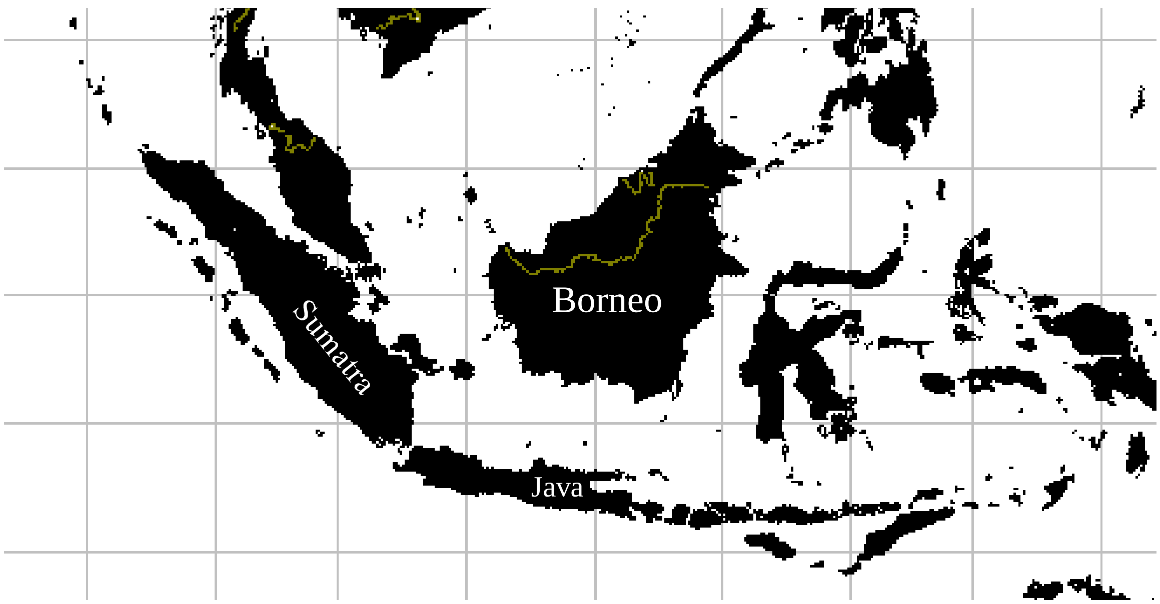
Borneo 1.0.2†

Adding IEEE 754 floating point support to Java



Joseph D. Darcy

revised version of a M.S. Project submitted to the
Computer Science Division
University of California, Berkeley
May 22, 1998

† Formerly known as Teak. In the future will be known as Kalimantan.

Borneo†
Adding IEEE 754 floating point support to Java

Joseph D. Darcy

† This material is based upon work supported under a National Science Foundation Graduate Fellowship.
Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1. Abstract

The design of Java relies heavily on experiences with programming languages past. Major Java features, including garbage collection, object-oriented programming, and strong static type checking, have all proven their worth over many years. However, Java breaks with tradition in its floating point support; instead of accepting whatever floating point formats a machine might provide, Java mandates use of the nearly ubiquitous IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985). Unfortunately, Java's specification creates two problems for numerical computation: only a strict subset of IEEE 754's required features are supported by Java and Java's bit-for-bit reproducibility goals for floating point computation cause significant performance penalties on popular architectures.

Java forbids using some distinguishing features of IEEE 754, features designed to make building robust numerical software by numerical experts and novices alike easier than in the past. Only simple floating point features common to IEEE 754 and obsolete floating point formats are allowed.

Legitimate differences exist among various standard-conforming realizations of IEEE 754. For example, the x86 processor family supports the IEEE 754 recommended 80 bit double extended format in addition to the float and double formats found on other architectures. In many instances, using the double extended format for intermediate results leads to more robust programs. To support its "write once, run anywhere" goals, Java specifies that only the float and double formats be used for intermediate results in numeric expressions. For recent x86 processors to emulate *exactly* a machine that only uses float and double entails a significant performance penalty; over an order of magnitude degradation has been reported. An analogous situation arises on architectures such as the PowerPC that support a fused multiply accumulate instruction; Java semantics preclude using a hardware feature that would usually give more accurate answers faster. However, even numerical analysts do not need or desire exact reproducibility in all cases. The disallowed x86 features were designed to allow numerically unsophisticated programs to have a better likelihood of getting reasonable results.

To address these concerns, the Java dialect Borneo is able to express all required features of IEEE 754. Borneo also aims to run efficiently on multiple hardware implementations of IEEE 754 and to allow convenient construction of new numeric types.

2. Introduction

Since the development of FORTRAN in the 1950's, floating point computation has been an important concern of computer users. Building on FORTRAN, later languages, such as ALGOL 60, provide more formal descriptions of the syntax and semantics of valid programs. However, due to the variety of architectures of the time, in ALGOL 60:

> No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. [75]

Therefore, the same Algol 60 program compiled and run on different architectures can produce different output due to varying range, precision, and other properties of a particular floating point format. In such a heterogeneous environment, a reasonable approach to cross-architecture portability is for a programming language to express those operations common to all (or most) contemporary architectures. This approach to supporting floating point in programming languages persists even though only one floating point standard is currently widely used.

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985) [4] was adopted in part to diminish the diversity of floating point formats in use during the early 1980's. One of the standard's design goals is to

> Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. [4]

Since its introduction, IEEE 754 has become universally available on all significant microprocessors for PC's and workstations (Intel x86 line and clones, Motorola 68000 series, Power PC, HP PA RISC, SPARC, SGI MIPS, and DEC Alpha, among others). IEEE 754 provides numerous features advantageous both to the numerical analyst and to the more casual programmer. With a small amount of education, programmers could directly use and benefit from IEEE 754 features now known only to experts. Even if not employed explicitly, many users could run efficient numerical libraries written to exploit IEEE 754's advanced features [25].

Although the standard's features are widely supported in hardware, these features are not supported in most current programming languages, including Java, even though programming language support for IEEE 754 was discussed before the standard was adopted [32], [33]. Even the recent ISO Standard for Language independent arithmetic [52] does not call for full IEEE 754 conformance. This lack of language support has not gone unnoticed or unlamented [56], [96]. Availability of IEEE 754 features would allow more robust numerical programs to be written more easily by both novice and expert programmers. Due to the ubiquity and utility of IEEE 754, all IEEE 754 specific features should now be supported by programming languages.

2.1. Portability and Purity

Be thou as chaste as ice, as pure as snow, thou shalt not escape calumny.
Get thee to a nunnery, go.
—William Shakespeare, Hamlet, Act III, Scene i

Java [38] aims to provide "write once, run anywhere" portability by rigorously defining the semantics of the language, specifying many details left to the implementor's discretion in languages such as C [65] and C++ [89]. Intuitively, a portable program produces "the same" result on different platforms. However, the meaning of "the same" is not well defined when platform dependent details are exploited. To finesse the problem of defining portability, Sun has proposed "100% Pure Java" as an alternative metric. Instead of measuring program portability in terms of what it accomplishes, 100% Pure Java defines purity in terms of what features a program uses. By confining a program to a subset of Java's features, purity is intended to predict portability. However, many key aspects of Java programming are platform dependent, from the characters that terminate a printed line to the size of the GUI screen. To work around such issues, Sun's *100% Pure Java Cookbook* [43] recommends using methods provided in the Java standard library to query the environment.

While many microprocessors conform to the IEEE 754 standard, there is non-negligible variance among the implementations. For other properties that vary from system to system, Java provides an abstraction layer for programmers to write portable code, for example, `println` always generates the correct platform-dependent line termination character and a `LayoutManager` can be used to arrange GUI elements on different sized displays.

However, Java does not provide similar facilities to deal with legitimate differences among IEEE 754 conforming processors. For example, there is no mechanism to determine the availability of the double extended format. Java's least common denominator approach unnecessarily burdens the x86 with usually unneeded and unwanted exact reproducibility [22]. As with other system-dependent properties, there should be established mechanisms to query the capabilities of the floating point environment and use the resources appropriately, either for exact reproducibility or better performance, while in all cases being predictable.

The spirit of the IEEE standard does not dictate rigid conformity; rather, the standard intends to "Enable rather than preclude further refinements and extensions" [4]. Using a disciplined approach, such as explicitly storing intermediate results, (with a few caveats) it is possible to achieve exact reproducibility with existing processors and compilers. Reproducibility may extract a price in terms of code clarity and speed. The IEEE standard allows, but does not mandate, exact reproducibility. Often the speed of a calculation exceeds the importance of the exact result. For example, a matrix multiplication routine tuned for a particular architecture and memory configuration can be three to four times faster than naive triply-nested loops [8]. Hardware vendors spend considerable time and effort, including hand coding assembly routines, to achieve these performance gains. Code optimized for one architecture when run on a different architecture can be *slower* than naive code on the second architecture.[1] The answers from the optimized routines are not exactly the same, but the great time saving is deemed worthwhile. Therefore, linguistically enforced exact reproducibility of scientific calculations across architectures is an impractical goal.

[1] A double precision matrix multiply tuned for the UltraSPARC I (32 floating point registers, 16K L1 cache, 512K L2 cache) using PHiPAC [8] ran slower than naive nested loops on the Pentium Pro (8 floating point registers, 8K L1 cache, 256K L2 cache) for square matrices smaller than 500x500. For larger matrices, both programs have comparable throughput. For square matrices smaller than 200x200, PHiPAC matrix multiply tuned for the Pentium Pro is approximately twice as fast as the naive code on the Pentium Pro and six times as fast as the UltraSPARC code.

2.2. Goals of Borneo

To correct Java's existing floating point deficiencies, the Borneo language extends Java so it is able to express the entire IEEE 754 floating point standard. For the remainder of the document, occurrences of "the standard" refer to the IEEE 754 Standard for Binary Floating Point Arithmetic [4]. The acronym JLS is used to refer to *The Java™ Language Specification* [38]. Borneo has a number of objectives:

1. Expressing all required and recommended IEEE 754 features.
2. Allowing convenient creation of user defined numeric types exploiting IEEE 754 features.
3. Minimizing changes to Java and the Java Virtual Machine (JVM) to ensure upwards compatibility.

Borneo's primary goal is allowing all the required features of the standard to be expressed while maintaining the existing advantages of Java. If possible, the language extensions should also support the standard's many recommended features. To achieve reasonable performance, processor specific features must be used. However, in some cases, the programmer may need to sacrifice speed for reproducibility. Since performance is usually a concern, it must be possible to generate efficient executable code, including JVM bytecode and native machine code.

For numeric programmers, the ability to add custom numeric types is quite useful; the language should be extensible enough so that new numeric classes can behave like and be operated on as conveniently as built-in floating point types.[2] However, the language should not become so unwieldy that building compilers and related tools is intractable.

Borneo preserves the semantics of existing Java programs. A Java program, *P*, compiled and run under Borneo semantics cannot be differentiated by a pure Java program from *P* compiled and run under Java semantics. However, in general, a Borneo program can observe differences in behavior from the same program compiled under Java and Borneo semantics. Borneo compilers accept programs with both "`.java`" and "`.born`" extensions. Although Borneo adds keywords not found in Java, these keywords are not available in Java programs with a "`.java`" extension; Borneo programs must uses a "`.born`" extension to access the new keywords. All valid "`.java`" programs are valid Borneo programs. Where possible, Borneo uses existing Java language constructs to preserve the flavor of Java. The results of arithmetic operations do not depend on whether a program is interpreted or compiled. Finally, programs are not unduly penalized for unused features.

2.3. Brief Description of an IEEE 754 Machine

Before discussing the language extensions, the features of IEEE floating point relevant to Borneo are briefly summarized. The standard discusses these features in much greater detail and should be referred to for full explanations. The related standard IEEE 854 [18] gives the reasoning behind some of both standards' design decisions.

2.3.1. IEEE 754 Floating Point Operations and Values

Certain capabilities in the standard are required while others are optional but recommended. The standard's arithmetic operations include base conversion, comparison, addition, subtraction, multiplication, division, remainder, and square root. One increasing popular extension to IEEE 754 is the fused mac (Multiply-ACcumulate) operation. A fused mac multiplies two numbers exactly (no overflow, underflow, or rounding) and adds a third number to the product, producing a single rounding error at the end. A fused mac can give a different answer than chained multiply and add operations on the same initial arguments.

IEEE 754 defines three relevant floating point formats of differing sizes; single double, and double extended (see Table 1).[3] The 32 bit single format is mandatory and the optional 64 bit double format is ubiquitous on conforming processors. Some architectures, such as the x86 and 68000 lines of processors, include support for a third optional, but recommended, 80 bit double extended format. For the remainder of the document, `double extended` refers to this particular 80 bit double extended format (other double extended formats are possible). Wider formats have both a larger range and more precision than narrower ones.

[2] In fact, certain of the standard's features were included specifically to construct new numeric types. One strong motivation for including directed rounding was to "Provide direct support for interval arithmetic at a reasonable cost" [4].

[3] The 40 bit single extended format used on some DSP chips will not be discussed.

Table 1 — Constraints on IEEE 754 number formats.

	Total size (bits)	bits for *p*	bits for *E*	E_{max}	E_{min}
`single`	32	24	8	127	–126
`double`	64	53	11	1023	–1022
`double extended`	≥ 79	≥ 64	≥ 15	≥ 16383	≤ –16382

The finite values representable by an IEEE number can be categorized by an equation with two integer parameters *k* and *n*, along with two format-dependent constants *N* and *K* [56]:

$$\text{finite value} = n \cdot 2^{k+1-N}.$$

The parameter *k* is the unbiased exponent with *K*+1 bits. The value of *k* ranges between $E_{min} = -(2^K-2)$ and $E_{max} = 2^K-1$. The parameter *n* is the *N*-bit *significand*, similar to the mantissa of other floating point standards. To avoid multiple representations for the same value, where possible the exponent *k* is minimized to allow *n* to have a leading 1. When this "normalization" process is not possible a *subnormal* number results. Subnormal numbers (also called denormals) have the smallest possible exponent. Unlike other floating point designs, IEEE 754 uses subnormal numbers to fill in the gap otherwise left between zero and the smallest normalized number. Including subnormal numbers permits *gradual underflow*.

Floating point values are encoded in memory using a three field layout. From most significant to least significant bits, the fields are sign, exponent, and significand:

$$\text{finite value} = (-1)^{sign} \cdot 2^{exponent} \cdot significand.$$

For a normalized number, the leading bit of the significand is always one. The single and double formats do not actually store this leading *implicit bit*. (The `double extended` format explicitly stores this implicit bit so the format is 80 bits wide instead of 79.) Subnormal values are encoding with an out-of-range exponent value, $E_{min} - 1$.

Besides the real numbers discussed above, IEEE 754 includes *special values* NaN (Not a Number) and ±∞. The special values are encoded using the out-of-range exponent $E_{max} + 1$. The values ±0.0 are distinguishable although they compare as equal. Together, the normal numbers, subnormals, and zeros are referred to as finite numbers. Except for NaNs, if the bits of an IEEE floating point numbers are interpreted as signed-magnitude integers, the integers have the same lexicographical order as their floating point counterparts.

NaN is used in place of various invalid results such as 0/0.[4] NaN is needed to make floating point arithmetic closed under the usual arithmetic operations. Actually, many different bit patterns encode a NaN value; the intention is to allow extra information, such as the address of the invalid operation generating a NaN, to be stored into the significand field of the NaN. These different NaN values can be distinguished only through non-arithmetic means. For assignment and IEEE arithmetic operations, if a NaN is given as an operand the same NaN must be returned as the result; if a binary operation is given two NaN operands one of them must be returned.

The standard defines the comparison relation between floating point numbers. Besides the usual, >, =, and <, the inclusion of NaN introduces an *unordered* relation between numbers. A NaN is neither less than, greater than, nor equal to any floating point value (even itself). Therefore, `NaN > a`, `NaN == a`, and `NaN < a` are all false. A NaN is unordered compared to any floating point value.

By default, an IEEE arithmetic operation behaves as if it first computed a result exactly and then rounded the result to the floating point number closest to the exact result. (In the case of a tie, the number with the last significand bit zero is returned.[5]) While rounding to nearest is usually the desired rounding policy, certain algorithms, such as interval arithmetic (see section 6.4.7), require other rounding conventions. To support these algorithms, the IEEE standard has three additional rounding modes, round to zero, round to +∞, and round to –∞. The rounding mode can be set and queried dynamically at runtime.

[4] The standard actually defines two kinds of NaNs, quiet NaNs and signaling NaNs. Since they cannot be generated as the result of an arithmetic operation, Borneo ignores signaling NaNs and assumes all NaNs are quiet.
[5] The VAX uses round to nearest, but in the case of a tie always rounds toward zero. This adds a slight statistical bias to computations and causes troublesome drift in some calculations.

2.3.2. IEEE 754 Exceptional Conditions

When evaluating IEEE 754 floating point expressions, various exceptional conditions can arise. The conditions indicate events have occurred which may warrant further attention (although Java omits flags are forbids traps). The five exceptional events are, in decreasing order of severity,

1. **invalid**, NaN created from non-NaN operands; for example 0/0 and $\sqrt{-1.0}$.
2. **overflow**, result too large to represent; depending on the rounding mode and the operands, infinity or the most positive or most negative number is returned. Inexact is also signaled in this case.
3. **divide by zero**, non-zero dividend with a zero divisor yields a signed infinity exactly.
4. **underflow**, a subnormal value has been created.
5. **inexact**, the result is not exactly representable; some rounding has occurred (actually a very common event).

The standard has two mechanisms for dealing with these exceptional conditions: sticky flags and traps. Flags are mandatory. Although optional, trapping mode is widely implemented on processors conforming to IEEE 754 and therefore has support in Borneo. The mechanism used for each exceptional condition can be set independently. In this document, signaling a condition refers to either setting the corresponding flag or generating the appropriate trap, depending on the trapping status. When sticky flags are used, arithmetic operations have the side effect of ORing the conditions raised by that operation into a global status flag. The sticky flag status can also be cleared and set explicitly. When a condition's trap is enabled, the occurrence of that condition causes a hardware trap to occur and a trap handler to be invoked. The standard requests that the trap handler be able to behave like a subroutine, computing and returning a value as if the instruction had executed normally.

Trapping mode requires some additional information beyond notification of an exceptional event. For trapping on overflow and underflow, the trap handler is to be able to return an exponent-adjusted result, a floating point number with the same significand as if the exponent range were unbounded but with an exponent adjusted up or down by a known amount so the number is representable. Instead of allowing users to specify their own trap handlers, Borneo integrates trapping on floating point operations into the existing Java exception mechanism. Therefore, it is not possible to return to the location which caused a floating point trap. Methods of the overflow and underflow exceptions return the exponent-adjusted result. A single operation can cause both overflow and inexact or both underflow and inexact to be signaled. If both of the involved conditions are being trapped on, overflow and underflow take precedence over inexact.

Underflow is signaled differently depending on the trapping status. If trapping on underflow is enabled, any attempt to create a subnormal number will cause a trap. In non-trapping mode, only a subnormal result that is also inexact will raise the underflow flag. A common feature of processors implementing IEEE 754 is a non-conforming flush to zero mode where all subnormal results are replaced with zero. It is considerably easier to make flush to zero fast in hardware than to make gradual underflow fast.

2.3.3. Java floating point conformance

Java's `float` type corresponds to the IEEE single format and Java's `double` type corresponds to the double format. Java specifies that all arithmetic operations occur under the round to nearest rounding mode[6] with no facility to change rounding modes. In Java, non-trapping mode is always on; floating point operations cannot throw exceptions and there is no mechanism to inspect or clear the sticky flags. Since a Java program does not allow inspection of the flag state, there is no obligation for a Java compiler to maintain the flags in a state consistent with all portions of the program having been executed. In particular, compile time optimizations such as constant folding can be performed without regard to side effects to the flag state. Borneo includes the IEEE 754 features lacking in Java (shown in Table 2) while adding awareness of the interplay between rounding modes, status flags, and evaluating arithmetic expressions.

[6] When converting a floating point number to an integer, Java rounds toward zero instead of to nearest, the same behavior as FORTRAN and ANSI C.

Table 2 — Java's IEEE 754 conformance.

IEEE Features	Status in Standard	Java Conformance
directed rounding	required ([4] §4.1, §4.2)	explicitly forbidden (JLS §4.2.4)
sticky flags	required ([4] §7)	omitted from specification
floating point exceptions (trapping mode)	recommended ([4] §8)	explicitly forbidden (JLS §4.2.4)
extended floating point formats	recommended ([4] §3.4)	omitted from specification
non-signaling comparison operators	recommended ([4] §5.7)	omitted from specification
fused mac	not part of IEEE 754	omitted from specification

The standard also recommends an IEEE 754 environment include several utility functions that perform basic tasks on floating point numbers. Java currently includes two of the ten recommended functions; Borneo adds the rest. The additional methods include `nextAfter`, `scalb`, and `logb`. The `nextafter` function can be used to find the floating point numbers adjacent to a given floating point number, useful for perturbing data and defining floating point constants. The `scalb` function scales a floating point number by a power of two (in effect `scalb` attempts to change the exponent of a number but not its significand). Scaling a number this way can lose precision if the result is subnormal; therefore, `scalb` could be affected by the dynamic rounding mode. It is also possible to use so large or so small a scaling factor that the result cannot be represented, even using a wrapped exponent. However, the standards' descriptions of the recommended functions are terse and the standards do not specify the desired behavior in these extreme cases. Also, IEEE 754 and 854 give different specifications for the behavior of `logb` on subnormal arguments. The `logb` function returns the unbiased exponent of a floating point number.

2.4. Language Features for Floating Point Computation

While support for floating point computation in programming languages has long been commonplace, the details of floating point support have often been overlooked in language specifications. In the following sections, issues related to floating point support are identified and the ways Java and Borneo address these needs are discussed. For further information, appendix 9.3 catalogs the variations in how notable languages support floating point.

2.4.1. Decimal to Binary and Binary to Decimal Conversion

> The [Windows 3.1] calculator program that Microsoft includes with every copy of Windows makes math errors… If you try to subtract 2.00 from 2.01, it gives an answer of 0.00. And it mishandles other numbers ending in .01. Microsoft knows of the problem, but doesn't plan to fix it until the next version of Windows ships next year (unless consumers raise a hue and cry). [Walter S. Mossberg, *Wall Street Journal* Thursday December 15, 1994, B1]
>
> *Microsoft to Fix a Flaw In Windows 3.1 Calculator*
> REDMOND, Wash. – Microsoft Corp. said it will offer software that fixes a flaw in the calculator portion of its Windows 3.1 operating system. The company said the software is being developed in response to a column in this newspaper. [*Wall Street Journal*, Monday December 19, 1994, B9]

Since nearly all computer arithmetic is in binary while floating point literals are in decimal, base conversion must be done to enter or display floating point numbers. While finite-length integer values can always be converted without loss of precision between different bases, in general, finite-length fractional values cannot. A fraction finitely representable in one base may be an infinitely repeating fraction in another. For example, in base 3 one third is just 0.1 as opposed to 0.33333… in base 10. Matula [69], [70] discusses necessary and sufficient conditions for lossless base conversion. Reading in and displaying floating point values provides the main interface between users and floating point numbers; if base conversion is suspect, the correctness of the remaining floating point infrastructure is obscured.

