NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A STATIC SECURE FLOW ANALYZER FOR A SUBSET OF JAVA

by

James D. Harvey

March, 1998

Thesis Advisor:	Dennis M. Volpano
Second Reader:	Craig W. Rasmussen

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 1998	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE A STATIC SECURE FLOW ANALYZER FOR A SUBSET OF JAVA	**5. FUNDING NUMBERS**
6. AUTHOR(S) Harvey James D.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

As the number of computers and computer systems in existence has grown over the past few decades, we have come to depend on them to maintain the security of private or sensitive information. The execution of a program may cause leaks of private or sensitive information from the computer. Static secure flow analysis is an attempt to detect these leaks prior to program execution.

It is possible to analyze programs by hand, but this is often impractical for large programs. A better approach is to automate the analysis, which is what this thesis explores.

We describe some previous research and give background information about secure flow analysis. A secure flow analyzer is presented. It implements a secure flow type inference algorithm, for a subset of Java 1.0.2, using a parser generator called Java Compiler Compiler (JavaCC). Semantic actions are inserted into a grammar specification to perform the secure flow analysis on a given program.

14. SUBJECT TERMS Secure Flow Analysis, Type Inference, Program Certification, Information Flow, Protection	**15. NUMBER OF PAGES** 98
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFI- CATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

A STATIC SECURE FLOW ANALYZER FOR A SUBSET OF JAVA

James D. Harvey
Lieutenant, United States Navy
B.S., The Ohio State University, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1998

Author: [illegible]
James D. Harvey

Approved by: Dennis M. Volpano
Dennis M. Volpano, Thesis Advisor

[illegible]
Craig W. Rasmussen, Second Reader

Dan C Boger
Dan Boger, Chairman
Department of Computer Science

ABSTRACT

As the number of computers and computer systems in existence has grown over the past few decades, we have come to depend on them to maintain the security of private or sensitive information. The execution of a program may cause leaks of private or sensitive information from the computer. Static secure flow analysis is an attempt to detect these leaks prior to program execution.

It is possible to analyze programs by hand, but this is often impractical for large programs. A better approach is to automate the analysis; which is what this thesis explores.

We describe some previous research and give background information about secure flow analysis. A secure flow analyzer is presented. It implements a secure flow type inference algorithm, for a subset of Java 1.0.2, using a parser generator called Java Compiler Compiler (JavaCC). Semantic actions are inserted into a grammar specification to perform the secure flow analysis on a given program.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

To Dr. Dennis Volpano, I would like to express my deepest thanks for being so patient while repeatedly explaining the required concepts. Your support, guidance, knowledge, instruction and ability to explain difficult concepts was instrumental to my being able to complete this thesis.

To Dr. Craig Rasmussen, I would like to express my sincere gratitude for your support, guidance, and dedication. Your attention to detail was essential through this process.

To my wife, Cindy, and daughter, Madison, who have cheerfully put up with late nights and missed dinners. I thank you for your support, understanding and devotion.

I. INTRODUCTION

The number of computers and computer networks has exploded over the past few decades, and computer security is a major concern. In a multi-level system where information exists with different security classifications, such as a military computer system, we want to protect information with a high security classification. It is desirable to have an automated tool to detect whether information we wish to keep secret in applications remains secret and is not leaked. This thesis introduces a program that will statically analyze a subset of Java programs to ensure that private information is not leaked.

A. SECURE INFORMATION FLOW

Verifying secure information flow within computer systems is necessary in order to protect sensitive information, especially in a military system. Denning and Denning state that information flow occurs from a storage object x to another storage object y when information stored in x is transferred to y, or used to derive information transferred to y. A flow may be either explicit or implicit [1].

Explicit information flow occurs when information is directly copied or transferred from one storage object to another. Consider the code segment "y := x". The information contained in x is directly copied into y, so information flows from x to y. The flow from x to y is independent of the value stored in x.

Implicit flow occurs when information is indirectly copied or transferred from one storage object to another. If the variable x contains either 0 or 1, then the following code segment will copy the value of x into y using an implicit flow:

y := 0;   **if** (x = 1) **then** y := 1

In this case, there is no direct flow from x to y. However, the value of x determines whether the **then** statement will be executed. The flow in both of these examples is allowed only if the security classification of y is at least that of x. For instance, if x were classified high then y must also be classified high in order for the code to be secure [1].

B. A TYPE-BASED TREATMENT OF SECURE INFORMATION FLOW

Goguen and Meseguer introduced a notion of security for deterministic computer systems called noninterference [2]. The basic idea is that a system has users who may supply information with various security classifications to the system. A system satisfies the noninterference property if its low-level outputs remain the same when its high-level inputs are changed.

Volpano and Smith [3] have applied this idea to programming languages. When applied to languages, the idea is that low-level program outputs are unaffected by changes in high-level program inputs.

C. A TYPE INFERENCE ALGORITHM

Volpano and Smith go on to describe an algorithm that is defined by cases on the phrases of a simple imperative language. The evaluation of an expression returns a principal type and a set of typing constraints. A typing constraint is an inequality between two types that are security levels. For example, if τ is type high and τ' is type low then $\tau' \leq \tau$ is a constraint. Note that $\tau' = \tau$ is equivalent to $\tau' \leq \tau$ and $\tau \leq \tau'$. It is important to note also that the algorithm produces constraints among type variables, where a type variable ranges over types like high and low. Constraint-set satisfiability can be used on the set of constraints to determine whether illegal flows exist in the program being analyzed, for instance, if a constraint set contains high $\leq$ low.

The classifications, or types, over which type variables range, depend on the system being modeled. In a typical military system, the types would be unclassified, confidential, secret, and top secret. For the purposes of this discussion, we consider a simple system of only two types, high and low, where low $\leq$ high.

As an example of how the algorithm works, consider the case of the preceding assignment statement, y := x. Assuming x and y have already been assigned the type variables τ_0 and τ_1 respectively, the following set of constraints will be generated by the type inference algorithm:

$$\{\tau_0 \leq \tau_2, \tau_1 = \tau_2, \tau_3 \leq \tau_2\}$$

Therefore, the principal type of the expression is τ_3 *cmd.* The constraint set can be simplified to $\{\tau_0 \leq \tau_1, \tau_3 \leq \tau_1\}$. So, for the assignment statement y := x, the algorithm states that the classification of y must be at least as high as the classification of x. The second constraint allows downward coercion on command types [7].

D. AN IMPLEMENTION OF THE ALGORITHM

This thesis presents a Java program that implements the type inference algorithm. The program is generated from a specification that is input to a compiler compiler called JavaCC. JavaCC is a tool that reads a grammar specification written in a LEX/ YACC-like manner and converts it into a parser for the grammar. The algorithm was incorporated into a grammar specification for Java 1.0.2 supplied with the JavaCC distribution. The actions specified by the algorithm were performed by adding Java code (semantic actions) to the corresponding productions in the grammar specification. The generated parser is a secure flow analyzer for a subset of Java. Several statements, expressions, and other Java functionality were removed from the grammar specification because they are not currently supported by the type inference algorithm

E. THESIS ORGANIZATION

Work in the area of secure information flow and a lattice model of secure information flow are discussed in Chapter II, followed by a description of the secure flow type system in Chapter III. The type-inference algorithm is discussed in Chapter IV. In Chapter V, the static analyzer and the Java subset we consider are discussed. Chapter VI gives an example run of the analyzer, and Chapter VII discusses some possible future work and presents conclusions about secure flow analysis and the static analyzer.

II. THE LATTICE MODEL OF SECURE INFORMATION FLOW

The security mechanisms of most computer systems do not attempt to detect or prevent insecure information flows. Computer system security requires that programs at high security levels be unable to transfer information to low security users or programs. Most access control mechanisms are concerned with direct access control and are not concerned with information flow channels that may exist. Other systems rely on the trustworthiness of processes [5].

In the lattice model of secure flow, a flow policy is represented by the poset $<S, \rightarrow>$ [5]. S is a set of security classes and $\rightarrow$ is a partial order, called the flow relation. The flow relation specifies permissible flows between the security classes. Every variable x is assigned a security class, denoted $\underline{x}$, that is statically bound to x and that can be determined at compile time from declarations given in the program. If x and y are variables in a program and an information flow from x to y exists, then the flow is allowed if $\underline{x} \rightarrow \underline{y}$ [6].

Each programming construct has a certification rule. Some rules, such as assignment statements, certify explicit flows and other rules, such as **if** statements, certify implicit flows. An assignment statement, x := y, will be certified if $\underline{x} \rightarrow \underline{y}$. The rules for conditional constructs such as the following **if** statement certify implicit flows.

if x = 0 **then** y := 0 **else** z := 1

This statement is certified if $\underline{x} \rightarrow \underline{y}$ and $\underline{x} \rightarrow \underline{z}$.

If the poset $<S, \rightarrow>$ is a lattice, then there is a unique least upper bound and greatest lower bound for any pair of classes. A simple grammar consisting of synthesized attributes can be given to certify programs. The attributes are security classes computed using the least upper bound, lub, and greatest lower bound, glb, operations. For example, the certification requirement for the above **if** statement becomes the single condition $\underline{x} \rightarrow \mathrm{glb}(\underline{y}, \underline{z})$ [6].

III. A SECURE FLOW TYPE SYSTEM

Volpano, Irvine, and Smith describe a type system consisting of a set of type inference rules and axioms for deriving typing judgements. The types of the system are divided into three levels. One level contains data types, which we refer to as τ types. These are the security classes of Denning's model and they are partially ordered, for example, low $\leq$ high.

At the next level, are the π types. They consist of the data types τ, command types τ *cmd* and the procedure types

$$\tau\ proc(\tau_1,\ \tau_2\ var,\ \tau_3\ acc)$$

A variable of type τ *var* means it can store information at level τ. A command has type τ *cmd* only if every assignment in the command is made to a variable whose security level is τ or higher. Lastly, the τ in the above procedure type refers to the security level of its body. That is, a call to a procedure of this type would have type τ *cmd.*

At the third and final level are the ρ, or phrase, types. They consist of are the π types, type τ *var* and type τ *acc* (we ignore type τ *acc*). So, our procedure types, in this this, are of the form:

$$\tau\ proc(\tau_1\ var,...,\tau_n\ var)$$

The partial order on τ types is extended to a subtype relation over phrase types. The subtype relation is anti-monotonic in the types of the commands, meaning if τ is a subtype of τ', then τ' *cmd* is a subtype of τ *cmd.* The intuition here is that if one can read level τ' (high) information then they can read level τ (low) information. There is also a typical type subsumption rule that states if a phrase has type ρ then it can be assigned a type ρ' if ρ is a subtype of ρ' [7].

The typing rules of the system guarantee secure explicit and implicit flow. Consider the typing rule for assignment:

$$\frac{\gamma \vdash x : \tau\ \mathit{var} \qquad \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ \mathit{cmd}}$$

where γ is an identifier typing that maps identifiers to ρ types. The rule states that the explicit flow from expression e to variable x is secure if e and x have the same security level. This does not prevent e from having a lower security level than x, because subtyping allows the level to be coerced upward.

