The Semantics of the Java Programming Language: Preliminary Version

Charles Wallace*
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122 USA
wallace@eecs.umich.edu

December 9, 1997

Abstract

A mathematical model of the Java programming language is constructed.

1 Introduction

We provide operational semantics for the Java programming language using the *Abstract State Machine (ASM)* methodology of [Gur95, ASM]. We use *Montages* ([KP97b]) to specify not only the runtime behavior of a Java program but also the actions performed at compile time. Following previous work in the area ([GH93, Wal95, KP97a], we present the specification of Java as a sequence of ASMs, each a refinement of its predecessor which provides new details of a particular aspect of the language. The goal is an orderly and understandable presentation that culminates in a complete specification of the language.

Amid the growing number of products and concepts bearing the name "Java", it is important to identify what this specification comprises and what it does not. We focus solely on the Java programming language (or simply *Java*, following standard usage), a *class-based*, *object-oriented* high-level programming language. Actions in a Java program have the form of operations performed on *objects*, instances of user-defined *classes* with named state variables (*fields*) and procedures (*methods*). The set of classes forms a hierarchy, with each class *inheriting* fields and methods from its superclasses. The programmer defines new methods and fields for a class at the time of its declaration.

100% Pure Java is an approach to program development that integrates the Java programming language with the *Java Platform*, a system-independent programming interface ([Jav]). The Java Platform consists of the set of *core classes* and the *Java Virtual Machine*. The core classes are members of the fixed set of classes included in the package `java.*`. The Virtual Machine is a specification of a mechanism for *interpreting* (decoding and executing) programs in a low-level language called *Java bytecode*. A high-level programming language using the Java Platform[1] has a *compiler* to translate its programs to bytecode. A *Java runtime system* is a software environment in which programs compiled for the Virtual Machine can run. In addition to implementing the Java Platform, a Java runtime system must have facilities for loading bytecode programs and managing memory, among other duties.

While Java is normally used in conjunction with the Java Platform, the language can be considered in isolation. A key feature of Java is its high level of abstraction; the details of the low-level representation are hidden from the programmer. This specification does not rely on the existence of a Java Platform beneath

*Partially supported by NSF grant CCR-95-04375 and ONR grant N00014-94-1-1182.

[1] Java, for instance. However, the Java Platform can be (and is) used by other programming languages (e.g. Ada ([App])).

the programming language. Where Java does make assumptions about the underlying system, we express these assumptions as abstractly as possible.

An important aspect of Java is its emphasis on compile-time safety checks. The language is designed to catch as many erroneous or malicious programs as possible before they execute. As a result, there are many subtle issues concerning compilation of Java programs. A mathematical model of the language, such as the one presented here, can present these issues in a clear and unambiguous way. Early work in the area (e.g. [GH93, Wal95]) considered only runtime semantics. All compile-time information was represented informally and was assumed to exist in the initial state of the model. More recently, the Montage approach has provided a convenient way to represent how this information is derived at compile time. We believe that the specification presented here can be useful to implementers of Java in clarifying complicated details about the language. Furthermore, as the specification is a mathematical object, it can be used to verify compile-time and runtime properties of the language.

In Section 2, we introduce terminology and concepts related to Java, ASMs and Montages. The first specification, in Section 3, concerns the creation of user-defined types or *classes*. The refinement in Section 4 provides the details of *control flow*. Section 5 specifies how *expressions* are evaluated. Section 6 deals with the invocation of *methods*. Section 7 explains the initialization of classes and class instances, and the finalization of class instances. Section 8 concerns the execution of *threads*. Finally, Section 9 deals with *locks* and *wait sets*.

Another effort to provide semantics for Java is the work of [BS97]. However, only an abstract of this paper is provided publicly. To the best of our knowledge, our paper is the first publicly available document providing both compile-time and runtime semantics for Java.

In this preliminary version, there are some aspects of the language which we do not cover. The specification covers only Java version 1.0, so the changes to the language in Java 1.1 do not appear. In addition, we do not handle volatile variables, or definite assignment of variables before use. We hope to have these language features included soon in our specification.

Acknowledgments

The following people richly deserve credit for their assistance in this work. Prof. Yuri Gurevich supervised this work and contributed some of the ideas. Philipp Kutter provided useful input on Montages. Matthias Anlauff developed the GemMex tool ([Gem]), which was used to produce the diagrams in this paper. Jim Huggins gave insightful comments on a draft of the paper.

2 Preliminaries

In this section, we provide brief introductions to ASMs and Montages. We use [Gur95, Gur97] as our references for ASMs. We use the definition of Montages in [KP97b], with a few extensions.

2.1 Abstract state machines

2.1.1 States and updates

A *vocabulary* is a finite collection of function names, each with a fixed arity. Relations are treated as Boolean-valued functions; we call them *relation functions*. Certain function names appear in every vocabulary: the *nullary* (0-ary) functions $true$, $false$ and $undef$, the binary equality function, and the standard Boolean operations.

A *state* S of an ASM M with vocabulary Υ consists of a nonempty set X, called the *superuniverse* of S, and an interpretation of each function name in Υ over X. The function names $true$, $false$ and $undef$ denote distinct elements of X. The Boolean operations are defined in the expected way over the interpretations of $true$ and $false$. the interpretation of $undef$ represents an undefined value and is used to represent partial functions. While every function f in the state is total, we may define a "domain" for f and set its value to

undef for every tuple outside its domain. A unary relation symbol U can be seen as representing a *universe*: the set of all elements for which $U(x)$ evaluates to *true*.

An ASM changes from state to state as it runs. An *update* is an atomic state change: a change in the interpretation of a single function name for a single tuple of arguments. We define an update as follows. A *location* of a state S is a pair $l = (f, \bar{x})$, where f is an r-ary function name in the vocabulary of S and $\bar{x}$ is an r-tuple of elements of (the superuniverse of) S. Then an update of S is a pair (l, y), where l is a location of S and y is an element of S.

At a given state, some elements of the superuniverse are inaccessible through functions of the vocabulary; we call the set of these elements the *reserve*. Elements of the reserve may be used later as the ASM requires more elements. An element is in the reserve if (1) every relational function returns $false$ when given the element as an argument; (2) every other function returns *undef* when given the element as an argument; (3) no function returns the element.

2.1.2 Terms and transition rules

Terms are defined as follows. We use $(t)_S$ to represent the interpretation of term t at state S.

- If $t = v$, where v is a variable name, then $(t)_S$ is the interpretation of v at S.
- If t is of the form $f(t_1 \ldots t_r)$, where f is an r-ary function name and $t_1 \ldots t_r$ are terms, then $(t)_S = (f)_S((t_1)_S \ldots (t_r)_S)$.
- If t is of the form (if g_0 then $t_0 \ldots$ elseif g_n then t_n), then $(t)_S = (t_i)_S$, where i is the minimum value for which $(g_i)_S = true$. If there is no such i, then $(t)_S = undef$.
- If t is of the form $\{t : v \in U : c\}$, where t is a term, v is a variable name, U is a universe name and c is a Boolean term, then $(t)_S$ is the set consisting of all $(t)_{S(v \to a)}$ for all elements a of S where $(U(v))_{S(v \to a)} = true$ and $(c)_{S(v \to a)} = true$. ($S(v \to a)$ is the state S extended to vocabulary $\Upsilon \cup \{v\}$, with v interpreted as a.)[2]
- If t is of the form $(\forall v : g)c$, where v is a variable name and g and c are Boolean terms, then $(t)_S$ is $true$ if forall elements a of S, either $(g)_{S(v \to a)} = false$ or $(c)_{S(v \to a)} = true$.
- If t is of the form $(\exists v : g)c$, where v is a variable name and g and c are Boolean terms, then $(t)_S$ is $true$ if for some element a of S, either $(g)_{S(v \to a)} = false$ or $(c)_{S(v \to a)} = true$.

In this paper, we use the following alternative notation. If f is a nullary function name, f abbreviates $f()$. If f is a unary function name, $x.f$ abbreviates $f(x)$, where x is a term. If f is an $(r+1)$-ary function name, $x.f(\bar{x})$ abbreviates $f(x, \bar{x})$, where x is a term and $\bar{x}$ is an r-tuple of terms. We also use $t.def?$ to abbreviate $t \neq undef$ and $t.undef?$ to abbreviate $t = undef$.

An ASM performs controlled state updates through *transition rules*. For a given state S, a rule gives rise to a set of updates, as follows.

- If R is a *skip instruction*, of the form **skip**, then the update set of R is the empty set.
- If R is an *update instruction*, of the form $f(t_1 \ldots t_n) := t_0$, where f is an r-ary function name and each t_i is a term, then the update set of R contains a single update (l, y), where $y = (t_0)_S$ and $l = (f, ((t_1)_S \ldots (t_r)_S))$. To execute R at S, set $f((t_1)_S \ldots (t_r)_S)$ to $(t_0)_S$.
- If R is a *block rule*, a sequence $R_1 \ldots R_n$ of transition rules, then the update set of R is the union of the update sets of each R_i. To execute R at S, execute all R_i simultaneously.

[2] This type of term first appeared in [BGS97].

- If R is a *conditional rule*, of the form **if** g_0 **then** $R_0 \ldots$ **elseif** g_n **then** R_n **endif**, where $g_1 \ldots g_n$ (the *guards*) are terms and $R_0 \ldots R_n$ are rules, then the update set of R is the update set of R_i, where i is the minimum value for which g_i evaluates to $true$. If all g_i evaluate to $false$, then the update set of R is empty. To execute R at S, execute R_i if it exists; otherwise do nothing.

- If R is an *import rule*, of the form **import** v R_0 **endimport**, where v is a variable name and R_0 is a rule, then the update set of R at S is the update set of R_0 at $S(v \rightarrow a)$, where a is a reserve element. To execute R at S, execute R_0 at $S(v \rightarrow a)$.

- If R is a *forall rule*, of the form **do-forall** $v : g$ R_0 **enddo**, then the update set of R is the union of the update sets of R_0 at all $S(v \rightarrow a)$, where a is any element of S for which g evaluates to $true$ at $S(v \rightarrow a)$. To execute R at S, execute R_0 at $S(v \rightarrow a)$ for all such a.

- If R is a *choice rule*, of the form **choose** $v : g$ R_0 **endchoose**, then the update set of R is the update set of R_0 at some $S(v \rightarrow a)$, where a is an element of S for which g evaluates to $true$ at $S(v \rightarrow a)$. To execute R at S, choose some such a and execute R_0 at $S(v \rightarrow a)$.

We use the following abbreviations for transition rules.

- **let** $v = t$ R_0 **endlet** abbreviates $R_0(v \rightarrow t)$, the result of substituting t for v in R_0 everywhere v is free.

- **if** g_0 **then** $R_0 \ldots$ **else** R_n **endif** abbreviates **if** g_0 **then** $R_0 \ldots$ **elseif** $true$ **then** R_n **endif**.

- **extend** U **with** v R_0 **endextend** abbreviates

  import v
  $\quad U(v) := true$
  $\quad R_0$
  endimport

- **choose among** $R_1 \ldots R_n$ abbreviates

  choose $v_0 : v_0 \in Bool \ldots$ **choose** $v_n : v_n \in Bool$
  $\quad$ **if** v_0 **then** $R_0 \ldots$ **else** R_{n+1} **endif**
  endchoose. . . **endchoose**

2.1.3 Runs

We are ready to describe how an ASM changes from state to state as it runs.

Let Υ be a vocabulary containing a universe $Agent$, a unary function mod, and a nullary function $Self$. A program Π of vocabulary Υ consists of a finite set of *modules*, each of which is a named transition rule over Υ. A *(global) state* of Π is a structure S of vocabulary $\Upsilon - \{Self\}$, where each module name is interpreted as a distinct element of S, and mod maps elements of $Agent$ to module names. If $Mod(\alpha) = M$, we say that α is an agent with program M.

For every agent α, $View_\alpha(S)$ (the *local state* of α) is the reduct of S to the functions appearing in $mod(\alpha)$, extended by interpreting the function name $Self$ as α. To fire α at state S, execute $mod(\alpha)$ at state $View_\alpha(S)$.

A *run* ρ of a program Π is a triple (M, A, σ), where

- M, the *moves* of ρ, is a poset where every set $\{\nu : \nu \leq \mu\}$ is finite. If M is totally ordered, we say that ρ is *sequential*.

- A is a function mapping agents to moves so that every nonempty set $\{\mu : A(\mu) = \alpha\}$ is linearly ordered. This condition asserts that every agent executes sequentially.

- σ is a function mapping finite initial segments of M to states of Π. $\sigma(X)$ is the state that results from performing all moves of X in an order according to the relation $\leq$. $\sigma(\emptyset)$ is the initial state S_0.
- (Coherence) If x is a maximal element in a finite initial segment X of M and $Y = X - \{x\}$, then $A(x)$ is an agent in $\sigma(Y)$, x is a move of $A(x)$ and $\sigma(X)$ is obtained from $\sigma(Y)$ by performing x at $\sigma(Y)$.

2.2 Montages

We give operational semantics for the Java programming language as an ASM which takes a Java program as input, *compiles* it (sets up the program's initial state and checks it for errors), and then executes it. The specification of the language is given as a collection of *montages*, each associated with a production rule of the syntax.[3]

A montage has four components: its production rule, an ASM transition rule for compilation (given partly in graphical format), a condition on the program's initial state, and a transition rule for program execution. Each montage provides a compile-time transition rule C and a runtime transition rule R. The ASM rules from all montages are included in the ASM J which gives the semantics of Java. Execution of J proceeds in two phases: construction and analysis of the program's initial state, followed by execution of the program.

The initial state of J contains the *compact derivation tree* of a program P. A compact derivation tree is derived from a parse tree by repeatedly collapsing each node n with a single child c to a single node, with the labels of both n and c and having only the children of c as children, until no such nodes remain. For every node label u, there is a universe U containing all nodes with that label. The initial state of J contains a universe of $Node$s in the derivation tree, sorted into the universes $Leaf$ (nodes without children) and $Nonleaf$ (nodes with children). *Selector functions* link a nonleaf with its children. Let m be a node labeled with u, and let n be a child of m labeled with v and generated by the production rule $u ::= \ldots v \ldots$. Then the function v maps m to n.[4]

A montage has the following general form:

[3] We use a grammar equivalent to the LALR(1) grammar of [GJS96], but with certain modifications to ease understanding.

[4] In our syntax for Java, a production rule has at most one occurrence of a given label on its right-hand side. Thus v selects a unique c.

PRODUCTION RULES
Main production rule Unit productions (optional) ⋮
COMPILE-TIME RULES
Control/Data flow graph (optional) Compile-time ASM rule (optional)
COMPILE-TIME CONDITION
condition ASM guard (optional) ⋮
RUNTIME RULES
Task Label: Runtime ASM rule (optional) ⋮

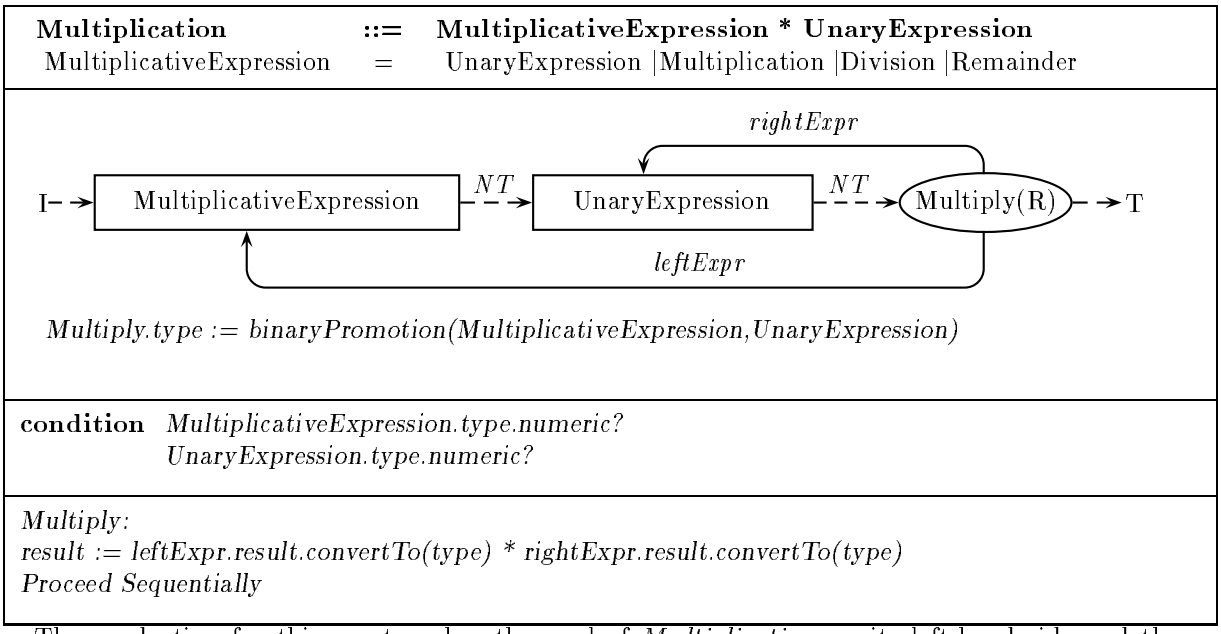
Compilation of the program consists of two passes, which visit the nodes of the tree in bottom-up, left-to-right order. Compile-time rules and conditions are characterized as first-pass or second-pass. The nullary function $curNode$ returns the currently visited node. (Terms of the form $curNode.f(\bar{x})$, where f is a selector function name and $\bar{x}$ is a sequence of terms, appear frequently in our montages and are usually abbreviated as $f(\bar{x})$.) The montage with production rule $u ::= v_1 \ldots v_n$ is selected when the visited node is in universe u and has descendants in universes $v_1 \ldots v_n$. Square brackets around a node on the right-hand side of a production signifies that the node is *optional* (the production for the node may be null). If n is an element of node universe u, in a production of the form $u ::= \ldots [v] \ldots$, the function v maps m either to an element n of the universe v, or to an element n of the universe $NoNode$.

A universe of *tasks* represents units of work in the program. Each agent running the program identifies its *current* task by the dynamic function $curTask$. The program has a *store* of data, represented by dynamic functions. At any stage of its execution, an agent fires transition rules which may alter the current task or the store.

During visitation of a node on a given pass, the conditions on the program's initial state are tested; if the result of any condition is $false$, the program is inconsistent and the evaluation ends. Otherwise, the compile-time rules for the currently visited node set up the compile-time information for the node. In the graphical portion of the rules, the nodes of the right-hand side of the production rule are represented as boxes and the tasks to create are represented as ovals. Control and data flow are represented as attributes of the tasks. The *elements* (nodes or tasks) containing the *initial task* (to which control passes from outside the node) and *terminal task* (from which control flows beyond the node) are designated in the graph by arrows labeled I and T, respectively. The functions $initialTask$ and $terminalTask$ map nodes to their initial and terminal tasks. A *result task* (from which data flows beyond the node) is designated in the graph by the label (R). The function $resultTask$ maps nodes to their result tasks. A dotted arrow between elements represents a control flow relationship between the terminal task of the source and the initial task of the destination. A solid arrow between elements represents a data flow relationship between the result task of the source and the result task of the destination. Any compile-time action not representable in terms of arrows between elements is given as an explicit ASM rule. Terms in compile-time rules may be abbreviated by omitting the

function name *resultTask* wherever its presence is obvious through context.

For a concrete example, consider the following montage, taken from Section 5.



The production for this montage has the nonleaf *Multiplication* on its left-hand side and the nonleaves *MultiplicativeExpression* and *UnaryExpression* on its right-hand side. Unit productions for *MultiplicativeExpression* are given below the main production rule. *MultiplicativeExpression* and *UnaryExpression* appear as boxes in the control/data flow graph, along with a task oval labeled *Multiply*. The task oval is labeled (R) and has an outgoing T-arrow, indicating that it is the terminal and expression-end task for the *Multiplication* production. *MultiplicativeExpression* has an incoming I-arrow which means that the initial task of the *MultiplicativeExpression* is also the initial task of the *Multiplication* production.

The *NT* arrow indicates that the terminal task of *MultiplicativeExpression* is to be mapped to the initial task of *UnaryExpression* by the *nextTask* function. The *leftExpr* and *rightExpr* arrows map the *Multiply* task to the expression-end tasks of *MultiplicativeExpression* and *UnaryExpression*. Additional compile-time rules are given below the graph. These set the *type* and *constant*? attributes for the *Multiply* task. Note that *curNode.Multiply.type* is abbreviated as *Multiply.type*.

The compile-time condition states that the types associated with the expression-end tasks of *MultiplicativeExpression* and *UnaryExpression* are both numeric. Note that the condition *curNode.MultiplicativeExpression.resultTask.type.numeric*? is abbreviated by omitting *resultTask*. Finally, the runtime rule for the task *Multiply* is given below the label *Multiply*. Note that *curTask.result* is abbreviated as *result*.

List productions, of the form $A = LIST(B)$ (or $A = LIST(B, s)$), produce lists of syntactic elements of the same type (delimited by the symbol s). For each application of such a production, the derivation tree has a list of B-elements with a common parent, called a *list node*. The function $ListNode[Nat] : Node$ maps a list node to each element, and $Node.Position : Nat$ returns each element's position in the list. The initial leaf of the list node is the initial leaf of the first element, and the terminal leaf is the terminal leaf of the last element. Between the terminal task of each list element and the initial task of its successor in the list, there is a control flow link *nextTask*.

Function and macro definitions

We assume that any node labeled *Identifier* has an associated *ID*, and any node labeled *Literal* has an associated *result* and *type*.

macro $N.memberOf?(L)$: $(\exists i)L[i] = N$

macro $N.firstInList?(L, C(N))$:
(Is node N the first member of list L for which condition C holds?)
$N.memberOf?(L)$ and $C(N)$ and not $(\exists p : C(p))p.position < N.position$

macro $N.lastInList?(L, C(N))$:
(Is node N the last member of list L for which condition C holds?)
$N.memberOf?(L)$ and $C(N)$ and not $(\exists p : C(p))N.position < p.position$

macro $P.next?(N, L, C(P))$:
(Is node P the next member of list L after node N for which condition C holds?)
$N.memberOf?(L)$ and $P.memberOf?(L)$ and $C(N)$ and $C(P)$ and $N.position < P.position$
and not $(\exists q : C(q)$ and $q.memberOf?(L))N.position < q.position < P.position$

macro For Each Member N Of List L $C(N)$
$(\forall N : N.memberOf?(L))C(N)$

macro For All Distinct Members M, N Of List L $C(M, N)$
For Each Member M Of List L
For Each Member N Of List L
$M \neq N \Rightarrow C(M, N)$

macro Process Each Node V In List L R:
(For each child c of the list node L, fire a copy of rule R with variable V set to c.)
do-forall i **with** $L[i].def?$
let $V = L[i]$
R

At points in the compilation phase, actions are performed in parallel on all the tasks contained within a certain node. A function $Node.contains?(Task) : Boolean$ determines whether the task is created during compilation of the node or one of its descendants. We add some rules to the compile-time block of each montage to establish the values of this function. For a montage whose production rule generates nonterminals $N_1 \ldots N_m$ and which generates tasks $t_1 \ldots t_n$, add the following:

$curNode.contains?(t_1) := true \ldots curNode.contains?(t_n) := true$
do-forall $t : t.isA?(Task) : N_1.contains?(t)$ or ... or $N_m.contains?(t)$
$curNode.contains?(t) := true$

3 Class and interface structure

Our first model focuses solely on the construction of a program's user-defined types (classes and interfaces). No details of the program's execution are considered, so the ASM has only a compilation phase. We specify how members are assigned to a class, and how the hierarchy of user-defined types is constructed.

Java has *primitive* and *reference* types and a *null* type. Primitive types are an inherent part of Java and cannot be modified by the programmer. Instances of primitive types do not change their states during program execution. The set of primitive types consists of the numeric types and the type Boolean. Numeric types consist of *floating-point* (Float and Double) and *integral* (Long, Int, Short, Char and Byte) types. The type Boolean has two instances, named True and False. Instances of the types Byte, Short, Int and Long are 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively. Instances of type Char

are 16-bit unsigned integers. Instances of Float and Double are 32-bit and 64-bit floating-point numbers, respectively.

Reference types are defined by the programmer. Reference types consist of *classes*, *interfaces* and *arrays*. Of these, only classes and arrays have instances. These instances, called *objects*, are created and may be destroyed (when no longer needed) during a program's execution. Associated with an object is a *class* which is the type which the object instantiates.[5] An object's state may be changed during execution, but its class remains constant.

Relations between types

Types are related to one another in the following ways. A class C may *extend* a class B, in which case B becomes the *parent* of C and C becomes a *subclass* of B and all classes of which B is a subclass. Each class has a unique parent (except the class Object, which has no parent) and is a subclass of its parent and its parent's subclasses. Every array type has exactly one parent: the class Object. An interface J may extend a number of interfaces $I_1 \ldots I_n$, in which case each I_i becomes a parent of J and J becomes a subclass of each I_i and all interfaces of which I_i is a subinterface. A class B may also *implement* an interface A. A class is a subclass of all interfaces it implements and all their superinterfaces.

An array type is built from another type, called its *component type*. Let t be a type; then $t[\,]$ is the array type with t as its component type. We use the notation $t[\,]^n$, defined as follows: $t[\,]^0 = t$, and $t[\,]^n = (t[\,]^{n-1})[\,]$. Arrays are *unidimensional*: an instance of an array has a set of components, each indexed by a single number. For example, an instance of type *int*[ ][ ] has components of type *int*[ ]. The *base type* of an array is defined as follows: the base type of a non-array type is the type itself; the base type of an array type is the base type of its component type. The *dimensionality* of an array is defined as follows: the dimensionality of a non-array type is zero; the dimensionality of an array type is the dimensionality of its component type plus one.

