JAVA SOURCE CODE ANALYSIS TOOL

By

DAVID P. DARUS

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

I dedicate this thesis to the glory of God, the Creator, Sustainer, and Redeemer of all things and to all Java programmers who desire to write robust software.

ACKNOWLEDGMENTS

I thank my committee, Drs. Joseph N. Wilson, Douglas D. Dankel, and Richard E. Newman, and employer, Gainesville Regional Utilities, for providing me with an excellent opportunity to advance my education.

TABLE OF CONTENTS

page

APPENDIX

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

JAVA SOURCE CODE ANALYSIS TOOL

By

David P. Darus

December 2002

Chair: Joseph N. Wilson
Major Department: Computer and Information Science and Engineering

The task of a compiler is to produce executable code from source code. To do this, all ambiguities of identifiers, expressions, and data types must be resolved. The generated code will reflect what is described in the source program. In general the intent of the programmer is unknowable, so it is proper for a compiler to limit itself to this task. Programming errors can be made that are not detected by a compiler as long as all the syntactical and semantic rules of the language are followed. Detecting these errors involves delving into the area of programmer intent. Therefore, no scheme can unequivocally detect and report these types of errors. Typically, these types of errors are exposed during the testing and debugging of source code.

There are various mechanisms employed by programmers to accomplish the task of debugging. Oftentimes the programmer has no aid in pointing out code that contains probable errors or is otherwise suspicious. The desire to produce robust code, code that is less likely to fail during runtime due to a coding error, points to the need for additional

source code analysis. This type of analysis goes beyond the syntactical and semantic analysis necessary for compilation. This analysis goes towards exposing code that follows common patterns of incorrect code. Just as a compiler is specific to a given language so must such a program analysis tool be language specific. In this case Java is the language of interest and JSCAT (Java Source Code Analysis Tool) is the tool I propose to aid in exposing suspicious code patterns to the programmer. Although JSCAT is specific to Java, there are elements of its architecture that can be used in producing a similar tool for other programming languages. This paper describes JSCAT, its internal structure, capabilities, limitations, and extendibility.

JSCAT works directly with program source code but assumes that the source code compiles correctly using standard Java compilers. A cornerstone of JSCAT is that the knowledgeable user can incorporate analysis checks beyond those that are provided. This is important because in general the intent of a program cannot be algorithmically determined. By extending JSCAT, a particular suspicious code pattern can be detected and reported. JSCAT can be applied to many different purposes, such as aiding a programmer to write very robust code, teaching a new programmer, or analyzing source code from some provider to determine how robust and error prone it may be.

JSCAT provides a very extensible framework for processing source code that is represented internally in the form of an abstract syntax tree. The framework constructs and directs the processing of the abstract syntax tree. Several information sources and mechanisms are employed to allow analysis checks to interrogate the abstract syntax tree for suspicious code patterns as it is being processed.

CHAPTER 1
INTRODUCTION

A compiler is limited to translating a source program in one language into an executable program of another, lower level language. It must insure that proper syntax is used in the program and no semantic ambiguities exist. A programmer may write a syntactically and semantically correct program and yet have deficient code. Such code deficiencies are errors in logic or code that, although they do not cause a compiler error, will cause the program to operate differently from what was intended. This type of code deficiency is in the realm of poor coding practice. It is beyond the scope of a compiler to expose such deficiencies. In general, a program cannot be written to determine what another program is doing much less what it is supposed to do. The compiler is concerned only with correctly implementing the one-to-one mapping of the source program into its corresponding executable program.

Leaving poor code uncorrected leads to unstable programs that exhibit runtime errors. To produce robust code, code that accurately and sufficiently handles runtime exception conditions, such poor code must be corrected. This is commonly left to the programmer to do without any assistance. Though programmer intent cannot be algorithmically determined, there are definite patterns for a given language that reveal poor coding practices with a high degree of certainty. These patterns fall into two broad categories. The first pattern is local to a particular token, statement, or expression. The second pattern encompasses a larger scope of the source code and is often related to the structure of language constructs. This is most typically apparent with either missing constructs or

minimal constructs. A minimal construct is one that provides enough structure to evade syntactical and semantic compiler errors but does not provide code logic to properly deal with the situation for which the construct is intended. An example of this would be a `try/catch` construct in Java where the `catch` clause has an empty block of statements. In this case, the exception is caught so that the program does not abort but it is not handled in an effective way. No information is produced to aid in diagnosing the exception and no recovery of the exception is done. A tool is needed to assist programmers in detecting code that is likely to be deficient and offer information on how to correct the suspicious code.

One way to address the problem of detecting suspicious code is to make assumptions of intent about code patterns and offer suggested diagnoses and remedies. Such a tool will be specific to a given language, and will deal with some common code patterns in that language that have been deemed to be prone to error. A superior tool will be able to be extended by the user of the tool to detect and remedy suspicious code patterns not known to the original tool developer. These code patterns may be applicable only to a certain type of application or for a particular programming group.

This thesis focuses on solving this problem for the Java programming language. JSCAT (Java Source Code Analysis Tool) is proposed as the automated means to aiding the programmer in producing robust code. In JSCAT, many code patterns are analyzed for suspicious code. The tool offers a wide array of facilities to recognize and investigate suspicious code patterns. Thus, the user of the tool can extend JSCAT to meet specific needs.

JSCAT is designed to parse source code into an abstract syntax tree. The abstract syntax tree is processed in a pre-order fashion to detect and report on code patterns that match specified analysis checks. To parse the source code an LALR parser is constructed using the JavaCUP programming tool and a grammar for the Java 1.2.2 language specification. The parser constructs a special tree data structure corresponding to the syntactical source code elements that form the abstract syntax tree. The abstract syntax tree is processed in a pre-order fashion by visiting each node in the tree starting with the root node. As each node is visited, any analysis checks that are registered to trigger when that type of node is encountered are then executed. This design allows for many analysis checks to trigger on a given node type but does limit the triggering event to the recognition of only one node type, regardless of context.

Facilities are provided to determine the context of the node occurrence so that additional filtering of the triggering event can be accomplished. Facilities are also provided to determine if other code pattern elements and code specifics are absent or present so that further determinations can be made as to whether a suspicious code message should be reported or not. In brief, these facilities include a symbol table for the fields and local variables declared in the source code, multiple pieces of source code context information, global data created and used by different analysis checks, and tree traversal capabilities. The analysis checks produce messages that report suspicious code and can additionally provide remedies for the suspicious code.

This thesis presents previous related academic and commercial work in Chapter 2. Next the JSCAT framework and its extensibility is described in Chapter 3. All aspects of the abstract syntax tree (AST) are described in Chapter 4. Chapter 5 discusses the

information facilities, analysis check registration and triggering, JSCAT configuration and messages, and analysis checks. Examples of analysis checks are provided as a part of the discussion. Closing remarks about JSCAT and future work are discussed in Chapter 6. Appendix A contains a list of all the system-provided analysis checks categorized by type. Appendix B contains an example program and corresponding output from JSCAT to illustrate the suspicious code patterns detected.

CHAPTER 2
BACKGROUND

Producing robust code is a difficult task. Several approaches have been explored to aid the programmer. A brief overview of these approaches is presented here to focus in on where JSCAT fits.

Verification is a rigorous way to produce correct programs. "[It uses] abstract specifications of the code with model checkers or theorem provers/checkers to check that the specification is internally consistent" [1:1]. This approach is quite difficult and impractical for most systems. To date it has not been widely adopted, mainly because proving correctness and writing robust specifications are very difficult. The Bandrea system [2] extracts rules from Java source code in an effort to address these difficulties.

Testing is the most common means of debugging programs. However, this requires a large effort to fully test a non-trivial program. A debugger is the standard tool for finding bugs during the execution of a program. Other tools have been proposed to model execution of programs such as PREfix [3] and to automate the use of a debugger type tool as does Diduce [4].

Manual inspection of source code can consider all semantic levels but is unreliable and tedious. Large programs have deep, complex code path. Reasoning about a single path can take anywhere from a few minutes to hours.

A distinct approach from those mentioned is static source analysis. Many techniques have been explored within this approach. These techniques attempt to address the

problems that verification, testing, and manual inspection present. An overview focusing on static source analysis is presented here.

The language-based technique uses the language itself to prevent the writing of poor source code. All programming languages by definition use this technique and probably provide the greatest single source of preventing and finding bugs. Often this is done by a strong type system in the language [5-8]. While this technique is useful, it is limited in its ability to express complex relationships. Furthermore, new language adoption has been haphazard in the past.

Some projects have used higher-level compilation to hard wire application-level information into compilers [9-10]. This technique can be effective for a given system, but requires much effort that is not transportable to another system.

A relaxed verification type approach is embodied in source code specification annotations. Such systems as ESC [11] and LCLint [12] stop short of program correctness proving but provide a more practical matching of specifications with source code. However, this technique burdens the programmer with writing specification in a separate language. The most recent tool is Extended Static Checking for Java (ESC/Java) from Compaq's Systems Research Center, which operates upon Java source code. ESC/Java uses program verification technology, but appears to operate like a type checker. It is more semantically thorough than decidable static analysis techniques. Common programming errors are detected by matching the annotation language, in which programmers express design decisions using lightweight specifications, with the source code. For example, one may give a method precondition that says that a parameter is not null, or declare an object invariant that says that an integer field lies

between zero and the length of some array field. ESC/Java has very sophisticated analysis checks and is extensible. However, the extension is done through a new annotation language imbedded in Java source code comments. This requires that the Java programmer learn this specification language to use the program.

JSCAT fit best within the extensible compilation technique of static source code analysis. This technique uses an AST to represent the source code and traverses the AST looking for suspicious code patterns. Many tools like Lord's Ctool [13], Crew's Prolog-based ASTLOG [14], and the products Anti-C/JLint, Sureshot's Jive, and Webgain's Audit Module utilize this technique with varying degrees of functionality and extensibility.

Each of these products is presented here in regards to their own strategies, strengths, and weaknesses. The JSCAT approach differs in varying degrees to these approaches; its differentiating characteristics are presented.

Surveying these product implementations reveals divergent design approaches to the problem of static source code analysis. The first choice is whether to use source program code or executable program code as the input. Secondly, a choice is made whether to look for specified bad code patterns or to match a program specification to the program code. Thirdly, a choice is made as to whether the analysis checking can be extended with customized analysis checks and if so, what language and facilities are available for writing the customized analysis checks.

AntiC/JLint [15] by Konstantin Knizhnik of the Research Computer Center at the Moscow State University is two separate programs. The AntiC program parses source code to look for suspicious code. What constitutes suspicious code is based on common

problems in the C programming language such as operator priorities, absence of breaks in switch code, and wrong assumptions about what is the body of certain constructs (i.e., the dangling else problem) to name a few. Since Java is very similar to C, most of the analysis done on a C source program applies to a Java source program. The suspicious code patterns that are detected by AntiC are coded into that tool; there is no facility to extend the tool with additional analysis checks. This limitation is a weakness that the JSCAT approach overcomes.

The JLint program is specific to Java. Its purpose is to find bugs, inconsistencies, and synchronization problems. JLint performs its analysis on a Java class file, not on the source code. This approach is taken because the Java class file format is easier to parse and less likely to change. JLint's strength lies in the sophisticated analysis done. The analysis is done via data flow analysis and the building of lock dependency graphs. Many of the data flow and synchronization checks are beyond the scope of JSCAT, though they could be added as a major enhancement. The JLint program is limited by its inability to be extended with new types of analysis checks.

Sureshot's JiveLint [16] implementation analyzes Java source code for a pre-defined set of analysis checks. It is similar to the AntiC program yet has more specific Java analysis checks. Again, this implementation has no facility to add additional analysis checks.

"[Webgain's Quality Analyzer Audit module] performs static analysis of Java source and class files to find and report errors of style and potential problems related to performance, maintenance, and general robustness" [17:Ch.4, Pg.1]. It provides more than fifty prepackaged rules and provides an API for fast and easy creation of custom

rules. "The APIs provide complete information on both the syntactic and semantic aspects of the Java code being audited, making it easy to develop complex rules" [17: Ch.4, Pg.1]. Audit works incrementally, reprocessing only the statements that have changed. It also is able to detect errors when the Java source file does not yet compile. Audit constructs a tree representing the Java source code text. On each node, well-known methods are called to enforce rules. Audit is extended by the user sub-classing specific nodes and overriding the well-known methods with customized code.

JSCAT is differentiated from AntiC/Jlint and JiveLint by providing a framework for adding additional analysis checks. The user of JSCAT writes the analysis checks in Java for his or her specific needs. Whereas ESC/Java is extendible, JSCAT is different in that the user does not have to learn yet another language as part of the development process. Also, JSCAT does not match a specification of what the source code is supposed to do with what is actually coded. The Webgain Audit approach is similar to JSCAT, but with one fundamental difference. Webgain Audit provides extensibility through an inheritance relationship by sub-classing tree nodes with user provided implementations. This approach by Webgain is less orthogonal than JSCAT. Each node type that is extended must implement all analysis checks that are triggered by that node type in the extended class. JSCAT is extended by an association relationship between the tree node type and the analysis check. JSCAT provides a more granular mechanism for coding different analysis checks and it also provides more information facilities to aid in determining source code context.

Each of these other approaches has strengths that are beyond the scope of JSCAT. They are useful in helping the Java programmer detect potentially incorrect code. JSCAT

concentrates its efforts on assisting the Java programmer in developing source programs. Such a programmer can use Java to extend the functionality of the analysis checker and does not need to learn any other new language to do so. As the Java language changes or as the specific task environment of a Java programmer changes, JSCAT can be extended to detect and report on newly discovered suspicious code patterns.

A very interesting technique that offers the greatest competition to JSCAT comes from the Stanford Compiler Group. They offer a meta-compilation technique [1, 18-19] to perform static source code analysis. Their technique uses a state-machine data structure for the source code and a language, *metal*, for writing analysis checks. They extend a compiler, which has the infrastructure to do source code analysis, to build state machines. As the extended compiler is processing the source code any analysis checks matching the input code pattern are executed. The analysis checks detect rule violations by transitioning between states that allow or disallow other operations. JSCAT could be augmented with control-flow and data-flow data structures to accomplish many of the goals of the meta-compiler technique. JSCAT also has the ability to annotate nodes with attributes as described in the Global Data Facility section. This is one thing that the meta-compiler cannot do.

CHAPTER 3
EXTENSIBLE FRAMEWORK

The JSCAT framework can be summarized as a process that transforms a source file into an abstract syntax tree (AST), visits each node in the AST, and performs actions in user extensible analysis checks that are registered with a particular AST node type. JSCAT operates upon one source file at a time, parsing to build an AST. This AST is visited in a pre-order tree traversal fashion starting with the root node. The root node represents the entire Java source file's compilation unit. As each AST node is visited any analysis checks that are registered with that particular node type are triggered to perform actions. These actions are the system-provided and user-written methods that analyze the source code starting with the triggering node. The analysis checks are coded to detect suspicious code patterns and report on possible coding problems that are specific to a particular AST node and its context. JSCAT provides many information facilities to aid the analysis check methods in determining if a particular suspicious code pattern exists. Each AST node has specific information related to its type and occurrence as transformed from the source file. Furthermore, JSCAT tracks the progress and context of each AST node visit. This information can be helpful in determining the larger context in the source file that a particular AST node is in. JSCAT, as a major design goal, provides a mechanism for user written analysis checks to be added to the processing of an AST. In addition to this, there are other process-controlling features that provide a high degree of granularity in analyzing a given AST. Figure 3-1 is a high-level view of the components of the JSCAT framework and their information and invocation relationships. The oval

shaped components signify information structures. The rectangular shaped components signify processes. The solid arrow lines signify the source and destination of invocations made by the components. The dotted arrow lines signify the flow of information from one component to another. The bold arrow lines signify the main process of visiting each AST node and calling each registered analysis check associated with each AST node type. Several of the processes draw upon the settings to affect JSCAT execution and output as indicated by an asterisk.