The next example shows a rule that deals with a situation where an implicit flow exists. Consider the following program phrase where x is either 0 or 1:

if x = 1 **then** y := 1 **else** y := 0

There is no explicit flow from x to y, but when the phrase is executed, y will contain the value of x. To guarantee the implicit flow from x to y is secure, the following typing rule is used:

$$\frac{\gamma \vdash e : \tau \qquad \gamma \vdash c : \tau\ \mathit{cmd} \qquad \gamma \vdash c' : \tau\ \mathit{cmd}}{\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' : \tau\ \mathit{cmd}}$$

The commands c and c' must have type τ *cmd*, because information of type τ is implicitly known by evaluating the predicate e. Therefore c and c' can only make assignments to variables at security level τ or higher. The rule requires e, c, and c' to have the same security level, namely τ. Nevertheless, an upward implicit flow from e to c and c' can be accommodated by subtyping.

There is also a rule for local variable declarations. A local variable declaration of the form

letvar x := e **in** c

creates a variable x with an initial value e, whose scope is command c. The initialization of x may cause an implicit flow, but it is always harmless.

Two lemmas are needed to prove type soundness: Simple Security and Confinement. Simple Security applies to expressions and Confinement applies to commands. If an expression e can be assigned type τ, then Simple Security states that only variables of type τ or lower will be read when e is evaluated (no read up). Confinement says that if a command c can be assigned type τ *cmd*, then every variable that is updated in c has security level τ or higher (no write down). These two lemmas are used to prove that the type system is sound. Soundness is formulated as a noninterference property. The noninterference property states that variables in a well-typed program do not interfere with variables at lower security levels.

It is possible to automatically check whether a program is well typed, using the techniques of *type inference*. The basic idea of type inference is to use type variables to represent unknown types in a program, and to generate constraints in the form of inequalities. An assignment of types to these variables must satisfy the constraints in order for the program to be well typed with respect to that assignment. A principal type can be formulated that represents all possible types the program can be given.

IV. A SECURE FLOW TYPE INFERENCE ALGORITHM

A type inference algorithm that ensures secure information flow is described in this chapter. Volpano and Smith have extended the type system discussed in the previous chapter to a simple language with first order procedures [3]. They also prove the noninterference property for the system in order to establish the type soundness in the context of procedures. Figure 1 shows the core language they considered.

expressions ::= $x \mid n \mid l \mid$
$e_1 + e_2 \mid$
proc(in x_1, inout x_2, out x_3) c

commands ::= $c_1; c_2 \mid$
if e then c_1 else $c_2 \mid$
while e do $c \mid$
$e_1 := e_2 \mid$
letvar $x := e$ in $c \mid$
letproc x (in x_1, inout x_2, out x_3) c in $c' \mid$
$e(e_1,e_2,e_3)$

Figure 1. Core Language

For expressions, meta-variable x ranges over identifiers, n ranges over integer literals, and l ranges over locations. Expressions also consist of anonymous procedure expressions. Their names are provided via letproc.

Commands consist of the following: composition of commands, **if**, **while** loops, assignment, variable declarations, procedure declarations, and procedure calls.

Volpano and Smith give a secure flow type inference algorithm in [3]. It is shown in Figure 2 and is defined by cases on the phrases of the core language. The algorithm takes as inputs a location typing λ, an identifier typing γ, a program phrase p, and a set of

```
$W(\lambda, \widehat{\gamma}, p, V) = \textbf{case } p \textbf{ of}$

$x : \textbf{case } \widehat{\gamma}(x) \textbf{ of}$
        $\widehat{\tau} : (\{\widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$
        $\widehat{\tau}\ var : (\{\widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$
        $\textbf{default : fail}$

$n : (\{\ \}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$l : (\{\lambda(l) \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$e_1 + e_2 :$
        $\textbf{let } (C_1, \widehat{\tau_1}, V') = W(\lambda, \widehat{\gamma}, e_1, V)$
        $\textbf{let } (C_2, \widehat{\tau_2}, V'') = W(\lambda, \widehat{\gamma}, e_2, V')$
        $\textbf{in } (C_1 \cup C_2 \cup \{\widehat{\tau_1} = \widehat{\tau_2}\}, \widehat{\tau_1}, V'')$

$\textbf{proc (in } x_1, \textbf{ inout } x_2, \textbf{ out } x_3\textbf{) } c :$
        $\textbf{let } (C, \widehat{\tau}\ cmd, V') = W(\lambda, \widehat{\gamma}[x_1 : \alpha, x_2 : \beta\ var, x_3 : \delta\ acc], c, V \cup \{\alpha, \beta, \delta\})$
        $\textbf{in } (C, \widehat{\tau}\ proc(\alpha, \beta\ var, \delta\ acc), V') \quad \alpha, \beta \textbf{ and } \delta \notin V$

$c_1;\ c_2 : \textbf{let } (C_1, \widehat{\tau_1}\ cmd, V') = W(\lambda, \widehat{\gamma}, c_1, V)$
        $\textbf{let } (C_2, \widehat{\tau_2}\ cmd, V'') = W(\lambda, \widehat{\gamma}, c_2, V')$
        $\textbf{in } (C_1 \cup C_2 \cup \{\widehat{\tau_1} = \widehat{\tau_2}\}, \widehat{\tau_1}\ cmd, V'')$

$\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 :$
        $\textbf{let } (C, \widehat{\tau}, V') = W(\lambda, \widehat{\gamma}, e, V)$
        $\textbf{let } (C_1, \widehat{\tau_1}\ cmd, V'') = W(\lambda, \widehat{\gamma}, c_1, V')$
        $\textbf{let } (C_2, \widehat{\tau_2}\ cmd, V''') = W(\lambda, \widehat{\gamma}, c_2, V'')$
        $\textbf{in } (C \cup C_1 \cup C_2 \cup \{\widehat{\tau} = \widehat{\tau_1} = \widehat{\tau_2}, \alpha \leq \widehat{\tau}\}, \alpha\ cmd, V''' \cup \{\alpha\}) \quad \alpha \notin V'''$

$\textbf{while } e \textbf{ do } c :$
        $\textbf{let } (C, \widehat{\tau}, V') = W(\lambda, \widehat{\gamma}, e, V)$
        $\textbf{let } (C', \widehat{\tau}'\ cmd, V'') = W(\lambda, \widehat{\gamma}, c, V')$
        $\textbf{in } (C \cup C' \cup \{\widehat{\tau} = \widehat{\tau}', \alpha \leq \widehat{\tau}\}, \alpha\ cmd, V'' \cup \{\alpha\}) \quad \alpha \notin V''$

$e_1 := e_2 :$
        $\textbf{let } (C, \widehat{\tau}', V') = W(\lambda, \widehat{\gamma}, e_2, V)$
        $\textbf{case } e_1 \textbf{ of}$
            $x : \textbf{ if } \widehat{\gamma}(x) = \widehat{\tau}\ var \textbf{ or } \widehat{\gamma}(x) = \widehat{\tau}\ acc \textbf{ then}$
                    $(C \cup \{\widehat{\tau} = \widehat{\tau}', \alpha \leq \widehat{\tau}'\}, \alpha\ cmd, V' \cup \{\alpha\}) \quad \alpha \notin V'$
                $\textbf{else fail}$
            $l : \ (C \cup \{\lambda(l) = \widehat{\tau}', \alpha \leq \widehat{\tau}'\}, \alpha\ cmd, V' \cup \{\alpha\}) \quad \alpha \notin V'$
            $\textbf{default : fail}$

$\textbf{letvar } x := e \textbf{ in } c :$
        $\textbf{let } (C, \widehat{\tau}, V') = W(\lambda, \widehat{\gamma}, e, V)$
        $\textbf{let } (C', \widehat{\tau}'\ cmd, V'') = W(\lambda, \widehat{\gamma}[x : \widehat{\tau}\ var], c, V')$
        $\textbf{in } (C \cup C', \widehat{\tau}'\ cmd, V'')$

$\textbf{letproc } x(\textbf{in } x_1, \textbf{ inout } x_2, \textbf{ out } x_3) \ c \textbf{ in } c' :$
        $\textbf{let } (C, \widehat{\tau}, V') = W(\lambda, \widehat{\gamma}, \textbf{proc (in } x_1, \textbf{ inout } x_2, \textbf{ out } x_3\textbf{) } c, V)$
        $\textbf{let } (C', \widehat{\tau}\ cmd, V'') = W(\lambda, \widehat{\gamma}, [\textbf{proc (in } x_1, \textbf{ inout } x_2, \textbf{ out } x_3\textbf{) } c/x]c', V')$
        $\textbf{in } (C \cup C', \widehat{\tau}\ cmd, V'')$
$e(e_1, e_2, e_3) :$
        $\textbf{let } (C, \widehat{\tau}\ proc(\widehat{\tau_1}, \widehat{\tau_2}\ var, \widehat{\tau_3}\ acc), V') = W(\lambda, \widehat{\gamma}, e, V)$
        $\textbf{let } (C', \widehat{\tau}', V'') = W(\lambda, \widehat{\gamma}, e_1, V')$
        $\textbf{let } C'' = \textbf{case } e_2 \textbf{ of}$
            $x : \textbf{ if } \widehat{\gamma}(x) = \widehat{\tau}''\ var \textbf{ then } C \cup C' \cup \{\widehat{\tau}' = \widehat{\tau_1}, \widehat{\tau}'' = \widehat{\tau_2}\} \textbf{ else fail}$
            $l : \ C \cup C' \cup \{\widehat{\tau}' = \widehat{\tau_1}, \lambda(l) = \widehat{\tau_2}\}$
            $\textbf{default : fail}$
        $\textbf{in case } e_3 \textbf{ of}$
            $x : \textbf{ if } \widehat{\gamma}(x) = \widehat{\tau}''\ var \textbf{ or } \widehat{\gamma}(x) = \widehat{\tau}''\ acc \textbf{ then } (C'' \cup \{\widehat{\tau}'' = \widehat{\tau_3}\}, \widehat{\tau}\ cmd, V'')$
                $\textbf{else fail}$
            $l : (C'' \cup \{\lambda(l) = \widehat{\tau_3}\}, \widehat{\tau}\ cmd, V'')$
            $\textbf{default : fail}$
```

Figure 2. Volpano-Smith Type Inference Algorithm

type variables V. A location typing maps addresses to τ types and an identifier typing maps variables to types τ and τ *var*, for some τ. The latter treats free variables in a program, while the former treats free addresses. We shall assume programs have no free addresses, and drop λ from the implementation of the type inference algorithm. The set V contains a list of previously-used type variables and allows the algorithm to choose new type variables. If the algorithm succeeds, it returns a triple consisting of a set of constraints C, a type π, and the updated set of stale variables V. The constraints in C are inequalities among type variables.

