Formal Grammar for Java

Conference Paper · January 1999

DOI: 10.1007/3-540-48737-9_1 · Source: DBLP

Jim Alves-Foss
University of Idaho

Formal Grammar for Java

Jim Alves-Foss and Deborah Frincke

Center for Secure and Dependable Software, Department of Computer Science,
University of Idaho, Moscow ID 83844-1010, USA

1 Introduction

This chapter presents an attribute grammar for the Java programming language (v. 1.1). This grammar is derived from the LALR grammar presented in the *Java Language Specification* (JLS) [1]. The purpose of this grammar is to formally specify not only the syntactic structure of Java programs, but also their static semantics. Specifically, in this chapter we try to formally capture all aspects of the language that would result in compile-time errors. These errors include, but are not limited to:

- Type checking for assignment statements, ensuring that the type of the right-hand side of the statement is assignment compatible with the left hand side.
- Type checking expression operands, ensuring that they are of compatible types.
- Type checking method parameters, ensuring that they are the correct type and number.
- Checking for duplicate variable and method names.
- Checking for undefined variables.

We do not actually capture all errors, but a sufficient body of them to demonstrate the approach we are using. We have left comments within the syntax for portions where we believe addition semantic checks are needed, as an exercise to the reader. The grammar is written using a BNF-like notation of the form of productions:

$$
\begin{array}{lll}
\langle\text{NonTerm}\rangle ::= & \textit{exp1} & \\
 & & \textit{semantic action 1} \\
 & \mid \textit{exp2} & \\
 & & \textit{semantic action 2}
\end{array}
$$

where the left hand side (LHS) non-terminal <NonTerm> can be defined in terms of either right hand side (RHS) expression *exp1* or *exp2*. Within the productions we use some abbreviations to shorten the specification. We define the following abbreviations: $exp^?$ to specify optional inclusion of the expression, exp^+ to specify one or more occurrences of the expression, exp^* to specify zero or more occurrences of the expression. On the lines immediately following the RHS expression are the semantic actions for that production. These actions involve propagation of the attributes, up and down the parse tree, and static correctness

Table 1. Language unit caller/callee relationship.

	Callees												
Callers	names & lits	pkgs	types	mods	inter decl	class decl	meth decl	field decl	var decl	inits	cnstr	blocks & stmts	exprs
names & lits	X												
pkgs	X	X					X	X					
types	X		X										
mods				X									
inter decl	X			X	X	X	X	X					
class decl	X			X	X	X	X	X			X		
meth decl	X		X	X					X			X	
field decl	X		X	X					X				
var decl													
inits										X		?	X
cnstr	X			X								X	?
blocks & stmts	X		X						X			X	X
exprs	X		X		X	X			X			X	X

checks (compile-time errors). Associated with every potential compiler-time error we have placed the semantic action **ERROR** which displays a string error message related to the compile time error.

1.1 Logical Units of the Grammar

The full grammar is broken down into several logical units, each consisting of a collection of productions that define non-terminals in the grammar. Table 1 depicts the hierarchical relationship between these units. A logical unit is said to *call* another logical unit if it uses a non-terminal of the other logical unit in the RHS of one of its productions. The called logical unit is the *callee*. Note that there are several self- and circular-references in this table. These logical units are defined as follows:

names and literals - these define the lowest level constructs in the Java language and provide abstract representations of the low-level syntax of the Java language.

packages - these define the overall structure of the Java source code files, package and import specifications.

types - these define the type definition facilities of Java, which includes primitive types, reference types, class types, array types and interface types.

modifiers - these define the modifiers of various Java constructs. Such modifiers include protection modes (e.g., public, private) and status (e.g., static, final).

interface declarations - these define the form and structure of interfaces specifications.

class declarations - these define the form and structure of class specifications.

method declarations - these define the form and structure of method specifications.

field declarations - these define the form and structure of class and interface field specifications.

variable declarations - these define the form and structure of local variable specifications.

initializers - these define the initialization expressions for variable (including array) initializations.

constructors - these define the constructor statements for classes.

blocks and statements - these define the instruction and scoping constructs of the Java language.

expressions - these define all expressions of the Java language.

1.2 Attributes

To specify the semantic aspects of the grammar, we define a set of attributes that are used during the traversal of the parse tree specified by the grammar. For simplicity sake, we define the attributes using Java field and method use notation (e.g., `non.in.env` defines the inherited environment from the non-terminal non). An attribute is considered *inherited* if it is passed down from the non-terminal (root of the subtree) and it is *synthesize* if is created by the right-hand side expression (child nodes). We assume that all inherited attributes are included as fields of the inherited object `in`, which is specified as a field of the non-terminal of the production. We assume that all synthesize attributes are included as fields of the synthesized object `out` which is specified as a field of the non-terminal of the right-hand side expression.

This section describes the attributes of the grammar. The use of these attributes by the logical units of the language is as depicted in Table 2.

context This defines the code type being executed, whether it is a static or normal method, etc. This attribute is only inherited. The methods of this attribute are:

- `addPackage(`*name*`)` which adds the specified package *name* to the current context.
- `addClass(`*name*`,` *mods*`)` which adds the specified class *name* with its correct modifiers *mods* to the current context.
- `addInterface(`*name*`,` *mods*`)` which adds the specified interface *name* with its correct modifiers *mods* to the current context.
- `addMethod(`*name*`,` *mods*`)` which adds the specified method *name* with its correct modifiers *mods* to the current context.
- `addSwitchExpr(`*type*`)` stores the type of the current switch expression.
- `switchExpr()` returns the type of the current switch expression.

Table 2. Use of attributes in logical units of the language

	context	env	vars	type	value	mods	ids
	Inher.	Both	Both	Synth.	Synth.	Synth.	Synth.
names & lits				S	S		
pkgs		SI					
types				S			
mods						S	
inter decl	CI	CI				U	
class decl	CI	SI				U	
meth decl	CI	SI				U	SU
field decl	CI	SI	UI	U		U	
var decl							SU
inits	I	I	I				
cnstr	CI	SI				U	
blocks & stmts	UI	SUI	SUI	SU			
exprs	UI	UI	UI	SU	U		

C = creates I = Inherits S = Synthesizes
U = used for static error check

- **`isInstanceMethod()`** which returns true if the current context is within an instance method.
- **`isClassMethod()`** which returns true if the current context is within a class method.
- **`isConstructor()`** which returns true if the current context is within a constructor method.
- **`className()`** which returns the string representation of the current class.
- **`getClass()`** which returns the reference of the current class.
- **`getSuper()`** which returns the reference of the *super* class of the current class.

env This defines the "environment" of the program, basically the definition of all types, class fields and class definitions accessible by the current code. This attribute is inherited by code, but synthesized by the declarations aspects of the code. For a truly correct environment, the compiler must first parse all relevant declarations to build the top-level environment. Then the compiler can use this information in the second pass to evaluate expressions and statements. Without these two passes, all information must be declared prior to its use. To compress the presentation of the grammar in this chapter, we have combined the two passes of the compiler into one presentation and have greatly simplified the operations of the first pass of the compiler. The method **`new()`** defined below activates the first pass of the compiler and returns its results for the second pass. The methods of the env attribute are:

- **`new(`*`CompUnit`*`)`** which runs the first pass of the compiler on the code, producing a top-level environment which is used for the second pass of the compiler. In this environment are definitions of all classes, their fields and methods, imported classes, and compiler defined environment information (e.g., classes defined in other files specified on the same command line to the compiler). This method, in effect, runs the attributed grammar by ignoring all error checks and returning the environment output by CompUnit.

- `typeCheck(`*type*$_1$`, `*type*$_2$`)` which returns true if *type*$_2$ is of the type specified by *type*$_1$.
- `lookupFieldType(`*ref*`, `*id*`)` which returns the type of the field *id* from the reference *ref*.
- `lookupFieldValue(`*ref*`, `*id*`)` which returns the value of the field *id* from the reference *ref* if that field is final and was initialized with a constant expression, otherwise it returns *undef*.
- `isDefined(`*name*`)` which returns true if *name* is defined in the current environment.
- `idCheck(`*PrimaryType*`, `*IdType*`)` which returns true if *IdType* is unambiguous and acceptable for *PrimaryType*.
- `isLabel(`*name*`)` which returns true if the specified *name* is a current statement label.
- `addLabel(`*name*`)` which stores *name* as a named label in the current environment.

vars This defines the set of local variable declarations and their types. This attribute is typically only inherited, the exception being the local variable declaration statement which modifies this attribute synthesizing a new one.

type This attribute is only synthesized to perform the necessary type checking. It is synthesized by variable declarations and expressions. The methods of this attribute are:

- `insert(item)` which adds `item` to the list
- `equals(`*type*`)` which returns true if the argument type is the same as the current attributes type. This is used by the typeCheck method of env.
- `promotableTo(`*type*`)` which returns true if the current attribute type is promotable to the argument type.
- `inc()` which takes the current array type, increments the number of dimensions and returns the new array type. If the current type is not an array type, this method creates a one-dimensional array of the current type. Note that in this method we do not keep track of the actual size of each dimension (that is a run-time check.)
- `inc(`*num*`)` which takes the current array type, increments the number of dimensions by *num* and returns the new array type. If the current type is not an array type, this method creates a *num*-dimensional array of the current type. Note that in this method we do not keep track of the actual size of each dimension (that is a run-time check.)