Recognizing that a decimal floating point number may not have an exact binary representation, a reasonable expectation is that the closest binary floating point number is used instead. Any floating point base conversion algorithm having this property is *correctly rounded*. However, many language standards do not require correctly rounded decimal to binary conversion (nor does IEEE 754 in all cases).[7] Since decimal representations of floating

[7] The standard does give constraints on base conversion. For a wide range of values, the conversion must be correctly rounded. The IEEE 754 committee expected base conversion to be included in floating point co-processors

point numbers can be used on computers adhering to different floating point standards, correctly rounded binary to decimal conversion is important for portability across heterogeneous floating point systems. Having different results from compile-time and runtime conversion of literals can also lead to inconsistent results.

Although published algorithms exist for both correctly rounded input [12] and output [87], conversion problems persist. Correctly rounded algorithms are also acceptably fast for common cases [11], [34]. While working on the BEEF tests for transcendental functions, it was discovered that the Turbo C 1.0 compiler did not convert "`11.0`" exactly into a floating point number equal to 11 [67]! Unrelated to the infamous Pentium divide bug, the calculator that for years shipped with Microsoft Windows 3.1 produces misleading output for some calculations. Due to a bug in a Microsoft library function, under certain circumstances a value of 0.01 is displayed as 0.00 [68]. The correct value is stored internally; only the display is faulty. First reported in Infoworld in late November 1994, the bug only received widespread attention after being published in the Wall Street Journal column excerpted above.

2.4.2. Floating Point Types

An IEEE 754 compliant architecture can support as few as one floating point format or as many as four different IEEE formats. In practice, IEEE 754 architectures either support two formats (single and double) or three (single, double, and double extended[8]). Other floating point standards also have defined multiple formats; the VAX eventually had four formats, one 32 bit format, two 64 bit formats, and one 128 bit format. Most programming languages have at least one "built-in" floating point type. Since the number of language floating point types may not match the number of available hardware floating point formats, some constraints on the language type to hardware format mapping are required.

New floating point types may also be created, either as totally user-defined types or possibly as subsets or subranges of existing floating point types.

2.4.3. Expression Evaluation

Many familiar laws of arithmetic (associativity of addition and multiplication, distributivity of multiplication over addition, etc.) do not hold for floating point computation. However, language standards may allow optimizing compilers to transform floating point expressions in non-equivalence preserving ways (that hopefully execute faster). Some compilers have documented options that list which rules of arithmetic are broken at different optimization levels. For example, the Sun cc compiler's "`-fsimple=2`" allows `x/y` in a loop to be replaced by `x * z` where `z = 1/y` if `y` has a constant value during the loop execution.[9] A few compilers may even ignore parentheses while evaluating expressions.

Numerous calculations can fail due to overly aggressive optimization. For example, in

$$X = (X + b) - b$$

if X is as large as the few least significant bits of b, a kind of rounded version of X is calculated. If the subtraction is done first instead of the addition, the value of X does not change. Another common idiom where order of evaluation can be crucial is

$$Z = (a*X)*Y$$

The values of X and Y may vary widely with scaling factor a ensuring that overflow and underflow do not occur. If instead the expression is evaluated as

---

so correctly rounded conversion was not required over the entire range of floating point values due to implementation difficulties [59].

[8] The 80 bit `double extended` format is widely implemented in hardware. Another extant IEEE double extended format, 128 bit quad, has instruction level support in some architectures. However, on existing processors implementing such architectures, the actual quad operation trap to software.

[9] `x/c` may not be equivalent to `x * (1/c)` in several ways: if `1/c` is not exact, multiplying by the reciprocal causes two rounding errors instead of a single rounding error for the divide and `1/c` may overflow when `x/c` does not. The two forms can also cause different exceptional conditions. If the numerator and denominator are both zero, `x/c` generates invalid while `x * (1/c)` generates divide by zero and invalid (both expressions evaluate to NaN). If `c` is very small, `1/c` can overflow to infinity (never yielding a normal product) while `x/c` can result in a normal number.

$$Z = a*(X*Y)$$

exceptions designed to be avoided may occur. Alternatively, ($a*X$) may be an exact calculation, with the only rounding error introduced by the second multiplication; using a different evaluation order can increase the calculation's rounding error [59].

For predictable floating point behavior, a language must respect parentheses and not treat computations that are mathematically equivalent on infinitely precise real numbers as computationally equivalent on limited precision floating point numbers. For example, while multiplication of real numbers is associative, multiplication of floating point numbers is not associative. To scrupulously support IEEE 754, optimizations cannot change the sticky flags set or floating point traps caused by evaluating an expression. Other optimizations, such as common subexpression elimination, must be aware of dynamic rounding and trapping modes. However, at times it is also desirable and warranted to sacrifice floating point fastidiousness for increased speed.

In the evaluation of numeric expressions of more than two terms, temporary values are needed to hold the intermediate results. Since these anonymous temporary values are not explicitly declared by the programmer, some convention is needed for determining the types of these locations. Expression evaluation rules are usually more of an issue if a language has multiple floating point types. A related issue is what coercions can be implicitly performed by the compiler. In addition to coercions in expression evaluation, implicit coercions between floating point types may occur during assignment and parameter passing. Several conventions for the typing of anonymous floating point values have been used:

1. *strict evaluation*: If the two operands are of the same type, the result is of that type. If the two operands are of different types, the "narrower" of the two operands is converted to the type of the "wider" operand and the final result is the type of the wider operand. Strict evaluation is used in Java and ANSI C, among other languages. Strict evaluation allows the compiler to easily determine the types of intermediate results since only the types of the operands need to be determined; no other environment needs to be consulted.
2. *widest available*: Some hardware platforms, such as the x86, have a preferred floating point format that runs more naturally than other formats. In the widest available strategy, this type is used for all intermediate expression evaluation. For example, in pre-ANSI C, all floating point expression evaluation takes place in a platform-dependent `double` precision even if all the operands are `float`. In widest available evaluation, expressions are often calculated in a higher precision than the input data. This policy can prevent unsophisticated numerical expressions from misbehaving while also achieving good hardware utilization.
3. *scan for widest (widest needed)*: In scan for widest, the expression is scanned to find the widest numeric type present (including the width of the destination in an assignment), then all leaves of the expression tree are coerced to the widest type. The intention of scan for widest is to use as much program context as possible to determine the most appropriate sizes for temporary values. However, scan for widest complicates determining what types intermediate results should have. Scan for widest has been partially implemented in a modified FORTRAN compiler [24].

2.4.4. Converting between floating point and integer

Languages differ in evaluation rules for mixed integer and floating point expressions. Some have implicit coercion of integers to floating point, others do not. Additionally, when converting floating point numbers to integer, a variety of rounding conventions may be used.

2.4.5. Floating Point State

The IEEE 754 standard defines several pieces of floating point state, the dynamic rounding mode, the current trapping status, and the value of the sticky flags. To fully support the standard, a language must have mechanisms to query and set these values. Some languages have defined their own floating point state variables. For example, in later versions of APL (see section 9.3.2), a program-wide comparison tolerance can be set.

2.4.6. Operator Overloading

Operator overloading is a common language feature allowing programmers to define and call infix functions. Some languages only allow a certain set of built-in operators to be overloaded; others allow novel user-defined operators as well. Often there are restrictions on operators acting on user-defined and built-in types, such as at least one argument must be a user-defined type. For user-defined types, operators may be defined to convert one type to another. Since user-defined types using operators are often numeric types, interaction with built-in numeric literals is important.

New numeric types usually have value semantics, that is, each "number" should not share state with any other number. An easy, but inefficient, way to implement these semantics is to always make a copy of the number object during assignment, parameter passing, and function return. For efficiency reasons, making unnecessary copies of objects should be avoided.

2.4.7. Java and Borneo

Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.
—Sun's buzzword compliant description of Java

Originally called Oak, the Java language is touted as the language for programming the Internet. To support its "write once, run anywhere" goals, Java strictly defines many aspects of the language left undefined or implementation dependent in other languages, such as C. Thread scheduling and the order `finalize` methods are called for garbage collected objects are not rigidly defined by the Java specification.

Java requires correctly rounded decimal to binary and binary to decimal conversion (JLS §3.10.2, §20.9.16, §20.9.17). Java has two floating point types, `float` and `double`, which correspond exactly to the IEEE 754 single and double formats. It is a compile time error to have floating point literals exceed a format's range. The strict evaluation policy is used for expression evaluation; implicit widening conversions occur between integer and floating point types. Rounding to zero is used for converting floating point numbers to integer. Parentheses must be respected, implicit right to left evaluation must be followed, and optimizations must preserve both the value and observable side effects of floating point expressions. However, since Java semantics do not include the IEEE sticky flags or exception handling, optimizations that would change these properties are legal in Java. While requiring IEEE 754 numbers, Java does not support all IEEE 754 features. Using non-default rounding modes is explicitly forbidden as are floating point exceptions (JLS §4.2.4).

Java does not have any operator overloading facilities but method calls are overloaded.

Borneo adds support for all IEEE 754 features while maintaining upwards compatibility with Java. Building on Java's specification, Borneo maintains Java's base conversion requirements and extends the side effects of floating point expression evaluation to include the IEEE 754 sticky flags and exceptions. By default, Borneo uses strict evaluation for expressions, but a block-level language declaration allows widest available to be used instead. In addition to `float` and `double`, Borneo's `indigenous` type designates the `double extended` format on processors supporting that format (and designates `double` elsewhere). Even though the `indigenous` type varies from platform to platform, a Borneo program remains predictable even if not exactly reproducible.

Borneo has declarations to express IEEE 754 features in a convenient, structured manner. Dynamic rounding modes are controlled by a lexically scoped declaration. Sticky flag behavior is included in a method's signature. Borneo also has a new control construct based on sticky flags in addition to library methods to access the sticky flags directly. A scoped language declaration allows floating point exceptions to be thrown. The Borneo library includes all the IEEE recommended functions.

Operator overloading in Borneo allows the creation of user-defined numeric types that can be used nearly as conveniently as the built-in primitive floating point types `float` and `double`. Avoiding past complications, operators on primitive types may not be redefined, but novel operators may be introduced. A programmer can also include existing operators in a new class. The text of an operator indicates its precedence and associativity.

Borneo programs end with a "`.born`" extension instead of the "`.java`" extension used for Java programs. A Borneo compiler also accepts Java programs, but to avoid name clashes, Borneo specific keywords are not recognized as keywords in Java programs. A Java program compiled with a Borneo compiler is compiled with Borneo's floating point semantics; some floating point optimizations legal in Java are not performed by Borneo.

3. Future Work

3.1. Incorporating Java 1.1 Features

The current version of Borneo is based on Java 1.0 as described in JLS. Java 1.1 adds a number of new language features, such as inner classes, as well as many new libraries and interfaces. Borneo has a third floating point type, `indigenous`, not present in Java. The `indigenous` type varies across platforms; it is `double extended` where that format is supported in hardware and `double` elsewhere. The addition of `indigenous` necessitates changes to a number of Java 1.1 features including the Java Native Interface, object Serialization, reflection, and Remote Method Invocation. The Java Native Interface to call native code must be extended to include the `indigenous` type. Additionally, since `indigenous` values are platform dependent, they cannot be serialized with the default serialization; custom `readExternal` and `writeExternal` methods need to be provided in the `Indigenous` class (analogous to the `Float` and `Double` classes). Therefore, the `Indigenous` class should implement the `java.io.Externalizable` interface instead of the `java.io.Serializable` interface. A new `Class` object must be created to represent `indigenous` in the reflection API. The Remote Method Invocation facilities also need to be updated to properly incorporate transmitting `indigenous` values.

3.2. Unicode Support

The Unicode character for infinity, ∞=0x221E, could be used to designate an infinity literal, but due to the lack of widespread Unicode support, having that character alone serve as a literal for infinity is not sufficient.[10]

The Unicode standard includes a number of mathematical operators (characters 0x2200 to 0x22FF), some of which could be useful for operator overloading.

3.3. Flush to Zero

Instead of gradual underflow, some processors can “flush to zero” small values. This flush to zero mode is non-IEEE 754 compliant but runs much more quickly on processors with this feature. Like Java, Borneo requires that gradual underflow be used at all times.

3.4. Variable Trapping Status

Currently in Borneo whether or not an exceptional condition is trapped on in a section of code is a static property of the program. To fully emulate processors which have trapping status as a dynamic property, some mechanism is needed to allow the trapping status of a section of code to be varied at runtime. One possibility is to allow an `enable` declaration (section 6.8.1.3) to take an integer argument, analogous to a `rounding` declaration (section 6.7.2). It must also be possible to query the dynamic trapping status. For example, to implement a class that fully models the behavior of the `double extended` format, it is necessary to query the dynamic trapping status.

3.5. Parametric Polymorphism

Among the new floating point types proposed for the Borneo library, some would have very similar code. For example, classes representing exponent extended floating point numbers, `WideExpFloat`, `WideExpDouble`, and `WideExpIndigenous` would be nearly identical except for the base floating point type used. Templates or some other parametric polymorphism facility could relieve the tedium of maintaining many similar numeric types.

4. Conclusion

I hate quotations. Tell me what you know.
—Ralph Waldo Emerson

While programming languages do not explore a very large design space for floating point support, the details from language to language and compiler to compiler often differ. Important issues such as correctly rounded base conversion are often not even acknowledged in language standards. More recent languages, such as Java, are aware of IEEE floating point and related issues, but often do not guarantee full support. Borneo extends Java to incorporates IEEE 754 features in a structured manner, allowing both the compiler and the programmer to reason about the code.

Borneo largely preserves both the syntax and semantics of Java; the same source code under Borneo has nearly identical semantics as under Java. Java’s existing exception handling mechanism is augmented to deal with floating point exceptions. New scoped declarations are added to control rounding mode and flag state. By default, a Borneo program is subject to less aggressive constant folding and related optimizations than the same program under Java, but the small loss of local optimizations is accompanied by the ability to write much faster, more robust

[10] Similarly, Unicode could even be used to denote a NaN literal by using the Mandarin homophone 難=0x96E3 (which means difficult).

algorithms. Borneo follows the letter and the spirit of IEEE 754, supporting the creation of portable, predictable numerical programs.

5. Acknowledgments

The work that would later become Borneo was begun by the author in the summer of 1996. Significant progress occurred during the spring 1997 semester when working on Borneo became the class project for Professor Kahan's CS279 class, System Support for Scientific Computation. Cedric Krumbein, Howard Robinson, Robert Yung, and Melody Ivory participated in discussions related to Borneo. Cedric Krumbein wrote the initial examples of code using exception handling and contributed to the BVM specification as well as editing and checking early drafts of the Borneo document. Howard Robinson provided the sticky flag and Sturm sequence examples and also helped edit early versions of this document.

The Borneo language and document have benefited from the feedback of many people. Vincent Fernando, John Hauser, and Geoff Pike provided useful comments on an extended abstract of earlier versions of this document. Jim Demmel provided several useful references and suggestions. Ben Liblit drew attention to aspects of the Borneo's operator overloading that required refinement. Jeff Foster's reading of this document was very helpful in identifying issues needing clarification. Eric Grosse and Richard Fateman helped track down some of the last remaining typos. Rich Vuduc ran PHiPAC to help determine the penalty of mis-optimization. Gregory Tarsy's floating point group at Sun, David Hough, Alex Liu, Stuart McDonald, K.C. Ng, Douglas Priest, and Neil Toda has helped and supported the Borneo effort, including having the author as a summer intern. Finally, thanks goes to Alex Aiken and William Kahan for advising and guiding the author throughout the development of Borneo.

6. Borneo Language Specification

The remainder of this document presents the Borneo language specification in terms of changes to Java (section 6) followed by changes to JVM, the Java Virtual Machine, (section 7), and changes to classes in the `java.lang` package (section 8). Section 7 does not give all the necessary details for the JVM changes to be implemented; for example, while the behavior of new instructions is described, opcode assignments are not given. Familiarity with Java and JVM is assumed. Borneo is based on Java 1.0; updating Borneo to incorporate Java 1.1 features is left for future work.

Most second level headings in section 6 directly correspond to new language features. The new language features usually address an IEEE 754 capability lacking in Java; however, other sub-sections, such as operator overloading, deal with language features not directly tied to IEEE 754. For each language feature, first the requirements of using the feature are introduced, followed by a presentation of Borneo's specification for that feature, and concluding with a discussion of alternatives and rationale, often with some examples illustrating intended usage.

The examples and discussion of processors assume an IEEE 754 compliant processor with dynamic rounding modes and dynamic trapping status. The Alpha architecture [83] can encode some rounding modes statically in a field of a floating point instruction; the interaction of this feature with Borneo semantics is noted on a number of occasions throughout the text.

6.1. `indigenous`

Allow me to introduce you to Ceti Alpha V's only remaining indigenous lifeform…
You see, their young enter through the ears, and wrap themselves around the cerebral cortex.
This has the effect of rendering the victim extremely susceptible to suggestion.
—Khan Noonian Singh, Star Trek II: The Wrath of Khan

Borneo adds many new floating point types to Java. One of those types, `indigenous`, is a new primitive type similar to `float` or `double`. The other types are new standard library classes which use Borneo's operator overloading facilities. The `indigenous` type is used by Borneo to represent the `double extended` format on the x86. On most other platforms, `indigenous` is another name for the `double` format. The next section motivates having any representation of the `double extended` format.

6.1.1. Requirements

A language designed to support floating point computation should allow the full capabilities of the underlying floating point hardware to be exploited. The x86 family comprises a large majority of desktop computers worldwide, so from a practical standpoint, supporting that processor line reasonably well is especially important.

On the x86, `double extended` is the most natural format for the processor to work in. Forcing an x86 to act as if it only supported single and double in *all* cases significantly restricts the speed of the x86. By setting a control word, the x86 can be made to round to float or double precision, but that rounding only curtails the significand, not the exponent range; the exponent range is the same as the wider `double extended` format.[11] To make the x86 round a register's exponent as well, it is necessary to issue a store to the appropriate format followed by a load back into the register. Such extra loads and stores have the unfortunate side effect of slowing the floating point engine due to increased memory traffic and additional dependencies between instructions limiting instruction level parallelism. While the extra stores make the x86 round properly with respect to the overflow threshold of the narrower formats, extra stores alone do not make the x86 implement the `double` underflow threshold exactly.[12] Essentially, for the underflowed value, the store to memory rounds the value a second time (the first time was during the calculation). This double rounding can give slightly different answers than a single rounding to the final format when the calculation is initially performed. The difference can be $\approx 10^{-324}$ for `double` numbers.

---

[11] This style of rounding is part of IEEE 754 (§4.3). The intention of rounding only the significand is to avoid overflows and underflows in calculations that have extreme intermediate results but a undistinguished final result [59]. Unlike the x86, the rounding width control on the 68000 rounds both significand and exponent [74].
[12] Setting the rounding precision to `double` or `double extended` keeps enough precision so that when values are stored to `float` and loaded back, no double rounding differences occur [45].

$$\underbrace{\underbrace{\underbrace{1.0\cdots000}_{\text{subnormal double}}1011}_{\text{double with extended exponent}}11xxx}_{\text{infinitely precise result}}$$

Figure 1 — Example to illustrate double rounding problems.

On an x86, when a `double` number is stored in a `double extended` register rounded to `double`, exponent values that would indicate subnormal `double` numbers are normal numbers in `double extended`. Since subnormal numbers effectively have fewer significand bits extra bits of precision can be present if the exponent is not also rounded. Figure 1 shows how the differences in value can arise. In this example, the exponent (not shown) is four less than the true `double` underflow threshold. If strict rounding to `double` were performed, the value of the significand to full precision would be

1.0…00010000

(the last four zeros would not be represented in the format) while rounding only the exponent gives a different value,

1.0…0001**1**000

On overflow or divide by zero a single infinity value is generated, so there is not a corresponding rounding problem since infinities are not affected by rounding when converted between formats. To get exactly the same answer, extra work is necessary on each floating point operation. The approach taken in [36] is to store after an operation (to round the exponent) and then test the inexact flag to see if double rounding occurred. On numerical kernels, this technique leads to slowdowns of over an order of magnitude compared to code that does not perform the extra stores and does not test the inexact flag [36]. Another option is to still perform the store after each operation and to trap on underflow, allowing an underflow trap handler to simulate the calculation on the original operands but rounding the result to the smaller width.

To achieve floating point performance comparable to C or FORTRAN, Borneo needs a mechanism to run the x86 using its `double extended` registers without regard to small rounding differences.[13]

Independent of increased speed, programmers would benefit significantly from using the `double extended` type in their codes. Some functions have singularities, either intrinsic or as an artifact of the computation. A region of the domain of the function maps to that singularity and values lying near that region have less accurate answers computed. However, usually if a computation is carried out using greater precision than the input data (and returned result), the extra precision reduces the quantity of data affected by the singularity. When the 11 extra significand bits of `double extended` are used during the computation, `double` input data is usually about 2,000 times farther away from the problematic region, meaning loss of accuracy would typically occur about 2,000 times less often as well. The `double extended` format increases the chances that simple textbook formulas will work over a sufficiently large subset of the range, leading to fewer situations requiring sophisticated numerical analysis [57] (see section 6.10.1 for an example). The `double extended` format also allows better algorithms to be used for certain problems, such as more accurate iterative refinement techniques for systems of linear equations [62] and binary to decimal conversion [21].

6.1.2. Specification

FloatingPointType: one of
```
float double indigenous
```

FloatTypeSuffix: one of
```
f F d D n N
```

Figure 2 — Modifications to Java's grammar to support Borneo's `indigenous` type.

Borneo adds `indigenous` as a primitive floating point type and "`indigenous`" as a keyword. The size and format of `indigenous` are platform-dependent, corresponding to the largest floating point format with direct

---

[13] After this thesis was filed, Sun released a proposal describing possible modifications to Java's floating point semantics to ameliorate Java's performance problems on the x86 architecture [91]. The proposal is discussed in section 9.3.19.

hardware execution on a given processor. The type `indigenous` corresponds exactly to either the IEEE 754 `double` or `double extended` format. In Borneo, values of type `float` and `double` are widened to `indigenous` in the same contexts `float` is widened to `double` in Java (i.e., `float` to `indigenous`, `double` to `indigenous`, and `long` to `indigenous` are widening primitive conversions (JLS §5.1.2)).[14] As with `float` and `double` in Java, if Borneo's "`+`" operator is given a `String` operand and an `indigenous` operand, the `indigenous` operand is converted to a `String` representing its decimal value.

Since Java floating point literals can be given a specific type, Borneo adds a new *FloatTypeSuffix* [nN] to designate an `indigenous` literal. For example, in Java, "`0.0`", "`0.0d`" and "`0.0D`" are `double` literals; "`0.0f`" and "`0.0F`" are `float` literals. The initial value for an `indigenous` variable is `0.0n`, an `indigenous` zero. Figure 2 lists the modifications to Java's grammar needed to support `indigenous`. As in Java, a Borneo literal without a type suffix is of type `double`. In Java, floating point literals that exceed the format's range are caught as compile time errors. Since the runtime range and precision of `indigenous` are platform-dependent, the range errors for `indigenous` cannot all be detected at compile time. Only `indigenous` literals that exceed the range of `double extended` are found during compilation. `indigenous` values that exceed the range of `double`, but not `double extended`, do not cause a compile time error. When used at runtime, such values signal overflow and inexact or underflow and inexact if used on a machine where `indigenous` is the same size as `double`. In general, use of an `indigenous` literal may signal inexact. Each time an `indigenous` literal is accessed, the signals associated with that conversion are raised. For example, if a literal is in a loop, the signals are raised on each iteration. With some analysis, a Borneo implementation may be able to avoid the full overhead of repeatedly converting a literal. For example, if the conversion does not signal, the value can be reused without recomputation. If the inexact flag is not cleared during loop execution, the first loop iteration with the conversion can be peeled and the converted value stored. This avoids repeated conversions while preserving the observable exceptional behavior.