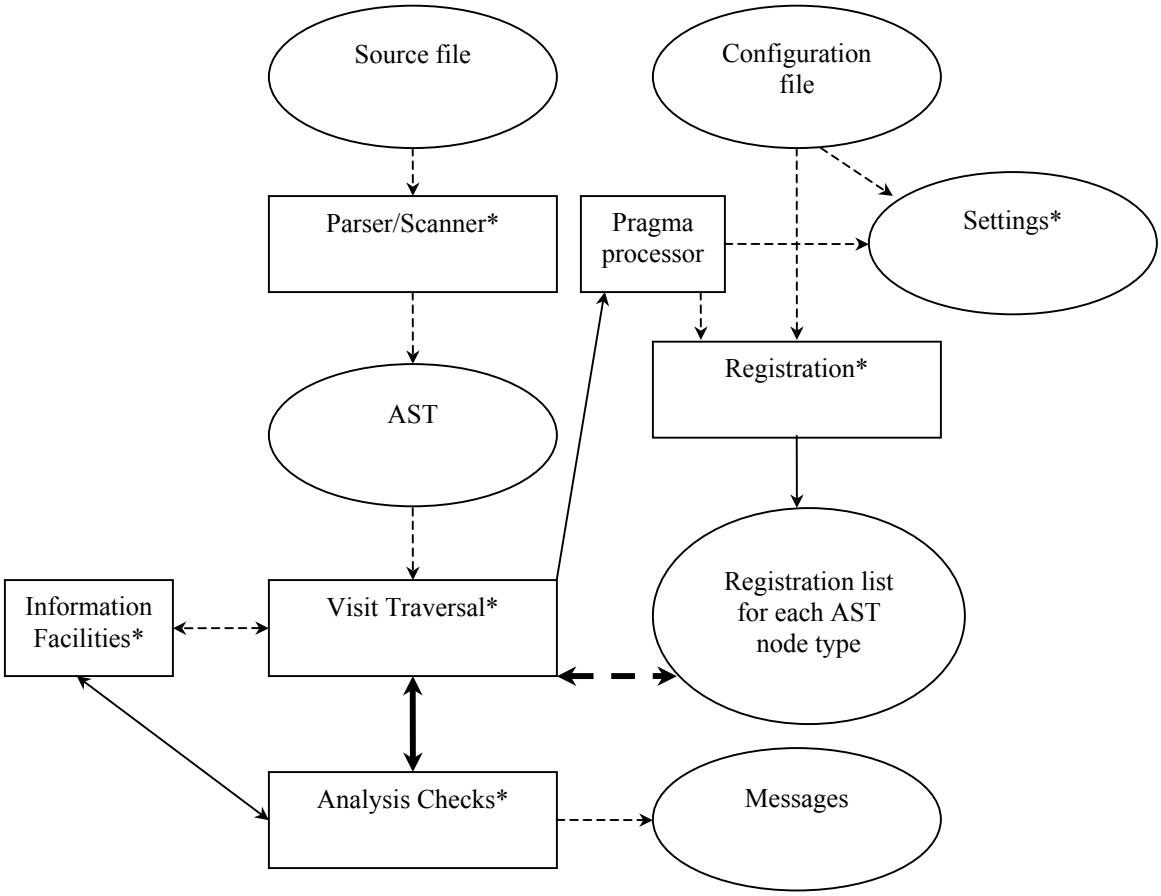

Figure 3-1 JSCAT Framework

A major goal of JSCAT is for its design to allow and promote user extensibility. This goal is driven by the principle that a programmer's intent cannot be obtained from the source code. Thus, all the source code analysis that may be appropriate for all possible

source code cannot be known in advance. Allowing the user of JSCAT to add new analysis checks provides a mechanism to provide specific solutions to specific problems. This extensibility is accomplished through Java classes that implement particular coding requirements and are registered with AST node types that trigger the analysis check actions to be done. Users of JSCAT also have control of turning individual analysis checks on or off at a global and statement-by-statement level. This allows for a customizable analysis check of particular source code. This capability is beneficial for eliminating messages that a particular users wishes to ignore.

Several characteristics of the JSCAT design provide for a powerful general–purpose extensibility framework. These characteristics include limited loss of data in the transformation from source code text to the AST, AST node information, AST navigability, AST processing traceability, a global data facility, analysis check inclusion, exclusion and ordering configuration file, the expressive power of the Java language, and strategically placed user exit points. The combination of these characteristics provides the powerful mechanisms to write essentially any analysis check desired.

Limited Loss of Data

The transformation of the program source code text into the AST structure via an LALR parser of the Java language has limited loss of data. The grammar used and the simplifications made when aggregating syntactical elements preserve most of the information from the source code. For all lexemes in the source code text, the line and column position is retained as well as the character sequence comprising the lexeme itself. To minimize unnecessary tree overhead some nodes incorporate multiple syntactical elements. This reduces storage requirements, tree processing, and simplifies tree navigation. The most extensive aggregation of source code tokens into a single AST

node occurs with Java modifiers. The Java language syntax allows access and specification modifiers in several different places. Where these modifiers are allowed, many can be specified in various orders. These Java modifiers are aggregated into a single AST node but the order and text position of each modifier present is maintained. The information that is lost in the transformation consists of comments and some formatting of aggregated lexemes. For example, the spacing between the left and right brackets of an array dimension lexeme is lost.

Node Information

Each AST node has associated information that is pertinent to what the node represents. For example, the field declaration node has information about the modifiers present, the data type of the field, the field symbol, and any initialization information. More specifically, the field declaration node, `FieldDecl`, has the following methods associated with it. The `getMods` method returns an object representing the modifier present. The `getType` method returns an object representing the field's data type. The `getVarDeclList` returns a list of the field symbols and any initialization information declared. The `getSemicolonToken` returns the semicolon token object. Since the `FieldDecl` node is the beginning of a unit, as defined in the JSCAT Information Facilities section, there is an `isBeginUnit` method and `getTypeName`, `getUnitName`, and `getScopeName` methods. A complete description of the methods available to each AST node type is in the JSCAT program documentation. The information associated with each node coupled with the limited loss of information during tree construction forms the basis of the extensible power of the JSCAT framework for user written analysis checks.

AST Navigability

The JSCAT framework provides various mechanisms for navigating the AST corresponding to a given source code text. During the pre-order processing of an AST, it is traversed based on the special structure of the AST for a given source code text. This navigation from the root node through all its children nodes is the automatic visiting of each node provided by JSCAT. Within an analysis check other navigation mechanisms can be employed to investigate the structure of a given AST. By calling methods that return a reference to AST nodes associated with a given AST node, the AST can be traversed in any way the programmer wishes. These methods mirror the grammar that is used to parse the Java source code. Each AST node has a reference to its parent node. Navigation from a given node up towards the root and possibly down another branch can be accomplished by calling methods that return a reference to another AST node or through the *iterator* mechanism. Each AST node provides an *iterator*, a mechanism to get each of its child nodes in sequential order. These other navigation mechanisms can be used by an analysis check to aid in determining the source code context in which a particular AST node exists. This navigation capability is key in providing extensible power to the JSCAT framework. The various navigational mechanisms allow for the entire AST to be traversed under user analysis check control and thus proper logic can determine the suspicious and non-suspicious code patterns starting at a particular node.

AST Traceability

During the processing of nodes in the AST, information is saved in stacks and lists providing a trace of what processing has been done. These information sources can be utilized to aid in determining the source code context of a triggered node. The JSCAT Information Facilities section describes in depth these information sources. Here they are

described briefly. The order in which AST nodes are visited is maintained in a list. As each node is visited it is appended to the end of the list. Symbol tables are maintained for local variables and fields within a Java class or interface declaration. The symbols tables maintain the symbol names, corresponding data types, and variable scope visibility information. Stacks are maintained as Java class and interface declarations, members, and variable scopes are entered and exited. These facilities for traceability provide another means, which analysis checks can employ to determine AST node context and thus suspicious code patterns.

Global Data Facility

Global data facilitates the saving and accessing of values across the processing of multiple AST nodes. Such a facility allows analysis checks to discover code pattern contexts as large as the AST itself. By this means JSCAT is extensible beyond the context of the triggering nodes of the analysis checks. Global data is analogous to variable storage in a computer program and can be used as such. Global data can be used for a variety of purposes. One such use is to save an AST node reference and retrieve it later for a tree traversal starting point. This demonstrates how the combination of extensibility characteristics forms a powerful and expressive framework. A particularly interesting use of global data is as a mechanism to annotate AST nodes with synthesized and inherited attributes. This can be accomplished by using a node's unique object identification as the key value for the global data entry and the desired attribute value as the value associated with that global data entry key.

Analysis Check Inclusion, Exclusion and Ordering

JSCAT is designed such that the same AST node type can trigger the processing of multiple analysis checks. The analysis checks are processed in the order in which they

are registered with the trigger node type. If an analysis check is already registered with an AST node type then the subsequent registrations are ignored. A configuration file allows for analysis checks to be included or excluded from registration if so desired. The configuration file also specifies the order in which analysis checks are registered. The order of the processing of the analysis checks triggered by a given AST node type may be important to avoid conflicts.

Analysis Check Use of Java

Java serves as the language to describe the analysis checks. Alternatives proposed were a special scripting language, Extensible Stylesheet Language Transformations (XSLT), and ASTLOG [14]. The scripting language could have had the benefit of simplifying the coding of analysis checks because of high-level constructs dedicated to the task. However, this also implies the development of a special language and the user would have to learn this single-purpose language to write analysis checks. XSLT may have provided the same benefit as a special scripting language and mitigated some of the development and learning costs. In essence the problem being solved by JSCAT is to transform a given AST into another AST by translating suspicious code into what is considered the actual intent of the code. This is precisely the task of XSLT. However, XSLT was also rejected because of the limited expressive power in the language to efficiently handle the potential realm of analysis checks. ASTLOG was similarly rejected because a new language would need to be learned by the user. ASTLOG is better suited for processing the AST in ad-hoc ways. The JSCAT framework only needs to navigate the AST in a pre-order fashion. Analysis checks may navigate the AST in an ad-hoc fashion but this is best done with the JSCAT and Java provided mechanisms.

Since Java programmers are the intended users of JSCAT, they will already know how to program using Java. Using Java to analyze Java source code is a natural choice.

User Exits

In addition to the processing of an analysis check before and after an AST node is visited in which the check is registered, there are other strategically placed user-exit points. User-exit objects are Java classes that implement a well-known method. The JSCAT framework has several user-exit reference variables that can be assigned an instance of a user-exit class. At the user-exit points in the framework the well-known method of the referenced user-exit object for that particular user-exit point is called. The method can perform whatever logic is desired and appropriate for a particular user-exit point. User-exit points occur at the appending of AST nodes to the visit list, and the pushing and popping of AST nodes on the various source code context-tracking stacks. The visit list and the context-tracking stacks are described in the JSCAT Information Facilities section below. User-exits provides yet another facility to finely control the processing of an AST and adds to the extensibility of the JSCAT framework.

The extensibility of JSCAT is built upon these characteristics. Full node information, navigability, traceability, global data, processing control, and the expressive power of Java are the foundation upon which nearly any analysis check logic can be written.

CHAPTER 4
PARSING AND AST CONSTRUCTION

JSCAT parses a single Java source code text file into an AST using an LALR parser and associated scanner built with the JavaCUP [20] and JavaFLEX [21] tools. The parser implements a customized grammar of the Java 1.2.2 language specification that builds a syntactically correct AST for processing in the JSCAT framework.

JSCAT works on the premise that the source code to be analyzed is available and compiles using a standard Java compiler. Also, any class files referenced in the source code are also available. This requirement is reasonable because this tool goes beyond compiling a program and if a program can be compiled then the class files must already be available. This tool is mainly for programmers who are developing or otherwise have access to Java source code in need of analysis. Other tools such as JLint [15] are available to analyze Java class files for similar suspicious code patterns.

Analysis is done on one Java source code file at a time. Each invocation of JSCAT processes one particular source file. This is deemed sufficient because JSCAT's purpose is to reveal suspicious code patterns to the user. The user in turn should carefully analyze the reported problems and correct them. Processing multiple files at a time could overwhelm the users with numerous messages.

Parsing

JSCAT parses Java source code into an abstract syntax tree using an LALR parser constructed by JavaCUP, a scanner constructed by JavaFLEX, and a Java 1.2.2 language grammar. JavaCUP is the moniker for Java Based Constructor of Useful Parsers.

> CUP is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC (S. C. Johnson, "YACC - Yet Another Compiler Compiler", CS Technical Report #32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.) and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers, which are implemented in Java. [20:1]

JavaCUP produces source code to implement the LALR parser corresponding to the specified grammar. This source code is compiled with access to the JavaCUP runtime classes to produce a Java class file for the parser. These class files are packaged with JSCAT classes to form the complete tool. At the appropriate place the parser class parse method is called to transform the Java source code file into an AST.

The JavaFLEX programming tool is used in conjunction with JavaCUP. JavaFLEX is the Fast Lexical Analyzer Generator for Java developed by Gerwin Klein. Though they are separate tools and both can be used with other parsers and scanners respectively, JavaFLEX has special provisions to work easily and efficiently with JavaCUP. The JavaFLEX specification describes the scanner portion of the JavaCUP LALR parser to be built. The parser at appropriate places calls the scanner. The task of the scanner is to read one character at a time from the Java source file and build a string of characters into a recognizable token. Since JavaFLEX scans Java source code, Unicode characters are supported. JSCAT needs to retain the exact source representation of the tokens returned by the scanner, so Unicode character encoding has special requirements. To parse Unicode characters that may appear in the Java source, the JavaFLEX Unicode specification is slightly different than when constructing a scanner for a compiler. In the case that a compiler is being written, the Unicode characters in the source code are converted from their textual representation to their binary representation. In the case of JSCAT the textual representation is retained so that it can be analyzed on a character-by-

character basis if necessary. For example, the character sequence \u0041 in Unicode is the capital letter A. The scanner for a compiler would transform this six-character sequence into the 16-bit binary representation for the hexadecimal value forty-one (sixty-five in decimal). The scanner specification for JSCAT has been modified such that the six-character sequence text representation is retained as the token's lexeme value. The JavaFLEX has been further modified for use by JSCAT to encapsulate tokens into corresponding JSCAT objects. Source code line and column position information is determined by the scanner and passed along to the parser, which in turn provides that information when JSCAT objects are formed that correspond to the syntactical elements. Additionally, the scanner was modified to recognize an alternate syntax for comments. This alternate syntax is to support the Pragma feature of JSCAT.

A grammar for Java 1.2.2 provided by JavaCUP served as a starting point for the grammar used by JSCAT. The Java language grammar specification has been modified to construct a special abstract syntax tree. The tree structure is reflective of the parent and child relationship of each production in the grammar specification of Java. At appropriate grammar productions, JSCAT objects representing node types are instantiated and added to the AST. Each node of the tree and can have zero or more children, so this structure is preferred over a strict order structure such as left-child, right-sibling. No advantage is gained with a strict ordering and such an ordering would add tree construction and navigation overhead. The AST is formed with nodes that represent syntactical elements of the source code. A given node may contain a reference to other nodes in the tree as representative of the grammar production formed. To minimize unnecessary tree overhead some nodes incorporate multiple syntactical elements. This

reduces storage requirements, tree processing, and simplifies tree navigation. Each lexeme and its location in the source code is captured in either a separate node or incorporated into a node. In some cases placeholder nodes are added to provide a consistent grammatical structure to the AST. Such nodes typically signal a path in the tree of related elements and make a convenient trigger point for a code pattern.

The JavaCUP grammar specification provides the root node of the AST once the source code has been parsed. Additionally, the specification provides access to the package name associated with the source code that is parsed and a list of package names that are imported into the source code. These pieces of information are quite useful for symbol name resolution, particularly for symbol names that are not declared in the source code.

AST Structure and Class Hierarchy

The special structure of the AST begins at the root node, which represents the entire source code's compilation unit and ends with leaf nodes that represent source code tokens. These nodes and all the nodes in between inherit from a common super-class. This super-class retains the beginning and ending line and column number for what the node represents. In the case of a single token this information identifies the source code position of that single lexeme. In the case of a node that contains other nodes, such as a block of statements, this information identifies the source code beginning and ending position of the block. Each node except the root node has a parent and this information is also retained in the super-class objects. Whereas the references of nodes within nodes starting from the root node provide the connection of nodes from top to bottom, the parent reference provides information such that the connections from leaf nodes to the

root are formed. A detailed description of the AST structure is provided in the next section.

The AST structure is a special tree that corresponds to the Java 1.2.2 grammar used to parse Java source code files. A node represents an element in the AST data structure, whereas a class represents the implementation of a node. The AST's root represents the entire compilation unit of the source code file and branches into all the elements of the source code, ending with leaf nodes representing specific token lexeme's from the source code. The Java class hierarchy to implement the AST has a common parent called the `ASTNode`. Other classes are a part of the hierarchy to categorize AST nodes into related nodes. This section describes in detail the AST structure and the implementation in Java classes.