value This attribute is synthesized from the low-level syntax of the language and is used to return the actual value associated with language literals, specifically identifier names and numeric, boolean, string, and character literals and the null constant. The methods of this attribute are:

- `defined()` returns true if the value is not `undefined`.
- `XX(`*value*`)` [for XX one of LT, GT, GE, LE, EQ] returns true if the value compares correctly with the parameter *value* (e.g., the value is less than the parameter for operation LT), and false otherwise.
- `bitXX(`*value*`)` returns the numeric result of performing the specified bitwise operation (bitAND, bitOR or bitXOR) on the numeric value and the numeric parameter *value*.
- `bitNOT()` returns the numeric result of performing the bitwise complement operation on the numeric value.
- `XX(`*value*`)` [for XX one of AND, OR or XOR] returns the boolean result of performing the specified logical operation (AND, OR or XOR) on the boolean value and the boolean parameter *value*.
- `NOT()` returns the boolean result of performing the logical complement operation boolean value.
- `XX(`*value*`)` [for XX one of LS, RSS, RSZ] returns the numeric result of performing the specified shift operation (<<, >, or >>>) on the numeric value and the numeric parameter *value*.

ids This attribute is only synthesized by variable declarations and is a list of declared variable ids.

mods This is the list of modifiers for classes, methods, fields and interfaces. The methods of this attribute are:

- `exclusive(`*list*`)` which returns true if the attribute only contains modifiers specified in *list*.
- `contains`*mod*`)` which returns true if the attribute contains the modifiers specified by *mod*.
- `insert(`*mod*`)` which adds `mod` to the list of modifiers.

The following methods are part of the output attribute of a term. They are part of a specific output attribute, since they utilize results of more than one attribute.

- `assignableTo(`*type*`)` returns true if the current expression can be converted to the specified *type* by assignment conversion.
- `isExpression(`*name*`)` returns true if the parameter *name* refers to a local variable or a field accessible in the current context.
- `getType(`*name*`)` returns the type of the parameter *name* within the current context (or undef if the type is unresolvable).
- `getValue(`*name*`)` returns the value of the parameter *name* within the current context if name refers to a final variable who's initializer was a constant expression, otherwise it returns undef.

In addition, the following auxiliary functions are used in this grammar

- `binaryNumericConversion(t1, t2)` which returns the resultant type after applying binary numeric conversion [1] to the two argument types `t1` and `t2`.

- `unaryNumericConversion(type)` which returns the resultant type after applying unary numeric conversion [1] to the argument type.
- `mkArrayType(`*`type`*`)` which returns the type equivalent to an array of the parameter *type*.
- `unmkArrayType(`*`type`*`)` which returns the type equivalent to a single element of the array specified by the parameter *type*.

2 The Grammar

In this section we present the full attributed grammar, for each of the logical units of the language defined above. A brief discussion of the attributes of each logical unit is provided.

2.1 Names and Literals

The following grammar specifies the syntax of names and literals in the Java language. Specific formatting details of these are not presented here, but rather are assumed to be those defined in the Java Language Specification [1]. The name entity in this specification returns a string representation of the name that is used by the higher level production to determine the appropriate type. The resulting name/type is returned in the type attribute. Literals, on the other hand return the appropriate literal type in the type attribute. Integer literals also return a value in the value attribute that can be evaluated in the assignment statement. This permits a direct assignment of a *small* integer to shorts, chars and bytes.

<Name> ::=
    <SimpleName>
      Name.out := SimpleName.out
  | <QualifiedName>
      Name.out := QualifiedName.out

<SimpleName>::=
    <Id>
      SimpleName.out.type := Id.out.value

<QualifiedName>::=
    <Name> `.` <Id>
      QualifiedName.type := Name.out.type+“.”+Id.out.value

<Literal> ::=
    <IntLit>
      Literal.out.type := int
      Literal.out.value := IntLit.out.value
  | <FloatLit>
      Literal.out.type := float

Literal.out.value := FloatLit.out.value
| <BoolLit>
Literal.out.type := bool
Literal.out.value := BoolLit.out.value
| <CharLit>
Literal.out.type := char
Literal.out.value := CharLit.out.value
| <StringLit>
Literal.out.type := java.lang.String
Literal.out.value := StringLit.out.value
| <NullLit>
Literal.out.type := null

2.2 Packages

The following grammar defines the high-level file syntax of Java programs. Specifically this aspect of the grammar is responsible for defining package membership, class imports and the top-level class and interface specifications. It is important to remember that all of the type-checking performed within the method bodies is performed only after all of these top-level definitions are parsed in the first pass. All the attributes at this level are just passed up and down the parse tree with the only changes being made are: the name of the current package is placed into the context (if no package is defined, the current package is the default package) and class definition imports are added to the environment.

Note that in this specification, there are some optional non-terminals on the RHS of the productions. The question arises as to how the attribute grammar handles the synthesized attributes of non-selected optional non-terminals. In this case, we adopt the convention that all synthesized-only attributes of an non-selected optional non-terminal are null, and that all inherited and synthesized attributes take on the value of the inherited attribute.

<Goal> ::=
<CompUnit>
CompUnit.in.env := env.new(<CompUnit>)
CompUnit.in.context := new context()

<CompUnit> ::=
<PackageDecl>$^{?}$ <ImportDeclList>$^{?}$ <TypeDeclList>$^{?}$
PackageDecl.in := CompUnit.in
ImportDeclList.in.context :=
CompUnit.in.context.addPackage(PackageDecl.out.type)
ImportDeclList.in.env := PackageDecl.out.env
TypeDeclList.in.context := ImportDeclList.in.context
TypeDeclList.in.env := ImportDeclList.out.env
CompUnit.out.env := TypeDeclList.out.env

<PackageDecl> ::=
package <Name> **;**
PackageDecl.out.type := Name.out.type

<ImportDeclList> ::=
<ImportDecl>
ImportDecl.in := ImportDeclList.in
ImportDeclList.out.env := ImportDecl.out.env
| <ImportDeclList$_1$> <ImportDecl>
ImportDeclList$_1$.in := ImportDeclList.in
ImportDecl.in.context := ImportDeclList.in.context
ImportDecl.in.env := ImportDeclList$_1$.out.env
ImportDeclList.out.env := ImportDecl.out.env

<TypeDeclList> ::=
<TypeDecl>
TypeDecl.in := TypeDeclList.in
TypeDeclList.out.env := TypeDecl.out.env
| <TypeDeclList> <TypeDecl>
TypeDeclList$_1$.in := TypeDeclList.in
TypeDecl.in.context := TypeDeclList.in.context
TypeDecl.in.env := TypeDeclList$_1$.out.env
TypeDeclList.out := TypeDecl.out.env

<ImportDecl> ::=
<SingleTypeImportDecl>
SingleTypeImportDecl.in := ImportDecl.in
ImportDecl.out.env := SingleTypeImportDecl.out.env
| <TypeImportOnDemandDecl>
TypeImportOnDemandDecl.in := ImportDecl.in
ImportDecl.out.env := TypeImportOnDemandDecl.out.env

<SingleTypeImportDecl> ::=
import <Name> **;**
SingleTypeImportDecl.out.env :=
SingleTypeImportDecl.in.env.import(Name.out.type)

<TypeImportOnDemandDecl> ::=
import <Name> **. * ;**
SingleTypeImportDecl.out.env :=
SingleTypeImportDecl.in.env.importOnDemand(Name.out.type)

<TypeDecl> ::=
;
TypeDecl.out.env := TypeDecl.in.env
| <ClassDecl>
ClassDecl.in := TypeDecl.in
TypeDecl.out.env := ClassDecl.out.env
| <InterfaceDecl>
InterfaceDecl.in := TypeDecl.in

TypeDecl.out.env := InterfaceDecl.out.env

2.3 Types

The following grammar presents the syntax of type definitions in Java. These productions simply pass back up the generated type of the term. If a reference type is expected, a compile-time check is made to ensure that the reference is defined, otherwise an error occurs. The same is true of array types.

<Type> ::=
<PrimType>
Type.out.type := PrimType.out.type
| <RefType>
Type.out.type := RefType.out.type

<PrimType> ::=
<NumType>
PrimType.out.type := NumType.out.type
| **boolean**
PrimType.out.type := boolean

<NumType> ::=
<IntType>
NumType.out.type := IntType.out.type
| <FloatType>
NumType.out.type := FloatType.out.type

<IntType> ::=
byte
IntType.out.type := byte
| **short**
IntType.out.type := short
| **int**
IntType.out.type := int
| **long**
IntType.out.type := long
| **char**
IntType.out.type := char

<FloatType> ::=
float
FloatType.out.type := float
| **double**
FloatType.out.type := double

<RefType> ::=
<ClassInterfaceType>

RefType.out.type := ClassInterfaceType.out.type
| <ArrayType>
RefType.out.type := ArrayType.out.type

<ClassInterfaceType>
<Name>
ClassInterfaceType.out.type := Name.out.type
if not(ClassInterfaceType.in.env.isDefined(Name.out.type))
ERROR ("Undefined Name" + Name.out.type)
ClassInterfaceType.out.type := null

<ClassType> ::=
<ClassInterfaceType>
ClassType.out.type := ClassInterfaceType.out.type

<InterfaceType> ::=
<ClassInterfaceType>
InterfaceType.out.type := ClassInterfaceType.out.type

<ArrayType> ::=
<PrimType> [ ]
ArrayType.out.type := mkArrayType(PrimType.out.type)
| <Name> [ ]
ArrayType.out.type :=
mkArrayType(ArrayType.out.env.lookupType(Name.out.value))
| <ArrayType> [ ]
ArrayType.out.type := mkArrayType(ArrayType.out.type)
**** Type check these **

2.4 Modifiers

The following grammar presents the syntax of modifiers, which return the *mods* attribute as a list of defined modifiers. These modifiers are used for classes, fields and methods in a Java file. It was decided to include all modifiers in a single grammatical structure here, and to perform restriction checking at a higher level; such as the illegal modification of an interface declaration with the **volatile** modifier. This structure does check for illegal duplicate modifiers, a condition that is not permitted in any use of modifiers in the Java language.