Table 3—Primitive floating point types in Borneo

Name	Size	Type Suffix
`float`	32 bit, single precision	`f`, `F`
`double`	64 bit, double precision	`d`, `D`
`indigenous`	processor dependent IEEE floating point format, at least as large as `double`	`n`, `N`

Since the translation of `indigenous` literals to binary cannot occur fully until runtime, the effective value of `indigenous` literals varies from platform to platform. Borneo guarantees that the floating point value of an `indigenous` literal is the properly rounded result in the final floating point format, avoiding problems of double rounding. Rounding a literal string to `double extended` precision at compile time, and then rounding the resulting `double extended` value to `double` at runtime can give different results than a single rounding to `double` precision at compile time. To avoid such inconsistencies, Borneo's decimal to binary conversion represents the binary form of an `indigenous` literal as the sum of a `double` and a `float`. The combined number of significand bits in a `double` and a `float` exceed the number of bits needed to correctly round a `double extended` value. Therefore, simple floating point addition of the `double` and `float` values can be used to generate the correct `indigenous` result at runtime. However, some additional testing and computation is necessary to decode very large or very small `indigenous` values. The encoding used for `indigenous` literals is further discussed in section 7.1.

By definition, calculations on `indigenous` values can differ from platform to platform. If more uniform answers are desired, the programmer can use `float` and `double` exclusively. If the `double extended` format is truly needed at any price, the `DoubleExtended` class (section 6.4.4) can be used. Expressions including `indigenous` values are not considered constant expressions (JLS §15.27). The calculation of `final indigenous` values cannot be done at compile time unless the `indigenous` expression can be evaluated exactly using `double` precision.

[14] In general, `byte`, `short`, `char`, `int`, `long`, `float`, and `double` can all be widened to `indigenous` via a widening primitive conversion. The `indigenous` type can be narrowed via a narrowing primitive conversion to any of the other 7 built-in numeric types.

Changes throughout the standard library are needed to support `indigenous` properly, such as including transcendental function methods in `java.lang.Math` that take and return `indigenous` values. New overloaded methods `abs`, `min`, and `max` that operate on `indigenous` values are added to `java.lang.Math`. The Java `double` constants `PI` and `E` have as `indigenous` counterparts methods `PI` and `E` which return `indigenous` values. The new methods given in section 8.9 and section 8.10 are necessary to support the `indigenous` to `String` conversion.

6.1.3. Alternatives

To avoid the current performance implications of Java on the x86, one could add `double extended` as a basic type. However, since Java is designed to be portable, nearly all other hardware platforms would be forced to perform a costly simulation of this type. A program designed to use `double extended` would run quite well on an x86, but orders of magnitude slower on a SPARC. Such unpredictability of performance is undesirable even if the same numerical results are generated. Using `indigenous` circumvents creating this potential performance problem by using the resources provided by a particular machine.

Although some IEEE 754 compliant architectures (SPARC, PA-RISC) have support for 128 bit quad floating point computation, such quad length numbers are not a suitable candidate for `indigenous`. While currently such processors have opcodes for manipulating quad values, the actual operations trap to software and are thus rather slower than operations on `float`, `double`, or `double extended` numbers. To implement quad-word floating point operations on SPARC platforms, by default the GCC compiler generates function calls specified in the SPARC ABI instead of using the hardware instructions since the function calls are considerably faster than trapping [85].

Borneo chooses the somewhat lengthy name "`indigenous`" since "`native`" is already used by Java to indicate methods written in other languages. The keyword "`indigenous`" has the same number of letters as C's "`long double`," but "`indigenous`" preserves the Java property that the names of all primitive types are one token. The shorter word "`endemic`" was briefly considered but rejected due to its association with disease.

6.1.4. An `indigenous` example

As shown in Figure 3, one use of `indigenous` variables is storing intermediate calculations in the highest hardware precision available so a more accurate answer can be calculated.

```
float dot(float[] a, float[] b)
{
   indigenous sum = 0.0;

   if(a.length == b.length)
      {
         for(int i = 0; i < a.length; i++)
            {
               // promote array values to indigenous to preserve precision
               sum += (indigenous)a[i] * (indigenous)b[i];
            }
         return (float) sum;
      }
   else
      throw new IndexOutOfBoundsException("Tried to compute the dot product of two arrays of different size.");
}
```

Figure 3 — Using `indigenous` in computing the dot product.

6.2. Floating Point Literals

The IEEE 754 floating point values NaN and infinity have no corresponding literals in Java. The current techniques to refer to these values have unwanted and unnecessary side effects.

Infinities and NaNs can be generated by the evaluation of an expression, such as `1.0f/0.0f` for a `float` infinity. The evaluation of such expressions can alter the floating point flag state of the program. For example, generating an infinity causes the overflow or divide by zero flag to be set. Java's `Float` and `Double` classes include `static final` fields assigned NaN and infinity values. The expressions setting those fields do not have to be evaluated at runtime, so the running program's flag state can remain unchanged; but names in another class have to be referenced. The methods `Float.intBitsToFloat` and `Double.longBitsToDouble` can be used to indirectly create arbitrary floating point numbers since they return a floating point number with the same bit pattern as the integer argument. However, since the size of `indigenous` varies from platform to platform and since there is no integer type guaranteed to be as wide as `indigenous`, an arbitrary `indigenous` value cannot be portably or directly created from a single integer constant.

Augmented Java Syntax
FloatingPointLiteral:
`infinity` *FloatTypeSuffix*$_{opt}$
`NaN` *FloatTypeSuffix*

Figure 4 — Changes to Java grammar to support `infinity`, and `NaN` literals.

To allow special floating point values to be used easily for all primitive floating point types, Borneo has floating point literals denoting infinity and NaN. For clarity, Borneo chooses "`infinity`" to represent infinite values. The string "`inf`" was not chosen since in mathematics *inf* refers to the greatest lower bound or smallest element of a set. While "`NaN`" could be chosen to represent a NaN value, `Float.NaN` and `Double.NaN` are existing `static` fields in Java standard library. To avoid name conflicts, Borneo NaN literals are represented as "`NaN`" followed by a *FloatTypeSuffix*; so "`NaNf`", "`NaNd`", and "`NaNn`" are, respectively, `float`, `double`, and `indigenous` NaN literals. Infinity literals can also have a *FloatTypeSuffix* with the usual interpretation. A NaN literal prefixed by "`+`" or "`-`" is a valid expression. Floating point literals are available in all contexts without qualification; therefore, "`infinity` " can be used instead of "`Double.POSITIVE_INFINITY`." Borneo adds thirteen new character sequences denoting floating point literals (enumerated in section 9.5), but conflicts with names in existing Java programs should be rare.

6.3. `Float`, `Double`, and `Indigenous` classes

Much of the functionality and utility of a language is captured in the language's standard library. Therefore, along with changes to Java proper, Borneo includes library modifications to aid in bringing Java to full IEEE 754 compliance. The Java `Float` class "wraps" the primitive type `float` in an object and defines many useful `static` methods operating on `float` values. The `Double` class has the analogous relation to the `double` type. Figure 5 and section 8.5 detail the specific changes Borneo makes to the `Float` class. The standard recommends a number of functions acting on floating point values. Java already includes two of the recommended functions as methods in the `Float` class (`isNaN` and `isInfinite`[15]), but fails to include six other useful methods that Borneo adds (`copySign`, `scalb`, `logb`, `nextAfter`, `unordered`, and `fpClass`). However, Borneo adds the remaining IEEE recommended functions to the `Math` class to make better use of Java's method overloading. Full specification of the IEEE 754 recommended functions taking advantage of Borneo features not found in Java is given in section 8.8. These methods make direct manipulation of the bit patterns of floating point values largely unnecessary. Borneo also includes a `logbn` method, which differs from `logb` in the treatment of subnormals. The `scalb` method is comparable to ANSI C's `ldexp` and `logb` is similar to ANSI C's `frexp`.

Certain constants associated with the `float` type, such as the maximum finite value, are also included in Java's `Float`. By using `nextAfter`, `logb`, and a few simple floating point literals such as `1.0f`, `0.0f`, and `infinityF`, all the current floating point constants in `Float` can be defined in a format independent manner. For example, for any IEEE floating point type, `MAX_VALUE` is given by `nextAfter(infinity,0.0)` for `0.0` and `infinity` of the appropriate type. Such a specification is clearer than the current specification of particular bit patterns (JLS §20.9.1–§20.9.5, §20.10.1–§20.10.5). New constants describing the exponent range, rounding threshold, and minimum normal value, are also added to `Float` since those quantities also provide useful information about the type. The rounding threshold and minimum normal value are given in the `<limits.h>` file in ANSI C. Corresponding changes are made to the `Double` class (section 8.6).

An `Indigenous` class modeled after `Float` and `Double` provides equivalent support for `indigenous`. Although the values of the `static final` fields in `Indigenous` are platform-dependent, the

---

[15] The standard actually calls for an `isFinite` predicate instead of Java's `isInfinite`. `isInfinite` is not the logical negation of `isFinite` since `isFinite(NaN)` and `isInfinite(NaN)` are both `false`.

specification is platform-independent by using methodology described above. An `abstract` method `indigenousValue` is added to the `Number` class so that all subclasses of `Number` can be converted to one another. Since the size of `indigenous` is platform dependent, `Indigenous` does not have methods corresponding to `Float.floatToIntBits` and `Float.intBitsToFloat`. The IEEE recommended functions can be used to take apart and create arbitrary floating point values. `Indigenous` does include two methods without counterparts in `Float` or `Double`. The method `static double[] decompose(indigenous value)` takes an `indigenous` value and returns a two-element array of `double` floating point numbers encoding the argument in the same manner `indigenous` literals are encoded in the constant pool. The second element of the returned array holds a `float` number promoted to `double`. The inverse functionality is provided by `static indigenous compose(double, float)`. A new constructor taking an `indigenous` value is also included in the `Float`, `Double`, and `Indigenous` classes.

//New constants

public static final float **MIN_NORMAL** = 1.17549435e-38f; *// smallest normal number, defined as*
// nextAfter(0.0, infinity)/(nextAfter(1.0,infinity) –1)
// equal to 2^{2-2^K} *where K = number of exponent bits –1*

//Information about format

public static final float **ROUNDING_THRESHOLD** = 5.960465e-8f; *// the least positive number such that under round to nearest*
// 1 + threshold ≠ 1, defined as
// nextAfter((nextAfter(1.0, infinity) – 1.0)/ 2.0, infinity)

public static final int **SIGNIFICAND_WIDTH** = 24; *// width of significand including possible implicit bit*
// defined as –logb(nextAfter(1.0, infinity) –1.0)+ 1

public static final int **MIN_EXPONENT =** –126; *// smallest (most negative) exponent of a normal number,*
// defined as (int)logb(MIN_NORMAL)

public static final int **MAX_EXPONENT** = 127; *// largest exponent of a normal number*
// defined as (int)logb(nextAfter(infinity,0.0))

public static final int **BIAS_ADJUST** = 192; *// amount by which exponent is adjusted in trapping on*
// overflow or underflow, defined as
// $3 \cdot 2^{n-2}$ *where n is the number of bits in the exponent*
// BIAS_ADJUST can be calculated by
// (int)(3.0 *
// scalb(2,(int)(ceil(log(logb(MAX_VALUE))/log(E)) –2)))

//New methods

public indigenous **indigenousValue**(); *// for symmetry with other primitive types*

//New Constructor

public **Float**(indigenous value);

//Constants to modify

public static final float **MIN_VALUE**= 1.4e–45f *// defined as* nextAfter(0.0, infinityF)
public static final float **MAX_VALUE**= 3.4028235e+38f *// defined as* nextAfter(infinityF, 0.0)
public static final float **POSITIVE_INFINITY**= infinityF;
public static final float **NEGATIVE_INFINITY**= –infinityF;
public static final float **NaN**= NaNf;

Figure 5 — Changes to the `Float` class.

6.4. New Numeric Types

While the primitive floating point types are adequate for many purposes, other kinds of numeric types are useful in certain circumstances. The classes described below use various IEEE features to implement numeric types appropriate to different domains. The exact specification for the new numeric types is not provided; their general behavior and purpose is given. All the numeric classes use Borneo's operator overloading capabilities (see section 6.9).

6.4.1. PseudoInt

The significand field of a floating point number can be used to provide enhanced, fast integer arithmetic. Such a numeric type is useful for financial and accounting calculations. Unlike Java's modulo 2's complement integers, `PseudoInt` indicates integer overflow by setting the inexact flag. To achieve a wider range, `indigenous` can be used as the base floating point type. On the x86, 64 bit signed-magnitude integers can be stored in the `double extended` floating point format. `PseudoInt` is similar to the `comp` type in SANE.

6.4.2. Wide Exponent Types

Certain calculations, such as computing the determinant of a matrix of high dimension, can overflow or underflow quite readily. Often the ratio of two such determinants is computed; the magnitude of the two determinants is not of direct interest. To handle such situations, where greater range but not greater precision is needed, wide exponent types are used. `WideExpFloat`, `WideExpDouble`, and `WideExpIndigenous` extend the base floating point type's exponent with a 32 bit integer. The range of these types is so large that user-visible overflow or underflow should occur extremely rarely when only basic arithmetic operations are performed. Wide exponent types implement the functionality of the KOUNT mode discussed in [56]. The basic approach to coding wide exponent types using exceptions is covered in [40].

6.4.3. DoubledDouble

`DoubledDouble` is a non-IEEE 128 bit format. Instead of using 128 bits to store a single number, `DoubledDouble` uses the bits to represent the unevaluated sum of two 64 bit `double` numbers (the sets of powers of two represented by the two `double` components do not have to be contiguous). `DoubledDouble` is primarily useful on hardware with a fused mac instruction since that operation allows `DoubledDouble` to run at a reasonable speed. See [84] for a detailed discussion of "multi-term" extra-precision floating point representations.

6.4.4. DoubleExtended

Depending on the platform, the `indigenous` type can either be the 64 bit `double` format or the 80 bit `double extended` format. When exactly 80 bits must be used, the `DoubleExtended` class explicitly implements 80 bit `double extended` floating point numbers. On the x86 and 68000 architectures, `DoubleExtended` should refer to `indigenous`; on other processors, `DoubleExtended` must be implemented in software. Techniques similar to those used in SoftFloat [42] can be used to implement IEEE floating point numbers with integer arithmetic in Java.

6.4.5. Extended

Analogous to the `BigInteger` and `BigDecimal` classes added in Java 1.1, the `Extended` class provides IEEE 754 style numbers of arbitrary length. A class variable controls the width to which current operations are rounded. The C++ library SciLib [76] provides approximately the intended functionality of `Extended`. At run time, Borneo's operator overloading can be used to build a data structure and dynamically determine the proper precision to use. For better speeds, operations on `float` and `double` length numbers could be recognized as special cases and performed with the primitive floating point types. Writing the transcendental methods for `Extended` is a large task since the best algorithm to use depends on the length of the number.

A 128 bit "quad" size floating point format is a special case of `Extended`. Quad size floating point numbers have some instruction-level support on current SPARC and PA-RISC architecture, but the actual operations are performed in software.

6.4.6. ExtraPrecision

Table 4 — Various reasonable implementation options for the `ExtraPrecision` class.

Architecture	Format	Processor Features
x86	80 bit `double extended`	native floating point format
PowerPC, RS/6000	doubled double	extensive use of fused mac instruction
SPARC	quad	existing instruction level or library support

`ExtraPrecision` is a processor dependent floating point type, used when a modest amount of extra precision is necessary at a modest price. The fastest kind of floating point numbers larger than `double` on a given platform are used. `ExtraPrecision` is not necessarily an IEEE style number format. Some reasonable implementation options for different processors are listed in Table 4. `ExtraPrecision` could be implemented by referring to other floating types already defined in the Borneo standard library.

6.4.7. Interval

Instead of representing numbers as single points, interval arithmetic [2], [73] aims to bound the error in the result of a calculation by representing each number as a range. Directed rounding is used in interval arithmetic to obtain the upper and lower bounds. Interval arithmetic is useful for determining if a calculation is sensitive to roundoff and should be rerun with additional precision to achieve the desired accuracy. Since decimal to binary conversion is often inexact, using a single value for a decimal floating point literal in interval arithmetic is not ideal. Instead, as with other interval operations, decimal to binary should return a non-trivial interval. Therefore, the `Interval` class includes methods such as `Interval` **`valueOf`**`(String s)` to perform the necessary conversion.

6.4.8. ExtendedInterval

`ExtendedInterval` combines the arbitrary precision of `Extended` with the upper and lower bounds of `Interval`. One option for implementing `ExtendedInterval` is to use two arbitrary precision numbers, one for the upper bound and the other for the lower bound. To save storage, one extended number (a midpoint) and a range can be stored. Such a variety of implementation options argue for users to be able to write their own interval types to supplement the Borneo library interval types.

6.4.9. Complex and Imaginary Numbers

The textbook example of complex numbers as an abstract data type promotes real values to complex whenever the two are combined. Unfortunately, such promotion creates a zero imaginary component which has adverse consequences for complex arithmetic in some applications; spurious overflows and underflows can occur and some complex number identities are violated. Using a separate imaginary type and the formulas discussed in [63] gives fewer computational irregularities. Example code for a portion of the `Complex` class is listed in section 6.9.6.1.

6.5. Floating Point System Properties

Processor designers have created three distinct families of IEEE 754 compliant machines. The first class of machines takes a straightforward and orthogonal approach to implementing the standard; they provide `float` and `double` formats and instructions to combine and manipulate numbers of those formats. The SPARC architecture is in this “orthogonal” category.

The RS/6000 and PowerPC take an alternative approach to realizing the standard and implement nearly all floating point operations as special cases of a single ternary operation, fused multiply accumulate (fused mac). A fused mac multiplies two numbers exactly and then adds a third number to the product, generating a single rounding error at the end (fused mac is not a part of the IEEE 754 standard). Addition is fused mac with one of the factors set to `1.0`. Multiplication is a fused mac with the summand set to `0.0`. Correctly rounded division is implemented by a series of fused mac operations solving a recurrence [56]. A fused mac machine can easily simulate an orthogonal family machine, but not vice versa.

The third architecture family is primarily comprised of the 68000 series and the x86 line; these machines most naturally use `double extended` values. In principle, such a machine should be able to simulate the results of a orthogonal machine easily, but some design choices of the x86 make that task very costly and subtle.

A hardware fused mac can give increased speed and improved accuracy. In keeping with the goals of Borneo, some access to fused mac is provided. However, because simulating a fused mac on a processor without that instruction is quite slow, the simulation should be avoided unless a fused mac is absolutely necessary for correctness or reproducibility.

To inquire about other properties that vary from platform for platform, Java provides a number of system properties that can be accessed through the `System.getProperties` method (JLS §20.18.7). To provide information about floating point differences, as shown in Table 5, Borneo augments the list of system properties to include whether or not a hardware fused mac is present. Since floating point hardware is optional on Java chips [78], the new `fp.hardware` property specifies whether or not underlying hardware is used to run floating point. Software implementation of IEEE 754 floating point is approximately 20 to 50 times slower than a hardware implementation on the same processor. The new system properties cannot be set by the user. The size of the hardware dependent `indigenous` type can be determined from information given in the `Indigenous` class.

Table 5 — New system properties describing floating point.

Key	Description of associated value
`fp.fmac`	"`true`" if a hardware fmac is available, "`false`" otherwise
`fp.hardware`	"`true`" if floating point support is in hardware, "`false`" otherwise

6.6. Fused mac

Although use of fused mac can lead to better answers for some codes (such as computing the dot product of two vectors), other algorithms depend on the precise properties of floating point multiplication and addition and do not work when a fused mac is substituted [56]. However, in many cases, algorithms do continue to work, and execute faster, when a fused mac is used. Other algorithms, such as many `DoubledDouble` arithmetic implementations, require a fused mac. To fully support fused mac, the programmer needs to be able to specify three cases: fused mac must be used, fused mac must *not* be used, and it is irrelevant whether or not fused mac is used. Borneo supports two of the three options; fused mac must be used and fused mac must not be used.

Fused mac must not be used is specified by writing programs with the traditional floating point operators; fused mac must be used is specified by explicitly using a fused mac method. To portably support a fused mac on machines that do and do not have fused mac hardware, Borneo provides explicit fused mac methods for all primitive floating point types. On platforms where `fp.fmac` (section 6.5) is "`true`", the `fmac` methods should use appropriate native instructions. Otherwise, integer calculations can be used to implement fused mac. Supporting fused mac through a library call is suggested by Gosling in [37].

For compatibility with Java, Borneo requires that all uses of fused mac be explicitly indicated; the compiler or interpreter is not free to combine consecutive multiplication and addition operators into a single fused mac.

Borneo does not support the third option of using fused mac if convenient because doing so would require extensive changes to the language. The semantics of "`a * b`" would be changed to depend on the surrounding expression. For predictable behavior, the precise conditions under which a multiply followed by an add in the source language can be collapsed into a fused mac need to be given. For increased speed, using fused mac if possible could be made the default; a declaration could be used to inhibit fused mac where inappropriate. Explicit storing to temporaries would also disable fused mac but possibly slow down the code. Alternatively, Borneo's `anonymous` declarations (section 6.10) could be overloaded to indicate using fused mac was permitted.

At the JVM level, support fused mac if convenient would also require changes. For example, two new sets of instructions, one for explicit fused mac and another for possibly fused mac could be added. Both options are necessary for portable code to be compiled on one machine and run on another while preserving source program semantics. For better compatibility with existing JVMs, the multiply and add operations allowed to be fused could be indicated in a separate table stored as an extra attribute in the `class` file ([66] §4.7.1). Using consecutive multiply and add instructions is invariably slower than a single fused mac instruction on hardware supporting fused mac. Therefore Borneo's current semantics can cause some degradation of performance on fused mac capable machines since fused mac cannot be used by default. However, making all uses of fused mac explicit limits semantic differences when compiling existing Java source with a Borneo compiler.

6.7. Rounding Modes

Round round get around
I get around
—Brian Wilson, "I Get Around"

Calculating with floating point numbers almost invariably requires approximation. Multiplying two floating point numbers can generate an infinitely precise result with twice as many significand bits. To exactly represent the result of floating point addition, hundreds and hundreds of bits might be necessary if the two summands are very different in magnitude. Therefore, a large loss of information can occur when a floating point operation on two numbers delivers a result in the same format. However, arbitrarily precise arithmetic is impractical (and unnecessary) in many circumstances, so the roundoff properties of a floating point standard are very important.

