1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

DISCOVERING ANOMALIES IN ACCESS MODIFIERS IN JAVA WITH A FORMAL SPECIFICATION

Wuu Yang
Department of Computer and Information Science
National Chiao-Tung University, Hsinchu, Taiwan, R.O.C.
E-mail: wuuyang@cis.nctu.edu.tw

ABSTRACT

We use attribute grammars to formally specify the semantics of the access modifiers in the Java language. This formal specification uncovers several situations that are irregular or counter-intuitive. These situations are confusing to the programmers and may create weaknesses in Java program. From this exercise, we learn that a formal specification is indispensable in designing a new language.

1. Introduction

Attribute grammars, proposed by Knuth in 1968 [10], were intended for specifying the semantics of programming languages. Most research in attribute grammars is focused on the evaluation strategies [1, 9, 12, and storage management [2, 7, 8, 11] of the attribute evaluators. Some researchers works on extending the language of attribute grammar for specifications [4, 6]. Only a few reports [13, 14] are available that demonstrate the actual use of attribute grammar in specifying (parts of) the semantics of a practical programming language. In this paper, we use attribute grammars to study the semantics of the access modifiers of the Java language.

Java is intended to be a *secure* language. To achieve this goal, it is necessary for Java to be defined unambiguously and completely because any ambiguities or omissions in the language specification could possibly lead to a breach of the security aspect of Java applications. Unfortunately, the semantics of Java is explained with an informal description [3]. It is hard to be certain that the informal description is unambiguous and complete. A formal specification of the semantics of Java is, thus, indispensable. Furthermore, a formal specification also makes it possible to study alternative semantics.

Java defines three access modifiers (`private`, `protected`, and `public`), which are intended to control the visibility of members (functions and variables) of a class. The three access modifiers constitute the basic mechanism for maintaining the integrity of Java applications. In order to clarify the semantics of the access modifiers, we developed a formal specification in terms of attribute grammars. In the course of developing the formal specification, we discovered a few situations that are irregular or counter-intuitive in the original Java language specification.

A common criticism of formal specifications is that it is too complicated for a practical programming language. In order to remedy this problem, rather than a single, complete formal specification of the whole language, it is feasible to write a set of small formal specifications, one for each critical aspect of the programming language. Because the small specifications are much simpler than the complete one, they are much easier to write and to understand. Furthermore, because the small specifications are based on a common context-free grammar, it is possible to integrate them together automatically. The formal specification presented here, which concentrates on the access modifiers, is an attempt in this approach.

2. Notations

In this section, we introduce attribute grammars. An attribute grammar is built from a context-free grammar (N, T, P, S), where N is a finite set of nonterminals, T is a finite set of terminals, S is a distinguished nonterminal, called the *start symbol*, and P is a set of productions of the form: $X \longrightarrow \alpha$, where X is a nonterminal and α is a string of terminals and nonterminals. For each nonterminal X, there is at least one production whose left-hand-side symbol is X. Furthermore, we assume that the start symbol does not appear in the right-hand side of any production. As usual, we require that the sets of terminals and nonterminals be disjoint.

Attached to each symbol X of the context-free grammar is a set of *attributes*. Intuitively, instances of attributes describe the properties of specific instances of symbols in a syntax tree. The attributes of a symbol are partitioned into two disjoint subsets, called the *inherited* attributes and the *synthesized* attributes. We will assume that the start symbol has no inherited attributes and that a terminal has only a synthesized attribute that represents

the character string comprising the terminal symbol. An attribute a of a symbol X is denoted by $X.a$.

There are attribution equations defining these attributes. In a production, there are attribution equations defining synthesized attributes of the left-hand-side symbol and inherited attributes of the right-hand-side symbols.

An attribute grammar may also contain semantic conditions and additional operations and global data structures. Semantic conditions are boolean expressions made up of the attributes (and constants). These semantic conditions enforce the semantic requirements of a programming language, such as the requirement that a variable must be declared before being used. These semantic conditions must evaluate to *true* in any legal syntax tree. The global data structure, such as a symbol table, stores information that cannot be easily coded into attributes.

