evaluator
243       condOper = expr.getFirstChild();
244       condIdent = condOper.getFirstChild();
245       condVal = condIdent.getNextSibling();
246
247       //Convert cond (exit_condition) to double, testing for unary
248       // minus and unary plus along the way
249       if (condVal.getText().equals("-")) {        //in case of unary minus
250           tmp = condVal.getFirstChild();
251           dblCond = -(Double.parseDouble(tmp.getText()));
252       } else if (condVal.getText().equals("+")) { //rare case: unary plus
253           tmp = condVal.getFirstChild();
254           dblCond = Double.parseDouble(tmp.getText());
255       } else dblCond = Double.parseDouble(condVal.getText());
256
257       condType = condVal.getType(); //138=int/141=float/142=long/
258                                     //143=double, but compiler does
259                                     //implicit casting when it sets actual
260                                     //coded val into loopCtrlIdent
261       //if (exitVal.getText().equals("EXPR")) then the expression is more
262       // complex and we will need a more general EXPRession evaluator
263       // Now we have the loop control variable (AST loopCtrlIdent),
264       // its exit value (AST exitVal) and type (int loopCtrlType) - though
265       // this last is implicitly cast by compiler into loopCtrlIdent's
266       // actual declared type.
267
268       //Next we search SLIST locating the EXPR that contains the iterator.
269       //Look for three forms of iterator: POST_INC, ASSIGN and PLUS_ASSIGN.
270       //Convert iter node to double
271       AST sListChild = sList.getFirstChild(); //drop into SLIST first
272       iterOper = findIterOper(sListChild, condIdent);
273       String strOper = iterOper.getText();
274       if (strOper.equals("=")) { //assumes expression form: x = x op n
275           iterVal = iterOper.getFirstChild();
276           arithOper = iterVal.getNextSibling(); //this could be: +, -, *, /
277           iterVal = arithOper.getFirstChild();
278           iterVal = iterVal.getNextSibling();
279           //check for a unary minus
280           if (iterVal.getText().equals("-")) {
281               tmp = iterVal.getFirstChild();
282               dblIter = -(Double.parseDouble(tmp.getText()));
283           } else if (iterVal.getText().equals("+")) {//rare: unary plus
284               tmp = iterVal.getFirstChild();
285               dblIter = Double.parseDouble(tmp.getText());
286           } else dblIter = Double.parseDouble(iterVal.getText());
287       } else {
288           dblIter = getIter(iterOper);
289       }//if
290
291       if (arithOper !=null) {
292           strOper = arithOper.getText() + "=";
293       }//if
294
295       //Next we locate the VARIABLE_DEF and/or EXPR that defines init
296       //condition.  findInitVal locates only declarations/expressions of
297       //the form: int x = n;
298       //int x; x = n;
299       //int x; x += n; (or *=, /=, -=)
300       initVal = null;
301       initVal = findInitVal(fullTree, condIdent, loopNode, initVal);
```

Appendix F (continued)

```
302         if (initVal.getText().equals("-")) {
303             tmp = initVal.getFirstChild();
304             dblInit = -(Double.parseDouble(tmp.getText()));
305         } else if (initVal.getText().equals("+")) {//rare: unary plus
306             tmp = initVal.getFirstChild();
307             dblInit = Double.parseDouble(tmp.getText());
308         } else dblInit = Double.parseDouble(initVal.getText());
309 
310         //There are 9 possible outcomes for a direction of the iterator:
311         //see comments in iterDirect method signature (line 470 approx.)
312         itDir = iterDirect(strOper, dblIter, dblInit);
313 
314         strCond = condOper.getText();
315 
316         //We now have all the values needed for analysis of while/do loops:
317         //   dblInit = initial starting value of loop control identifier
318         //   dblIter and strOper = iterator value and operation
319         //   dblCond = exit value of loop control identifier
320         //Logic is similar to that in the for loop analysis.
321 
322         //display feedback analysis (line 530 approx)
323         feedBack(dblInit, dblIter, dblCond, itDir, strCond);
324 
325     }//processWhileDo
326 
327     //recursive method that starts walking entire AST tree finding first the
328     // VARIABLE_DEF of condIdent (the exit_condition of a while/do loop),
329     // examining it for an initial value of condIdent, then continuing to
330     // walk forward through AST t looking for any EXPRessions involving a re-
331     // valuation of condIdent.  It stops looking when it encounters loopNode
332     // (the current while/do loop that spawned this search).