We define a *narrower-than* relation between types as follows. A class is narrower than the classes and interfaces of which it is a subclass. An interface is narrower than the interfaces of which it is a subinterface. The numeric primitive types have the following narrower-than relationship. Byte is narrower than all other primitive numeric types. Short is narrower than all other primitive numeric types except Byte. Char is narrower than all other primitive numeric types except Byte. Int is narrower than Long, Float and Double. Long is narrower than Float and Double. Float is narrower than Double. Finally, an array A is narrower than another array B if the component type of A is narrower than the component type of B.

An expression that computes a result returns either a *value* or a *variable*. A value is an instance of a type, and a variable is a reference to a value. Variables are typed. A variable of primitive type may refer only to a value of exactly that type. A variable of reference type may refer only to an instance of a reference type or to a *null* value. A non-null variable of class type may refer only to an instance of that class or one of its subclasses. A non-null variable of interface type may refer only to an instance of a class that implements the interface. A variable of array type of component type T may refer to an array of component type U only if a variable of type T may refer to a value of type U.

Declarations

In a *declaration* of a class, a programmer defines its name, *members*, *static initializers* and *constructors*. Members are *fields* or *methods*. A field is a named variable, specified as either a *class variable* (one variable shared by all instances of the class) or an *instance variable* (one unique variable for each instance of the class). A method is a named procedure that performs operations on the fields of the class. A method is specified as either a *class method* (which operates only on the class's class variables and needs no access to a particular object of the class) or an *instance method* (which operates on the variables of a given class instance). Every program in Java is an invocation of a method. Static initializers are procedures that set

[5] Note that an array instance has a class.

the initial state of the class (for instance, the values of its class variables). Constructors are procedures that set the initial state of a class instance.

Members, constructors, classes and interfaces are *declared entities*. Their declarations may include certain keywords representing *modifiers*. The presence of one of these keywords indicates that the entity being declared has the modifier as an attribute. We describe the meanings of particular modifiers later. The modifiers `public`, `protected` and `private` are *access modifiers*. A declaration may contain at most one of these modifiers. In addition, a declaration may contain at most one instance of the same modifier.

3.1 ASM J_0

Later in this section, we define a set of montages M_0. The ASM J_0 has a single agent, the *compiler*, whose module is C_0, the compile-time rule of M_0. All compile-time rules and conditions introduced in this section apply to the first compilation pass. The first pass establishes information on the structure of classes and the relations between classes, which is needed in the actions of the second pass.

3.2 Function and macro definitions

$PrimitiveType$	Universe of primitive types.
$Boolean, Byte, Short,$ $Char, Int, Long,$ $Float, Double : PrimitiveType$	Particular primitive types.
$Class$	Universe of classes.
$String, Throwable : Class$	Particular classes.
$Interface$	Universe of interfaces.
$Array$	Universe of array types.
$NullType$	Universe of null types.
$Null : NullType$	The null type.
$VarDeclaration$	Universe of variable declarations.
$Method$	Universe of methods.
$Constructor$	Universe of constructors.
$IDString$	Universe of identifier strings.
$IDString.class : Class$	The class with the given identifier.
$IDString.interface : Interface$	The interface with the given identifier.

macro $T.integral?$: $T \in \{Byte, Short, Char, Int, Long\}$
macro $T.numeric?$: $T.integral?$ or $T \in \{Float, Double\}$
macro $ReferenceType$: $Class \cup Interface \cup Array$
macro $Type$: $PrimitiveType \cup ReferenceType \cup NullType$

$Variable$	Universe of variables.
$PrimitiveValue$	Universe of primitive values.
$ClassInst$	Universe of class instances.
$ArrayInst$	Universe of array instances.
$NullRef$	Universe of null references.
$null : NullRef$	The null reference.
$StringValue$	Universe of string values.
$ClassInst.stringVal : StringValue$	String contents of the (String) class instance.
$ArrayInst.component(Nat) : Variable$	Array component with the given index.
$ArrayInst.length : Nat$	Length of the array.
$ReferenceType.declared? : Boolean$	Has the class or interface been declared?

macro $Object$: $ClassInst \cup ArrayInst \cup NullRef$
macro $Value$: $PrimitiveValue \cup Object$
macro $Result$: $Variable \cup Value$

$Object.class : ReferenceType$	Class of the object.
$Value.representableIn?(Type) : Boolean$	Can the value be represented in the given type?
$Value.convertTo(Type) : Value$	Result of converting the value to the given type.

Relations between types

$Class.subclassOf?(ReferenceType) : Boolean$	Is the class a subclass of the reference type?
$Interface.subinterfaceOf?(Interface) : Boolean$	Is the first interface a subinterface of the second interface?
$Type.array(Nat) : Type$	Type $t[\,]^n$, given type t and dimensionality n.
$Array.componentType : Type$	Component type of the array.
$Type.baseType : Type$	Base type of the type.
$Type.numDims : Type$	Dimensionality of the type.

macro $T.narrowerThan?(U)$:
$(T.subclassOf?(U)$ or $T.subinterfaceOf?(U))$
or $(T = Byte$ and $U \in \{Double, Float, Long, Int, Short, Char\})$
or $(T = Short$ and $U \in \{Double, Float, Long, Int, Char\})$
or $(T = Char$ and $U \in \{Double, Float, Long, Int, Short\})$
or $(T = Int$ and $U \in \{Double, Float, Long\})$
or $(T = Long$ and $U \in \{Double, Float\})$
or $(T = Float$ and $U = Double)$

Declarations

macro $DeclaredEntity$: $Class \cup Interface \cup VarDeclaration \cup Method \cup Constructor$

$Modifier$	Universe of declaration modifiers.
$public, protected, private,$ $static, abstract, final, native,$ $synchronized, transient, native$	Identify particular modifiers.
$Node.modifier : Modifier$	Modifier contained in given node.
$DeclaredEntity.accessStatus : Modifier$	Access status of entity.
$DeclaredEntity.static? : Boolean,$	
$DeclaredEntity.abstract? : Boolean,$	
$DeclaredEntity.final? : Boolean,$	
$DeclaredEntity.native? : Boolean,$	
$DeclaredEntity.synchronized? : Boolean,$	
$DeclaredEntity.transient? : Boolean,$	
$DeclaredEntity.volatile? : Boolean$	Does the entity have this modifier?

macro $MODLIST.hasModifier?(MOD)$:
(Does modifier list node $MODLIST$ contain modifier MOD?)
$(\exists n : n.memberOf?(MODLIST))n.modifier = MOD$

macro Assign $MODLIST$ To E:
(Assigns modifiers in modifier list node $MODLIST$ to entity E.)
Process Each Node n In List $MODLIST$
if $n.modifier \in \{public, protected, private\}$ **then** $E.accessStatus := n.modifier$
elseif $n.modifier = static$ **then** $E.static? := true$
elseif $n.modifier = abstract$ **then** $E.abstract? := true$
elseif $n.modifier = final$ **then** $E.final? := true$
elseif $n.modifier = native$ **then** $E.native? := true$
elseif $n.modifier = synchronized$ **then** $E.synchronized? := true$
elseif $n.modifier = transient$ **then** $E.transient? := true$
elseif $n.modifier = volatile$ **then** $E.volatile? := true$

macro $MODLIST.consistent?$:
(Is modifier list node $MODLIST$ free of duplicate modifiers and conflicting access modifiers?)
For All Distinct Members m, n Of List $MODLIST$
$m.modifier \neq n.modifier$
and $(m.modifier \in \{public, protected, private\} \Rightarrow n.modifier \notin \{public, protected, private\})$

3.3 Type references

A given primitive type is identified by the appropriate keyword (e.g. `boolean`). A given class or interface is identified by its identifier string. An array is identified by a type reference followed by a sequence of `[ ]` tokens, each representing a dimension of the array.

$Node.type : Type$	Type specified in the given node.
$Node.ID : IDString$	Identifier specified in the given node.

Type = **PrimitiveType** |**ReferenceType**
PrimitiveType = boolean |byte |short |char |int |long |float |double
ReferenceType = ClassType |InterfaceType |ArrayType

if *curNode.type.*undef*?* **then**
curNode.type :=
(**if** *curNode.isA?(booleanNode)* **then** *Boolean*
elseif *curNode.isA?(byteNode)* **then** *Byte*
elseif *curNode.isA?(shortNode)* **then** *Short*
elseif *curNode.isA?(charNode)* **then** *Char*
elseif *curNode.isA?(intNode)* **then** *Int*
elseif *curNode.isA?(longNode)* **then** *Long*
elseif *curNode.isA?(floatNode)* **then** *Float*
elseif *curNode.isA?(doubleNode)* **then** *Double)*

TypeName = **Identifier**
curNode.type := (**if** *curNode.ID.class.declared?* **then** *curNode.ID.class* **else** *curNode.ID.interface)*
condition *curNode.ID.class.declared? <=> not curNode.ID.interface.declared?*

ClassType = TypeName
condition *curNode.type.isA?(Class)*

InterfaceType = TypeName
condition *curNode.type.isA?(Interface)*

ArrayType ::= Type [ ]
curNode.type := Type.type.array(1)

3.4 Field declaration

By including a *field declaration* in the declaration of a class c, a programmer declares a set of fields for c. Each field in the declaration has a distinct identifier. A common type and set of modifiers are specified for all the fields in the declaration. The modifiers are any of the following:

- access status: determines where references to these fields may appear. (Discussed further in Section 5.)
- final: selected if classes that extend c are forbidden to hide any of these fields. (Discussed further in Section 3.8.)
- static: determines whether these fields are static (class) variables, each with one instantiation per class, or instance variables, each with one instantiation per class instance.

Field declaration

A $FieldDeclaration$ contains a $Type$, a list of $FieldDeclarations$ and a list of $VariableDeclarators$. The identifiers in $VariableDeclarators$ must be distinct. The modifiers in $FieldModifiers$ must be distinct and must specify at most one access status.

To process $FieldDeclaration$, create a new field for each identifier in the list $VariableDeclarators$ and assign each the type specified in $Type$ and the modifier attributes in $FieldModifiers$. If the fields are declared static, create a new variable for each one.

$Node.field(IDString) : VarDeclaration$	Field declared in given declaration node with given ID.
$Node.declaration : VarDeclaration$	Variable declaration in given declarator node.
$VarDeclaration.type : Type$	Type of field's variable.
$VarDeclaration.initExpr : Task$	Initializer expression for field.

FieldDeclaration	**::=**	**[FieldModifiers] Type VariableDeclarators ;**
FieldModifiers	=	LIST(FieldModifier)
FieldModifier	=	public |protected |private |final |static |transient |volatile
VariableDeclarators	=	LIST(VariableDeclarator,)

Process Each Node n In List VariableDeclarators
curNode.field(n.ID) := n.declaration
Assign FieldModifiers To n.declaration
n.declaration.type := Type.type

condition *FieldModifiers.consistent?*
(For All Distinct Members m,n Of List VariableDeclarators) m.ID ≠ n.ID

Variable declarator

VariableDeclarator ::= VariableDeclaratorId [= VariableInitializer]

curNode.ID := VariableDeclaratorId.ID
extend *Declaration* **with** *d*
curNode.declaration := d
d.initExpr := VariableInitializer
VariableDeclaratorId.ID.localVarDecl := d

condition *VariableDeclaratorId.ID.localVarDecl.*undef*?*

3.5 Method declaration

By including a *method declaration* in the declaration of a class c, a programmer declares a method for class c. A method has an identifier, a list of *parameter types*, a *return type*, a list of *thrown exceptions*, and may have any of the following attributes:

- access status: determines where invocations of this method may appear. (Discussed further in Section 5.)
- final: selected if classes that extend c are forbidden to hide or override this method. (Discussed further in Section 3.8.)
- static: determines whether this method is a static (class) method (invoked without reference to a particular object) or an instance method (invoked upon an instance of this class).
- abstract: selected if the programmer has supplied only the *signature* (identifier, return type and parameter types) for the method.
- native: selected if the implementation of the method is given in terms of non-Java code.

$TypeSeq$	Universe of indexed sets of types.
$Method.type : Type$	Return type of the method.
$Method.throws?(Class) : Boolean$	Does the method throw the given class?
$Method.ID : IDString$	Identifier of the given method.
$Method.paramTypes : TypeSeq$	Types of the given method's parameters.
$Method.paramDecl(Nat) : VarDeclaration$	Parameter declaration with given index for the method.

Method declaration

A *MethodDeclaration* contains a *MethodHeader* and a *MethodBody*. If *MethodBody* is a block of code, then the method created in *MethodHeader* must not be specified as abstract.

Node.method : *Method*	Method created within the given node.

MethodDeclaration ::= MethodHeader MethodBody MethodBody = Block |;
curNode.method := MethodHeader.method
condition *MethodBody.isA?(BlockNode) =>* *not (MethodHeader.method.abstract? or MethodHeader.method.native?)*

Method header

A *MethodHeader* contains an optional list of *MethodModifiers*, a *ReturnType*, a *MethodDeclarator* and an optional list *Throws* of thrown exception types. There must be at most one instance of each modifier, and at most one access status modifier, in *MethodModifiers*. If the method is declared abstract, then it must not be declared private, static, final, native or synchronized.

To process *MethodHeader*, assign the modifiers in *MethodModifiers* and the list of thrown exceptions in *Throws* to the method created in *MethodDeclarator*. Assign the method the return type $t[\,]^n$, where t is the type specified in *ReturnType*, and n is the number of dimensions specified in *MethodDeclarator*.

$Node.numDims : Nat$	Number of array dimensions specified in the given node.
$Node.throws?(Class) : Boolean$	Has the given class been specified as a thrown exception in the given node?

MethodHeader	**::=**	**[MethodModifiers] ReturnType MethodDeclarator [Throws]**
ReturnType	=	Type |void
MethodModifiers	=	LIST(MethodModifier)
MethodModifier	=	public |protected |private |final |static |abstract |native

```
let m = MethodDeclarator.method
  curNode.method := m
  Assign MethodModifiers To m
  m.type := ReturnType.type.array(MethodDeclarator.numDims)
  do-forall e: e.isA?(Class): Throws.throws?(e)
    m.throws?(e) := true

condition MethodModifiers.consistent?
          MethodModifiers.hasModifier?(abstract) =>
            (forall mod: mod in {private,static,final,native,synchronized})
              not MethodModifiers.hasModifier?(mod)
          ReturnType.isA?(voidNode) => MethodDeclarator.numDims = 0
```

Method declarator

A *MethodDeclarator* contains an *Identifier*, an optional *FormalParameterList* and an optional list *Dims* of `[ ]` tokens. The identifiers in *FormalParameterList* must be distinct.

To process *MethodDeclarator*, create a method and assign it the identifier in *Identifier* and the parameter types in *FormalParameterList*.

MethodDeclarator	**::=**	**Identifier ([FormalParameterList]) [Dims]**
FormalParameterList	=	LIST(FormalParameter;)
Dims	=	LIST()

```
curNode.numDims := Dims.length
extend Method with m
  curNode.method := m
  m.ID := Identifier.ID
  m.paramTypes :=
    {(i,FormalParameterList[i].declaration.type): FormalParameterList[i].def?}
  Process Each Node n In List FormalParameterList
    m.paramDecl(n.position) := n.declaration

condition (For All Distinct Members m,n Of List FormalParameterList) m.ID ≠ n.ID
```

Formal parameter

A *FormalParameter* specifies a *VariableDeclaratorId* identifier and *Type* for a variable. The identifier must not name any previously declared local variable or parameter.

FormalParameter ::= Type VariableDeclaratorId
curNode.ID := VariableDeclaratorId.ID **extend** *Declaration* **with** *d* *curNode.declaration := d* *d.type := Type.type* *VariableDeclaratorId.ID.localVarDecl := d*
condition *VariableDeclaratorId.ID.localVarDecl.undef?*

Throws clause

A *Throws* clause specifies a set of thrown exception classes in *ClassTypeList*. Each thrown class must be a subclass of the class Throwable.

Throws ::= throws ClassTypeList ClassTypeList = LIST(ClassType,)
Process Each Node n In List ClassTypeList *curNode.throws?(n.type) :=* true
condition *(For Each Member n Of List ClassTypeList) n.type.subclassOf?(Throwable)*

3.6 Static initializer

A *StaticInitializer* declaration specifies a block of code to execute when a class is initialized. We discuss static initializers further in Section 7.

StaticInitializer ::= static Block


3.7 Constructor declaration

By including a *constructor declaration* in the declaration of a class c, a programmer declares a constructor for class c. A constructor has an identifier (which must match the identifier of the class), a list of parameter types, a list of thrown exceptions, and an access status modifier.

Constructor.throws?(Class) : Boolean	Has the given class been specified as a thrown exception in the given constructor?
Constructor.paramTypes : TypeSeq	Types of the given constructor's parameters.
Constructor.ID : IDString	Identifier of the given constructor. (must match identifier of its class).
Constructor.paramDecl(Nat) : VarDeclaration	Parameter declaration with given index.

Constructor declaration

Node.constructor : *Constructor*	Constructor created within the given node.

ConstructorDeclaration	**::=**	**[ConstructorModifiers] ConstructorDeclarator [Throws] ConstructorBody**
ConstructorModifiers	=	LIST(ConstructorModifer)
ConstructorModifier	=	public |protected |private

let c = ConstructorDeclarator.constructor
curNode.constructor := c
Assign ConstructorModifiers To c
*do-***forall** *e: e.isA?(Class): Throws.throws?(e)*
c.throws?(e) := true

condition *ConstructorModifiers.consistent?*

Constructor declarator

A *ConstructorDeclarator* contains a *TypeName* and an optional *FormalParameterList*. The identifiers in *FormalParameterList* must be distinct.

To process *ConstructorDeclarator*, create a constructor and assign it the parameter types in *FormalParameterList*.

ConstructorDeclarator ::= TypeName ([FormalParameterList])

extend *Constructor* **with** *c*
curNode.constructor := c
c.ID := TypeName.ID
c.paramTypes :=
{(i,FormalParameterList[i].declaration.type): FormalParameterList[i].def?}
Process Each Node n In List FormalParameterList
c.paramDecl(n.position) := n.declaration

3.8 Class declaration

In a *class declaration*, a programmer defines a new class. A class has an identifier specified in *Identifier*, a parent class specified in *Super*, a list of implemented interfaces specified in *Interfaces*, a set of members, static initializers and constructors specified in *ClassBody*, and may have any of the following attributes specified in *ClassModifiers*:

- public: to be discussed later.
- abstract: selected if the class has an incomplete definition because (1) it declares an abstract method, (2) it inherits an abstract method, or (3) it fails to implement a method declared in an interface.
- final: selected if the class is forbidden to have subclasses.

Members are *installed* in a class by either explicit declaration or *inheritance*. A class *inherits* the fields of its superclass and superinterfaces. If it declares a field with the same identifier as an inherited field, the inherited field is *hidden*. A field declared as final may not be hidden. If a class inherits a field from its parent class and another field of the same identifier from an interface, or if it inherits two fields of the same identifier from different interfaces, the fields are not installed. A class inherits any method of its superclass for which it does not declare a method of the same signature. If it declares a method with the same identifier and parameter types as an inherited method, the declared method *overrides* the inherited method. A method declared as final may not be overridden.

$Class.parentClass : Class$	Parent of the class.
$Class.field(IDString, ReferenceType) : VariableDeclaration$	Field of the class with the given ID, inherited from the given superclass.
$Class.method(IDString, TypeSeq) : Method$	Method of the class with the given identifier and parameter types.

Class declaration

The identifier of the class must not be used by another class or interface. The declared parent class must not be final. If the class definition is incomplete, the class must be declared abstract, and must not be declared final. An interface must not be named more than once in the *Interfaces* clause, and any methods they share, matching in identifier and parameter types, must also agree in return type. Interface methods must also agree with any matching declared or inherited methods in terms of return type.

$Node.hasInterface?(Interface) : Boolean$	Does the given interface list node contain the given interface?
$Node.method(IDString, TypeSeq) : Method$	Method declared in the node with given identifier and parameter types.
$Node.constructor(TypeSeq) : Constructor$	Constructor declared in the node with given parameter types.
$Task.curClass : Class$	Class in whose declaration the task appears.
$DeclaredEntity.declarer : Class$	Class in which the field or method is declared.

macro *declaredParent*:
(parent class of new class)
(if $Identifier.ID = "Object"$ then *undef* else (if $Super.isPresent?$ then $Super.type$ else $Object$))

macro $NAME.methodToInstall(TYPES)$:
(Method with given identifier and parameter types to install in the class:
method declared in $ClassBody$, if present; otherwise, method of parent)
(if $ClassBody.method(NAME, TYPES).def?$ then $ClassBody.method(NAME, TYPES)$
else $declaredParent.method(NAME, TYPES)$)

macro *interfacesToImplement*:
(set of interfaces listed in *Interfaces* clause)
$\{int : int.isA?(Interface) : Interfaces.hasInterface?(int)\}$

macro *incompleteClassDefinition?*:
(Does the new class install an abstract method?)
$(\exists id, t : id.isA?(IDString)$ and $t.isA?(TypeSeq))$
$id.methodToInstall(t).abstract?$
or $(id.methodToInstall(t).undef?$ and $(\exists i \in interfacesToImplement)i.method(id, t).def?)$

macro $CLASS$ Inherit From Parent Class:
(Establish $CLASS$ as subclass of its parent and of all its parent's subclasses;
inherit fields and methods from parent class.)
do-forall $b : b.isA?(Class) : declaredParent.subclassOf?(b)$
    $CLASS.subclassOf?(b) := true$
    do-forall $id : id.isA?(IDString)$
        $CLASS.field(id, b) := declaredParent.field(id, b)$
do-forall $id : ClassBody.field(id).undef?$ and $declaredParent.field(id, declaredParent).def?$
    if $(\forall int \in interfacesToImplement : int.field(id, int).def?)$
        $int.field(id, int) = declaredParent.field(id, declaredParent)$ **then**
            $CLASS.field(id, CLASS) := declaredParent.field(id, declaredParent)$
do-forall $id, t : ClassBody.method(id, t).undef?$ and $declaredParent.method(id, t).def?$
    $CLASS.method(id, t) := declaredParent.method(id, t)$

macro $CLASS$ Inherit From Interfaces:
(Establish $CLASS$ as subclass of the interfaces it implements; inherit fields from interfaces.)
do-forall $i : i \in interfacesToImplement$
    do-forall $j : i = j$ or $i.subinterfaceOf?(j)$
        $CLASS.subclassOf?(j) := true$
        do-forall $id : id.isA?(IDString)$
            $CLASS.field(id, j) := i.field(id, j)$
    do-forall $id : ClassBody.field(id).undef?$ and $i.field(id, i).def?$
        if $declaredParent.field(id, declaredParent).undef?$
            and $(\forall j \in interfacesToImplement$: $j.field(id, j).def?)$ $j.field(id, j) = i.field(id, i))$ **then**
            $CLASS.field(id, CLASS) := i.field(id, i)$

macro Set Up Initializers:
(Establish static and instance initializers for class. Details given in Section 7.)
skip

ClassDeclaration	**::=**	**[ClassModifiers] class Identifier [Super] [Interfaces] ClassBody**
ClassModifiers	=	LIST(ClassModifier)
ClassModifier	=	public |abstract |final

```
let c = Identifier.ID.class
  c.declared? := true
  do-forall t: t.isA?(Task): ClassBody.contains?(t)
    t.curClass := c
  Assign ClassModifiers To c
  c.parentClass := parent
  c Inherit From Parent Class
  c Inherit From Interfaces
  do-forall id: ClassBody.field(id).def?
    c.field(id,c) := ClassBody.field(id)
    ClassBody.field(id).declarer := c
  do-forall id,t: ClassBody.method(id,t).def?
      c.method(id,t) := ClassBody.method(id,t)
      ClassBody.method(id,t).declarer := c
  do-forall t: ClassBody.constructor(t).def?
    c.constructor(t) := ClassBody.constructor(t)
  Set Up Initializers
```

condition *not (Identifier.ID.class.declared? or Identifier.ID.interface.declared?)*
declaredParent.def? => declaredParent.declared?
Identifier.ID = "Object" => not Super.isPresent?
incompleteClassDefinition? =>
(ClassModifiers.hasModifier?(abstract) and not ClassModifiers.hasModifier?(final))
not declaredParent.final?
(forall *id,t: id.methodToInstall(t).def?)*
(forall *int: int in interfacesToImplement: int.method(id,t).def?)*
id.methodToInstall(t).type = int.method(id,t).type
(forall *t: ClassBody.constructor(t).def?)*
ClassBody.constructor(t).ID = Identifier.ID

Super clause

Super	**::=**	**extends ClassType**

```
curNode.type := ClassType.type
```

Interfaces clause

An *Interfaces* clause specifies a list of interfaces in *InterfaceTypeList*. Each interface must be distinct, and any method declarations between interfaces that match in terms of identifier and parameter types must also agree in terms of return type.