3. The attribute grammar for access modifiers

In the attribute-grammar specification, we need two operations *overrideF* and *overrideV* to compute the inherited member functions and the inherited member variables of a class, respectively. The reason we need two separate operations is due to the different ways in which functions and variables are inherited. In Java, a new member variable overrides an inherited one if the two have the same name. It does not matter whether the two variables have the same type. By contrast, a new member function can override an inherited one only if the new function has both the same name and the same type as the inherited function. (The *type* of a function, also called the *signature*, consists of the number and the types of parameters and the type of the return value.) If the new function has a different type than the inherited function, the overloading mechanism of Java makes both the new and the inherited member functions visible simultaneously. This reflects an irregularity of Java.

Let X and Y be two sets of elements. Each element has the form $[f:(A, type)]$, where f is a member (a function or a variable) of a class, A is the class in which f is declared, and *type* is the type of f. The *overrideF* and *overrideV* operations are defined as follows.

$overrideF(X, Y) = \{ [f:(A, type)] \mid$
$[f:(A, type)] \in X$ and $[f:(B, type)] \notin Y$,
for any class B $\}$

$overrideV(X, Y) = \{ [f:(A, type)] \mid$ .ps 10
$[f:(A, type)] \in X$ and $[f:(B, newtype)] \notin Y$,
for any class B and any type *newtype* $\}$

The *hiddenF* and *hiddenV* operations collect the inherited but hidden member functions and variables of a class, respectively.

$hiddenF(X, Y) = \{ [f:(A, type)] \mid [f:(A, type)]$
$\in X$ and $[f:(B, type)] \in Y$, for some class B $\}$

$hiddenV(X, Y) = \{ [f:(A, type)] \mid [f:(A, type)]$
$\in X$ and $[f:(B, newtype)] \in Y$,
for some class B and some type *newtype* $\}$

We may verify that

$hiddenF(X, Y) = X - overrideF(X, Y)$

$hiddenV(X, Y) = X - overrideV(X, Y)$

3.1. Classes

There are four kinds of access modifications for members of a class: `public`, `protected`, `private`, or no modifier at all. The following declaration of class `B`, which is a subclass of class `A` and contain eight members, is a typical example. The first four members—`f`, `g`, `h`, `m`— are member functions; the remaining four are member variables.

```
public class B extends A {
  public    int     f(boolean a)
                    { . . . }
  protected boolean g(float b)
                    { . . . }
  private   float   h(int c)
                    { . . . }
            boolean m(int d)
                    { . . . }
  public    int     u;
  protected boolean v;
  private   float   w;
            boolean m;
}
```

In order to specify the semantics of the access modifiers, we first translate the class declarations into a context-free grammar. Each class `A` corresponds to a nonterminal A. The production rules are derived from the inheritance relationships among classes. If class `B` is a subclass of class `A`, there is a production rule $B \longrightarrow A$. If class `B` has no superclass, there is a production rule $B \longrightarrow \varepsilon$. In Java, every class is a descendant of the `Object` class. Only the `Object` class has no superclass.

Since Java allows only single inheritance, the production rules are very simple. However, our approach is applicable to the general case of multiple inheritance (as in C++). The complicated resolution rules for name conflicts among members that are inherited from different superclasses would be encoded in the *overrideF* and *overrideV* operations.

The above class declaration induces the following production rule and a set of attribution equations:

$B \longrightarrow A$
$B.modifier$ = "public"
$B.publicF = \{ [f : (B, boolean \longrightarrow int)] \}$
$B.protectedF = \{ [g : (B, float \longrightarrow boolean)] \}$
$B.privateF = \{ [h : (B, int \longrightarrow float)] \}$
$B.noneF = \{ [m : (B, int \longrightarrow boolean)] \}$
check $noDuplicateF(B.publicF, B.protectedF, B.privateF, B.noneF) = true$
$B.publicV = \{ [u : (B, boolean)] \}$
$B.protectedV = \{ [v : (B, float)] \}$
$B.privateV = \{ [w : (B, int)] \}$
$B.noneV = \{ [x : (B, int)] \}$
check $noDuplicateV(B.publicV, B.protectedV, B.privateV, B.noneV) = true$
$B.localF = B.publicF \cup B.protectedF \cup B.privateF \cup B.noneF$
$B.localV = B.publicV \cup B.protectedV \cup B.privateV \cup B.noneV$
$B.inheritedPublicF = overrideF(A.exportPublicF, B.localF)$
$B.inheritedProtectedF = overrideF(A.exportProtectedF, B.localF)$
$B.inheritedPrivateF = overrideF(A.exportPrivateF, B.localF)$
$B.inheritedNoneF = overrideF(A.exportNoneF, B.localF)$
check $hiddenF(A.exportPublicF, B.privateF \cup B.protectedF \cup B.noneF) = \emptyset$
check $hiddenF(A.exportProtectedF, B.privateF \cup B.noneF) = \emptyset$
check $hiddenF(A.exportNoneF, B.privateF) = \emptyset$
$B.inheritedPublicV = overrideV(A.exportPublicV, B.localV)$
$B.inheritedProtectedV = overrideV(A.exportProtectedV, B.localV)$
$B.inheritedPrivateV = overrideV(A.exportPrivateV, B.localV)$
$B.inheritedNoneV = overrideV(A.exportNoneV, B.localV)$
$B.visibleInClassF = B.localF \cup B.inheritedPublicF \cup B.inheritedProtectedF \cup \{ [f : (C, type)] \mid [f : (C, type)] \in B.inheritedNoneF$ and classes B and C are in the same package $\}$
$B.visibleInClassV = B.localV \cup B.inheritedPublicV \cup B.inheritedProtectedV \cup \{ [f : (C, type)] \mid [f : (C, type)] \in B.inheritedNoneV$ and classes B and C are in the same package $\}$
$B.visibleInClass = B.visibleInClassF \cup B.visibleInClassV$
$B.exportPublicF = B.publicF \cup B.inheritedPublicF$
$B.exportProtectedF = B.protectedF \cup B.inheritedProtectedF$
$B.exportPrivateF = B.privateF \cup B.inheritedPrivateF$
$B.exportNoneF = B.noneF \cup B.inheritedNoneF$
$B.exportPublicV = B.publicV \cup B.inheritedPublicV$
$B.exportProtectedV = B.protectedV \cup B.inheritedProtectedV$
$B.exportPrivateV = B.privateV \cup B.inheritedPrivateV$
$B.exportNoneV = B.noneV \cup B.inheritedNoneV$

The first attribute, *modifier*, of a class nonterminal denotes the modifier of the class itself. In Java, a class may or may not have the `public` modifier. The eight synthesized attributes, *publicF*, *protectedF*, *privateF*, *noneF*, *publicV*, *protectedV*, *privateV*, and *noneV*, contains the members (functions and variables) of the class that are declared with the respective modifiers.

Because Java does not allow duplicate definitions of the same member in a class, two checks are employed to enforce this rule. Java allows member functions with different types to have the name. On the other hand, Java forbids member variables of the same class to have the same name, even if they have different types. This is a second irregularity of Java. The two checks—*noDuplicateF* and *noDuplicateV*—prevent duplicate definition of member functions and variables, respectively.

The synthesized attribute *localF* is the set of member functions defined in the class, which, by definition, is the union of the four attributes *publicF*, *protectedF*, *privateF*, and *noneF*. Similarly, the synthesized attribute *localV* is the set of member variables defined in the class, which, by definition, is the union of the four attributes *publicV*, *protectedV*, *privateV*, and *noneV*.

The four synthesized attributes *inheritedPublicF*, *inheritedProtectedF*, *inheritedPrivateF*, and *inheritedNoneF* are the sets of member functions that are inherited from the superclass with the respective access modifiers. Similarly, the four synthesized attributes *inheritedPublicV*, *inheritedProtectedV*, *inheritedPrivateV*, and *inheritedNoneV* are the sets of member variables that are inherited from the superclass with the respective access modifiers.