333     public static AST findInitVal(AST t, AST condIdent, AST loopNode, AST result) {
334 
335         //Declarations and initialization
336         AST thisNode, child;   //needed for recursion
337         String nodeText;
338         thisNode = t;
339         nodeText = thisNode.getText();
340 
341         if (nodeText.equals("VARIABLE_DEF")) {
342             AST tmp;
343             tmp = thisNode.getFirstChild(); //gets MODIFIERS node
344             tmp = tmp.getNextSibling(); //gets TYPE node
345             tmp = tmp.getNextSibling(); //gets IDENT (check against condIdent
346             if (tmp.getText().equals(condIdent.getText())) {
347                 tmp = tmp.getNextSibling();
348                 if (tmp != null) {                //gets ASSIGN node
349                     tmp = tmp.getFirstChild();    //gets EXPR node
350                     result = tmp.getFirstChild(); //gets initial initVal
351                 }                                 // (assuming NUM_type)
352                 thisNode = thisNode.getNextSibling();
353                 nodeText = thisNode.getText();
354             }//if
355         }//if
356 
357         //walk forward from VARIABLE_DEF looking for EXPRessions that might
358         // re-valuate the initVal identified in VARIABLE_DEF
359         if (nodeText.equals("EXPR")) {
360             AST tmp;        //subnodes
```

Appendix F (continued)

```
361           String snText;  //subnode text
362           tmp = thisNode.getFirstChild();
363           snText = tmp.getText();
364           if (snText.equals("++")
365              || snText.equals("+=") || snText.equals("=")
366              || snText.equals("--") || snText.equals("-=")
367              || snText.equals("*=") || snText.equals("/=")) {
368                   tmp = tmp.getFirstChild();
369                   if (tmp.getText().equals(condIdent.getText())) {
370                       result = tmp.getNextSibling(); //get initVal
371                   }//if                              // (assuming NUM_type)
372           }//if
373           thisNode = thisNode.getNextSibling();
374       }//if
375
376       //recursive loop
377       while (thisNode != null) {
378           if (thisNode.equalsTree(loopNode)) {  //if loopNode, end search
379                 return result;
380           } else {
381                child = thisNode.getFirstChild();
382                if (child != null) {      //recursive call is on next line
383                    result = findInitVal(child, condIdent, loopNode, result);
384                }//if
385                thisNode = thisNode.getNextSibling();
386           }//if
387       }//while
388
389       return result;
390
391   }//findInitVal
392
393   //Recursive method that walks portion of tree finding an iterator node
394   // within a while/do loop (POST_INC, ASSIGN or PLUS_ASSIGN); initially,
395   // condIdent is the iterator Identifier.
396   //Note: Java 1.3.1 grammar makes no distinction between PRE_INC, PRE_DEC
397   // and POST_INC, POST_DEC in AST trees, so PRE- and POST- are handled
398   // identically in this code.
399   public static AST findIterOper(AST sList, AST condIdent) {
400
401        //declarations and initialization
402        AST thisNode, child, result;
403        String nodeText;
404        result = null;
405        thisNode = sList;
406        nodeText = thisNode.getText();
407
408        //possible iterator expressions
409        if (nodeText.equals("++")
410               || nodeText.equals("+=") || nodeText.equals("=")
411               || nodeText.equals("--") || nodeText.equals("-=")
412               || nodeText.equals("*=") || nodeText.equals("/=")) {
413             child = thisNode.getFirstChild();
414             if (child.getText().equals(condIdent.getText())) {
415                 result = thisNode;
416                 return result;
417             }//if
418         }//if
419
420        //recursive loop
```

Appendix F (continued)

```
421         while (thisNode != null) {
422             child = thisNode.getFirstChild();
423             if (child != null) {
424                 result = findIterOper(child, condIdent); //recursive call
425             }//if
426             thisNode = thisNode.getNextSibling();
427         }//while
428 
429         return result;
430 
431     }//findIterOper
432 
433     //For simple iterator expressions of the form:
434     //     x++, x--, ++x, --x, x+=n, x-=n, x*=n, x/=n
435     // this finds n.  Expressions of the form x = x op n are processed in-
436     // line before this call. ++x and --x are structurally identical to x++
437     // and x-- in the AST tree and are processed in this code identically
438     public static double getIter(AST i) {
439 
440         //Declarations
441         AST tmp = null;