The IEEE 754 default rounding mode, round to nearest even, is appropriate for most calculations. The other rounding modes are useful for various sorts of error analysis and for interval arithmetic. Dynamically changing rounding modes can also be used to find numerically unstable calculations. The next section discusses more detailed requirements for using different rounding modes.

6.7.1. Requirements of algorithms using rounding modes

Algorithms that exploit rounding modes have four requirements reflecting different usages.

1. *dynamic rounding modes*: Some codes have the property that they can be run meaningfully under different rounding modes. For example, as listed in section 6.7.5.1, a method that calculates a tight upper bound of a polynomial at a given argument when run under round toward positive infinity finds a tight lower bound of the polynomial when run under round toward negative infinity. For such programs, the rounding mode should be an implicit or explicit parameter that can be given at runtime.
2. *static rounding modes*: On the other hand, interval arithmetic should not take a rounding mode from the environment. Instead, the rounding modes for interval arithmetic should be statically specified at compile time.
3. *scoping*: A common idiom when using rounding modes is to implement scoping as shown in below; modifying the rounding mode of a method's caller is rarely desired.

```
savedRM = getRound();
setRound(newRM);
Calculation
setRound(savedRM);
```

4. *numerical sensitivity detection*: Rounding modes can also provide a powerful testing mechanism for determining if an algorithm is poorly behaved on certain inputs. An input that causes an otherwise working code to generate nonsense answers can be run under different rounding modes with the same troublesome input. If the answer varies greatly, sensitivity to rounding is the likely culprit.

The two formulas in Table 6 calculate the area of a triangle given the lengths of its sides. However, the classical Heron's formula in the second column is quite sensitive to rounding differences for needle-like triangles [57]. For the data in Table 6, when $(a+b) + c$ is calculated, c gets rounded away. The more sophisticated formula in the third column is much more stable. Of the two equations, Heron's formula is commonly given in textbooks. All calculations in Table 6 are done to `float` precision. The same effect occurs with reduced frequency under wider precisions. As shown by the data, the answers given by Heron's formula when run under different rounding modes vary greatly while the more sophisticated formula is nearly unaffected (while also being correct). Instead of changing the rounding mode to find such sensitivities, changing that data slightly can also be attempted. However, as shown by the second set of input values, even a correct formula can be very sensitive to changes in the input data. Also, knowing how to perturb the data without violating a program constraint is not always obvious: if c were perturbed too much, the three lengths would no longer form a triangle. To find sensitivities, no one rounding mode is superior; each one should be tried in turn.

Table 6 — Sensitivity to rounding of two different formulas to calculate the area of a triangle from the lengths of its sides (calculations done in single precision).

Rounding mode	**Heron's Formula** $s=((a+b)+c)/2$ $\sqrt{s\cdot(s-a)\cdot(s-b)\cdot(s-c)}$ (unstable)	$\frac{\sqrt{(a+(b+c))\cdot(c-(a-b))\cdot(c+(a-b))\cdot(a+(b-c))}}{4}$ (stable)
a=12345679, b=12345678, c=1.01233995 > $a-b$		
to nearest	0.00	972730.06
to +∞	17459428.00	972730.25
to -∞	0.00	972729.88
to 0	-0.00	972729.88
a=12345679, b=1234567**9**, c=1.01233995 > $a-b$		
to nearest	12345680.00	6249012.00
to +∞	12345680.00	6249013.00
to -∞	0.00	6249011.00
to 0	0.00	6249011.00

To support finding such sensitivities, the standard mandates that rounding modes can be dynamically changed at runtime to any of the four possibilities. Microprocessors conforming to the standard provide FPU control bits to represent and change the rounding mode.[16] Some Unix systems provide library routines to sense and change the rounding mode dynamically. Unfortunately, the library interface is not uniform across operating systems. Even if the interface were standardized, library calls alone are inadequate to properly support rounding mode control.

Original Code

```
{
    int r;
    float f;

    f = 1.0f + Float.ROUNDING_THRESHOLD;

    if(f  == 1.0f) // rounding mode is to zero or to –infinity
    {
        f = –1.0f – Float.ROUNDING_THRESHOLD;
        if(f == –1.0f)
            r = Math.TO_ZERO;
        else
            r = Math.TO_NEGATIVE_INFINITY;
    }
    else // rounding mode is to nearest or to +infinity
    {
        f = –1.0f – Float.ROUNDING_THRESHOLD;
        if(f  == –1.0f)
            r = Math.TO_POSITIVE_INFINITY;
        else
            r = Math.TO_NEAREST;
    }
}
```

After constant folding under round to nearest

```
{
    int r;
    float f;

    f = 1.0000001f;

    if(f  == 1.0f)
    {
        f = –1.0000001f;
        if(f == –1.0f)
            r = Math.TO_ZERO;
        else
            r = Math.TO_NEGATIVE_INFINITY;
    }
    else
    {
        f = –1.0000001f;
        if(f  == –1.0f)
            r = Math.TO_POSITIVE_INFINITY;
        else
            r = Math.TO_NEAREST;
    }
}
```

After constant propagation, dead code elimination, etc.

```
{
    int r;
    r =
        Math.TO_NEAREST;
}
```

Figure 6 — Code fragment which determines the rounding mode using the results of arithmetic operations and ruinous optimization sequence.

The code in the first column of Figure 6 yields different results under different rounding modes; in particular, it uses the results of well-chosen arithmetic operations to determine the current rounding mode setting. This small segment of code has several tempting opportunities for optimization. If rounding modes can be ignored, the expression `1.0f + Float.ROUNDING_THRESHOLD` has constant operands and can therefore be constant folded at compile time. Once that is done, the value of `f` is a constant which can be propagated to the first comparison operation. The comparison also involves constant operands and can also be evaluated at compile time, allowing the true branch to be recognized as unreachable code and not even compiled. A compiler at even modest levels of optimization may perform such an optimization sequence. Current Java semantics do not preclude such efforts. Thus, if code similar to the code in Figure 6 were used to try to determine the current rounding mode at runtime, the optimizer may defeat the purpose of the code. If the rounding mode is known to be constant at compile time, the compiler could perform the optimization sequence while preserving the desired semantics.[17] Therefore, to generate correct optimized code, the compiler must know when the rounding mode may be varied at runtime and what values the rounding mode can assume. The presence of library calls sensing or changing the rounding mode is not necessary for rounding modes to influence a computation.

6.7.2. `rounding` Declarations

In Borneo a new language declaration, `rounding` *`Expression`*, informs the compiler when rounding modes other than round to nearest might be used. As shown in Figure 8, the integers from 0 to 3 are used to denote rounding modes; mnemonic constants for the rounding modes are defined in the `Math` class (Java does not have enumerated types). A `rounding` declaration is effective from the declaration point in a block to the close of that block or to the next `rounding` declaration. If the expression given to a `rounding` declaration evaluates to an

[16] The Alpha [83] can statically encode three of the four rounding modes into two bits of arithmetic opcodes; the fourth bit pattern is used to take the rounding mode from the FPCR (Floating point Control Register).

[17] Borneo semantics preserve the flag effects of expression evaluation. If evaluating an expression causes no flags to be raised, the expression can be evaluated at compile time regardless of rounding mode (with one exception). When an expression is exact (the inexact flag is not raised) the same answer is delivered under all rounding modes, except in the case of $x - x = \pm 0.0$ where the sign of zero depends on the dynamic rounding mode.

integer outside of [0, 3] an unchecked `UnknownRoundingModeException` is thrown. It is not a compile time error to give a `rounding` declaration a constant integer expression (JLS §15.27) outside of [0, 3]. The compiler treats such a declaration as equivalent to

```
throw new UnknownRoundingModeException()
```

which can affect the reachability (JLS §14.19) of other statements.

Augmented Java syntax
LocalVariableDeclarationStatement:
    FloatingPointRoundingDeclaration `;`

New Borneo Productions
FloatingPointRoundingDeclaration:
    `rounding` *Expression*

Figure 7 — Changes and additions to the Java grammar to support rounding mode declarations.

```
// enumerated rounding modes
public static final int  TO_NEAREST              = 0;
public static final int  TO_ZERO                 = 1;
public static final int  TO_POSITIVE_INFINITY    = 2;
public static final int  TO_NEGATIVE_INFINITY    = 3;

// getRound returns the current rounding mode using the above encoding
public static int getRound();

// setRound sets the current rounding mode to the rounding mode represented by its argument
public static void setRound(int rm) throws UnknownRoundingModeException;

// New exception class
public class UnknownRoundingModeException extends RuntimeException {}
```

Figure 8 —Changes to the `Math` class and a new exception class to support IEEE 754 rounding modes.

Rounding modes are lexically scoped; a method does not inherit the rounding mode of its caller, although blocks do inherit the rounding mode of the textually enclosing block. A `rounding` declaration can appear in the optional initialization code of a `for` loop. Such a `rounding` declaration affects the remainder of the loop initialization, the loop test, and the loop update portions of the `for` loop as well as the loop body. The default rounding mode is round to nearest. Since uses of non-default rounding modes in Borneo must be explicit, existing Java programs cannot break by being run under an unintended rounding mode.[18] The code regions influenced by `rounding` declarations are lexically scoped to facilitate optimizations such as constant folding, constant propagation, and common subexpression elimination. Constant folding can be performed under any rounding mode as long as the rounding mode does not vary at runtime. A Java compiler's existing machinery to find and evaluate constant integer expressions can be used to determine when the expression given to a `rounding` declaration is a compile-time constant. The `rounding` declarations give the compiler enough information to determine when such transformations are valid, otherwise, erroneous programs can result [35].

While `rounding` declarations aid compiler optimizations, more importantly, explicit declarations make clear to a reader of the program what portions of the code can be run under different rounding modes. Long lasting code is often read more often than modified, making easy understanding important.

The initialization of `static` class variables by constant expressions is not influenced by different rounding modes; such expressions on `float` and `double` values can be evaluated at compile time. To use rounding modes other than round to nearest to initialize fields, a method call can be used or a `rounding` declaration can be placed inside a `static` initializer block (JLS §8.5).

---

[18] As discussed in section 6.7.3, Borneo actually does allow existing code to be run under a different rounding mode. However, this feature is not strictly part of the language; the same effect could be achieved by using a native assembly language program to alter the rounding mode.

Borneo Code	**Equivalent Java code with `native` methods**
{ *Calculation$_1$* rounding *Expression$_1$*; *Calculation$_2$* rounding *Expression$_2$;* *Calculation$_3$* }	{ int savedRM = getRound(); try { *Calculation$_1$* *//rounding* Expression$_1$*;* setRound(*Expression$_1$*); *Calculation$_2$* *//rounding* Expression$_2$*;* setRound(*Expression$_2$*); *Calculation$_3$* } finally { setRound(savedRM); } }

Figure 9 — Desugaring Borneo `rounding` declarations into Java with `native` methods.

No matter how a method or block affected by a `rounding` declaration is exited, the previous rounding mode must be restored. Restoring the rounding mode can be realized with the same code generation techniques as used for `try-finally`. Assuming optimizations are inhibited, `rounding` declarations can be desugared into Java using `native` methods to manipulate the rounding mode, as shown in Figure 9.[19] When generating native code, the `setRound` and `getRound` calls can be implemented with a few assembly language instructions. The desugaring introduces a new scope, via a set of braces, for the variable holding the previous rounding mode. Assuming there are no method calls in the original block, in the desugared code `getRound` only has to be called once per original source block with a `rounding` declaration. The desugared code has one call to `setRound` for each `rounding` declaration in a block (plus an addition call to `setRound` after exiting the block to restore the original rounding mode).

As discussed further in the next section, a block with a `rounding` declaration must set the rounding mode to round to nearest before any method call and restore the rounding mode after the call returns; a valid desugaring is given in Figure 10. This requirement prevents explicit calls to `setRound`, including calls to `setRound` from a called method, from circumventing the rounding mode set by a `rounding` declaration. Taking asynchronous exceptions into consideration imposes constraints on the saving and restoring of rounding modes around a method call. Asynchronous exceptions can occur at any point during a thread's execution. Java only has two asynchronous exceptions, `ThreadDeath` caused by calling the `stop` family of methods for `Thread` or `ThreadGroup` and `InternalError` in the JVM. When asynchronous exceptions are generated, `finally` clauses still get executed. Even though a thread handling an asynchronous exception may soon terminate, the dynamic rounding mode should be properly set so that floating point code in pending `finally` clauses executes properly. To ensure this, calls to `setRound` generated from `rounding` declarations or from restoring the rounding mode for method calls should be in `try-finally` blocks that restore the previous rounding mode in the `finally` clause. To properly restore the rounding mode, the previous rounding mode must be available in the `finally` clause. If asynchronous exceptions could be ignored, the desugaring in Figure 10 could avoid introducing a new variable by using the existing `savedRM` variable to hold two rounding modes, the rounding mode before the scope was entered and the rounding mode before the method call. However, if that were done, at least one of the calls to `setRound` in the compiler generated `finally` clauses could incorrectly set the rounding mode if an asynchronous exception occurred.

---

[19] These desugarings assume arithmetic operations can be influenced by setting the dynamic rounding mode. This is not the default code generation technique used on Alpha platforms (where some rounding modes can be statically indicated in an instruction field).

Borneo Code

```
{
    // Calculation1 and Calculation2 do not have
    // any method calls
    rounding Expression;
    Calculation1

    // a method call
    foo(ParamExpr1, ParamExpr2);.

    Calculation2
}
```

Equivalent Java code with `native` methods

```
{
    // Calculation1 and Calculation2 do not have
    // any method calls
    int savedRM = getRound();

    try
        {
            // rounding Expression;
            setRound(Expression);
            Calculation1

            // evaluate arguments to the method under the current rounding mode
            ParamExprType1 t1 = ParamExpr1;
            ParamExprType2 t2 = ParamExpr2;

            // record current rounding mode
            int currentRM = getRound();

            try
                {
                    setRound(Math.TO_NEAREST);
                    // a method call
                    foo(t1, t2);
                }
            finally
                {
                    // restore rounding mode to value before m ethod call
                    setRound(currentRM);
                }
            Calculation2
        }
    finally
        {
            setRound(savedRM);
        }
}
```

Figure 10 — Desugaring of a method call in a block with a `rounding` declaration.

As shown in Figure 11, the desugaring for a `rounding` declaration inside a loop is the same as a `rounding` declaration outside a loop. When a `rounding` declaration appears at the start of a loop body, if the compiler can prove the loop tests (and loop updates in the case of a `for` loop) are not affected by the rounding mode, the more efficient translation in Figure 12 is also valid.

Borneo Code

```
int i=0;
while( i < MAX)
{
    Calculation1
    rounding i % 4;
    Calculation2
}
```

Equivalent Java code with `native` methods

```
int i = 0;
while( i < MAX)
{
    {
        int savedRM = getRound();
        try
            {
                Calculation1
                //rounding i % 4;
                setRound(i %4);
                Calculation2
            }
        finally
            {
                setRound(savedRM);
            }
    }
}
```

Figure 11 — Translating a Borneo loop with a `rounding` declaration into Java with `native` methods.

Borneo Code

```
int i =0;
while(int i < MAX)
{
    rounding i % 4;
    Calculation
}
```

Equivalent Java code with `native` methods

```
int i = 0;
while(i < MAX)
{
    {
        int savedRM = getRound();
        try
            {
                // rounding i % 4;
                setRound(i %4);
                Calculation
            }
        finally
            {
                setRound(savedRM);
            }
    }
}
```

More efficient translation

```
{
    int i = 0;
    // rounding mode is not restored on each loop
    // iteration
    int savedRM = getRound();
    try
        {
            while(i < MAX)
                {
                    // rounding i % 4;
                    setRound(i %4);
                    Calculation
                }
        }
    finally
        {
            setRound(savedRM);
        }
}
```

Figure 12 — Different translations of loops with `rounding` declarations.

To achieve true dynamic behavior, a parameter or other variable can be used as the *`Expression`* for a `rounding` declaration. A class variable could also be used to determine the rounding mode for operations on objects of that class.

6.7.3. Finding Sensitivities and Code Generation Requirements

The `rounding` declarations directly meet three of the four requirements discussed earlier. To achieve dynamic behavior, a `rounding` declaration can be given a non-constant expression as an argument. A fixed rounding mode can be set by giving `rounding` a constant integer expression, such as one of the `static final` fields representing rounding modes in the `Math` class. The semantics of `rounding` implement scoped rounding modes without cluttering the code with numerous explicit `getRound` and `setRound` calls. However, `rounding` declarations prevent code not designed for use under non-standard rounding modes to be run under different rounding modes to test for sensitivities. Since finding sensitivities by changing rounding modes is useful, Borneo mandates certain code generation requirements to allow the benefits of structured rounding mode control while still permitting dynamic rounding mode control over existing code.

In the absence of `rounding` declarations, a method should be run under round to nearest, the Java and IEEE 754 default. Semantically, the compiled version of a method lacking `rounding` declarations could set the rounding mode to round toward nearest at the start of the method to ensure round to nearest was in effect. However, such setting of the rounding mode would be unnecessary since the default rounding mode is already round to nearest. A Borneo program that does not explicitly use a `rounding` declaration *cannot* defensively set the rounding mode to round to nearest.[20] Instead, blocks with `rounding` declarations are responsible for saving the rounding mode of the calling method *and* setting the rounding mode to round to nearest for the callee. Methods that alter the rounding mode should be in the small minority so some extra overhead in their usage is preferable to slowing down all methods that do not alter the rounding mode.

Since Borneo methods that do not set the rounding mode cannot take defensive action, if an explicit call to `setRound` were made in such a method, all subsequent floating point operations would be run under the new rounding mode. For example, the method `rounder` in Figure 13 sets the rounding mode and then calls `tolerant`, a method without a `rounding` declaration (see Figure 14 for an activation tree of this example). In the first call to `tolerant` and the subsequent call to `adder`, the rounding mode set in `rounder` is in effect. However, when `adder` is called from `defensive`, `adder` always runs under round to nearest. The rounding mode set in `defensive` is irrelevant; a block with a `rounding` declaration must set the rounding mode to round to nearest for its callees. This example also demonstrates a possible hazard to using explicit `setRound` calls; the

[20] This requirement implies Borneo code compiled on the Alpha will by default have to take the rounding mode from the status register.

rounding mode of `rounder`'s caller is changed to round to zero, possibly causing unintentional repercussions such as invalidating optimizations leading to incorrect code being executed.

```
static void rounder(void)
{
    Math.setRound(TO_ZERO);                     // explicitly set rounding mode
    tolerant(1.0, ROUNDING_THRESHOLD);          // the sum of the two arguments rounded to zero due to call to setRound
    defensive(1.0, ROUNDING_THRESHOLD);         // the sum of the two arguments unaffected by the call to setRound
    // original rounding mode not restored!  Almost certainly an error!
}

static double tolerant(double a, double b)
{
    return adder(a,b);    // sum not protected against explicit rounding mode changes made in caller
}

static double defensive(a, b)
{
    rounding Random.nextInt() % 4;
    return adder(a, b);  // before calling adder, the rounding mode is restored to round to nearest,
                         // the rounding mode in affect in defensive is irrelevant
}

static double adder(a, b)
{
    return a + b;
}
```

Figure 13 — Code demonstrating interplay between explicit `setRound` calls and `rounding` declarations.

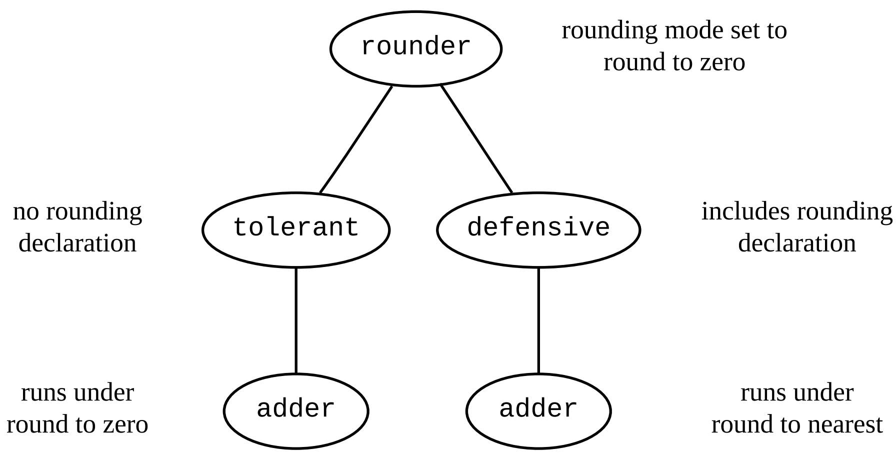


Figure 14 — Activation tree for the code in Figure 13.

Following these guidelines, even previously compiled code can be tested for roundoff sensitivity without ruining explicit rounding mode settings. Borneo's rounding mode related code generation requirements allow previously compiled programs to be called under different rounding modes. Currently, not all JVM `class` files are generated from Java programs; other languages can be supported as well.[21] Borneo maintains Java's ability to interoperate with code from other languages.

This technique for supporting dynamic rounding modes on arbitrary code is extra-lingual. The structured `rounding` declarations should be used for normal programming purposes. The compiler is not obligated to defend against rounding mode changes outside of `rounding` declarations. For example, a Borneo compiler can perform common subexpression elimination of `b + c` on the code in Figure 15 even though the call to `foo` can potentially change the rounding mode and thus alter the evaluation of `b + c`.

[21] For example, the Ada 95 compilers AppletMagic by Intermetrics and ObjectAda by Aonix can produce `class` files.

Borneo Code
```
static final double b = 1.0;
static final double c = Double.ROUNDING_THRESHOLD;
{
    double a, d;
    a = b + c;
    foo();
    d = b + c;
}
```

Borneo Code after legal common subexpression elimination
```
static final double b = 1.0;
static final double c = Double.ROUNDING_THRESHOLD;
{
    double a, d;
    a = b + c;
    foo();
    d = a;
}
```

Figure 15 — Rounding modes and common subexpression elimination.

6.7.4. Rationale and Alternatives

To ease running identical code under different rounding modes and testing for sensitivities, an alternative proposal is to treat the rounding mode as an implicit parameter to all methods and to have all rounding mode changes be made by calls to `setRound`. This approach is taken by RealJava [23] and C9X [94]. In that case, the rounding mode of a method is inherited from the caller and dynamic rounding modes are always supported. However, explicit calls to `setRound` are unstructured and there is no guarantee that scoping is enforced, even though that is usually the desired behavior. The `rounding` declarations allow the compiler and the programmer to reason about and identify the portions of the code that could be run under non-default rounding modes. Therefore, Borneo chooses `rounding` declarations to control the common uses of rounding modes while still permitting more dynamic control when necessary.

The `getRound` method is included for completeness and convenience. No `getRound` method is strictly necessary since a method does not inherit the rounding mode of its caller under normal circumstances. Since the rounding mode is only changed under programmer direction, the programmer can keep the current rounding mode in a variable. The rounding mode can also be determined by performing a known series of simple computations, such those in Figure 6.