The `ASTnode` class contains fields and methods common to all nodes in the AST. In particular, a reference to the parent node of a node is maintained – allowing a path from a node up towards the root, the beginning and ending line, and column positions that the node represents. If a node has children then a list of references to those nodes are maintained. The method to visit each node in the tree is placed in the `ASTNode` class. It appends to the *visit list* what node is being visited, polymorphically dispatches to the child node to perform any node specific pre-actions, and then calls the pre-action methods of all analysis checks registered with the child node type. Then any nodes in the child list are visited. Upon completion of visiting any and all children, then the post-action methods of all analysis checks registered with the child node type are called and finally any node specific post-actions specified by the child node type are processed through a polymorphic dispatch.

The AST nodes are broadly divided into two groups. One group represents nodes that are terminal elements of the Java source code. The other group represents nodes that are non-terminal elements. Terminal elements are the tokens in the Java source code being parsed and grammar productions that have been reduced to their specific meaning. Non-terminal elements are language constructs ultimately formed by a collection of terminal nodes in a specific pattern that represents a syntactically correct sequence of Java code.

The token nodes are formed by the scanner portion of the LALR parser and are described in the JavaFLEX specification. Tokens are Java keywords, separators, operators, literals, and identifiers. The string of characters represented by a token node is the lexeme value of the token. All token nodes have a common parent named `Token`; this class has `ASTNode` as its parent.

Terminal nodes that are not tokens typically encapsulate a token node and have no children. They have been reduced to their specific meaning. These types of nodes are implemented in classes to represent Java syntactical elements. For example, the Java keyword *this* is represented as a token by the `TokenThis` class. The `TokenThis` class is encapsulated in the class named `This`. The class `This` represents the Java construct to refer to the current object's reference, the `TokenThis` class represents the source code text of the *this* keyword. Token classes, except for literal tokens, are not individual nodes in the AST and as a consequence they are not visited. This is not a deficiency because they are encapsulated into another node, like in the example given, and in effect are visited. The information about the token node is available in the encapsulating node. Literal tokens are an exception to this because their textual value may need to be analyzed, and there are various literal types. Analysis checks may need to trigger on

particular literal types so this exception is made. The literal types are Boolean, character, string, floating-point in both the single and double precision variety, and integer. The integer literals come in several varieties each determined by the JavaFLEX scanner. Integers can be 32-bit (int) or 64-bit (long) and in decimal, octal, or hexadecimal format. Each of these is represented by a specific token class and is visited if present in an AST. In the set of provided analysis checks there are some that specifically trigger on these individual literal token node types.

Non-terminal nodes form the bulk of the Java language grammar and build upon themselves to create larger and larger Java constructs until a complete compilation unit is formed. The compilation unit is the root of the AST and a pre-order processing of the AST matches a reading of the Java source code from top to bottom, left to right. There are several non-terminal nodes that have common characteristics; these characteristics are factored into super-classes. One common type of non-terminal nodes is a list of other nodes, each of the same type.

A list of nodes is a node in the AST itself and is also a super-class for the classes implementing nodes in the list. This super-class, `ASTNodeList`, encapsulates the management of the list of nodes. Lists are used to group a repeating syntactical element at the place it is located in the source code. For example, the class `ArgList` is a subclass of `ASTNodeList`. The `ArgList` class implements a list of expression nodes in an AST where arguments would appear (e.g., A method call). The `ArgList` is a node in an AST and is visited accordingly. When it is visited, all the expression arguments in the list are also visited. The data structure for `ASTNodeList` is unique and interesting. It is a doubly linked list that appends items to the end of the list. The unique feature is

that the list is composed of elements that are also a list, but of only a single item. This is done so that the list and the elements of the list are of the same class and node type. Continuing the example of `ArgList`, the class `Expr` implements expression nodes. Expression nodes could exist outside the context of a list as well as in a list. To associate a node with the `ASTNodeList` hierarchy, the nodes are encapsulated in the list. When the first element of a list is encountered during parsing then an instance of the list type is created with the node element made the head of the list. When other elements are parsed belonging to the list, they are appended to the list. The list reference is propagated through the parser by referencing the head node in the list. When the node is appended, an instance of an `ASTNodeList` with that node as its single item is created. Then, that list is appended to the list that already exists. This technique means the no extra node type is needed just to represent the list as a whole. This structure allows for simpler processing in the both the parser and in the AST visiting.

At many places in the Java language grammar it is possible for a syntactical element to be present or absent. When an element is absent this is represented in the AST as a null option node. This type of node serves as a placeholder for optional and missing syntactical elements for a given Java source code unit. An example of this is a Java method call with no arguments. In this case instead of an `ArgList` class to contain the arguments, a `NullOpt` class is referenced instead.

To simplify the AST structure three classes are provided to aggregate information into a consistent form. These classes are `Modifiers`, `Name`, and `Type`. Several Java constructs utilize these classes. They are described in detail below.

The declaration of Java classes, interfaces, fields, and methods allow for modifiers to signal certain characteristics. These Java modifiers are aggregated and encapsulated into the `Modifiers` class. This aggregation is done so as to simplify the AST at the places where the modifiers occur in the Java source code. The order and position of each modifier lexeme in the source code is retained in the Modifiers class. The AST node of which the modifiers are a part references this aggregated `Modifiers` class. The individual modifiers are no longer nodes in the AST and thus are not visited. However, all the information related to them is available.

Several Java constructs require a name. For example, the package statement requires the name of the package to place the class and interfaces declared in a source file. This name could be a simple one-word name or a qualified name. A qualified name in Java has each word separated by a dot character. The AST node type of `Name` is used in just such constructs. Typically the `Name` node becomes an attribute on the construct's node and is not a part of the AST branch structure. This node simplifies the AST structure and provides a consistent and convenient encapsulation of names.

Java has eight primitive data type and an unlimited number of reference types. Each of these two categories of types can also be expressed as arrays. To provide a simple, consistent, and polymorphic way of handling data types, the `Type` class is provided. Wherever a data type is encountered in the Java syntax it is encapsulated in the `Type` class. Typically the `Type` class is an attribute to a Java construct that has a data type as a part of its syntax. AST nodes that incorporate a data type in some fashion, either by declaring or accessing are populated with methods to determine the data type. Declarations are straightforward because the data type and symbolic name are in the

same Java construct. Code that specifies an access to a symbolic name is more complex. The symbolic name must be resolved to determine what it refers to before its data type can be determined. Much as a compiler provides symbolic name data typing, so does JSCAT.

In the continued effort to simplify the AST structure, classes are placed in the `ASTNode` hierarchy. These classes are placed between the parent to all AST nodes, `ASTNode`, and the specific node types that represent the Java grammar. These classes group similar classes into a common class type. The `Accessors` class groups AST node classes that represent an access to a field or local variable. The `Callers` class groups AST node classes that represent a call to a method or constructor.

Java expressions are built on a precedence hierarchy. When parsing Java source code an expression could be encountered that starts with many different types of nodes that compose the precedence hierarchy. To provide a consistent trigger point for expression a placeholder node is built into the AST wherever an expression is found. Furthermore, there are a few constructs that require an expression that evaluates to a Boolean value. In such constructs, a placeholder node is built into the AST to signal that a conditional expression follows. This provides a convenient trigger point for conditional expression and distinguishes them from expressions in general.

AST Extensibility and Processing

The AST structure linked with parent and child references provides the basic structure necessary to navigate the tree. Each node type has appropriate methods to obtain references to the specific children associated with it. Each node also has other attributes that provide information to the analysis checks appropriate for the Java grammar it

represents. The AST structure and these characteristics implement the extensibility characteristics of navigability and node information.

Once the source code file has been parsed and the AST constructed, it is then processed by the JSCAT framework. This processing is defined as calling the visit method of each node in the AST. First, the root node, which is the entire compilation unit, is obtained from the parser. Then the visit method of the root node is called. This starts in motion the pre-order visiting of each node in the AST. After the root node is visited, then each of its children is visited in order of the source code from left to right, top to bottom. Of course as each child is visited, its children are visit and so on. As each node is visited it is appended to the visit order list and any node specific pre-actions are done. These node specific pre-actions are a part of the JSCAT framework and are not extensible. For example, when a class declaration node is visited, the node specific pre-actions create a symbol table associated with the class name among other things. After these pre-actions are done, then all the analysis checks registered with the node are processed so as to call the pre-action method. This is the user extensible part of the JSCAT framework. Next, all the children of the node are visited. Upon the completion of visiting all the children the registered analysis checks post-action method is called and then the node specific post-action method is called.

The JSCAT framework also provides a mechanism for user exit methods to be called at strategic points in the processing. A user exit is a class that implements a well-known method named `process`. At the strategic points if the corresponding user exit reference is valid then the `process` method is called. The user can place logic within the method to do whatever may be desired. When a node is visited and appended to the visit list,

then the *visit list user exit* process method is called. Other user exits are discussed in the JSCAT Information Facilities section with the information source with which they are associated.

The visiting of nodes and calling of analysis check methods, which is the heart of the framework, and the extra feature of other user exits, provide the extensibility characteristics of the JSCAT framework.

CHAPTER 5
INFORMATION FACILITIES

JSCAT provides several facilities to obtain information about the source code being analyzed. This information is useful in writing analysis checks. The source code context of a particular node in the AST is provided by various means. The first is a list of nodes in the order in which they have been visited. Secondly, three individual lists of nested nodes that signal the beginning of a type, unit and scope in Java. Lastly, there is a stack of source code context information objects. Symbol tables, a data structure to implement global data, and the source code being analyzed provide other information facilities. Each of these facilities is described in detail below.

Data Structures

The *visit list* structure contains a reference to those nodes that have been visited. Nodes are visited in a pre-order processing sequence. This list grows until processing is complete. The nodes that have been visited remain in the list. This list can be interrogated to determine the source code context in which a particular AST node exists. For example, a `BreakStmt` node may be visited after either a `WhileStmt` node or a `SwitchCase` node. By tracing the visit list in reverse order, whichever one is encountered first is the context in which the `break` statement occurred. This type of information may prove useful and necessary to perform an analysis check.

The Java language has several organizing elements that JSCAT mimics. At the highest-level Java organizes objects into either a class or an interface. A class or interface declaration defines a type. Within a type, several Java organizing constructs

can be declared. JSCAT refers to these constructs as units. Units are methods, constructors, field declarations, and static blocks. Class and interface declarations are also units. Units can contain a variety of Java constructs, namely statements and expressions. Many of the statements, such as `for` or `while` loops, define a scope. Typically this scope is related to a variable scope space. However, in JSCAT type and unit members are also scope members. The purpose of this is so that the current type, unit, and scope can be determined. Three lists are provided for this traceability purpose.

The *type*, *unit,* and *scope lists* contain references to nodes that signal the beginning of a *type*, *unit,* or *scope*. These lists behave like stacks of nested items of the same type. Nodes that signal the beginning of a *type* are a class or interface declaration. Nodes that signal the beginning of a *unit* are a constructor, method, field, and static block declaration and also class or interface declarations. Nodes that signal the beginning of a *scope* are nodes that define a new symbol scope in the source code in accordance with the Java language specification and all the type and unit nodes except field declarations.

The *source code context stack* contains source code context objects encompassing such information as the current class or interface node, the current unit (constructor, method, static block, and field) node, and the current symbol scope node. A context object is placed on the stack whenever any of the type, unit, or scope of the current AST node is changed. The *source code context stack* is an alternative information source to the individual *type*, *unit,* and *scope* lists, with the added benefit of tracking the statement number that is being visited within the current scope.

A hash map named *globalData* is keyed on an object reference; typically a string that represents a symbolic name, and associated with an object reference facilitates global

data. The associated object is the value of the keyed object (the symbolic name). This data structure implements the information source and extensibility characteristic of global data.

A compiler typically builds a symbol table to track identifiers that symbolically refers to a data value. These symbols usually have a data type and a compiler uses this information to enforce many of the semantic rules of a programming language. Java follows this typical style of languages and thus JSCAT builds and maintains a symbol table for possible use by analysis checks.

JSCAT maintains a symbol table for each class and interface declaration encountered in a source file being analyzed. Fields and local variables are separated into two tables for each class or interface declaration. In the case of interface declarations the local variable table will be empty because Java does not allow local variables within an interface. These symbol tables are stored in a hash map keyed on the class or interface name. This is particularly useful when dealing with inner classes. Each JSCAT symbol table is structured as a hash map keyed on the symbol name. Symbol scope is supported by the symbol table storage and retrieval methods. Each symbol in the table has attributes related to the symbol's declaration (modifiers and type), the symbol's definition (name and possibly initialization nodes), and source code context information. Context information encompasses the current class or interface node, the current unit (constructor, method, static block, and field) node, and the current symbol scope node.

The source code text of the source file being analyzed is stored in a string array indexed by line number. This provides a convenient facility for producing messages that include the source of the suspicious code detected by a particular analysis check.

These information facilities provide the implementation of the traceability characteristic for the JSCAT Framework. Each of these facilities provides information managed in a structured way to trace the path leading to the current AST node being visited.

Analysis Check Registration and Triggering

Analysis check's must be registered with the AST node type that, when visited, will trigger the analysis check to be processed. Registering of analysis checks takes place when the JSCAT application is launched and the configuration file is processed. The configuration file is described in the JSCAT Configuration section below. Once the analysis checks are registered the source code text is parsed and each node in the built AST is visited. Analysis checks registered with that node type are triggered for processing as each AST node is visited. The registration, triggering, and coding requirements of analysis checks form a major component to which approach to implement the JSCAT framework.

Two different techniques for registering analysis checks were considered. The first is termed the *manual* technique. It required the analysis check constructor to call the register method of each AST node that acted as a trigger. Furthermore, the analysis check pre and post action methods accepted the common parent AST node as an argument. This requirement caused the analysis check code to determine which specific AST node type was passed to it and cast it to that type. In Java, this would be done using the `instanceof` operator and the casting expression. This technique benefited from slightly less processing and ease of coding in the framework code. However, burdens were placed on the analysis checks to code trigger registrations in the constructor and

casting from general to specific AST node types. This complicated the main feature of JSCAT, its extensibility, by making user written analysis checks more difficult to write.

The second technique, termed the *automatic* technique, overcomes both of the first technique's deficiencies. The framework for the analysis checks does the registration. The pre and post methods can now accept parameters of specific AST node types, eliminating the need for casting logic in the analysis checks. The Java Reflection API is extensively used to implement this technique.

The Java Reflection API consists of the `java.lang.Class` object and the `java.lang.reflect` package. Its purpose is for the programmatic interrogation of a Java class file. The Class object provides methods to obtain the characteristics of the class and the members (constructors, method, and fields) of the class. The methods that get the members of a class return objects of that member type and are packaged in `java.lang.reflect`. There are methods in each of these objects to provide information about the characteristics of the members.

To accomplish the *automatic* registration technique, analysis checks must extend a class in the framework. Also, the analysis check class declaration can specify which specific AST node's pre and post action methods to implement. This causes static checking by the compiler to ensure that all the necessary pre and post action methods for each specific AST node type, that triggers the analysis check to be processed, exist. The class that the analysis checks are required to extend interrogates the analysis check class using the Java Reflection API to find pre and post action methods and the specific AST node type they accept. There can be multiple pre and post action methods, each with a different AST node type parameter. This allows many AST node types to trigger the

analysis check to be processed. During this registration process messages are generated if a deficiency is detected in regards to the pre and post action methods. One deficiency occurs if there are missing pre or post action methods in the analysis check. Another deficiency occurs if the method signature of a pre or post action method does not conform to the requirements specified in the interfaces.

This *automatic* registration technique has increased processing needs due to the use of the Java Reflection API. Once a correct pre or post action method signature is found in the analysis check class, its registration method is invoked using the invoke method of the Reflection Method object. During the visiting of the AST nodes the analysis checks that are registered with it are processed. This is done by again using the Java Reflection API to invoke the pre and post action methods that correspond to the specific AST node type.

As a consequence of this technique there can be many method signatures, two for each AST node type that triggers the check, to be coded. Oftentimes, the bodies of these methods may be empty. Furthermore, the code in the methods for each different AST node type may be the same and thus duplicated. However, the object-oriented features of Java can be employed by the analysis check code to accomplish code reuse.