Interfaces	**::=**	**implements InterfaceTypeList**
InterfaceTypeList	=	LIST(InterfaceType,)

Process Each Node n In List InterfaceTypeList
curNode.hasInterface?(n.type) := true

condition *(For All Distinct Members m,n Of List InterfaceTypeList) m.type* $\neq$ *n.type*
(**forall** *id,t: id.isA?(IDString) and t.isA?(TypeSeq))*
(For All Distinct Members m,n Of List InterfaceTypeList)
m.type.method(id,t).def? and n.type.method(id,t).def?) =>
m.type.method(id,t).type = n.type.method(id,t).type

Class body

A *ClassBody* contains a list of *ClassBodyDeclarations*; each is a declaration of a field, method, static initializer or constructor. There must be no two method declarations that match in terms of identifier and parameter types, no two constructor declarations that match in terms of parameter types, and no two field declarations which declare fields with the same identifier.

macro Add Class Initializer:
(Add static initializer for class. Details given in Section 7.)
skip

macro Add Field Initializers:
(Add initializers of instance fields for class. Details given in Section 7.)
skip

ClassBody	**::=**	**{ [ClassBodyDeclarations] }**
ClassBodyDeclarations	=	LIST(ClassBodyDeclaration)
ClassBodyDeclaration	=	ClassMemberDeclaration |StaticInitializer
.	=	|ConstructorDeclaration
ClassMemberDeclaration	=	FieldDeclaration |MethodDeclaration

Process Each Node n In List ClassBodyDeclarations
  if *n.isA?(MethodDeclarationNode)* **then**
    curNode.method(n.method.ID, n.method.paramTypes) := n.method
  elseif *n.isA?(ConstructorDeclarationNode)* **then**
    curNode.constructor(n.constructor.paramTypes) := n.constructor
  elseif *n.isA?(StaticInitializerNode)* **then**
    Add Class Initializer
  elseif *n.isA?(FieldDeclarationNode)* **then**
    do-**forall** *id in IDString* **with** *n.field(id).def?*
      field(id) := n.field(id)
    if *n.static?* **then** *Add Class Initializer*
    else *Add Field Initializers*

condition *(For All Distinct Members m,n Of List ClassBodyDeclarations)*
m.isA?(MethodDeclarationNode) and n.isA?(MethodDeclarationNode)) =>
not (m.method.ID = n.method.ID
and m.method.paramTypes = n.method.paramTypes)
(For All Distinct Members m,n Of List ClassBodyDeclarations)
m.isA?(ConstructorDeclarationNode) and n.isA?(ConstructorDeclarationNode)) =>
m.constructor.paramTypes ≠ n.constructor.paramTypes
(For All Distinct Members m,n Of List ClassBodyDeclarations)
(**forall** *id: id.isA?(IDString)) not (m.field(id).def? and n.field(id).def?)*

3.9 Interface declaration

In an interface declaration, a programmer defines a new interface. An interface has an identifier specified in $Identifier$, a list of parent interfaces specified in $ExtendsInterfaces$, and a set of members specified in $InterfaceBody$. Every interface is implicitly public and abstract. If an interface specifies no parent interfaces, it implicitly extends the interface Cloneable.

macro $interfacesToExtend$:
(Set of interfaces listed in $ExtendsInterfaces$ clause.)
$\{int : int.isA?(Interface) : ExtendsInterfaces.hasInterface?(int)\}$

macro $INTERFACE$ Inherit From Parent Interfaces:
(Establish $INTERFACE$ as subinterface of all the interfaces it extends; inherit fields from interfaces.)
if $Interfaces.isPresent?$ **then**
  do-forall $i : i \in interfacesToExtend$
    do-forall $j : i = j$ or $i.subinterfaceOf?(j)$
      $INTERFACE.subinterfaceOf?(j) := true$
      do-forall $id : id.isA?(IDString)$
        $INTERFACE.field(id, j) := i.field(id, j)$
    do-forall $id : id.isA?(IDString) : InterfaceBody.field(id).undef?$ and $i.field(id, i).undef?$
      if $(\forall j \in interfacesToExtend : j.field(id, j).def?) j.field(id, j) = i.field(id, i)$ **then**

$INTERFACE.field(id, INTERFACE) := i.field(id, i)$
else $INTERFACE.subinterfaceOf?(Cloneable) := true$

InterfaceDeclaration	**::=**	**[InterfaceModifiers] interface Identifier [ExtendsInterfaces] InterfaceBody**
InterfaceModifiers	=	LIST(InterfaceModifier)
InterfaceModifier	=	public |abstract

let int = Identifier.ID.interface
int.declared? := true
int.accessStatus := public
int.abstract? := true
int Inherit From Parent Interfaces
*do-***forall** *id: id.isA?(IDString): InterfaceBody.field(id).def?*
int.field(id,int) := InterfaceBody.field(id)
InterfaceBody.field(id).declarer := int
*do-***forall** *id, t: id.isA?(IDString) and t.isA?(TypeSeq): InterfaceBody.method(id,t).def?*
int.method(id,t) := InterfaceBody.method(id,t)

condition *not (Identifier.ID.class.declared? or Identifier.ID.interface.declared?)*
(**forall** *int: int in interfacesToExtend) int.declared?*

Interface extension clause

An *ExtendsInterfaces* clause specifies a list of interfaces in *InterfaceTypeList*. Each interface must be distinct, and any method declarations between interfaces that match in terms of identifier and parameter types must also agree in terms of return type.

ExtendsInterfaces	**::=**	**extends InterfaceTypeList**

Process Each Node n In List InterfaceTypeList
curNode.hasInterface?(n.type) := true

condition *(or All Distinct Members Of List InterfaceTypeList) m.type* $\neq$ *n.type*
(**forall** *id,t: id.isA?(IDString) and t.isA?(TypeSeq))*
(For All Distinct Members m,n Of List InterfaceTypeList)
m.type.method(id,t).def? and n.type.method(id,t).def? =>
m.type.method(id,t).type = n.type.method(id,t).type

Interface body

An *InterfaceBody* contains a list of *InterfaceBodyDeclarations*; each is a declaration of a constant field or an abstract method. There must be no two method declarations that match in terms of identifier and parameter types, and no two field declarations which declare fields with the same type.

InterfaceBody	**::=**	**{ [InterfaceMemberDeclarations] }**
InterfaceMemberDeclarations	=	LIST(InterfaceMemberDeclaration)
InterfaceMemberDeclaration	=	ConstantDeclaration |AbstractMethodDeclaration

Process Each Node n In List InterfaceMemberDeclarations
  if *n.isA?(AbstractMethodDeclarationNode)* **then**
    curNode.method(n.method.ID, n.method.paramTypes) := n.method
  elseif *n.isA?(ConstantDeclarationNode)* **then**
    do-**forall** *id: id.isA?(IDString): n.field(id).def?*
      curNode.field(id) := n.field(id)

condition *(For All Distinct Members m,n Of List InterfaceMemberDeclarations)*
m.method.def? and n.method.def? => not (m.method.ID = n.method.ID
and m.method.paramTypes = n.methodParamTypes)
(For All Distinct Members m,n Of List InterfaceMemberDeclarations)
m.constructor.def? and n.constructor.def? =>
m.constructor.paramTypes ≠ n.constructor.paramTypes
(For All Distinct Members m,n Of List InterfaceMemberDeclarations)
(**forall** *id: id.isA?(IDString)) not (m.field(id).*undef*? and n.field(id).*undef*?)*

Constant declaration

A *ConstantDeclaration* is a restricted form of *FieldDeclaration* in which each declared field is implicitly public, static and final and is required to have an initializing value.

ConstantDeclaration	**::=**	**ConstantModifiers Type VariableDeclarators**
ConstantModifiers	=	LIST(ConstantModifier)
ConstantModifier	=	public |static |final

Process Each Node n In List VariableDeclarators
  curNode.field(n.ID) := n.declaration
  n.declaration.accessStatus := public
  n.declaration.static? := true
  n.declaration.final? := true
  n.declaration.type := Type.type

condition *(For All Distinct Members m,n Of List VariableDeclarators) m.ID ≠ n.ID*

Abstract method declaration

An *AbstractMethodDeclaration* is a restricted form of method declaration in which the declared method is implicitly public and abstract.

AbstractMethodDeclaration	**::=**	**[AbstractMethodModifiers] ReturnType MethodDeclarator [Throws] ;**
AbstractMethodModifiers	=	LIST(AbstractMethodModifier)
AbstractMethodModifier	=	public |abstract

let m = MethodDeclarator.method
  curNode.method := m
  m.accessStatus := public
  m.abstract? := true
  m.type := ReturnType.type.array(MethodDeclarator.numDims)
  do **forall** *e: e.isA?(Class): Throws.throws?(e)*
    m.throws?(e) := true

4 Execution of statements

Control flow, the ordering of program actions during a run, is determined by *statements*, syntactic constructs representing commands. This ASM specifies for each statement type how control flow is established for statements of that type.

4.1 Preliminaries

In *normal* execution mode, control passes from a statement to one of a fixed set of successors. Normal execution may be interrupted by the *abrupt* completion of a statement. This may arise from the execution of a statement that orders an abrupt completion, or the occurrence of an *exception* during the evaluation of an expression. When a statement completes abruptly, an enclosing *target* statement is established, and all statements intervening between the current and target statements complete abruptly. Of these statements, only those components marked as *mandatory* are executed. Normal execution resumes when the target is reached. We refer to the process of executing the mandatory portions of the intervening statements as *jumping toward* the target.

Tasks are atomic units of work that serve as statement components. They may be classified according to their control flow characteristics. For a sequential task, there is a unique successor always chosen as the next task. For a branch task, selection of the next task is based on a previously calculated *test result*. The task has fixed successors for certain results and a default successor for all others. For a jump task, a target is chosen. If the target is reachable immediately, control passes to it. Otherwise, there are intervening mandatory tasks to execute, so control passes to the first intervening mandatory task. If a target had previously been set, it is discarded in favor of the new target.

4.2 ASM J_1

The ASM J_1 has two agents. The *compiler* has module C_1, the compile-time rules of M_1, and the *executor* has module R_1, the runtime rules of M_1. M_1 consists of the montages of M_0 plus the montages defined in the remainder of this section. In all runs of J_1, the compiler runs to completion before the executor.

4.3 Function and macro definitions

All compile-time rules and conditions introduced in this section and all following sections apply to the second compilation pass.

$normal? : Boolean$	is execution in normal mode?
$curTask : Task$	Current task to execute.
$curTargetTask : Task$	current target task (if jumping).
$Task.nextTask : Task$	task to which control passes in normal execution mode.
$Task.testExpr : Task$	test expression for branch task.
$Task.branchTask(Result) : Task$	task to which control passes given the test value.
$Task.nextMandatoryBlock : Task$	first task of the next mandatory block.
$Task.endTask : Task$	final task of the given task's method or constructor.
$Task.targetTask : Task$	target of the given jump task.
$Task.catchTarget(Class) : Task$	target task to establish if an exception of the given class is thrown.
$Task.inMandatoryBlock? : Boolean$	Is this task part of a mandatory block?

Terms of the form $curTask.f(\bar{x})$, where f is a function name and $\bar{x}$ is a sequence of terms, are usually abbreviated as $f(\bar{x})$.

macro Jump Toward t
(Jump toward a target task. If there are mandatory tasks between the current and target tasks,
start a jump toward the target. Otherwise, pass control directly to the target.)
if $nextMandatoryBlock \neq t.nextMandatoryBlock$ **then**
    $normal? := false$
    $curTargetTask := t$
    $curTask := nextMandatoryBlock$
elseif $t.undef?$ **then** $curTask := endTask$
else
    $normal? := true$
    $curTask := t$

macro Proceed Sequentially
(Pass control from a sequential task to another task. If the execution mode is normal
or a mandatory block is being executed, pass control to the task's successor.
Otherwise, continue the jump in progress toward a target task.)
if $normal?$ or $nextTask.inMandatoryBlock?$ **then** $curTask := nextTask$
else Jump Toward $curTargetTask$

macro BranchOnTestResult:
(Choose a successor based on a previously computed test result.
If a successor is defined for the test result, pass control to that successor;
otherwise pass to a default successor.)
if $branchTask(testExpr.result).def?$ **then** $curTask := branchTask(testExpr.result)$
else Proceed Sequentially

macro Throw E:
(Throw the exception given by E.)
$curException := E$
Jump Toward $catchTarget(E.class)$

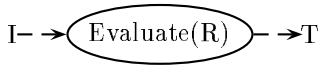
macro Throw Exception Of Class C:
(Create an exception object of class C and throw it.)
extend $ClassInst$ **with** e
    $curException := e$
    $e.class := C$
    Jump Toward $catchTarget(C)$

4.4 Construction of statements

For each type of statement, we describe the actions performed when executing a statement of that type, and how to construct a representation of the statement in the initial state. Since we consider only the flow of control between statements here, we ignore the internal structure of expressions. The montage for expressions contains a single task *Evaluate*. This task is assigned an arbitrary type and executes by either (1) returning an arbitrary result and proceeding sequentially or (2) throwing an exception.

Task.result : *Result*	result of expression evaluation.
Task.type : *Type*	type of result returned by the task.
Task.constant? : *Boolean*	Is the expression (task) constant?

Expression ::= .



choose t: t.isA?(Type)
Evaluate.type := t

Evaluate:
choose among

do
choose r: r.isA?(Result): r.representableIn?(type)
result := r
Proceed Sequentially

choose c in Class **with** *c.subclassOf?(Throwable)*
Throw Exception Of Class c

ConstantExpression = Expression

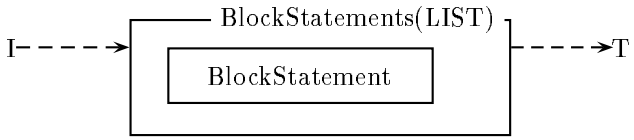
condition *Expression.constant?*

4.4.1 Block

A *Block* contains a list of *BlockStatements*. To execute *Block*, execute the statements in *BlockStatements* sequentially. (Any local variables declared within the *Block* are undefined outside the *Block*.)

IDString.localVarDecl : *VariableDeclaration*	Declaration of local variable or parameter with given identifier.

Block	**::=**	**{ [BlockStatements] }**
BlockStatements	=	LIST(BlockStatement)
BlockStatement	=	LocalVariableDeclarationStatement |Statement
Statement	=	StatementNoTrailingSubstatement |LabeledStatement
.	=	IfThenStatement |IfThenElseStatement
.	=	|WhileStatement |ForStatement
StatementNoTrailingSubstatement	=	Block |EmptyStatement |ExpressionStatement
.	=	|SwitchStatement |DoStatement |BreakStatement
.	=	|ContinueStatement |ReturnStatement
.	=	|SynchronizedStatement |ThrowStatement
.	=	|TryStatement



Process Each Node m in List BlockStatements
  if *m.isA?(LocalVariableDeclarationStatement)* **then**
    Process Each Node n in List m.VariableDeclarators
      n.ID.varDecl := undef

4.4.2 Local variable declaration statement

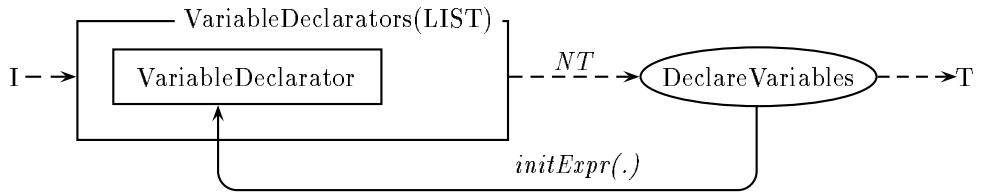
A *LocalVariableDeclarationStatement* contains a list of *VariableDeclarators*. Each variable declaration in *VariableDeclarators* is assigned the type given in *Type*. The variable identifiers of *VariableDeclarators* must be distinct, and the initializer expression of each *VariableDeclarator* (if it is present) must be assignable to the type given in *Type*.

To execute *LocalVariableDeclarationStatement*, create a variable of the type specified by *Type* for each declarator in *VariableDeclarators*. If the declarator specifies an initial value for the variable, assign it to the variable.

$VariableDeclaration.localVar : Variable$	Local variable created by the declaration.

macro $E.assignableTo?(T)$:
(Is the result of the expression E assignable to the type T?
Only if E's type is narrower than or equal to T,
or if E is a constant int value and T is narrower than Int.) $E.type.numDims = T.numDims$
 and ($E.type.baseType = T.baseType$
  or $E.type.baseType.narrowerThan?(T.baseType)$
  or ($T.baseType.narrowerThan?(E.type.baseType)$ and $E.constant?$ and
   $T.baseType \in \{Byte, Short, Char\}$ and $E.result.representableIn?(T.baseType)$)

LocalVariableDeclarationStatement **::=** **Type VariableDeclarators ;**
VariableDeclarators = LIST(VariableDeclarator,)



Process Each Node n In List VariableDeclarators
n.declaration.type := Type.type
DeclareVariables.declaration(n.position) := n.declaration

condition *(For All Distinct Members m,n Of List VariableDeclarators) m.ID ≠ n.ID*
(For Each Member n Of List VariableDeclarators) n.assignableTo?(Type.type)

DeclareVariables:
*do-***forall** *i: declaration(i).def?*
extend *Variable* **with** *var*
declaration(i).localVar := var
if *initExpr(i).def?* **then** *var.value := initExpr(i).result*
Proceed Sequentially

4.4.3 Empty statement

An *EmptyStatement* is not executed.

EmptyStatement **::=** **;**

Proceed: Proceed Sequentially

4.4.4 Labeled statement

A *LabeledStatement* contains an *Identifier* and a *Statement*. *Statement* must not contain any labeled statements with the same label as that of *Identifier*. If *Statement* contains a *ContinueStatement* with the same label as that of *Identifier*, then *Statement* must be a loop statement (*WhileStatement*, *DoStatement* or *ForStatement*).

To execute *LabeledStatement*, execute *Statement*. (For any break or continue statement within *Statement* with the label of *Identifier*, *Statement* contains its target.)

Default	Universe of default values.
default : *Default*	The default label value.

macro *JumpLabel*: $IDString \cup Default$

$Task.label : JumpLabel$	Label of this break or continue task.
$Node.continueTarget : Task$	target of any continue task within this node.

LabeledStatement ::= Identifier : Statement

*do-***forall** *t: t.isA?(BreakTask): Statement.contains?(t) and t.label = Identifier.ID*
t.targetTask := Statement.terminalTask
*do-***forall** *t: t.isA?(ContinueTask): Statement.contains?(t) and t.label = Identifier.ID*
t.targetTask := Statement.continueTarget

condition *((***exists** *t : Statement.contains?(t) and t.isA?(ContinueTask)) t.label = Identifier.ID) =>*
(Statement.isA?(WhileStatementNode) or Statement.isA?(DoStatementNode)
or Statement.isA?(ForStatementNode))
*not (***exists** *n : Statement.contains?(n) and n.isA?(LabeledStatement))*
n.Identifier.ID = Identifier.ID

4.4.5 Expression statement

An *ExpressionStatement* contains a *StatementExpression*. *StatementExpression* must not have a type.

To execute *ExpressionStatement*, evaluate *StatementExpression*.

ExpressionStatement ::= StatementExpression ;

condition *StatementExpression.type.*undef*?*

4.4.6 If statements

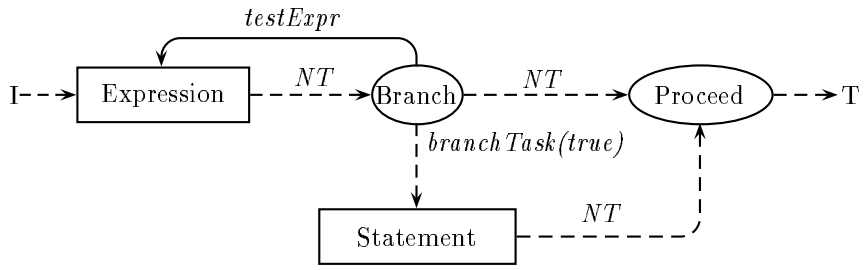
An *IfThenStatement* contains an *Expression* and a *Statement*. The type of *Expression* must be Boolean.

To execute *IfThenStatement*, evaluate *Expression*. If the result is *true*, execute *Statement*.

An *IfThenElseStatement* contains an *Expression*, a *Statement* and a *StatementNoShortIf*. The type of *Expression* must be Boolean.

To execute *IfThenElseStatement*, evaluate *Expression*. If the result is *true*, execute *StatementNoShortIf*; if it is *false*, execute *Statement*.

IfThenStatement ::= if (Expression) Statement

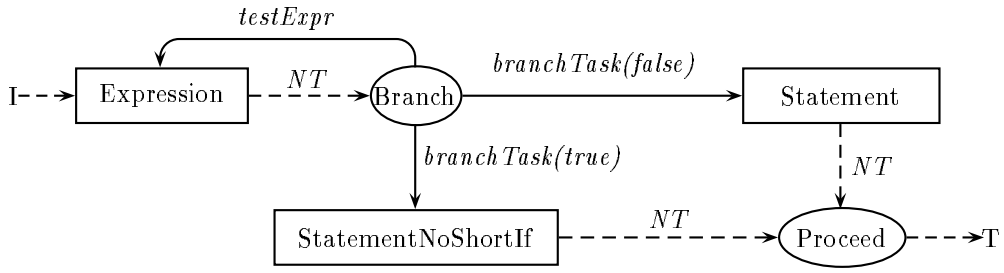


condition *Expression.type = Boolean*

Branch: Branch On Test Result

IfThenElseStatement ::= if (Expression) StatementNoShortIf else Statement

StatementNoShortIf = StatementNoTrailingSubstatement |LabeledStatementNoShortIf
. = |IfThenElseStatementNoShortIf |WhileStatementNoShortIf
. = |ForStatementNoShortIf



condition *Expression.type = Boolean*

4.4.7 Switch statement

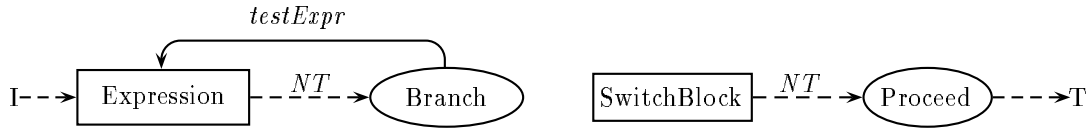
A *SwitchStatement* contains an *Expression* and a *SwitchBlock*. The type of *Expression* must be Char, Byte, Short or Int. Every *SwitchLabel* in *SwitchBlock* must be assignable to the type of *Expression*.

To execute *SwitchStatement*, evaluate *Expression* to get a result *r*. If there is a *SwitchBlockStatementGroup* in the *SwitchBlock* with a *SwitchLabel* of the form `case r :`, pass control to the first of its *BlockStatements*. Otherwise, if there is a *SwitchBlockStatementGroup* with a *SwitchLabel* `default :`, pass control to the first of its *BlockStatements*.

macro *SwitchLabelValue*: $Value \cup Default$

$Node.caseExpr(SwitchLabelValue) : Task$	result task of (constant) expression returning the given switch label value.
$Node.caseTask(SwitchLabelValue) : Task$	initial task associated with the switch label value.

SwitchStatement ::= switch (Expression) SwitchBlock



do-**forall** *val: val.isA?(Value): SwitchBlock.caseTask(val).def?*
Branch.branchTask(val) := SwitchBlock.caseTask(val)
Branch.nextTask :=
*(***if** *SwitchBlock.caseTask(default).def?* **then** *SwitchBlock.caseTask(default)* **else** *Proceed)*

condition *Expression.resultType in {Char,Byte,Short,Int}*
(**forall** *val: val.isA?(Value): SwitchBlock.caseTask(val).def?)*
SwitchBlock.caseExpr(val).assignableTo?(Expression.type)

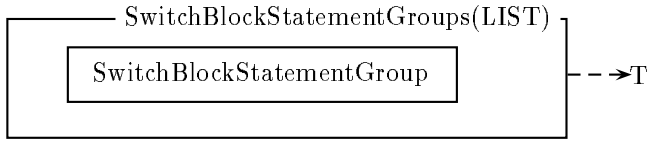
Switch block

A *SwitchBlock* contains a list of *SwitchBlockStatementGroups* and a list of *SwitchLabels*. There is a set of label values associated with each *SwitchBlockStatementGroup*, and these sets must not overlap. The label values in *SwitchLabels* must be distinct from one another and from the label values in *SwitchBlockStatementGroups*.

The *SwitchLabel*s within the *SwitchBlockStatementGroups* list associate label values with statements within the block. The *SwitchLabels* at the end of *SwitchBlock* associate label values with the statement following the block.

Node.switchLabelValue : SwitchLabelValue	Value specified in the switch label node.

SwitchBlock ::= { [**SwitchBlockStatementGroups**] [**SwitchLabels**] }
SwitchBlockStatementGroups = LIST(SwitchBlockStatementGroup)
SwitchLabels = LIST(SwitchLabel)
SwitchLabel = CaseSwitchLabel |DefaultSwitchLabel



Process Each Node n In List SwitchBlockStatementGroups
do-**forall** *val: val.isA?(SwitchLabelValue): n.caseTask(val).def?*
curNode.caseTask(val) := n.caseTask(val)
if *n.caseExpr(val).def?* **then**
curNode.caseExpr(val) := n.caseExpr(val)

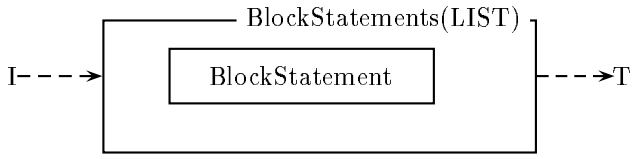
condition *(For All Distinct Members m,n Of List SwitchBlockStatementGroups)*
(**forall** *val: val.isA?(SwitchLabelValue)*
not (m.caseTask(val).def? and n.caseTask(val).def?)
(For All Distinct Members m,n Of List SwitchLabels)
$m.switchLabelValue \neq n.switchLabelValue$
(For Each Member m Of List SwitchBlockStatementGroups)
*(For Each Member n Of List SwitchLabels) m.caseTask(n.switchLabelValue).*undef?