<Modifiers> ::=
<Modifier>
Modifiers.out.mods := Modifer.out.mods
| <Modifiers> <Modifier>
if Modifiers_1.out.mods.contains(Modifer.out.value)
ERROR ("The modifiers should contain only one instance of"+
Modifier.out.value)

Modifiers.out.mods := Modifiers_1.out.mods
else
Modifiers.out.mods := Modifiers.out.mods.insert(Modifier_1.out.value)
endif

<Modifier> ::=
public
Modifer.out.value := public
| **private**
Modifer.out.value := private
| **protected**
Modifer.out.value := protected
| **static**
Modifer.out.value := static
| **abstract**
Modifer.out.value := abstract
| **final**
Modifer.out.value := final
| **native**
Modifer.out.value := native
| **synchronized**
Modifer.out.value := synchronized
| **transient**
Modifer.out.value := transient
| **volatile**
Modifer.out.value := volatile

2.5 Interface Declarations

The following grammar presents the syntax for interface declarations.

<InterfaceDecl> ::=
<Modifiers>$^?$ <UnmodInterfaceDecl>
**** Modifiers abstract or public

<UnmodInterfaceDecl> ::=
interface <Id> <Extends>$^?$ <InterfaceBody>

<Extends>::=
extends <InterfaceType>
| <Extends> **,** <InterfaceType>

<InterfaceBody> ::=
{ <InterfaceMemberDeclList>$^?$ **}**

<InterfaceMemberDeclList> ::=
<InterfaceMemberDecl>

| <InterfaceMemberDeclList> <InterfaceMemberDecl>

<InterfaceMemberDecl> ::=
<ClassDecl>
| <InterfaceDecl> | <ConstDecl>
| <AbsMethodDecl>

<ConstDecl> ::=
<FieldDecl>
**** Public, static and/or final. Field declaration in body of interface is all 3

<AbsMethodDecl> ::=
<MethodHdr> **;**

2.6 Class Declarations

The following grammar presents class declarations.

<ClassDecl> ::=
<Modifiers> <UnmodClassDecl>
if not(Modifiers.out.mods.exclusive([public, abstract, final]))
ERROR "Classes may only be public, abstract and/or final")
endif
if not(Modifiers.out.mods.contains(abstract) and
Modifiers.out.mods.contains(final))
ERROR ("Classes can not be both abstract and final")
endif
UnmodClassDecl.in.context :=
ClassDecl.context.addClassMods(Modifiers.out.mods)
UnmodClassDecl.in.env := ClassDecl.in.env

<UnmodClassDecl> ::=
class <Id> <Super>$^?$ <Interfaces>$^?$ <ClassBody>
let con = UnmodClassDecl.in.context.addClassName(Id.out.value) in
let con1 = con.addSuper(Super.out.type)in
let con2 = con1.addInterfaces(Interfaces.out.type) in
ClassBody.in.context := con2
ClassBody.in.env := UnmodClassDecl.in.env

<Super> ::=
extends <ClassType>
Super.out.type := ClassType.out.type

<Interfaces> ::=
implements <InterfaceTypeList>
Interfaces.out.type := InterfaceTypeList.out.type

<InterfaceTypeList> ::=
<InterfaceType>
InterfaceTypeList.out.type := InterfaceType.out.type
| <InterfaceTypeList>$_1$ **,** <InterfaceType>

InterfacesTypeList.out.type :=
InterfaceTypeList.out.type.insert(InterfaceType$_1$.out.type)

<ClassBody> ::=
{ <ClassBodyDeclList>$^?$ **}** ClassBodyDeclList.in := ClassBody.in
ClassBody.out := ClassBodyDeclList.out

<ClassBodyDeclList> ::=
<ClassBodyDecl>
ClassBodyDecl.in := ClassBodyDeclList.in
ClassBodyDeclList.out := ClassBodyDecl.out
| <ClassBodyDeclList$_1$> <ClassBodyDecl>
ClassBodyDeclList$_1$.in := ClassBodyDeclList.in
ClassBodyDecl.in := ClassBodyDeclList$_1$.out
ClassBodyDeclList.out := ClassBodyDecl.out

<ClassBodyDecl> ::=
<ClassDecl>
ClassDec.in := ClassBodyDecl.in
ClassBudyDecl.out := ClassDecl.out
**** Nested classes may be static, abstract, final, public, protected, or private **
| <InterfaceDecl>
ClassDec.in := ClassBodyDecl.in
ClassBudyDecl.out := ClassDecl.out
| <ClassMemberDecl>
ClassMemberDecl.in := ClassBodyDecl.in
ClassBudyDecl.out := ClassMemberDecl.out
| <StaticInit>
StaticInit.in := ClassBodyDecl.in
ClassBudyDecl.out := StaticInit.out
| <ConstrDecl>
ConstrDecl.in := ClassBodyDecl.in
ClassBudyDecl.out := ConstrDecl.out

<ClassMemberDecl> ::=
<FieldDecl>
FieldDecl.in := ClassMemberDecl.in
ClassMemberDecl.out := FieldDecl.out
| <MethodDecl> MethodDecl.in := ClassMemberDecl.in
ClassMemberDecl.out := MethodDecl.out

2.7 Method Declarations

The following grammar presents the syntax for class method declarations.

<MethodDecl> ::=
<MethodHdr> <MethodBody>

<MethodHdr> ::=
<Modifiers>$^?$ <Type> <MethodDef> <Throws>$^?$
| <Modifiers>$^?$ **void** <MethodDef> <Throws>$^?$

<MethodDef> ::=
<Id> **(** <FormalParmList>$^?$ **)**
| <MethodDef> [ ]

<FormalParmList> ::=
<FormalParam>
| <FormalParmList> **,** <FormalParam>

<FormalParam> ::=
<Modifier> <Type> <VarDeclId>
**** Modifier may be final **

<Throws> ::=
throws <ClassTypeList>

<ClassTypeList> ::=
<ClassType>
| <ClassTypeList> **,** <ClassType>

<MethodBody> ::=
;
| <Block>

2.8 Field and Variable Declarations

The following grammar presents the syntax for class field declarations, and variable declarations.

<FieldDecl> ::=
<Modifiers>$^?$ <Type> <VarDecl> **;**
**** Modifiers one of (public, protected, private) final, static, transient, volatile

<VarDeclList> ::=
<VarDecl>

| <VarDeclList> **,** <VarDecl>

<VarDecl> ::=
<VarDeclId>
| <VarDeclId> **=** <VarInit>
*** Need Declared before used check here for Field inits***

<VarDeclId> ::=
<Id>
| <VarDeclId> [ ]

2.9 Initializers

The following grammar presents the syntax for variable and array initializers.

<StaticInit> ::=
static <Block>

<VarInits> ::=
<VarInit>
| <VarInits> **,** <VarInit>
*** Need Declared before used check here for Field inits***

<ArrayInit> ::=
{ <VarInits>$^?$,$^?$ **}**

<VarInit> ::=
<Expr>
| <ArrayInit>

myline

2.10 Constructor Declarations

The following grammar presents the syntax for constructors.

<ConstrDecl> ::=
<Modifiers>$^?$ <ConstrDef> <Throws>$^?$ <ConstrBody>
**** Modifiers one of of public, provate, protected

<ConstrDef> ::=
<SimpleName> **(** <FormalParmList>$^?$ **)**

<ConstrBody> ::=
{ <ExplConstrInv>$^?$ <BlockStmtList>$^?$ **}**

<ExplConstrInv> ::=
this **(** <ArgList>$^?$ **)** **;**
| **super** **(** <ArgList>$^?$ **)** **;**

2.11 Blocks and Statements

The following grammar presents the syntax for statements and blocks in the Java language. The pertinent attributes of blocks and statements are the environment (*env*) and local variable (*vars*) attributes.