In the actual implementation both techniques, *manual* and *automatic*, are available and which one to use is determined by a setting that is global to the entire framework and all the checks to use. However, separate analysis checks must be written for each technique. An extra feature of the *automatic* technique is that the parameter to one of the pre and post action method pair can be the common AST node type. This is a useful code reuse technique, but requires the trigger nodes to be registered in the analysis check constructor

and casting in the pre and post action methods. In other words, the *automatic* technique can be used in such a way as to emulate the *manual* technique on a check-by-check basis.

As the AST nodes are visited, any analysis checks registered with that node type are triggered at that node instance. Triggering is done on a single node no matter what its source code context. For example, the `Conditional` AST node type triggers any registered analysis checks whether the `Conditional` is part of a `while`, `for`, or `if` statement. This design was chosen to simplify the implementation and the triggering registration mechanism for the analysis check. A simple one-to-one relationship of AST node type and analysis check is formed to implement the design choice. Each AST node type maintains a list of analysis checks registered with it. While this design is simple, it is still quite effective. Many of the identified analysis checks that are provided only need a single node for triggering. The source code context in which the node occurs is not important to the analysis check. Other analysis checks provided must determine in what source code context they occur. This burden is placed on the specific analysis check code. It must detect the context and process accordingly. The JSCAT framework provides information as described in the JSCAT Information Facilities section to aid the analysis check code to determine the source code context of an AST node occurrence.

Analysis Check Examples

The analysis checks provided are categorized into several groups. These groups are suspicious `tokens`, `expressions`, `statements`, `object components`, and `exception handling`. Each of these groups has multiple analysis checks that report on suspicious code patterns. Most of all the provided JSCAT information facilities and extensibility characteristics are employed in the various analysis checks. The most

frequent facilities and characteristics used are data methods related to an AST node, global data, and navigation of the AST to determine the source code context. Each group is described below with an example analysis check. The appendix provides a complete list and short description of each system-provided analysis check.

Suspicious tokens cover such analysis as reporting the use of hard-coded literal values in expressions, unclear octal and hexadecimal literal usage, and integer literal values of type long not specified as such.

To report on hard-coded literal values in an expression, the system-provided analysis check is triggered on encountering a `Literal` AST node. The purpose of this analysis check is to indicate hard-coded values in the source code and promote the use of constants instead. AST navigation and node data methods determine if the hard-coded value is part of a constant declaration. A constant in Java is declared as a class field with the static and final modifiers applied to it. When a Literal node triggers this analysis check, a search through the parent links in the AST is performed to determine if the literal is part of a constant field declaration. Hard-coded values yield a message when encountered in a non-constant declaration context, whereas a hard-coded value in a constant is silently bypassed. A detailed explanation of this analysis check is given using the

```
megaBytes = numBits/BITS_PER_BYTE/1024;
```

Java code snippet. Obviously, 1024 is the hard-coded literal value. Figure 5-1 shows the AST structure and visit order for this code snippet. The AST sub-tree begins with the `Assignment` node. The nodes are visited in a pre-order traversal of the tree, numbered 1 to 8. The text within the parenthesis indicates the source code lexeme associated with

the AST node. When the `Literal` node is visited, the analysis check is triggered to perform its actions. Once the parent AST links

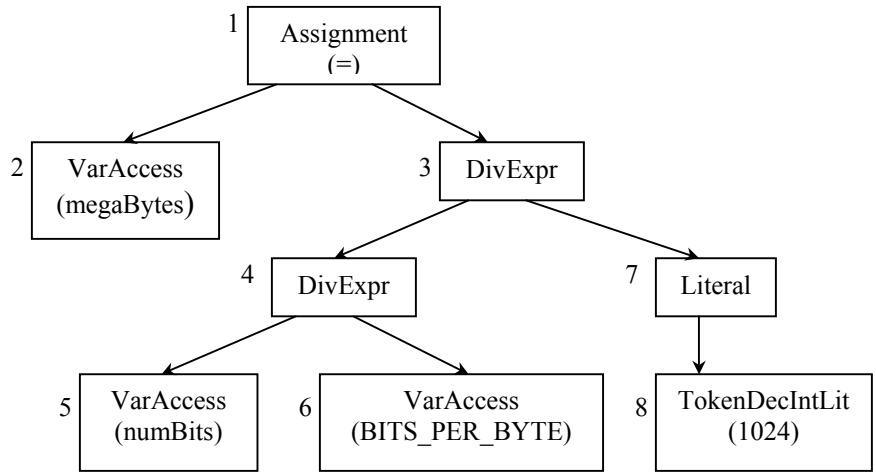

Figure 5-1 Hard-Coded Literal Example AST

are searched to determine the `Literal` node is not within a constant declaration context, a message is produced. The message indicates that the hard-coded value 1024 should be made a constant value and that constant symbolic name used in the code snippet. To gain a deeper understanding of the code involved in the analysis checks, this example includes the source code for the analysis check in Figure 5-2. The other examples presented below have similar source code modules for their analysis checks.

In Figure 5-2, the bolded portions of the analysis check code is common to all analysis checks. The imports provide access to the JSCAT framework and the Java AST nodes. The `ICheck` class that is extended provides the registration mechanism through a super-class constructor that is called be default when the analysis check is instantiated during the processing of the JSCAT configuration file.

The package in which to place the analysis check is chosen at the discretion of the user for their code organization scheme. The name of the class,

```
package JScatChecks;

import JScatFrame.*;
import JScatFrame.Java122.*;

public class HardCodedValue extends ICheck implements ILiteralCheck
{
  public void preAction(Literal node)
  {
      ASTNode tnode=node;
      while (tnode.getParent() != null && tnode.isUnitBegin()==false)
      {
            tnode = tnode.getParent();
            if (tnode instanceof FieldDecl)
            {
                  Modifiers mods = ((FieldDecl)tnode).getMods();
                  if (mods.isFinal() && mods.isStatic()) { return; }
            }
      }
      JScatStatics.printCheckMsg("Use a constant declaration instead of
                        a hard-coded value.", node, this);
  }

  public void postAction(Literal node)
  {
  }
}
```

Figure 5-2 Hard-Coded Literal Example Source Code

`HardCodedValue`, is also at the discretion of the user. The `ILiteralCheck` interface is implemented because the pre and post action methods that are implemented accept the `Literal` AST node type. The `while` loop uses AST navigation - the `getParent` method - to find the beginning of the JSCAT unit. While searching for the beginning of the unit, if a `FieldDecl` that specifies a constant declaration is found then the method is exited. Node information methods are employed, as in the

```
((FieldDecl)tnode).getMods();
```

code snippet.

Suspicious expressions covers such analyses as reporting the use of the Java ternary conditional operator `?:` without parenthesis around the conditional expression, embedded

variable declarations and assignments, castings that may have data loss, usage of the assignment = and equals == operator when the other is more commonly correct, and the use of pre and post increment and decrement operators in complex expressions.

Complex expressions are expressions containing more than one operator. In such expressions, the use of the increment and decrement operators (++, --) is confusing in regards to when they are performed. An analysis check is provided to report such usage. To accomplish this the global data facility is used to count the number of increment and decrement operators. This analysis check is first triggered on visiting any expression AST node and then triggered on visiting any increment or decrement nodes within the expression sub-tree. The Java code snippet

```
a = b++ * 3;
```

produces the AST structure and visit order shown in Figure 5-3. The Java evaluation order for this expression is

```
a = ((b++) * 3);.
```

The `ExprStmt` node triggers the pre-action of an analysis check to initialize the count of increment and decrement operators in a global data variable to zero. The global data facility is populated with a unique string value to serve as the global data variable. The value associated with this key value is set to zero. As the other nodes are visited as children of the `ExprStmt` AST node, any increment and decrement nodes trigger an analysis check to increment the global data variable value. After the expression sub-tree is visited, the post-action of the expression AST node retrieves the global data value. A message is produced if a complex expression with increment and decrement operators is present. This is known if the count of these operators is greater than zero and the expression is more than just a simple increment or decrement. The global data facility is

essential here because of the complexity of expressions and their AST representations. There are numerous code patterns that could contain conditions producing a

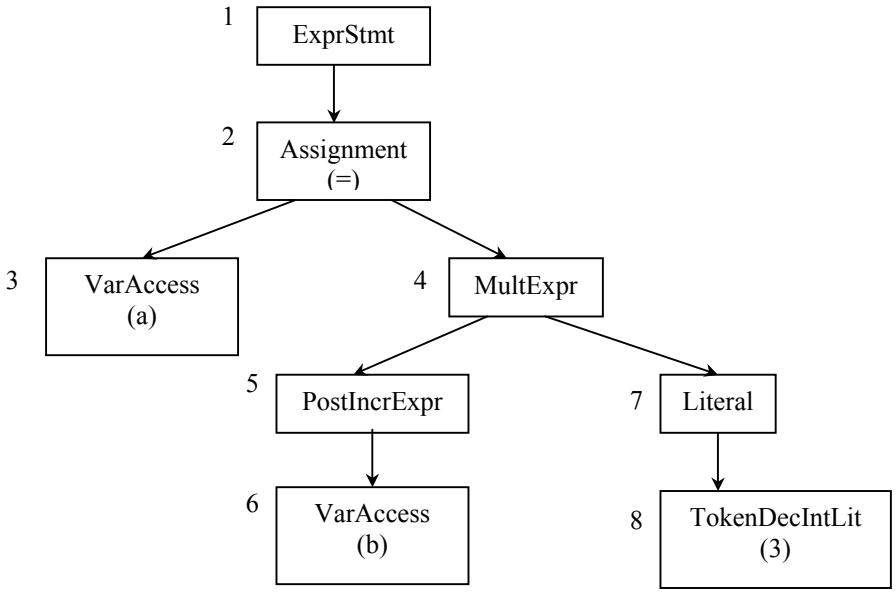

Figure 5-3 Complex Expression Example AST

message in this analysis check. Since JSCAT triggers on only one AST node at a time, the source code context of the expression is unknown. By utilizing global data the simple and elegant solution described above is implemented for detecting all the possible code patterns related to this analysis check without being concerned with the source code context.

Suspicious statements cover such analyses as reporting the use of single statements where blocks could be used, missing `break` statements in the `switch` construct. As well as, loop indexing that is used as an array index in the loop body and does not end the loop by reaching the length of the array, and using the same symbol name for field assignments and parameters.

Node data and symbol table usage are illustrated using the analysis check, which detects same symbol name usage for field assignments and parameters. This analysis check is described using the

```
private int value;
public void setValue(int value)
{
      value=value;
}
```

Java code snippet. The most common intent for this code snippet is to assign the field *value* to the argument *value*. For this to occur the code should be `this.value = value`. The analysis check determines if the left-hand side of the assignment is an argument symbol name that matches a field symbol name. If there is a match then a message is produced. Figure 5-4 shows the AST structure, visit order, and symbol table operations for the field declaration, Figure 5-5 shows the method declaration.

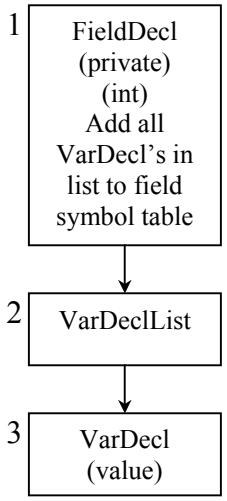

Figure 5-4 Same Symbol Example Field Declaration AST

When an `Assignment` node is visited this analysis check is triggered for processing. The analysis check searches the local symbol table for a symbol matching the left-hand side variable name of the assignment. If it is found and it is a local variable or argument

then the field symbol table is searched for the same variable name. If the same symbol name is found in both tables then a message is produced indicating a potential problem is the source code. The symbol table information facility is essential to performing this analysis check. Because the JSCAT framework provides the symbol table information, the scope of possible analysis checks is broadened - making JSCAT quite extendible.

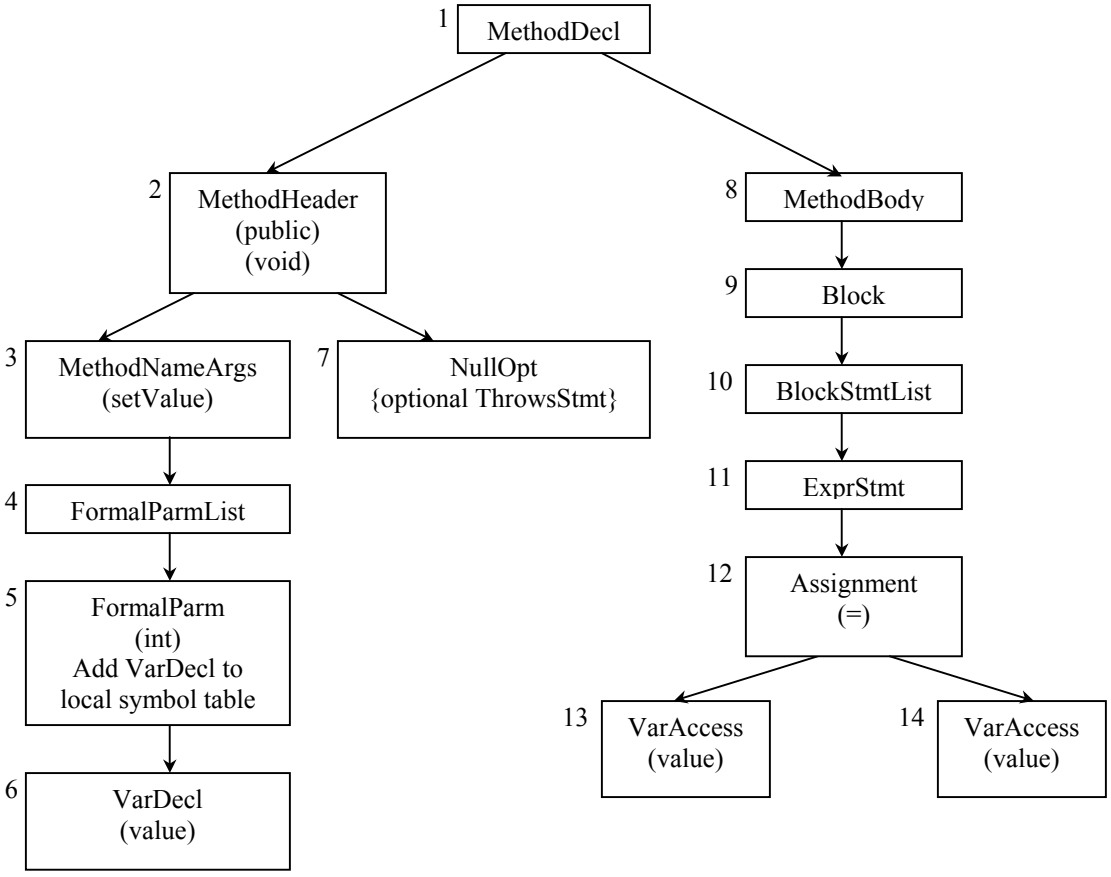

Figure 5-5 Same Symbol Example Method Declaration AST

Suspicious object components are divided into two categories, those dealing with fields and those with methods. Suspicious field usage covers such analysis as reporting the use of class and instance fields without the `private` modifier, without getter and setter methods, and shadowing in sub-classes. Suspicious method usage covers such

analysis as sub-class method overloading instead of method overriding, not coding class specific `equals`, `clone`, `finalize`, `toString`, and `hashcode` methods.

To illustrate the use of `ASTNodeList` navigation and the Java Reflection API, the field shadowing analysis check is described. Figure 5-6 is applicable to this analysis check. It shows the AST portion of the field declarations in each class of the code snippets. The Java code snippets for this analysis check are

```
class A
{
      private int field1, field2;
}

class B extends A
{
      private int field1, field2;
}.
```

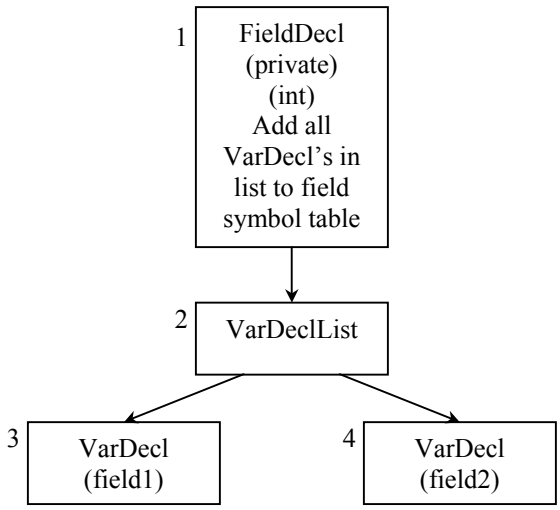

Figure 5-6 Shadowed Field Example AST

This analysis check is triggered on visiting a `FieldDecl` node. The analysis check searches all the parent classes of the current class, using the Java Reflection API, for a field with the same name. All the variable names in the field declaration are processed to find a match. A message is produced for each field that has a match, indicating that the

parent field is being shadowed. This is commonly not desired and the sub-class field name should be changed.