Since some of the inherited members might be redefined in a class, the operations *overrideF* and *overri-*

deV are used to eliminate the overridden members. By the definition of the Java language, an inherited member function is overridden if a member function with the same name and the same type is defined in the subclass. This rule allows the existence of overloaded function names. When a function is referenced (that is, invoked), the Java compiler would find one of the overloaded functions that possesses the appropriate type. Note that the rule for member variables are different from that for member functions. An inherited member variable is overridden if a new variable with the same name, which may or may not have the same type, is defined in the subclass. The difference is reflected in the definitions of the *overrideF* and *overrideV* operations.

Java imposes a rule which says that, when a new member function hides an inherited one (of the same name, of course), the new member function must have equal or greater visibility than the inherited one. For instance, a private member function cannot override an inherited public member function (of the same name, of course). Note, however, that this rule does not apply to member variables. A private member variable *can* override an inherited public one. This reflects a third irregularity of Java. The three checks of hidden member functions in the specification enforce this rule; there are no corresponding checks of hidden member variables. The following example illustrates the rule.

```
class Red {
  public int x = 1;
  public int f() { return 99; }
}
class Blue extends Red {
  private int x = 2;  // ok
  //  private int f()
      // { return 200; } - error
  //  protected int f()
      // { return 200; } - error
  public int f()
      { return 200; } // ok
}
```

Class `Blue` is a subclass of `Red`. The public member variable `x` defined in `Red` may be overridden by a new private member variable of the same name in class `Blue`. However, the public member function `f` defined in `Red` may *not* be overridden by a new private or protected one with the same name in `Blue`.

Note that the inheritance and overriding of member variables and member functions are performed independently in Java. This is why we need two independent sets of attributes, one set for member functions, the other for member variables. Consider the following example.

```
class Alpha {
  int a = 3;
}
class Beta extends Alpha {
  boolean a (int x)
    { return (x == 1); }
}
public class VarFunction {
  public static void
  main(String[] args) {
    Beta b = new Beta(3);
    System.out.println(
      "b.a(1) = " + b.a(1));
    System.out.println(
      "b.a = " + b.a);
  }
}
```

Class `Beta` inherits the member variable `a` from class `Alpha`. Class `Beta` defines a new member function which is also called `a`. The member function `a` in class `Beta` does not hide the inherited member variable `a`. Both the member variable `a` and the member function `a` are visible simultaneously in class `VarFunction`.

The attribute *visibleInClassF* is the set of member functions that are visible in the class, which includes the locally defined functions and the visible inherited functions. The visible inherited member functions include all inherited public or protected functions and the inherited no-modifier functions defined in the same package. Similarly, the attribute *visibleInClassV* is the set of member variables that are visible in the class, which includes the locally defined variables and the visible inherited variables. The visible inherited member variables include all inherited public or protected variables and the inherited no-modifier variables defined in the same package.

Some, but not necessarily all, of the members that are visible in a class are visible in the subclasses. The eight attributes *exportPublicF*, *exportProtectedF*, *exportPrivateF*, *exportNoneF*, *exportPublicV*, *exportProtectedV*, *exportPrivateV*, and *exportNoneV* are the member functions and variables with the respective access modifiers that are exported to subclasses.

In Java as well as many object-oriented languages, members of a class may be declared as *class members* with the keyword `static`. The visibility rules governing class members are the same as those governing instance members.

There is a slight difference between the rules governing class functions and class variables: A class variable of a superclass may be overridden by a new declaration in a subclass; however, a class function may *not* be overridden. In the above formal specification, we have ignored the issues concerning static members.

3.2. Compilation units

Java introduces the `package` structure. A package consists of one or more compilation units. A compilation unit is usually made up of a file in a system with a traditional file system. Every class belongs to exactly one compilation unit, which, in turn, belongs to exactly one package. A class may be declared with or without the `public` keyword. A public class may be referenced without qualification in other packages if it is imported by the referencing packages. A non-public class may be referenced only within the package to which it belongs. In Java, importing is based on individual compilation units [3].