Initially a more object-oriented language representation of rounding modes was considered. Instead of integers, a `RoundingMode` class with subclasses for each rounding mode was proposed. The type of such an object would encode the rounding mode. However, since `RoundingMode` would be a reference type, it would be possible for a `RoundingMode` variable to be `null`. Therefore, a `rounding` declaration would throw two kinds of exceptions instead of one; an `UnknownRoundingMode` exception (due to a user-defined subclass of `RoundingMode`) and a `NullPointerException`. Using an integer representation for rounding modes eliminates the `NullPointerException`, more closely matches the hardware representation, and allows for more natural iteration over rounding modes. With the integer representation, if the *`Expression`* given to `rounding` is a constant, using existing integer constant folding machinery the compiler may be able to infer whether a particular `rounding` declaration cannot throw (or always throws) an exception.

6.7.5. Rounding Mode Examples

The following examples show how rounding modes can be used dynamically in polynomial evaluation and statically in interval arithmetic.

6.7.5.1. Polynomial evaluation

The evaluation of a polynomial with floating point coefficients usually involves some roundoff errors. Sometimes it is useful to obtain upper and lower bounds of a polynomial and its derivative at a given value. Directed rounding modes can be used to accomplish this task, as shown in Figure 16.

```
double[] extrema(   double a[],         // array of coefficients a0x^n + a1x^(n-1) + a2 x^(n-2)… where n is the degree of the polynomial
                    double x,           // where to evaluate the polynomial
                    boolean upper)      // whether to calculate an upper or lower bound
{
    double    q,                        // value of the derivative of the polynomial
              p,                        // value of the polynomial
              answer[] = new double[2]; // returned pair
    boolean   negated = false;          // whether or not coefficients have been negated to evaluate at x < 0
    int i;

    if(x < 0.0)     //adjust coefficients so correct extrema  is calculated
        {
          // to correctly find the extrema of a polynomial evaluated at a negative value, the coefficients o  f the
          // odd powers must be negated, instead of evaluating  f(x) = sum_{i=0}^{n} a_i x^(n-i) , when x < 0, we evaluate
          // g(-x) = sum_{i=0}^{n} b_i x^(n-i)  where  b_i = (-1)^((n-i) mod 2) · a_i

          negated = true;
          x = –x;

          for(i = (a.length %2?1:0); i <= a.length; i += 2)
              a[i] = –a[i];
        }

    // set rounding mode to ±∞
    rounding (upper?   Math.TO_POSITIVE_INFINITY:
                       Math.TO_NEGATIVE_INFINITY);

    // Use Horner's method to evaluate the polynomial and its derivative
    q  = 0.0;
    p = a[0];

    for(i=1; i <= a.length; i++)
    {
        q = x * q + p;
        p = p * x + a[i];
    }

    //adjust array coefficients back to original values
    if(negated)
        {
          for(i =(a.length %2?1:0); i <= a.length; i += 2)
              a[i] = –a[i];
        }

    answer[0] = p;
    answer[1] = q;
    return answer;
}
```

Figure 16 — Dynamic rounding modes used to find the extreme value of a polynomial at a given input.

6.7.5.2. Interval arithmetic

Interval arithmetic at a reasonable cost was the main impetus for including directed rounding in the IEEE 754 standard. Figure 17 and Figure 18 show the core computation for interval addition and multiplication. For interval addition, the new lower bound is calculated by adding the lower bounds of the argument intervals rounding toward negative infinity while the new upper bound is gotten by adding the input upper bounds rounding toward positive infinity. Due to the possibility of intervals whose endpoints have different signs, interval multiplication must examine all four pair-wise products of the input upper and lower bounds. These two small pieces of code are only the skeletons needed in an actual interval package; a usable interval package must be concerned with representing various types of degenerate intervals [58] as well as limiting spurious floating point exceptions [80]. Interval division with a divisor interval containing zero also causes complications [81].

```
Interval add(Interval a, Interval b)
{
    double lower_bound, upper bound;

    rounding Math.TO_NEGATIVE_INFINITY;
    lower_bound = a.lower_bound + b.lower_bound;

    rounding Math.TO_POSITIVE_INFINITY;
    upper_bound = a.upper_bound + b.upper _bound;

    return new Interval(lower_bound, upper_bound);
}
```

Figure 17 — Core computation for interval addition.

```
Interval multiply(Interval a, Interval b)
{
    double lower_bound, upper bound;

    rounding Math.TO_NEGATIVE_INFINITY;
    lower_bound = min(   a.lower_bound * b.lower_bound,     a.lower_bound * b.upper_bound,
                         a.upper_bound * b.lower_bound,     a.upper_bound * b.upper_bound);

    rounding Math.TO_POSITIVE_INFINITY;
    upper_bound = max(   a.lower_bound * b.lower_bound,     a.lower_bound * b.upper_bound,
                         a.upper_bound * b.lower_bound,     a.upper_bound * b.upper_bound);

    return new Interval(lower_bound, upper_bound);
}
```

Figure 18 — Core computation for interval multiplication.

6.8. Floating Point Exception Handling

The IEEE 754 standard provides two techniques for dealing with "exceptional" floating point conditions, setting sticky flags and calling a trap handler. However, no existing modern high-level language provides a structured mechanism for using these features. Borneo has support for both exception handling policies. A Java method's declaration includes information on the return type, the number and type of parameters, and on what checked exceptions can be thrown. Borneo method declarations also include a "flag signature," the sticky flags a method accepts from its caller and the flags a method returns to its caller. A new control structure and library methods are provided to access the flags. For floating point exceptions, instead of allowing users to provide their own trap handlers, Borneo integrates floating point traps into the existing Java exception mechanism

By using the traps and sticky flags robust algorithms can be made to run much faster. Full language support enables better optimization. Simple algorithms tend to be acceptable for typical data, but other valid inputs can cause such algorithms to fail. While a given algorithm that uses exception handling, either flags or some form of trapping, can be replaced by an analogous algorithm without exception handling, removing the exception handling capability often exacts a considerable penalty in decreased performance and increased complexity. Using floating point exception handling allows simple algorithms to run quickly in the common case; exceptional cases tend to be rare and can be dealt with using slower, more cautious algorithms. This approach to making fast, robust numerical algorithms was applied in [25] to several LAPACK [13] routines. By using flags to record exceptional conditions, significant speed improvements were made to a number of linear algebra algorithms.

6.8.1. Exceptions and Trapping Floating Point Operations

As discussed in [56], existing numerical programs use floating point exception handling capabilities to perform a limited number of tasks.

1. *IEEE 754 non-trapping mode*: Knowledge of how NaNs and infinities arise and propagate can lead to shorter code with fewer branches.

2. *stopping the computation*: Stopping the computation can be used when a simple algorithm's limits have been exceeded and a more robust algorithm is needed. Alternatively, the entire calculation can be aborted if a semantic constraint, such as no NaNs, has been violated.
3. *presubstitution*: Presubstitution is a generalization of returning IEEE special values as the result of an exceptional operation. Instead of a returning a fixed value, such as infinity on divide by zero, presubstitution computes a value to be returned when an exceptional condition arises. This technique can make some computations, such as computing a continued fraction and its derivative, run with fewer defensive tests and branches in the inner loop [60].
4. *extended exponents*: Exponent extension augments the existing exponent field of a floating point number with a 32 bit integer. This yields an enormous range, but does not increase the precision of the number. However, some computations, such as computing the determinant of a high-dimensionality matrix benefit from extended range without extended precision.

6.8.1.1. Changing Trapping Status

Borneo uses lexically scoped `enable` and `disable` declarations to control the trapping status of a section of code. Enabled conditions (any combination of invalid, overflow, divide by zero, underflow, and inexact) allow the corresponding floating point exception to be thrown as a side effect of an arithmetic operation. The Borneo trap handler handles a hardware floating point trap and throws the corresponding Borneo exception (Figure 19). [22] The default trapping status is `disable all` (non-trapping mode). If a non-default trapping status is being used, the default trapping status must be restored before a method call and the non-default status restored afterwards.

In general, a single kind of floating point operation (addition, division, etc.) is not capable of generating all five exceptional conditions, regardless of input. Table 7 lists what conditions different operations can generate. Enabling divide by zero in a section of code having only multiplies and adds would have no affect since multiplies and adds cannot generate the divide by zero signal. While addition and multiplication can generate four of the five exceptional conditions and division can generate all five, at most two of the conditions can be generated simultaneously by executing a single instruction (overflow and inexact, underflow and inexact). If both overflow/inexact or underflow/inexact are being trapped on, the overflow/underflow exception is thrown instead of inexact.

Like most Java exceptions, Borneo floating point exceptions are synchronous. Floating point exceptions are also source-code precise. Since Borneo's floating point exceptions are checked exceptions, they must be explicitly caught by a `catch` block or declared in the method's `throws` clause. When a condition is `enabled`, the corresponding sticky flag is not set. Figure 22 details the syntax changes to support floating point exceptions and sticky flags. Each condition can appear at most once in a *TrappingConditions* list, and `all` and `none` must only appear by themselves.

Exceptions of type `FloatingPointException` and its subclasses can also be created and thrown explicitly by the programmer. Besides the usual exception constructors and methods, `OverflowException` and

---

[22] The IEEE 754 standard has different rules for generating the underflow condition based on the trapping status. If trapping on underflow is off, loss of precision and tininess are required for the underflow flag to be set. If trapping on underflow is on, any non-zero result smaller in magnitude than the minimum normal value will generate an underflow trap; no loss of precision needs to occur. The `enable` blocks throw exceptions according to these trapping enabled semantics.

The standard allows two options for detecting tininess: either before or after rounding. Therefore, different IEEE implementations can yield different signals (but the same value) for some combinations of operations and arguments. The differences are only visible if the answer is equal to ±MIN_NORMAL.
One computation that can be used to determine what policy a processor uses is

```
// clear flags
float f = 2.3509886e-38f * 0.5f;
// if underflow is set, tininess is detected before rounding, otherwise tininess is detected after rounding
```

The left hand operand is equal to `scalb(Float.MAX_VALUE, -253)`, a significand of all ones with the minimum normal exponent. Multiplying this value by `0.5` rounds up to `Float.MIN_NORMAL`.

A program designed to handle underflow should work properly if tininess is detected before or after rounding.

`UnderflowException` include additional constructors and methods to initialize and convey floating point values. For the format causing the overflow or underflow exception, the corresponding *`type`*`Value` method of the exception returns the wrapped exponent result as required by the standard. For example, in the following small method, if the multiplication of `a` and `b` overflows, a result with the same significand bits as `a * b` but with an exponent 192 less than the true result is returned.

```
public static float wrapped_multiply(float a, float b)
{
   try
      {
         enable overflow
         return a * b;
      }
   catch (OverflowException e)
      {
         return e.floatValue();      // same as value as (float)scalb( (double)a * (double b), –192)
      }
}
```

6.8.1.2. Inferring Floating point Exceptions

Table 7 — Floating point operations and the exceptional conditions they can generate; different causes of invalid are distinguished.

	Overflow	Underflow	Inexact	Invalid	Divide by Zero
+, –	X	X	X	X, $\infty - \infty$	
*	X	X	X	X, ∞*0	
/	X	X	X	X, ∞/∞, 0/0	X
remainder[23]				X, invalid remainder from *x* REM 0 or ∞ REM *y*	
$\sqrt{\ }$		X	X	X, $\sqrt{x}$, x<0	
comparison other than ==, !=				X, compare with NaN	
conversion between float and int types (such as casts)[24]			X	X, bad format conversion	
conversion from a float format to a narrower float format	X	X	X		

[23] Java's remainder operation on floating point numbers is not the IEEE 754 remainder but Borneo will throw an invalid remainder exception under the same conditions as the invalid remainder exception would be thrown for the IEEE 754 remainder.
[24] Unlike IEEE 754, the IEEE 854 standard clearly states that floating point to integer conversion can raise the inexact flag.

```
public class FloatingPointException extends Exception {} //checked exception
//IEEE 754 floating point exceptions
public class InvalidException extends FloatingPointException {}
    public final class InfinityMinusInfinityException extends InvalidException{}
    public final class InfinityTimesZeroException extends InvalidException{}
    public final class ZeroOverZeroException extends InvalidException{}
    public final class InfinityOverInfinityException extends InvalidException{}
    public final class InvalidRemainderException extends InvalidException{}
    public final class SquareRootOfNegativeException extends InvalidException{}
    public final class BadFormatConversionException extends InvalidException{}
    public final class ComparisionOnNaNException extends InvalidException{}

public final class InexactException extends FloatingPointException {}

public class OverflowException extends FloatingPointException{/* see section 8.12 for a full definition */}

public class UnderflowException extends FloatingPointException{/* see section 8.12 for a full definition */}

public final class DivideByZeroException extends FloatingPointException {}
```

Figure 19 — IEEE floating point exception class hierarchy.

Since floating point exceptions are checked exceptions, the Borneo compiler must infer from `enable` blocks, floating point operations, and method calls what floating point exceptions can be thrown by a block of code and which (if any) floating point exceptions need to be included in a method's `throws` clause. Borneo uses a simple, conservative, flow-insensitive analysis to determine which exceptions may be thrown. When a condition is enabled, the exceptions a given operation can throw are determined by examining Table 7. Different causes for invalid are distinguished. Borneo assumes non-literal arguments to a floating point operation can take on all possible values, even when analysis of the code could determine otherwise. Borneo only uses local information to infer which exceptions can be thrown. For example, in Figure 21 Borneo infers the expression in the first `return` statement, `2.0 / x`, can throw a divide by zero exception even though this expression is guarded by a test for `x != 0.0`. Borneo does take into account the value of explicit literals; for example, `x /2.0` is known not to cause a divide by zero exception. If two `float` values are promoted to `double` and operated on, the `double` operation signals neither underflow, overflow, nor inexact, but Borneo does not attempt to use such relationships.

There are situations where a compiler could infer an exception is always thrown. For example, in a region where invalid is enabled, `x >= NaN` always causes an exception to be thrown. However, Borneo does not use such analyses to determine the reachability of statements (JLS §14.19). Borneo only determines which floating point exceptions an expression may throw, not what exceptions must be thrown. This is equivalent to modeling operators on primitive floating point types as methods having `throws` clauses listing the appropriate exceptions.

Borneo's algorithm for inferring the floating point exceptions an expression may throw is given in Figure 20. If neither operand to a binary operator is a literal, the full set of exceptions for that operator in Table 7 is intersected with the enabled conditions (enabling invalid allows subclasses of `InvalidException` to be thrown). When a single operand is a literal, the algorithm checks for special floating point values and eliminates exceptions that cannot be thrown. For example, if the divisor is a non-zero literal, the divide by zero exception cannot be thrown. However, Borneo's algorithm is not as precise as possible. In particular, the value of a literal is not used to determine if overflow or underflow is not possible. For example, if one operand is between –1.0 and 1.0, a multiplication cannot overflow since the magnitude of the result is less than or equal to the magnitude of the other operand. Similar properties hold for division, addition, and subtraction (see Table 8). Borneo does not use the values of literals in this manner for several reasons. First, the exception limits for addition and subtraction are dependent on the rounding mode. For example, under round to nearest, there are non-zero values small enough such that adding these values to another number cannot cause overflow. Such numbers do not exist under round to $\pm\infty$. Second, the various threshold values differ for each format. Since `indigenous` does not correspond to one format, `float` and `double` expressions would have more precise exception information calculated. Therefore, `double` and `indigenous` expressions with literals of the same value would have different exceptions inferred even when `indigenous` was implemented as `double`.

```
// determine what floating point exceptions an IEEE 754 floating point expression may throw given the trapping status and rounding mode
SetOfExceptions infer(Expression,   // Expression to have exceptions inferred
                      FPState)      // Trapping status and rounding mode, rounding mode can be a particular value or indeterminate
{ if(isliteral(Expression) return {};   // the empty set
else if (the expression is a unary operation of the form op Expression1 where op is unary + or unary – or a widening cast ),
    return infer(Expression1, FPState)
else if (the expression is a narrowing cast ) return ({Overflow, Underflow, Inexact} ∩ FPState.trapping) ∪ infer(Expression1, FPState);
else // Expression is of the form Expression1 op Expression2
    if(isLiteral(Expression1) && isLiteral(Expression2) {
        // perform operation under trapping status and rounding mode in FPState and see what exceptions are thrown.
        /* if the rounding mode is indeterminate, take the union of exceptions thrown under all rounding modes */                }
    else  /* at least one operand is not a literal */       {
        switch (op){
            case "<", ">", "<=", ">=":
                if(!isLiteral(Expression1) && !isLiteral(Expression2)
                    return (infer(Expression1, FPState) ∪ infer(Expression2, FPState)) ∪ ({ComparisionOnNaNException } ∩ FPState.trapping);
                else {
                    if (isLiteral(Expression1)) {LitExpr = Expression1; Expr=Expresion2} else {LitExpr = Expression2; Expr=Expression1}
                    if  (isNaN(LitExpr.literal_value) ) {
                        return ({ComparisionOnNaNException} ∩ FPState.trapping) ∪ infer(Expr, FPState);}
                    else
                        return infer(Expr, FPState); }
        case "/":
            if (!isLiteral(Expression1) && !isLiteral(Expression2)) {
                return infer(Expression1, FPState) ∪ infer(Expression2, FPState) ∪ (FPState.trapping ∩
                        {Overflow, Underflow, Inexact, ZeroOverZeroException, InfinityOverInfinityException, DivideByZero})}
            // different exceptions are thrown depending on if the divisor or dividend is a literal
            else if (isLiteral(Expression1) && !isLiteral(Expression2)) {   // literal dividend
                switch (Expression1.literal_value) {
                    case ±∞:        return ({InfinityOverInfinityException} ∩ FPState.trapping) ∪ infer(Expression2, FPState);
                    case ±0.0:      return ({ZeroOverZeroException} ∩ FPState.trapping) ∪ infer(Expression2, FPState);
                    case NaN:       return infer(Expression2, FPState);
                    default:    return ({Overflow, Underflow, Inexact, DivideByZero}∩ FPState.trapping) ∪ infer(Expression2, FPState);
                }}
            else if (!isLiteral(Expression1) && isLiteral(Expression2)) {   // literal divisor
                switch (Expression2.literal_value) {
                    case ±∞:        return ({InfinityOverInfinityException} ∩ FPState.trapping) ∪ infer(Expression1, FPState;
                    case ±0.0:      return ({ZeroOverZeroException, DivideByZero}∩ FPState.trapping) ∪ infer(Expression1, FPState);
                    case ±1.0, NaN: return return infer(Expression1, FPState);
                    default:    return ({Overflow, Underflow, Inexact}∩ FPState.trapping) infer(Expression1, FPState); }}
        case "+", "–":
            // + and – are commutative, same exceptions result if left or right operand is a literal
            if(!isLiteral(Expression1) && !isLiteral(Expression2)) {
                return infer(Expression1, FPState) ∪ infer(Expression2, FPState) ∪ (FPState.trapping  ∩
                        {Overflow, Underflow, Inexact, InfinityMinusInfinityException})}
            else {
                if(isLiteral(Expression1)  {LitExpr = Expression1, Expr = Expression2} else {LitExpr = Expression2; Expr = Expression2;}
                switch(Expr.literal_value) {
                    case ±∞:        return infer(Expr, FPState) ∪ ({InfinityMinusInfinityException} ∩ FPState.trapping);
                    case NaN, ±0.0: return infer(Expr, FPState);
                    default:        return infer(Expr, FPState) ∪ ({Overflow, Underflow, Inexact}∩ FPState.trapping); }}
        case "*":
            if(!isLiteral(Expression1) && !isLiteral(Expression2)) {
                return infer(Expression1, FPState) ∪ infer(Expression2, FPState) ∪ (FPState.trapping  ∩
                                                {Overflow, Underflow, Inexact, InfinityTimesZeroException}) }
            else {
                if(isLiteral(Expression1)  {LitExpr = Expression1, Expr = Expression2}else {LitExpr = Expression2; Expr = Expression2};
                switch(Expr.literal_value) {
                    case ±∞, ±0:        return infer(Expr, FPState) ∪ ({InfinityTimesZeroException} ∩FPState.trapping);
                    case ±1.0, NaN:     return infer(Expr, FPState);
                    default:            return infer(Expr, FPState) ∪ ({Overflow, Underflow, Inexact}∩FPState.trapping);        }}
    }}
}
```

Figure 20 — Pseudocode to determine what floating point exceptions an expression may throw.

Table 8 — Conditions that could be used for more precise floating point exception inference.

Operation	Exception Limitations if one operand is known[25] (assume operands are of the same format)
multiplication	• if the known operand has an absolute value ≤ 1.0, overflow cannot occur
division	• if the divisor has an absolute value ≥ 1.0, overflow cannot occur • if the divisor has an absolute value ≤ `scalb(MIN_VALUE, MAX_EXPONENT – 2)`, underflow cannot occur • if the dividend has an absolute value ≤ `scalb(MAX_VALUE, –(SIGNIFICAND_WIDTH – 2 +MAX_EXPONENT))`, overflow cannot occur • if the dividend has an absolute value ≥ `nextAfter(4.0, 0.0)`, underflow cannot occur
addition, subtraction	• under round to nearest, if the known operand has an absolute value less than `scalb(MAX_VALUE, –(SIGNIFICAND_WIDTH+1))`, overflow cannot occur (no such value exists under round to ±∞ and round to zero has a different threshold) • if the known operand has an absolute value greater than `nextAfter(scalb(MIN_NORMAL, (SIGNIFICAND_WIDTH –1), ∞)`, no underflow can occur

Since the overflow and underflow behavior of an operation depends on the rounding mode, even if an operation has two literal arguments, the runtime exceptional behavior may vary. In the absence of `rounding` declarations, Borneo assumes round to nearest is in effect. Therefore, changes to the rounding mode other than through `rounding` declarations can cause unexpected exceptions to be thrown. If an in-scope `rounding` declaration has a constant valid argument, that rounding mode is used for exception inference. Otherwise, the compiler assumes any rounding mode can be in effect when an expression executes. Therefore, in such cases, the union of all possible exceptions is returned.

[25] This table only addresses literal values that imply overflow or underflow cannot occur; inexact is not discussed since for a given operation and one particular regular floating point value (other than ±1.0 for multiplication/division and ±0.0 for addition/subtraction), another floating point value can be constructed that signals inexact with the first value. In addition and subtraction, for any regular floating point literal, *L*, there exists another number, *R*, such that *R*'s exponent is either p+1 greater or smaller than *L*'s exponent. Therefore, when *R* and *L* are added (or subtracted) one of the operands rounds away, an inexact operation. For multiplication, multiplying MIN_VALUE by any regular value with an absolute value ≤ ½ will cause an underflow and inexact. Multiplying MAX_VALUE by a number greater in magnitude than 1.0 will cause overflow and inexact. The remaining floating point numbers between ½ and 1.0 in magnitude have at least two 1's in their significands. Therefore, multiplying numbers between ½ and 1.0 by a number with all 1's in its significand will signal inexact. A similar construction exists for division.

```
double calculate(double x) throws InvalidException, DivideByZeroException
{
double y;
enable divideByZero;
if(x != 0.0)
   {
      y = x / 2.0;        // Borneo infers this statement cannot throw DivideByZeroException

      return 2.0 / x;     // Borneo infers this statement can throw DivideByZeroException
   }

disable divideByZero;
enable invalid;
if (!Double.IsInfinite(x))
   {
      y = x – x;          // Borneo infers this statement can throw InfinityMinusInfinityException
                          // even though the statement would not be reached if x were infinite

      return y * x;       // Borneo infers this statement can throw InfinityTimesZeroException
                          // even though this statement would not be reached if x were infintite
   }
else
   {
      y  = x/x;           // Borneo infers this can throw InfinityOverInfinityException and
                          // ZeroOverZeroException
      return y;
   }
}
```

Figure 21 — Code to illustrate limits of Borneo floating point exception inference.