Suspicious exception handling analysis checks reports such code patterns as empty `catch` blocks, array index or string character position out of bounds exceptions being caught, and `catch` blocks with only output related method calls in it. The analysis check that reports if an array access is not guarded by a `try/catch` block with the array index out of bounds exception is described.

This analysis check uses the AST navigation, node information methods, and list-navigation to accomplish its task. The Java code snippet

```
int i = -1;
b = a[i]; //unguarded array access; produce message
try
{
    b = a[i]; //guarded array access; no message
}
catch (ArrayIndexOutOfBoundsException e)
{}
```

is used to explain this analysis check. Assume that the variable `b` and array `a` have been declared as integer and properly initialized. The first array access is unguarded and produces a message. The second array access is guarded by an appropriate `try/catch` statement and does not produce a message. Figure 5-7 shows the AST structure and visit order for the first array access. Figure 5-8 shows the AST structure and visit order for the second array access.

When an `ArrayAccess` node is visited the AST parent nodes are visited until either a `TryStmt` is found or the parent node is the beginning of a unit. If the beginning of a unit is encountered and no `TryStmt` was found within the unit then the array access is unguarded. If a `TryStmt` is found then the `CatchList` is searched for a `CatchStmt`

clause with the `ArrayIndexOutOfBoundsException` node. If this exception is not found then a message is produced, otherwise there is no message.

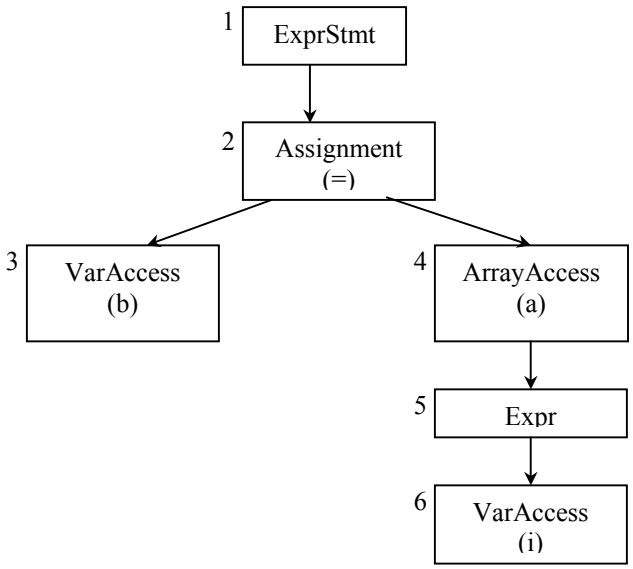

Figure 5-7 Unguarded Array Access Example AST

Control Features

The message settings discussed with the JSCAT messages below can be controlled in the configuration file and in JSCAT pragma comments embedded in the source code being analyzed. Essentially wherever there can be a Java statement in the source code, there can be a JSCAT pragma comment. The pragmas are recognized by the `//#` sequence of characters. A pragma analysis check, which can be extended, is provided that parses the rest of the comment. The parsing in the pragma check gives maximum flexibility to its remaining format. The `Pragma` AST node does parse the first three words in the comment into separate pieces of information for the convenience of the pragma analysis check. However, this information does not have to be used and the entire unparsed pragma comment can be obtained from the `Pragma` AST node.

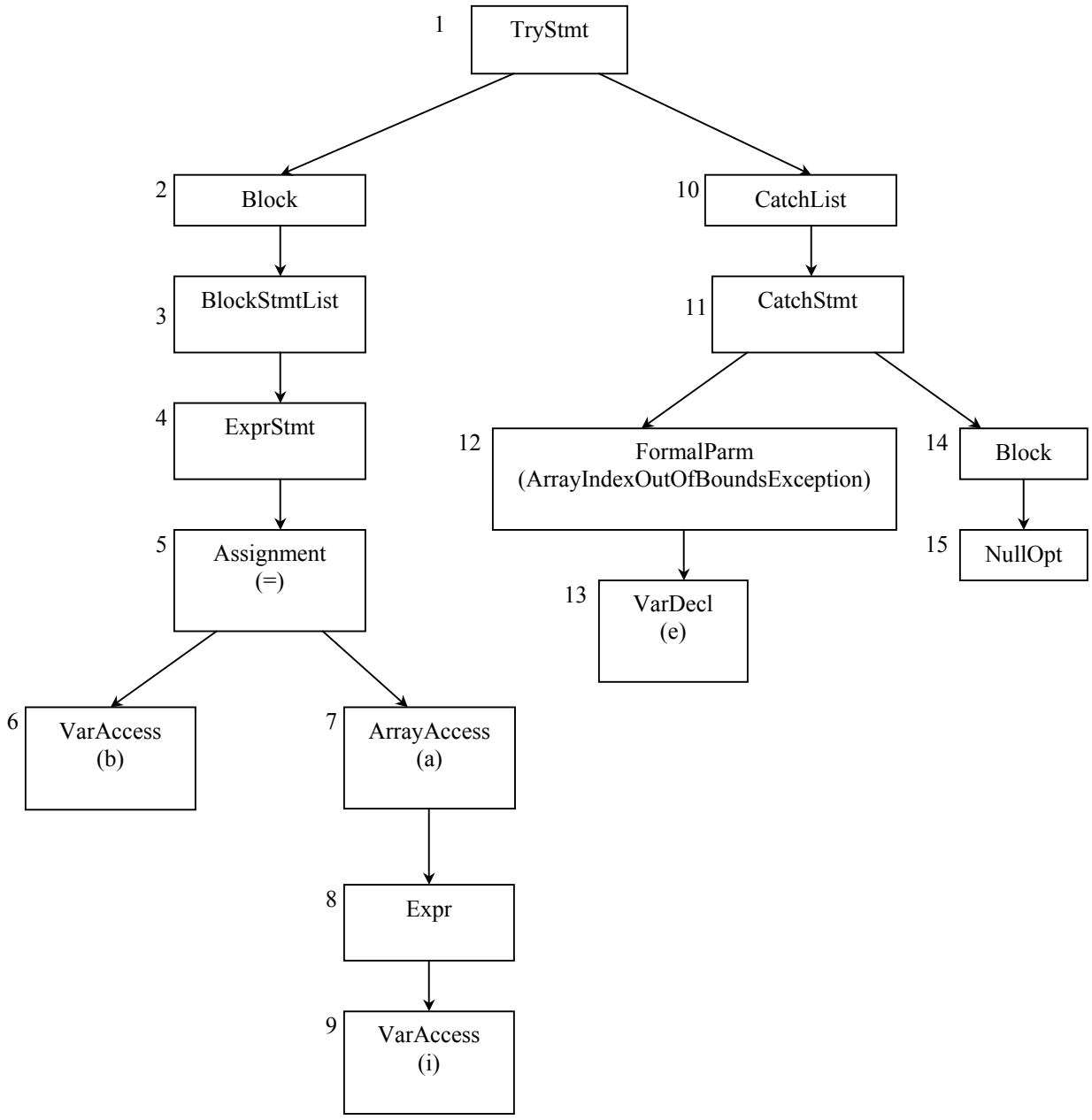

Figure 5-8 Guarded Array Access Example AST

The pieces of information that are provided are the facility, the type, and the value. The facility is a word to categorize the pragma into a major group. The provided pragma analysis check only recognizes pragma comments that are coded with JSCAT as the facility name. The type word further categorizes within a facility to which the pragma is related. Currently two types are recognized, the settings and check type. The settings

type controls whether JSCAT messages are on or off. The check type controls whether a particular analysis check is on or off. The value word is specific to the type category. For the provided types, setting and checks, in the JSCAT facility the value specifies the setting or check class name respectively, an equals sign and either the keyword on or off. For example, `//#JSCAT CHECKS JScatChecks.Check1=off` turns the Java class `JScatChecks.Check1` off.

Pragmas can be extended by replacing or modifying the provided pragma check or by writing other pragma checks and registering them with the `Pragma` AST node type. The extensibility framework and usage is the same as for analysis checks.

Pragmas to control message output can be used to target where verbose output should occur that traces the processing of the AST. The verbose messages can be beneficial to understand the structure of a particular AST and the order of processing that is occurring while visiting each node in the AST. Such an understanding may help to write and debug an analysis check that is being added to the JSCAT framework.

Pragmas to control analysis check triggering can be used to target where a particular check should be active or not. This facility allows the user to strategically place pragmas in the source code being analyzed to suppress the message of a particular check in a particular section of source code.

This pragma mechanism implements granular control for the analysis check's processing inclusion, exclusion, and ordering extensibility characteristic.

A method is provided in the framework that should be used whenever an analysis check generates a message. This method records the number of messages produced by

each check, prints the specified message and optionally the source line. The source line can optionally indicate the token that triggered the analysis check.

Other messages, useful for debugging analysis checks and understanding the structure of a given AST can be set on or off. These messages are granularly categorized to give a high degree of control over what messages are produced. These categories are AST construction, analysis check registration and processing, AST node visiting, statistics, Java exception stack trace, and AST contexts. The contexts are type, unit, scope, and symbol table operations. Contexts are described in the JSCAT Information Facilities section. When they are entered, exited, or some other operation is done, a message can be produced.

JSCAT has extensive configuration options. Configuration can be done through pragma's as mentioned above on a granular basis. To set initial configuration options, a configuration file is used. This file can set which message output to produce and more importantly, which analysis checks to register and thus set on. The configuration file is the only place in which to register analysis checks.

A configuration file is used for registering analysis checks instead of searching a particular directory for Java class files that implement an analysis check. The configuration file provides greater control. Within the configuration file the analysis checks package and class name are placed. The file is processed sequentially from the beginning. Analysis checks are registered in the order in which they occur in the file. There is one analysis checks name per line in the file. Any line can be commented using the // characters at the beginning of the line. Lines that are commented are not processed. This facility allows for particular analysis checks not to be registered at all for

a particular run of the JSCAT program. The configuration file implements the extensibility characteristic of including, excluding, and ordering analysis checks.

CHAPTER 6
CLOSING REMARKS

Static source analysis is a promising field of research to aid programmers in producing robust code. It is just one of many techniques for producing and debugging source code. JSCAT's abstract syntax tree processing is one particular approach to static source analysis. Tools similar in use and usefulness to compilers offer the brightest hope for writing robust source code before the code is run and tested. JSCAT is an attempt to provide an easy to use and familiar tool. The extensibility characteristics of JSCAT are the foundation for widespread applicability.

I believe that JSCAT brings the state of AST-based static source analysis to a point of consolidation for previous ideas and provides a framework for extensible work. JSCAT was used to analyze the Zip utility package source code from the Java Development Kit version 1.3.0 to gain a flavor for the types of results expected with the use of JSCAT. The Zip package implements the ubiquitous tool to compress files and consolidate them into a single file. The Zip tool makes it easy to transport many files together while also reducing their size. The Java Zip package provides the programmer with an interface for using the tool within Java applications. The eighteen source files that make up the Java Zip package yielded the following problem counts. In the suspicious token category, hard-coded values lead the problem counts with 461. The use of a long data type without a long literal indicator was encountered 32 times. The suspicious expression category revealed nine problems of not clearly parenthesizing the ternary conditional operator expression. Embedded assignments were found two times.

Use of the assignment operator in a conditional expression was found two times. Usually in a conditional expression an equality operator is what is desired. Equality operator instead of an assignment operator usage was found once in an assignment statement. Upon source code inspection, the suspicious uses of the equality and assignment operators proved not to be problems. Expression containing more than one operator and also containing post/pre-increment/decrement operators was detected 17 times. The use of signed and unsigned right shift operators was found 19 times. These have the potential of not being what is desired but are most likely correct. Short-circuit evaluation of conditional expressions was found 46 times. These also have the potential of not being what is desired but are most likely correct. Five instances of unguarded object casting were found. These have the potential of leading to runtime problems. The suspicious statements category contained three uses of a single statement where a block of statements could be coded; each proved to be correct upon code inspection. Assignments involving fields and local variables without the "`this.`" qualifier were found four times. While each of these proved to be correct they were hard to quickly understand. Missing `break` statements were found 30 times. Many problems were detected in the suspicious object-coding category. Class and instance fields without the `private` modifier were encountered 114 times. The use of a class object reference instead of the class name for class fields was encountered four times. Field shadowing and local variable field shadowing were found two and eight time respectively. The `finalize` method failing to call the `super.finalize` method was discovered three times. Method overloading occurred 33 times and was reported as suspicious because typically what is desired is method overriding. Hardly any of the 18 source files included methods for `equals`,

`clone`, `finalize`, `toString`, and `hashcode`; 80 problems were generated related to these omissions. The suspicious exception-handling category reported 25 instances of unguarded array accesses and one instance of an unguarded string-indexed access. While many of the reported problems were false-positives they did reveal places where the code could be made more readable. Other reported problems detected truly potential errors, such as the unguarded object-castings, array accesses and string-indexed accesses. I believe that JSCAT could have been a great aid in the development of the Java Zip package.

Future work to the JSCAT framework is to update the tool to use and recognize the Java 1.4 language specification. Message reporting needs new features such as the ability to sort and rank messages for output. Outputting messages in an order different from order of occurrence would be beneficial for programmers to target their efforts in correcting suspicious code. Furthermore, the framework could be applied to other languages by simply implementing a grammar for the new language and providing the AST structure appropriate for the language. The framework could still be written in Java or could, of course, be ported to another language, though an Object-Oriented language would be best suited to such an effort.

A tool for inferring bugs in source code [22] could be used or JSCAT compatible version written to aid in writing analysis checks and deriving patterns from source code. A significant enhancement would be to provide the programmer with a method of performing data-flow analysis, similar to JLint [15] and the Stanford meta-compiler work [1, 18-19]. A greater library of analysis checks would be beneficial in providing

programmers with the aids to produce robust source code. Finally, more experimental results should be obtained to validate the usefulness of the JSCAT tool.

APPENDIX A
SYSTEM-PROVIDED ANALYSIS CHECKS

Suspicious Tokens

Hard-Coded Values

Report literal values coded in variable assignments and expressions that are not constant declarations.

Octal Digit Expected

Report octal escape sequences within a string that has less than three valid octal digits. (eg. `print("\128");`) Java interprets this string as a linefeed followed by the number 8.

Too Many Octal Digits

Report octal escape sequences within a string that has more than three valid octal digits. (eg. `print("\1234");`) Java interprets this string as the octal value 123 followed by the number 4.

Too Many Hexadecimal Digits with Unicode Character

Report Unicode escape sequences that are followed by a valid hexadecimal character. (eg. `print("\uABCDE");`) Java interprets this string as the Unicode hexadecimal value ABCD followed by the letter E.

Lower-case Long Value Indicator Used

Report the use of lower-case (el) for literals of type long; it is too similar to the number one character. (eg. `long a = 0x1111111;`)

Multi-dimensional Array Syntax Coded with Both the Data Type and Array Name

Report array dimension specification that is present in both the data type and array name. (eg. `byte[] arrayOfArrayOfBytes[];`)

Long Data Type Literal Value without Long Literal Indicator

Report any long literal values that do not use the long literal indicator. This can occur in variable declarations, assignments, method return statements and method call arguments.

Suspicious Expressions

Conditional (Ternary) Operator Boolean Expression Without Parenthesis

Report any conditional operator expression that does not have parenthesis around the first, Boolean, operand. (eg. `x>=0 ? x : -x;`)

Embedded Assignments

Report any assignments within an assignment; either in a declaration (eg. `int a = b = 1;`) or an expression (eg. `d = (a = b + c) + r;`)

Loss of Data Castings

Report any automatic widening cast where least significant digits may be lost. This occurs in automatic casts from `int` to `float` and `long` to `float` or `double` data types.

Assignment Instead of Equality Operator

Report any use of the assignment operator (=) instead of the equality operator (==) in conditional expressions.

Equality Instead of Assignment Operator

Report any use of the equality operator (==) instead of the assignment operator (=) in declarations and assignments.