442         double result = 0.0;
443         String s = i.getText();
444 
445         if (s.equals("++")) {
446             //process x++
447             result = 1.0;
448         } else if (s.equals("+=") || s.equals("-=")
449                 || s.equals("*=") || s.equals("/=")) {
450             //process x '+-*/'= n.  Watch out: in odd cases this could
451             //be an expression like: x *= -1;
452             tmp = i.getFirstChild();
453             tmp = tmp.getNextSibling();
454             if (tmp.getText().equals("-")) { //in case of unary minus
455                 tmp = tmp.getFirstChild();
456                 result = -(Double.parseDouble(tmp.getText()));
457             } else if (tmp.getText().equals("+")) { //rare case: unary plus
458                 tmp = tmp.getFirstChild();
459                 result = Double.parseDouble(tmp.getText());
460             } else result = Double.parseDouble(tmp.getText()); //iter found
461         } else {  //process x--
462             result = -1.0;
463         }//if
464 
465         return result;
466 
467     }//getIter
468 
469     //defines 9 possible conditions plus one error condition for direction of
470     //iterator - see comments below
471     public static int iterDirect(String o, double n, double i){
472 
473         int result = 0;
474 
475         //x++ or x += n or x-= -n; also covers x = x + n and x = x - (-n)
476          if ((o.equals("++"))
477              || (o.equals("+=") && n > 0)
478              || (o.equals("-=") && n < 0))
479              result = 1; //iterator moves additively toward +inf
480          //with initVal > 0: x*=n (n>1) or x/=n (0<n<1);
481          // also covers x=x op n equivalents
```

Appendix F (continued)

```
482         else if (((o.equals("*=") && n > 1)
483             || (o.equals("/=") && (n > 0 && n < 1))) && i > 0)
484             result = 1; //iterator moves multiplicatively toward +inf
485         //x-- or x+= -n or x-=n; also covers x=x+(-n) and x=x-n
486         else if ((o.equals("--"))
487             || (o.equals("+=") && n < 0) || (o.equals("-=") && n > 0))
488             result = 2; //iterator moves additively toward -inf
489         //with initVal < 0: x*=n (n>1) or x/=n (0<n<1);
490         // also covers x=x op n equivalents
491         else if (((o.equals("*=") && n > 1)
492             || (o.equals("/=") && (n > 0 && n < 1)))
493             && i < 0)
494             result = 2; //iterator moves multiplicatively to -inf
495         //with initVal = 0: x*=n or x/=n;
496         // also covers x=x op n equivalents - result is 0
497         else if ((o.equals("*=")
498             || o.equals("/=")) && i == 0)
499             result = 7; //iterator never leaves 0
500         //x*=n (0<n<1) or x/=n (n>1); also covers x=x op n equivalents
501         else if (((o.equals("*=")) && (n >0 && n < 1))
502             || (o.equals("/=") && n > 1))
503             result = 3; //iterator converges toward 0 from + or from -
504         //x*=n (-1<n<0) or x/=n (n<-1); also covers x=x op n equivalents
505         else if ((o.equals("*=") && (n > -1 && n < 0))
506             || (o.equals("/=") && n < -1))
507             result = 4; //iterator converges toward 0 from both sides
508         //x*=n (n<-1) or x/=n (-1<n<0); also cover x=x op n equivalents
509         else if ((o.equals("*=") && n < -1)
510             || (o.equals("/=") && (n > -1 && n < 0)))
511             result = 5; //iterator alternates growing toward -inf and +inf
512         //x*=-1 or x/=-1;
513         // also covers x=x op n equivalents - causes iteration from x to -x
514         else if ((o.equals("*=")
515             || o.equals("/=")) && n == -1)
516             result = 6; //iter oscillates: -initVal and +initVal (infinite)
517         //x+=0 or x-=0 or x*=1 or x/=1;
518         // also covers x=x op n equivalents - result is always x
519         else if (((o.equals("+=") || o.equals("-=")) && n == 0)
520             || ((o.equals("*=") || o.equals("/=")) && n == 1))
521             result = 7; //iterator is stuck at initVal (infinite)
522         //special case: x*=0 - becomes 0 in 1 iteration
523         else if (o.equals("*=") && n == 0)
524             result = 8;
525         //special case: x/=0; iterator becomes undefined
526         else if (o.equals("/=") && n == 0)
527             result = 9;
528         //undefined error
529         else result = -1;
530
531         return result;
532
533     }//iterDirect
534
535     //For a loop control identifier, given an initial value, iterator value,
536     // exit value and a direction for the iterator, provide feedback
537     public static void feedBack (double n, double t, double c, int d, String o) {
538
539         //logic which decides if for loop is properly formed