Switch block statement group

A *SwitchBlockStatementGroup* contains a list of *BlockStatements* and a list of *SwitchLabels*. For each label value, the initial statement of the *BlockStatements* is associated with the label.

The labels in *SwitchLabels* must be distinct.

SwitchBlockStatementGroup ::= **SwitchLabels BlockStatements**



do-**forall** *n: n.memberOf?(SwitchLabels)*
curNode.caseTask(n.switchLabelValue) := BlockStatements.initialTask
if *n.isA?(CaseSwitchLabelNode)* **then**
curNode.caseExpr(n.switchLabelValue) := n

condition *(For All Distinct Members m,n Of List SwitchLabels)*
$m.switchLabelValue \neq n.switchLabelValue$

Switch label

A *SwitchLabel* takes one of two forms: either specifying a label value or a default value.

CaseSwitchLabel ::= case ConstantExpression :
ConstantExpression = Expression

ConstantExpression (R)

curNode.switchLabelValue := ConstantExpression.result

DefaultSwitchLabel ::= default :

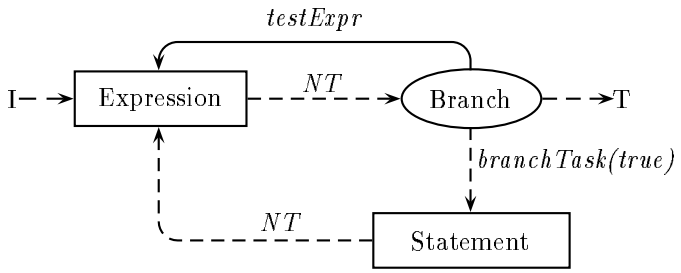
curNode.switchLabelValue := default

4.4.8 While statement

WhileStatement contains an *Expression* and a *Statement*. The type of *Expression* must be Boolean.

To execute *WhileStatement*, evaluate *Expression*. If the result is *true*, execute *Statement* and then repeat execution of *WhileStatement*.

WhileStatement ::= while (Expression) Statement



*do-***forall** *t: Statement.contains?(t) and t.label = default and t.targetTask.*undef*?*
if *t.isA?(BreakTask)* **then**
t.targetTask := branch
elseif *t.isA?(ContinueTask)* **then**
t.targetTask := Expression.initialTask
curNode.continueTarget := Expression.initialTask

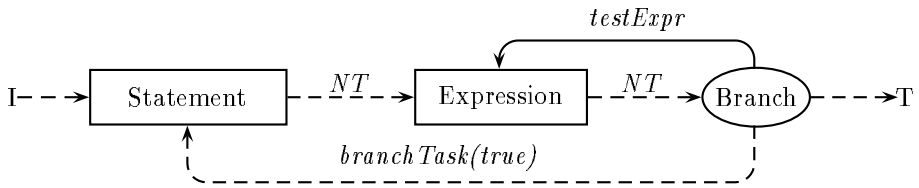
condition *Expression.type = Boolean*

4.4.9 Do statement

DoStatement contains an *Expression* and a *Statement*. The type of *Expression* must be Boolean.

To execute *DoStatement*, execute *Statement* and then evaluate *Expression*. If the result is *true*, repeat execution of *DoStatement*.

DoStatement ::= do Statement while (Expression) ;



do-**forall** *t: Statement.contains?(t) and t.label = default and t.targetTask.*undef*?*
  if *t.isA?(BreakTask)* **then**
    t.targetTask := Branch
  elseif *t.isA?(ContinueTask)* **then**
    t.targetTask := Expression.initialTask
curNode.continueTarget := Expression.initialTask

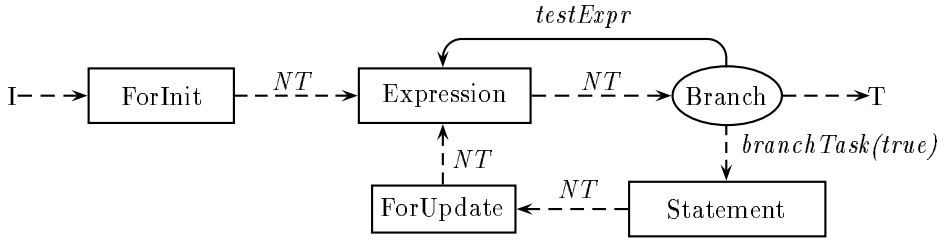
condition *Expression.type = Boolean*

4.4.10 For statement

ForStatement contains an optional *ForInit* block, an optional *Expression*, an optional *ForUpdate* block and a *Statement*. The type of *Expression* must be Boolean. (Any local variables declared within the *ForInit* block are undefined outside the *ForStatement*.)

To execute *ForStatement*, execute *ForInit*. Then evaluate *Expression*; if the result is *true*, execute *Statement*, then execute *ForUpdate*, and repeat execution of *ForStatement*, starting with the evaluation of *Expression*.

ForStatement ::= **for ([ForInit] ; [Expression] ; [ForUpdate]) Statement**
ForInit = StatementExpressionList |LocalVariableDeclarationStatement
StatementExpressionList = LIST(StatementExpression)
ForUpdate = StatementExpressionList



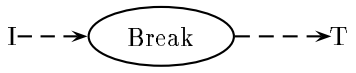
do-**forall** *t: Statement.contains?(t) and t.label = default and t.targetTask.*undef*?*
  if *t.isA?(BreakTask)* **then**
    t.targetTask := Branch
  elseif *t.isA?(ContinueTask)* **then**
    t.targetTask := ForUpdate.initial
curNode.continueTarget := ForUpdate.initialTask
if *ForInit.isA?(LocalVariableDeclarationStatementNode)* **then**
  Process Each Node n In List ForInit.VariableDeclarators
    n.ID.localVarDecl := undef

condition *Expression.type = Boolean*

4.4.11 Break statement

BreakStatement contains an optional *Identifier*. To execute *BreakStatement*, jump toward the target of the break (determined at compile time). If *Identifier* is present, the target is the task immediately after the enclosing statement labeled with *Identifier*. Otherwise, the target is the task immediately after the innermost enclosing *WhileStatement*, *DoStatement* or *ForStatement*.

BreakStatement ::= **break [Identifier] ;**



*Break.label := (***if** *Identifier.def?* **then** *Identifier.ID* **else** *default)*

Break: Jump Toward targetTask.nextTask

4.4.12 Continue statement

ContinueStatement contains an optional *Identifier*. To execute *ContinueStatement*, jump toward the target of the continue (determined at compile time). If *Identifier* is present, the target is the test expression of the enclosing *WhileStatement* or *DoStatement* labeled with *Identifier*, or the update statement of the enclosing *ForStatement* labeled with *Identifier*. Otherwise, the target is the test expression of the

innermost enclosing *WhileStatement* or *DoStatement* or the update statement of the innermost enclosing *ForStatement*.

ContinueStatement ::= continue [Identifier] ;

Continue.label := (**if** *Identifier.def?* **then** *Identifier.ID* **else** *default)*

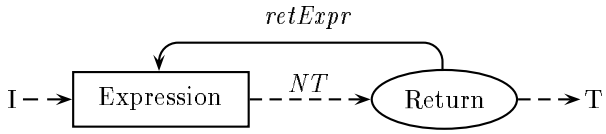
Continue: Jump Toward targetTask

4.4.13 Return statement

ReturnStatement contains an optional *Expression*. To execute *ReturnStatement*, evaluate *Expression* (if it is present) and jump toward the end of the method.

macro Return:
(Jump toward end of method/constructor. Further details in Section 6.)
Jump Toward *endTask*

ReturnStatement ::= return [Expression] ;



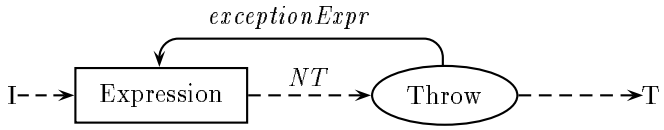
Return: Return

4.4.14 Throw statement

ThrowStatement contains an *Expression*. The type of *Expression* must be a subclass of the class Throwable.

To execute *ThrowStatement*, evaluate *Expression* to get a result *r* and jump toward the initial statement of the catch block appropriate to the class of *Expression*'s result.

ThrowStatement ::= throw Expression ;



condition *Expression.type.subclassOf?(Throwable)*

Throw:
Throw exceptionExpr.result

4.4.15 Synchronized statement

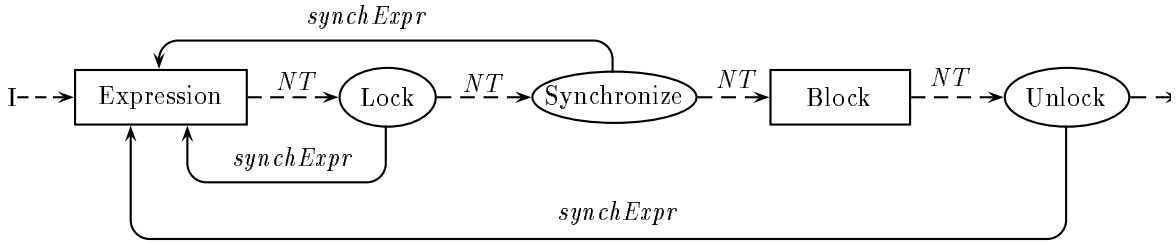
A *SynchronizedStatement* contains an *Expression* and a *Block*. To execute *SynchronizedStatement*, evaluate *Expression*. If the result is null, throw a NullPointerException. Otherwise, issue a lock request for the object returned by *Expression*. Wait until the lock is granted, then execute *Block*. Finally, issue an unlock request for the object. (We treat locks in detail in Section 9.

macro Issue Lock Request:
(Issue lock request for indicated object. Details given in Section 9.)
Proceed Sequentially

macro Synchronize:
(Wait for lock to be granted. Details given in Section 9.)
Proceed Sequentially

macro Issue Unlock Request:
(Issue unlock request. Details given in Section 9.)
Proceed Sequentially

SynchronizedStatement ::= synchronized (Expression) Block



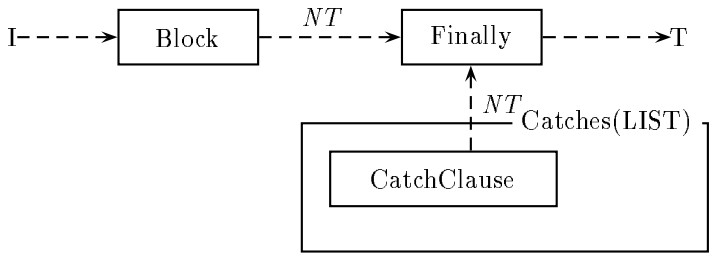
condition *Expression.type.isA?(ReferenceType)*

Lock: Issue Lock Request
Synchronize: Synchronize
Unlock: Issue Unlock Request

4.4.16 Try statement

A *TryStatement* contains a *Block*, an optional list of *Catches* clauses and an optional *Finally* block. To execute *TryStatement*, execute *Block*. (Statements in *Finally* are mandatory. In the case of an exception of class *e* during execution of *Block*, control passes to the leftmost *CatchClause* whose formal parameter type can be assigned *e*. After completion of *Block* or any *CatchClause*, control passes to *Finally*.)

TryStatement ::= try Block [Catches] [Finally]
Catches = LIST(CatchClause)



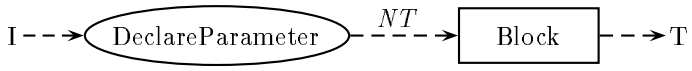
*do-***forall** *t,e: t.isA?(Task) and e.isA?(Class): Block.contains?(t) and e.subclassOf?(Throwable)*
choose n: n.firstInList?(Catches, e = n.type or e.subclassOf?(n.type))
t.catchTarget(e) := n.initialTask
*do-***forall** *t: t.isA?(Task): Block.contains?(t) or Catches.contains?(t)*
if *t.nextMandatoryBlock.*undef*?* **then** *t.nextMandatoryBlock := Finally.initialTask*

condition *not Catches.isPresent? => Finally.isPresent?*

Catch clause

A *CatchClause* contains a *FormalParameter* and a *Block*. The parameter is associated with the block. The declared type of the parameter must be a subclass of Throwable.

CatchClause ::= catch (FormalParameter) Block



curNode.type := FormalParameter.type
DeclareParameter.declaration := FormalParameter.declaration
FormalParameter.ID.localVarDecl := undef

condition *FormalParameter.type.subclassOf?(Throwable)*

DeclareParameter:
extend *Variable* **with** *var*
  declaration.localVar := var
  var.value := curException
curException := undef
Proceed Sequentially

Finally block

A *Finally* block contains a *Block*. All statements in *Block* are mandatory.

Finally ::= finally Block

do-**forall** *t in Task* **with** *Block.contains?(t)*
  t.inMandatoryBlock? := true

4.5 Compile-time checks for method and constructor declarations

At the level of a method or constructor declaration, several compile-time checks of the statements in the declaration body are performed. In addition, some extra control-flow information is added. We provide the details of these compile-time actions.

Method declaration

A *MethodDeclaration* contains a *MethodBody*, which contains the code of the newly declared method m, and a *MethodHeader*, which specifies everything else. The validity of the *MethodBody* and its compatibility with the information in *MethodHeader* is checked as follows.

1. The targets of all break and continue statements in *MethodBody* must be defined somewhere within *MethodBody*.
2. If m is static, there must be no *ThisExpression* or *SuperExpression* within *MethodBody*.
3. If m is a void method, then there must be no *ReturnStatement* in *MethodBody* containing an *Expression*. If m is not void, there must be no *ReturnStatement* in *MethodBody* without an

Expression; moreover, it must be impossible to reach the end of the method through any means other than a jump.

4. For any exception e thrown within *MethodBody*, one of the following must be true.

   (a) e is a subclass of the class RunTimeException;

   (b) e is a subclass of the class Error;

   (c) e is caught within *MethodBody*;

   (d) e is listed as a thrown exception of m.

MethodDeclaration ::= MethodHeader MethodBody

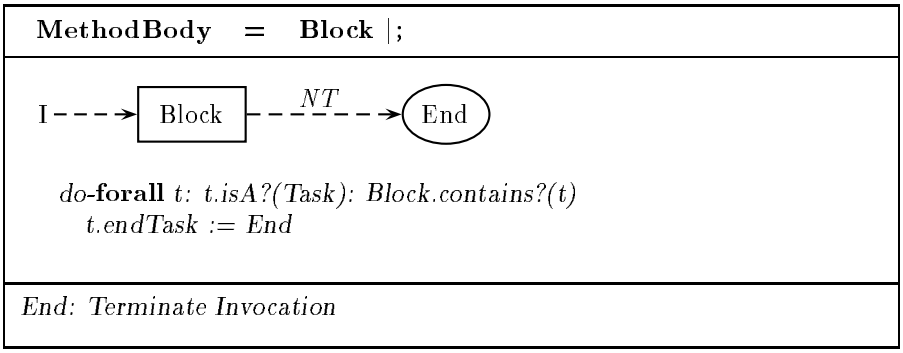
```
let m = MethodHeader.method
  m.firstTask := MethodBody.initialTask
  curNode.method := m
```

condition
```
let m = MethodHeader.method
  MethodBody.isA?(Block) <=> not (m.abstract? or m.native?)
(forall t: MethodBody.contains?(t) and t.isA?(ThrowStatement))
  t.exceptionExpr.type.subclass?(RuntimeException)
  or t.exceptionExpr.type.subclass?(Error)
  or t.catchTarget(exceptionExpr.type).def?
  or m.throws?(exceptionExpr.type)
(forall t: MethodBody.contains?(t) and (t.isA?(BreakTask) or t.isA?(ContinueTask)))
  t.targetTask.def?
m.type.undef? =>
  (forall t: MethodBody.contains?(t) and t.isA?(ReturnTask)) t.retExpr.undef?
m.type.def? =>
  (forall t: MethodBody.contains?(t) and t.isA?(ReturnTask))
    t.retExpr.type = MethodHeader.method.type
    and not (exists t: MethodBody.contains?(t)) t.nextTask = t.endTask
m.static? =>
  not (exists n: MethodBody.contains?(n))
    n.isA?(ThisExpressionNode) or n.isA?(SuperExpressionNode)
```

Method body

A *MethodBody* is either a *Block* (representing the code of the method) or a token **;** (representing an abstract or native method). If the *MethodBody* is a *Block*, then execution of the method begins at the beginning of *Block* and terminates at the end of *Block*.

macro Terminate Invocation:
(Stop execution of the current method. Details given in Section 6.) *curTask* := *undef*



Constructor declaration

A *ConstructorDeclaration* contains a *ConstructorBody*, which contains the code of the constructor c, an optional access status modifier in *ConstructorModifiers*, an optional list of thrown exceptions in *Throws*, and the rest of the constructor's specification in *ConstructorDeclarator*. The *ConstructorBody* is checked for validity and compatibility with the other information about c, as follows.

1. The targets of all break and continue statements in *MethodBody* must be defined somewhere within *MethodBody*.
2. There must be no *ReturnStatement* in *ConstructorBody* containing an *Expression*.
3. For any exception e thrown within *MethodBody*, one of the following must be true.
   (a) e is a subclass of the class RunTimeException;
   (b) e is a subclass of the class Error;
   (c) e is caught within *MethodBody*;
   (d) e is listed as a thrown exception of m.

ConstructorDeclaration	**::=**	**[ConstructorModifiers] ConstructorDeclarator [Throws] ConstructorBody**
ConstructorModifiers	=	LIST(ConstructorModifer)
ConstructorModifier	=	public |protected |private
ConstructorBody	=	ConstructorBodyNoInvocation
.	=	|ConstructorBodyWithInvocation

let c = ConstructorDeclarator.constructor
curNode.constructor := c
Assign ConstructorModifiers To c
*do-***forall** *e: e.isA?(Class): Throws.throws?(e)*
c.throws?(e) := true
c.firstTask := ConstructorBody.initial

condition *ConstructorModifiers.consistent?*
(forall *t: ConstructorBody.contains?(t) and t.isA?(ThrowStatement))*
t.exceptionExpr.type.subclass?(RuntimeException)
or t.exceptionExpr.type.subclass?(Error)
or t.catchTarget(exceptionExpr.type).def?
or m.throws?(exceptionExpr.type)
(forall *t: ConstructorBody.contains?(t) and*
(t.isA?(BreakTask) or t.isA?(ContinueTask)))
t.targetTask.def?
(forall *t: MethodBody.contains?(t) and t.isA?(ReturnTask)) t.retExpr.*undef*?*

Constructor body

A *ConstructorBody* is either a *MethodBody* (containing a list of *BlockStatements* or it contains an *ExplicitConstructorInvocation* followed by a list of *BlockStatements*. If it is of the form *MethodBody*, execution proceeds as follows. First the default constructor of the current class's parent is invoked. The default constructor for a class takes no arguments. Next, the initializer expressions for the non-static fields declared in the current class are evaluated in order and assigned to the target object's fields. Then the *MethodBody* is executed.

An *ExplicitConstructorInvocation* is of the form `this` `(` [*ArgumentList*] `)` (an invocation of a constructor declared in the current class) or of the form `super` `(` [*ArgumentList*] `)` (an invocation of a constructor declared in the parent of the current class). To execute a *ConstructorBody* with an *ExplicitConstructorInvocation*, first evaluate the expressions in *ArgumentList*, then invoke the appropriate constructor. If the constructor invocation is of the form `super` `(` [*ArgumentList*] `)`, evaluate the initializer expressions for the non-static fields declared in the current class and assign them to the target object's fields. Finally, execute *BlockStatements*.

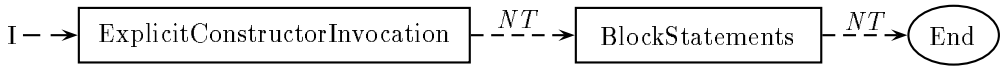
macro Initialize Fields:
(Assign initial values to the fields of the new class instance. Details given in Section 7.)
Proceed Sequentially

ConstructorBodyNoInvocation = MethodBody

*do-***forall** *t: t.isA?(Task) and MethodBody.contains?(t)*
t.endTask := End

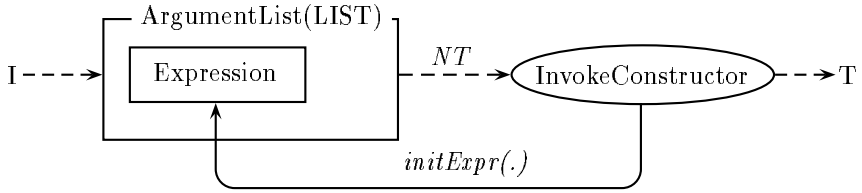
InvokeConstructor: Invoke Constructor
InitFields: Initialize Fields

ConstructorBodyWithInvocation ::= { ExplicitConstructorInvocation [BlockStatements] }
ExplicitConstructorInvocation = ThisConstructorInvocation
. = |SuperConstructorInvocation

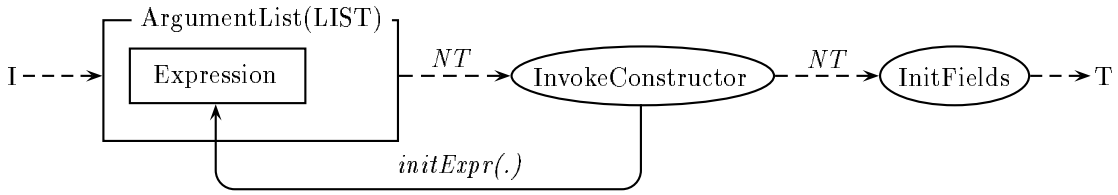


*do-***forall** *t: t.isA?(Task) and MethodBody.contains?(t)*
t.endTask := End

ThisConstructorInvocation ::= this ([ArgumentList]) ;



SuperConstructorInvocation ::= super ([ArgumentList]) ;



5 Evaluation of expressions

Like statements, expressions are defined recursively; expressions may be built from subexpressions. Evaluating an expression involves evaluating its subexpressions and computing a result based on the results of

its subexpressions. This ASM represents expression evaluations as multi-step procedures and details how expression results are computed.

5.1 Preliminaries

Java uses *strong type checking* to ensure the integrity of expression evaluation. Each expression that returns a result as part of its evaluation has a type, and the set of possible results returned by an expression is restricted to the set of instances of its type. An expression that returns no result is called *void* and has no type. Such an expression cannot be part of a larger expression, as any subexpression must return a result to its superexpression. Hence a void expression can only serve as an expression statement.

The type of an arithmetic expression (e.g. negation, multiplication) depends on the types of its operands. *Numeric promotion* determines the type of an arithmetic expression given the (numeric) types of its subexpressions. It may also require that the results of the operands be converted to the promoted type. *Unary promotion* is applied to an expression with a single operand. If the type of the operand is Byte, Short or Int, the promoted type is Int; otherwise, the promoted type is the type of the operand. Binary promotion is applied to an expression with two operands. If either operand is of type Double, the promoted type is Double; otherwise, if either operand is of type Float, the promoted type is Float; otherwise, if either operand is of type Long, the promoted type is Long; otherwise, the promoted type is Int.

Certain expressions access variables to derive their results. An expression name accesses a local variable, parameter variable, or field variable; a field access accesses a field variable; an array access accesses an array variable. Depending on the context, such expressions may be required to return either the value stored in the variable, or the variable itself. In the latter case, the expression is called an *lvalue* expression. An lvalue expression is used in an assignment; the returned variable is assigned a new value.

The evaluation of certain expressions may result in thrown exceptions. The creation of a class instance or array or the concatenation of strings requires allocation of memory for new objects. If the allocation is unsuccessful due to a lack of available memory, an OutOfMemoryException is thrown. The creation of an array may also result in a NegativeArraySizeException, if an array of negative size is requested. A field or array access or method invocation results in a NullPointerException if the accessee is null. An array access beyond the bounds of an array results in an IndexOutOfBoundsException. A ClassCastException occurs in a cast expression which attempts a conversion that is found to be illegal at runtime. A division or remainder operation that attempts to divide by zero throws an ArithmeticException. Finally, an assignment to an array component results in an ArrayStoreException if the assignment conversion is found to be illegal at runtime.

5.2 ASM J_2

M_2 consists of the montages of M_1 plus the montages defined later in this section. The ASM J_2 has two agents: The *compiler* with module C_2 (compile-time rules of M_2) and the *executor* with module R_2 (runtime rules of M_2).