<Block> ::=
{ <BlockStmtList>$^?$ **}**
BlockStmtList.in.context := Block.in.context
BlockStmtList.in.env := Block.in.env
BlockStmtList.in.vars := BlockStmtList.in.vars.newBlock()
Block.out.vars := Block.in.vars
Block.out.env := Block.in.env

<BlockStmtList> ::=
<BlockStmt>
BlockStmt.in := BlockStmtList.in
BlockStmtList.out := BlockStmt.out
| <BlockStmtList$_1$> <BlockStmt>
BlockStmtList$_1$.in := BlockStmtList.in
BlockStmt.in.context := BlockStmtList.in.context
BlockStmt.in.vars := BlockStmtList$_1$.out.vars
BlockStmt.in.end := BlockStmtList$_1$.out.env
BlockStmtList.out := BlockStmt.out

<BlockStmt> ::=
<LocalVarDeclStmt>
LocalVarDeclStmt.in := BlockStmt.in
BlockStmt.out := LocalVarDeclStmt.out
| <Statement>
Statement.in := BlockStmt.in
BlockStmt.out := Statement.out
| <UnmodClassDecl>
UnmodClassDecl.in := BlockStmt.in
BlockStmt.out := UnmodClassDecl.out

<LocalVarDeclStmt> ::=
<LocalVarDecl> **;**
LocalVarDecl.in := LocalVarDeclStmt.in
LocalVarDeclStmt.out := LocalVarDecl.out

<LocalVarDecl> ::=
<Type> <VarDeclList>

```
    Type.in := LocalVarDecl.in
    VarDeclList.in := LocalVarDecl.in
    LocalVarDecl.out.vars :=
      LocalVarDecl.in.vars.insert(Type.out.type, VarDeclList.out.ids)
    if DeclConflict(Type.out.type, VarDeclList.out.ids, LocalVarDecl.in.vars)
      ERROR ("Illegal Local Variable Declaration")
    endif
    LocalVarDecl.out.env := LocalVarDecl.in.env

<Statement> ::=
   <StmtNoTrailing>
    StmtNoTrailing.in := Statement.in
 | <LabeledStmt>
    LabeledStmt.in := Statement.in
 | <IfStmt>
    IfStmt.in := Statement.in
 | <IfElseStmt>
    IfElseStmt.in := Statement.in
 | <WhileStmt>
    WhileStmt.in := Statement.in
 | <ForStmt>
    ForStmt.in := Statement.in

<StmtNoShortIf> ::=
   <StmtNoTrailing>
    StmtNoTrailing.in := StmtNoShortIf.in
 | <LabeledStmtNoShortIf>
    LabeledStmtNoShortIf.in := StmtNoShortIf.in
 | <IfElseStmtNoShortIf>
    IfElseStmtNoShortIf.in := StmtNoShortIf.in
 | <WhileStmtNoShortIf>
    WhileStmtNoShortIf.in := StmtNoShortIf.in
 | <ForStmtNoShortIf>
    ForStmtNoShortIf.in := StmtNoShortIf.in

<StmtNoTrailing>
   <Block>
    Block.in := StmtNoTrailing.in
 | <EmptyStmt>
    EmptyStmt.in := StmtNoTrailing.in
 | <ExprStmt>
    ExprStmt.in := StmtNoTrailing.in
 | <SwitchStmt>
    SwitchStmt.in := StmtNoTrailing.in
 | <DoStmt>
    DoStmt.in := StmtNoTrailing.in
 | <BreakStmt>
    BreakStmt.in := StmtNoTrailing.in
 | <ContStmt>
    ContStmt.in := StmtNoTrailing.in
```

| <RetStmt>
RetStmt.in := StmtNoTrailing.in
| <SynchStmt>
SynchStmt.in := StmtNoTrailing.in
| <ThrowStmt>
ThrowStmt.in := StmtNoTrailing.in
| <TryStmt>
TryStmt.in := StmtNoTrailing.in

<EmptyStmt> ::=
;

<LabeledStmt> ::=
<Id> **:** <Statement>
Statement.in.context := LabeledStmt.in.context
Statement.in.vars := LabeledStmt.in.vars
if not(LabeledStmt.in.env.isLabel(Id.out.value))
ERROR("Label "+Id.out.value+" already in use.")
Statement.in.env := LabeledStmt.in.env
else
Statement.in.env := LabeledStmt.in.env.addLabel(Id.out.value)
endif

<LabeledStmtNoShortIf> ::=
<Id> **:** <StmtNoShortIf>
StmtNoShortIf.in.context := LabeledStmtNoShortIf.in.context
StmtNoShortIf.in.vars := LabeledStmtNoShortIf.in.vars
if not(LabeledStmtNoShortIf.in.env.isLabel(Id.out.value))
ERROR("Label "+Id.out.value+" already in use.")
StmtNoShortIf.in.env := LabeledStmtNoShortIf.in.env
else
StmtNoShortIf.in.env := LabeledStmtNoShortIf.in.env.addLabel(Id.out.value)
endif

<ExprStmt> ::=
<Assign>
Assign.in := ExprStmt.in
| <PreIncExpr>
PreIncExpr.in := ExprStmt.in
| <PreDecExpr>
PreDecExpr.in := ExprStmt.in
| <PostIncExpr>
PostIncExpr.in := ExprStmt.in
| <PostDecExpr>
PostDecExpr.in := ExprStmt.in
| <MethodInv>
MethodInv.in := ExprStmt.in
| <ClassInstCreationExpr>
ClassInstCreationExpr.in := ExprStmt.in

<IfStmt> ::=
if (<Expr> **)** <Statement>
Expr.in := IfStmt.in
Statement.in := IfStmt.in
if not (Expr.out.type.equals(boolean))
ERROR("Condition of if statement must be boolean")
endif

<IfElseStmt> ::=
if (<Expr> **)** <StmtNoShortIf> **else** <Statement>
Expr.in := IfElseStmt.in
Statement.in := IfElseStmt.in
StmtNoShortIf.in := IfElseStmt.in
if not (Expr.out.type.equals(boolean))
ERROR("Condition of if statement must be boolean")
endif

<IfElseStmtNoShortIf> ::=
if (<Expr> **)** <StmtNoShortIf> **else** <StmtNoShortIf>
Expr.in := IfElseStmtNoShortIf.in
Statement.in := IfElseStmtNoShortIf.in
StmtNoShortIf.in := IfElseStmtNoShortIf.in
if not (Expr.out.type.equals(boolean))
ERROR("Condition of if statement must be boolean")
endif

<SwitchStmt> ::=
switch (<Expr> **)** <SwitchBlock>
Expr.in := SwitchStmt.in
SwitchBlock.in.env := SwitchStmt.in.env
if not(Expr.out.type.equals(integral))
ERROR("Switch statement expression must be integral")
SwitchBlock.in.context := SwitchStmt.in.context.addSwitchExpt(int)
else
SwitchBlock.in := SwitchStmt.in.context.addSwitchExpfr(Expr.out.type)
endif

<SwitchBlock> ::=
{ <SwitchBlockStmtList>$^?$ <SwitchLabelList>$^?$ **}**
SwitchBlockStmtList.in := SwitchBlock.in
SwitchLabelList.in := SwitchBlock.in

<SwitchBlockStmtList> ::=
<SwitchBlockStmt>
SwitchBlockStmt.in := SwitchBlockStmtList.in
| <SwitchBlockStmtList$_1$> <SwitchBlockStmt>
SwitchBlockStmtList$_1$.in := SwitchBlockStmtList.in
SwitchBlockStmt.in := SwitchBlockStmtList.in

<SwitchBlockStmt> ::=
<SwitchLabelList> <BlockStmtList>
SwitchLabelList.in := SwitchBlockStmt.in
BlockStmtList.in := SwitchBlockStmt.in

<SwitchLabelList> ::=
<SwitchLabel>
SwitchLabel.in := SwitchLabelList.in
| <SwitchLabelList$_1$> <SwitchLabel>
SwitchLabelList$_1$.in := SwitchLabelList.in
SwitchLabel.in := SwitchLabelList.in

<SwitchLabel> ::=
case <ConstExpr>
ConstExpr.in := SwitchLabel.in
if not(ConstExpr.out.assignableTo(SwitchLabel.in.context.switchExpr()))
ERROR("Case label must be compatible with switch expression type.")
endif
| **default :**

<WhileStmt> ::=
while (<Expr> **)** <Statement>
Expr.in := WhileStmt.in
Statement.in := WhileStmt.in
if not(Expr.out.type.equals(boolean))
ERROR("While statement expression must be boolean")
endif

<WhileStmtNoShortIf> ::=
while (<Expr> **)** <StmtNoShortIf>
Expr.in := WhileStmt.in
StmtNoShortIf.in := WhileStmt.in
if not(Expr.out.type.equals(boolean))
ERROR("While statement expression must be boolean")
endif

<DoStmt> ::=
do <Statement> **while (** <Expr> **)**
Expr.in := DoStmt.in
Statement.in := DoStmt.in
if not(Expr.out.type.equals(boolean))
ERROR("Do statement expression must be boolean")
endif

<ForStmt> ::=
for (<ForInit>$^?$ **;** <Expr>$^?$ **;** <ForUpdate>$^?$ **)** <Statement>
ForInit.in := ForStmt.in
Expr.in.context := ForStmt.in.context
Expr.in.env := ForStmt.in.env

Expr.in.vars := ForInit.out.vars
ForUpdate.in.context := ForStmt.in.context
ForUpdate.in.env := ForStmt.in.env
ForUpdate.in.vars := ForInit.out.vars
Statement.in.context := ForStmt.in.context
Statement.in.env := ForStmt.in.env
Statement.in.vars := ForInit.out.vars

<ForStmtNoShortIf> ::=
for (<ForInit>$^?$ **;** <Expr>$^?$ **;** <ForUpdate>$^?$ **)** <StmtNoShortIf>
ForInit.in := ForStmt.in
Expr.in.context := ForStmt.in.context
Expr.in.env := ForStmt.in.env
Expr.in.vars := ForInit.out.vars
ForUpdate.in.context := ForStmt.in.context
ForUpdate.in.env := ForStmt.in.env
ForUpdate.in.vars := ForInit.out.vars
StmtNoShortIf.in.context := ForStmt.in.context
StmtNoShortIf.in.env := ForStmt.in.env
StmtNoShortIf.in.vars := ForInit.out.vars