Object Comparison with Equality and Inequality Operators instead of Equals Method

Report any use of the equality (==) and inequality (!=) operator for object comparison instead of the `equals` method. The operator comparison determines if the object references are equal or not. The `equals` method comparison determines if the object's are equivalent in value, not reference.

Pre/Post-Increment/Decrement Operators in Complex Expressions

Report any use of the pre and post increment (++) and decrement (--) operators in an expression that contains other expression operators.

Using Signed Right Shift and Unsigned Right Shift Operators

Report any use of the signed right shift (>>) and unsigned right shift (>>>) operators because the one other than the coded one maybe the desired one.

Order of Expression Evaluation

Show the order of evaluation for an expression by using parentheses and reordering operators and operands based of the expression hierarchy of Java.

Conditional Expression with Short-Circuit Evaluation

Report any conditional-And (&&) and conditional-Or (||) operators usage; stating the effect of the evaluation of the first operand and its Boolean value.

Unguarded Object Castings

Report any casting of an object from one type to another that is not guarded by a `try/catch ClassCastException` block.

Suspicious Statements

Assignments Involving Field, Parameter and Local Variable Names That Are the Same

Report any assignment where the right-hand side symbolic name is the same as a field name in the class. Most likely the field name is being assigned to but the '`this.`' qualifier is missing.

Statements Not In a Block

Report any single statements where a block of statements could be coded.

Empty Statement Followed by a Block of Statements

Report any occurrence of an empty statement (`;`) followed by a block `{ … }` of statements. This is probably an inadvertently coded semicolon. (eg. `if (x > y); { int tmp = x; x = y; y = tmp; }`

Dangling-Else Problem

Report which `if` statement an `else` statement is associated with when blocks of statements are not used.

Missing Break Statement Before Case or Default Clauses

Report any `case` or `default` clauses except the first clause after the `switch` statement that is not preceded by a `break` statement.

eg.

```
case '+':
case '-':
     sign = 1;
     break;
```

Array Access with Loop Indexing Variable Not Terminating with Array Length Value

Report any loops that contain array accesses that are indexed with the loop indexing value where the loop is not terminated by comparing the loop index variable with the length of the array.

Suspicious Object Coding

Class and Instance Fields without the Private Modifier

Report any field declarations that do not have the `private` access modifier.

Class Fields Not Accessed Via the Class Name the Field is Declared In

Report any class field accesses that do not use the class name the field is declared in. The use of an object reference to a class field is not good practice.

Fields without Getter and Setter Methods

Report any fields declared in a class that do not also have a `get<FieldName>` and `set<FieldName>` method and visa versa.

Field Shadowing

Report any fields declared in a class that are the same name as a field in a super class.

Local Variable Field Shadowing

Report any local variables declared that are the same name as a field in a super class.

Method Finalize Not Calling the Super.Finalize Method

Report when a `finalize` method in a class is present that does not contain a call to the `finalize` method in the super class.

Class and Super Class Method Overloading

Report any method that overloads a method in the super classes. Typically, this should be an override.

Classes without the equals, clone, finalize, toString and hashcode Methods

Report any class that does not contain a method signature for the `equals`, `clone`, `finalize`, `toString`, and `hashcode` methods.

Suspicious Exception Handling

Missing ArrayIndexOutOfBoundException

Report any array access that is not guarded by a `try/catch` `ArrayIndexOutOfBoundsException` block.

Missing StringIndexOutOfBoundException

Report any string accesses via an index that is not guarded by a `try/catch` `StringIndexOutOfBoundsException` block.

Empty Catch Block

Report any `catch` blocks that are empty.

Catch Block with Only Output Type Statements

Report any `catch` blocks that only contain statements that outputs a message.

APPENDIX B
SAMPLE PROGRAM AND JSCAT OUTPUT

Source Files

File:SampleParent.java

```
  1:package JScatChecks;
  2:public class SampleParent
  3:{
  4://Check30:Field shadowing
  5: private int cv30;
  6: private boolean iv30;
  7://Check33:class and super class method overloading
  8: public void doTest33(float a) {}
  9:}
```

File:Sample.java

```
  1:package JScatChecks;
  2:import java.util.*;
  3:public class Sample extends SampleParent
  4:{
  5://Turn off hard coded literals
  6://#JSCAT CHECK Check1=off
  7://Check24:field & locals assignment
  8: private int c24a=0;
  9: private int c24b=0;
 10: private void setC24(int c24a)
 11: {
 12:      this.c24a=c24a; //ok
 13:      c24b=c24a; //ok
 14:      c24a=c24a; //not ok should be this.c24a=c24a
 15:      c24a=1; //not ok, should be this.c24a=1
 16: }
 17://Check27:fields not private
 18: public int c27a; //not ok
 19: private int c27b; //ok
 20: protected int c27c; //not ok
 21: int c27d; //not ok
 22://Check28:class field not access via class name
 23: private static int c28a; //class field
 24: private int c28b; //instance field
 25://Check29:Missing field or get or set methods
```

```
26: private int c29a;
27: //no getC29a()
28: public void setC29a(int c29a) {this.c29a=c29a;}
29:
30: private int c29b;
31: public int getC29b() {return c29b;}
32: //no setC29b()
33:
34: //no field c29c
35: public int getC29c() {return 1;}
36: public void setC29c(int c29c) {}
37://Check30:Field shadowing
38: private int cv30; //shadows class field in parent
39: private boolean iv30, iv30a;
    //field shadowed, not shadowed
40://Check32:no super.finalize() call
41: public void finalize() throws Throwable
42: {
43:      //super.finalize(); not called
44: }
45://Check33:class and superclass method overloading
46: public void doTest33(int a)
    //parent is void doTest33(float a)
47: {
48: }
49://Check34:missing overridden Object methods
50://public abstract boolean equals(Object o);
51://public abstract Object clone() throws
    CloneNotSupportedException;
52://public String toString() { return ""; }
53://public abstract void finalize() throws
    Throwable;
54://public abstract int hashcode();
55://Check42 & Check43:long literal with L specifier
56: private static long m43(long a, int b, long c)
57: {
58:      return 1; //missing L at end
59: }
60:
61: public static void main(String[] args)
62:
63://Turn on hard coded literals
64://#JSCAT CHECK Check1=on
65://Check1:hard coded literals
66:      String A="Hard Coded Literal";
67://Turn off hard coded literals
68://#JSCAT CHECK Check1=off
```

```
 69://Check2:octal digit expected
 70:      String A0="\\12 Z"; //not 3 octal digits
 71:      String A1="\\128"; //3rd digit non-octal
 72://Check3:too many octal digits
 73:      String A2="\\123:\\1234";
          //octal 123, ':', octal 123, '4'
 74://Check4:too many hex digits with unicode
 75:      String A3="\\\uABCDE"; //unicode ABCD, 'E'
 76://Check6:lower case l (el) long value literal
     indicator used
 77:      long A5=0x1111111; //lower case l (el) at
          end should be L
 78://Check8:Var dims in both type and var
 79:      String[] arrayOfArrayOfStrings[];
 80://Check10:conditional operator without parenthesis
 81:      int A6 = -1;
 82:      int A7 = 0;
 83:      //A7 < 0 not parenthesized
 84:      int A8 = (A6 >=0) ? A6 : A7 < 0 ? -A6 : 5;
 85://Check11:embedded assignment
 86:      int A9 = 1, A10 = 1, A11 = 1, A12 = 1;
 87:      A11=(A9=A10+A11)+A12;
 88://Check12:loss of data castings
 89:      int A13 = 0;
 90:      byte A14 = 1;
 91:      float A15 = (float)(A13+A14);
 92://Check13:assignment instead of equality operator
 93:      boolean A16=false;
 94:      boolean A17=true;
 95:      if (A17=A16) {}
 96://Check14:equality instead of assignment operator
 97:      boolean A18=false;
 98:      boolean A19=true;
 99:      boolean A20=A18==A19;
100://Check15:object compare not using equals method
101:      String A21="", A22="";
102:      if (A21==A22) {}
103:      if (A21!=A22) {}
104://Check16:pre/post incr/decr in complex
     expressions
105:      int A23 = 0;
106:      A23 = A23++ * 3;
107://Check17:use of right shift warning
108:      int A24=0;
109:      int A25=A24>>1;
110:      A25=A24>>>1;
111://Check18:expression evaluation order
```

```
112:      int a=0, b=0, c=0, d=0, e=0;
113:      a=a+b*c/d-e;     // ((a+((b*c)/d))-e)
114:      a=(a+b*c/d)-e;   // ((a+((b*c)/d))-e)
115:      a=(a+b)*(c/d)-e; // (((a+b)*(c/d))-e)
116:      a=-a+b*c/d-e;    // (((-a)+((b*c)/d))-e)
117://Check19:conditional expression short-circuit
     evaluation
118:      Object A26=null;
119:      if (A26 != null && A26 instanceof Object) {}
120:      int A27=0;
121:      int A28=0;
122:      int[] A29 = new int[10];
123:      if (A27 < 3 || A29[A28++]==4) {}
124://Check20:statements not in a block
125:      int A30=0;
126:      if (A30==1)
127:           A30=0;
128:           A30=1;
129://Check21:empty statement followed by a block
130:      int A31=0,A32=0;
131:      if (A31 > A32); //notice empty (;) statement
132:      { //this block is independent of if stmt
133:           A31=1;
134:      }
135://Check22:dangling-else problem
136:      int A33=1,A34=0;
137:      if (A33 != 0) //next if-else a single stmt
          w/this if
138:           if (A34==0) A34=2; //notice single
               empty statement
139:      else A33=2; //this else really goes with
                         A34==0
140://Check23:missing break statement before case or
     default clauses
141:      char A35='+';
142:      int A36=1;
143:      switch (A35)
144:      {
145:           case '+':
146:           case '-':
147:                A36=2;
148:                break;
149:           case '*':
150:                A36=3;
151:           case '/':
152:                switch (A36)
153:                {
```

```
154:                    case 0:
155:                    case 1:
156:                    case 2:
157:                    default:
158:                }
159:           default:
160:                break;
161:      }
162://Check25:unguarded object castings
163:      ArrayList A37 = new ArrayList();
164:      //get object out of list & cast to String
165:      //should have a try/catch ClassCastException
166:      //because object in list may not be castable
          to a String.
167:      String A38 = (String)A37.get(0);
168:
169:      //this is ok, in ClassCastException
          hierarchy but not best.
170:      try { A38=(String)A37.get(0); }
171:      catch (Exception e1) {}
172:
173:      try { A38=(String)A37.get(0); } //ok.
174:      catch (ClassCastException e1) {}
175:
176:      try
177:      {
178:       try { A38=(String)A37.get(0); }
179:       catch(ConcurrentModificationException e2){}
180:      }
181:      catch (ClassCastException e1) {}
          //catch in an outer try
182://Check26:array access index loop not using array
     length to end
183:      int A39[]=new int[10];
184:      int i1=0;
185:      for (i1=0; i1<10; i1++)
186:      {
187:           A39[i1]=i1;
188:      }
189:      i1=0;
190:      while (i1<10)
191:      {
192:           A39[i1]=i1;
193:           i1++;
194:      }
195:      i1=0;
196:      do
```

```
197:      {
198:           A39[i1]=i1;
199:           i1++;
200:      } while (i1<10);
201://Check28:class field not accessed via class name
202:      Sample s1 = new Sample();
203:      s1.c28a=1; //not ok
204:      s1.c28b=1; //ok
205:      Sample.c28a=1; //ok
206://Check31:local variable field shadowing
207:      int cv30; //shadows class field parent
208:      boolean iv30, iv30a;
          //shadowed, not shadowed
209://Check35,37:no ArrayIndexOutOfBoundsException
     empty catch block
210:      int[] A40 = {1, 2, 3, 4, 5};
211:      int i2 = -1;
212:      int A41=0;
213:      A41=A40[i2]; //unguarded array access
214:      try
215:      {
216:           A41=A40[i2];
217:      }
218:      catch (ArrayIndexOutOfBoundsException e3)
219:      {} //empty catch block
220://Check36, 37:no StringIndexOutOfBoundsException
     empty catch block
221:      String A42="hello";
222:      char A43=A42.charAt(0);
          //unguarded string index access
223:      try
224:      {
225:           A43=A42.charAt(0);
226:      }
227:      catch (StringIndexOutOfBoundsException e4)
228:      {;;} //essentially an empty catch block
229://Check38:catch block with only output type
     statements
230:      String A44="hello";
231:      char A45=A44.charAt(0);
232:      try
233:      {
234:           A45=A44.charAt(0);
235:      }
236:      catch (StringIndexOutOfBoundsException e5)
237:      {
238:           System.out.println("");
```

```
239:           System.err.println("");
240:           e5.printStackTrace();
241:      }
242://Check40:long declaration without L specifier
243:      long A46=2147483647;
          //missing L at end for declaration
244://Check41:assignment without L specifier
245:      A46=2147483647;
          //missing L at end for assignment
246://Check43:method call args without L specifier
247:      m43(1, 1, 1L); //1st arg is missing L at end
248: }
249:}
```

Configuration File

```
File:JScat.cfg
#JSCAT SETTING OUTPUT_ASTBUILD_INFO=off
#JSCAT SETTING OUTPUT_GENERAL_INFO=on
#JSCAT SETTING OUTPUT_REGISTER_INFO=off
#JSCAT SETTING OUTPUT_VISIT_INFO=off
#JSCAT SETTING OUTPUT_VISIT_SOURCE_INFO=off
#JSCAT SETTING OUTPUT_TYPE_INFO=off
#JSCAT SETTING OUTPUT_UNIT_INFO=off
#JSCAT SETTING OUTPUT_SCOPE_INFO=off
#JSCAT SETTING OUTPUT_CONTEXT_INFO=off
#JSCAT SETTING OUTPUT_CHECK_INFO=off
#JSCAT SETTING OUTPUT_SYMBOL_INFO=off
#JSCAT SETTING OUTPUT_CHECKMSG_INFO=on
#JSCAT SETTING OUTPUT_SUMMARY_STATISTIC_INFO=on
#JSCAT SETTING OUTPUT_DETAILED_ASTBUILT_STATISTIC=on
#JSCAT SETTING OUTPUT_DETAILED_ASTVISIT_STATISTIC=on
#JSCAT SETTING OUTPUT_DETAILED_PROBLEM_STATISTIC=on
#JSCAT SETTING OUTPUT_EXCEPTION=on
#JSCAT SETTING OUTPUT_EXCEPTION_STACKTRACE=off
#JSCAT SETTING OUTPUT_DEBUG_INFO=off
#JSCAT SETTING QUICKREGISTER=off
PragmaCheck
Check1 //Hard-coded values
Check2 //Octal digit expected
Check3 //Too many octal digits
Check4 //Too many hex digits with Unicode character
Check6 //Lower-case long value indicator used
Check8 //Multi-dim array syntax coded with type and name
Check9 //Multi-dim array syntax coded with type and name
```

```
Check10 //Conditional operator Boolean without parens
Check11 //Embedded assignments
Check11a //Check11 helper
Check12 //Loss of data castings
Check13 //Assignment instead of equality operator
Check13a //Check13 helper
Check14 //Equality instead of assignment operator
Check14a //Check14 helper
Check15 //Object compare with operators instead of equals()
Check16 //Pre/post-Incr/Decr in complex expression
Check16a //Check16 helper
Check17 //Using signed and unsigned right shift operators
Check18 //Order of expression evaluation
Check19 //Conditional expression with short-circuit eval
Check20 //Statement not in a block
Check21 //Empty statement followed by a block
Check22 //Dangling-else problem
Check23 //Missing break statement before case of default
Check24 //Assignment of field & locals that have same name
Check25 //Unguarded object castings
Check26 //Array access w/looping index var not using length
Check27 //Class & instance field not private
Check28 //Class fields not accessed via class name
Check29 //Field without getter & setter methods
Check29a //Check29 helper
Check29b //Check29 helper
Check30 //Field shadowing
Check31 //Local variable field shadowing
Check32 //Method finalize not calling super.finalize()
Check32a //Check32 helper
Check33 //Class & super class method overload
Check34 //Classes without equals,clone,finalize,… methods
Check34a //Check34 helper
Check35 //Missing ArrayIndexOutOfBoundsException
Check36 //Missing StringIndexOutOfBoundsException
Check37 //Empty catch block
Check38 //Catch block with only output type statements
Check40 //Long data type literal without long indicator
Check41 //Long data type literal without long indicator
Check42 //Long data type literal without long indicator
Check43 //Long data type literal without long indicator
```