540         //check basic structure of loop
541
```

Appendix F (continued)

```
542       //Warnings:
543       //Use of '!=' operator in an loop exit condition
544       if (o.equals("!=")) {
545           System.out.println(" Warning: using '!=' as the operator for the exit
    condition from a loop");
546           System.out.println("  can cause problems if the loop control variable
    doesn't land right on the");
547           System.out.println("  exit value.  Usually, '<=' or '>=' are
    preferred.");
548       }//if
549
550       //Iterators involving *= or /= that converge toward 0
551       if (d == 3) {
552           System.out.println(" This iterator converges toward 0 from either the
    positive direction or the");
553           System.out.println("  negative direction.  Make sure the relation of the
    initial and exit");
554           System.out.println("  conditions of the loop are appropriate for this
    situation.");
555       } else if (d == 4) {
556           System.out.println(" This iterator converges toward zero, alternating
    between negative and positive");
557           System.out.println("  values on each iteration.  Make sure the relation
    of the initial and exit");
558           System.out.println("  conditions of the loop are appropriate for this
    situation, or choose a simpler");
559           System.out.println("  method of iterating from the start to the finish
    of the loop.");
560       }//if
561
562       //Properly formed loop
563       if ((n < c && (o.equals("<") || o.equals("<=")) && d == 1)
564           || (n > c && (o.equals(">") || o.equals(">=")) && d == 2)) {
565           System.out.println(" Basic structure of loop is okay");
566       //Exit condition boolean operator is the inverse of what it should be
567       } else if ((n < c && (o.equals(">") || o.equals(">=") || o.equals("==")))
568           || (n > c && (o.equals("<") || o.equals("<=") || o.equals("==")))) {
569           System.out.println(" Exit condition operator should be inverse - caused
    loop termination without execution of loop body");
570       //Iterator is headed in the wrong direction
571       } else if (n < c && d == 2 || n > c && d == 1) {
572           System.out.println(" Loop iterator headed in wrong direction!");
573       }//if
574
575       //Drop through and test if entrance and exit conditions are equal
576       if (n == c) {
577           System.out.println(" Loop init and exit are same values!");
578       }//if
579
580       //Values of d below result from unusual iterator expressions and are
581       // most likely semantic errors in the source being examined
582
583       if (d == 5) {
584           System.out.println(" Iterator alternates between negative and positive
    values as it increases");
585           System.out.println("  in absolute value.  Be sure the relation between
    initial and exit conditions");
586           System.out.println("  is appropriate for this situation, or choose a
    simpler way to iterate from the");
```

Appendix F (continued)

```
587           System.out.println("  start to the finish of the loop.");
588       } else if (d == 6) {
589           System.out.println(" Iterator alternates between same neg/pos values -
    infinite loop");
590       } else if (d == 7) {
591           System.out.println(" Iterator is not changing - infinite loop");
592       } else if (d == 8) {
593           System.out.println(" Iterator is of the form 'x *= 0' which goes to 0
    after one iteration");
594       } else if (d == 9) {
595           System.out.println(" Iterator is of the form 'x /= 0' which is
    undefined");
596       }//if
597
598       //Drop through and test to see if there is excessive looping,
599       // at least for +/- loops; arbitrarily set at >1000000 loops
600       if ((d == 1 || d == 2) && Math.abs(c - n) > Math.abs(t) * 1000000) {
601           System.out.println(" This output will execute more than 1 million
    times!");
602       }
603
604       //Other loop tests can be coded here
605
606     }//feedback
607
608     //builds enumeration from walk through AST
609     public static ASTEnumeration findOutputNodes(AST t) {
610         Vector roots = new Vector(10);
611         if (t == null) return null;
612         searchSubtrees(roots, t);
613         return new ASTEnumerator(roots);
614     }//findOutputNodes
615
616     //recursive method called by findOutputNodes()
617     private static boolean searchSubtrees(Vector v, AST t) {
618         AST thisNode, child;
619         //AST varDefChild, varDefNextSibling, identSibling;
620         boolean itemFound = false;
621         thisNode = t;
622
623         //recursive loop
624         while (thisNode != null) {
625             child = thisNode.getFirstChild();