5.3 Function and macro definitions

$targetObject : Object$	target object of the current method.
$Task.lvalue? : Boolean$	is this the result task of an lvalue expression?
$Task.rightTask : Task$	initial task of operator's right subexpression.
$Task.subexpr : Task$	result task of operator's (unique) subexpression.
$Task.leftExpr : Task$	result task of operator's left subexpression.
$Task.rightExpr : Task$	result task of operator's right subexpression.
$Task.argExpr(Nat) : Task$	result task of argument expression with given index.

macro Get New Memory R:
(For expression task that involves allocation of memory,
either execute task rule R or throw OutOfMemoryException.)
choose among
    R
    Throw Exception Of Class $OutOfMemoryException$

macro $A.unaryPromotion$:
(Promoted type, given single operand A.)
(if $A.type \in \{Byte, Short, Char\}$ then Int else $A.type$)

macro $binaryPromotion(A, B)$:
(Promoted type, given two operands A and B.)
(if $A.type = Double$ or $B.type = Double$ then $Double$
 elseif $A.type = Float$ or $B.type = Float$ then $Float$
 elseif $A.type = Long$ or $B.type = Long$ then $Long$
 else Int)

macro Access VAR:
(Access the variable VAR. If current expression is an lvalue, the result is the variable itself;
otherwise, it is the value of the variable.)
if $lvalue$? **then**
    $result := VAR$
    Proceed Sequentially
else
    $result := VAR.value$
    Proceed Sequentially

macro Assign VAL To VAR:
(Assign the value VAL to the variable VAR.)
$VAR.value := VAL$

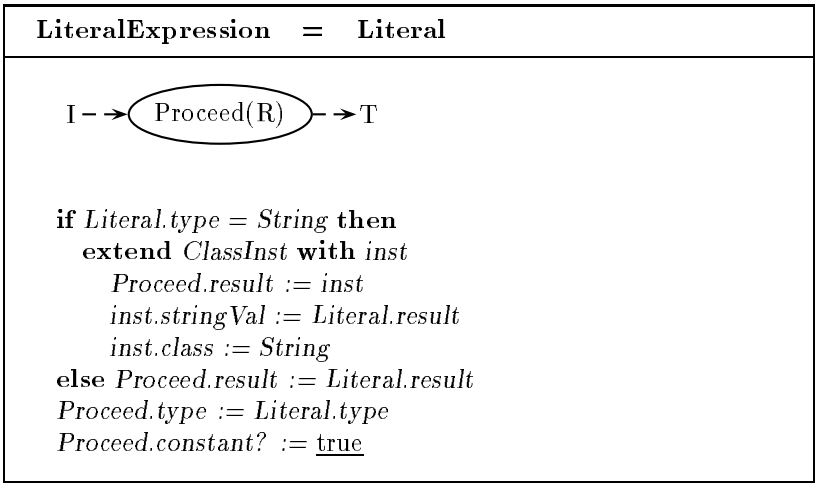
5.4 Construction of expressions

For each type of expression, we describe the actions performed when evaluating an expression of that type, and how to construct a representation of the expression in the initial state. We delay a detailed treatment of method invocation expressions to Section 6. Since certain groups of expression types behave similarly to one another, we do not give montages for all expression types. For a group of related expression types, we give a montage for one representative example and simply give the production rules for the other expression types.

5.4.1 Primary expression

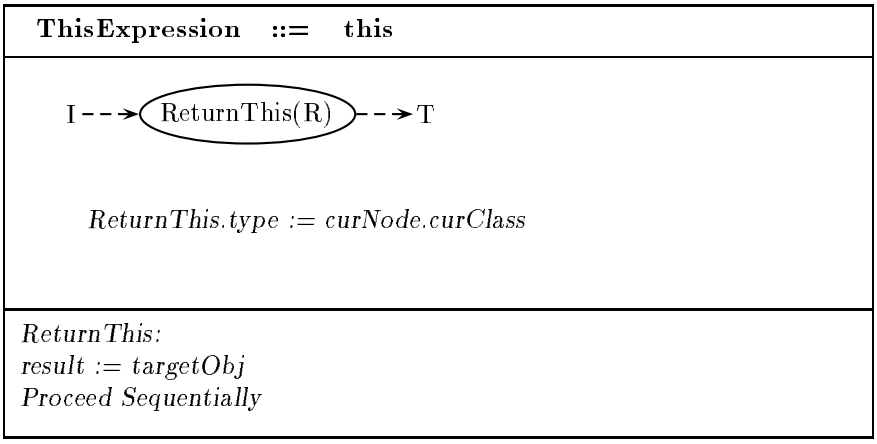
Literal expression

A *LiteralExpression* contains a *Literal*, which denotes a fixed value. A *LiteralExpression* is never evaluated; its result is the value of the *Literal*, which is computed at compile time. The type of the *LiteralExpression* is the type of the *Literal*.



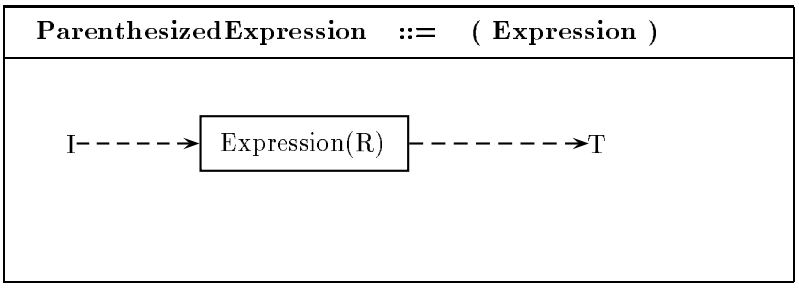
This expression

To evaluate *ThisExpression*, return the target object of the method. The type of the expression is the class in which the expression appears.



Parenthesized expression

A *ParenthesizedExpression* contains an *Expression*. To evaluate *ParenthesizedExpression*, evaluate *Expression* and return its result. The type of the *ParenthesizedExpression* is the type of *Expression*.

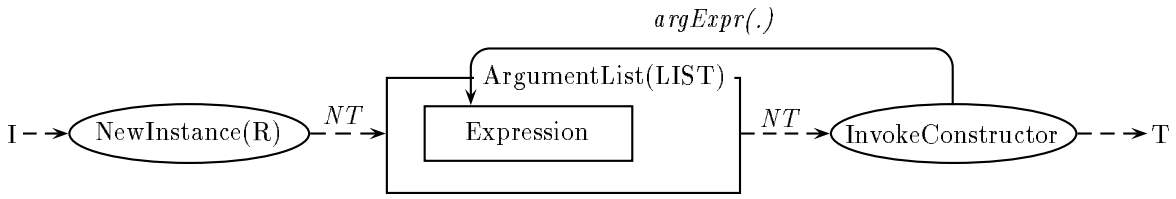


Class instance creation expression

A *ClassInstanceCreationExpression* contains a *ClassType* and an *ArgumentList*, a list of *Expressions*. The class given by *ClassType* must not be abstract.

To evaluate *ClassInstanceCreationExpression*, create an object of the class denoted by *ClassType*. If there is insufficient space to allocate the object, throw an OutOfMemoryError. Create instances of all the fields of the class and initialize them to their default values. Next, evaluate the *Expressions* in *ArgumentList* from left to right, and then invoke the appropriate constructor with the results of *ArgumentList* as arguments. Finally, return the newly created object. The type of the *ClassInstanceCreationExpression* is the class denoted by *ClassType*.

ClassInstanceCreationExpression **::=** **new ClassType ([ArgumentList])**
ArgumentList = LIST(Expression,)



NewInstance.type := ClassType.type

condition *not ClassType.type.abstract?*

NewInstance:
Get New Memory
extend *ClassInst* **with** *obj*
obj.class := type
result := obj

Array creation expression

An *ArrayCreationExpression* contains an *ArrayComponentType*, a list *DimExprs* of dimension size expressions, and a list *Dims* of `[ ]` tokens. Each expression in *DimExprs* must be of type Byte, Short or Int.

Let m be the number of expressions in *DimExpr*, let n be the number of tokens in *Dims*, and let t be the type denoted by *ArrayComponentType*. To evaluate *ArrayCreationExpression*, first evaluate the dimension size expressions in *DimExprs* from left to right, performing unary numeric promotion on each. Then check that each of the resulting values $dimsize_1 \ldots dimsize_m$ is nonnegative, from left to right. If a dimension size is negative, throw a NegativeArraySizeException. Next, create an array object a. The type of a is $t[\,]^{m+n}$. Then for each dimension i, set the sizes of each i-dimension subarray of a to $dimsize_i$. If the elements of the ith dimension are of array type, create enough objects or arrays to give each i-dimension subarray $dimsize_i$ unique components, and initialize these components to their default values. Return the new array object a. The type of the *ArrayCreationExpression* is $t[\,]^{m+n}$.

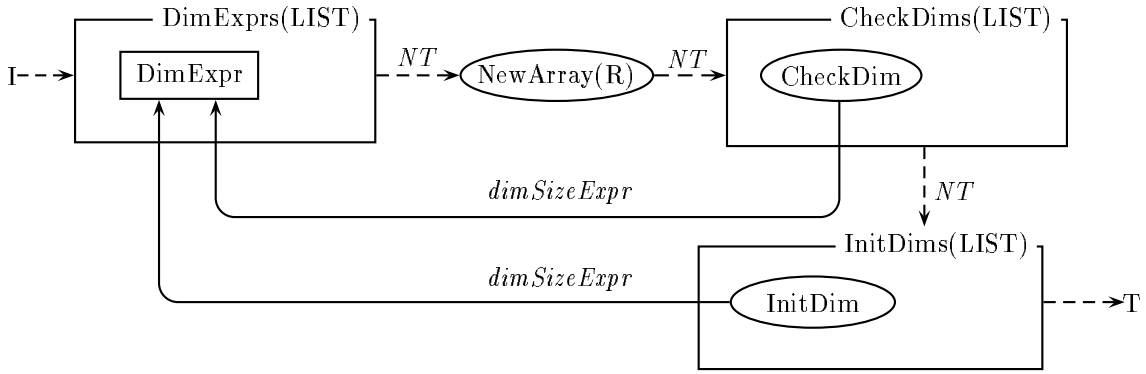
Each expression in *DimExprs* must be of Byte, Short or Int type.

$Task.dimSizeExpr : Task$	Task which returns size of given dimension.
$Task.previousDim : Task$	Creation task for next higher dimension.
$Task.numNewComponents : Nat$	Number of array components created by this task.
$Task.newArray(Nat) : Array$	Array with given index created by this task.
$Task.createDefaultComponents? : Boolean$	Shall default values be assigned to the components of this dimension?

macro Create $DIMSIZE$ Components For $NUMARRAYS$ Arrays:
(For each of the $NUMARRAYS$ arrays created by the previous dimension-initializer task,
initialize $DIMSIZE$ components. This means either assigning default values for the new components,
or assigning arrays as components (in which case the component arrays
must in turn be initialized by a later dimension-initializer task).)

```
if createDefaultComponents? then
    do-forall i : i < NUMARRAYS
        previousDim.newArray(i).length := DIMSIZE
        do-forall j : j < DIMSIZE
            extend Variable with v
                previousDim.newArray(i).component(j) := v
                v.value := type.defaultValue
else
    numNewComponents := NUMARRAYS * DIMSIZE
    do-forall i : i < NUMARRAYS
        previousDim.newArray(i).length := DIMSIZE
        do-forall j : j < DIMSIZE
            extend Variable with v, Array with a
                previousDim.newArray(i).component(j) := v
                v.value := a
                newArray(DIMSIZE * i + j) := a
```

ArrayCreationExpression ::= new ArrayComponentType DimExprs [Dims]
ArrayComponentType = PrimitiveType |TypeName

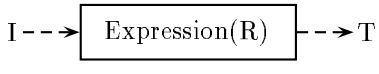


let numdims = DimExprs.length + Dims.length
NewArray.type := ArrayComponentType.type.array(numdims)
Process Each Node t in List InitDims
*t.previousDim := (***if** *t.position = 1* **then** *NewArray* **else** *InitDims[t.position - 1])*
InitDims[InitDims.length].createDefaultComponents? := true
InitDims[InitDims.length].type := ArrayComponentType.type.array(Dims.length)

condition *(For Each Member n Of List DimExprs) n.type in {Byte,Short,Int}*

NewArray:
Get New Memory
let dimsize = dimSizeExpr.result.convertTo(Int)
extend *Array* **with** a
result := a
numNewComponents := 1
newArray(0) := a
CheckDim:
if *dimSizeExpr.result.convertTo(Int) < 0* **then**
Throw Exception Of Class NegativeArraySizeException
else *Proceed Sequentially*
InitDim:
Get New Memory
let dimsize = dimSizeExpr.result.convertTo(Int)
let numarrays = previousDim.numNewComponents
Create dimsize Components For numarrays Arrays

DimExpr ::= [Expression]



Field access expression

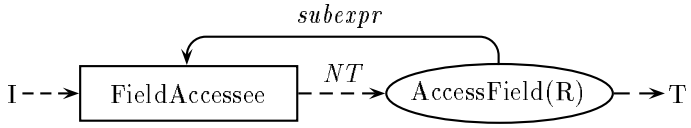
A *FieldAccess* contains a *FieldAccessee* and an *Identifier*. Let *id* be the identifier specified in *Identifier*, and let *t* be the type of *FieldAccessee*. *t* must be a reference type, and *id* must identify an accessible field in *t*.

To evaluate *FieldAccess*, evaluate *FieldAccessee*, then return the value of the field of the resulting object, given the identifier *id* and type *t*. The type of *FieldAccess* is the type of the field of *t*, given the identifier *id* and type *t*.

VariableDeclaration.staticVar : *Variable*	Variable associated with the static field.
Object.fieldVar(*VariableDeclaration*) : *Variable*	Variable associated with the given object's instance field.
Task.finalFieldAccess? : *Boolean*	Is this task an access of a final field?
Task.staticFieldAccess? : *Boolean*	Is this task an access of a static field?
Task.field : *VariableDeclaration*	Field to access for the given field access task.

macro *F.accessibleField*?(*T*):
(Is the field *F* accessible in the current class, given field accessee task *T*?)
F.def? and (*F.accessStatus* = *private* ⇒ *curClass* = *F.declarer*)
and (*F.accessStatus* = *protected* ⇒ (*curClass* = *F.declarer* or (*curClass.subclassOf*?(*F.declarer*)
and (*T.SuperExpression.isPresent*? or *T.type* = *curClass* or *T.type.subclassOf*?(*curClass*)))))

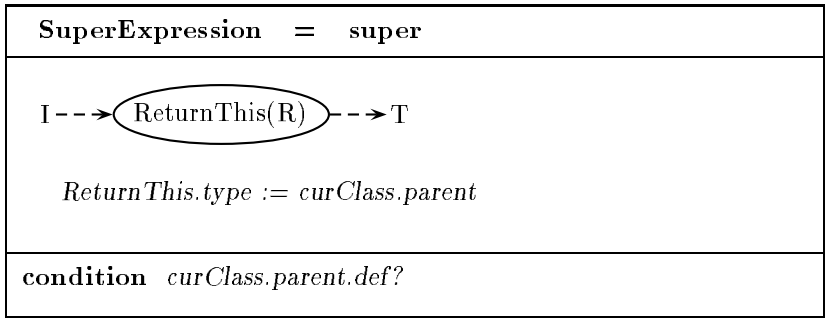
FieldAccess	**::=**	**FieldAccessee . Identifier**
FieldAccessee	=	Primary |SuperExpression
Primary	=	PrimaryNoNewArray |ArrayCreationExpression
PrimaryNoNewArray	=	LiteralExpression |ThisExpression |ParenthesizedExpression
.	=	|ClassInstanceCreationExpression |FieldAccess |MethodInvocation
.	=	|ArrayAccess



AccessField.ID := *Identifier.ID*
let f = *FieldAccessee.type.field*(*Identifier.ID*,*FieldAccessee.type*)
AccessField.type := *f.type*
AccessField.field := *f*
AccessField.finalFieldAccess? := *f.final*?
AccessField.staticFieldAccess? := *f.static*?

condition *let f* = *FieldAccessee.type.field*(*Identifier.ID*,*FieldAccessee.type*)
f.accessibleField?(*FieldAccessee*)

AccessField:
if (*subexpr.result* = *null and not staticFieldAccess*?) **then**
Throw Exception Of Class NullPointerException
else *let var* = (**if** *staticFieldAccess*? **then** *field.staticVar* **else** *subexpr.result.fieldVar*(*field*))
Access var



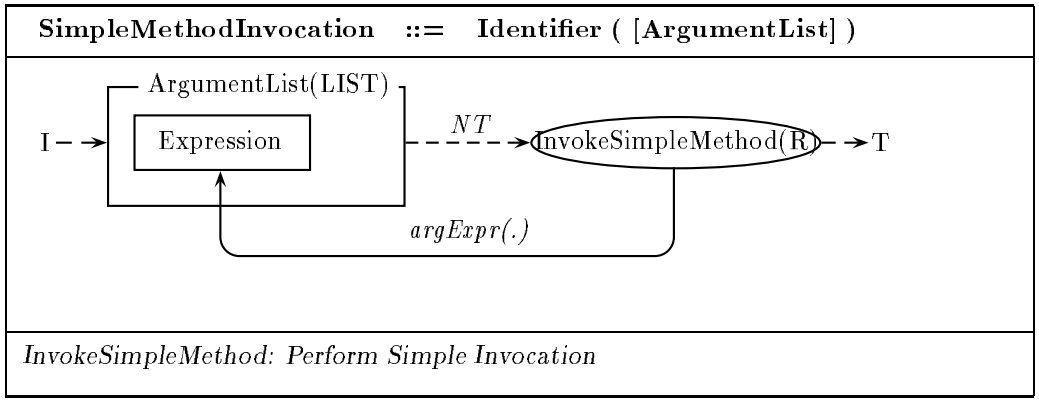
Method invocation expression

A *MethodInvocation* takes one of the following forms:

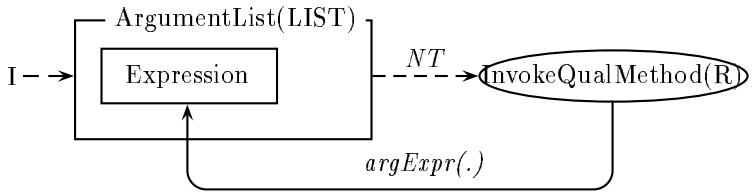
1. *Identifier* ([*ArgumentList*]). To evaluate, first evaluate the expressions in *ArgList*, then invoke the appropriate method.
2. *TypeName* . *Identifier* ([*ArgumentList*]). To evaluate, first evaluate the expressions in *ArgList*, then invoke the appropriate method.
3. *Invoker* . *Identifier* ([*ArgumentList*]), where *Invoker* is an *ExpressionName*, *Primary* or *SuperExpression*. To evaluate, first evaluate *Invoker*, then the expressions in *ArgumentList*, and finally invoke the appropriate method.

The "appropriate method" to invoke is defined in Section 6.

macro Perform Invocation:
(Choose and invoke the appropriate method. Details in Section 6.)
Proceed Sequentially

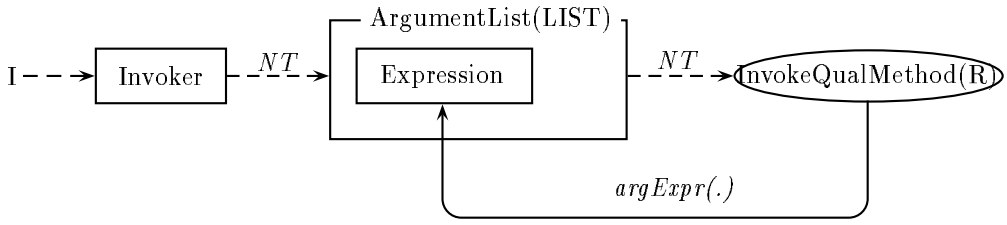


QualifiedMethodInvocation1 ::= TypeName . Identifier ([ArgumentList])



InvokeQualMethod: Perform Qualified Invocation

QualifiedMethodInvocation2 ::= Invoker . Identifier ([ArgumentList])
Invoker = ExpressionName |Primary |SuperExpression



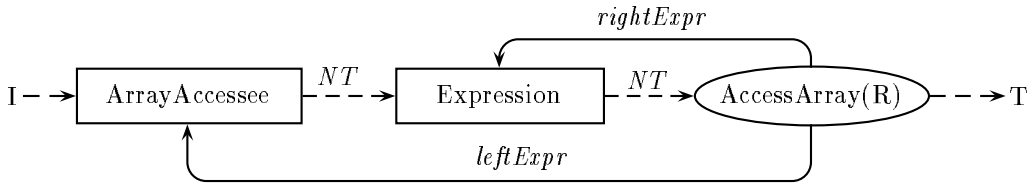
Array access expression

An *ArrayAccess* contains an *ArrayAccessee* and an *Expression*. The type of *ArrayAccessee* must be an array type, and the type of *Expression* must be Byte, Short or Int.

To evaluate *ArrayAccess*, evaluate the *ArrayAccessee* to get an array a, then evaluate *Expression* to get an integer i. Next, if a is null, throw a NullPointerException; otherwise, if i is negative or greater than or equal to the length of a, throw a IndexOutOfBoundsException. Finally, return the ith component of array a. The type of *ArrayAccess* is the component type of *ArrayAccessee*.

Task.checkAssignability? : *Boolean*	Must an assignment expression using this array access perform a runtime type check?

ArrayAccess ::= ArrayAccessee [Expression]
ArrayAccessee = ExpressionName |PrimaryNoNewArray



AccessArray.type := ArrayAccessee.type.componentType
AccessArray.checkTypes? := (ArrayAccessee.type.componentType.isA?(ReferenceType))

condition *ArrayAccessee.type.isA?(Array)*
Expression.type in {Byte,Short,Int}

AccessArray:
let arr = leftExpr.result
if *arr = null* **then** *Throw Exception Of Class NullPointerException*
else *let ind = rightExpr.result.convertTo(Int)*
if *ind < 0 or ind >= arr.length* **then** *Throw Exception Of Class IndexOutOfBoundsException*
else *let var = arr.component(ind)*
Access var

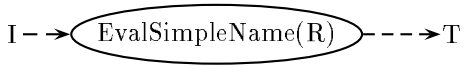
5.4.2 Postfix Expression

Expression name

An *ExpressionName* may take one of three forms:

- The simple form *Identifier*. Let *id* be the identifier specified by *Identifier*. If the *ExpressionName* appears within the scope of a parameter or local variable with identifier *id*, return that variable; the type of the expression is the type of the variable or parameter. If there is no local variable or parameter in scope with identifier *id*, then access the field of the target object with that name; the type of the expression is the type of the field. There must be either a local variable or parameter or field with identifier *id*.
- The form *TypeName.Identifier*. Let *id* be the identifier specified by *Identifier*, and let *t* be the class or interface type denoted by *TypeName*. Return the value of the static field of *t* named by *id*; the type of the expression is the type of the field. There must be a static field with that identifier, and it must be static and accessible.
- The form *ExpressionName.Identifier*. Let *id* be the identifier specified by *Identifier*. Return the field named by *id* of the class instance returned by *ExpressionName*.

SimpleExpressionName = Identifier

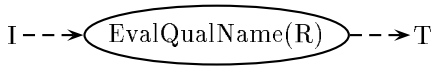


EvalSimpleName.ID := Identifier.ID
if *Identifier.ID.localVarDecl.def?* **then**
  EvalSimpleName.varReference? := true
  EvalSimpleName.type := Identifier.ID.localVarDecl.type
else
  EvalSimpleName.varReference? := false
  let f = curClass.field(Identifier.ID,curClass)
    EvalSimpleName.type := f.type
    EvalSimpleName.field := f
    if *f.final?* **then**
      EvalSimpleName.finalFieldAccess? := true
      if *f.initExpr.constant?* **then**
        EvalSimpleName.constant? := true
        EvalSimpleName.result := f.initExpr.result
    EvalSimpleName.staticFieldAccess? := f.static?

condition *(not Identifier.ID.localVarDecl.def?) =>*
curClass.field(Identifier.ID,curClass).accessibleField?(curClass)

EvalSimpleName:
if *varReference?* **then** *Access ID.localVarDecl.localVar*
elseif *staticFieldAccess?* **then** *Access field.staticVar*
else *Access targetObj.fieldVar(field)*

QualifiedExpressionName1 ::= TypeName . Identifier

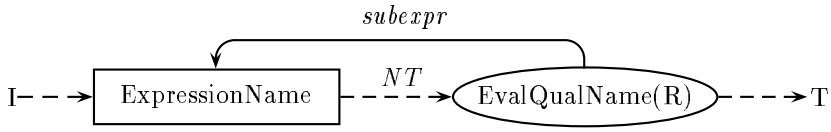


EvalQualName.ID := Identifier.ID
let f = TypeName.type.field(Identifier.ID,TypeName.type)
EvalQualName.type := f.type
EvalQualName.field := f
if *f.final?* **then**
EvalQualName.finalFieldAccess? := true
if *f.initExpr.constant?* **then**
EvalQualName.constant? := true
EvalQualName.result := f.initExpr.result
EvalQualName.staticFieldAccess? := true

condition *let f = TypeName.type.field(Identifier.ID,TypeName.type)*
f.static?
f.accessibleField?(TypeName)

EvalQualName:
if *(subexpr.result = null and not staticFieldAccess?)* **then**
Throw Exception Of Class NullPointerException
elseif *staticFieldAccess?* **then** *Access field.staticVar*
else *Access subexpr.result.fieldVar(field)*

QualifiedExpressionName2 ::= ExpressionName . Identifier
expressionName = SimpleExpressionName |QualifiedExpressionName1
. = |QualifiedExpressionName2



EvalQualName.ID := Identifier.ID
let f = ExpressionName.type.field(Identifier.ID,ExpressionName.type)
EvalQualName.type := f.type
EvalQualName.field := f
EvalQualName.finalFieldAccess? := f.final?
EvalQualName.staticFieldAccess? := f.static?

condition *let f = ExpressionName.type.field(Identifier.ID,ExpressionName.type)*
f.accessibleField?(ExpressionName)

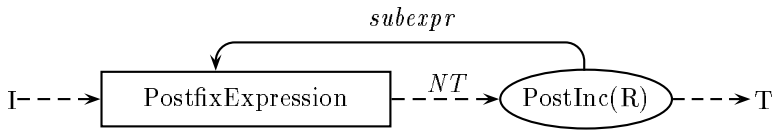
Postfix increment/decrement expressions

Postfix increment expression

PostIncrementExpression contains a *PostfixExpression*. The *PostfixExpression* must be of numeric type, must not return a field that is final, and must be able to return an lvalue. Binary numeric promotion is performed on the result of *PostfixExpression* and the (Int) constant 1. The type of *PostfixIncrementExpression* is the binary promotion of the type of *PostfixExpression* and the type (Int) of 1.