In the specification, every compilation unit `X` corresponds to a nonterminal X. There is a production $X \longrightarrow A\ B\ C \ldots$ for the nonterminal X, where A, B, C, etc., are all the classes defined in the compilation unit `X`. The order of A, B, C, etc. is insignificant.

Suppose that X represents a compilation unit, A, B, and C represents the classes defined in X. The following grammar rule is introduced into the specification.

$X \longrightarrow A\ B\ C$
$X.localClasses = \{A, B, C\}$
$X.publicClasses =$
$(if\ A.modifier="public"\ then\ \{A\}\ else\ \emptyset) \cup$
$(if\ B.modifier="public"\ then\ \{B\}\ else\ \emptyset) \cup$
$(if\ C.modifier="public"\ then\ \{C\}\ else\ \emptyset)$
$X.importedPac$ = the set of packages imported into compilation unit X
$X.importedClasses$ = the set of public classes imported into compilation unit X
$A.visibleClasses = X.visibleClasses$
$B.visibleClasses = X.visibleClasses$
$C.visibleClasses = X.visibleClasses$

The *localClasses* attribute of a compilation unit X is the set of classes defined in the compilation unit. The *publicClasses* attribute is the set of public classes defined in the compilation unit (which usually contains one class). The *importedPac* attribute of X contains the packages that are imported into the compilation unit X. The *visibleClasses* attribute of a class nonterminal, say A, is the set of classes that are either public classes imported into the compilation unit containing A or are defined in the package containing A.

Java also allows a compilation unit to selectively import individual classes of another package. In this case, the imported classes may be added to the *importedClasses* attribute of the compilation unit.

Because $X.localClasses \subseteq X.visibleClasses$ (see the next subsection), the class A belongs to $A.visibleClasses$. This implies that class A is visible inside its own definition. The following example demonstrates this issue of visibility. The class `CC` may be referenced in its instance method `selfRef`. This issue is concisely captured by the specification.

```
class CC {
  public int a;
  CC(int b) { a = b; };
  public int selfRef() {
    CC c5 = new CC(5);
    return (a+c5.a);
  }
}
```

Compilation units are designed for separate compilation. Each class is placed in a separate compilation unit. When a class is changed, only that class, not the whole package, needs to be re-compiled. However, compilation units are also involved in the visibility issue. When a compilation unit imports a package, it may import individual public classes or it may import the whole package, which may be made up of several compilation units. On the other hand, the import statement only affects the compilation unit that contains the statement. Other compilation units in the same package are not affected by the import statement.

3.3. Packages

A package in Java contains one or more compilation units. In the specification, every package `P` is represented by a nonterminal P. Every compilation unit `X` is similarly represented by a nonterminal X (in fact, compilation units do not have names; but we can imagine that each compilation unit is implicitly given a name in a systematical way). There is a production $P \longrightarrow X\ Y\ Z \ldots$ in the specification, where X, Y, Z, etc., are all the compilation units of package P. The order of the compilation units in the right-hand side of the production is insignificant.

Suppose that P represents a package, X, Y, and Z are all the compilation units defined in P. The following grammar rule is introduced into the specification.

$P \longrightarrow X\ Y\ Z$
$P.localClasses = X.localClasses \cup$
$Y.localClasses \cup Z.localClasses$
$P.publicClasses = X.publicClasses \cup$