626             if (child != null) {
627                 itemFound = searchSubtrees(v, child); //recursive call
628             }//if
629             if (outHT.contains(thisNode.getText())) {
630                 itemFound = true;
631                 v.appendElement(thisNode);
632             }//if
633             thisNode = thisNode.getNextSibling();
634         }//while
635
636         return itemFound;
637
638     }//searchSubtrees
639
640     //read Java source file(s) and attempt to build AST tree
641     public static AST buildAST(String[] args) throws Exception {
642
643         //declarations and initialization
```

Appendix F (continued)

```
644         String f;
645         AST t = null;  //AST is an antlr class that defines structure of an
646                        // Abstract Syntax Tree
647
648         //testfile is hardcoded for testing purposes - in actual use, test
649         //file is a command line argument and -showtree is a switch to turn
650         //on Java frames which display the AST
651         f = "c:/antlr/SimpleLoop.java"; //<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
652         //showTree = true;//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
653         t = parseFile(f, new BufferedReader(new FileReader(f))); //<<<<<<<
654 /* //this sectioned commented out for testing purposes - see note above
655         try {
656             if (args.length > 0 ) { //if we have at least 1 command-line argument
657                 System.err.println("Parsing...");
658                 for(int i = 0; i < args.length; i++) {
659                     if (args[i].equals("-showtree")) {
660                         showTree = true;
661                     } else {
662                         f = args[i];
663                         t = parseFile(f, new BufferedReader(new FileReader(f)));
664                     }//if
665                 }//for
666             } else
667                 System.err.println("Usage: java IOScan [-showtree] " +
668                                    "<directory or file name>");
669         } catch (Exception e) {
670             System.err.println("exception: "+e);
671             e.printStackTrace(System.err);   // so we can get stack trace
672         }//try
673 */
674         return t;
675
676     }//readSource
677
678     //parse Java source file into AST tree
679     public static AST parseFile(String f, Reader r) throws Exception {
680
681         //initialization
682         AST t = null;
683
684         try {
685             //JavaLexer: antlr class for items to be lexed and tokenized
686             JavaLexer lexer = new JavaLexer(r);
687             lexer.setFilename(f);
688             lexer.setTokenObjectClass("TokenWithIndex");
689             //TokenStreamTracker: antlr class, used here to recognize and
690             // discard whitespace (WS) tokens
691             TokenStreamTracker tracker = new TokenStreamTracker(lexer);
692             tracker.discard(JavaLexer.WS);     //ignore WS (whitespace)
693             //JavaRecognizer: antlr class for building AST trees
694             JavaRecognizer parser = new JavaRecognizer(tracker);
695             parser.setFilename(f);
696             parser.compilationUnit();
697             t = parser.getAST();          //builds AST tree for source file
698             //call method to build visual representation of tree
699             doTreeAction(f, parser.getAST(), parser.getTokenNames());
700         } catch (Exception e) {
701             System.err.println("parser exception: "+e);
702             e.printStackTrace();   // so we can get stack trace
```

Appendix F (continued)

```
703         }//try
704
705         System.out.println("AST tree building done");
706
707         return t;
708
709     }//parseFile
710
711     //build a visual representation of the AST tree in a frame
712     public static void doTreeAction(String f, AST t, String[] tokenNames) {
713         if (t==null) return;
714         if (showTree) {
715             ((CommonAST)t).setVerboseStringConversion(true, tokenNames);
716             ASTFactory factory = new ASTFactory();
717             //TreeParser tp = new TreeParser();
718             AST r = factory.create(0,"AST ROOT");
719             r.setFirstChild(t);
720             final ASTFrame frame = new ASTFrame("Java AST", r);
721             frame.setVisible(true);
722             frame.addWindowListener(
723                 new WindowAdapter() {
724                     public void windowClosing (WindowEvent e) {
725                         frame.setVisible(false); // hide the Frame
726                         frame.dispose();
727                         System.exit(0);
728                     }//windowClosing
729                 }//WindowAdapter