To evaluate *PostIncrementExpression*, add 1 to the promoted value of the variable returned by *PostfixExpression*, store the incremented result in the variable, and return the variable's value before the increment.

PostIncrementExpression	**::=**	**PostfixExpression ++**
PostfixExpression	=	Primary |ExpressionName
.	=	|PostIncrementExpression |PostDecrementExpression



PostfixExpression.lvalue? := true
PostInc.type := PostfixExpression.type
PostInc.operandType :=
*(***if** *PostfixExpression.resultType in {Long,Float,Double}* **then** *PostfixExpression.type* **else** *Int)*

condition *PostfixExpression.type.numeric?*
not PostfixExpression.final?
PostfixExpression in {FieldAccess,ArrayAccess,ExpressionName}

PostInc:
Assign (subexpr.result.value.convertTo(operandType) + 1).convertTo(resultType)) To subexpr.result
result := subexpr.result.value
Proceed Sequentially

Postfix decrement expression

PostDecrementExpression ::= PostfixExpression --

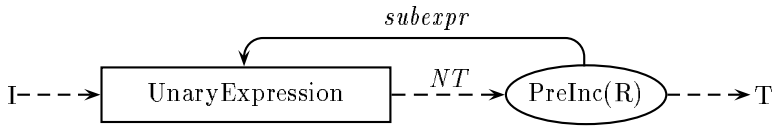
5.4.3 Unary expression

Prefix increment expression

PreIncrementExpression contains a *UnaryExpression*. The *UnaryExpression* must be of numeric type, must not return a field that is final, and must be able to return an lvalue. Binary numeric promotion is performed on the result of *UnaryExpression* and 1. The type of *PreIncrementExpression* is the binary promotion of the type of *UnaryExpression* and the type of 1.

To evaluate *PreIncrementExpression*, add 1 to the promoted value of the variable returned by *UnaryExpression*, store the incremented result in the variable, and return the variable's incremented value.

PreIncrementExpression	**::=**	**++ UnaryExpression**
UnaryExpression	=	PreIncrementExpression |PreDecrementExpression
.	=	|PositiveExpression |NegativeExpression
.	=	|PostfixExpression |BitwiseComplementExpression
.	=	|LogicalComplementExpression |CastExpression



UnaryExpression.lvalue? := true
PreInc.type := UnaryExpression.type
PreInc.operandType :=
*(***if** *UnaryExpression.type in {Long,Float,Double}* **then** *UnaryExpression.type* **else** *Int)*

condition *UnaryExpression.type.numeric?*
not UnaryExpression.final?
UnaryExpression in {FieldAccess,ArrayAccess,ExpressionName}

PreInc:
Assign (subexpr.result.value.convertTo(operandType) + 1).convertTo(type)) To subexpr.result
result := (subexpr.result.value.convertTo(operandType) + 1).convertTo(type)
Proceed Sequentially

Prefix decrement expression

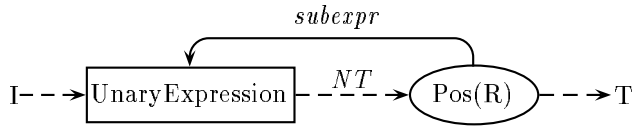
PreDecrementExpression ::= UnaryExpression --

Positive expression

A *PositiveExpression* contains a *UnaryExpression*. The type of the *UnaryExpression* must be numeric. Unary promotion is performed on the result of *UnaryExpression*. The type of *PositiveExpression* is the unary promotion of the type of *UnaryExpression*.

To evaluate *PositiveExpression*, return the promoted result of *UnaryExpression*.

PositiveExpression ::= + UnaryExpression




```
let t = UnaryExpression.unaryPromotion
   Pos.type := t
   if UnaryExpression.constant? then
      Pos.constant? := true
      Pos.result := UnaryExpression.result.convertTo(t)
```

condition *UnaryExpression.type.numeric?*

Pos:
result := subexpr.result.convertTo(type)
Proceed Sequentially

Negative expression

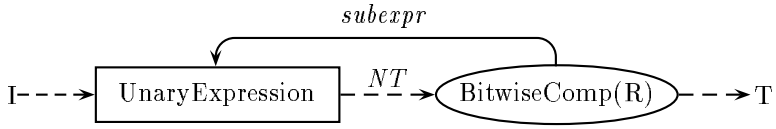
NegativeExpression ::= - *UnaryExpression*

Bitwise complement expression

A *BitwiseComplementExpression* contains a *UnaryExpression*. The type of the *UnaryExpression* must be integral. Unary numeric promotion is performed on the result of *UnaryExpression*. The type of *BitwiseComplementExpression* is the unary promotion of the type of *UnaryExpression*.

To evaluate *BitwiseComplementExpression*, evaluate *UnaryExpression*, then return the bitwise complement of the result of *UnaryExpression*. For a result x of *UnaryExpression*, *BitwiseComplementExpression* returns $(-x) - 1$.

BitwiseComplementExpression ::= ~ UnaryExpression




```
let t = UnaryExpression.unaryPromotion
   BitwiseComp.type := t
   if UnaryExpression.constant? then
      BitwiseComp.constant? := true
      BitwiseComp.result := -(UnaryExpression.result.convertTo(t)) - 1
```

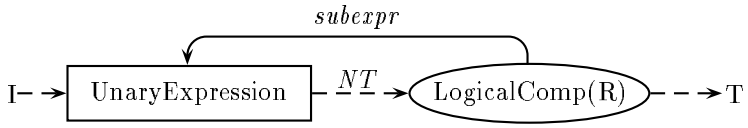
BitwiseComp:
result := -(subexpr.result.convertTo(resultType)) - 1
Proceed Sequentially

Logical complement expression

A *LogicalComplementExpression* contains a *UnaryExpression*. The type of the *UnaryExpression* must be Boolean. The type of *LogicalComplementExpression* is Boolean.

To evaluate *LogicalComplementExpression*, return the logical complement of the result of *UnaryExpression* (*true* if *UnaryExpression* returns *false*; *false* if *UnaryExpression* returns *true*).

LogicalComplementExpression ::= ! UnaryExpression



LogicalComp.type := *Boolean*
if *UnaryExpression.constant?* **then**
LogicalComp.constant? := true
LogicalComp.result := *not UnaryExpression.result*

condition *UnaryExpression.type* = *Boolean*

LogicalComp:
result := *not subexpr.result*
Proceed Sequentially

Cast expression

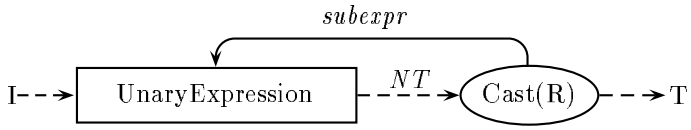
A *CastExpression* contains a *Type* and a *UnaryExpression*. Let s be the type of *UnaryExpression* and let t be the type specified by *Type*. s must be castable to t. This is true if s is equal to or narrower than t, or if t is narrower than s. It is also true if a narrower-than relationship may hold between subclasses or subinterfaces of s and t. The type of *CastExpression* is t.

To evaluate *CastExpression*, evaluate *UnaryExpression*. If the type of *UnaryExpression* is assignable to t or the result of *UnaryExpression* is *null*, convert the result of *UnaryExpression* to t and return the result. Otherwise, check whether the class of the result of *UnaryExpression* is assignable to t. If it is, convert the result of *UnaryExpression* to t and return the result; otherwise, throw a ClassCastException.

Task.checkCastability? : *Boolean*	Must a runtime check be performed on the result before casting?

macro $S.castableTo?(T)$
$S.numDims = T.numDims$
and ($S.baseType = T.baseType$
or $S.baseType.narrowerThan?(T.baseType)$ or $T.baseType.narrowerThan?(S.baseType)$
or ($S.baseType.isA?(Class)$ and $T.baseType.isA?(Interface)$ and not $S.baseType.final?$)
or ($T.baseType.isA?(Class)$ and $S.baseType.isA?(Interface)$ and not $T.baseType.final?$)
or ($S.baseType.isA?(Interface)$ and $T.baseType.isA?(Interface)$
and ($\forall id, t : id.isA?(IDString)$ and $t.isA?(TypeSeq)$:
$S.baseType.method(id,t).def?$ and $T.baseType.method(id,t).def?$)
$S.baseType.method(id,t).type = T.baseType.method(id,t).type$

CastExpression ::= (Type) UnaryExpression




```
let casttype = Type.type
  Cast.type := casttype
  Cast.checkCastability? := not UnaryExpression.type.assignableTo?(casttype)
  if (casttype.isA?(PrimitiveType) or casttype = String) then
    Cast.constant? := true
    Cast.result := UnaryExpression.result.convertTo(casttype)
```

condition *UnaryExpression.type.castableTo?(Type.type)*
Type.type.baseType.isA?(PrimitiveType) =>
not (UnaryExpression.isA?(PreIncrementExpression)
or UnaryExpression.isA?(PreDecrementExpression)
or UnaryExpression.isA?(PositiveExpression)
or UnaryExpression.isA?(NegativeExpression))

Cast:
if *checkCastability?*
and not subexpr.result.class.assignableTo?(type) and subexpr.result $\neq$ *null* **then**
Throw Exception Of Class ClassCastException
else
result := subexpr.result.convertTo(type)
Proceed Sequentially

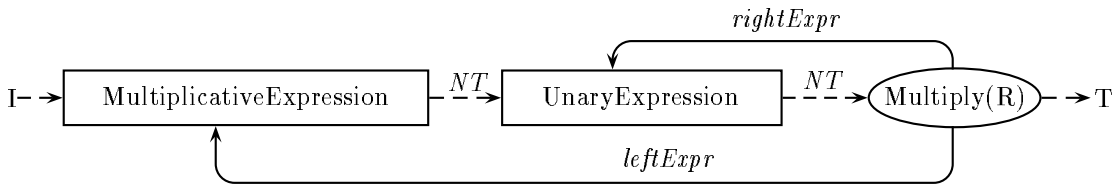
5.4.4 Multiplicative expressions

Multiplication expression

A *Multiply* expression contains a *MultiplicativeExpression* and a *UnaryExpression*. The types of *MultiplicativeExpression* and *UnaryExpression* must be numeric. Binary promotion is performed on the results of *MultiplicativeExpression* and *UnaryExpression*. The type of *Multiply* is the binary promotion of *MultiplicativeExpression* and *UnaryExpression*.

To evaluate *Multiply*, evaluate *MultiplicativeExpression* and then *UnaryExpression*. Then return the result of multiplying the promoted result of *MultiplicativeExpresion* with the promoted result of *UnaryExpression*.

Multiplication **::=** **MultiplicativeExpression * UnaryExpression**
MultiplicativeExpression = UnaryExpression |Multiplication |Division |Remainder



let t = binaryPromotion(MultiplicativeExpression,UnaryExpression)
Multiply.type := t
if *(MultiplicativeExpression.constant? and UnaryExpression.constant?)* **then**
Multiply.constant? := true
Multiply.result :=
*MultiplicativeExpression.result.convertTo(t) * UnaryExpression.result.convertTo(t)*

condition *MultiplicativeExpression.type.numeric?*
UnaryExpression.type.numeric?

Multiply:
*result := leftExpr.result.convertTo(type) * rightExpr.result.convertTo(type)*
Proceed Sequentially

Other multiplicative expressions

Division ::= *MultiplicativeExpression* / *UnaryExpression*
Remainder ::= *MultiplicativeExpression* % *UnaryExpression*

5.4.5 Additive expressions

Addition expression

An *Addition* expression contains an *AdditiveExpression* and a *MultiplicativeExpression*. If *AdditiveExpression* or *MultiplicativeExpression* is of type *String*, then the type of *Addition* is *String*, and if either *AdditiveExpression* or *MultiplicativeExpression* is not of type *String*, string conversion is performed on its result. If neither *AdditiveExpression* nor *MultiplicativeExpression* is of type *String*, then both *AdditiveExpression* and *MultiplicativeExpression* must be of numeric type, binary numeric promotion is performed on *AdditiveExpression* and *MultiplicativeExpression*, and the type of *Addition* is the binary promotion of *AdditiveExpression* and *MultiplicativeExpression*.

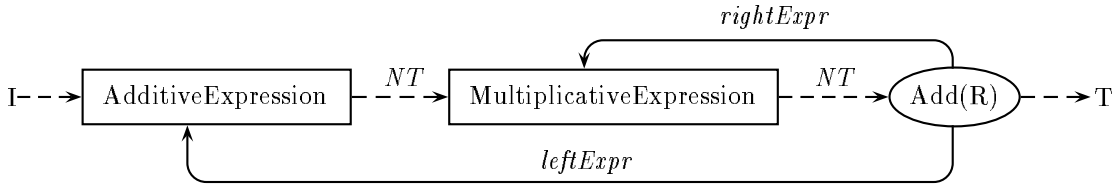
To evaluate *Addition*, evaluate *AdditiveExpression* and then *MultiplicativeExpression*. Then do one of the following:

- If *AdditiveExpression* and *MultiplicativeExpression* are of type *String*, then append the value of the String object returned by *MultiplicativeExpression* to the value of the String object returned by *AdditiveExpression*, and create a new *String* object whose value is the appended result. If there is no space for the new object, throw an OutOfMemoryError.
- If one of *AdditiveExpression* and *MultiplicativeExpression* is of type String and the other is not, perform string conversion on the non-String result. Append the value of the String object returned by *MultiplicativeExpression* to the value of the String object returned by *AdditiveExpression*, and

create a new *String* object whose value is the appended result. If there is no space for the new object, throw an OutOfMemoryError.

- If *AdditiveExpression* and *MultiplicativeExpression* are of numeric type, add the promoted result of *AdditiveExpression* to the promoted result of *MultiplicativeExpression* and return the result.

Addition **::=** **AdditiveExpression + MultiplicativeExpression**
AdditiveExpression = MultiplicativeExpression |Addition |Subtraction



let t =
*(***if** *(AdditiveExpression.type = String or MultiplicativeExpression.type = String)* **then** *String*
else *binaryPromotion(AdditiveExpression,MultiplicativeExpression))*
Add.type := t
if *(AdditiveExpression.constant? and MultiplicativeExpression.constant?)* **then**
Add.constant? := true
if *t = String* **then extend** *ClassInst* **with** *str*
Add.result := str
str.class := String
str.stringVal := AdditiveExpression.result.convertTo(String) +
MultiplicativeExpression.result.convertTo(String)
else *Add.result :=*
AdditiveExpression.result.convertTo(t) + MultiplicativeExpression.result.convertTo(t)

condition *AdditiveExpression.type = String or MultiplicativeExpression.type = String*
or (AdditiveExpression.type.numeric? and MultiplicativeExpression.type.numeric?)

Add:
if *type = String* **then**
Get New Memory
extend *ClassInst* **with** *str*
result := str
str.class := String
str.stringVal := leftExpr.result.convertTo(String) + rightExpr.result.convertTo(String)
else
result := leftExpr.result.convertTo(type) + rightExpr.result.convertTo(type)
Proceed Sequentially

Subtraction expression

Subtraction ::= AdditiveExpression - MultiplicativeExpression

5.4.6 Shift expressions

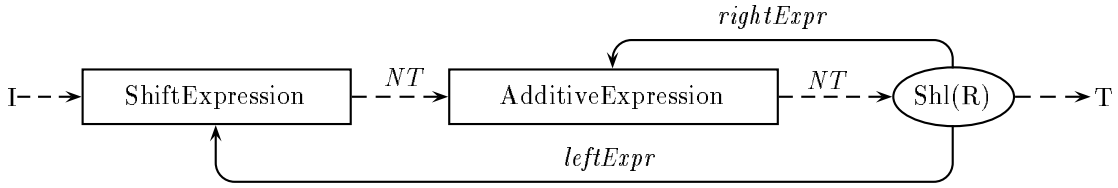
Shift-left expression

A *ShiftLeft* expression contains a *ShiftExpression* and an *AdditiveExpression*. The type of *ShiftExpression* and *AdditiveExpression* must be integral. Unary numeric promotion is performed on each of *ShiftExpression* and *AdditiveExpression*. The type of *ShiftLeft* is the unary promotion of *ShiftExpression*.

To evaluate *ShiftLeft*, evaluate *ShiftExpression* and then *AdditiveExpression*. Then return the result of multiplying the result of *ShiftExpression* by two to the power s, where s is the shift distance. If the promoted type of *AdditiveExpression* is Int, s is the result of *AdditiveExpression* modulo 32; if it is Long, s is the result of *AdditiveExpression* modulo 64.

Task.maxShiftDist : *Nat*	Maxmimum distance for left shift: either 32 or 64.
Task.leftOperandType : *Type*	Promoted type of left operand.
Task.rightOperandType : *Type*	Promoted type of right operand.

ShiftLeft ::= ShiftExpression << AdditiveExpression
ShiftExpresion = AdditiveExpression |ShiftLeft |ShiftRightSigned |ShiftRightUnsigned



let t = ShiftExpression.unaryPromotion
*let maxshift = (***if** *ShiftExpression.unaryPromotion = Int* **then** *32* **else** *64)*
let righttype = AdditiveExpression.unaryPromotion
let lefttype = ShiftExpression.unaryPromotion
Shl.type := t
Shl.maxShiftDist := maxshift
Shl.rightOperandType := righttype
Shl.leftOperandType := lefttype
if *ShiftExpression.constant? and AdditiveExpression.constant?* **then**
Shl.constant? := true
let l = ShiftExpression.result.convertTo(lefttype)
let r = (AdditiveExpression.result.convertTo(righttype)) mod maxshift
*Shl.result := l * 2↑r*

condition *ShiftExpression.type.integral? and AdditiveExpression.type.integral?*

Shl:
let l = leftExpr.result.convertTo(leftOperandType)
let r = (rightExpr.result.convertTo(rightOperandType)) mod maxShiftDist
*result := l * 2↑r*
Proceed Sequentially

Other shift expressions

ShiftRightSigned ::= *ShiftExpression* >> *AdditiveExpression*

ShiftRightUnsigned ::= *ShiftExpression* >>> *AdditiveExpression*

5.4.7 Relational and equality operations

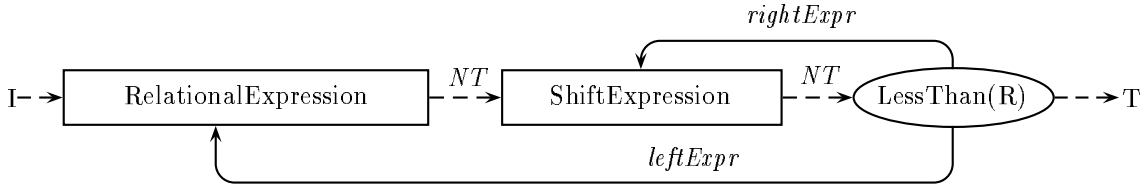
Less-than expression

A *LessThan* expression contains a *RelationalExpression* and a *ShiftExpression*. The types of *RelationalExpression* and *ShiftExpression* must be numeric. Binary numeric promotion is performed on *RelationalExpression* and *ShiftExpression*. The type of *LessThan* is Boolean.

To evaluate *LessThan*, evaluate *RelationalExpression* and then *ShiftExpression*. If the result of *RelationalExpression* is less than the result of *ShiftExpression*, return True; otherwise, return False.

Task.operandType : *Type*	Promoted type of operands.

LessThan **::=** **RelationalExpression < ShiftExpression**
RelationalExpression = ShiftExpression |LessThan |GreaterThan
. = LessThanEqualTo |GreaterThanEqualTo |TypeComparison



LessThan.type := *Boolean*
let t = binaryPromotion(RelationalExpression,ShiftExpression)
LessThan.operandType := *t*
if *RelationalExpression.constant? and ShiftExpression.constant?* **then**
LessThan.constant? := true
LessThan.result :=
(RelationalExpression.result.convertTo(t) < ShiftExpression.result.convertTo(t))

condition *RelationalExpression.type.numeric? and ShiftExpression.type.numeric?*

LessThan:
result := (leftExpr.result.convertTo(operandType) < rightExpr.result.convertTo(operandType))
Proceed Sequentially

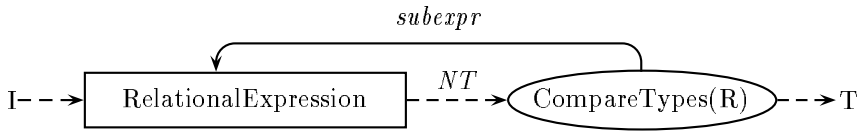
Type comparison expression

A *TypeComparison* expression contains a *RelationalExpression* and a *ReferenceType*. The type of *RelationalExpression* must be a reference type or null, and the type of *RelationalExpression* must be castable to *ReferenceType*. The type of *TypeComparison* is Boolean.

To evaluate *TypeComparison*, evaluate *RelationalExpression*. If the result of *RelationalExpression* is not null and is castable to the type denoted by *ReferenceType*, return True; otherwise, return False.

Task.testType : *Type*	(Fixed) type to test for in type comparison expression.

TypeComparison ::= RelationalExpression instanceof ReferenceType



CompareTypes.type := Boolean
CompareTypes.testType := ReferenceType.type

condition *RelationalExpression.type.isA?(ReferenceType)*
RelationalExpression.type.castableTo?(ReferenceType.type)

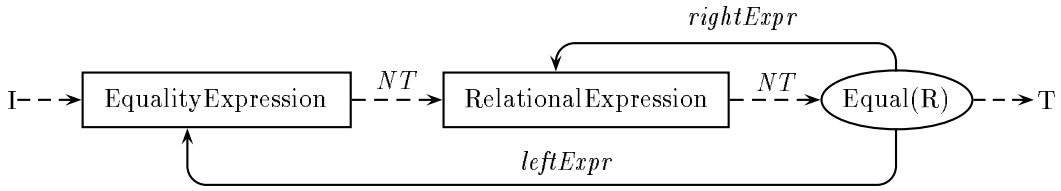
CompareTypes:
result := subexpr.result ≠ null and subexpr.result.class.castableTo?(testType)
Proceed Sequentially

Equality expression

An *Equality* expression contains an *EqualityExpression* and a *RelationalExpression*. The types of *EqualityExpression* and *RelationalExpression* must be both Boolean, both numeric or both reference. If the types of *EqualityExpression* and *RelationalExpression* are both numeric, binary numeric promotion is performed on the results of *EqualityExpression* and *RelationalExpression*. The type of *Equality* is Boolean.

To evaluate *Equality*, evaluate *EqualityExpression* and then *RelationalExpression*. If the (promoted) result of *EqualityExpression* is equal to the (promoted) result of *RelationalExpression*, return True; otherwise, return False. In the case where *EqualityExpression* and *RelationalExpression* are reference types, equality is defined as whether the results of *EqualityExpression* and *RelationalExpression* are the same object.

Equality **::=** **EqualityExpression == RelationalExpression**
EqualityExpression = RelationalExpression |Equality |Inequality



Equal.type := Boolean
let t = binaryPromotion(EqualityExpression,RelationalExpression)
Equal.operandType := t
if *EqualityExpression.constant? and RelationalExpression.constant?* **then**
Equal.constant? := true
Equal.result :=
(EqualityExpression.result.convertTo(t) = RelationalExpression.result.convertTo(t))

condition *EqualityExpression.type = Boolean <=> RelationalExpression.type = Boolean*
EqualityExpression.type.numeric? <=> RelationalExpression.type.numeric?
EqualityExpression.type.isA?(ReferenceType) <=>
RelationalExpression.type.isA?(ReferenceType)

equal:
result := (leftExpr.result.convertTo(operandType) = rightExpr.result.convertTo(operandType))
Proceed Sequentially

Inequality expression

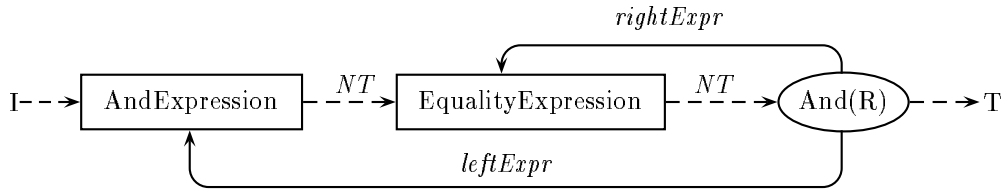
Inequality ::= *EqualityExpression* != *RelationalExpression*

5.4.8 Bitwise and logical operations

An *And* expression contains an *AndExpression* and an *EqualityExpression*. The types of *AndExpression* and *EqualityExpression* must be both integral, in which case binary numeric promotion is performed on the results of *AndExpression* and *EqualityExpression* and the type of *And* is the binary promotion of *Andexpression* and *EqualityExpression*, or both Boolean, in which case the type of *And* is Boolean.

To evaluate *And*, evaluate *AndExpression* and then *EqualityExpression*. If the types of *AndExpression* and *EqualityExpression* are numeric, return the bitwise AND of the results of *AndExpression* and *EqualityExpression*. Otherwise, return True if the results of *AndExpression* and *EqualityExpression* are both True, and return False otherwise.