6.8.1.3. Specification

Augmented Java syntax
LocalVariableDeclarationStatement:
    FloatingPointTrappingDeclaration `;`

New Borneo Productions
FloatingPointTrappingDeclaration:
    `enable` *TrappingConditions*
    `disable` *TrappingConditions*

TrappingConditions:
    TrappingCondition
    TrappingConditions `,` *TrappingCondition*

TrappingCondition: one of
    `overflow underflow divideByZero invalid inexact all none`

Figure 22 — Changes to Java grammar to support throwing floating point exceptions.

The `enable`/`disable` declarations are lexically scoped; a condition is trapped on until the end of the enclosing block or an overriding declaration is encountered. An `enable`/`disable` declaration in the optional initialization code of a `for` loop affects the remainder of the initialization, the loop test, loop update, and loop body. Multiple `enable` declarations in the same block of code have a cumulative effect, as shown in Figure 23. The extent of an `enable`/`disable` declaration is strictly textual; if an `enable` declaration appears in the middle of a loop, the trapping status of one iteration does not carry over to the next; see Figure 24 for an example. If the inexact trap is enabled, code may run very slowly due to pipelining effects on modern processors.

```
trapper(float a, float b) throws OverflowException, UnderflowException;
{
    float c;
    enable overflow;
    c = a + b;      //Overflow Enabled

    enable underflow;
    c = a * b;      //Overflow and underflow Enabled

    disable overflow;
    c = a /b;       //Only underflow Enabled
}
```

Figure 23 — Borneo code to illustrate scoping of `enable`/`disable` declarations.

```
float a, b, c, d;
for(i = 0; i < MAX; i++)
{
    a = b + c;      // this statement can never thrown an exception, a equals  ∞ if b + c overflows
    enable overflow;
    d = b + c;      // this statement throws an overflow exception if b + c would round to a value  larger than MAX_VALUE
}
```

Figure 24 — Scoping of an `enable` declaration in a loop.

Floating point exceptions are treated much like ordinary Java exceptions. Subclasses of `FloatingPointException` may be thrown and caught explicitly in addition to being generated by operations on primitive floating point values. For example, user-defined numeric classes may wish to throw IEEE 754 style exceptions. Since `FloatingPointException` and its subclasses are checked exceptions, arithmetic exceptions must be part of a method's signature if not caught inside the method. Floating point exceptions can escape `enable` blocks if the appropriate `catch` clause is not present. Enabling exceptions and allowing exceptions to escape methods can be used to halt the calculation once an infinity or NaN is generated. The subclasses of `InvalidException` can be used by the programmer to help determine which operation caused an exception so that the appropriate action can be taken.

The classes `OverflowException` and `UnderflowException` are used to return the exponent-adjusted result to the user, as requested by the standard. For example, if a computation on `float` values overflows and is caught with the following structure,

```
catch(OverflowException e){return e.floatValue()}
```

calling the `floatValue` method returns the significand bits of the overflowed operation with an exponent adjusted by `-Float.BIAS_ADJUST`. Underflowed values have `BIAS_ADJUST` added to their exponent.

With Borneo's floating point exception hierarchy, the programmer cannot immediately determine the type of the operands in an overflowing or underflowing operation. The type of the operands can be inferred from results of calling `floatValue`, `doubleValue`, and `indigenousValue` on the overflow or underflow exception. On overflow, if a platform has `float`, `double`, and `indigenous` as distinct formats, the narrowest format returning a finite value is the one which caused the exception. Similarly, on underflow, the narrowest format with a non-zero value caused the exception. If `indigenous` maps to the `double` format, using only information available from the exception it is not possible to distinguish an exception on `double` operands from an exception on `indigenous` operands. Table 11 gives additional information on the values held by overflow and underflow exceptions under different exception generation conditions.

Table 9 — Values returned by *`type`*`Value` by arithmetic operations under different exceptional conditions.

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="3">Magnitude of result</td></tr>
<tr><td><code>floatValue()</code></td><td><code>doubleValue()</code></td><td><code>indigenousValue()</code></td></tr>
<tr><td colspan="2">Overflow</td><td></td><td></td><td></td></tr>
<tr><td colspan="2"><code>float</code></td><td>exponent adjusted value $\approx 2^{-65}$ to $\approx 2^{62}$</td><td>full precision result $\approx 2^{127}$ to $\approx 2^{254}$</td><td>full precision result $\approx 2^{127}$ to $\approx 2^{254}$</td></tr>
<tr><td rowspan="2"><code>double</code></td><td>indigenous≡double extended</td><td rowspan="2">infinity</td><td rowspan="2">exponent adjusted value $\approx 2^{-513}$ to $\approx 2^{510}$</td><td>rounded result $\approx 2^{1032}$ to $\approx 2^{2046}$</td></tr>
<tr><td>indigenous≡double</td><td>same as <code>doubleValue</code></td></tr>
<tr><td rowspan="2"><code>indigenous</code></td><td>indigenous≡double extended</td><td rowspan="2">infinity</td><td>infinity</td><td>exponent adjusted result $\approx 2^{-8292}$ to $\approx 2^{7990}$</td></tr>
<tr><td>indigenous≡double</td><td>same as <code>indigenousValue</code></td><td>exponent adjusted value $\approx 2^{-513}$ to $\approx 2^{510}$</td></tr>
<tr><td colspan="2">Underflow</td><td></td><td></td><td></td></tr>
<tr><td colspan="2"><code>float</code></td><td>exponent adjusted result $\approx 2^{-106}$ to $\approx 2^{42}$</td><td>full precision result $\approx 2^{-298}$ to $\approx 2^{-150}$</td><td>full precision result $\approx 2^{-298}$ to $\approx 2^{-150}$</td></tr>
<tr><td rowspan="2"><code>double</code></td><td>indigenous≡double extended</td><td rowspan="2">0.0</td><td rowspan="2">exponent adjusted result $\approx 2^{-614}$ to $\approx 2^{461}$</td><td>rounded result $\approx 2^{-2150}$ to $\approx 2^{-1076}$</td></tr>
<tr><td>indigenous≡double</td><td>same as <code>doubleValue</code></td></tr>
<tr><td rowspan="2"><code>indigenous</code></td><td>indigenous≡double extended</td><td rowspan="2">0.0</td><td>0.0</td><td>exponent adjusted result $\approx 2^{-8316}$ to $\approx 2^{8130}$</td></tr>
<tr><td>indigenous≡double</td><td>same as <code>indigenousValue</code></td><td>exponent adjusted result $\approx 2^{-614}$ to $\approx 2^{461}$</td></tr>
</table>

6.8.1.4. Alternatives and Rationale

The interface to floating point trap handlers varies greatly from system to system so supporting construction of portable user-defined trap handlers would be difficult. By incorporating floating point traps into the existing exception mechanism, the implementation burden rests on the compiler/runtime writer instead of the compiler user. Since floating point exceptions can only occur at well-defined points in the program (namely arithmetic operations), floating point exceptions are synchronous, as opposed to asynchronous exceptions which can occur at any time.

An alternative design to the current exception hierarchy is to have overflow and underflow subclasses for each primitive floating point type. However, on machines where `indigenous` and `double` are implemented with the same format, the hardware has no a priori way of discriminating between `indigenous` and `double` operations to determine which exception to throw. The trap handler would need additional information from the running Borneo program; communicating such information to the trap handler would be troublesome.

The current proposal does require additional checking to distinguish between exceptions from different formats in mixed-format code, but if a finer granularity is needed, homogenous-format regions can have their own `catch` blocks. Having unified overflow and underflow exceptions also simplifies the compiler's determination that all floating point exceptions are either caught or declared in the `throws` clause of a method. New numeric types can also more easily and uniformly extend a single `UnderFlowException` or `OverFlowException` class.

6.8.1.5. Floating Point Exception Examples

The following examples demonstrate different techniques of improving numerical algorithms by using floating point exceptions. Computing the geometric mean serves as the basis for an extended example illustrating a number of techniques for making robust algorithms.

6.8.1.5.1. Overflow and underflow while finding the geometric mean

A general approach for using floating point exceptions is

```
try
    {
        enable exceptions;
        Do the calculation using a simple fast algorithm
    }
catch (FloatingPointException e)
    {
        Repeat the calculation using a fault-tolerant algorithm
    }
```

The common (non-exceptional) case uses a simple algorithm to perform the calculation. If an exception occurs, the calculation is repeated using a more sophisticated or fault-tolerant algorithm. Exceptions can also be used to record some information (such as a how many over/underflows have occurred) so that the calculation can be continued and the answer adjusted at the end.

For example, the geometric mean of a series of positive real numbers is defined as follows:

$$\bar{x} \equiv \sqrt[n]{\prod_{i=1}^{n} x_i}$$

The “naive” version of the algorithm in Figure 25 simply multiplies all array elements together (in the following geometric mean algorithms the input array is assumed to contain neither infinities nor NaNs).

```
double geometricMean(double array[])
{
    double product = 1.0;

    // check for zero length array
    if(array.length == 0)
        return NaNd;

    // Iterate over the array elements
    for (int i = 0; i < array.length; i++)
        {
            double element = array[i];

            // Check for illegal non-positive array elements
            if (element <= 0.0)
                return NaNd;

            // Multiply the array elements together
            product *= element;
        }

    // Return the nth root of the product
    return Math.pow(product, 1.0/(double)array.length);
}
```

Figure 25 — Simple algorithm to compute the geometric mean.

The limitation of this simple algorithm is that for a large array, or an array containing extremely large or small values, the variable `product` can potentially overflow or underflow, returning infinity or zero even though the answer is representable. The ordering of the numbers in the array also affects whether or not a proper answer is returned. For example, using the following array of `double` numbers with the above program

2^{1023}	2^{1023}	2^{-1023}	2^{-1023}

generates an overflow to infinity on the first multiply, while on a permutation of this array

2^{1023}	2^{-1023}	2^{1023}	2^{-1023}

the algorithm returns the correct answer of 1.0.

```
double geometricMean(double array[])
{
    double product = 1.0;
    // first try the simple algorithm..

    // check for zero length array
    if(array.length == 0)
        return NaNd;

    try
    {
        enable overflow, underflow;
        // Iterate over the array elements
        for (int i = 0; i < array.length; i++)
            {
                double element = array[i];

                // Check for illegal non-positive array elements
                if (element <= 0.0)
                    return NaNd;

                // Multiply the array elements together
                product *= element;
            }

        // Return the nth root of the product
        return Math.pow(product, 1.0/(double)array.length);
    }
    // if the simple algorithm doesn't work, try a more expensive one
    catch(FloatingPointException e)
        {
            //call sophisticated program...
        }
}
```

Figure 26 — Simple algorithm to compute the geometric mean, calls a robust algorithm when necessary.

One solution is to enable exceptions in the simple routine and if an overflow or underflow is encountered, to fallback to a slower, more robust algorithm, as show in Figure 26. However, the robust routine still needs to be written.

Using the mathematical properties of the problem, another way to eliminate the troublesome exceptions is to scale the array elements so that overflow or underflow never occur. Knowing that floating point numbers are represented internally as $\pm s \cdot 2^e$, the computation can be modified to use this identity as shown in Figure 27.

$$\bar{x} = \sqrt[n]{\prod_{i=1}^{n} x_i} = \sqrt[n]{\prod_{i=1}^{n} \left(s_i \cdot 2^{e_i}\right)} = \exp\left(\frac{\ln\left(\prod_{i=1}^{n}\left(s_i \cdot 2^{e_i}\right)\right)}{n}\right) =$$

$$\exp\left(\frac{\ln\left(\prod_{i=1}^{n} s_i\right) + \ln\left(\prod_{i=1}^{n} 2^{e_i}\right)}{n}\right) = \exp\left(\frac{\ln\left(\prod_{i=1}^{n} s_i\right) + \left(\sum_{i=1}^{n} e_i\right) \cdot \ln 2}{n}\right)$$

Figure 27 — Rewriting the geometric mean.

This leads us to write the robust algorithm for calculating the geometric mean shown in Figure 28.

```
static final double ln2 = Math.log(2.0);

double geometricMean(double array[])
{
    double product = 1.0;
    int exp_adjust = 0;          // assume exp_adjust will not experience integer overflow

    // check for zero length array
    if(array.length == 0)
        return NaNd;

    for (int i = 0; i < array.length; i++)
        {
            int e;
            double element = array[i];

            // Check for illegal non-positive array elements
            if (element <= 0.0)
                return NaNd;

            // Separate each array element into an integer exponent and a significand normalized
            // to the range 0.0 < n < 2.0; accumulate the exponent and significand parts separately
            e = (int) Math.logb(element);                // logb(0) == –∞ does not occur, assuming no NaN's or infinities in array
            product *= Math.scalb(element, –e);
            exp_adjust += e;

            if (product >= 2.0)
                {
                    product = Math.scalb(product, –1);
                    exp_adjust++;
                }
        }

    // Calculate the nth root of the product, using either the pow method or the formula in  Figure 27
    return (exp_adjust == 0)
        ? Math.pow(product, 1.0/(double)array.length);
        : Math.exp((Math.log(product) + (double)exp_adjust * ln2) / (double)array.length);
}
```

Figure 28 — Robust geometric mean algorithm without using exceptions.

This version works correctly for every case without using exceptions. However, it is considerably slower than the naive version, even though for most common inputs the naive version would have worked correctly.

As in this example, many robust numerical algorithms must perform a number of tests to ensure that the algorithm works correctly for every possible input. These tests often take longer to perform than the actual calculation, and for most inputs they are not necessary. A much better solution is to detect and handle the rare troublesome inputs using exceptions. In the algorithm in Figure 29, exceptions are used to detect when overflow or underflow actually occurs, and only then perform scaling.

```
static final double ln2 = Math.log(2.0);

double geometricMean(double array[])
{
    double product = 1.0;
    int exp_adjust = 0;          // assume exp_adjust will not experience integer overflow

    // check for zero length array
    if(array.length == 0)
        return NaNd;

    for (int i = 0; i < array.length; i++)
        {
            double element = array[i];

            // Check for illegal non-positive array elements
            if (element <= 0.0)
                return NaNd;

            // First try finding the product using naive multiplication.
            try
                {
                    enable overflow, underflow;
                    product *= element;
                }

            // If an overflow or underflow exception occurs, grab the scaled value of the product,
            // and increment or decrement the exponent adjustment.  Then continue the calculation as before.
            catch (OverflowException e)
                {
                    product = e.doubleValue();
                    exp_adjust += Double.BIAS_ADJUST;
                }
            catch (UnderflowException e)
                {
                    product = e.doubleValue();
                    exp_adjust –= Double.BIAS_ADJUST;
                }
        }

    // Calculate the nth root of the product, using either the pow method or the formula in  Figure 27
    return (exp_adjust == 0)
            ? Math.pow(product, 1.0/(double)array.length);
            : Math.exp((Math.log(product) + (double)exp_adjust * ln2)/ (double)array.length);
}
```

Figure 29 — Robust geometric mean algorithm using exceptions.

This version works correctly for every case and can potentially run almost as fast as the naive version. Section 6.8.1.6 discusses how the loop in Figure 29 can be optimized to run quickly by changing trapping status less often.

6.8.1.5.2. Presubstitution and Singularities

Presubstitution [60] is used to return values for functions known to behave poorly for certain input values, such as singularities. Presubstituted algorithms usually have the following general structure:

```
try
   {
      enable exceptions;
      Do the calculation using a simple algorithm
   }
catch (FloatingPointException e)
   {
      Return a pre-computed value
   }
```

That is, if the simple algorithm causes an exception because the method is ill-behaved for the given input, return (presubstitute) a value pre-computed for that input.

Consider the sinc function, which is used extensively in signal processing:

$$\operatorname{sinc} x = \frac{\sin x}{x}$$

This calculation is poorly behaved for $x = 0$ for this formulation of the sinc function since both $\sin x$ and x are 0, causing an invalid exception and generating a NaN. By l'Hopital's rule, this function has a removable singularity at $x = 0$:

$$\lim_{x \to 0} \frac{\sin x}{x} = \lim_{x \to 0} \frac{\sin' x}{x'} = \lim_{x \to 0} \frac{\cos x}{1} = 1$$

Therefore, 1 is presubstituted when $x = 0$. Implementing this using exceptions:

```
public static double sinc(double x)
try
   {
      enable invalid;
      return Math.sin(x) / x;
   }
catch (InvalidException e)  // only ZeroOverZeroException can be generated, InfinityOverInfinityException cannot occur since sin( ∞) is NaN
   {
      // Presubstitute one if division by zero occurs
      return 1.0;
   }
```

Figure 30 — Using exceptions to implement presubstitution.

For the sinc function it may be less expensive and easier to simply test for 0.0 explicitly.

```
public static double sinc(double x)
{
   If (x == 0.0)
      return 1.0
   else
      return = Math.sin(x) / x;
}
```

However, for algorithms that involve loops, the cost of explicit tests in the inner loops can be larger than the overhead for exceptions.

6.8.1.5.3. Floating point Exponent extension

Another use of floating point exceptions is to implement floating point numbers with extended exponents (and therefore greater range). One possible implementation of multiplying two `double` numbers with extended exponents is given in Figure 31. In this implementation, adapted from [40], the wide exponent `double` numbers are not normalized after every operation; for faster execution normalization only occurs when the `double` format overflows or underflows. This code resembles portions of the robust geometric mean algorithm in Figure 29.

```
static WideExpDouble multiply(WideExpDouble x, WideExpDouble y)
{
   WideExpDouble product = new WideExpDouble(0.0);

   try
      {
         enable overflow, underflow;
         // the exp fields are integers
         product.exp = x.exp + y.exp;              // add exponents when multiplying, assume integer overflow will not occur

         // the sig fields are double floating point numbers
         product.sig = x.sig * y.sig;              // multiply significands
      }
   catch(OverflowException e)
      {
         product.sig = e.doubleValue();            // use correct significand bits
         product.exp += Double.BIAS_ADJUST;  // and adjust exponent accordingly
      }
   catch(UnderflowException e)
      {
         product.sig = e.doubleValue();            // use correct significand bits
         product.exp –= Double.BIAS_ADJUST;  // and adjust exponent accordingly
      }
   return product;
}
```

Figure 31 — Using exceptions for `WideExpDouble` multiply. Adapted from [40].

6.8.1.6. Compiling Floating Point Exceptions

As with most exceptions, floating point exceptions are assumed to be relatively rare events, so the common non-exceptional case should be optimized to run quickly. While simple code generation techniques can implement the discussed exception behavior, the speed of the resulting code may be unacceptably slow. On current architectures, installing trap handlers and changing the trapping status involves non-negligible costs. A Borneo environment needs to install its own trap handlers, which in turn throw the desired Borneo exceptions. The Borneo handlers can be installed once at the start of the program or, more generally, before any floating point operation that may throw exceptions. After the handlers are installed, the trapping status of the five conditions can be set independently. Changing trapping status is likely to cause the floating point pipeline to be flushed. Therefore, loops which implicitly or explicitly change the trapping status of a condition may run considerably slower than expected. While rather inelegant, the KOUNT mode described in [56] avoids the performance overheads associated with changing trapping status.

```
static final double ln2 = Math.log(2.0);
double geometricMean(double array[])
{
    //Assume proper trap handlers have already been installed.
    double product = 1.0;
    int exp_adjust = 0;          // assume exp_adjust will not experience integer overflow

    // check for zero length array
    if(array.length == 0)
        return NaNd;

    for (int i = 0; i < array.length; i++)
        {
            double element = array[i];

            // Check for illegal non-positive array elements
            if (element <= 0.0)
                return NaNd;

            // First try finding the product using naive multiplication.
            try
                {
                    try
                        {
                            // enable overflow, underflow;
                            setTrap(Math.OVERFLOW_FLAG | Math.UNDERFLOW_FLAG);
                            product *= element;
                        }
                    finally
                        {
                            //restore non-trapping mode
                            setTrap(Math.NONE);
                        }
                }

            // If an overflow or underflow exception occurs, grab the scaled value of the product,
            // and increment or decrement the exponent adjustment.  Then continue the calculation as before.
            catch (OverflowException e)
                {
                    product = e.doubleValue();
                    exp_adjust += Double.BIAS_ADJUST;
                }
            catch (UnderflowException e)
                {
                    product = e.doubleValue();
                    exp_adjust –= Double.BIAS_ADJUST;
                }
        }

    // Calculate the nth root of the product, using either the pow method or the formula in  Figure 27.
    return (exp_adjust == 0)
            ? Math.pow(product, 1.0/(double)array.length);
            : Math.exp((Math.log(product) + (double)exp_adjust * ln2)/ (double)array.length);
}
```

Figure 32 — Example Java code with `native` methods implementing Borneo program from Figure 29.

Figure 32 shows a straightforward translation of the geometric mean code from Figure 29 desugared into Java with calls to `native` methods manipulating the trapping status. However, the code in Figure 32 changes trapping status twice per iteration; the overhead of altering the floating point state dwarfs the time spent on computation. A more sophisticated compilation of the loop from Figure 29 is shown in Figure 33. Using information about the particular exceptions floating point operations can throw, the trap setting operations have been moved outside the loop, allowing the code to run at nearly full speed in the common non-exceptional case.

```
// move setTrap() outside of loop
try
  {
    setTrap (Math.OVERFLOW_FLAG | Math.UNDERFLOW_FLAG);
    // loop body before the try block cannot throw an overflow or underflow exception

    for (int i = 0; i < array.length; i++)
      {
        double element = array[i];

        // Check for illegal non-positive array elements
        if (element <= 0.0)
          return NaNd;

        // First try finding the product using naive multiplication.
        try
          {
            // enable overflow, underflow;
            product *= element;
          }

        // If an overflow or underflow exception occurs, grab the scaled value of the product,
        // and increment or decrement the exponent adjustment.  Then continue the calculation as b  efore.

        // The code in the catch clauses does not generate any floating point exceptions
        catch (OverflowException e)
          {
            product = e.doubleValue();
            exp_adjust += Double.BIAS_ADJUST;
          }
        catch (UnderflowException e)
          {
            product = e.doubleValue();
            exp_adjust –= Double.BIAS_ADJUST;
          }
        }
  }
finally
  {
    // The code in the catch blocks cannot throw floating point exceptions either so clearing the trapping status can be pulled outside of loop
    setTrap (Math.NONE);
  }
```

Figure 33 — Inner loop from Figure 32 transformed to minimize changing trapping status.

Since floating point exceptions are detected by hardware, no explicit exception-checking instructions are needed in the virtual machine. A degenerate (but portable) implementation of floating point exceptions could save and test the flags around each floating point operation.[26] While this implementation does not require writing a trap handler, it introduces the explicit checking that using exceptions is meant to avoid.