$Y.publicClasses \cup Z.publicClasses$
Enter the pair $(P, P.publicClasses)$ into the global symbol table *PubClass*.
$X.multiDefinedClasses = \{A \mid A \in PubClass(U) \cap PubClass(V),$ where $U, V \in X.importedPac, U \neq V\} \cup (X.importedClasses \cap (\cup_{U \in X.importedPac} \{PubClass(U)\}))$
$Y.multiDefinedClasses = \{A \mid A \in PubClass(U) \cap PubClass(V),$ where $U, V \in Y.importedPac, U \neq V\} \cup (Y.importedClasses \cap (\cup_{U \in Y.importedPac} \{PubClass(U)\}))$
$Z.multiDefinedClasses = \{A \mid A \in PubClass(U) \cap PubClass(V),$ where $U, V \in Z.importedPac, U \neq V\} \cup (Z.importedClasses \cap (\cup_{U \in Z.importedPac} \{PubClass(U)\}))$
$X.visibleOutsideClasses = X.importedClasses \cup (\cup_{U \in X.importedPac} \{PubClass(U)\}) - (X.multiDefinedClasses \cup P.localClasses)$
$Y.visibleOutsideClasses = Y.importedClasses \cup (\cup_{U \in Y.importedPac} \{PubClass(U)\}) - (Y.multiDefinedClasses \cup P.localClasses)$
$Z.visibleOutsideClasses = Z.importedClasses \cup (\cup_{U \in Z.importedPac} \{PubClass(U)\}) - (Z.multiDefinedClasses \cup P.localClasses)$
$X.visibleClasses = P.localClasses \cup X.visibleOutsideClasses$
$Y.visibleClasses = P.localClasses \cup Y.visibleOutsideClasses$
$Z.visibleClasses = P.localClasses \cup Z.visibleOutsideClasses$

The *localClasses* attribute of a package nonterminal is the set of classes defined in the package. The *publicClasses* attribute of a package nonterminal is the set of public classes defined in the package.

In the attribute equations for *multiDefinedClasses* and *visibleOutsideClasses*, it is necessary to retrieve the *publicClasses* attributes of the packages that are imported into package P. Since the nonterminals for these imported packages do not occur in the current production, it is not possible to retrieve their attributes directly. Therefore, a global symbol table *PubClass* is set up that contains a pair $(P, P.publicClasses)$, for each package nonterminal P. In order to inquire the *publicClasses* attribute of a package U, the function $PubClass(U)$ is provided.

Since packages are developed independently, it is possible that two packages define distinct public classes that happen to have the same name. When both packages are imported into a compilation unit, there is a name conflict among the imported classes. It is also possible that a local class of a package, say P, has the same name as a public class of another package that is imported into a compilation unit of P. In such situations, Java dictates that, whenever there is a name conflict among classes, the local class is preferred, if one exists. If the name conflict does not involve a local class, classes with the conflicting name must be explicitly qualified with the names of the defining packages when they are referenced.

The *importedPac* attribute of a compilation unit, defined in the previous subsection, contains the packages imported into the compilation unit. The *importedClasses* attribute contains the public classes imported into the compilation unit. The *multiDefinedClasses* attribute of a compilation unit contains the names of classes that are defined in more than one imported package. The *visibleOutsideClasses* attribute contains the public classes defined in the imported packages that will not cause name conflicts. The *visibleClasses* attribute of a compilation unit is the set of classes that are either public classes imported into the compilation unit or are defined in the compilation unit.

One of the visibility rules of Java is quite counter-intuitive in that an inherited member of a class, say class P, might not be visible in P but becomes visible in a subclass of P. This situation does not arise in other object-oriented languages, such as C++. Consider the attribution equation for *B.visibleInClassV* in Section 3.1. A no-modifier member of a class, say A, is visible in all classes in the same package but it is not visible in a subclass of A that is located in a different package. For instance, consider the following example. In the first file (that is, compilation unit), AAA.java, two classes—AAA and CCC— are defined. Both classes belong to package R1.

```
// in file AAA.java
package R1;
import R2.*;
public class AAA {
  int alpha = 111;
}
class CCC extends BBB {
  int gamma = 333;
  public int inquireAlpha()
    { return alpha; } // ok
}
```

In the second file, BBB.java, a class, BBB, is defined and placed in package R2.

```
// in file BBB.java
package R2;
import R1.*;
public class BBB extends AAA {
  int beta = 222;
//  public int inquireAlpha()
//    { return alpha; }
//  The above line causes an
//  error: Variable alpha in
//  R1.AAA not accessible from
//  R2.BBB
}
```

The no-modifier member variable alpha is defined in class AAA, then is inherited by BBB, and is again passed to CCC. Note that, due to the visibility rules of packages and classes, alpha is invisible in class BBB but is visible in a subclass of BBB (namely, CCC). This counter-intuitive phenomenon is due to the fact that visibility in Java is transmitted not only along the inheritance hierarchy but also along the package structure. Packages and inheritance are totally independent.