And **::=** **AndExpression & EqualityExpression**
AndExpression = EqualityExpression |And



let t =
*(***if** *AndExpression.type.numeric?* **then** *binaryPromotion(AndExpression,EqualityExpression)*
else *Boolean)*
And.type := t
if *AndExpression.constant? and EqualityExpression.constant?* **then**
And.constant? := true
*And.result := (***if** *t = Boolean* **then** *(AndExpression.result and EqualityExpression.result)*
else *AndExpression.result.convertTo(t).and(EqualityExpression.result.convertTo(t)))*

condition *(AndExpression.type.numeric? and EqualityExpression.type.numeric?)*
or (AndExpression.type = EqualityExpression.type = Boolean)

And:
result :=
*(***if** *resultType = Boolean* **then** *(leftExpr.result and rightExpr.result)*
else *leftExpr.result.convertTo(type).bitwiseAnd(rightExpr.result.convertTo(type)))*
Proceed Sequentially

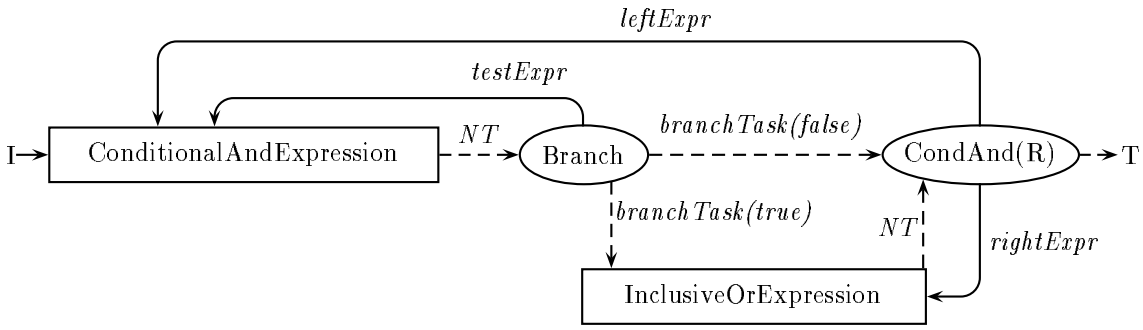
ExclusiveOr ::= ExclusiveOrExpression ^ AndExpression
InclusiveOr ::= InclusiveOrExpression | ExclusiveOrExpression

5.4.9 Conditional logical operations

A *ConditionalAnd* contains a *ConditionalAndExpression* and an *InclusiveOrExpression*. The types of *ConditionalAndExpression* and *InclusiveOrExpression* must be Boolean. The type of *ConditionalAnd* is Boolean.

To evaluate *ConditionalAnd*, evaluate *ConditionalAndExpression*. If its result is False, return False; otherwise, evaluate *InclusiveOrExpression* and return its result.

ConditionalAnd **::=** **ConditionalAndExpression && InclusiveOrExpression**
ConditionalAndExpression = InclusiveOrExpression |ConditionalAnd



CondAnd.type := Boolean
if *ConditionalAndExpression.constant? and InclusiveOrExpression.constant?* **then**
CondAnd.constant? := true
*CondAnd.result := (***if** *ConditionalAndExpression.result =* false **then** false
else *InclusiveOrExpression.result)*

Condand:
*result := (***if** *leftExpr.result =* false **then** false **else** *rightExpr.result)*
Proceed Sequentially

ConditionalOr ::= ConditionalOrExpression || ConditionalAndExpression
ConditionalOrExpression = ConditionalAndExpression | ConditionalOr

5.4.10 Conditional operation

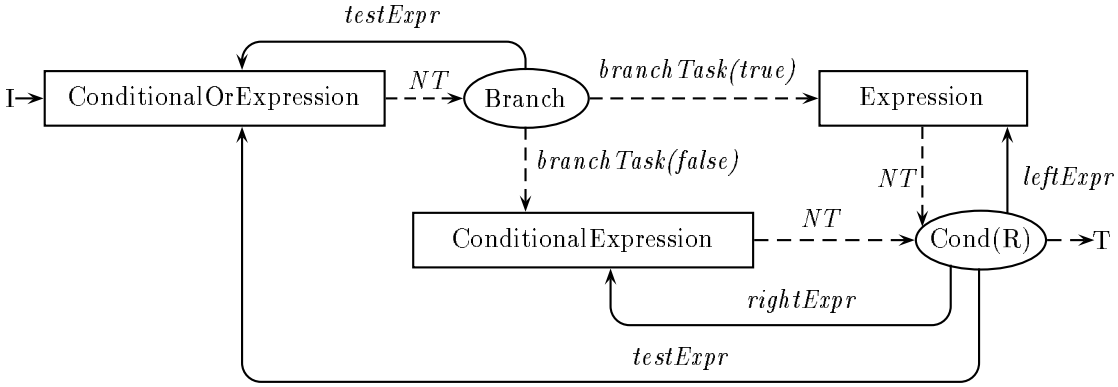
A *Conditional* contains a *ConditionalOrExpression*, an *Expression* and a *ConditionalExpression*. The type of *ConditionalOrExpression* must be Boolean. The types of *Expression* and *ConditionalExpression* must be both numeric, both Boolean or both reference. (One of the types must be assignable to the other.) The type of *Conditional* is determined as follows:

- If *Expression* and *ConditionalExpression* have the same type, then *Conditional* has that type.
- Otherwise, if *Expression* and *ConditionalExpression* have numeric types, then one of the following applies:
  - If one of the types is Byte and the other is Short, then *Conditional* has type Short.
  - If one operand has type Byte, Short or Int (call it T) and the other operand is a constant expression of type Int whose value is representable in T, then the type of *Conditional* is T.
  - Otherwise, binary numeric promotion is applied to the operands, and the type of *Conditional* is the binary promotion of the types of the operands.
- Otherwise, if one of *Expression* and *ConditionalExpression* is of the null type and the other is of reference type, then the type of *Conditional* is the reference type.
- Otherwise, if *Expression* and *ConditionalExpression* are of different reference types, then it must be possible to convert one of the types (S) to the other (T) by assignment conversion. The type of *Conditional* is T.

To evaluate *Conditional*, evaluate *ConditionalOrExpression*. If the result is True, evaluate *Expression* and return its (promoted) result; otherwise, evaluate *ConditionalExpression* and return its (promoted) result.

Conditional **::=** **ConditionalOrExpression ? Expression : ConditionalExpression**

ConditionalExpression = ConditionalOrExpression |Conditional



let A = Expression, let B = ConditionalExpression
let atype = Expression.type, btype = ConditionalExpression.type
let t = (**if** *atype = btype* **then** *atype*
elseif *atype in {Byte,Short} and btype in {Byte,Short}* **then** *Short*
elseif *atype in {Byte,Short,Char} and btype = Int and*
B.constant? and B.result.representableIn?(atype) **then** *atype*
elseif *btype in {Byte,Short,Char} and atype = Int and*
A.constant? and A.result.representableIn?(btype) **then** *btype*
elseif *atype.numeric? and btype.numeric?* **then** *binaryPromotion(A,B)*
elseif *atype = Null and btype.reference?* **then** *btype*
elseif *btype = Null and atype.reference?* **then** *atype*
elseif *atype.assignableTo?(btype)* **then** *btype*
elseif *btype.assignableTo?(atype)* **then** *atype)*
CondOp.resultType := t
if *ConditionalOrExpression.constant? and Expression.constant?*
and ConditionalExpression.constant? **then**
CondOp.constant? := true
CondOp.result :=
(**if** *ConditionalOrExpression.result =* true **then** *ConditionalOrExpression.result*
else *Expression.result)*

Cond:
result := (**if** *testExpr.result =* true **then** *leftExpr.result* **else** *rightExpr.result)*
Proceed Sequentially

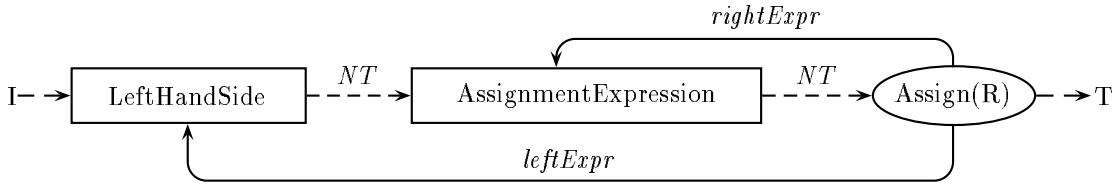
5.4.11 Assignment operations

Simple assign expression

An *Assignment* expression contains a *LeftHandSide* and an *AssignmentExpression*. The type of *AssignmentExpression* must be assignment-convertible to the type of *LeftHandSide*. *LeftHandSide* must be able to return an lvalue. The type of *Assignment* is the type of *LeftHandSide*.

To evaluate *Assignment*, evaluate *LeftHandSide*, then evaluate *AssignmentExpression*. If the *LeftHandSide* is an array access, and *LeftHandSide* and *AssignmentExpression* are of reference type, check that the result of *AssignmentExpression* is of a class that is assignment-convertible to the type of *LeftHandSide*; if not, throw an ArrayStoreException. Finally, assign the result of *AssignmentExpression* to the variable returned by *LeftHandSide*.

Assignment	**::=**	**LeftHandSide = AssignmentExpression**
LeftHandSide	=	ExpressionName |FieldAccess |ArrayAccess
AssignmentExpression	=	ConditionalExpression |Assignment



LeftHandSide.lvalue? := true
Assign.type := LeftHandSide.type

condition *AssignmentExpression.assignableTo?(LeftHandSide)*
not LeftHandSide.finalFieldAccess?
LeftHandSide.isA?(FieldAccess) or LeftHandSide.isA?(ArrayAccess)
or LeftHandSide.isA?(ExpressionName)

Assign:
if *leftExpr.checkAssignability? and not rightExpr.result.class.assignableTo?(type)* **then**
Throw Exception Of Class ArrayStoreException
else
Assign rightExpr.result.convertTo(type) To leftExpr.result
result := rightExpr.result.convertTo(type)
Proceed Sequentially

Add-assign expression

An *AddAssignment* expression contains a *LeftHandSide* and an *AssignmentExpression*. If the type of *LeftHandSide* is String, then string conversion is performed on the result of *AssignmentExpression* if it is not of type String. If the type of *LeftHandSide* is not String, then the types of both *LeftHandSide* and *AssignmentExpression* must be both numeric. The type of *AddAssignment* is the type of *LeftHandSide*.

To evaluate *AddAssignment*, evaluate *LeftHandSide*, then evaluate *AssignmentExpression*.

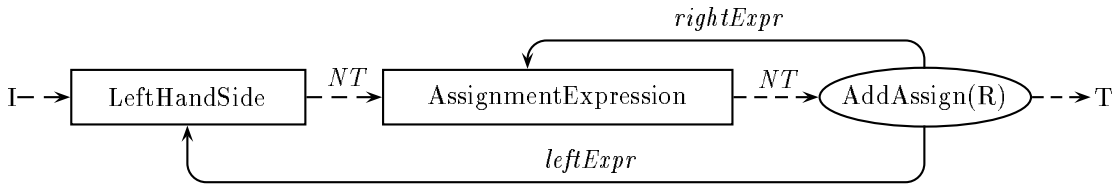
- If the types of *LeftHandSide* and *AssignmentExpression* are both String, append the value of the String object returned by *AssignmentExpression* to the value of the String object returned by

LeftHandSide, and create a new *String* object whose value is the appended result. If there is no space for the new object, throw an OutOfMemoryError.

- If the type of *LeftHandSide* is String but the type of *AssignmentExpression* is not, perform string conversion on the result of *AssignmentExpression*. Append the value of the string-converted result of *AssignmentExpression* to the value of the String object returned by *LeftHandSide*, and create a new String object whose value is the appended result. If there is no space for the new object, throw an OutOfMemoryError.
- Otherwise, the types of *LeftHandSide* and *AssignmentExpression* are both numeric. Add the result of *LeftHandSide* to the result of *AssignmentExpression* and then convert the result to the type of *LeftHandSide*.

Store the result in the variable result of *LeftHandSide* and return it.

AddAssignment	**::=**	**LeftHandSide += AssignmentExpression**
LeftHandSide	=	ExpressionName |FieldAccess |ArrayAccess
AssignmentExpression	=	ConditionalExpression |Assignment



LeftHandSide.lvalue? := true
Assign.type := LeftHandSide.type

condition *LeftHandSide.type ≠ String => AssignmentExpression.assignableTo?(LeftHandSide)*
not LeftHandSide.finalFieldAccess?
LeftHandSide.isA?(FieldAccess) or LeftHandSide.isA?(ArrayAccess)
or LeftHandSide.isA?(ExpressionName)

```
AddAssign:
if leftExpr.type = String then
  Get New Memory
    extend ClassInst with str
      str.class := String
      str.stringVal := leftExpr.result.value + rightExpr.result.convertTo(String)
      result := str
      Assign str To leftExpr.result
elseif leftExpr.checkAssignability? and not rightExpr.result.class.assignableTo?(type) then
  Throw Exception Of Class ArrayStoreException
else
  Assign (leftExpr.result.value + rightExpr.result.convertTo(type)) To leftExpr.result
  result := (leftExpr.result.value + rightExpr.result.convertTo(type))
  Proceed Sequentially
```

6 Method and constructor invocation

When a method is invoked upon an object, execution of the current method (the *invoker*) is suspended and execution of the newly invoked method (the *invokee*) begins. Control returns to the invoker when the invokee terminates. involve actions not specified in the previous section. In the ASM of this section, the actions of passing control to an invokee and returning it to an invoker are defined explicitly.

6.1 Preliminaries

Methods may invoke methods; when a method invocation expression is evaluated during execution of a method, control passes to the indicated invokee. Before the invokee executes, information about the invoker is recorded: the target object and argument values supplied in the invocation expression, and the point to which control returns after the invokee terminates. As methods may be invoked recursively, multiple invocations of the same method may be active at once. A *stack* of method invocations is maintained, whose top element (or *frame*) corresponds to the current invocation.

A *method invocation* contains an identifier and a number of arguments (subexpressions), and may contain a *target reference*. The target reference is an expression that evaluates to the target object of the invocation. All the arguments of an invocation are evaluated, in their order of appearance, before the invocation itself.

Compile-time method selection

A *descriptor method* is chosen for the invocation expression at compile time. If the invocation is not *virtual* (see below), this is the method to invoke when the expression is evaluated. Otherwise, the *descriptor* (signature and return type) of the method is used at runtime to select a method to invoke. The choice of descriptor method at compile time is narrowed to methods of a single class; then the *most specific applicable* and *accessible* method installed in this class is chosen. The choice of class proceeds as follows.

1. If the expression is of the simple form $Identifier$ ([$ArgumentList$]), then the descriptor method is to be found in the current class.
2. If the expression is of the form $TypeName$. $Identifier$ ([$ArgumentList$]), then the descriptor method is to be found in the (class) type of $TypeName$. $TypeName$ must not name an interface, and the method must be static.
3. If the expression is of the form $ExpressionName$. $Identifier$ ([$ArgumentList$]), then the descriptor method is to be found in the (class) type of $ExpressionName$.
4. If the expression is of the form $Primary$. $Identifier$ ([$ArgumentList$]), then the descriptor method is to be found in the (class) type of $Primary$.
5. If the expression is of the form $SuperExpression$. $Identifier$ ([$ArgumentList$]), then the descriptor method is to be found in the type of $SuperExpression$, i.e. the parent of the current class. The current class must not be Object, and the method must not be static. An expression of this form must not appear in a static initializer or an initializer for a static variable.

Of the methods installed in the chosen class, a method is *applicable* to the invocation if the number of its parameters equals the number of argument expressions in the invocation expression, and each argument can be converted via *method invocation conversion* to the type of the corresponding parameter. Method invocation conversion is possible if the argument type is equal to or narrower than the parameter type.

Accessibility is defined as follows. A method is accessible if the method has public access status. If the method has protected status, it is accessible if the invocation is of the simple form $Identifier$ ([$ArgumentList$]) or the form Q . $Identifier$ ([$ArgumentList$]), where the type of Q is a subclass of the current class. If the method has private status, it is accessible only if the invocation is of the simple form $Identifier$ ([$ArgumentList$]).

Of the applicable, accessible methods in the class, a *maximally specific* method is one for which the parameter types are not convertible to the parameter types of any other method. There must be exactly one maximally specific method, and this is the descriptor method chosen at compile time.

Runtime method selection

The runtime actions taken in a method invocation depend on the characteristics of the method and the form of the expression:

1. If the method is declared static, no target object is given for the invocation (although a target reference expression may appear). Reference to the target object is impossible within the method. The descriptor method is invoked.

2. If the method is declared private but not static, a target object is given for the invocation. If a target reference expression evaluates to null, a NullPointerException is thrown. The descriptor method is invoked.

3. Otherwise, the invocation is *virtual*. A target object is given for the invocation, and if a target reference expression evaluates to null, a NullPointerException is thrown. At runtime, a search for a method matching the signature of the compile-time method occurs. Starting from a class c (which is the class of the target object, or the parent of the class of the target object if the invocation is of the form $SuperExpression$. $Identifier$ ([$ArgumentList$]), the search locates the lowest superclass of c in which a matching method is installed and accessible. The result may be the descriptor method.

6.2 ASM J_3

M_3 consists of the montages of M_2 plus the montages defined later in this section. The ASM J_3 has a compiler agent with module C_3 (compile-time rules of M_3), and an executor agent with module R_3 (runtime rules of M_3) for each element of the universe $Frame$. The module for each executor has the guard $Self.active?$. Initially, there is a single element of $Frame$ and thus a single executor agent. The compiler agent runs before all executors and no two executors run concurrently.

6.3 Function and macro definitions

Compile-time method selection

macro $M.installedIn?(C)$
(Is method M installed in class C?)
$M = C.method(M.ID, M.paramTypes)$

macro $T.methodInvocationAssignableTo?(U)$
(Is type T assignable to type U through method invocation conversion?)
$T = U$ or $T.narrowerThan?(U)$

macro $M.applicableTo?(ID, TYPES)$
(Does method M match the identifier ID and are its parameter types assignable to those in $TYPES$?)
$M.ID = ID$ and $|M.paramTypes| = |TYPES|$ and
$(\forall (i,t) \in TYPES, (j,u) \in M.paramTypes : i = j) t.methodInvocationAssignableTo?(u)$

macro $M.accessibleTo?(C)$
(Is method M accessible, given the invoker class C?)
$(M.accessStatus = public)$ or $(M.accessStatus = private$ and $M.declarer = curClass)$
or $(m.accessStatus = protected$ and $(C = curClass$ or $C.subclassOf?(curClass)))$

macro $M.maximallySpecificIn?(C)$
(Does method M have a maximally narrow set of parameter types among the accessible methods of C?)
$(\forall m' \neq M : m'.installedIn?(C))$
$m'.accessibleTo?(C) \Rightarrow$ not $m'.applicableTo?(M.ID, M.paramTypes)$

macro $M.validDescriptorMethod?(ID, TYPES, C)$
(Is M a valid descriptor method for the invocation
(with identifier ID, parameter types $TYPES$, invoker class C)?)
$M.installedIn?(C)$ and $M.applicableTo?(ID, TYPES)$ and $M.accessibleTo?(C)$
and $M.maximallySpecificIn?(C)$

macro Set Up Invocation With Invoker Class C
let $argtypes = \{(i, ArgumentList[i].type) : ArgumentList[i].def?\}$
choose $m : m.isA?(Method) : m.validDescriptorMethod?(Identifier.ID, argtypes, C)$
$InvokeMethod.descriptorMethod := m$
$InvokeMethod.type := m.type$
$InvokeMethod.virtualInvocation? :=$ not $(m.static?$ or $m.accessStatus = private)$
$InvokeMethod.superInvocation? := Invoker.isA?(SuperExpression)$

macro $C.hasUniqueDescriptorMethod?$
let $argtypes = \{(i, ArgumentList[i].type) : ArgumentList[i].def?\}$
$(\exists m : m.isA?(Method))m.validDescriptorMethod?(Identifier.ID, argtypes, C)$
and $(\forall m, n : m.isA?(Method)$ and $n.isA?(Method))$ and $m \neq n$
not $(m.validDescriptorMethod?(Identifier.ID, argtypes, C)$
and $n.validDescriptorMethod?(Identifier.ID, argtypes, C))$

Runtime method selection

macro $M.invokableUpon?(C)$
(Can M be invoked upon an object of class C? i.e. Is C a subclass of M's class?)
$M.declarer = C$ or $C.subclassOf?(M.declarer)$

macro $M'.matchesDescriptor?(M)$
(Does method M' match method M in terms of descriptor?)
$M'.ID = M.ID$ and $M'.type = M.type$ and
$(\forall (i, t) \in M'.paramTypes, (j, u) \in M.paramTypes : i = j)t = u$
and $(M'.declarer = M.declarer$ or $M'.declarer.subclassOf?(M.declarer))$

macro $M'.mostRelevantApplicableMethod?(M, C)$
(Is M' a method that matches M, is invokable upon C,
and is declared in the lowest possible superclass of C?)
$M'.matchesDescriptor?(M)$ and $M'.invokableUpon?(C)$
and $(\forall m'' \in Method : m'' \neq M'$ and $m''.matchesDescriptor(M)$ and $m''.invokableUpon?(C))$
$M'.class.subclassOf?(m''.class)$

$Frame$	Universe of invocation frames.
$Frame.active? : Boolean$	Is this the active invocation frame (i.e. top of the stack)?
$Frame.invoker : Task$	method invocation task that created this frame.
$Frame.returnValue : Value$	Return value computed in this invocation.
$Frame.targetObj : Object$	Target object of this invocation frame.
$Task.virtualInvocation? : Boolean$	Is this a virtual method invocation?
$Task.superInvocation? : Boolean$	Is this a method invocation of the form `super(...)`?
$Task.descriptorMethod : Method$	Descriptor method for the invocation.

Different invocations execute different tasks, have distinct local variables, and compute different results even when they execute the same code. Thus the functions $curTask$, $curTargetTask$, $localVar$ and $result$ take an additional $Frame$ argument. In all previous ASM rules, add the term $Self$ as an argument to all occurrences of $curTask$, $curTargetTask$, $result$ and $localVar$.

$Frame.curTask : Task$	Current task for the invocation to execute.
$Frame.curTargetTask : Task$	Current target task of the invocation.
$Task.result(Frame) : Result$	Result of expression evaluation for the invocation.
$VariableDeclaration.localVar(Frame) : Variable$	Local variable created by the declaration for this invocation.

Terms of the form $Self.curTask.f(\bar{x})$, where f is a function name and $\bar{x}$ is a sequence of terms, are usually abbreviated as $f(\bar{x})$.

macro Activate New Frame F R:
(Create frame, make it the active frame, and fire rule R.) **extend** $Frame$ with F
    $F.prevFrame := self$
    $self.active? := false$
    $F.active? := true$
    R

macro Invoke M On OBJ
(Activate a new frame and supply it with target reference and arguments.)
Activate New Frame f
    $f.curTask := M.firstTask$
    $f.invoker := self.curTask$
    $f.targetObj := OBJ$
    do-forall $i : argExpr(i).def?$
        extend $Variable$ **with** var
            $m.paramDecl(i).localVar(f) := var$
            $var.value := argExpr(i).result(self)$
Proceed Sequentially
macro Invoke Static Method M
(Activate a new frame and supply it with arguments (but no target reference).)
Activate New Frame f
    $f.curTask := M.firstTask$
    $f.invoker := self.curTask$
    do-forall $i : argExpr(i).def?$
        extend $Variable$ **with** var
            $m.paramDecl(i).localVar(f) := var$
            $var.value := argExpr(i).result(self)$
Proceed Sequentially
macro Perform Simple Invocation:
if $virtualInvocation?$ **then**

choose m : $m.mostRelevantApplicableMethod?(descriptorMethod, self.targetObj.class)$
Invoke m On $Self.targetObj$
else Invoke StaticMethod $descriptorMethod$

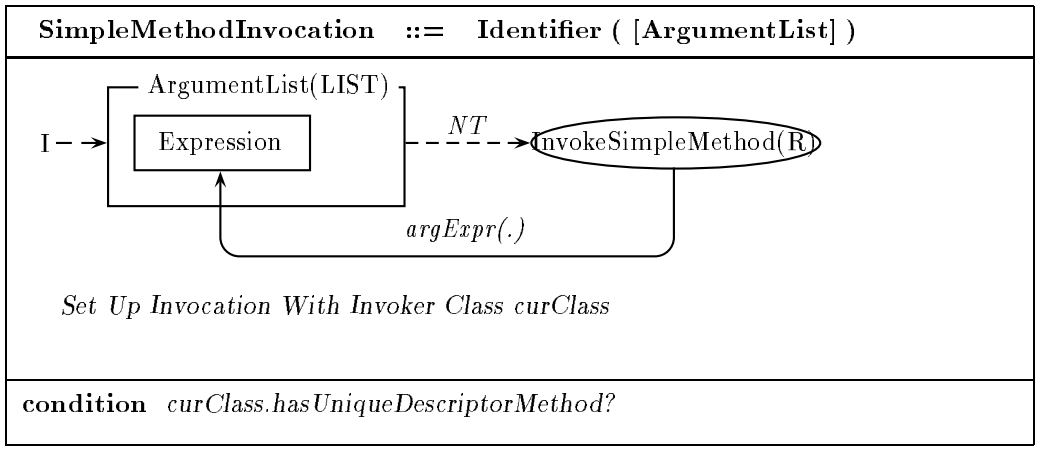
macro Perform Qualified Invocation:
if $virtualInvocation?$ **then**
if $superInvocation?$ **then**
choose m : $m.mostRelevantApplicableMethod?(descriptorMethod,$
$self.targetRefExpr.result.class.parent)$
Invoke m On $targetRefExpr.result(self)$
else
choose m : $m.mostRelevantApplicableMethod?(descriptorMethod,$
$self.targetObj.result.class.parent)$
Invoke m On $targetRefExpr.result(self)$
else Invoke StaticMethod $descriptorMethod$

macro Return:
(Record return value, if one is present; jump toward end of method/constructor.)
if $retExpr.def?$ **then** $self.returnValue := retExpr.result(self)$
Jump Toward $endTask$

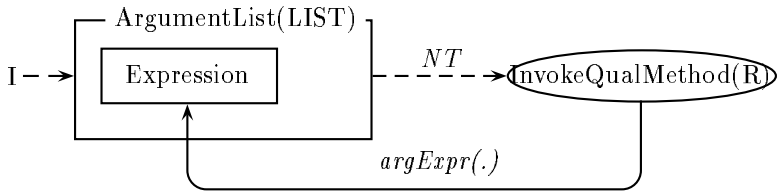
macro Deactivate Current Frame
(Remove current (active) frame; make next lowest frame active.)
$self.prevFrame.active? := true$
Remove $self$

macro Terminate Invocation:
(Deactivate current invocation frame; record return value in next lowest frame.)
Deactivate Current Frame
if $self.prevFrame.def?$ **then**
if $curException.def?$ **then**
$self.prevFrame.curTargetTask := self.prevFrame.curTask.catchTarget(curException.class)$
elseif $self.returnValue.def?$ **then**
$self.invoker.result(self.prevFrame) := self.returnValue$

6.4 Construction of method invocation expressions and return statement



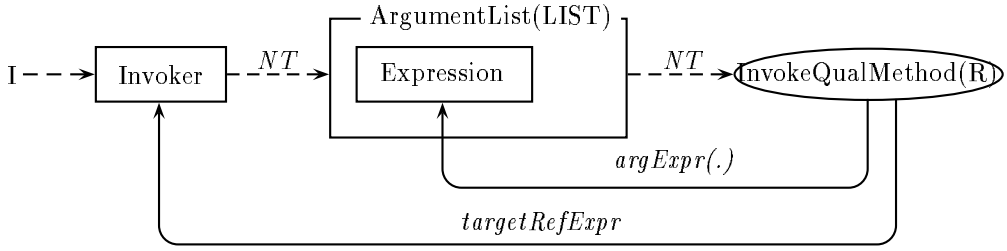
QualifiedMethodInvocation1 ::= TypeName . Identifier ([ArgumentList])



Set Up Invocation With Invoker Class TypeName.type

condition *TypeName.type.hasUniqueDescriptorMethod?*

QualifiedMethodInvocation2 ::= Invoker . Identifier ([ArgumentList])
Invoker = ExpressionName |Primary |SuperExpression



Set Up Invocation With Invoker Class Invoker.type

condition *Invoker.type.hasUniqueDescriptorMethod?*

7 Initialization and finalization

Classes and objects are implicitly initialized before their use. In addition, unreachable objects are implicitly finalized before their removal via garbage collection. In this section, we provide the details of these processes.