6.8.2. Sticky Flags

The sticky flags work in partnership with the default IEEE non-trapping mode for handling exceptional conditions. The flags are an alternate and sometimes less costly mechanism for handling floating point exceptions. Flags record that an exceptional condition occurred sometime during an aggregate computation. The flag state can be inspected after the computation is finished in contrast to exceptions, which interrupt a computation while it is running. Within a basic block, dependency-preserving permutations of floating point operations do not alter the flags raised by executing the block. Such code rearrangements are not necessarily legal with precise exception handling of floating point exceptions. Therefore, using flags instead of exceptions allows for better code scheduling. (However, a single sticky flag status register precludes speculatively executing floating point operations, such as hoisting a

[26] To implement trapping on underflow, testing the underflow flag is not sufficient since underflow is signaled differently under trapping and non-trapping modes. An underflow trap would occur whenever the result is subnormal.

loop-invariant floating point division outside of a loop.) With judicious flag use, codes can effectively run at "full speed" for usual inputs and special cases can be recognized and dealt with at the end of the computation if exceptional conditions are encountered.

6.8.2.1. Specification

Augmented Java Syntax
ConstructorDeclaration:
    Modifiers$_{opt}$ ConstructorDeclarator Throws$_{opt}$ Admits$_{opt}$ Yields$_{opt}$ ConstructorBody

MethodHeader:
    Modifiers$_{opt}$ Type MethodDeclarator Throws$_{opt}$ Admits$_{opt}$ Yields$_{opt}$
    Modifiers$_{opt}$ `void` *MethodDeclarator Throws$_{opt}$ Admits$_{opt}$ Yields$_{opt}$*

StatementWithoutTrailingSubstatement:
    FlagStatement

New Borneo Productions
FlagStatement:
    `flag` *Admits$_{opt}$ Yields$_{opt}$ Block Waves$_{opt}$*

Waves:
    WaveClause
    Waves WaveClause

WaveClause:
    `waved` *TrappingConditions Admits$_{opt}$ Yields$_{opt}$ Block*

Admits:
    `admits` *TrappingConditions*

Yields:
    `yields` *TrappingConditions*

TrappingConditions:
    TrappingCondition
    TrappingConditions `,` *TrappingCondition*

TrappingCondition: one of
    `overflow underflow divideByZero invalid inexact all none`

Figure 34 — Changes to Java grammar to support IEEE 754 sticky flags.

Java includes in a method declaration the checked exceptions a method may throw. These exceptions are part of the method's contract. Similarly, the sticky flags that a method sets and the sticky flags examined from the caller's environment are also part of a method's contract. Therefore, Borneo adds this information to method and constructor declarations; the `admits` list specifies which of the caller's flags the called method may inspect and the `yields` list specifies which flags a method may modify. For example, the declaration `admits overflow, underflow` means a method can test whether its call occurred when the overflow or underflow flags were set. The declaration `yields invalid` means the caller can determine if invalid was raised when the callee returned. If a method does not specify either `admits` or `yields`, the defaults are `admits all` and `yields all`. These defaults allow for easy inlining and these defaults imply methods that do not use floating point do not have to save or restore the sticky flag state. Blocks inside methods see and return the entire flag state. At least in the initial version of Borneo, a method overridden in a subclass must have the same flag signature as the method in the parent class. In the future, this could be relaxed to require the `admits` list in the overriding method in the child class to be a superset of the `admits` list of the method being overridden (covariance) and the `yields` list be subset (contravariance). If an interface inherits more than one method with the same type signature, the `admits` and `yields` clauses of the inherited methods must be the same. The flag effects of class initialization expressions are not visible to methods in the executing program.

Table 10 lists the actions that need to be taken for a flag under each possible combination of `admits` and `yields` settings for that flag. Unless a flag is both `admit`ted and `yield`ed, the initial value of the flag must be saved either to be restored upon method exit or to be merged with the flag status as a side effect of executing the method. An example desugaring is given in Figure 35. Table 11 lists the operational meaning of various combinations of `admits` and `yields` specifiers. Declaring `yields none` may allow more aggressive constant folding and code motion inside a method. Any method not dependent on the flag state of its caller can declare `admits none`.

Table 10 — Actions needed for a sticky flag *f* under different `admits`/`yields` settings.

`admit` status for *f*	`yield` status for *f*	On method entry	On method exit
do not admit	do not yield	save current value of *f*	restore saved value of *f*
do not admit	yield	save current value of *f*	OR current value of *f* with saved value
admit	do not yield	save current value of *f*	restore saved value of *f*
admit	yield	do nothing	do nothing

Table 11 — Operational meanings of various flag signatures.

Flag signature	Operational meaning
`admits all,  yields all`	callee sees caller's flag state, caller's flag state reflects execution of callee
`admits all,  yields none`	callee sees caller's flag state, caller's flag state preserved across call
`admits none, yields all`	callee gets clean flag state, caller's flag state reflects execution of callee
`admits none, yields none`	callee gets clean flag state, callers state preserved across call

Borneo Code

```
void only_flags()
    admits overflow, underflow yields invalid
{
    Calculation
}
```

Equivalent Java code with `native` methods

```
void only_flags()
{
    int callers_flags;
    //get caller's current flags
    caller_flags = getFlags();
    //mask out unwanted flags
    setFlags(   caller_flags &
                ~(Math.OVERFLOW_FLAG | Math.UNDERFLOW_FLAG));
    try
        {
            Calculation
        }
    finally
        {
            setFlags(   (getFlags() & Math.INVALID_FLAG) |   // isolate yielded flag
                        callers_flags);                      // ... and merge with caller's
        }
}
```

Figure 35 — Example desugaring of Borneo flag manipulation into Java with `native` methods.

To conveniently control flag state inside a method, the new `flag-waved` statement is used. The `flag-waved` statement has a similar structure to the `try-catch` statement; each `flag` clause can be followed by zero or more `waved` clauses. The `flag` code block is executed until completion, then the first `waved` clause with a trapping condition matching a set flag is executed. It is a compile time error for a single trapping condition to appear implicitly or explicitly in more than one `waved` clause for a given `flag` statement. The `flag` and `waved` clauses can be modified with `admits` and `yields` specifiers in the same manner as methods.

Explicit `getFlags` and `setFlags` methods can be used to sense and manipulate an integer representation of the flag state. Individual flags may be set using a `setFlag` method which takes two arguments, the flags to set and a boolean value. In addition to the flag manipulation methods, the Borneo `Math` class also defines integer constants with bit patterns representing each flag position. For example, bitwise ANDing the results of `getFlags` with `Math.OVERFLOW_FLAG` isolates the overflow sticky bit.

6.8.2.2. Examples Using Sticky Flags

Figure 36 contains several algorithms that compute the vector 2-norm. Initially, a naive, fast approach is used; upon completion the flags are consulted. If an underflow or relevant overflow has occurred during the naive algorithm, the slower robust algorithm recomputes the norm. (As the computation progresses it is unknown whether or not encountering underflowed values matters.) This approach is preferable to always using the robust algorithm since the naive code runs quickly in the common case. Overflow and underflow exceptions could also be used, but halting the naive algorithm on all underflow exceptions would lead to some unnecessary uses of the slow algorithm.

```
double norm(int n, double x[])
    admits none                  // do not want previous flag state.
    yields overflow, underflow  // expose underflow and overflow when deserved.
{
    int i;
    double sum, scale_factor, adjustment=1.0;

    // Attempt to compute the dot product x • x
    // Will work in the vast majority of cases
    flag
        {
            sum = 0.0;
            scale_factor = 1.0
            for (i=0; i<n; i++)
                sum += x[i] * x[i];
        }
    // Check for overflow, underflow
    waved overflow
        {
            double scaled_element;
            // Repeat scaling down
            Math.setFlag(Math.OVERFLOW_FLAG, false);  // lower flag
            scale_factor = Math.scalb(1.0, –640);
            adjustment = 1.0 / scale_factor;
            sum = 0.0;
            for (i=0; i<n; i++)
                {
                    scaled_element = scale_factor * x[i];
                    sum += scaled_element * scaled_element;
                }
        }

    waved underflow
        {
            if sum < Math.scalb(1.0, –970))
                {
                    double scaled_element;
                    // Repeat scaling up
                    scale_factor = Math.scalb(1.0, 1022);
                    adjustment = 1.0f / scale_factor;
                    sum = 0;
                    for (i=0; i<n; i++)
                        {
                            scaled_element = scale_factor * x[i];
                            sum += scaled_element * scaled_element;
                        }
                    Math.setFlag(Math.UNDERFLOW_FLAG, 0);        // lower flag
                }
        }
    return adjustment * sqrt(sum);                  // may overflow or underflow, as deserved
}
```

Figure 36 — Code to calculate two-norm of vector using sticky flags to improve average performance.

Some of the geometric mean examples using exceptions can also be written in terms of flags as shown in Figure 37. Unlike the corresponding code in Figure 26, the loop in Figure 37 runs over the entire array if an overflow or underflow occurs in the middle of the calculation.

```
double geometricMean(double array[])
{
    double product = 1.0;

    // check for zero length array
    if(array.length == 0)
        return NaNd;

    // first try the simple algorithm..
    flag
        {
            // Iterate over the array elements
            for (int i = 0; i < array.length; i++)
                {
                    double element = array[i];

                    // Check for illegal non-positive array elements
                    if (element <= 0.0)
                        return NaNd;

                    // Multiply the array elements together
                    product *= element;
                }

            // Return the nth root of the product
            return Math.pow(product, 1.0/(double)array.length);
        }
    // if the simple algorithm doesn't work, try a more expensive one
    waved underflow, overflow
        {
            //call sophisticated program...
        }
}
```

Figure 37 — Geometric mean algorithm using flags.

The relative speed of flags and traps to handle exceptional floating point conditions depends on a number of factors. Flag-based algorithms usually test for exceptional conditions periodically. Between flag checks, computation can occur on NaNs, infinities, and subnormals. Some processors calculate more slowly on these special values than regular floating point numbers. Throwing floating point exceptions can stop a computation once a special value is encountered, possibly improving the worst case performance, but code scheduling with precise exceptions may be less aggressive than with flags, slowing down the common non-exceptional case.

6.9. Operator Overloading

From childhood people are trained to use infix operators when dealing with numerical expressions. While prefix functions do not lack expressiveness, infix operators are more familiar. Operators are also more succinct than the equivalent expression coded with methods. For an extreme example, compare the two lines of the inner loop of Figure 38, the actual code written in Java without operator overloading and the much shorter equivalent expression with operator overloading. The method `ComputeSturmPolynomials` in Figure 38 computes the Sturm sequence [90] of a polynomial. Sturm sequences are used to count the number of real roots of a polynomial *P* which lie in a given interval. The sequence of polynomials resembles:

$$Q_0 = Q$$
$$Q_1 = Q'$$
$$Q_k = -\left(Q_{k-2} \bmod Q_{k-1}\right)$$

where Q is $P / \gcd(P, P')$ and only has simple roots. Each polynomial in the sequence must be computed exactly with no loss of precision; in practice, arbitrary precision rational numbers are used to guarantee this condition. `ComputeSturmPolynomials` computes the sequence using Java's arbitrary precision `BigInteger` class to ensure accuracy.

Operator overloading also improves code reuse. The code that operates on built-in integers can, with minor editing (potentially as little as changing a few declarations), operate on non-basic types such as `BigInteger`. For example, a programmer might want to use high precision floating point numbers with Heron's formula (see Table 6) to lessen rounding problems when calculating the area of a triangle. If operator overloading is available, the same textual expressions can be used for built-in and user-defined types, such as high precision floating point. Otherwise, stark differences can result, as shown by the two representations of Heron's equations in Figure 39.

Operator overloading reduces the differences between the behavior of primitive numeric types and reference types. On the downside, operator overloading does increase the complexity of the language, but the increases in usability for the programmer outweigh the added compiler and language complications. At least for numeric types, operator overloading can also improve readability. As with any language feature, operator overloading can be misused, but that does not imply reasonable operator overloading is not worthwhile. Borneo's operator overloaded is designed to support new numeric types and Borneo avoids certain excesses of previous operator overloading schemes.

```
/**
 * Poly == polynomial with BigInteger coefficients
 */
class Poly {
    private int size;
    private BigInteger coeff[];

    public Poly(int maxSize);
    public BigInteger GetAt(int n);
    public void SetAt(int n, BigInteger val);
    public void Set(Poly other);
    public int GetDegree();
};

/**
 * Given a polynomial, computes the sequence of Sturm polynomials.
 * Arbitrary precision rationals or arbitrary precision integers
 * are required to compute the sequence accurately, hence
 * polynomials with BigInteger coefficients are used.
 */
void ComputeSturmPolynomials(Poly first[], Poly p[])
{
    int i, j, k, n, n2;

    // Initialize sequence
    p[0] = first;
    for (i=0; i<n-1; i++)
        p[1].SetAt(i, first.GetAt(i+1).multiply(i+1));

    for (i=2; i<first.GetDegree(); i++) {
        // poly[i] = rem( poly[i-2] / poly[i-1] )
        n2 = p[i-1].GetDegree();
        // p[i] = p[i-2]
        p[i].Set(p[i-2]);
        while (p[i].GetDegree() >= n2) {
            n = p[i].GetDegree();
            for (j=n2-1; j>=0; j--) {
                k = j+(n-n2);
                // with operator overloading
                // p[i][k] = p[i-2][k] * p[i-1][n2] - p[i-1][j] * p[i-2][n];

                // without operator overloading
                p[i].SetAt(k, p[i].GetAt(k).multiply(p[i-1].GetAt(n2)).subtract(p[i-1].GetAt(j).multiply(p[i].GetAt(n)))));
            }
            p[i].SetAt(n, 0);
        }
    }
}
```

Figure 38 — Code using `BigIntegers` that would benefit from operator overloading.

Original code with operators	Equivalent code without operators
s = ((a + b) + c)/2;	s = ((a.add(b)).add(c)).divide(2);
return sqrt(s * (s - a)*(s - b)*(s - c));	return (((s.multiply(s.subtract(a))).multiply(s.subtract(b))).multiply(s.subtract(c))).sqrt();

Figure 39 — Equivalent code with and without operator overloading.

6.9.1. Operator Overloading and Value classes

There are two kinds of types in Java, reference types (classes) and primitive types (integer and floating point types along with `boolean`). For new numeric types, it is desirable to have semantics analogous to the primitive types; however, in Java all user-defined classes are reference types. To address this discrepancy, Borneo has a third kind of type, a `value` class type (similar to a proposal by Bill Joy [54]), which has a mixture of the properties of reference and primitive types.

When primitive types are assigned to one another, passed as arguments to a method, or returned from a method, the value of one variable is copied into another. Afterwards, there is no further sharing of state between the variables. In contrast, the implementation of reference types copies pointers to objects on assignment, parameter passing, and method return. Therefore, if two references point to the same object and the object is modified by access through one reference, the other reference sees the changes to the object. Equality comparison also differs between reference and primitive types: reference types are checked for pointer equality, not equal values of the fields of the objects being pointed to (this latter notion of equality is often provided by a class's `equals` method). Casts behave differently on the two kinds of types: casts between reference types do not actually generate a new object while casts between primitive types create a new value. Since Borneo adds operator overloading to create new numeric classes, the value semantics of primitive types are appropriate for user-defined numeric classes.

Including operator overloading in a class should change the semantics of assignment, parameter passing, and method return, as well as casting and comparison. For efficiency reasons, in classes that use operator overloading Borneo only changes the semantics of assignment, comparison, and casting, not parameter passing or method return. Using pass by reference for parameter passing and method return avoids overhead in copying objects. A disciplined programmer can avoid aliasing errors stemming from reference semantics for parameter passing and method return (see the `Complex` example in section 6.9.6.1).

Java does not currently have declarations to enforce call-by-value semantics with a call-by-reference implementation. In C++, there are reference parameters to `const` objects; the reference points to the same object throughout the lifetime of the method and the object pointed to is not modified. (From an implementation perspective, a C++ reference to a `const` object is a `const` pointer to a `const` object.) A method obeying these restrictions has value semantics for that reference parameter. Java 1.1 only has partial support for such declarations; Java 1.1 allows parameters to be declared `final`. A `final` reference is analogous to a `const` pointer (but not analogous to a `const` pointer to a `const` object). In Java, all the fields of a class may be declared `final`, meaning an object of that type cannot be modified after it is created. However, this restriction applies to all objects of that type; it cannot be declared on a per object basis. One possible benefit to using `value` classes is that (with some analysis) objects can be allocated on the stack instead of the heap.[27] Many common numeric types, such as complex numbers, are small objects; heap allocation overhead may be the main cost of using these types. Copying large structures, such as matrices, is also prohibitively expensive.

Borneo has a new keyword `value` that acts as a class modifier; only classes declared to be `value` classes can use operator overloading. `Value` classes can also call and declare traditional instance and `static` methods. Fields of `value` class objects can be accessed in the usual manner. All types in Java have a default value; for reference types, the default value is `null`. Therefore, to allow `value` classes to behave like primitive types, before any other operation is performed on a `value` class variable, the variable is initialized by calling the compiler-generated no-arg constructor for that class. For arrays of `value` class variables, the no-arg constructor is called for each element. Additionally, it is a compile-time error for `null` to be assigned to a `value` class variable. These requirements imply that a `value` class variable always points to an object before the variable can be used.

The operators acting on the primitive types have various relationships programmers can depend on. For example, `a+=` *`Expression`* is semantically equivalent to `a = a +` *`Expression`*. Maintaining such relationships among `value` class operators is the responsibility of the programmer. The existence of one operator does not cause the Borneo compiler to infer the existence of related operators. For example, defining a `>=` operator does not cause `<` to be inferred as `!(a >= b)` nor is `+=` inferred from `+` and `=`. The return type of operators is not constrained. For example, an overloaded comparison operator can return a non-boolean type. Programmers are encouraged to implement `value` classes having the expected semantics for their operators.

Borneo lets programmers overload most existing Java operators while also allowing novel user-defined operators, such as `**` for exponentiation, to be declared and overloaded. Borneo aims to avoid past mistakes and unnecessary complications stemming from operator overloading while providing a sufficiently flexible language feature.

6.9.2. Overloading existing operators

In the definition of a `value` class, an operator is declared by declaring a method named "`op`" followed by the text of the operator being overloaded (the exact grammatical changes for Borneo operator overloading are given in section 9.6). For example, "`op*`" is the name of an binary operator with the precedence and associativity of multiplication. All operators that can be overloaded are either unary or binary operators (except for a special ternary

[27] If an object cannot be referred to from variables outside of scope in which the object is created, the object can be stack allocated.

subscripting-assignment operator explained in section 6.9.2.4). Operator methods have the same modifiers and optional clauses as ordinary Borneo methods (`throws` clauses, `admits` and `yields`, etc.). Table 13 lists the Java operators that can be overloaded in Borneo. The precedence, arity, and associativity of built-in operators cannot be changed via operator overloading. Both unary and binary `+` and `-` can be overloaded in a given `value` class. A unary operator must be an instance method taking no arguments. In general, a binary operator may either be an instance method taking one argument or a `static` method taking two arguments. If a binary operator has a left hand operand of the type of the defining class, the operator must be a one-argument instance method instead of a two-argument `static` method. When an existing operator is overloaded as a `static` method, at least one of the parameters must be a user-defined type and the second parameter must be the type of the class defining the operator. Therefore, the meanings of existing operators on primitive types cannot be hidden by operator overloading.

In addition to the infix notation, operators may also be called by explicit dispatch using the method name of the operator. For example, in the following code, the call to `op+` and to the infix `+` are equivalent.[28]

```
Complex b, c; double d;
b = c.op+(d);
b = c + d;          // equivalent to explicit dispatch above
```

6.9.2.1. Operators as `static` methods

Many object oriented languages, including C++ and Java, support calling methods by dispatching on an object; the object is passed as a special implicit parameter to the method and the type of this object determines what set of methods can possibly be called. This presents an orthogonality problem for operator overloading and primitive types. It is easy to create a `Complex +` operator that takes a `Complex` number as the left operand and a `double` as the right operand. Borneo operators are typically desugared into method dispatch; however, the `Complex` class cannot declare an instance method that takes a `double` as the left hand operand. There is not even a `double` class that can be changed to interoperate with `Complex`. Therefore, to allow expressions such as `double_d + Complex_c`, the `Complex` class needs to be able to define `static` operators such as `static Complex op+(double d, Complex c)`. Allowing the declaration of `static` operators also allows `value` classes to declare operators that act on previously existing `value` classes.[29]

If a class declares a `static` operator (which must have two parameters), the second parameter (corresponding to the right operand) must have the type of the class declaring the operator. Operators declared to be `static` have different scoping rules than other `static` methods. For a binary operator with two `value` class arguments, there are two ways a callable method can be defined: an instance method in the class of the left operand or as a `static` method in the class of the right operand. To resolve such a binary operator call, both possibilities are considered; it is a compile-time error if an ambiguity exists between such methods. Ordinary classes cannot declare operators; therefore, to allow reference types to appear as the left operand, the `value` class needs to declare a `static` operator. Although the scoping of `static` operators is different than for other `static` methods, the policy to resolve which operator method to call uses the same criteria as normal `static` method resolution. In particular, the return type of an operator is not taken into account when deciding which version of an operator to call. The implementation options for various operator overloading tasks are summarized in Table 12.

---

[28] The expression "`c + d`" cannot appear alone as a statement in Java. However, a method invocation followed by a semi-colon does constitute a statement. Therefore,

```
c + d;       //expression cannot be a statement
```

is an error, while

```
c.op+(d);  //method invocation can be a statement
```

is not.

[29] Some language, such as CLOS, support *multi-methods* where arguments other than the first (leftmost) can be used to find an appropriate method to call.

Table 12 — Operator overloading options in Borneo. *U* and *V* are value classes.

Operator overloading task	Implementation options
Overload an existing unary operator	
on a primitive or reference type	*illegal*
on a value class *V*	declare the operator as instance method in *V*, e.g. `V op-(){…}  // unary negation`
Overload an existing binary operator	
both parameters are reference or primitive types	*illegal*
left operand is of type *V* and the right operand has a primitive or reference type, e.g. $+: V \times$ `double` $\rightarrow V$	declare the operator as an instance method in *V* with one parameter, e.g. `V op+(double d){…} // binary addition`
left operand has a primitive or reference type and the right operand is of type *V*, e.g. $+:$`double` $\times V \rightarrow V$	declare `static` method in *V* with two parameters, e.g. `static V op+(double d, V v){…}`
left operand is of type *V* and the right operand is of type *U* e.g. $+: V \times U \rightarrow V$	declare an instance method in *V* with one parameter, e.g. `V op+(U u) {…}` or declare a `static` method with two parameters in class *U*, e.g. `static V op+(V v, U u) {…}` (compile time error if both possibilities are defined)

Table 13 — Existing unary and binary Java operators and whether or not they can be overloaded for `value` classes in Borneo.