3.4. Programs

A Java program consists of one or more packages. In the specification, there is a distinct nonterminal S that represents the program. Every package P is also represented by a nonterminal P. There is a production $S \longrightarrow P\ Q\ R\ \ldots$ in the specification, where P, Q, R, etc., are all the packages in the program S. The order of these package nonterminals in the right-hand side of the production is insignificant.

Suppose that S represents a program, P, Q, and R are all the packages in S. The following grammar rule is introduced into the specification.

$S \longrightarrow P\ Q\ R$

There is no attribute associated with the program nonterminal S in the specification.

4. Checking references with attributes

The attribute-grammar specification for access modifiers may be used when the compiler decides whether a reference is legal according to the visibility rules of Java.

4.1. Class references

When a class A is referenced in another class B, such as

```
class B {
  A a1;
}
```

we may check the permission with the *visibleClasses* attribute of nonterminal A. The above reference is legal if and only if $A \in B.visibleClasses$.

In Java, all the packages are located in a global name space. A compilation unit may import any packages or any public classes in any packages. After a class is imported into a compilation unit, it may be referenced without qualification. Alternatively, it is possible to refer to a class with a fully qualified name without importint it first. Consider the following example.

```
package P;
// there is no import statement
class B {
  Q.A a1;
}
```

The above reference is legal if and only if $A \in Q.publicClasses$.

4.2. Member references

A member function or a member variable of a class may be referenced in four ways in a Java program: First, it is possible to refer to the member without any qualification. Second, a member is referenced with a prefix that is the name of an object or a class. Third, a member is referenced with the prefix this. Finally, a member is referenced with the prefix super. The following code segment illustrates the four kinds of references.

```
class X extends Y {
  int k(int a) {
    ... name ...
    ... d.name ...
    ... this.name ...
    ... super.name ... }
}
```

For the first kind of reference, the following condition must be satisfied:

$[name : (A, type)] \in X.visibleInClass$, for some class A and some type $type$, or
name is a formal parameter or a local variable of function k.

For the second kind of reference, the following condition must be satisfied:

Let Z be the declared class of object d
(if d is a class, let Z be d).
$[name : (A, type)] \in X.visibleInClass$, for some class A and some type $type$,
if $Z = X$
$[name : (A, type)] \in Z.exportPublicF \cup Z.exportProtectedF \cup Z.exportPublicV \cup Z.exportProtectedV \cup \{ [f : (C, type)] \mid$

$[f : (C, type)] \in Z.exportNoneF \cup Z.exportNoneV$
and classes C and X are in the same package },
for some class A and some type *type*,
if X is a subclass of Z
$[name : (A, type)] \in Z.exportPublicF \cup$
$Z.exportPublicV \cup \{ [f : (C, type] \mid$
$[f : (C, type)] \in Z.exportNoneF \cup Z.exportNoneV$
and classes C and X are in the same package },
for some class A and some type *type*,
if X is not a subclass of Z

For the third kind of reference, the actual class of `this` must be class `X` or a subclass of `X`. Since the actual class of `this` is not known at compile time, it is safe to assume that `this` is an instance of class `X`. The following condition must be satisfied:
$[name : (A, type)] \in X.visibleInClass$, for some class A and some type *type*