7.1 Preliminaries

Class initialization

A class must be initialized before it can be used in a program. Initialization consists of executing the class's static initializer block and static field initializers, in the order in which they appear in the declaration. Class initialization may occur concurrently with program execution.

Initialization of class instance fields

The non-static fields of a class instance are initialized during the invocation of the constructor, after the superclass constructor has terminated. The field initialization expressions are evaluated in the order in which they appear in the class declaration.

Object finalization and destruction

While the Java programmer explicitly creates objects (through class instance and array creation expressions), their removal when no longer needed is implicit. A *garbage collector* agent runs concurrently with the program executor to test the *reachability* of each object: whether it may ever be an evaluation result in a continuation of the computation. If not, the object is removed to provide storage space for other objects. How and when the garbage collector finds and removes unreachable objects are outside the specification of Java.

A method `finalize()` is defined for the Object class. This method is invoked after an object has been deemed unreachable but before its removal. The intended use of `finalize()` is to perform cleanup actions on the target object (e.g. closing files opened by the object) before its removal. While the `finalize()` method of class Object does nothing, it may be overridden by other classes.

Each object is characterized by its finalization status and its reachability status. An object is *finalized* if its finalizer has been automatically invoked. (A program may explicitly invoke `finalize()`; this does not affect the finalization status of any object.) An object is *finalizable* if it is not finalized but may have its finalizer automatically invoked at the present time. An object is *unfinalized* if it is neither finalized nor finalizable. An object is *reachable* if it can be accessed in any potential continuing computation. This is true if and only if there is a chain of references from a variable declaration of an active method invocation to the object. An object is *finalizer-reachable* if it can be accessed from a finalizable object through a chain of references. An *unreachable* object cannot be accessed by either means.

Finalization and removal of objects proceed as follows. Initially an object is unfinalized and reachable. During the course of a computation , a reachable object may become finalizer-reachable or unreachable. If its class inherits the vacuous definition of `finalize()` from class Object, it may be immediately removed once it becomes unreachable. Otherwise, the garbage collector may identify it as finalizable. Once an object becomes finalizable, its finalizer may be invoked automatically, at which point it becomes finalized (and reachable, through the invocation of `finalize()`). At the point that the object becomes once again unreachable, it may be removed.

7.2 ASM J_4

The ASM J_4 has a compiler agent with module C_4 and executor agents with module R_4, each associated with an element of the $Frame$ universe. There are a number of *initializer* agents, each associated with an element of the $Class$ universe, with module R_4. In addition, there are a number of *finalizer* agents, each with module $R_3 \cup \{FinalizeObjects\}$. The executor agents run exclusively with regard to one another, but concurrently with the finalizers.

7.3 Function and macro definitions

Class initialization

macro $N.classInitializer?$:
(Is node N a declaration of a static initializer or static fields?)
$N.isA?(StaticInitializerNode)$ or ($N.isA?(FieldDeclarationNode)$ and $N.static?$)

macro Add Class Initializer:
(Add initializer code to the class initializer block.)
if $n.firstInList?(ClassBodyDeclarations, n.classInitializer?)$ **then**
    $staticInitTask := n.initialTask$
choose $p : p.nextInList?(n, ClassBodyDeclarations, p.classInitializer?)$
    $n.terminal.nextTask := p.initialTask$

Initialization of class instance fields

macro $N.instanceInitializer?$:

(Is node N a declaration of instance fields?)
$N.isA?(FieldDeclarationNode)$ and not $N.static?$

macro Add Field Initializers:
(Add initializer code to the field initializer block.)
if $n.firstInList?(ClassBodyDeclarations, n.instanceInitializer?)$ **then**
    $curNode.fieldInitTask := n.initialTask$
if $n.lastInList?(ClassBodyDeclarations, n.instanceInitializer?)$ **then**
    extend $ReturnTask$ **with** t
        $n.terminal.nextTask := t$
else
    choose $p : p.nextInList?(n, ClassBodyDeclarations, p.instanceInitializer?)$
        $n.terminal.nextTask := p.initialTask$

macro Set Initializers:
$c.curTask := ClassBody.staticInitTask$
$c.fieldInitTask := ClassBody.fieldInitTask$

macro Initialize Fields:
Activate New Frame f
    $f.curTask := curClass.fieldInitTask$
    $f.targetObj := self.targetObj$
    Proceed Sequentially

Object finalization and destruction

macro $immediatelyReachableVariables$:
(All variables immediately reachable through a static field, local variable/parameter,
or instance field of a target object.)
$\{d.staticVar : d.isA?(VariableDeclaration) : d.staticVar.def?\}$
$\cup\{d.localVar(f) : d.isA?(VariableDeclaration), f.isA?(Frame) : d.localVar(f).def?\}$
$\cup\{f.targetObj.fieldVar(d) : f.isA?(Frame), d.isA?(VariableDeclaration) :$
    $f.targetObj.fieldVar(d).def?\}$

macro $currentReferences$:
$\{(var, var.value) : var.isA?(Variable) : var.value.def?\}$
$\cup\{(obj, obj.fieldVar(d)) : obj.isA?(Object), d.isA?(VariableDeclaration) : obj.fieldVar(d).def?\}$
$\cup\{(arr, arr.component(i)) : arr.isA?(Array) : arr.component(i).def?\}$

macro $OBJ.reachable?$:
(Is there a sequence of variable-value pairs, starting from an immediately reachable variable
and reaching the object OBJ?)
$(\exists refseq : refseq.isA?(ReferenceSequence))$
    $refseq[1] \in immediatelyReachableVariables$ and $refseq[refseq.length] = OBJ$
        and $(\forall i : 1 \leq i < refseq.length)(refseq[i], refseq[i+1]) \in currentReferences$

macro $OBJ.finalizerReachable?$:
(Is there a sequence of variable-value pairs starting from an object marked as finalizable
and reaching the object OBJ?)
$(\exists refseq : refseq.isA?(ReferenceSeq))$
    $refseq[1].isA?(Object)$ and $refseq[1].finalizable?$ and $refseq[refseq.length] = OBJ$

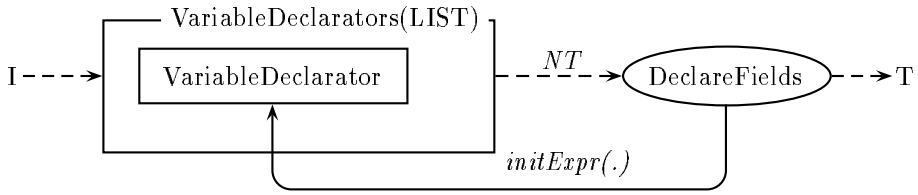
and $(\forall i : 1 \leq i < refseq.length)(refseq[i], refseq[i+1]) \in currentReferences$

macro *OBJ.unreachable?*:
(Is object *OBJ* neither reachable nor finalizer-reachable?)
not *OBJ.reachable?* and not *OBJ.finalizerReachable?*

7.4 Montage and rule definitions

Initialization of class instance fields

FieldDeclaration	**::=**	**[FieldModifiers] Type VariableDeclarators ;**
FieldModifiers	=	LIST(FieldModifier)
FieldModifier	=	public |protected |private |final |static |transient |volatile
VariableDeclarators	=	LIST(VariableDeclarator,)



Process Each Node n In List VariableDeclarators
curNode.field(n.ID) := n.declaration
Assign FieldModifiers To n.declaration
n.declaration.type := Type.type
DeclareFields.declaration(n.position) := n.declaration
DeclareFields.staticFieldDecl? := (FieldModifiers.hasModifier?(static))

condition *FieldModifiers.consistent?*
(For All Distinct Members m,n Of List VariableDeclarators) m.ID ≠ n.ID

DeclareFields:
do-**forall** *i: initExpr(i).def?*
extend *Variable* **with** *var*
if *staticFieldDecl?* **then** *declaration(i).staticVar := var*
else *targetObj.fieldVar(declaration(i)) := var*
if *initExpr(i).def?* **then** *var.value := initExpr(i).result*
Proceed Sequentially

Object finalization and destruction

rule *FinalizeObjects*:
if $self.curTask = FinalizeObjects$ **then**
choose *obj* **in** *Object*
if $obj.finalizeMode = finalized$ and *obj.unreachable?* **then**
Remove *obj*
elseif $obj.finalizeMode = unfinalized$ and *obj.finalizerReachable?* **then**
$obj.finalizable? := true$
elseif $obj.finalizeMode = finalizable$ **then**

$obj.finalizeMode := finalized$
Invoke Method $obj.class.method("finalize", \emptyset)$
endif
endchoose

8 Threads

8.1 Preliminaries

In a Java program, multiple agents or *threads* may execute various parts of the program simultaneously. These agents, once created and initiated explicitly in the program code, may execute serially but concurrently on a single processor, or in parallel on multiple processors. As threads may access and update common data, the order of their actions affects the results of an execution. Java establishes some conditions on the interaction of threads with respect to shared data, but implementations of Java may differ in their policies of thread scheduling. This introduces an element of nondeterminism: executions of a multi-threaded program on different implementations of Java may produce different results, if there is data shared among the threads.

During its execution, a thread performs actions on variables: it may *use* the value of a variable, or it may *assign* a value to a variable. Each thread has a *working memory* where it keeps temporary *working copies* of used and assigned variables. There is also a *main memory* which contains the *master copy* of each variable. Threads use and assign variable values by respectively reading from and writing to main memory.

A thread's *execution engine* (which executes code) and its *working memory manager* (which handles the transfer of data between the working memory and main memory) are separate agents. A use or assign action is a tightly coupled interaction between a thread's execution engine and its working memory manager; the used or assigned value is transmitted in a single atomic action. In contrast, the transfer of data between a thread's working memory and main memory is a loosely coupled interaction. In transferring a value from main memory to a thread's working memory, the main memory manager issues a *read* action, which transfers the master value to the working memory. This is followed by a *load* action by the working memory manager, which installs the value into working memory. A transfer in the opposite direction begins with a *store* action by the working memory manager, which transfers the working value to main memory. This is followed by a *write* action by the main memory manager, which installs the value into main memory.

Rule for actions of a thread

- The actions of a single thread are totally ordered. (Threads view the actions of a single thread T in the order in which T performed them.)

Rule for actions on a variable

- The actions by the main memory manager on any single variable are totally ordered. (Threads view the actions of main memory in the order in which it performed them.)

Rules for actions of a thread on a variable

- A use or assign action by T is issued only when dictated by T's code, and in the order dictated by the code.
- If T assigns to x and later loads from x, then it must store to x in the interim. (This ensures that if T issues a sequence of assigns to x, at least the latest value is transmitted to the main memory.)
- If T loads from or stores to x, and later stores to x, then it must assign to x in the interim. (This ensures that T issues a store only if there is a new value to transmit to the main memory.)

- If T uses or stores to x, it must first assign to or load from x. (This ensures that the contents of T's working memory come only from the execution engine or the main memory.)

Rules for the interaction of a thread and main memory on a variable

- If T loads from x, main memory must first read x to T, and the load must put the value of the read into T's working memory. (This ensures that each load is "uniquely paired" with a read.)
- If T stores to x, main memory must later write to x, and the write must put the value of the store into main memory. (This ensures that each store is "uniquely paired" with a write.)
- Let a_1 be a load from or store to x, and let b_1 be the corresponding read or write. Let a_2 be another load from or store to x, and let b_2 be the corresponding read or write. If T issues a_1 before a_2, then main memory issues b_1 before b_2. (This ensures that a thread's actions on a variable are performed in the order that the thread requested.)

8.2 ASM J_5

The ASM J_5 has a compiler agent with module C_5, an executor agent with module R_5 for each element of the *Frame* universe, a single *main memory* agent with module $MainMemory$, and a *working memory* agent with module $WorkingMemory$ for each element of the universe *Thread*.

8.3 Function and macro definitions

$Thread$	Universe of threads.
$Frame.thread : Thread$	Thread associated with the frame.
$Variable.masterValue : Value$	Master copy of variable in main memory.
$Thread.workingValue(Variable) : Value$	Value for variable in thread's working memory.
$Thread.storePending?(Variable) : Boolean$	Is there a new value for the variable, to be stored by the thread's working memory?
$Thread.curWMIndex : Nat$	Current timestamp value for actions by this thread's working memory.
$Thread.curMMIndex : Nat$	Current timestamp value for actions by main memory on behalf of this thread.
$Store$	Universe of store actions.
$Read$	Universe of read actions.

Different threads may throw different exceptions concurrently. Thus the functions *normal*? and $curException$ take an additional *Thread* argument. In all previous ASM rules, add the term $Self.thread$ as an argument to all occurrences of $Normal?$ and $curException$.

macro *Action*: $Store \cup Load$

$Action.MMIndex : Nat$	Timestamp recording when main memory processed this store/read.
$Store.WMIndex : Nat$	Timestamp recording when working memory processed this store.
$Read.loaded? : Boolean$	Has this read been loaded by the recipient's working memory?

macro Access *var*:
(Access value of variable from working memory, if it exists there.)
if *lvalue*? **then**
    $result(self) := var$
    Proceed Sequentially

elseif $self.thread.workingValue(var).def?$ **then**
    $result := self.thread.workingValue(var)$
    Proceed Sequentially

macro Assign val To var:
(Assign new value to variable in working memory,
signaling that a store must copy this value to main memory.
$self.thread.workingValue(var) := val$
$self.thread.storePending?(var) := true$

macro $R.loadable?$:
(May the value of read action R be loaded by the thread's working memory?
Only if all previous stores by this thread to this value have been written by the main memory,
and only if this is the most recent read of the variable.)
$(\forall s : s.isA?(Store)$ and $s.var = R.var$ and $s.thread = R.thread) s.MMIndex < R.MMIndex$
and $(\forall r' : r'isA?(Read)$ and $r'.var = R.var$ and $r'.thread = R.thread$ and $r'.loaded?)$
    $r'.MMIndex < R.MMIndex$

macro $S.written?$:
(Has main memory written the value of store action S?
$S.MMIndex.def?$

macro $S.writeable?$:
(May the value of store action S be written by main memory?
Only if all previous stores by this thread to this value have been written by main memory.)
$(\forall s' : s'.isA?(Store)$ and $s'.thread = S.thread$ and $s'.var = S.var$ and $S'.WMIndex < S.WMIndex)$
    $s'.written?$

8.4 Rule definitions

rule $WorkingMemory$:
choose $var : var.isA?(Variable)$
    if $self.storePending?(var)$ **then**
        extend $Store$ **with** s
            $s.thread := self$
            $s.var := var$
            $s.value := self.workingValue(var)$
            $s.WMIndex := self.curWMIndex$
        $self.storePending?(var) := false$
        $self.curWMIndex := self.curWMIndex + 1$
    else choose $r : r.isA?(Read) : r.thread = self$ and $r.var = var$ and $r.loadable?$
        $self.workingValue(var) := r.value$
        $r.loaded? := true$

rule $MainMemory$:
choose among

    choose $s : s.isA?(Store) : s.writeable?$
        $s.MMIndex := t.curMMIndex$
        $s.var.masterValue := s.value$

$s.thread.curMMIndex := s.thread.curMMIndex + 1$

choose $var, t : var.isA?(Variable)$ and $t.isA?(Thread)$
  extend *Read* **with** r
    $r.thread := t$
    $r.var := var$
    $r.value := var.masterValue$
    $r.MMIndex := t.curMMIndex$
  $t.curMMIndex := t.curMMIndex + 1$

9 Locks and Wait sets

9.1 Preliminaries

If multiple threads execute a common code region that accesses shared data, the result of the execution may not be determinable from the Java specification. A programmer may wish to impose determinism upon such regions by restricting them to one thread at a time. To ensure that only one thread is granted access, any thread trying to enter the region must first acquire a *lock*. Each object has an associated lock. By issuing a *lock* action, a thread requests a claim on a lock. Only one thread at a time may have a claim on a lock, and a thread may gather multiple claims. An *unlock* action relinquishes one claim on a lock. When a thread's claims on a lock increase from zero, we say that it *gains* the lock; likewise, when a thread's claims on a lock decrease to zero, we say that it *releases* the lock. A lock or unlock action is a tightly coupled interaction between a thread's execution engine and the main memory manager.

Rule for actions on a lock

- The actions by the main memory manager on any single lock are totally ordered.

Rules for the actions of a thread on a lock

- If T gains a lock on l, then for all threads other than T the number of lock and unlock actions on l must be equal. (This ensures that one thread at a time may own a lock; a thread may obtain multiple claims on a lock and surrenders ownership only when it has given up all claims.)
- If T relinquishes a lock on l, then the number of T's lock actions on l must be greater than the number of its unlock actions on l. (This ensures that a thread may not relinquish a claim on a lock if it has none.)

Rules for the actions of a thread on a lock and a variable

- If T assigns to x and later relinquishes a lock on l, in the interim it must first store to x and then main memory must write to x.
- If T gains a lock on l and later uses or stores to x, then one of two things happens in the interim:
  - T assigns to x, or
  - main memory reads from x and later T loads from x.

When a thread reaches a state from which it cannot progress by itself, it may temporarily give up control to other threads. In doing so, it may need to release a lock so that other threads may use it. The method `wait` of class Object releases the lock on an object and temporarily disables the thread. Other threads may then signal to the thread that further progress is possible through the `notify` or `notifyAll` methods. In

this way, a waiting thread may resume execution when it is possible, without repeatedly checking its state (and possibly regaining a lock).

Associated with each object is a *wait set*. This set, empty when the object is created, contains all threads waiting for the lock on the object.

The method `wait` is invoked by a thread T on an object *obj*. Let n be the number of claims the T has on the lock of *obj*. If n is zero, an IllegalMonitorStateException is thrown. Otherwise, n unlock actions are performed, and t is added to the wait set of *obj* and disabled. When t is re-enabled (through a `notify` or `notifyAll`), it attempts to regain n claims on the lock; it may need to compete with other threads to do this. Once the lock claims are restored, the method terminates.

The method `notify` is also invoked by a thread T on an object *obj*. If the number of claims that T has on the lock of *obj* is zero, an IllegalMonitorStateException is thrown. Otherwise, if the wait set of *obj* is not empty, one thread is removed from it and enabled. The method `notifyAll` operates similarly but removes and enables all threads from the wait set. Neither method releases any of T's claims on the lock of *obj*.

9.2 ASM J_6

The ASM J_5 has a compiler agent with module C_5, an executor agent with module R_5 for each element of the *Frame* universe, a single *main memory* agent with module $MainMemory$, and a *working memory* agent with module $WorkingMemory$ for each element of the universe *Thread*, and a *lock manager* agent with module $LockManager$ for each element of the universe $Variable$. The module for each executor has the guard $Self.waitsFor.undef?$ and $Self.active?$.

9.3 Function and macro definitions

$Thread.numLocks(Object) : Nat$	Number of locks the thread has on the object.
$LockRequest$	Universe of lock requests.
$UnlockRequest$	Universe of unlock requests.
$Thread.waitsFor : Object$	Object for which the thread is waiting.
$Thread.oldNumLocks : Nat$	Number of locks that the waiting thread had on the object before it began waiting.

macro *Request*: $LockRequest \cup UnlockRequest$

$Request.thread : Thread$	Thread issuing this request.
$Request.obj : Object$	Object of request.

macro Issue Lock Request:
(Issue a request for a lock.)
if $synchExpr.result(Self) = null$ **then** Throw Exception Of Class $NullPointerException$
else
    extend $LockRequest$ **with** r
        $r.thread := self.thread$
        $r.obj := synchExpr.result(Self)$
    Proceed Sequentially

macro Synchronize:
(Wait until a lock for the object has been granted, then proceed.)
if $thread.numLocks(synchExpr.result(Self)) > 0$ **then** Proceed Sequentially

macro Issue Unlock Request:

(Issue an unlock request for a lock on the given variable.)
extend *UnlockRequest* **with** *r*
    r.thread := *self.thread*
    r.obj := *synchExpr.result*(*Self*)
Proceed Sequentially

9.4 Rule definitions

rule Lock Manager:
choose among

    choose $t : t.isA?(Thread) : (\forall t' : t'.isA?(Thread)$ and $t' \neq t)t'.numLocks(self) = 0$
        choose $l : l.isA?(LockRequest) : l.thread = t$ and $l.obj = self$
            if (not *t.storePending*?(*self*))
                and not $(\exists r : r.isA?(Read))r.thread = t$ and *r.loadable*? **then**
                $t.numLocks(self) := t.numLocks(self) + 1$
                Remove *l*

    choose $t : t.isA?(Thread) : t.numLocks(self) > 0$
        choose $u : u.isA?(UnlockRequest) : u.thread = t$ and $u.var = self$
            if $(\forall s : s.isA?(Store)$ and $s.thread = t)s.written?$ **then**
                $t.numLocks(self) := t.numLocks(self) - 1$
                Remove *u*

if $curTask = waitUnlock$ **then**
    if $self.thread.numLocks(self.targetObj) > 0$ then
        $self.thread.waitsFor := self.targetObj$
        $self.thread.oldNumLocks := self.thread.numLocks$
        do-forall $i : i < self.thread.numLocks(self.targetObj)$
            Unlock *self.targetObj*
        $curTask := waitRelock$
    else
        Throw Exception Of Class *IllegalMonitorStateException*

if $curTask = waitRelock$ **then**
    if $self.thread.numLocks(self.targetObj) = 0$ **then**
        do-forall $i : i < self.thread.oldNumLocks(self.targetObj)$
            Lock *self.targetObj*
            $curTask := waitReturn$

if $curTask = waitReturn$ **then**
    if $self.thread.numLocks = self.thread.oldNumLocks$ **then**
        Deactivate Current Frame

if $curTask = notify$ **then**
    choose *t* **in** *Thread* **with** $thread.waitsFor(targetObj)$
        $t.waitsFor := undef$
    Deactivate Current Frame

if $curTask = notifyAll$ **then**
    do-forall $t : t.isA?(Thread) : thread.waitsFor(targetObj)$
        $t.waitsFor := undef$

Deactivate Current Frame

References

[App] AppletMagic home page. `http://www.appletmagic.com`. Intermetrics, Inc.

[ASM] ASM home page. `http://www.eecs.umich.edu/groups/gasm/`. University of Michigan.

[BGS97] A. Blass, Y. Gurevich, and S. Shelah. Choiceless polynomial time. Technical Report CSE-TR-338-97, University of Michigan, 1997.

[Bör95] E. Börger, editor. *Specification and Validation Methods*. Oxford University Press, 1995.

[BS97] E. Börger and Wolfram Schulte. Proving the correctness for compiling Java programs to Java VM code. In *Proceedings of IFIP Working Group 2.2 Meeting*, 1997.

[Gem] GemMex home page. `http://www.first.gmd.de/~ma/gem/index.html`. Matthias Anlauff, GMD FIRST.

[GH93] Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M.M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 274–309. Springer, 1993.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari guide. In Börger [Bör95], pages 9–36.

[Gur97] Y. Gurevich. May 1997 draft of the ASM guide. University of Michigan, 1997.

[Jav] Java glossary. `http://java.sun.com/docs/glossary.html`. Sun Microsystems.

[KP97a] P.W. Kutter and A. Pierantonio. The formal specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[KP97b] P.W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

[Wal95] C. Wallace. The semantics of the C++ programming language. In Börger [Bör95], pages 131–164.