Command Session

```
>java JScatDriver C:\JScatChecks Sample.java C:\JScat
JScat.cfg
```

JScat Output

```
Source File=C:\JScatChecks\Sample.java
Config File=C:\JScat\JScat.cfg

Check24:Assignment right-hand side name is both a field and
an argument or local.
14:c24a=c24a; //not ok, should be this.c24a=c24a
   ^^^^^^^^^

Check24:Assignment right-hand side name is both a field and
an argument or local.
15:c24a=1; //not ok, should be this.c24a=1
   ^^^^^^

Check27:Field access modifier is not private, thus field is
not completely hidden.
18:public int c27a; //not ok
   ^^^^^^^^^^^^^^^^

Check27:Field access modifier is not private, thus field is
not completely hidden.
20:protected int c27c; //not ok
   ^^^^^^^^^^^^^^^^^^

Check27:Field access modifier is not private, thus field is
not completely hidden.
21:int c27d; //not ok
   ^^^^^^^^^

Check30:Field cv30 is shadowed in JScatChecks.SampleParent
38:private int cv30; //shadows class field in parent
   ^^^^^^^^^^^^^^^^

Check30:Field iv30 is shadowed in JScatChecks.SampleParent
39:private boolean iv30, iv30a;
//field shadowed, not shadowed
   ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Check32:A super.finalize(); was not found within the void
finalize() method.
41:public void finalize() throws Throwable
   ^

Check33:Method name doTest33 matches a super class method
name, but args do not match.
This could be an incorrect attempt at a method override.
Sub-class method signature:public void doTest33(int
a)Super-class method signature:public void
JScatChecks.SampleParent.doTest33(float)
46:public void doTest33(int a) //parent is void
doTest33(float a)
   ^

Check42:Return literal should end with a capital L when
return type is long.
58:return 1; //missing L at end
          ^

Check1:Use a constant declaration instead of a hard-coded
value.
66:String A="Hard Coded Literal";
            ^^^^^^^^^^^^^^^^^^

Check2:Suspect octal escape sequence in string; should code
3 octal digits after \.
\12
70:String A0="\\12 Z"; //not using 3 octal digits
               ^^^^^

Check2:Suspect octal escape sequence in string; last digit
is non-octal.
\128
Octal escape sequences should use three octal digits.
71:String A1="\\128"; //3rd digit non-octal
               ^^^^

Check3:Suspect octal escape sequence in string; four (4)
octal digits specified.\1234
73:String A2="\\123:\\1234"; //octal 123, ':', octal 123,
'4'
                ^^^^^^^^^^
```

```
Check4:Suspect unicode escape sequence in string; five (5)
hex digits specified.\uABCDE
75:String A3="\\\uABCDE"; //unicode ABCD, 'E'
               ^^^^^^^^

Check6:Long Literals should end with capital L not
lowercase l; aids in readability.
77:long A5=0x1111111; //lower case l (el) at end should be
L
           ^^^^^^^^^

Check8:Multiple dimensions should be coded either with the
data type or the array name, not both.
79:String[] arrayOfArrayOfStrings[];
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Check10:Parenthesize the expression A7 < 0 on the left of
the ? operator in a ternary condition (?:) expression.
84:int A8 = (A6 >=0) ? A6 : A7 < 0 ? -A6 : 5;
                            ^^^^^^^^^^^^^^^^

Check18:Expression evaluation order:((A9=(A10+A11))+A12)
87:A11=(A9=A10+A11)+A12;
   ^^^^^^^^^^^^^^^^^^^

Check11:Avoid assigning several variables in the same
statement
87:A11=(A9=A10+A11)+A12;
   ^^^^^^^^^^^^^^^^^^^^

Check18:Expression evaluation order:(((A13+A14))
91:float A15 = (float)(A13+A14);
               ^^^^^^^^^^^^^^^^

Check12:Casting int to float is automatic widening but some
least significant digits may be lost.
91:float A15 = (float)(A13+A14);
               ^^^^^^^^^^^^^^^^

Check13:An = was used in conditional expression, probably
should be ==
95:if (A17=A16) {}
       ^^^^^^^
```

```
Check14:An == was used in an assignment expression, maybe
should be =
99:boolean A20=A18==A19;
           ^^^^^^^^^^^^

Check15:Object comparison with == operator instead of
.equals()
102:if (A21==A22) {}
        ^^^^^^^^

Check15:Object comparison with != operator instead of
.equals()
103:if (A21!=A22) {}
        ^^^^^^^^

Check18:Expression evaluation order:((A23++)*3)
106:A23 = A23++ * 3;
    ^^^^^^^^^^^^^^^

Check16:Avoid ++ or -- in expressions with other operators.
106:A23 = A23++ * 3;
    ^^^^^^^^^^^^^^^^

Check17:The >> operator shifts the bits of the left operand
to the right by the number of places specified by the right
operand.
The low bits of the left operand are shifted away & lost.
The high bits shifted in are the same as the original high-
order bit of the left operand.
The is known are sign extension. Positive numbers shift in
a zero, negative numbers shift in a one.
109:int A25=A24>>1;
            ^^^^^^

Check17:The >>> operator shifts the bits of the left
operand to the right by the number of places specified by
the right operand.
The low bits of the left operand are shifted away & lost.
The high bits shifted in are always zero, this is zero
extension.
The low-order bits of the left operand are shifted away and
lost.
Zero extension is appropriate when the left operand is
being treated as an unsigned value.
110:A25=A24>>>1;
        ^^^^^^^
```

```
Check18:Expression evaluation order:((a+((b*c)/d))-e)
113:a=a+b*c/d-e;     // ((a+((b*c)/d))-e)
    ^^^^^^^^^^^

Check18:Expression evaluation order:((a+((b*c)/d))-e)
114:a=(a+b*c/d)-e;   // ((a+((b*c)/d))-e)
    ^^^^^^^^^^^^^

Check18:Expression evaluation order:(((a+b)*(c/d))-e)
115:a=(a+b)*(c/d)-e; // (((a+b)*(c/d))-e)
    ^^^^^^^^^^^^^^^

Check18:Expression evaluation order:(((-a)+((b*c)/d))-e)
116:a=-a+b*c/d-e;    // (((-a)+((b*c)/d))-e)
    ^^^^^^^^^^^^

Check18:Expression evaluation
order:((A26!=null)&&(A26instanceofObject))
119:if (A26 != null && A26 instanceof Object) {}
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Check19:If the left operand {A26 != null} is false then the
right operand {A26 instanceof Object} is not evaluated at
all.
If the right operand has side-effects then they will not be
done in this situation.
119:if (A26 != null && A26 instanceof Object) {}
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Check18:Expression evaluation
order:((A27<3)||(A29[A28++]==4))
123:if (A27 < 3 || A29[A28++]==4) {}
        ^^^^^^^^^^^^^^^^^^^^^^^^

Check19:If the left operand {A27 < 3} is true then the
right operand {A29[A28++] == 4} is not evaluated at all.
If the right operand has side-effects then they will not be
done in this situation.
123:if (A27 < 3 || A29[A28++]==4) {}
        ^^^^^^^^^^^^^^^^^^^^^^^^

Check35:Unguarded array access; place within a try/catch
(ArrayIndexOutOfBoundsException) structure.
123:if (A27 < 3 || A29[A28++]==4) {}
                   ^^^^^^^^^^
```

```
Check16:Avoid ++ or -- in expressions with other operators.
123:if (A27 < 3 || A29[A28++]==4) {}
                       ^^^^^

Check20:A block {} should be used with constructs that can
have one or more statements.
Only
 if (A30 == 1)
  A30=0;
are associated together.
Should be coded as:
 if (A30 == 1)
 {
  A30=0;
 }
126:if (A30==1)
    ^^^^^^^
Check20:A block {} should be used with constructs that can
have one or more statements.
Only
 if (A31 > A32)
  ;
are associated together.
Should be coded as:
 if (A31 > A32)
 {
  ;
 }
131:if (A31 > A32); //notice empty (;) statement here
    ^^^^^^^^^^^^^^^

Check21:Empty statement followed by a block is possible an
erroneous ;
131:      if (A31 > A32); //notice empty (;) statement here
132:{ //this block is independent of if statement
    ^

Check20:A block {} should be used with constructs that can
have one or more statements.
Only
 if (A33 != 0)
  if (A34 == 0)
   A34=2;
  else
   A33=2;
are associated together.
Should be coded as:
```

```
 if (A33 != 0)
 {
  if (A34 == 0)
   A34=2;
  else
   A33=2;
 }
137:if (A33 != 0) //next if-else a single stmt w/this if
    ^^^^^^^^^^^

Check22:Else statement goes with:
if (A34 == 0)
 A34=2;
else
 A33=2;
NOT:
if (A33 != 0)
137:if (A33 != 0) //next if-else a single stmt w/this if
    ^^^^^^^^^^^

Check20:A block {} should be used with constructs that can
have one or more statements.
Only
 if (A34 == 0)
  A34=2;
are associated together.
Should be coded as:
 if (A34 == 0)
 {
  A34=2;
 }
138:if (A34==0) A34=2; //notice single empty statement
    ^^^^^^^^^^^^^^^^^

Check20:A block {} should be used with constructs that can
have one or more statements.
Only
 else
  A33=2;
are associated together.
Should be coded as:
 else
 {
  A33=2;
 }
139:else A33=2; //this else really goes with A34==0
    ^^^^^^^^^^^
```

```
Check23:No break statement with case or default.
145:case '+':
    ^^^^^^^^^

Check23:No break statement with case or default.
149:case '*':
    ^^^^^^^^^

Check23:No break statement with case or default.
151:case '/':
    ^^^^^^^^^

Check23:No break statement with case or default.
154:case 0:
    ^^^^^^^

Check23:No break statement with case or default.
155:case 1:
    ^^^^^^^

Check23:No break statement with case or default.
156:case 2:
    ^^^^^^^

Check23:No break statement with case or default.
157:default:
    ^^^^^^^^

Check25:Unguarded cast; place within a try/catch
(ClassCastException) structure.
167:String A38 = (String)A37.get(0);
                 ^^^^^^^^^^^^^^^^^^

Check25:An explicit catch (ClassCastException) should be
coded.
170:try { A38=(String)A37.get(0); }
             ^^^^^^^^^^^^^^^^^^

Check37:Catch block is empty, should code proper exception
handling logic.
171:catch (Exception e1) {}
    ^^^^^^^^^^^^^^^^^^^^^^

Check37:Catch block is empty, should code proper exception
handling logic.
174:catch (ClassCastException e1) {}
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Check37:Catch block is empty, should code proper exception
handling logic.
179:catch (ConcurrentModificationException e2) {}
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Check37:Catch block is empty, should code proper exception
handling logic.
181:catch (ClassCastException e1) {} //catch in an outer
try
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Check35:Unguarded array access; place within a try/catch
(ArrayIndexOutOfBoundsException) structure.
187:A39[i1]=i1;
    ^^^^^^^

Check35:Unguarded array access; place within a try/catch
(ArrayIndexOutOfBoundsException) structure.
192:A39[i1]=i1;
    ^^^^^^^

Check35:Unguarded array access; place within a try/catch
(ArrayIndexOutOfBoundsException) structure.
198:A39[i1]=i1;
    ^^^^^^^

Check28:Refer to class fields using the class name, not an
object reference.
203:s1.c28a=1; //not ok
    ^^^^^^^

Check31:Local variable cv30 is shadowed in
JScatChecks.SampleParent
207:int cv30; //shadows class field parent
    ^^^^^^^^^

Check31:Local variable iv30 is shadowed in
JScatChecks.SampleParent
208:boolean iv30, iv30a; //shadowed, not shadowed
    ^^^^^^^^^^^^^^^^^^^

Check35:Unguarded array access; place within a try/catch
(ArrayIndexOutOfBoundsException) structure.
213:A41=A40[i2]; //unguarded array access
        ^^^^^^^
```

```
Check37:Catch block is empty, should code proper exception
handling logic.
218:catch (ArrayIndexOutOfBoundsException e3)
    ^^^^^

Check36:Unguarded string index access; place within a
try/catch (StringIndexOutOfBoundsException) structure.
222:char A43=A42.charAt(0); //unguarded string index access
             ^^^^^^^^^^^^^

Check37:Catch block has only empty statements, should code
proper exception handling logic.
227:catch (StringIndexOutOfBoundsException e4)
    ^^^^^

Check38:Catch block has only output statements, should code
proper exception handling logic.
227:catch (StringIndexOutOfBoundsException e4)
    ^^^^^

Check36:Unguarded string index access; place within a
try/catch (StringIndexOutOfBoundsException) structure.
231:char A45=A44.charAt(0);
             ^^^^^^^^^^^^^

Check38:Catch block has only output statements, should code
proper exception handling logic.
236:catch (StringIndexOutOfBoundsException e5)
    ^^^^^

Check40:Identifer's of type long assignment literal should
end with a capital L.
243:long A46=2147483647; //missing L at end for declaration
             ^^^^^^^^^^

Check41:Identifer's of type long assignment literal should
end with a capital L.
245:A46=2147483647; //missing L at end for assignment
        ^^^^^^^^^^

Check43:Identifer's of type long assignment literal should
end with a capital L.
247:m43(1, 1, 1L); //1st arg is missing L at end
        ^

Check29:Field c24a has no corresponding method getC24a
Check29:Field c24b has no corresponding method getC24b
```

```
Check29:Field c27a has no corresponding method getC27a
Check29:Field c27b has no corresponding method getC27b
Check29:Field c27c has no corresponding method getC27c
Check29:Field c27d has no corresponding method getC27d
Check29:Field c28a has no corresponding method getC28a
Check29:Field c28b has no corresponding method getC28b
Check29:Field c29a has no corresponding method getC29a
Check29:Method getC29c has no corresponding field c29c
Check29:Field cv30 has no corresponding method getCv30
Check29:Field iv30 has no corresponding method getIv30
Check29:Field iv30a has no corresponding method getIv30a
Check29:Method setC24 has no corresponding field c24
Check29:Field c24a has no corresponding method setC24a
Check29:Field c24b has no corresponding method setC24b
Check29:Field c27a has no corresponding method setC27a
Check29:Field c27b has no corresponding method setC27b
Check29:Field c27c has no corresponding method setC27c
Check29:Field c27d has no corresponding method setC27d
Check29:Field c28a has no corresponding method setC28a
Check29:Field c28b has no corresponding method setC28b
Check29:Field c29b has no corresponding method setC29b
Check29:Method setC29c has no corresponding field c29c
Check29:Field cv30 has no corresponding method setCv30
Check29:Field iv30 has no corresponding method setIv30
Check29:Field iv30a has no corresponding method setIv30a
Check34:Method equals() not found in class declaration
Check34:Method clone() not found in class declaration
Check34:Method toString() not found in class declaration
Check34:Method hashcode() not found in class declaration

** Statistics **
Number of source lines = 250

Number of ASTNodeList AST nodes built = 144
Number of AddExpr AST nodes built = 7
Number of ArgList AST nodes built = 11
Number of ArrayAccess AST nodes built = 6
Number of ArrayDimList AST nodes built = 4
Number of ArrayIdentifier AST nodes built = 2
Number of ArrayInitSingle AST nodes built = 1
Number of Assignment AST nodes built = 38
Number of AssignmtOp AST nodes built = 38
Number of Block AST nodes built = 32
Number of BlockStmtList AST nodes built = 23
Number of BreakStmt AST nodes built = 2
Number of CITQualifiedIdentifier AST nodes built = 10
Number of CatchList AST nodes built = 7
```

```
Number of CatchStmt AST nodes built = 7
Number of ClassBody AST nodes built = 1
Number of ClassBodyDeclList AST nodes built = 1
Number of ClassDecl AST nodes built = 1
Number of ClassType AST nodes built = 1
Number of ClassTypeList AST nodes built = 1
Number of CompilationUnit AST nodes built = 1
Number of CondAndExpr AST nodes built = 1
Number of CondExpr AST nodes built = 2
Number of CondOrExpr AST nodes built = 1
Number of Conditional AST nodes built = 12
Number of CreateObject AST nodes built = 2
Number of DimExpr AST nodes built = 2
Number of DimExprList AST nodes built = 2
Number of DivExpr AST nodes built = 4
Number of DoStmt AST nodes built = 1
Number of ElseStmt AST nodes built = 1
Number of EmptyStmt AST nodes built = 3
Number of EqualExpr AST nodes built = 5
Number of Expr AST nodes built = 118
Number of ExprCast AST nodes built = 4
Number of ExprStmt AST nodes built = 41
Number of ExtendsStmt AST nodes built = 1
Number of FieldAccess AST nodes built = 2
Number of FieldDecl AST nodes built = 12
Number of ForStmt AST nodes built = 1
Number of FormalParm AST nodes built = 15
Number of FormalParmList AST nodes built = 6
Number of GreaterThanEqualExpr AST nodes built = 1
Number of GreaterThanExpr AST nodes built = 1
Number of Identifier AST nodes built = 239
Number of IfElseStmt AST nodes built = 1
Number of IfStmt AST nodes built = 9
Number of ImportDeclList AST nodes built = 1
Number of ImportDeclMulti AST nodes built = 1
Number of InstanceOfExpr AST nodes built = 1
Number of LRBrack AST nodes built = 4
Number of LessThanExpr AST nodes built = 5
Number of Literal AST nodes built = 104
Number of LocalVar AST nodes built = 48
Number of LocalVarDecl AST nodes built = 48
Number of MethodBody AST nodes built = 9
Number of MethodCall AST nodes built = 12
Number of MethodDecl AST nodes built = 9
Number of MethodHeader AST nodes built = 9
Number of MethodNameArgs AST nodes built = 9
Number of ModifiersList AST nodes built = 21
```

```
Number of MultExpr AST nodes built = 5
Number of Name AST nodes built = 26
Number of NotEqualExpr AST nodes built = 3
Number of NullOpt AST nodes built = 34
Number of PackageDecl AST nodes built = 1
Number of ParenExpr AST nodes built = 6
Number of PostIncrExpr AST nodes built = 5
Number of Pragma AST nodes built = 23
Number of PrimitiveArray AST nodes built = 2
Number of PrimitiveArrayCreate AST nodes built = 2
Number of PrimitiveCast AST nodes built = 1
Number of PrivateMod AST nodes built = 11
Number of ProtectedMod AST nodes built = 1
Number of PublicMod AST nodes built = 9
Number of QualifiedIdentifier AST nodes built = 17
Number of ReferenceArray AST nodes built = 2
Number of ReturnStmt AST nodes built = 3
Number of RightShiftExpr AST nodes built = 1
Number of StaticMod AST nodes built = 3
Number of StmtExprList AST nodes built = 2
Number of SubExpr AST nodes built = 4
Number of SwitchBlockBlocksAndLabels AST nodes built = 1
Number of SwitchBlockGroup AST nodes built = 4
Number of SwitchBlockGroupList AST nodes built = 1
Number of SwitchBlockLabelsOnly AST nodes built = 1
Number of SwitchCase AST nodes built = 7
Number of SwitchDefault AST nodes built = 2
Number of SwitchLabelList AST nodes built = 5
Number of SwitchStmt AST nodes built = 2
Number of This AST nodes built = 2
Number of ThrowsStmt AST nodes built = 1
Number of TokenAndAnd AST nodes built = 1
Number of TokenBoolean AST nodes built = 7
Number of TokenBooleanLit AST nodes built = 4
Number of TokenBreak AST nodes built = 2
Number of TokenByte AST nodes built = 1
Number of TokenCase AST nodes built = 7
Number of TokenCatch AST nodes built = 7
Number of TokenChar AST nodes built = 7
Number of TokenChrLit AST nodes built = 5
Number of TokenClass AST nodes built = 1
Number of TokenColon AST nodes built = 11
Number of TokenComma AST nodes built = 20
Number of TokenDecIntLit AST nodes built = 80
Number of TokenDecLngLit AST nodes built = 1
Number of TokenDefault AST nodes built = 2
Number of TokenDiv AST nodes built = 4
```

```
Number of TokenDo AST nodes built = 1
Number of TokenDot AST nodes built = 26
Number of TokenElse AST nodes built = 1
Number of TokenEq AST nodes built = 95
Number of TokenEqEq AST nodes built = 5
Number of TokenExtends AST nodes built = 1
Number of TokenFloat AST nodes built = 2
Number of TokenFor AST nodes built = 1
Number of TokenGt AST nodes built = 1
Number of TokenGtEq AST nodes built = 1
Number of TokenHexLngLit AST nodes built = 1
Number of TokenId AST nodes built = 280
Number of TokenIf AST nodes built = 9
Number of TokenImport AST nodes built = 1
Number of TokenInstanceof AST nodes built = 1
Number of TokenInt AST nodes built = 66
Number of TokenLBrace AST nodes built = 36
Number of TokenLBrack AST nodes built = 14
Number of TokenLParen AST nodes built = 55
Number of TokenLong AST nodes built = 9
Number of TokenLt AST nodes built = 5
Number of TokenMinus AST nodes built = 8
Number of TokenMult AST nodes built = 6
Number of TokenNew AST nodes built = 4
Number of TokenNotEq AST nodes built = 3
Number of TokenNull AST nodes built = 4
Number of TokenOrOr AST nodes built = 1
Number of TokenPackage AST nodes built = 1
Number of TokenPlus AST nodes built = 7
Number of TokenPlusPlus AST nodes built = 5
Number of TokenPrivate AST nodes built = 11
Number of TokenProtected AST nodes built = 1
Number of TokenPublic AST nodes built = 9
Number of TokenQuestion AST nodes built = 2
Number of TokenRBrace AST nodes built = 36
Number of TokenRBrack AST nodes built = 14
Number of TokenRParen AST nodes built = 55
Number of TokenRShift AST nodes built = 1
Number of TokenReturn AST nodes built = 3
Number of TokenSemicolon AST nodes built = 114
Number of TokenStatic AST nodes built = 3
Number of TokenStrLit AST nodes built = 11
Number of TokenSwitch AST nodes built = 2
Number of TokenThis AST nodes built = 2
Number of TokenThrows AST nodes built = 1
Number of TokenTry AST nodes built = 7
Number of TokenURShift AST nodes built = 1
```

```
Number of TokenVoid AST nodes built = 6
Number of TokenWhile AST nodes built = 2
Number of TryStmt AST nodes built = 7
Number of TypeBoolean AST nodes built = 7
Number of TypeByte AST nodes built = 1
Number of TypeChar AST nodes built = 7
Number of TypeDeclList AST nodes built = 1
Number of TypeFloat AST nodes built = 2
Number of TypeInt AST nodes built = 66
Number of TypeLong AST nodes built = 9
Number of TypeNull AST nodes built = 2
Number of TypeVoid AST nodes built = 6
Number of UnaryMinusExpr AST nodes built = 4
Number of UnsignedRightShiftExpr AST nodes built = 1
Number of VarAccess AST nodes built = 106
Number of VarDecl AST nodes built = 30
Number of VarDeclInit AST nodes built = 57
Number of VarDeclList AST nodes built = 60
Number of VarInit AST nodes built = 62
Number of VarInitList AST nodes built = 1
Number of WhileStmt AST nodes built = 1

Total number of AST nodes built = 2814

Number of AddExpr AST nodes visited = 7
Number of ArgList AST nodes visited = 11
Number of ArrayAccess AST nodes visited = 6
Number of ArrayInitSingle AST nodes visited = 1
Number of Assignment AST nodes visited = 38
Number of Block AST nodes visited = 32
Number of BlockStmtList AST nodes visited = 23
Number of BreakStmt AST nodes visited = 2
Number of CatchList AST nodes visited = 7
Number of CatchStmt AST nodes visited = 7
Number of ClassBody AST nodes visited = 1
Number of ClassBodyDeclList AST nodes visited = 1
Number of ClassDecl AST nodes visited = 1
Number of ClassType AST nodes visited = 1
Number of ClassTypeList AST nodes visited = 1
Number of CompilationUnit AST nodes visited = 1
Number of CondAndExpr AST nodes visited = 1
Number of CondExpr AST nodes visited = 2
Number of CondOrExpr AST nodes visited = 1
Number of Conditional AST nodes visited = 12
Number of CreateObject AST nodes visited = 2
Number of DimExpr AST nodes visited = 2
Number of DimExprList AST nodes visited = 2
```

```
Number of DivExpr AST nodes visited = 4
Number of DoStmt AST nodes visited = 1
Number of ElseStmt AST nodes visited = 1
Number of EmptyStmt AST nodes visited = 3
Number of EqualExpr AST nodes visited = 5
Number of Expr AST nodes visited = 118
Number of ExprCast AST nodes visited = 4
Number of ExprStmt AST nodes visited = 41
Number of ExtendsStmt AST nodes visited = 1
Number of FieldAccess AST nodes visited = 2
Number of FieldDecl AST nodes visited = 12
Number of ForStmt AST nodes visited = 1
Number of FormalParm AST nodes visited = 15
Number of FormalParmList AST nodes visited = 6
Number of GreaterThanEqualExpr AST nodes visited = 1
Number of GreaterThanExpr AST nodes visited = 1
Number of IfElseStmt AST nodes visited = 1
Number of IfStmt AST nodes visited = 9
Number of ImportDeclList AST nodes visited = 1
Number of ImportDeclMulti AST nodes visited = 1
Number of InstanceOfExpr AST nodes visited = 1
Number of LessThanExpr AST nodes visited = 5
Number of Literal AST nodes visited = 104
Number of LocalVar AST nodes visited = 48
Number of LocalVarDecl AST nodes visited = 48
Number of MethodBody AST nodes visited = 9
Number of MethodCall AST nodes visited = 12
Number of MethodDecl AST nodes visited = 9
Number of MethodHeader AST nodes visited = 9
Number of MethodNameArgs AST nodes visited = 9
Number of MultExpr AST nodes visited = 5
Number of NotEqualExpr AST nodes visited = 3
Number of NullOpt AST nodes visited = 28
Number of PackageDecl AST nodes visited = 1
Number of ParenExpr AST nodes visited = 6
Number of PostIncrExpr AST nodes visited = 5
Number of PrimitiveArrayCreate AST nodes visited = 2
Number of PrimitiveCast AST nodes visited = 1
Number of ReturnStmt AST nodes visited = 3
Number of RightShiftExpr AST nodes visited = 1
Number of StmtExprList AST nodes visited = 2
Number of SubExpr AST nodes visited = 4
Number of SwitchBlockBlocksAndLabels AST nodes visited = 1
Number of SwitchBlockGroup AST nodes visited = 4
Number of SwitchBlockGroupList AST nodes visited = 1
Number of SwitchBlockLabelsOnly AST nodes visited = 1
Number of SwitchCase AST nodes visited = 7
```

```
Number of SwitchDefault AST nodes visited = 2
Number of SwitchLabelList AST nodes visited = 5
Number of SwitchStmt AST nodes visited = 2
Number of This AST nodes visited = 2
Number of ThrowsStmt AST nodes visited = 1
Number of TokenBooleanLit AST nodes visited = 4
Number of TokenChrLit AST nodes visited = 5
Number of TokenDecIntLit AST nodes visited = 80
Number of TokenDecLngLit AST nodes visited = 1
Number of TokenHexLngLit AST nodes visited = 1
Number of TokenNull AST nodes visited = 2
Number of TokenStrLit AST nodes visited = 11
Number of TryStmt AST nodes visited = 7
Number of TypeDeclList AST nodes visited = 1
Number of UnaryMinusExpr AST nodes visited = 4
Number of UnsignedRightShiftExpr AST nodes visited = 1
Number of VarAccess AST nodes visited = 106
Number of VarDecl AST nodes visited = 30
Number of VarDeclInit AST nodes visited = 57
Number of VarDeclList AST nodes visited = 60
Number of VarInit AST nodes visited = 62
Number of VarInitList AST nodes visited = 1
Number of WhileStmt AST nodes visited = 1

Total number of AST nodes visited = 1165

Number of problems reported for Check1 = 1
Number of problems reported for Check10 = 1
Number of problems reported for Check11 = 1
Number of problems reported for Check12 = 1
Number of problems reported for Check13 = 1
Number of problems reported for Check14 = 1
Number of problems reported for Check15 = 2
Number of problems reported for Check16 = 2
Number of problems reported for Check17 = 2
Number of problems reported for Check18 = 9
Number of problems reported for Check19 = 2
Number of problems reported for Check2 = 2
Number of problems reported for Check20 = 5
Number of problems reported for Check21 = 1
Number of problems reported for Check22 = 1
Number of problems reported for Check23 = 7
Number of problems reported for Check24 = 2
Number of problems reported for Check25 = 2
Number of problems reported for Check27 = 3
Number of problems reported for Check28 = 1
Number of problems reported for Check29 = 27
```

```
Number of problems reported for Check3 = 1
Number of problems reported for Check30 = 2
Number of problems reported for Check31 = 2
Number of problems reported for Check32 = 1
Number of problems reported for Check33 = 1
Number of problems reported for Check34 = 4
Number of problems reported for Check35 = 5
Number of problems reported for Check36 = 2
Number of problems reported for Check37 = 6
Number of problems reported for Check38 = 2
Number of problems reported for Check4 = 1
Number of problems reported for Check40 = 1
Number of problems reported for Check41 = 1
Number of problems reported for Check42 = 1
Number of problems reported for Check43 = 1
Number of problems reported for Check6 = 1
Number of problems reported for Check8 = 2

Total number of problems reported = 108

Parse Time=1.622 seconds.
Visit Time=1.112 seconds.

Process completed.
```