To illustrate how the algorithm works, we give an example from [3], shown in Figure 3, of a procedure that indirectly copies a variable x to another variable y.

```
proc (in x, out y)
   letvar a := x in
   letvar b := 0 in
      while a > 0 do
         b := b + 1;
         a := a - 1;
      y := b
```

Figure 3. Example Program

Figure 4 shows the results of calling the algorithm on the procedure. The algorithm yields a triple consisting of a set of stale type variables V, the list of generated constraints and the type of the procedure, here denoted by π. This triple is used to form the principal type for the procedure.

Type simplification can be used to simplify the constraint set C and type π [8]. The static analyzer developed for this thesis does not include any mechanism to perform type simplification and such simplification is shown here for demonstration purposes

$$\begin{array}{ll} V = & \{\alpha, \gamma, \nu, o, \varepsilon, \iota, \zeta, \upsilon, \delta, \eta, \theta, \kappa, \lambda, \beta, \xi\} \\ C = & \{\alpha \leq \gamma,\ \nu = o,\ \varepsilon = \iota,\ \nu \leq \varepsilon,\ \varepsilon = \zeta,\ \gamma \leq \varepsilon,\ \iota = \upsilon,\ \delta = \eta,\ \iota \leq \delta, \\ & \eta = \theta,\ \delta \leq \eta, \gamma = \kappa,\ \upsilon \leq \gamma,\ \kappa = \lambda,\ \gamma \leq \kappa,\ \beta = \xi,\ o \leq \beta,\ \delta \leq \xi\} \\ \pi = & (\nu \text{ proc}(\alpha, \beta\, acc)) \end{array}$$

Figure 4. Algorithm Results of Sample Program

only. The first step collapses the strongly connected types and produces a more useful form, as shown in Figure 5.

$$\begin{array}{ll} V = & \{\alpha, o, \delta, \xi\} \\ C = & \{\delta \leq \xi, o \leq \lambda,\ \lambda \leq \delta,\ \alpha \leq \lambda\} \\ \pi = & (o \text{ proc}(\alpha, \xi\, acc)) \end{array}$$

Figure 5. Algorithm Results after Type Simplification

Further simplification is possible leading to the π in Figure 6.

$$\pi = (\xi \text{ proc}(\xi, \xi\, acc))$$

Figure 6. Principal Type after Applying Monotonicity-Based Instantiations

V. IMPLEMENTATION OF THE TYPE INFERENCE ALGORITHM

The static analyzer that performs the security checks specified by the type inference algorithm was developed using the Java Compiler Compiler (JavaCC). JavaCC takes, as input, a grammar specification. The output is a Java program that will parse the specified language and perform the semantic actions indicated in the grammar specification.

Rather than start from scratch and build a JavaCC specification for the language in Figure 1, we started with a grammar specification for Java 1.0.2, which we modified to reflect the language in Figure 1. Semantic actions were added to encode the type inference algorithm. The specification is given in Appendix A. There are several restrictions imposed on the kinds of Java programs that the static analyzer can check because there are many constructs in the Java language that are not currently treated in the type inference algorithm. Each of the phrases in Figure 1 was mapped to a corresponding expression or statement in the Java grammar specification.

A. A BRIEF LOOK AT JAVACC

JavaCC constructs a Java program that acts as a recursive descent parser for the language described by the grammar specification. A sample from the Java 1.0.2 grammar specification is shown in Figure 7. The sample shows three productions that are used to parse a Java method declaration and parameters. JavaCC converts each production into a method in the generated parser.

```
void MethodDeclarator() :
{}
{
  <IDENTIFIER> FormalParameters() ( "[" "]" )*
}

void FormalParameters() :
{}
{
  "(" [ FormalParameter() ( "," FormalParameter() )* ] ")"
}

void FormalParameter() :
{}
{
  Type() VariableDeclaratorId()
}
```

Figure 7. Sample Productions

Each production begins with the return type of the corresponding method in the parser, which is `void` for the three productions in Figure 7. The name of the production will also be the name of the method in the parser. Parameter passing can be adding to the productions in the same way it is used in Java programs.

There is a notion of "calling" a production because of its relationship with the corresponding method in the generated parser. For example, if the production `FormalParameter()` in Figure 7 is called, it will in turn call the productions `Type()` and `VariableDeclaratorId()`.

Java code can be added anywhere in the production, but must be enclosed in curly braces, "{ }". When JavaCC converts the production into its corresponding method, the added code will remain where it was placed. Local variable declarations for any production should be inserted in the first set of curly braces of that production. In the three productions shown in Figure 7, there are no local variable declarations.

B. IMPLEMENTING THE ALGORITHM USING JAVACC

There are two main data structures in the implementation of the algorithm. The first is called `gamma`, and contains identifier typings. The second is called `triple`, and consists of the items returned by the type inference algorithm, namely, a set of constraints C, a type π, and a list of stale type variables V.

The initial attempt to implement the algorithm used two Stacks from the Java utility package. The gamma stack held objects called gamma items. A gamma item consisted of a variable name and its type variable. The triple stack contained the triple items consisting of the constraint set in the form of a linked list and the principal type. The set of stale type variables was kept in a separate symbol generator for the entire program.

The idea of the gamma stack was to push a gamma item whenever a new variable was encountered and to pop the stack when the variable's scope ended. It became apparent that determining when the variable's scope ended was going to be a difficult task unless the analyzer kept track of more information about the variables being declared. The analyzer soon had four separate stacks to keep track of the important information. The triple stack had similar problems.

It was determined that all of the external stacks could be eliminated if the run time stack was utilized. In this implementation, gamma became a linked list of gamma items that is passed as a parameter from one production to those productions it calls. In addition, each production returns a triple that contains all the constraints generated in the program. This did pose one problem. A local variable declaration requires an update to the gamma list with the variable's type information and it also requires the generation of a new triple item. Both must be returned to the calling production.

This problem may be overcome by adding new productions to the specification but the productions were not added in this implementation. Instead, a new data structure was developed to simply hold the new gamma list and the generated triple so that both the gamma list and the generated triple could be returned. The structure, called Dual, was later updated to also hold a string when a similar situation arose in the method declaration production that required a gamma list and the string representation of a token to be returned.

Each of the commands and expressions of the core language listed in Figure 1 are "mapped" to one or more productions in the Java 1.0.2 grammar specification. Mapping the algorithm to the Java specification was performed in two steps. The first step was to determine which productions in the grammar specification correspond to commands or expressions in the core language. Once the relationship between the core language and grammar specification was established, the second step entailed encoding the semantic actions specified by the algorithm and placing the code in the corresponding productions of the grammar specification. We consider, in turn, each of the cases of the algorithm in Figure 2.

Case "x"

The `Name()` production in the grammar file is an instance of case x. `Name()` returns a string representation of the current token when the production is called. The type inference algorithm requires the type of x, τ or τ *var*, to be determined. The type resolution of the token that corresponds to x is performed within the production that calls `Name()`.

Case "n"

The `Literal()` production is an instance of case n. `Literal()` accepts the Java primitive types of integers, floating point numbers, characters, strings, boolean values "true" and "false", and "null".

Case "l"

The third case statement, *l*, deals with locations and is not implemented in the Java grammar.

Case "$e_1 + e_2$"

The expressions below are all instances of case $e_1 + e_2$:

```
ConditionalOrExpression()
ConditionalAndExpression()
InclusiveOrExpression()
ExclusiveOrExpression()
AndExpression()
EqualityExpression()
RelationExpression()
ShiftExpression()
AdditiveExpression()
MultiplicativeExpression()
```

Case "proc(in x_1, inout x_2, out x_3) c"

The case in the algorithm for procedure declarations has the following form:

proc(in x_1, **inout** x_2, **out** x_3) c

The modes of the parameters, **in**; **inout**; and **out**, are similar to those used in the Ada programming language. The productions dealing with procedures starts with the `MethodDeclaration()` production. The name of the procedure and the parameters are treated in a call to the `MethodDeclarator()` production. The parameters are added to the environment with a call to the `FormalParameters()` production so they may be referenced in the body of the procedure. `MethodDeclarator()` returns the procedure name and the types of the parameters. All parameters are considered to be **inout** mode and are typed as such, meaning they have type τ *var* for some τ. Finally the body of the procedure, c, is handled in a call to the `Block()` production. The static analyzer does not handle recursive procedures or method declarations.

Case "c_1; c_2"

Next in the algorithm is the statement for composition, c_1; c_2. Composition within a block, delimited by `{}`, is handled by the `BlockStatementList()` production. The original Java grammar specification handled composition in the `Block()` production. It was necessary to add the production `BlockStatementList()` to handle the **letvar** statement. Changes to the grammar specification for the **letvar** statement are explained later in this section.

Case "if e then c_1 else c_2"

If-then-else statements are handled by the `IfStatement()` production in the grammar specification. The **else** portion of the statement is not mandatory in Java. If it is not used, then the semantic actions in the algorithm pertaining to the else statement are not executed.

Case "while e do c"

The next case is the **while** loop of the form, **while** e **do** c. It has been mapped to both the `WhileStatement()` and `DoStatement()` productions in the Java specification.

Case "x := e"

The assignment statement x := e is mapped to `Assignment()`. Note that "`:=`" is not the only assignment operator allowed; others include: "`*=`", "`/=`", "`+=`", and "`-=`". A modification to the grammar specification was required here. The Java 1.0.2 grammar specification `Assignment()` production is listed in Figure 8. The production, `PrimaryExpression()`, may be evaluated as a `literal()`, `Name()`, `Expression()`, or `AllocationExpression()`. `PrimaryExpression()` is also called from a number of other productions as well and those productions require that `PrimaryExpression()` return a triple consisting of a constraint set, a type, and a list of stale type variables. However, the `Assignment()` production requires that `PrimaryExpression()` return the type of x from the identifier typing γ. For this reason, a new production, `PrimaryLeftExpression()`, was introduced into the Grammar specification. It returns the string representation of x, so that it may be referenced in γ, and replaces `PrimaryExpression()` in the `Assignment()` production.

```
void Assignment() :
{}
{
  PrimaryExpression()AssignmentOperator()Expression()
}
```

Figure 8. Assignment Production

Case "letvar x := e in c"

Mapping the **letvar** statement to the Java language required another modification to the Java grammar specification. The original specification handled local variable declarations at the same level as all other statements within `BlockStatement()`. The original Java specification productions that handle local variable declarations are shown in Figure 9.

```
void Block() :
{}
{
  "{" ( BlockStatement() )* "}"
}

void BlockStatement() :
{}
{
  LOOKAHEAD(Type() <IDENTIFIER>)
  LocalVariableDeclaration() ";"
|
  Statement()
}

void LocalVariableDeclaration() :
{}
{
  Type() VariableDeclarator() ( "," VariableDeclarator() )*
}
```

Figure 9. Java Specification Productions to Handle Local Variable Declarations

In the original grammar specification, composition is handled in the `Block()` production. The * operator indicates that the production(s) within the preceding set of parentheses is called zero or more times. Two new productions, `BlockStatementList()` and `LetvarStatement()`, were added to the grammar specification because it is necessary to pass the identifier typing γ, updated with a typing for **x**, to the production that parses **c** in **letvar x** = **e** in **c**. The original Java 1.0.2 grammar specification had no productions specified for c, so `BlockStatementList()` was introduced to handle this problem. In the modified grammar specification, `Block()` calls `BlockStatementList()` once per `BlockStatement()`. `BlockStatementList()`, the production used to handle composition, calls `BlockStatement()` zero or more times. `BlockStatement()` calls `LetvarStatement()` if a local variable declaration is found, otherwise, `Statement()` is called. `LetvarStatement()` first calls `LocalVariableDeclaration()` to handle the declaration, then `BlockStatementList()` to parse the rest of the program that is within the scope the new variable. The section of the modified grammar file is listed in Figure 10.