<ForInit> ::=
<ExprStmtList>
ExprStmtList.in := ForInit.in
ForInit.in.vars := ExprStmtList.out.vars
| <LocalVarDecl>
LocalVarDecl.in := ForInit.in
ForInit.out.vars := LocalVarDecl.out.vars

<ForUpdate> ::=
<ExprStmtList>
ExprStmtList.in := ForUpdate.in

<ExprStmtList> ::=
<ExprStmt>
ExprStmt.in := ExprStmtList.in
| <ExprStmtList$_1$> **,** <ExprStmt>
ExprStmt.in := ExprStmtList.in
ExprStmtList$_1$.in := ExprStmtList.in

<BreakStmt> ::=
break <Id>$^?$ **;**
Id.in := BreakStmt.in
if not(BreakStmt.in.env.isLabel(Id.out.value))
ERROR("Undefined Label "+Id.out.value+" in Break statement")
endif

<ContStmt> ::=
continue <Id>$^?$ **;**

```
      Id.in := ContStmt.in
      if not(ContStmt.in.env.isLabel(Id.out.value))
         ERROR("Undefined Label "+Id.out.value+" in Continue statement")
      endif

<RetStmt> ::=
    return <Expr>? ;
      Expr.in := RetStmt.in
      if not(Expr.out.assignableTo(RetStmt.in.context.returnType())
         ERROR(Expr.out.type+ " not compatible with return type")
      endif

<ThrowStmt> ::=
    throw <Expr> ;
      Expr.in := RetStmt.in
      if not(RetStmt.in.context.throws(Expr.out.type))
         ERROR("Statment does not throw exception: "+Expr.out.type)
      endif

<SynchStmt> ::=
    synchronized ( <Expr> ) <Block>
      Expr.in := SynchStmt.in
      Block.in := SynchStmt.in
      if not(Expr.out.type.equals(ref))
         ERROR("Argument of synchronized statement must be reference type")
      endif

<TryStmt> ::=
    try <Block> <Catches>
      Block.in := TryStmt.in
      Catches.in := TryStmt.in
  | try <Block> <Catches>? <Finally>
      Block.in := TryStmt.in
      Catches.in := TryStmt.in
      Finally.in := TryStmt.in

<Catches> ::=
    <CatchClause>
      CatchClause.in := Catches.in
  | <Catches_1> <CatchClause>
      Catches_1.in := Catches.in
      CatchClause.in := Catches.in

<CatchClause> ::=
    catch ( <FormalParam> ) <Block>
      FormalParam.in := CatchClause.in
      Block.in := CatchClause.in
      if not(FormalParam.out.type.promotableTo(Throwable))
         ERROR("Catch clause parameter must be of type throwable.")
```

endif

<Finally> ::=
finally <Block>
Block.in := Finally.in

2.12 Expressions

The following grammar presents the syntax for expressions in the Java language. For expressions, the pertinent output (synthesized) attributes are *types* and *values*, the input (inherited) attributes are *context*, *environment* and *variables*.

The JLS specifies the types of expressions, dependent on the types of the subexpressions and the form of the expression. However, in the case where there is a compile-time error (e.g., a type mismatch error), the JLS does not specify either a default or calculated return type. This enables compiler writers to make their own interpretation of the return type, resulting in incompatible behavior during compilation when compile-time errors are present. In this specification we have chosen return types that either follow the convention of the Sun JDK, or result in a relatively intuitive result. For select expressions, the return type and value are both *undef* an undefined value. For type checking methods, an undefined type is compatible with all types. For these errors, we have made the following decisions:

- For the conditional expression <CondExpr>, experimentation with the Sun JDK indicates that the resulting type is the type of the right most expression <CondExpr_1>. We followed that precedent in this specification.
- For the overloaded operators |, **&**, and $\wedge$, which can be used for either boolean operations or for numeric bit-wise operations, we follow the convention that the expected and return types are boolean.
- for shift, addition, subtraction and multiplication operations, the default return type is int.
- for the or, and, and xor operations the default value is boolean (even if the programmer intended on a bitwise operation).

<ConstExpr> ::=
<Expr>
Expr.in := ConstExpr.in
ConstExpr.out := Expr.out

<Expr> ::=
<AssignExpr>
AssignExpr.in := Expr.in
Expr.out := AssignExpr.out

```
<AssignExpr> ::=
    <Assign>
      Assign.in := AssignExpr.in
      AssignExpr.out := Assign.out
  | <CondExpr>
      CondExpr.in := AssgnExpr.in
      AssignExpr.out := CondExpr.out

<Assign> ::=
    <LHS> <AssignOp> <AssignExpr>
      LHS.in := Assign.in
      AssignExpr.in := Assign.in
      if (AssignOp.out.value == EQ)
         if not(AssignExpr.out.assignableTo(LHS.out.type))
            ERROR("Assignment conversion error, cannot convert" +
               AssignExpr.out.type + "to" + LHS.out.type)
         endif
      else if (AssignOp.out.value == NumEQ)
         if not(LHS.out.type.equals(numeric) and
               AssignExpr.out.type.equals(numeric))
            ERROR("Operands of "+AssignOp+" must be numeric")
         endif
      else // AssignOp.out.value == BitEQ
         if not(LHS.out.type.equals(numeric) and
               AssignExpr.out.type.equals(numeric)) or
               not(LHS.out.type.equals(boolean) and
               AssignExpr.out.type.equals(boolean))
            ERROR ("Operands of "+AssignOp+" must be
               both either boolean or numeric")
         endif
      endif
      Assign.out.type := LHS.out.type
      Assign.out.value := undef

<LHS> ::=
    <Name>
      Name.in := LHS.in
      LHS.out := Name.out
  | <FieldAccess>
      Name.in := FieldAccess.in
      FieldAccess.out := Name.out
  | <ArrayAccess>
      Name.in := ArrayAccess.in
      ArrayAccess.out := Name.out

<AssignOp> ::=
    =
      AssignOp.out.value := EQ
  | * =
      AssignOp.out.value := NumEQ
```

```
  | /=
      AssignOp.out.value := NumEQ
  | %=
      AssignOp.out.value := NumEQ
  | +=
      AssignOp.out.value := NumEQ
  | -=
      AssignOp.out.value := NumEQ
  | <<=
      AssignOp.out.value := NumEQ
  | >>=
      AssignOp.out.value := NumEQ
  | >>>=
      AssignOp.out.value := NumEQ
  | &=
      AssignOp.out.value := BitEQ
  | ^=
      AssignOp.out.value := BitEQ
  | |=
      AssignOp.out.value := BitEQ

<CondExpr>::=
    <CondOrExpr>
      CondOrExpr.in := CondExpr.in
      CondExpr.out := CondOrExpr.out
  | <CondOrExpr> ? <Expr> : <CondExpr_1>
      CondOrExpr.in := CondExpr.in
      Expr.in := CondExpr.in
      CondExpr_1.in := CondExpr.in
      // Check type of conditional expression and evaluate
      if not(CondOrExpr.out.type.equals(boolean))
          ERROR ("Expression on LHS of ? must be boolean")
      else if (CondOrExpr.out.value == undef)
              CondOrExpr.out.value := undef
          else
          if (CondOrExpr.out.value == true)
              CondExpr.out.value := Expr.out.value
          else
              CondExpr.out.value := CondExpr_1.out.value
          endif
      endif
      // Handle case if both right-hand subexpressions are boolean
      if (Expr.out.type.equals(boolean) and
              CondExpr_1.out.type.equals(boolean))
          CondExpr.out.type := boolean
      // Handle case if both right-hand subexpressions are numeric
      else if (Expr.out.type.equals(numeric) and
              CondExpr_1.out.type.equals(numeric))
          if (Expr.out.type.equals(CondExpr_1.out.type))
              CondExpr.out.type := Expr.out.type
```

```
        else if ((Expr.out.type.equals(byte) and
                CondExpr_1.out.type.equals(short)) or
                (Expr.out.type.equals(short) and
                CondExpr_1.out.type.equals(byte))
            CondExpr.out.type := short
        else if (Expr.out.type.inList([short;char;byte]) and
                CondExpr_1.out.assignableTo(Expr.out.type))
            ConExpr.out.type := Expr.out.type
        else if (CondExpr_1.out.type.inList([short;char;byte]) and
                Expr.out.assignableTo(CondExpr_1.out.type))
            CondExpr.out.type := CondExpr_1.out.type
        else
            CondExpr.out.type :=
                binaryNumericConversion(Expr.out.type, CondExpr_1.out.type)
        endif
    // Handle case if both right-hand subexpresions are references
    else if (Expr.out.type.equals(ref) and
            CondExpr_1.out.type.equals(ref))
        if (Expr.out.type.promotableTo(CondExpr_1.out.type))
            CondExpr.out.type := CondExpr_1.out.type
        else if (CondExpr_1.out.type.promotableTo(Expr.out.type))
            CondExpr.out.type := Expr.out.type
        else
            ERROR("Can't convert "+Expr.out.type+"to "+CondExpr_1.out.type)
            CondExpr.out.type := Expr.out.type
        endif
    else
        ERROR("Can't convert "+Expr.out.type+"to "+CondExpr_1.out.type)
        CondExpr.out.type := Expr.out.type
    endif

<CondOrExpr>::=
   <CondAndExpr>
    CondAndExpr.in := CondOrExpr.in
    CondOrExpr.out := CondAndExpr.out
 | <CondOrExpr_1> || <CondAndExpr>
    CondOrExpr_1.in := CondOrExpr.in
    CondAndExpr.in := CondOrExpr.in
    if not(CondOrExpr_1.out.type.equals(boolean) and
            CondAndExpr.out.type.equals(boolean))
        ERROR("Both arguments to || must be boolean")
        CondOrExpr.out.value := undef
    else if not (CondOrExpr_1.out.value.defined())
        CondOrExpr.out.value := undef
    else if (CondOrExpr_1.out.value == true)
        CondOrExpr.out.value := true
    else if not (CondAndExpr.out.value.defined())
        CondOrExpr.out.value := undef
    else if (CondAndExpr.out.value == true)
        CondOrExpr.out.value := true
```

```
    else
        CondOrExpr.out.value := false
    endif
    CondOrExpr.out.type := boolean

<CondAndExpr>::=
    <IncOrExpr>
    IncOrExpr.in := CondAndExpr.in
    CondAndExpr.out := IncOrExpr.out
  | <CondAndExpr_1> && <IncOrExpr>
    CondAndExpr_1.in := CondAndExpr.in
    IncOrExpr.in := CondAndExpr.in
    if not(CondAndExpr_1.out.type.equals(boolean) and
            IncOrExpr.out.type.equals(boolean))
        ERROR("Both arguments to && must be boolean")
        CondAndExpr.out.value := undef
    else if not (CondAndExpr_1.out.value.defined())
        CondAndExpr.out.value := undef
    else if (CondAndExpr_1.out.value == false)
        CondAndExpr.out.value := false
    else if not (IncOrExpr.out.value.defined())
        CondAndExpr.out.value := undef
    else if (IncOrExpr.out.value == true)
        CondAndExpr.out.value := true
    else
        CondAndExpr.out.value := false
    endif
    CondAndExpr.out.type := boolean

<IncOrExpr>::=
    <XORExpr>
    XORExpr.in := IncOrExpr.in
    IncOrExpr.out := XORExpr.out
  | <IncOrExpr_1> | <XORExpr>
    IncOrExpr_1.in := IncOrExpr.in
    XORExpr.in := IncOrExpr.in
    if (IncOrExpr_1.out.type.equals(integral) and
            XORExpr.out.type.equals(integral))
        IncOrExpr.out.type :=
                binaryNumericConversion(IncOrExpr_1.out.type,XORExpr.out.type)
        IncOrExpr.out.value := IncOrExpr_1.out.value.bitOR(XORExpr.out.value)
    else if not(IncOrExpr_1.out.type.equals(boolean) and
            XORExpr.out.type.equals(boolean))
        ERROR("Both arguments to | must be boolean or numeric")
        IncOrExpr.out.value := undef
        IncOrExpr.out.type := boolean
    else
        IncOrExpr.out.value := IncOrExpr_1.out.value.OR(XORExpr.out.type)
        IncOrExpr.out.type := boolean
    endif
```