LIST OF REFERENCES

[1] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 1-16, October 2000.

[2] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandrea: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 439-448, 2000.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software Practice and Experience*, volume 30 number 7, pages 775-802, 2000.

[4] S. Hangal and M. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 291-301, May 2002.

[5] V. Kuncak, P. Lam, and M. Rinard. Role Analysis. In *Conference Record of the 29th ACM Symposium of Principles of Programming Languages (POPL 2002)*, pages 17-32, January 2002.

[6] R. E. Storm and S. Yemini. Typestate a Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, volume 12 number 1, pages 157-171, January 1986.

[7] R. DeLine and M. Fahndrich. Enforcing High-level Protocols in Low-level Software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 59-69, June 2001.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, pages 220-242, June 1997.

[9] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI 1996)*, pages 3-17, 1996.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programming. *ACM Transactions on Computer Systems*, volume 15 number 4, pages 391-411, 1997.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 234-245, June 2002.

[12] D. Evans, J Guttag, J. Horning, and Y. M. Tan. LCLint: A Tool for Using Specification to Check Code. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT 1994)*, pages 87-96, December 1994.

[13] T. Lord. Application Specific Static Code Checking for C Programs: Ctool. July 14, 1998 ftp://ftp2.lanminds.com/lord/src/systas-1998-07-14.tar.gz. September 30, 2002.

[14] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the 1st Conference on Domain Specific Languages*, pages 229-242, October 1997.

[15] K. Knizhnik. AntiC/JLint. 1998 www.ispras.ru/~knizhnik/jlint/ReadMe.htm. September 30, 2002.

[16] Sureshot, Inc. JiveLint. 2000 www.bysoft.se/sureshot/javalint/. September 30, 2002.

[17] Webgain, Inc. *Webgain Studio Quality Analyzer User's Guide*. Webgain, Inc., Santa Clara, CA, 2001.

[18] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 69-82, June 2002.

[19] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *2002 IEEE Symposium on Security and Privacy*, pages 131-147, May 2002.

[20] Scott E. Hudson. JavaCUP User Manual. July, 1999 www.cs.princeton.edu/~appel/modern/java/CUP/manual.html. September 30, 2002.

[21] Gerwin Klein. JavaFlex. October 25, 2001. www.jflex.de. September 30, 2002.

[22] D. Engler, D. Y. Chen, S. Hallen, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings 18th Symposium on Operating System Principles (SOSP 2001)*, pages 57-72, ACM, 2001.

BIOGRAPHICAL SKETCH

As a seventh-generation native of Florida born in 1962 and raised in Gainesville, Florida, the University of Florida was a natural choice to continue my education after graduation from Buchholz High School in 1980. I earned and received a Bachelor of Science degree in Business Administration at the University of Florida in December of 1984. Computer and information sciences was the major area of study for the business degree. Since that time I have worked at Gainesville Regional Utilities in Gainesville, Florida, in various computer related positions. Currently I am a Systems Programmer with over ten years of experience and responsible for the primary business systems.

After graduating in 1984 and for the next seven years I also owned a software development business. Through that business I published several articles and programs in periodicals related to the Commodore 64 and 128 computer systems. I consulted and wrote educational software and developed a graphical language that extended the Commodore 128 BASIC language. This extension was groundbreaking in providing higher-resolution graphics than was exploited by the manufacturer for that computer.