Case "letproc x(in x_1, inout x_2, out x_3)c in c' "

The next case in the type inference algorithm, **letproc,** allows procedures to be used polymorphically and was not implemented in the Java grammar specification. Therefore, all procedures are treated as monomorphic in the analyzer specification. Moreover, only static methods are allowed because that is the only kind of method the algorithm treats.

Case "e(e_1, e_2, e_3)"

The final case in the algorithm types procedure calls. The Java specification handles procedure calls in the `PrimaryPrefix()` production. First, the name of the procedure is found in the identifier typing , γ, then the types of the arguments are compared with those retrieved from γ. The original grammar specification for Java allowed arguments to be expressions. In the modified specification, all parameters must be either a literal or a previously declared and initialized variable name.

```
void Block() :
{}
{
  "{" BlockStatementList() "}"
}

void BlockStatementList() :
{}
{
  ( LOOKAHEAD(2) BlockStatement() )*
}

void BlockStatement() :
{}
{
  LOOKAHEAD( Type() <IDENTIFIER> )
  LetvarStatement()
|
  Statement()
}

void LetvarStatement () :
{}
{
  LocalVariableDeclaration() ";" BlockStatementList()
}

void LocalVariableDeclaration() :
{}
{
  Type() VariableDeclarator() ( "," VariableDeclarator() )*
}
```

Figure 10. Specification Changes for letvar Statement

All of the source code files used to implement the static analyzer are given in Appendix B.

C. RESTRICTIONS IMPOSED ON PROGRAMS

The type inference algorithm in [3] does not treat an object-oriented language like Java. Although we started with a JavaCC specification for Java, the result was not an analyzer for full Java but rather an analyzer for that subset of Java corresponding to the simple language in Figure 1. So how big is this subset?

First, the subset that can be analyzed has no objects, and consequently no instance variables or instance methods.

Second, all expressions must be free of any side effects. This is the reason that assignment expressions in Java are prohibited, as are pre and post increment "expressions". They all violate the confinement property.

Other restrictions on Java programs include that they be closed (no free variables), that they have only non-recursive static methods, that they have no methods with a return type other than `void`, and that they have no forward references. Yet, other restrictions are imposed because certain constructs were not treated in the algorithm of [3]. They include try-catch blocks, synchronized blocks and so on. In summary, the following features of Java are not analyzed:

1. Static Initializes
2. Arrays
3. Explicit Constructor Invocation
4. Conditional Expressions
5. Instanceof Expressions
6. PreIncrement and PreDecrement Expressions
7. PostIncrement and PostDecrement Expressions
8. Cast Expressions
9. Allocation Expressions - (object creation)
10. Labeled Statements
11. Switch Statements
12. For Statements
13. Break Statements
14. Continue Statements
15. Return Statements
16. Throw Statements
17. Synchronized Statements
18. Try Statements
19. Catch Statements
20. Finally Statements

The constructs that have been disallowed have only been commented out in the grammar specification file listed in Appendix A in order to allow for their implementation in the future. This means they cannot be parsed in the current implementation.

VI. AN EXAMPLE RUN OF THE STATIC ANALYZER

The program in Figure 11 illustrates an application of the static analyzer. It corresponds to the example program of Figure 3, in Chapter III, written in Java. However, it is not identical, for Java has no parameter-passing mode corresponding to mode **out**. Nevertheless, it serves to illustrate the analyzer. The results of the static analyzer when run on this program are shown in Figure 12.

```
class test
{
  public static void p(int x, int y)
  {
    int a = x;
    int b = 0;
    while (a > 0){
      b = b + 1;
      a = a - 1;
    }
    y = b;
  }
}
```

Figure 11. Static Analyzer Test Program

V = $\{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7, \tau_8, \tau_9, \tau_{10}, \tau_{11}, \tau_{12}, \tau_{13}, \tau_{14}\}$

C = $\{\tau_{14} = \tau_{12}, \tau_{12} \leq \tau_4, \tau_8 = t_4, \tau_5 = \tau_4, \tau_2 \leq \tau_4, \tau_{11} = \tau_8, \tau_8 \leq \tau_6, \tau_6 = \tau_3, \tau_7 = \tau_6,$
$\tau_3 \leq \tau_6, \tau_{11} \leq \tau_9, \tau_9 = \tau_2, \tau_{10} = \tau_9, \tau_2 \leq \tau_9, \tau_{14} \leq \tau_{13}, \tau_1 = \tau_{13}, \tau_3 \leq \tau_{13}, \tau_0 \leq \tau_2\}$

π = τ_{12} proc($\tau_0$var, $\tau_1$var)

Figure 12. Test Program Results

We sketch a trace of the analyzer on part of the program. The parameters, `x` and `y`, are the first tokens to be analyzed. They are assigned the type variables τ_0 and τ_1

respectively. Then the variable declaration:

```
int a = x
```

is analyzed. A new type variable for x, namely τ_2, is created and the constraint set $\{\tau_0 \leq \tau_2\}$ is generated. The constraint is generated by the case for identifiers where an upward coercion is introduced (see Figure 2). The variable a is assigned the type variable τ_2 in analyzing the rest of the program.

Next, the variable declaration:

```
int b = 0
```

is analyzed in the same manner, except that no constraint is generated since 0 is an integer. This is the integer literal case of the type inference algorithm. Finally, b is assigned the type variable τ_3. At this point, gamma contains the following types:

$$\{x : \tau_0 , y : \tau_1, a : \tau_2, b : \tau_3\}$$

and only one constraint, $\tau_0 \leq \tau_2$, exists.

Next, the while loop

```
while(a > 0)
```

is analyzed. The predicate, a > 0, is checked first and generates the following new constraints:

$$\tau_2 \leq \tau_4, \tau_4 = \tau_5$$

The first comes from the identifier case of the algorithm (upward coercion of a's type) and the second comes from $\hat{\tau}_1 = \hat{\tau}_2$ in the case for $e_1 + e_2$ in the algorithm of Figure 2, where $\hat{\tau}_1 = \tau_4$ and $\hat{\tau}_2 = \tau_5$ The rest of the program is analyzed in the same manner.

VII. CONCLUSIONS

As we rely more on computer systems, secure flow analysis is a necessary tool to protect the information stored on these systems. Denning's work [1] [5] provides a good base of knowledge for secure information flow. The Lattice Model consists of a set of storage objects, a set of processes, and a set of security classes. Each storage object is bound statically or dynamically to a security class. Security classes are required to form a lattice, hence the name. A flow relation indicates permitted information flows between security classes. The lattice shows all allowed information flows within the system.

Volpano and Smith [3] treat the model in the context of a type system and prove the soundness of the type system. They also give a type inference algorithm for the system. This thesis describes an implementation of that algorithm using JavaCC. The result is a static analyzer that checks for secure information flow at compile-time.

The static analyzer can only analyze a subset of the Java 1.0.2 language. It may be too limited to allow one to write interesting and useful programs. Future work might focus on analyzing a larger subset of Java.

LIST OF REFERENCES

1. Denning, D. E. and Denning, P. E., "Certification of Programs for Secure Information Flow," *Communications of the ACM*, vol. 20 no. 7, pp. 504-513, July 1977.

2. Goguen, J. and Meseguer, J., "Security Policies and Security Models," *Proceedings 192 IEEE Symposium on Security and Privacy*, pp. 11-20, 1982.

3. Volpano D, and Smith, G., "A Type-Based Approach to Program Security," *Proceedings of TAPSOFT '97, Colloquium on Formal Approaches in Software Engineering*, 14-18 April, 1997.

4. "Java Compiler Compiler," Sun Microsystems, http://www.suntest.com/JavaCC/index.html

5. Denning, D. E., "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol. 19 no. 5, pp. 236-243, May 1976.

6. Volpano, D., Irvine, C., "Secure Flow Typing," *Computers and Security*, vol. 16 no.2, pp. 137-144,1997.

7. Volpano, D., Irvine, C., and Smith, G., "A Sound Type System for Secure Flow Analysis," *Journal of Computer Security*, vol. 4, pp. 167-187, 1996.

8. Smith, G., "Polymorphic Type Inference for Languages with Subtyping and Overloading," Cornell University, Technical Report, 91-1230, 1991.

APPENDIX A - JAVA GRAMMAR SPECIFICATION

The following pages represent the modified Java 1.0.2 grammar specification that is the input to the Java Compiler Compiler. The original grammar file was developed by Sriram Sankar on 6/11/96 and is copyrighted by Sun Microsystems Inc. Semantic actions were added to the original grammar to perform secure flow analysis on a subset of Java 1.0.2 programs.

```
/**
 *
 * Copyright (C) 1996, 1997 Sun Microsystems Inc.#
 *
 * Use of this file and the system it is part of is constrained by the
 * file COPYRIGHT in the root directory of this system. You may,
 * however, make any modifications you wish to this file.
 *
 * Java files generated by running JavaCC on this file (or modified
 * versions of this file) may be used in exactly the same manner as
 * Java files generated from any grammar developed by you.
 *
 * Author: Sriram Sankar
 * Date: 6/11/96
 *
 * This file contains a Java grammar and actions that implement a
 * front-end.
 *
 * Modified 24 Feb 98 by LT James D. Harvey, USN.
 *
 * Modifications have been made to incorporate a type checker into the
 * compiler. Several portions of the Java language have been disabled
 * in this version because the type checker does not support them. The
 * portions that are not implemented are as follows:
 *
 *    Static Initializers
 *    Arrays
 *    Explicit Constructor Invocation
 *    Conditional Expressions
 *    Instanceof Expressions
 *    PreIncrement and PreDecrement expressions
 *    Cast Expressions
 *    Allocation Expressions
 *    Labeled Statements
 *    Switch Statements
 *    For Statements
 *    Break Statements
 *    Continue Statements
 *    Return Statements
 *    Throw Statement
 *    Synchronized Statement
 *    Try Statement
 *
 *
 *
 *
 */
```