<XORExpr>::=
<AndExpr>
AndExpr.in := XORExpr.in
XORExpr.out := AndExpr.out
| <XORExpr_1> ^ <AndExpr>
XORExpr_1.in := XORExpr.in
AndExpr.in := XORExpr.in
if (XOROrExpr_1.out.type.equals(integral) and
AndExpr.out.type.equals(integral))
XORExpr.out.type :=
binaryNumericConversion(XORExpr_1.out.type,AndExpr.out.type)
XORExpr.out.value := XORExpr_1.out.value.bitXOR(AndExpr.out.value)
else if not(XORExpr_1.out.type.equals(boolean) and
AndExpr.out.type.equals(boolean))
ERROR("Both arguments to $\wedge$ must be boolean or numeric")
XORExpr.out.value := undef
XORExpr.out.type := boolean
else
XORExpr.out.value := XORExpr_1.out.value.XOR(AndExpr.out.type)
XORExpr.out.type := boolean
endif

<AndExpr>::=
<EqualExpr>
EqualExpr.in := AndExpr.in
AndExpr.out := EqualExpr.out
| <AndExpr_1> **&** <EqualExpr>
AndExpr_1.in := AndExpr.in
EqualExpr.in := AndExpr.in
if (AndExpr_1.out.type.equals(integral) and
EqualExpr.out.type.equals(integral))
AndExpr.out.type :=
binaryNumericConversion(AndExpr_1.out.type,EqualExpr.out.type)
AndExpr.out.value := AndExpr_1.out.value.bitAND(EqualExpr.out.value)
else if not(AndExpr_1.out.type.equals(boolean) and
EqualExpr.out.type.equals(boolean))
ERROR("Both arguments to **&** must be boolean or numeric")
AndExpr.out.value := undef
AndExpr.out.type := boolean
else
AndExpr.out.value := AndExpr_1.out.value.AND(EqualExpr.out.type)
AndExpr.out.type := boolean
endif

<EqualExpr> ::=
<RelatExpr>
RelatExpr.in := EqualExpr.in
EqualExpr.out := RelatExpr.out
| <EqualExpr_1> **==** <RelatExpr>

```
      EqualExpr.out.type := boolean
      if not(EqualsExpr1.out.type.compatibleWith(RelatExpr.out.type))
         ERROR("Operands of == must be compatible types")
         EqualExpr.out.value := undef
      else
         EqualExpr.out.value := EqualExpr1.out.value.EQ(RelatExpr.out.value)
      endif
   | <EqualExpr1> != <RelatExpr>
      EqualExpr.out.type := boolean
      if not(EqualsExpr1.out.type.compatibleWith(RelatExpr.out.type))
         ERROR("Operands of != must be compatible types")
         EqualExpr.out.value := undef
      else
         EqualExpr.out.value := not(EqualExpr1.out.value.EQ(RelatExpr.out.value))
      endif

<RelatExpr>::=
    <ShiftExpr>
      ShiftExpr.in := RelatExpr.in
      RelatExpr.out := ShiftExpr.out
   | <RelatExpr1> < <ShiftExpr>
      RelatExpr1.in := RelatExpr.in
      ShiftExpr.in := RelatExpr.in
      if (TypeCheck(numeric, RelatExpr1.out.type) and
            TypeCheck(numeric, ShiftExpr.out.type))
         RelatExpr.out.type := boolean
         RelatExpr.out.value := RelatExpr1.out.value.LT(ShiftExpr.out.value)
      else
         ERROR ("Both arguments to < must be numeric type")
         RelatExpr.out.type := boolean
         RelatExpr.out.value := undef
      endif
   | <RelatExpr1> > <ShiftExpr>
      RelatExpr1.in := RelatExpr.in
      ShiftExpr.in := RelatExpr.in
      if (TypeCheck(numeric, RelatExpr1.out.type) and
            TypeCheck(numeric, ShiftExpr.out.type))
         RelatExpr.out.type := boolean
         RelatExpr.out.value := RelatExpr1.out.value.GT(ShiftExpr.out.value)
      else
         ERROR ("Both arguments to > must be numeric type")
         RelatExpr.out.type := boolean
         RelatExpr.out.value := undef
      endif
   | <RelatExpr1> <= <ShiftExpr>
      RelatExpr1.in := RelatExpr.in
      ShiftExpr.in := RelatExpr.in
      if (TypeCheck(numeric, RelatExpr1.out.type) and
            TypeCheck(numeric, ShiftExpr.out.type))
         RelatExpr.out.type := boolean
```

```
        RelatExpr.out.value := RelatExpr_1.out.value.LE(ShiftExpr.out.value)
    else
        ERROR ("Both arguments to <= must be numeric type")
        RelatExpr.out.type := boolean
        RelatExpr.out.value := undef
    endif
  | <RelatExpr_1> >= <ShiftExpr>
    RelatExpr_1.in := RelatExpr.in
    ShiftExpr.in := RelatExpr.in
    if (TypeCheck(numeric, RelatExpr_1.out.type) and
            TypeCheck(numeric, ShiftExpr.out.type))
        RelatExpr.out.type := boolean
        RelatExpr.out.value := RelatExpr_1.out.value.GE(ShiftExpr.out.value)
    else
        ERROR ("Both arguments to >= must be numeric type")
        RelatExpr.out.type := boolean
        RelatExpr.out.value := undef
    endif
  | <RelatExpr_1> instanceof <RefType>
    RelatExpr_1.in := RelatExpr.in
    RefType.in := RelatExpr.in
    if (typeCheck(refOrNull, RelatExpr_1.out.type) and
        typeCheck(ref, ShiftExpr.out.type) and
            RefType.out.type.promotableTo(RelatExpr_1.out.type)
        RelatExpr.out.type := boolean
        RelatExpr.out.type := undef
    else
        ERROR ("Impossible for "+RelatExpr_1.out.type+
            " to be instance of "+RefType.out.type)
        RelatExpr.out.type := boolean
        RelatExpr.out.value := undef
    endif

<ShiftExpr> ::=
    <AddExpr>
    AddExpr.in := ShiftExpr.in
    ShiftExpr.out := AddExpr.out
  | <ShiftExpr_1> << <AddExpr>
    ShiftExpr_1.in := ShiftExpr.in
    AddExpr.in := ShiftExpr.in
    if (TypeCheck(integral, ShiftExpr_1.out.type) and
            TypeCheck(integral, AddExpr.out.type))
        Shift.out.type := promote(ShiftExpr_1.out.type, AddExpr.out.type)
    else ERROR ("Both arguments to << must be integral type")
        ShiftExpr.out.type := int
        Shift.out.value := Shift_1.out.value.LS(AddExpr.out.value)
    endif
  | <ShiftExpr_1> >> <AddExpr>
    ShiftExpr_1.in := ShiftExpr.in
    AddExpr.in := ShiftExpr.in
```

```
    if (TypeCheck(integral, ShiftExpr_1.out.type) and
            TypeCheck(integral, AddExpr.out.type))
        Shift.out.type := promote(ShiftExpr_1.out.type, AddExpr.out.type)
        Shift.out.value := Shift_1.out.value.RSS(AddExpr.out.value)
    else ERROR ("Both arguments to >> must be integral type")
        ShiftExpr.out.type := int
        ShiftExpr.out.value := undef
    endif
  | <ShiftExpr_1> >>> <AddExpr>
    ShiftExpr_1.in := ShiftExpr.in
    AddExpr.in := ShiftExpr.in
    if (TypeCheck(integral, ShiftExpr_1.out.type) and
            TypeCheck(integral, AddExpr.out.type))
        Shift.out.type := promote(ShiftExpr_1.out.type, AddExpr.out.type)
        Shift.out.value := Shift_1.out.value.RSZ(AddExpr.out.value)
        ShiftExpr.out.type := int
        ShiftExt.out.value := undef
    else ERROR ("Both arguments to >>> must be integral type")
        ShiftExpr.out.type := int
        ShiftExt.out.value := undef
    endif

<AddExpr> ::=
   <MultExpr>
    MultExpr.in := AddExpr.in
    AddExpr.out := MultExpr.out
  | <AddExpr_1> + <MultExpr>
    AddExpr_1.in := AddExpr.in
    MultExpr.in := AddExpr.in
    if (TypeCheck(string, AddExpr_1.out.type) or
            TypeCheck(string, MultExpr.out.type))
        AddExpr.out.type := string
        AddExpr.out.value := AddExpr_1.out.value.string+( MultExpr.out.value)
    else if (TypeCheck(numeric, AddExpr_1.out.type) and
            TypeCheck(numeric, MultExpr.out.type))
        AddExpr.out.type := promote(AddExpr_1.out.type, MultExpr.out.type)
        AddExpr.out.value := AddExpr_1.out.value + MultExpr.out.value
    else ERROR ("Both arguments to + must be numeric, or one a string")
        AddExpr.out.type := int
        AddExpr.out.value := undef
    endif
  | <AddExpr> - <MultExpr>
    AddExpr_1.in := AddExpr.in
    MultExpr.in := AddExpr.in
    if (TypeCheck(numeric, AddExpr_1.out.type) and
            TypeCheck(numeric, MultExpr.out.type))
        AddExpr.out.type := promote(AddExpr_1.out.type, MultExpr.out.type)
        AddExpr.out.value := AddExpr_1.out.value - MultExpr.out.value
    else ERROR ("Both arguments to + must be NumType")
        AddExpr.out.type := int
```