730             );//addWindowListener
731         }//if (showTree)
732
733         JavaTreeParser tparse = new JavaTreeParser();
734         try {
735             tparse.compilationUnit(t);
736         } catch (RecognitionException e) {
737             System.err.println(e.getMessage());
738             e.printStackTrace();
739         }//try
740     }//doTreeAction
741
742     //hash java output methods (57 of them in Java 1.3.1)
743     public static void hashOutputMethods() throws Exception {
744         String oNode = "";                //holds each currently read method
745         Integer hVal;                     //used to generate hash code
746
747         BufferedReader inf = new BufferedReader(new
    FileReader("c:/antlr/outputMethods.txt"));
748         while (inf.ready()) {
749             oNode = inf.readLine();
750             hVal = new Integer(oNode.hashCode());
751             outHT.put(hVal, oNode);
752         }
753         inf.close();
754         System.out.println("Hashing of 57 Java output methods complete.");
755     }//hashOutputMethods
756
757     //hash Java loop keywords (for, while, do)
758     public static void hashJavaLoops() throws Exception {
759         String block = "";              //holds each currently read command
760         Integer hVal;                   //used to generate hash code
761
```

Appendix F (continued)

```
762         BufferedReader inf = new BufferedReader(new
    FileReader("c:/antlr/blocks.txt"));
763         while (inf.ready()) {
764             block = inf.readLine();
765             hVal = new Integer(block.hashCode());
766             blockHT.put(hVal, block);
767         }
768         inf.close();
769         System.out.println("Hashing of for, while, do complete.");
770     }//hashJavaLoops
771
772     //hash Java language tokens (163 of them)
773     public static void hashTokens() throws Exception {
774         String token = "";                //holds each currently read token
775         Integer hVal;                     //used for hashcode
776         int hV = 0;
777
778         BufferedReader inf = new BufferedReader(new
    FileReader("c:/antlr/tokens.txt"));
779         while (inf.ready()) {
780             token = inf.readLine();
781             hVal = new Integer(Integer.toString(hV));
782             tokensHT.put(token, hVal);
783             hV++;
784         }
785         inf.close();
786         System.out.println("Hashing of 163 Java tokens complete.");
787     }//hashTokens
788
789 }//class IOScan
```

Appendix G: Resources for Information on Semantic Parsing

[1] antlr home page: http://www.antlr.org/

[2] Tutorials on getting started with antlr: http://www.antlr.org/doc/getting-started.html

[3] Sun articles on Java parsers (series) – 1997. Sun's general introduction to antlr and yacc as tools for parsing Java source code:
http://developer.java.sun.com/developer/technicalArticles/Parser/SeriesPt1/
http://developer.java.sun.com/developer/technicalArticles/Parser/SeriesPt2/
http://developer.java.sun.com/developer/technicalArticles/Parser/SeriesPt3/

[4] Clark, Chris (1999): Build a Tree – Save a Parse. *ACM SIGPLAN Notices*, **34**(4): 19-24. A periodic column on various practical issues concerning parsing.

[5] http://www.netaxs.com/people/nerp/automata/syllabus.html. Outline and content for a complete course on language grammars, BNF notation, including information on semantic parsing and various kinds of parsers.

[6] http://en.wikipedia.org/wiki/Main_Page. The following search terms provide a series of articles on topics related to semantic parsing: "semantic analysis", "parser", "lexer", "top-down parsing", "bottom-up parsing", "context-free grammar", "Chomsky", "LL parser", "LR parser", "Backhus-Naur form"

[7] https://javacc.dev.java.net. The home page for javacc, Sun's own parser generator for Java.

[8] https://javacc.dev.java.net/servlets/ProjectDocumentList?folderID=110. Download site for a variety of grammars that work with javacc (27 languages including C, Java and Visual Basic, Python and Oberon).

[9] http://home.earthlink.net/~slkpg. Home page for SLK, which claims to be "the only true LL(k) parser" and "the only known near-solution to this NP-complete problem".

[10] http://www.gnu.org/software/bison/bison.html. GNU home page for Bison, a parser generator that has been available for UNIX and Linux systems for many years.

[11] http://dinosaur.compilertools.net. An introduction to, and tutorial for, standard UNIX tools, lex and yacc, as well as an introduction with links to a variety of other related tools.

[12] http://cedet.sourceforge.net/info/semantic.html. Semantic: technically a 'bovinator' which is a partial lexer. This is a long article that describes 'bovination', BNF grammars and how to work with the Semantic product.