Permission to reproduce has been obtained from Sriram Sankar of Sun Microsystems.

```
options {
  LOOKAHEAD = 1;
  JAVA_UNICODE_ESCAPE = true;
}

PARSER_BEGIN(JavaParser)

import thesis.*;

public class JavaParser {

  static SymbolGenerator sg = new SymbolGenerator();

  public static void main(String args[]) {

    JavaParser parser;
    Triple ConstraintSet;
    Gamma gamma = new Gamma("myGamma");

    if (args.length == 0) {
      System.out.println("Java Parser Version 1.0.2:  Reading from
                                    standard input . . .");
      parser = new JavaParser(System.in);
    } else if (args.length == 1) {
      System.out.println("Java Parser Version 1.0.2:  Reading from file
                                    " + args[0] + " . . .");
      try {
        parser = new JavaParser(new java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
        System.out.println("Java Parser Version 1.0.2:  File " +
                                    args[0] + " not found.");
        return;
      }
    } else {
      System.out.println("Java Parser Version 1.0.2:  Usage is one
                                    of:");
      System.out.println("         java JavaParser < inputfile");
      System.out.println("OR");
      System.out.println("         java JavaParser inputfile");
      return;
    }
    try {
      ConstraintSet = parser.CompilationUnit(gamma);
      System.out.println("Java Parser Version 1.0.2:  Java program
                                    parsed successfully.");
    } catch (ParseError e) {
      System.out.println("Java Parser Version 1.0.2:  Encountered
                                    errors during parse.");
    }
  }
}

PARSER_END(JavaParser)
```

```
SKIP : /* WHITE SPACE */
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])*
"*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])*
"*"))* "/">
}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
| < BREAK: "break" >
| < BYTE: "byte" >
| < CASE: "case" >
| < CATCH: "catch" >
| < CHAR: "char" >
| < CLASS: "class" >
| < CONST: "const" >
| < CONTINUE: "continue" >
| < _DEFAULT: "default" >
| < DO: "do" >
| < DOUBLE: "double" >
| < ELSE: "else" >
| < EXTENDS: "extends" >
| < FALSE: "false" >
| < FINAL: "final" >
| < FINALLY: "finally" >
| < FLOAT: "float" >
| < FOR: "for" >
| < GOTO: "goto" >
| < IF: "if" >
| < IMPLEMENTS: "implements" >
| < IMPORT: "import" >
| < INSTANCEOF: "instanceof" >
| < INT: "int" >
| < INTERFACE: "interface" >
| < LONG: "long" >
| < NATIVE: "native" >
| < NEW: "new" >
| < NULL: "null" >
| < PACKAGE: "package">
| < PRIVATE: "private" >
| < PROTECTED: "protected" >
| < PUBLIC: "public" >
| < RETURN: "return" >
| < SHORT: "short" >
| < STATIC: "static" >
| < SUPER: "super" >
| < SWITCH: "switch" >
| < SYNCHRONIZED: "synchronized" >
| < THIS: "this" >
| < THROW: "throw" >
| < THROWS: "throws" >
| < TRANSIENT: "transient" >
| < TRUE: "true" >
| < TRY: "try" >
| < VOID: "void" >
| < VOLATILE: "volatile" >
| < WHILE: "while" >
}

TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
(["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
|
  < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
|
< STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            ( ["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

TOKEN : /* IDENTIFIERS */
{
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}
```

```
TOKEN : /* SEPARATORS */
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
}

TOKEN : /* OPERATORS */
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < BANG: "!" >
| < TILDE: "~" >
| < HOOK: "?" >
| < COLON: ":" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < SC_OR: "||" >
| < SC_AND: "&&" >
| < INCR: "++" >
| < DECR: "--" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < BIT_AND: "&" >
| < BIT_OR: "|" >
| < XOR: "^" >
| < REM: "%" >
| < LSHIFT: "<<" >
| < RSIGNEDSHIFT: ">>" >
| < RUNSIGNEDSHIFT: ">>>" >
| < PLUSASSIGN: "+=" >
| < MINUSASSIGN: "-=" >
| < STARASSIGN: "*=" >
| < SLASHASSIGN: "/=" >
| < ANDASSIGN: "&=" >
| < ORASSIGN: "|=" >
| < XORASSIGN: "^=" >
| < REMASSIGN: "%=" >
| < LSHIFTASSIGN: "<<=" >
| < RSIGNEDSHIFTASSIGN: ">>=" >
| < RUNSIGNEDSHIFTASSIGN: ">>>=" >
}
```

```
/*****************************************
 * THE JAVA LANGUAGE GRAMMAR STARTS HERE *
 *****************************************/

/*
 * Program structuring syntax follows.
 */

Triple CompilationUnit(Gamma gamma) :
{Triple cs = null;}
{
//[ PackageDeclaration() ]
//( ImportDeclaration() )*
  ( cs = TypeDeclaration(gamma) )*
  <EOF>
  {return cs;}
}

void PackageDeclaration() :
{}
{
  "package" Name() ";"
}

void ImportDeclaration() :
{}
{
  "import" Name() [ "." "*" ] ";"
}

Triple TypeDeclaration(Gamma gamma) :
{Triple cs = null;}
{
 (LOOKAHEAD( ( "abstract" | "final" | "public" )* "class" )
  cs = ClassDeclaration(gamma)
|
  InterfaceDeclaration(gamma)
|
  ";")
{return cs;}
}


/*
 * Declaration syntax follows.
 */
```

```
Triple ClassDeclaration(Gamma gamma) :
{
  Triple cs = null;
  Dual d = new Dual(cs,gamma);
}
{
  ( "abstract" | "final" | "public" )*
  "class" <IDENTIFIER> [ "extends" Name() ] [ "implements" NameList() ]
  "{" ( d = ClassBodyDeclaration(d.gamma) )* "}"
  {
    if (d != null){
    } return d.cs;
    else{
      return cs;
    }//end if
  }
}

Dual ClassBodyDeclaration(Gamma gamma) :
{
  Triple cs = null;
  Dual d = null;
}
{
  (
/*
  LOOKAHEAD(2)
  StaticInitializer()
|
*/
  LOOKAHEAD( [ "public" | "protected" | "private" ] Name() "(" )
  cs = ConstructorDeclaration(gamma)
  {d = new Dual(cs,gamma);}
|
  LOOKAHEAD( MethodDeclarationLookahead() )
  d = MethodDeclaration(gamma)
|
  d = FieldDeclaration(gamma) )
  {
    System.out.println("Constraint set: " + d.cs);
    System.out.println("Gamma: " + d.gamma);
    return d;
  }
}

// This production is to determine lookahead only.
void MethodDeclarationLookahead() :
{}
{
  ( "public" | "protected" | "private" | "static" | "abstract" |
"final" | "native" | "synchronized" )*
  ResultType() <IDENTIFIER> "("
}
```

```
void InterfaceDeclaration(Gamma gamma) :
{Triple cs = null;}
{
  ( "abstract" | "public" )*
  "interface" <IDENTIFIER> [ "extends" NameList() ]
  "{" ( InterfaceMemberDeclaration(gamma) )* "}"
}

void InterfaceMemberDeclaration(Gamma gamma) :
{}
{
  LOOKAHEAD( MethodDeclarationLookahead() )
  MethodDeclaration(gamma)
|
  FieldDeclaration(gamma)
}

Dual FieldDeclaration(Gamma gamma) :
{
  Dual d = null;
}
{
  ( "public" | "protected" | "private" | "static" | "final" |
"transient" | "volatile" )*
  Type()  d = VariableDeclarator(gamma) ";"
  {
    return d;
  }
}

Dual VariableDeclarator(Gamma gamma) :
{
  Triple cs = new Triple(sg.NextSymbol(),"");
  String id;
}
{
  id = VariableDeclaratorId() ( "=" cs = VariableInitializer(gamma) |
cs = Default() )
  {
    gamma = gamma.Append(new GammaItem(id,cs.getType(),"var"));
    Dual d = new Dual(cs, gamma);
    return d;
  }
}

Triple Default() :
{}
{
  {return new Triple(sg.NextSymbol(), "");}
}
```

```
String VariableDeclaratorId() :
{String id;}
{
  <IDENTIFIER>
  {id = token.image;}
//  ( "[" "]" )*
  {return id;}
}

Triple VariableInitializer(Gamma gamma) :
{Triple cs = null;}
{
/*
  "{" [ VariableInitializer()( LOOKAHEAD(2) "," VariableInitializer()
)* ] [ "," ] "}"
|
*/
  cs = Expression(gamma)
  {return cs;}
}

Dual MethodDeclaration(Gamma gamma) :
{
  Triple cs = null;
  Dual d = new Dual(cs,gamma);
  Gamma temp;
  Gamma param = new Gamma("param");
}
{
  ( "public" | "protected" | "private" | "static" | "abstract" |
"final" | "native" | "synchronized" )*
  ResultType()
  temp = MethodDeclarator(gamma,d)
  {
    while(!(temp.isEmpty())){
      GammaItem gi = (GammaItem)temp.getFromList();
      gamma = gamma.Append(gi);
      param = param.Append(gi);
      temp = temp.removeFromList();
    }//end while
  }
  [ "throws" NameList() ]
  ( cs = Block(gamma) | ";" )
  {
    GammaItem GI = new GammaItem(d.id, cs.getType(), "proc");
    GI.setParam(param);
    d.gamma = d.gamma.Append(GI);
    return new Dual(cs,d.gamma);
  }
}
```

```
Gamma MethodDeclarator(Gamma gamma, Dual d) :
{String id;}
{
  <IDENTIFIER> {id = token.image;}
  gamma = FormalParameters() ( "[" "]" )*
  {
    d.id = id;
    return gamma;
  }
}

Gamma FormalParameters() :
{Gamma temp = new Gamma("temp");}
{
  "(" [ temp = FormalParameter(temp) ( "," temp = FormalParameter(temp)
)* ] ")"
  {return temp;}
}

Gamma FormalParameter(Gamma gamma) :
{String id;}
{
  Type() id = VariableDeclaratorId()
  {
    gamma = gamma.Append(new GammaItem(id, sg.NextSymbol(),"var"));
    return gamma;
  }
}

Triple ConstructorDeclaration(Gamma gamma) :
{Triple cs = null;}
{
  [ "public" | "protected" | "private" ]
  <IDENTIFIER> gamma = FormalParameters() [ "throws" NameList() ]
  "{" //[ LOOKAHEAD(2) ExplicitConstructorInvocation() ]
      ( cs = BlockStatement(gamma) )* "}"
  {return cs;}
}

/*
void ExplicitConstructorInvocation() :
{}
{
  "this" Arguments() ";"
|
  "super" Arguments() ";"
}
```

```
void StaticInitializer() :
{}
{
  "static" Block())
}
*/

/*
 * Type, name and expression syntax follows.
 */

void Type() :
{}
{
  ( PrimitiveType() | Name() ) ( "[" "]" )*
}

void PrimitiveType() :
{}
{
  "boolean"
|
  "char"
|
  "byte"
|
  "short"
|
  "int"
|
  "long"
|
  "float"
|
  "double"
}

void ResultType() :
{}
{
  "void"
|
  Type()
}
```

```
String Name() :
/*
 * A lookahead of 2 is required below since "Name" can be followed
 * by a ".*" when used in the context of an "ImportDeclaration".
 */
{String id;}
{
  <IDENTIFIER>
  {id = token.image;}
//  ( LOOKAHEAD(2) "." <IDENTIFIER> )*
  {return id;}
}

void NameList() :
{}
{
  Name()
  ( "," Name()
  )*
}


/*
 * Expression syntax follows.
 */

Triple Expression(Gamma gamma) :
{Triple cs;}
{
  ( LOOKAHEAD( PrimaryExpression(gamma) AssignmentOperator() )
  cs = Assignment(gamma)
|
  cs = ConditionalOrExpression(gamma) )
  {return cs;}
}
```

```
Triple Assignment(Gamma gamma) :
{
  String id;
  Triple cs;
}
{
  id = PrimaryLeftExpression() AssignmentOperator() cs =
Expression(gamma)
  {
    GammaItem item = gamma.FindType(id);
    if(item != null){
      String mod = item.getModifier();
      if(mod.equals("var") || mod.equals("acc")){
        String tau = item.getType();
        String tauPrime = cs.getType();
        String alpha = sg.NextSymbol();
        ConstraintItem ci1 = new ConstraintItem(tau,tauPrime);
        ConstraintItem ci2 = new ConstraintItem(tauPrime,tau);
        ConstraintItem ci3 = new ConstraintItem(alpha,tauPrime);
        cs = cs.Append(ci1);
        cs = cs.Append(ci2);
        cs = cs.Append(ci3);
        cs.setModifier("cmd");
        cs.setType(alpha);
      }
      else{
        System.err.println("Secure Parse failed");
        System.exit(0);
      }//end if
    }
    else{
       System.out.println("Unrecognized variable " + id);
       System.exit(0);
    }//end if
    return cs;
  }
}

void AssignmentOperator() :
{}
{
  "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | ">>>=" |
"&=" | "^=" | "|="
}

/*
void ConditionalExpression() :
{}
{
  ConditionalOrExpression() [ "?" Expression() ":"
ConditionalExpression() ]
}
*/
```

```
Triple ConditionalOrExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;}
{
  cs1 = ConditionalAndExpression(gamma) ( "||" cs2 =
ConditionalAndExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
  )*
  {return cs1;}
}

Triple ConditionalAndExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = InclusiveOrExpression(gamma) ( "&&" cs2 =
InclusiveOrExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
  )*
  {return cs1;}
}
```

```
Triple InclusiveOrExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = ExclusiveOrExpression(gamma) ( "|" cs2 =
ExclusiveOrExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
  )*
  {return cs1;}
}

Triple ExclusiveOrExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = AndExpression(gamma) ( "^" cs2 = AndExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