AddExpr.out.value := undef
endif

<MultExpr> ::=
<UnaryExpr>
UnaryExpr.in := MultExpr.in
MultExpr.out := UnaryExpr.out
| <MultExpr$_1$> ***** <UnaryExpr>
MultExpr$_1$.in := MultExpr.in
UnaryExpr.in := MultExpr.in
if (typeCheck(numeric, MultExpr$_1$.out.type) and
typeCheck(numeric,UnaryExpr.out.type))
MultExpr.out.type := promote(MultExpr$_1$.out.type, UnaryExpr.out.type)
AddExpr.out.value := MultExpr$_1$.out.value - AddExpr.out.value
else **ERROR** (“Both arguments to ***** must be numeric”)
MultExpr.out.type := int
MultExpr.out.value := undef
endif
| <MultExpr$_1$> **/** <UnaryExpr>
MultExpr$_1$.in := MultExpr.in
UnaryExpr.in := MultExpr.in
if (typeCheck(numeric, MultExpr$_1$.out.type) and
typeCheck(numeric,UnaryExpr.out.type))
MultExpr.out.type := promote(MultExpr$_1$.out.type, UnaryExpr.out.type)
MultExpr.out.value := MultExpr$_1$.out.value - AddExpr.out.value
else **ERROR** (“Both arguments to **/** must be numeric”)
MultExpr.out.type := int
MultExpr.out.value := undef
endif
| <MultExpr$_1$> **%** <UnaryExpr>
MultExpr$_1$.in := MultExpr.in
UnaryExpr.in := MultExpr.in
if (typeCheck(numeric, MultExpr$_1$.out.type) and
typeCheck(numeric,UnaryExpr.out.type))
MultExpr.out.type := promote(MultExpr$_1$.out.type, UnaryExpr.out.type)
MultExpr.out.value := MultExpr$_1$.out.value **%** AddExpr.out.value
else **ERROR** (“Both arguments to **%** must be numeric”)
MultExpr.out.type := int
MultExpr.out.value := undef
endif