The inherited, but overridden, members can still be referenced through the `super` pseudo variable. For the fourth kind of reference, the (declared) class of `super` is `Y`. The following condition must be satisfied:
$[name : (A, type)] \in Y.exportPublicF \cup$
$Y.exportProtectedF \cup Y.exportPublicV \cup$
$Y.exportProtectedV \cup \{ [f : (C, type] \mid$
$[f : (C, type)] \in Y.exportNoneF \cup Y.exportNoneV$
and classes C and X are in the same package },
for some class A and some type *type*

5. Conclusion

During writing the formal specification of the semantics of the access modifiers, we discovered a major weakness of Java, namely, member functions and member variables are not treated equally. In particular, overloading is allowed for member functions but not for member variables. The rule of a new member variable overriding an inherited one is also different from that for a member function. Furthermore, when a new member function overrides an inherited one, the new member function must have equal or greater visibility than the inherited one. This rule is only applicable to member functions, but not to member variables. The rules governing class functions and class variables are also different. We consider the unequal statuses of member functions and member variables an unnecessary complication. Our formal specification grows twice as big as if they were treated equally.

Another weakness that we discovered is that the visibility of members of a class is transmitted through two independent channels: the class hierarchy and package structures. Packages in Java are intended as a structuring mechanism for large-scale software development [3]. They should not be involved in the visibility of members of classes. For instance, in Java, packages and subpackages did not share any access privileges. We consider the Java rule which says that a member without any access modifier is visible to all classes in the same package a demerit. We suggest that the visibility of members of a class should be transmitted only through the class hierarchy.

Acknowledgement. This work was supported in part by National Science Council, Taiwan, R.O.C. under grants NSC 87-2213-E-009-024.

References

1. P. Deransart, M. Jourdan, and B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography* (Lecture Notes in Computer Science 323), Springer-Verlag, New York (1988).
2. J. Engelfriet and W. de Jong, Attribute storage optimization by stacks, *Acta Informatica* **27** pp. 567-581 (1990).
3. J. Gosling, W. Joy, and G. Steele, Jr., *The Java Language Specification*, Addison-Wesley, Reading, MA (1996).
4. R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite, Eli: A complete, flexible compiler construction system, *Comm. ACM* **35**(2) pp. 121-131 (February 1992).
5. U. Kastens, Ordered attribute grammars, *Acta Informatica* **13** pp. 229-256 (1980).
6. U. Kastens, B. Hutt, and E. Zimmermann, *GAG: A Practical Compiler Generator*, Springer-Verlag, New York (1982).
7. U. Kastens, Lifetime analysis for attributes, *Acta Informatica* **24** pp. 633-651 (1987).
8. U. Kastens and W.M. Waite, Modularity and reusability in attribute grammars, *Acta Informatica* **31**(7) pp. 601-627 (1994).
9. T. Katayama, Translation of attribute grammars into procedures, *ACM Trans. Programming Languages and Systems* **6**(3) pp. 345-369 (July 1984).
10. D.E. Knuth, Semantics of context-free languages, *Mathematical System Theory* **2**(2) pp. 127-145 (June 1968). Correction. *ibid.* 5, 1 (March 1971), 95-96.
11. R. op den Akker and E. Sluiman, Storage allocation for attribute evaluators using stacks and queues, *Proceedings of the International Summer School SAGA*, (Prague, Czechoslovakia, June 1991), *Lecture Notes in Computer Science* **545** pp. 234-255 Springer-Verlag, (1991).
12. J. Paakki, Attribute grammar paradigms—A high-level methodology in language implementation, *ACM Computing Surveys* **27**(2) pp. 196-255 (June 1995).
13. W.M. Waite, The Eli 4.1 distribution, 1998, Dept. Computer Sciences, Colorado Univ., Boulder, CO (). Available from *http://www.cs.colorado.edu/~eliuser*
14. W.M. Waite, A complete specification of a simple compiler, CU-CS-638-93, Computer Science Dept., Univ. of Colorado at Boulder, Boulder, CO (January 1993).
15. W. Yang, *A classification of non-circular attribute grammars*, Computer and Information Science Dept., National Chiao-Tung Univ., Hsinchu, Taiwan (1997).