  )*
  {return cs1;}
}
```

```
Triple AndExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = EqualityExpression(gamma) ( "&" cs2 = EqualityExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
  )*
  {return cs1;}
}

Triple EqualityExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = RelationalExpression(gamma) ( ( "==" | "!=" ) cs2 =
RelationalExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
  )*
  {return cs1;}
}

/*
void InstanceOfExpression() :
{}
{
  RelationalExpression() [ "instanceof" Type() ]
}
*/
```

```
Triple RelationalExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = ShiftExpression(gamma) ( ( "<" | ">" | "<=" | ">=" ) cs2 =
ShiftExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
)*
  {return cs1;}
}

Triple ShiftExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = AdditiveExpression(gamma) ( ( "<<" | ">>" | ">>>" ) cs2 =
AdditiveExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  }
  )*
  {return cs1;}
}
```

```
Triple AdditiveExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = MultiplicativeExpression(gamma) ( ( "+" | "-" ) cs2 =
MultiplicativeExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  } )*
  {
    return cs1;
  }
}

Triple MultiplicativeExpression(Gamma gamma) :
{
  Triple cs1;
  Triple cs2 = null;
}
{
  cs1 = UnaryExpression(gamma) ( ( "*" | "/" | "%" ) cs2 =
UnaryExpression(gamma)
  {
    if(cs2 != null){
      String tau1 = cs1.getType();
      String tau2 = cs2.getType();
      ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
      cs1 = cs1.Union(cs2).Append(ci1).Append(ci2);
    }
  } )*
  {
    return cs1;
  }
}
```

```
Triple UnaryExpression(Gamma gamma) :
{Triple cs;}
{
  (( "+" | "-" ) cs = UnaryExpression(gamma)
|
/*
  PreIncrementExpression()
|
  PreDecrementExpression()
|
*/
  cs = UnaryExpressionNotPlusMinus(gamma) )
  {return cs;}
}

/*
void PreIncrementExpression() :
{}
{
  "++" PrimaryExpression()
}

void PreDecrementExpression() :
{}
{
  "--" PrimaryExpression()
}
*/

Triple UnaryExpressionNotPlusMinus(Gamma gamma) :
{Triple cs;}
{
  (( "~" | "!" ) cs = UnaryExpression(gamma)
|
/*
  LOOKAHEAD( CastLookahead() )
  CastExpression()
|
*/
  cs = PostfixExpression(gamma) )
  {return cs;}
}
```

```
/*
// This production is to determine lookahead only.  The LOOKAHEAD
// specifications below are not used, but they are there just to
// indicate that we know about this.
void CastLookahead() :
{}
{
  LOOKAHEAD(2)
  "(" PrimitiveType()
|
  LOOKAHEAD("(" Name() "[")
  "(" Name() "[" "]"
|
  "(" Name() ")" ( "~" | "!" | "(" | <IDENTIFIER> | "this" | "super" |
"new" | Literal() )
}
*/

Triple PostfixExpression(Gamma gamma) :
{Triple cs;}
{
  cs = PrimaryExpression(gamma) //[ "++" | "--" ]
  {return cs;}
}

/*
void CastExpression() :
{}
{
  (LOOKAHEAD(2)
  "(" PrimitiveType() ( "[" "]" )* ")" UnaryExpression()
|
  "(" Name() ( "[" "]" )* ")" UnaryExpressionNotPlusMinus() )
}
*/

Triple PrimaryExpression(Gamma gamma) :
{Triple cs = null;}
{
  cs = PrimaryPrefix(gamma) //( PrimarySuffix(gamma) )*

  {return cs;}
}
```

```
Triple PrimaryPrefix(Gamma gamma) :
{
  Triple cs = null;
  Triple cs1 = null;
  Triple cs2 = null;
  String id = null;
  Gamma temp = null;
}
{
  (cs = Literal()
|
  ["this" "."] id = Name()
  {
    GammaItem item = gamma.FindType(id);
    if(item != null){
      String mod = item.getModifier();
      if(mod.equals("var") || mod.equals("")){
        String tau = item.getType();
        String alpha = sg.NextSymbol();
        ConstraintItem ci1 = new ConstraintItem(tau,alpha);
        cs = new Triple(ci1,alpha,"");
      }
      else if(mod.equals("proc")){
        temp = item.getParam();
      }
      else{
        System.err.println("Secure Parse failed");
        System.exit(0);
      }//end if
    }
    else{
      System.out.println("Undefined variable: " + id);
//    System.exit(0);
      temp = new Gamma("temp").Append(new GammaItem("",sg,""));
    }//end if
  }
  [ "(" [ cs1 = PrimaryPrefix(gamma)
  {
    //create constraint type(cs1) = type(param)
    String tauPrime = cs1.getType();
    String tau1 = ((GammaItem)temp.getFromList()).getType();
    temp.removeFromList();
    ConstraintItem ci1 = new ConstraintItem(tau1,tauPrime);
    ConstraintItem ci2 = new ConstraintItem(tauPrime,tau1);

    //add constraint to cs1
    cs1 = cs1.Append(ci1).Append(ci2);
    cs = cs1;
  }
  ( "," cs2 = PrimaryPrefix(gamma)
  {
    //create constraint type(cs2) = type(param)
    String tauDoublePrime = cs2.getType();
    String tau2 = ((GammaItem)temp.getFromList()).getType();
    temp.removeFromList();
    ConstraintItem ci3 = new ConstraintItem(tau2,tauDoublePrime);
```

```
      ConstraintItem ci4 = new ConstraintItem(tauDoublePrime,tau2);

      //cs1 Union cs2
      cs1 = cs1.Union(cs2);

      //add constraint to cs1
      cs1 = cs1.Append(ci3).Append(ci4);
      cs = cs1;
    }
    )* ] ")" ]
  /*
  |
    "this"
  |
    "super" "." <IDENTIFIER>
  */
  |
    "(" cs = Expression(gamma) ")"
  /*
  |
    AllocationExpression()
  */
    )
    { return cs;}
  }

  /*
  Triple PrimarySuffix() :
  {}
  {
    "[" Expression() "]"
  |
    "." <IDENTIFIER>
  |
    Arguments()
  }
  */

  String PrimaryLeftExpression() :
  {}
  {
    [ "(" ][ "this" "." ] Name() [ ")" ]
    {return token.image;}
  }
```

```
Triple Literal() :
{}
{
  ( <INTEGER_LITERAL>
|
  <FLOATING_POINT_LITERAL>
|
  <CHARACTER_LITERAL>
|
  <STRING_LITERAL>
|
  BooleanLiteral()
|
  NullLiteral() )
  {return new Triple(sg.NextSymbol(),"");}
}

void BooleanLiteral() :
{}
{
  "true"
|
  "false"
}

void NullLiteral() :
{}
{
  "null"
}

/*
void Arguments() :
{}
{
  "(" [ ArgumentList() ] ")"
}

Triple ArgumentList(Gamma gamma) :
{}
{
  Expression() ( "," Expression() )*
}
```

```
void AllocationExpression() :
{}
{
  LOOKAHEAD(2)
  "new" PrimitiveType() ArrayDimensions()
|
  "new" Name() ( Arguments() | ArrayDimensions() )
}
*/

/* The second LOOKAHEAD specification below is to parse to
 * PrimarySuffixif there is an expression between the "[...]". */
/*
void ArrayDimensions() :
{}
{
  ( LOOKAHEAD(2) "[" Expression() "]" )+ ( LOOKAHEAD(2) "[" "]" )*
}
*/

/*
 * Statement syntax follows.
 */
Triple Statement(Gamma gamma) :
{Triple cs = null;}
{
  (LOOKAHEAD(2)
/*
  LabeledStatement()
|
*/
  cs = Block(gamma)
|
  cs = EmptyStatement(gamma)
|
  cs = StatementExpression(gamma) ";"
|
/*
  SwitchStatement()
|
*/
  cs = IfStatement(gamma)
|
  cs = WhileStatement(gamma)
|
  cs = DoStatement(gamma)
/*
|
  ForStatement()
|
  BreakStatement()
|
  ContinueStatement()
|
  ReturnStatement()
|
  ThrowStatement()
|
  SynchronizedStatement()
|
  TryStatement()
*/
  )
  {return cs;}
}

/*
void LabeledStatement() :
{}
{
  <IDENTIFIER> ":" Statement()
}
*/

Triple Block(Gamma gamma) :
{Triple cs;}
{
  "{" cs = BlockStatementList(gamma) "}"
  {return cs;}
}

Triple BlockStatementList(Gamma gamma) :
{
  Triple cs1 = null;
  Triple cs2;
}
{
  ( LOOKAHEAD(2) cs2 = BlockStatement(gamma)
  {
    if(cs2 != null){
      if(cs1 == null){
        cs1 = cs2;
      }
      else{
        String tau1 = cs1.getType();
        String tau2 = cs2.getType();
        ConstraintItem ci1 = new ConstraintItem(tau1,tau2);
        ConstraintItem ci2 = new ConstraintItem(tau2,tau1);
        cs1 = cs1.Union(cs2);
        cs1 = cs1.Append(ci1);
        cs1 = cs1.Append(ci2);
      }//end if
    }//end if
  }
  )*
  {return cs1;}
}
```

```
Triple BlockStatement(Gamma gamma) :
{Triple cs;}
{
  (LOOKAHEAD(Type() <IDENTIFIER>)
  cs = LetvarStatement(gamma)
|
  cs = Statement(gamma))
  {return cs;}
}

Triple LetvarStatement (Gamma gamma) :
{
  Dual d;
  Triple cs = null;
}
{
  d = LocalVariableDeclaration(gamma) ";"
  {gamma = d.gamma;}
  cs = BlockStatementList(gamma)
  {
    if(cs != null){
      cs.Union(d.cs);
      cs.setModifier("cmd");
    }
    else{
      cs = d.cs;
    }
    return cs;
  }
}

Dual LocalVariableDeclaration(Gamma gamma) :
{Dual d;}
{
  Type()
  d = VariableDeclarator(gamma)
  ( "," VariableDeclarator(gamma) )*
  {return d;}
}

Triple EmptyStatement(Gamma gamma) :
{}
{
  ";"
  {return new Triple(sg.NextSymbol(),"cmd");}
}
```

```
Triple StatementExpression(Gamma gamma) :
/*
 * The last expansion of this production accepts more than the legal
 * Java expansions for StatementExpression.
 */
{Triple cs;}
{
  ( LOOKAHEAD( PrimaryExpression(gamma) AssignmentOperator(gamma) )
  cs = Assignment(gamma)
|
  cs = PostfixExpression(gamma) )
  {return cs;}
}

/*
void SwitchStatement() :
{}
{
  "switch" "(" Expression() ")" "{"
    ( SwitchLabel() ( BlockStatement() )* )*
  "}"
}

void SwitchLabel() :
{}
{
  "case" Expression() ":"
|
  "default" ":"
}
*/
```

```
Triple IfStatement(Gamma gamma) :
/*
 * The disambiguating algorithm of JavaCC automatically binds dangling
 * else's to the innermost if statement.  The LOOKAHEAD specification
 * is to tell JavaCC that we know what we are doing.
 */
{
  Triple cs;
  Triple cs1;
  Triple cs2 = null;
}
{
  "if" "(" cs = Expression(gamma) ")"
  cs1 = Statement(gamma)
  [ LOOKAHEAD(1) "else" cs2 = Statement(gamma) ]
  {
    String tau  = cs.getType();
    String tau1 = cs1.getType();
    String alpha = sg.NextSymbol();
    ConstraintItem ci1 = new ConstraintItem(tau,tau1);
    ConstraintItem ci2 = new ConstraintItem(tau1,tau);
    ConstraintItem ci3 = new ConstraintItem(alpha,tau);
    cs = cs.Union(cs1).Append(ci1).Append(ci2).Append(ci3);
    cs.setType(alpha);
    cs.setModifier("cmd");
    if(cs2 != null){
      String tau2 = cs2.getType();
      ConstraintItem ci4 = new ConstraintItem(tau,tau2);
      ConstraintItem ci5 = new ConstraintItem(tau2,tau);
      ConstraintItem ci6 = new ConstraintItem(tau1,tau2);
      ConstraintItem ci7 = new ConstraintItem(tau2,tau1);
      cs =
cs.Union(cs2).Append(ci4).Append(ci5).Append(ci6).Append(ci7);
    }//end if
    return cs;
  }
}
```

```
Triple WhileStatement(Gamma gamma) :
{
  Triple cs1 = null;
  Triple cs2 = null;
}
{
  "while" "(" cs1 = Expression(gamma) ")" cs2 = Statement(gamma)
  {
    String tau = cs1.getType();
    String tauPrime = cs2.getType();
    String alpha = sg.NextSymbol();
    ConstraintItem ci1 = new ConstraintItem(tau,tauPrime);
    ConstraintItem ci2 = new ConstraintItem(tauPrime,tau);
    ConstraintItem ci3 = new ConstraintItem(alpha,tau);
    cs1 = cs1.Union(cs2).Append(ci1).Append(ci2).Append(ci3);
    cs1.setType(alpha);
    cs1.setModifier("cmd");
    return cs1;
  }
}

Triple DoStatement(Gamma gamma) :
{
  Triple cs1 = null;
  Triple cs2 = null;
}
{
  "do" cs2 = Statement(gamma) "while" "(" cs1 = Expression(gamma) ")"
";"
  {
    String tau = cs1.getType();
    String tauPrime = cs2.getType();
    String alpha = sg.NextSymbol();
    ConstraintItem ci1 = new ConstraintItem(tau,tauPrime);
    ConstraintItem ci2 = new ConstraintItem(tauPrime,tau);
    ConstraintItem ci3 = new ConstraintItem(alpha,tau);
    cs1 = cs1.Union(cs2).Append(ci1).Append(ci2).Append(ci3);
    cs1.setType(alpha);
    cs1.setModifier("cmd");
    return cs1;
  }
}

/*
void ForStatement() :
{}
{
  "for" "(" [ ForInit() ] ";"
            [ Expression() ] ";"
            [ ForUpdate() ] ")"
  Statement()
}
```

```
void ForInit() :
{}
{
  LOOKAHEAD( Type() <IDENTIFIER> )
  LocalVariableDeclaration()
|
  StatementExpressionList()
}

void StatementExpressionList() :
{}
{
  StatementExpression() ( "," StatementExpression() )*
}


void ForUpdate() :
{}
{
  StatementExpressionList()
}

void BreakStatement() :
{}
{
  "break" [ <IDENTIFIER> ] ";"
}

void ContinueStatement() :
{}
{
  "continue" [ <IDENTIFIER> ] ";"
}

void ReturnStatement() :
{}
{
  "return" [ Expression() ] ";"
}

void ThrowStatement() :
{}
{
  "throw" Expression() ";"
}

void SynchronizedStatement() :
{}
{
  "synchronized" "(" Expression() ")" Block()
}
```

```
void TryStatement() :
{}
{
  "try"  Block()
  ( "catch" "(" FormalParameter() ")" Block() )*
  [ "finally" Block() ]
}
*/
```