<UnaryExprNotPlusMinus> ::=
<CastExpr>
CastExpr.in := UnaryExprNotPlusMinus.in
UnaryExprNotPlusMinus.out := CastExpr.out
| <PostExpr>
PostExpr.in := UnaryExprNotPlusMinus.in
UnaryExprNotPlusMinus.out := PostExpr.out
| ~ <UnaryExpr>
UnaryExpr.in := UnaryExprNotPlusMinus.in

```
      if not(UnaryExpr.out.type.equals(integral))
         ERROR(" Argument of ~ must be primitive Integral Type")
         UnaryExprNotPlusMinus.out.type := int
         UnaryExprNotPlusMinus.out.value := undef
      else
         UnaryExprNotPlusMinus.out.type := UnaryExpr.out.type
         UnaryExprNotPlusMinus.out.value := UnaryExpr.out.value.bitNOT()
      endif
   | ! <UnaryExpr>
      if not(UnaryExpr.out.type.equals(integral))
         ERROR(" Argument of ! must be boolean")
         UnaryExprNotPlusMinus.out.type := boolean
         UnaryExprNotPlusMinus.out.value := undef
      else
         UnaryExprNotPlusMinus.out.type := UnaryExpr.out.type
         UnaryExprNotPlusMinus.out.value := UnaryExpr.out.value.NOT()
      endif

<CastExpr>::=
******* Needs to check types for validity of cast ***********
     ( <PrimType> <Dims>? ) <UnaryExpr>
      PrimType.in := CastExpr.in
      Dims.in := CastExpr.in
      UnaryExpr.in := CastExpr.in
      CastExpr.out.type := array(PrimType.out.type,Dims.out.type)
   | ( <Expr> ) <UnaryExprNotPlusMinus>
      Expr.in := CastExpr.in
      UnaryExprNotPlusMinus.in := CastExpr.in
      CastExpr.out.type := Expr.out.type
   | ( <Name Dims> ) <UnaryExprNotPlusMinus>
      Name.in := CastExpr.in
      Dims.in := CastExpr.in
      UnaryExprNotPlusMinus.in := CastExpr.in
      CastExpr.out.type := array(Name.out.type, Dims.out.type)

<PostExpr> ::=
     <Primary>
      Primary.in := PostExpr.in
      PostExpr.out := Primary.out
   | <Name>
      Name.in := PostExpr.in
      if (Name.out.isExpression())
         PostExpr.out.type := Name.out.getType()
         PostExpr.out.value := Name.out.getValue()
      else
         ERROR("Undefined variable "+ Name.out.value)
         PostExpr.out.type := undef
         PostExpr.out.value := undef
      endif
```

```
  | <PostIncExpr>
     PostIncExpr.in := PostExpr.in
     PostExpr.out := PostIncExpr.out
  | <PostDecExpr>
     PostDecExpr.in := PostExpr.in
     PostExpr.out := PostDecExpr.out

<PostIncExpr> ::=
    <PostExpr> ++
     PostExpr.in := PostIncExpr.in
     if not(PostExpr.out.type.equals(numeric))
         ERROR("Postfix Expr must be a variable of numeric type")
         PostIncExpr.out.type := int
         PostIncExpr.out.value := undef
     else
         PostIncExpr.out.type := PostExpr.out.type
         if (PostExpr.out.value.defined())
             PostIncExpr.out.value := PostExpr.out.value + 1
         else
             PostIncExpr.out.value := undef
         endif
     endif

<PostDecExpr> ::=
    <PostExpr> - -
     PostExpr.in := PostDecExpr.in
     if not(PostExpr.out.type.equals(numeric))
         ERROR("Postfix Expr must be a variable of numeric type")
         PostDecExpr.out.type := int
         PostDecExpr.out.value := undef
     else
         PostDecExpr.out.type := PostExpr.out.type
         if (PostExpr.out.value.defined())
             PostDecExpr.out.value := PostExpr.out.value - 1
         else
             PostDecExpr.out.value := undef
         endif
     endif

<UnaryExpr>::=
    <PreIncExpr>
     PreIncExpr.in := UnaryExpr.in
     UnaryExpr.out := PreIncExpr.out
  | <PreDecExpr>
     PreDecExpr.in := UnaryExpr.in
     UnaryExpr.out := PreDecExpr.out
  | + <UnaryExpr1>
     UnaryExpr1.in := UnaryExpr.in
     if not(UnaryExpr1.out.type.equals(numeric))
         ERROR("Argument of unary + must be numeric')
```

```
            UnaryExpr.out.type := int
            UnaryExpr.out.value := undef
        else
            UnaryExpr.out := UnaryExpr_1.out
        endif
    | - <UnaryExpr_1>
        UnaryExpr_1.in := UnaryExpr.in
        if not(UnaryExpr_1.out.type.equals(numeric))
            ERROR("Argument of unary - must be numeric')
            UnaryExpr.out.type := int
            UnaryExpr.out.value := undef
        else
            UnaryExpr.out.type := UnaryExpr_1.out.type
            if (UnaryExpr_1.out.value.defined())
                UnaryExpr.out.value := 0 - UnaryExpr_1.out.value
            else
                UnaryExpr.out.value := undef
            endif
        endif
    | <UnaryExprNotPlusMinus>
        UnaryExprNotPlusMinus.in := UnaryExpr.in
        UnaryExpr.in := UnaryExprNotPlusMinus.in

<PreIncExpr> ::=
     ++ <UnaryExpr>
        UnaryExpr.in := PreIncExpr.in
        PreIncExpr.out.value := undef
        if not(UnaryExpr.out.type.equals(numeric))
            ERROR("Preincrement expr must be a variable of numeric type")
            PreIncExpr.out.type := int
        else
            PreIncExpr.out.type := UnaryExpr.out.type
            endif
        endif

<PreDecExpr> ::=
     – <UnaryExpr>
        UnaryExpr.in := PreDecExpr.in
        PreDecExpr.out.value := undef
        if not(UnaryExpr.out.type.equals(numeric))
            ERROR("Predecrement expr must be a variable of numeric type")
            PreDecExpr.out.type := int
        else
            PreDecExpr.out.type := UnaryExpr.out.type
            endif
        endif

<Primary> ::=
     <PrimaryNoNewArray>
        PrimaryNoNewArray.in := Primary.in
```

```
    Primary.out := PrimaryNoNewArray.out
  | <ArrayCreationExpr>
    ArrayCreationExpression := Primary.in
    Primary.out := ArrayCreationExpression.out

<PrimaryNoNewArray> ::=
    <Literal>
    Literal.in := PrimaryNoNewArray.in
    PrimaryNoNewArray.out := Literal.out
  | this
    if not(PrimaryNoNewArray.context.isInstanceMethod() or
          PrimaryNoNewArray.context.isConstructor())
      ERROR("this permitted only in an instance method or constructor")
    endif
    PrimaryNoNewArray.out.value := undef
    PrimaryNoNewArray.out.type = Primary.NoNewArray.in.context.getClass()
  | ( <Expr> )
    Expr.in := PrimaryNoNewArray.in
    PrimaryNoNewArray.out := Expr.out
  | <ClassInstCreationExpr>
    ClassInstCreationExpr.in := PrimaryNoNewArray.in
    PrimaryNoNewArray.out := ClassInstCreationExpr.out
  | <FieldAcc>
    FieldAcc.in := PrimaryNoNewArray.in
    PrimaryNoNewArray.out := FieldAcc.out
  | <MethodInv>
    MethodInv.in := PrimaryNoNewArray.in
    PrimaryNoNewArray.out := MethodInv.out
  | <ArrayAccess>
    ArrayAccess.in := PrimaryNoNewArray.in
    PrimaryNoNewArray.out := ArrayAccess.out

<ArrayCreationExpr> ::=
    new <PrimType> <DimExprList> <Dims>?
    PrimType.in := ArrayCreationExpr.in
    DimExprList.in := ArrayCreationExpr.in
    Dims.in := ArrayCreationExpr.in
    ArrayCreationExpr.out.value := undef
    ArrayCreationExpr.out.type :=
      PrimType.out.type.inc(DimExprList.out.value + Dims.out.value)
  | new <ClassInterfaceType> <DimExprList> <Dims>?
    ClassInterfaceType.in := ArrayCreationExpr.in
    DimExprList.in := ArrayCreationExpr.in
    Dims.in := ArrayCreationExpr.in
    ArrayCreationExpr.out.value := undef
    ArrayCreationExpr.out.type :=
      PrimType.out.type.inc(DimExprList.out.value + Dims.out.value)

<ClassInstCreationExpr>::=
```

```
    new <ClassType> ( <ArgList>? ) <ClassBody>?
        ClassType.in := ClassInstCreationExpr.in
        ArgList.in := ClassInstCreationExpr.in
        ClassBody.in := ClassInstCreationExpr.in
        ClassInstCreationExpr.out.type := ClassType.out.type
        ClassInstCreationExpr.out.value := undef
********* Finish this to check argument types for constructor ***********
  | new <InterfaceType> () <ClassBody>
        InterfaceType.in := ClassInstCreationExpr.in
        ArgList.in := ClassInstCreationExpr.in
        ClassBody.in := ClassInstCreationExpr.in
        ClassInstCreationExpr.out.type := InterfaceType.out.type
        ClassInstCreationExpr.out.value := undef
********* Finish this to check types ***********

<FieldAcc> ::=
    <Primary> . <Id>
        Primary.in := FieldAcc.in
        Id.in := FieldAcc.in
        if not (FieldAcc.in.env.typeCheck(ref,Primary.out.type))
            ERROR(Primary.out.type + "must be a reference type")
            FieldAcc.out.type := null
        else if not(FieldAcc.in.env.idCheck(Primary.out.type,Id.out.type)))
            ERROR(Id.out.value + "must be non-ambiguous and accessible")
            FieldAcc.out.type := null
        else
            FieldAcc.out.type :=
                FieldAcc.in.env.lookupFieldType(Primary.out.type, Id.out.type)
            FieldAcc.out.value :=
                Field.in.env.lookupFieldValue(Primary.out.type, Id.out.type)
        endif
  | super . <Id>
        Id.in := FieldAcc.in
        FieldAcc.out := Id.out
        if FieldAcc.context.className() == "Java.lang.Object"
            Error("Term super not permitted in class Object")
        else if not(PrimaryNoNewArray.context.isInstanceMethod() or
                PrimaryNoNewArray.context.isConstructor())
            ERROR("super permitted only in an instance method or constructor")
        else
            FieldAcc.out.type := FieldAcc.in.env.lookupFieldType
                (FieldAcc.in.context.getSuper(), Id.out.type)
            FieldAcc.out.value := FieldAcc.in.env.lookupFieldValue
                (FieldAcc.in.context.getSuper(), Id.out.type)
        endif

<MethodInv> ::=
    <Name> ( <ArgList>? )
        *** 15.11.1 Type Name ID not interface
```

```
  | <Primary> . <Id> ( <ArgList>? )
      *** Id must be non-ambiguous and accessible
  | super . <Id> ( <ArgList>? )
      if FieldAcc.context.className() == "Java.lang.Object"
         Error("Term super not permitted in class Object")
         FieldAcc.out := FieldAcc.in
      else if not(PrimaryNoNewArray.context.isInstanceMethod() or
         PrimaryNoNewArray.context.isConstructor())
         ERROR("super permitted only in an instance method or constructor")
      else
         FieldAcc.out.type = FieldAcc.out.context.getSuper() + Id.out.type
      endif

<ArrayAccess> ::=
    <Name> [  <Expr> ]
      Name.in := ArrayAccess.in
      Expr.in := ArrayAccess.in
      ArrayAccess.out.value := undef
      if not(Expr.out.type.promotableTo(int))
         ERROR"Array indicies must be integers")
      endif
      if not(typeCheck(array, ArrayAccess.env.lookupType(Name.out.type)))
         ERROR(Name.value+"must be of array type")
         ArrayAccess.out.type := undef
      else
         ArrayAccess.out.type =
            unmkArrayType(ArrayAccess.env.lookupType(Name.out.type))
      endif
  | <PrimaryNoNewArray> [ <Expr> ]
      PrimaryNoNewArray.in := ArrayAccess.in
      Expr.in := ArrayAccess.in
      ArrayAccess.out.value := undef
      if not(Expr.out.type.promotableTo(int))
         ERROR("Array indicies must be integers")
      endif
      if not(typeCheck(array, PrimaryNoNewArray.type)))
         ERROR(Name.value+"must be of array type")
         ArrayAccess.out.type := undef
      else
         ArrayAccess.out.type =
            unmkArrayType(PrimaryNoNewArray.out.type)
      endif

<ArgList> ::=
    <Expr>
      Expr.in := ArgList.in
      ArgList.out := Expr.out
  | <ArgList_1> , <Expr>
      ArgList_1.in := ArgList.in
```

Expr.in := ArgList$_1$.out
ArgList.out := Expr.out

<DimExprList> ::=
<DimExpr>
DimExpr.in := DimExprList.in
DimExprList.out := DimExpr.out
| <DimExprList$_1$> <DimExpr>
DimExprList$_1$.in := DimExprList.in
DimExpr.in := DimExprList.in
DimExprList.out.type := undef
DimExprList.out.value := DimExprList$_1$.out.value + 1

<DimExpr> ::=
[<Expr>]
Expr.in := DimExpr.in
if not (typeCheck(integral, Expr.out.type))
ERROR ("Dimension declaration must be IntType")
endif
DimExpr.out := Expr.out
DimExpr.out.type := undef
DimExpr.out.value := 1

<Dims> ::=
[ ]
Dims.out := Dims.in
Dims.out.type := undef
Dims.out.value := 1
| <Dims$_1$> [ ]
Dims$_1$.in := Dims.in
Dims.out.value := Dims$_1$.out.value + 1
Dims.out.type := undef

References

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.