APPENDIX B - STATIC ANALYZER SOURCE CODE

```
//*********************************************************************
//  File: Gamma.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer.  Basically a linked list.
//*********************************************************************
package thesis;

import java.io.*;

public class Gamma
{
  protected Object obj;
  protected Gamma next;
  protected Gamma rear = null;
  public String name;

  public Gamma(String name)
  {
    this.obj = null;
    this.next = this;
    this.name = name;

    if(rear == null)
    rear = this;
  }

  private Gamma()
  {
    this.obj = null;
    this.next = this;
  }

  public Object getFromList()
  {
    return this.obj;
  }

  public Gamma removeFromList()
  {
    return this.next;
  }

  public synchronized boolean isEmpty()
  {
    if(this == rear)
      return true;
    else
      return false;
  }
public Gamma Append(GammaItem gi)
  {
    Gamma g = new Gamma();
    g.obj = gi;
    g.next = this;
    return g;
  }

  public GammaItem FindType(String name)
  {
    GammaItem temp = null;
    Gamma list = this;
    boolean matchFound = false;

    do{
      if(list.obj == null)
        return null;

      String item = ((GammaItem)list.obj).Name;

      if(item.equals(name)){
        temp = (GammaItem)list.obj;
        matchFound = true;
      }
      else{
        list = (Gamma)list.next;
      }//end if
    }while(!matchFound);

    return temp;
  }


  public String toString()
  {
    if(isEmpty())
      return "";
    else
      return(this.obj + "" + this.next);
  }
}//end gamma class
```

```
//*********************************************************************
//  File: GammaItem.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer.  It is an item to be placed into gamma. The
//            structure consists of a name and a type.  The type may
//            consist of 1-3 fields.
//*********************************************************************
package thesis;

import java.io.*;

public class GammaItem
{
  protected String Name;
  protected String Type;
  protected String Modifier;
  private Gamma param;

  public GammaItem(String Name, SymbolGenerator sg, String mod)
  {
    this.Name = Name;
    this.Type = sg.NextSymbol();
    this.Modifier = mod;
  }

  public GammaItem(String Name, String Type, String mod)
  {
    this.Name = Name;
    this.Type = Type;
    this.Modifier = mod;
  }

  public void setParam(Gamma gamma)
  {
    this.param = gamma;
  }

  public Gamma getParam()
  {
    return this.param;
  }

  public String getName()
  {
    return this.Name;
  }

  public String getType()
  {
    return this.Type;
  }
  public String getModifier()
  {
    return this.Modifier;
  }

  public String toString()
  {
    if(Modifier.equals("proc")){
      return ("("+ Name +":"+ Type + Modifier +"("+ param +")"+")");
    }
    else{
      return ( "(" + Name + ":" + Type +  Modifier + ")" );
    }//end if
  }
}//end class
```

```
//*********************************************************************
//  File: Triple.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer.  The structure consists of a constraint set
//            and a principle type.  The type may consist of 1-2
//            fields.
//*********************************************************************
package thesis;

public class Triple
{
  private LinkedList ConstraintSet;
  private String Type;
  private String TypeModifier;

  public Triple()
  {
    this.Type = "Type";
    this.TypeModifier = "mod";
    ConstraintSet = new LinkedList("name");
  }

  public Triple(ConstraintItem ci, String Type, String Modifier)
  {
    ConstraintSet = new LinkedList("name");
    this.Type = Type;
    this.TypeModifier = Modifier;
    ConstraintSet = ConstraintSet.addToList(ci);
  }

  public Triple(LinkedList ConstraintSet, String Type, String Modifier)
  {
    this.Type = Type;
    this.TypeModifier = Modifier;
    this.ConstraintSet = ConstraintSet;
  }

  public Triple(String Type, String Modifier)
  {
    this.Type = Type;
    this.TypeModifier = Modifier;
    ConstraintSet = new LinkedList("name");
  }

  public String getType()
  {
    return this.Type;
  }
  public String getModifier()
  {
    return this.TypeModifier;
  }

  public void setModifier(String Modifier)
  {
    this.TypeModifier = Modifier;
  }

  public void setType(String type)
  {
    this.Type = type;
  }

  public Triple Union(Triple setTwo)
  {
    LinkedList temp = this.ConstraintSet;
    if((ConstraintItem)temp.obj == null){
      return new Triple(setTwo.ConstraintSet,this.Type,
                                                    this.TypeModifier);
    }

    while(temp.next.obj != null){
      temp = temp.next;
    }

    temp.next = setTwo.ConstraintSet;
    this.ConstraintSet.rear = setTwo.ConstraintSet.rear;
    return  new Triple(this.ConstraintSet,this.Type,this.TypeModifier);
  }

  public Triple Append(ConstraintItem C)
  {
    return new Triple(this.ConstraintSet.addToList(C), this.Type,
                                                      this.TypeModifier);
  }

  public String toString()
  {
    return("{"+"["+ ConstraintSet +"]"+","+ Type + TypeModifier +"}" );
  }
}//end class
```

```
//*******************************************************************
//  File: ConstraintItem.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer.  The structure consists two types.
//******************************************************************
package thesis;

public class ConstraintItem
{
  protected String Type1;
  protected String Type2;

  public ConstraintItem(String Type1, String Type2)
  {
    this.Type1 = Type1;
    this.Type2 = Type2;
  }

  public String toString()
  {
    return ( "(" + Type1 + "," + Type2 + ")" );
  }
}
```

```
//*******************************************************************
//  File: LinkedList.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer.
//******************************************************************
package thesis;

public class LinkedList
{
  protected Object obj;
  protected LinkedList next;
  protected LinkedList rear = null;
  public String name;

  public LinkedList(String name)
  {
    this.obj = null;
    this.next = this;
    this.name = name;

    if(rear == null)
      rear = this;
  }

  private LinkedList()
  {
    this.obj = null;
    this.next = this;
  }

  public LinkedList addToList(Object o)
  {
    LinkedList l = new LinkedList();
    l.obj = o;
    l.next = this;
    return l;
  }

  public Object getFromList()
  {
    return this.obj;
  }

  public LinkedList removeFromList()
  {
    return this.next;
  }
  public synchronized boolean isEmpty()
  {
    if(this == rear)
      return true;
    else
      return false;
  }

  public String toString()
  {
    if(isEmpty())
      return "";
    else
      return(this.obj + " " + this.next);
  }
}//end class
```

```
//*******************************************************************
//  File: SymbolGenetator.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer. Generates new type variables
//******************************************************************
package thesis;

import java.io.*;
import java.lang.*;

public class SymbolGenerator
{
  private int counter = 0;
  private static String TAU = "tau";

  public synchronized String NextSymbol()
  {
    String Symbol = TAU + counter;
    counter++;
    return Symbol;
  }

  public static void main(String [] args)
  {
    SymbolGenerator sg = new SymbolGenerator();

    for(int i = 0; i < 10; i++){
      System.out.println(sg.NextSymbol());
    }
  }
}//end class
```

```
//*********************************************************************
//  File: SymbolGenetator.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer. A data structure
//*******************************************************************
package thesis;

public class Dual
{
  public Triple cs;
  public Gamma gamma;
  public String id;

  public Dual(Triple cs, Gamma gamma)
  {
    this.cs = cs;
    this.gamma = gamma;
  }
}
```

APPENDIX C - TEST PROGRAMS

```
//*******************************************************************
//  File: test.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer.
//*******************************************************************
class test
{
  public static void p1(int x, int y)
  {
    y = x;
  }
}
```

The output of the static analyzer on the above program produced the following results:

Constraint set: $\{\tau_3 \leq \tau_2, \tau_2 = \tau_1, \tau_0 \leq \tau_2\}$

Gamma: `p1` : τ_3 proc ($\tau_0$var, $\tau_1$var)

Results show, with x: $\tau_0$var and y: $\tau_1$var, that $\tau_0 \leq \tau_1$. This is what we would expect to ensure secure flow since the program assigns the value of `x` to `y`.

```
//*********************************************************************
//  File: test.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information flow static
//            analyzer.
//*******************************************************************
class test
{
  public static void p1(int x, int y)
  {
    if(x == 0)
      y = 0;
    else
      y = 1;
  }
}
```

The output of the static analyzer on the above program produced the following results:

Constraint set: $\{\tau_7 = \tau_5,\ \tau_7 = \tau_2,\ \tau_8 \leq \tau_2,\ \tau_5 = \tau_2,\ \tau_3 = \tau_2,$
$\tau_0 \leq \tau_2,\ \tau_5 \leq \tau_4,\ \tau_4 = \tau_1,\ \tau_7 \leq \tau_6,\ \tau_6 = \tau_1\ \}$

Gamma: p1 : τ_8 proc ($\tau_0$var, $\tau_1$var)

```
//*********************************************************
//  File: test.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information
//                flow static analyzer.
//*********************************************************
class test
{
  public static void p1(int x, int y)
  {
    int a = x;
    int b = 0;
    while (a > 0){
      b = b + 1;
      a = a - 1;
    }
    y = b;
  }
}
```

The output of the static analyzer on the above program produced the following results:

Constraint set: $\{\tau_{14} = \tau_{12}, \tau_{12} \leq \tau_4, \tau_8 = \tau_4, \tau_5 = \tau_4, \tau_2 \leq \tau_4, \tau_{11} = \tau_8,$
$\tau_8 \leq \tau_6, \tau_6 = \tau_3, \tau_7 = \tau_6, \tau_3 \leq \tau_6, \tau_{11} \leq \tau_9, \tau_9 = \tau_2,$
$\tau_{10} = \tau_9, \tau_2 \leq \tau_9, \tau_{14} \leq \tau_{13}, \tau_1 = \tau_{13}, \tau_3 \leq \tau_{13}, \tau_0 \leq \tau_2\}$

Gamma = `p1`: τ_{12} proc($\tau_0$var, $\tau_1$var)

A partial trace of the analysis of this program is shown in Chapter VI.

```
//*********************************************************
//  File: test.java
//  Date: 24 Feb 98
//
//  Author: LT James D. Harvey, USN
//
//  Purpose:  Developed as part of a secure information
//            flow static analyzer.
//*********************************************************
class test
{
  public static void p1(int x, int y)
  {
    int a = x;
    int b = 0;
    while (a > 0){
      b = b + 1;
      a = a - 1;
    }
    y = b;
  }

  public static void p2(int a, int b)
  {
    a = a + 4;
    b = b + 2;

    if(a > b){
       p1(b,a);
    }else{
       p1(a,b);
    }
    b = a + b;
  }
  public static void main()
  {
    int s = 1;
    int t = 8;
    do{
      p2(2,t);
      t = t - 1;
    }while(t > 3);
  }
}
```

The output of the static analyzer on the above program produced the following results:

1. The First procedure, p1, produces:

   Constraint set: $\{\tau_{14} = \tau_{12},\ \tau_{12} \leq \tau_4,\ \tau_6 = \tau_4,\ \tau_5 = \tau_4,\ \tau_2 \leq \tau_4,\ \tau_9 = \tau_6,$
   $\tau_8 \leq \tau_6,\ \tau_6 = \tau_3,\ \tau_7 = \tau_6,\ \tau_3 \leq \tau_6,\ \tau_{11} \leq \tau_9,\ \tau_9 = \tau_2,$
   $\tau_{10} = \tau_9,\ \tau_2 \leq \tau_9,\ \tau_{14} \leq \tau_{13},\ \tau_{13} = \tau_1,\ \tau_3 \leq \tau_{13},\ \tau_0 \leq \tau_2\}$

   Gamma: `p1:` τ_{12}proc `(`$\tau_0$var`,` $\tau_1$var`)`

2. The second procedure, p2, produces:

   Constraint set: $\{\tau_{30} = \tau_{17},\ \tau_{29} = \tau_{17},\ \tau_{20} = \tau_{17},\ \tau_{19} \leq \tau_{17},\ \tau_{17} = \tau_{15},$
   $\tau_{18} = \tau_{17},\ \tau_{15} \leq \tau_{17},\ \tau_{22} \leq \tau_{20},\ \tau_{20} = \tau_{16},\ \tau_{21} = \tau_{20},$
   $\tau_{16} \leq \tau_{20},\ \tau_{27} = \tau_{25},\ \tau_{27} = \tau_{23},\ \tau_{29} \leq \tau_{23},\ \tau_{25} = \tau_{23},$
   $\tau_{24} = \tau_{23},\ \tau_{15} \leq \tau_{23},\ \tau_{16} \leq \tau_{24},\ \tau_{26} = \tau_0,\ \tau_{25} = \tau_0,$
   $\tau_{16} \leq \tau_{25},\ \tau_{15} \leq \tau_{26},\ \tau_{28} = \tau_0,\ \tau_{27} = \tau_0,\ \tau_{15} \leq \tau_{27},\ \tau_{16} \leq \tau_{28},$
   $\tau_{32} \leq \tau_{30},\ \tau_{30} = \tau_{16},\ \tau_{31} = \tau_{30},\ \tau_{15} \leq \tau_{30},\ \tau_{16} \leq \tau_{31}\ \}$

   Gamma: `p2 :` τ_{17}proc `(`τ_{15}var`,` τ_{16}var`) ,`
   `p1 :` τ_{12}proc `(`$\tau_0$var`,` $\tau_1$var`)`

3. The third procedure, main, produces:

   Constraint set: $\{\tau_{42} \leq \tau_{40},\ \tau_{35} = \tau_{40},\ \tau_{41} = \tau_{40},\ \tau_{34} \leq \tau_{40},$
   $\tau_{37} = \tau_{35},\ \tau_{36} = \tau_{15},\ \tau_{35} = \tau_{15},\ \tau_{34} \leq \tau_{36},$
   $\tau_{39} \leq \tau_{37},\ \tau_{37} = \tau_{34},\ \tau_{38} = \tau_{37},\ \tau_{34} \leq \tau_{37}\}$

   Gamma: `main :` τ_{42}proc `(),`
   `p2 :` τ_{17}proc `(a:`τ_{15}var`,` `b:`τ_{16}var),
   `p1 :` τ_{12}proc `(x:`$\tau_0$var`,` `y:`$\tau_1$var`)`

4. Gamma is updated with each procedure.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center 8725 John J. Kingman Road, Suite 0944 Fort Belvoir, VA 22060	2
2.	Dudley Knox Library Naval Postgraduate School 411 Dyer Road Monterey, CA 93940	2
3.	Dr. Dan Boger, Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93940	1
4.	Dr. Dennis Volpano, Code CS/Vd Computer Science Department Naval Postgraduate School Monterey, CA 93940	3
5.	Dr. Craig Rasmussen, Code MA/Ra Department of Mathematics Naval Postgraduate School Monterey, CA 93940	1
6.	LT. James D. Harvey 7090 Brook Dr. Morrow, OH 45152	3