Operator	Can be overloaded	Must always be an instance method	Java Functionality (b=boolean, i=integer, f=floating point, n=numeric, r=reference)
[]	yes	yes	subscripting
`.`	no	—	member access
`(`*`params`*`)`	no	—	method call
`expr``++`, *`expr`*`--`	no	—	postfix increment/decrement • $n \rightarrow n$
`++`*`expr`*, `--`*`expr`*	no	—	prefix increment/decrement • $n \rightarrow n$
`new(type)`*`expr`*	no	—	create new object
`~` (unary)	yes	yes	bitwise complement • $i \rightarrow i$
`!` (unary)	no	—	logical negation • $b \rightarrow b$
`+`, `-` (unary)	yes	yes	indicate signs • $n \rightarrow n$
`*`, `/`, `%`	yes	no	multiplicative • $n \times n \rightarrow n$
`+`, `-`	yes	no	additive • $n \times n \rightarrow n$
`<<`, `>>` , `>>>`	yes	no	shift • $i \times i \rightarrow i$
`instanceof`	no	—	dynamic type test • $r \rightarrow b$
`<`, `>`, `>=`, `<=`	yes	no	relational • $n \times n \rightarrow b$
`==`, `!=`	yes	no	equality • $n \times n \rightarrow b$ • $b \times b \rightarrow b$ • $r \times r \rightarrow b$
`&`	yes	no	bitwise AND • $i \times i \rightarrow i$ • $b \times b \rightarrow b$
`^`	yes	no	bitwise XOR • $i \times i \rightarrow i$ • $b \times b \rightarrow b$
`|`	yes	no	bitwise OR • $i \times i \rightarrow i$ • $b \times b \rightarrow b$
`&&`	no	—	logical AND • $b \times b \rightarrow b$
`||`	no	—	logical OR • $b \times b \rightarrow b$
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `>>>=`, `&=`, `^=`, `|=`	yes	yes	assignment • $n \times n \rightarrow n$ • $b \times b \rightarrow b$ • $r \times r \rightarrow r$

6.9.2.2. Restrictions on `value` Classes

`Value` classes cannot implicitly or explicitly extend any other class, including another `value` class, an `abstract` class, or `Object`. Constructors for `value` classes may not explicitly invoke `super`. `Value` classes are implicitly `final`; a `value` class can be explicitly declared `final` as well. `Value` classes cannot implement interfaces. It is a compile-time error to give the `instanceof` operator an expression having the type of a `value` class. It is also a compile time error to try to use `instanceof` to test for a `value` class type. The methods defined for `Object` can be dispatched from a `value` class object. There are no legal coercions between `value` classes and reference types. Unlike for reference types, a programmer cannot prevent the instantiations of a `value` class by declaring all constructors to be `private`. For `value` classes, the Borneo compiler always generates a default constructor that cannot be overridden.

`Value` classes have `Class` objects created for them. The `getSuperclass` method returns `null` for a `value` class.

6.9.2.3. Equality and assignment

A `value` class does not have the usual definitions of assignment (`=`) and equality (`==`) operations on that type:

- `=` defaults to calling the assignment operator on each field in the order the fields are declared and returning `this`.
- `==` defaults to calling the comparison operator on each field in the order the fields are declared and returning the `boolean` value of ANDing together the results of the memberwise comparisons. The comparisons are short-circuiting. `!=` is defined as the negation of `==`.

If these definitions are not appropriate, the assignment and equality operators can be redefined by declaring `op=` and `op==` methods, respectively. The compiler generated assignment and equality methods may not be legal. For example, a `value` class can overload `==` to return a type other than `boolean`.

Assignment must be an instance method but equality can be either an instance method or `static` method (subject to the restrictions in section 6.9.2.1). Assignment and comparison can be overloaded to implement interaction with primitive types. For example, a `Complex` class can have a method like

```
Complex op=(double d)
{
   this.r = d;
   this.i = 0.0;
}
```

to support statements such as:

```
Complex c;
c = 3.0;
```

The assignment is desugared into

```
c.op=(3.0);
```

Since `Complex` is a `value` class, `c` is initialed before `op=` is invoked, avoiding dispatching on a `null` pointer. Explicit constructors can still be invoked for `value` classes. Figure 40 gives a valid desugaring for an apparent initialization of a `Complex` object by a constructor.

Borneo Code with Operators	**Equivalent Java code**
Complex c = new Complex(2.0, 1.0);	Complex c = new Complex(); *// call default constructor, assignment is pointer assignment* *//operator name mangling to create a legal Java method name* c.op$3d(new Complex(2.0, 1.0)); *// copy newly created object into c*

Figure 40 — Initialization and assignment of Borneo value classes.

6.9.2.4. Subscripting

Retrieving a value from an array using the subscripting operator and assigning into an array location using the subscripting operator are two different operations having separate overloadable operators in Borneo. The subscripting operator used to retrieve values, `[]`, is a binary operator that must be defined as a one-argument instance method. The index can be of any type. The subscripting operator can be used to support array-like access, for example, `v[i]` where `v` is a `SparseVector`. However, the result of this subscripting operator cannot appear as the left hand side of an assignment statement since (unlike C++) Java cannot return references to primitive values and therefore cannot modify the array location. To allow `[]` to be syntactically used both to retrieve a value from a data structure and to change the data structure, Borneo has a special ternary `[]=` operator. The operator `[]=` is an instance method taking two arguments, the first argument is the index and the second argument is the expression used for the assignment. This ordering of parameters ensures the proper order of evaluation. In a Borneo program, whitespace, but not parentheses, can separate the right bracket, "`]`", and the "`=`". In Java `a[0] = a[1]` and `(a[0]) = a[1]` are equivalent, but Borneo does not recognize the latter idiom as being an instance of the `[]=` operator. If a value class defines `[]`, `[]=` is not defined by the compiler. If `[]=` is not defined, separate `[]` and assignment methods are called. Figure 41 gives a sample definition of the subscripting operators and Figure 42 shows a desugaring of both kinds of subscripting operator into Java method calls.

```
/**
 * MyArray provides arrays whose first index is 1 instead of 0
 */
value class MyArray
{
  private double data[]; // array of doubles

  // constructors, other methods…
  public double op[](int index)
  {
    return data[index–1];
  }

  public double op[]=(int index, double value)
  {
    return data[index–1] = value;
  }
}
```

Figure 41 — Sample usage of the subscripting operators.

Borneo	**Equivalent Java Code**
MyArray a;	MyArray a = new MyArray(); *//default constructor*
…	…
a[i] = a[i–1] + 4;	*// name mangling for legal Java methods names*
	a.op$5b$5d$3d(i, a.op$5b$5d(i–1) + 4)

Figure 42 — Desugaring of both kinds of subscripting operators.

6.9.2.5. Rationale

Not all existing operators can be overloaded in Borneo. As shown in Table 13, most Java operators that act on numeric values can be overloaded in Borneo, but operators that only act on `boolean` values cannot be overloaded. Operators with short-circuit evaluation, such as "&&" and "`||`" cannot be overloaded since they have different evaluation rules than all other operators. Since constructors are already overloaded, overloading `new` is not required. Some C++ uses for overloading `new`, such as controlling the details of memory allocation, are not relevant in Java due to language features such as garbage collection. Similarly, overloading "`.`", the member access operator, seems undesirable. Using overloaded "`()`" to create iterators, as suggested in [89], can be useful but is not necessary since iterators can be created conveniently using the *inner classes* added to Java 1.1 (although the current version of Borneo is based on Java 1.0). Overloading the compound assignment operators such as "`+=`" allows the programmer to save the creation and copying of some temporary objects. Avoiding unnecessary temporaries and copying can be important if the numeric objects involved are large objects, such as matrices. Although the present version of Borneo only allows `value` classes to overload operators, as long as overloading `=` and `==` was prohibited, reference types could overload operators as well.

To avoid some ad hoc solution to differentiating *`expr`*`++` and `++`*`expr`*, neither the prefix nor postfix version of `++` or `--` can be overloaded in Borneo. C++ programs often overload prefix and postfix `++` to implement "smart pointer" classes; however, such data types are not needed in Java due to garbage collection and the lack of explicit pointer types. Moreover, the Java definition of ++ on floating point types is not as meaningful as possible. For integers, instead of adding 1, `++` can be thought of as the successor function, giving the next larger integer value. Likewise, for floating point types, `++` could have been defined as the successor function, giving the next larger floating point number (this functionality is given by `nextAfter(`*`number`*`, infinity)`). For moderately large values, adding `1.0` to a floating point number does not result in a different value. Once a floating point number is larger than 2 raised to the number of bits in the significand, adding `1.0` is lost during rounding. For example, the `float` type ranges over approximately $\pm 10^{38}$ while adding `1.0` is lost to rounding on numbers with magnitudes larger than approximately 10^{6}. A related problem exists for numbers smaller than the rounding threshold ($\approx 10^{-8}$ for `float`); the original number is rounded away when added to `1.0`.

Borneo operator overloading has similarities to a proposal made by Bill Joy [54] and to the operator overloading mechanism in the Titanium language [47], [103].

6.9.3. Casting

Methods to cast a `value` class to another type are declared as unary operators. The name of a casting operator is "`op`" followed by a space and the name of the target type; for example, `double op double(){return r}` in a `Complex` class. It a compile time error for the declared return type and the type name in a cast operator to be different. Otherwise, casting operators are treated similarly to other unary operators except that they cannot be explicitly dispatched. Borneo does not implicitly call user-defined casting operators to perform type conversion during parameter passing or in other contexts.

6.9.4. Overloading novel operators

In addition to overloading existing operators, Borneo programmers can define novel operators not built into the language. Mathematical notation often uses operators other than `+`,`-`, `*`, and `/`, for example, transpose and inverse on matrices. Numerical analysts are acquainted with numerous such matrix operators available in packages like Matlab. Novel operators can be defined similarly to other overloading operators with the additional capability that novel operators can have all their operands be primitive types. All novel operators are either unary or binary.

6.9.4.1. Syntax of novel operators

To simplify compiler construction and to reduce the information a programmer must remember, Borneo conveys the precedence and associativity of a novel operator via the characters constituting the operator. In Borneo, if the characters forming a novel operator have as a prefix the characters of a built-in operator, the novel operator has the same precedence and associatively as the existing operator. For example, `op**` can be defined as a left associative exponentiation operator with the same precedence as multiplication. Since the parser must already distinguish unary and binary `+` and `-`, both unary and binary operators can have "`+`" or "`-`"as a starting character. New unary operators can only start with "`+`", "`-`", or "`~`" since those characters start existing unary operators. All operators starting with other characters are binary. The characters [`'@`] that do not start any existing operator can also be used to create novel binary operators; all such operators are left associative and have the same precedence as addition. As a result, a given textual operator has the same precedence and associativity in all Borneo programs.

Table 16 gives regular expressions specifying the novel operators that can be defined; the allowed operators avoid some possible syntactic trouble with adding new operators. Table 14 summarizes the syntactic restrictions on novel operators. The underscore character "`_`" can appear in both identifiers and operators. This introduces a small syntactic discrepancy between Java and Borneo. The underscore character cannot start an operator, but it can start an identifier and terminate an operator. Therefore, in a Borneo program "`a+_b`" is treated as `a +_ b` while in a Java program "`a+_b`" is parsed as `a + _b` where "`_b`" is a variable name. This difference should rarely be visible since user-named identifiers starting with an underscore should be uncommon.

Table 14 — Syntactic restrictions on novel operators.

Operator Restriction	Rationale
multi-character operators cannot have the characters “!”, “+”, “-”, or “~” appearing after the first character.	Minimizes importance of whitespace in tokenizing a program, for example “a+^~b” can be interpreted as “a +^ (~b)” without any white space separating “^” and “~”. Prevents a unary or binary operator from having the text of another unary operator as a suffix.
“/” cannot appear after the first character, “/*” cannot be a prefix of an operator	avoid creating operators that conflict with comment syntax
do not allow “#” in operators	do not thwart attempts at using the C preprocessor with Java programs (“#” and “##” are preprocessor operators in ANSI C)
do not allow “$” in operators	follows the Java standard’s suggestion that the dollar sign character “should be used only in mechanically generated Java code or, rarely, to access preexisting names on legacy systems” (JLS §3.8).

Table 15 — Definitions used to ease defining operator syntax in Table 16.

<table>
<tr><th>Character Class</th><th>Definition</th></tr>
<tr><td>OP_CHAR</td><td>[‘@%^&*_|<>?]</td></tr>
<tr><td>INIT_OP_CHAR</td><td>[‘@]</td></tr>
</table>

Table 16 — Flex style regular expression for novel operators. The notation {OP_CHAR – ‘*’} means the set difference of the characters represented by {OP_CHAR} and the character ‘*’.

<table>
<tr><th>Base Operator</th><th>Regular expression</th></tr>
<tr><td>bitwise complement (unary)</td><td>‘~’{OP_CHAR}+</td></tr>
<tr><td>addition (binary and unary)</td><td>‘+’{OP_CHAR}+</td></tr>
<tr><td>subtraction (binary and unary)</td><td>‘-’{OP_CHAR}+</td></tr>
<tr><td>multiplication</td><td>‘*’{OP_CHAR}+</td></tr>
<tr><td>division</td><td>‘/’{OP_CHAR -‘*’}{OP_CHAR}*</td></tr>
<tr><td>remainder</td><td>‘%’{OP_CHAR}+</td></tr>
<tr><td>shifting</td><td>“<<”{OP_CHAR}+ |
“>>”{OP_CHAR - ‘>’}{OP_CHAR}*</td></tr>
<tr><td>comparison</td><td>‘<’{OP_CHAR - ‘<’}{OP_CHAR}* |
‘>’{OP_CHAR - ‘>’}{OP_CHAR}* |
“<=”{OP_CHAR}+ |
“>=”{OP_CHAR}+</td></tr>
<tr><td>bitwise AND</td><td>‘&’{OP_CHAR -‘&’}{OP_CHAR}*</td></tr>
<tr><td>bitwise XOR</td><td>‘^’{OP_CHAR}+</td></tr>
<tr><td>bitwise OR</td><td>‘|’{OP_CHAR -‘|’}{OP_CHAR}*</td></tr>
<tr><td>completely novel</td><td>{INIT_OP_CHAR}{OP_CHARS}*</td></tr>
</table>

6.9.4.2. Restrictions on novel operators

If at least one of the operands to a novel operator is of the type of the class defining the operator, overloading the novel operator has the same restrictions as overloading existing operators (unary operators must be defined as instance methods, etc.). Novel operators acting solely on primitive or reference types must be declared as `static` methods. Unlike `static` operators acting on `value` classes, `static` operators acting on only non-`value` types have same scoping rules as `static` methods with simple names. In other words, `static` operators on non-value classes can only be used in the defining class. As a special case, the `Math` class defines a number of `static` operators on primitive types; these operators are always available in all contexts. Novel operator methods can be explicitly dispatched.

6.9.5. Implementing Operator Overloading in JVM

Implementing Borneo `value` classes is possible using existing JVM instructions that act on references and objects. Borneo `value` classes can be implemented as classes that inherit directly from `Object`. It is illegal for a Borneo program to observe a `value` class variable as a `null` pointer. To enforce this invariant, the Borneo compiler creates a default no-argument constructor that initializes each field of the `value` class object to the default value for that type. This constructor is not dependent on any external program state. Therefore, `static value` class fields can be initialized before other `static` fields and `value` class local variables can be initialized at the beginning of the enclosing scope. These policies prevent the programmer from seeing an uninitialized `value` class variable. The initialization of `value` class variables can also be enforced via a source-to-source transformation into Java.

All textual assignment operations to `value` classes in a Borneo program are desugared into `op=` method calls. Objects in Java are pass by reference, giving the appropriate semantics to parameter passing and method return of `value` class variables.

All method calls on `value` class variables can be resolved at compile time since all `value` classes are `final`. Therefore, while some `static` operators have different scoping rules than other `static` methods, this difference can be hidden from the JVM. A Borneo compiler must perform a source-to-source transform of overloaded Borneo operators into legal Java method calls. Since Java names cannot include characters such as "`+`" and "`-`" some name mangling scheme is needed to encode Borneo operator names in Java. As shown in the examples in sections 6.9.2.3 and 6.9.2.4, Borneo uses the "`$`" symbol as an escape in front of the lower case hexadecimal ASCII code of each operator character. For example, "`op+^`" is represented as "`op$2b$54`".

6.9.6. Operator Overloading Examples

The following examples demonstrate a number of uses of Borneo operator overloading.

6.9.6.1. `Complex` and `Imaginary` Classes

As discussed in [63], having separate `Complex` and `Imaginary` types gives better numerical properties to complex arithmetic. The following code is a partial implementation of a `Complex` class; overloaded equality and assignment operators are shown as well as various addition operators. An `Imaginary value` class is defined similarly to the `Complex` class.

```
public value class Complex
{
private double r;     // real part
private double i;     // imaginary part

// Constructors

// Implicit no arg constructor initializes a Complex object to zero
/* public Complex()
    {
        r = 0.0;
        i = 0.0;
    }
*/

public Complex(double r, double i)
    {
        this.r = r;
        this.i = i;
    }

public Complex(double r)
    {
        this.r = r;
        this.i = 0.0;
    }
```

```
public Complex(Imaginary i)
    {
        this.r=0.0;
        this.i = i.imag();
    }

// Assignment and equality operators

public Complex op=(Complex c)
    {
        // update object in place, self-assignment is okay
        r = c.r;
        i = c.i;
        return this;
    }

// overload op= on double to interact with literals
public Complex op=(double d)
    {
        r = d;
        i = 0.0;
        return this;
    }

public Complex op=(Imaginary i)
    {
        this.r = 0.0;
        this.i = i.imag();
        return this;
    }

public boolean op==(Complex c)
    {
        return this.r == c.r && this.i == c.i;
    }

public boolean op==(double d)
    {
        return this.r == d && this.i == 0.0;
    }

public boolean op==(Imaginary i)
    {
        return this.r == 0.0 && this.i == i.imag();
    }

// Arithmetic operations

// unary + operator
public Complex op+()
    {
        return new Complex(this.r, this.i);
    }

// binary + operators

// +:Complex × Complex → Complex
public Complex op+(Complex c)
    {
        return new Complex(this.r + c.r, this.i + c.i);
    }

// +:Complex × double → Complex
public Complex op+(double d)
    {
        return new Complex(this.r + d, this.i);
    }
```

```
// +:double × Complex → Complex
public static Complex op+(double d, Complex c)
    {
        return new Complex(c.r + d, c.i);
    }

// +:Complex × Imaginary → Complex
public Complex op+(Imaginary i)
    {
        return new Complex(this.r, this.i + i.imag());
    }

// +:Imaginary × Complex → Complex
public static Complex op+(Imaginary i, Complex c)
    {
        return new Complex(c.r, c.i + i.imag());
    }

// Selected compound assignment operators
public Complex op+=(Complex c)
    {
        this.r += c.r;
        this.i += c.i;
        return this;
    }

public Complex op+=(double d)
    {
        this.r += d;
        return this;
    }

public Complex op+=(Imaginary i)
    {
        this.i += i.imag();
        return this;
    }

// Casts
public double op double()
{
    return r;
}

public Imaginary op Imaginary()
{
    return new Imaginary(i);
}

// Utility functions
public double real()
    {
        return r;
    }

public double imag()
    {
        return i;
    }

public String toString()
    {
        return "" + r + " + " + i + "i";
    }
}
```

6.9.6.2. Exponentiation

The Borneo `Math` class includes three exponentiation operators (one for each primitive floating point type) that call appropriate versions of the `pow` method. The exponentiation operator is written "`**`" as in FORTRAN. Since the leading character of "`**`" is "`*`", the exponentiation operator has the same precedence as multiplication and is left associative (not right associative as in customary in mathematical notation). Therefore `2.0 ** 3.0 ** 3.0` evaluates to 512.0 ($(2^3)^3$) instead of 134217728.0 (2^{3^3}).

6.9.6.3. Quiet Comparison Operators

One of four mutually exclusive relationships can hold when comparing two IEEE floating point numbers, the first may be greater than the second, they may be equal, the first may be less than the second, or the two numbers could be unordered. The unordered relation occurs when at least one of the arguments is a NaN. Therefore, in IEEE arithmetic, $a < b \neq !(a >= b)$ due to the unordered relation. The standard calls for quiet comparison operators that include the unordered relation. When the usual comparison operators other than `==` and `!=` act on a NaN, the standard requests the invalid flag be set. The operators that mention unordered are quiet; they do set the invalid flag on a NaN argument. Thus, while the logical value of the quiet operators is expressible with combinations of existing operators, the lack of setting the invalid flag cannot be expressed using current Java floating point operators.[30]

Table 17 — New quiet comparison operators.

Operator	Meaning
`<?`	less than or unordered
`<=?`	less than, equal to, or unordered
`>?`	greater than or unordered
`>=?`	greater than, equal to, or unordered

For example, "`a >= b`" following the IEEE 754 standard returns true if `a` is greater than `b` or if `a` is equal to `b` and signals invalid if either operand is a NaN. In contrast, "`a >=? b`" returns true if `a` is greater than `b`, if `a` is equal to `b`, or if `a` is unordered with respect to `b`. In addition, "`>=?`" does *not* signal invalid if any of its operands are NaN. While the logical value of "`>=?`" can be composed from other comparisons and the invalid flag can be discarded if raised, distinguishing such operators can allow better compilation since the total effect of "`>=?`" and related operators can be gotten from one machine instruction on many platforms. Table 17 catalogs Borneo's new comparison operators. Figure 43 give a Borneo implementation of "`>=?`". These operators are included in Borneo's `Math` class for all three floating point types.

An alternate proposal for the functionality of "`>=?`" is to use "`!<`" (not less than) instead [56]. In IEEE floating point, "greater than, equal to, or unordered" has the same logical value as "not less than." The first proposal is clearer since the operators' characters list when the relation returns true ("`>=?`", true if greater than, equal to, or unordered) as opposed to when the relation returns false ("`!<`", false if less than). Additionally, the "`>=?`" syntax is closer to the notation used in the standard document. While using questions marks in operators that act on NaN may imply to some that NaN is undefined, programmers using such comparison operators should be familiar enough with the details of the standard to differentiate undefined from unordered.

---

[30] In JVM, the current instructions for comparing floating point numbers return a code of –1, 0, or 1 depending on whether the first argument is less than, equal to, or greater than the second. Therefore, all floating point comparison operators must use the same instructions; there are no existing quiet instructions nor can existing separate strict equality instructions be redefined to not signal. Therefore, the Borneo Virtual Machine defines the current floating point comparison instructions to signal on NaN and adds new comparison instructions that are quiet. This means floating point equality in Borneo a program compiled down to BVM will use different opcodes than the program compiled with Java to JVM.

```
static final boolean op>=?(float a, float b)//all flag effects of evaluating the arguments will have taken place in the proper order
    admits none yields none
{
    // Not yielding any flags is a little too strong, if one of the operands is a signaling NaN,   the comparison
    // operator should signal.  However, since signaling NaNs are not included in Borneo and since
    // signaling NaNs are not created as the result of an arithmetic operation, this detail will be overlooked.
    // Since comparisons can only generate the invalid signal, yielding "none" as opposed to "overflow, underflow, inexact, divideByZero"
    // is correct.
    return a >= b || Math.unordered(a,b);
}
```

Figure 43 — Code for an unordered comparison operator.

6.9.6.4. Rounding Mode Operators

Certain algorithms, such as a straightforward interval arithmetic implementation, need to control the rounding mode at a fine granularity, potentially switching rounding modes at each floating point operation.[31] For such codes, using the `rounding` declarations is awkward and verbose. To alleviate this problem, rounding mode operators are added to the `Math` class. The operators are declared `final`, allowing them to be readily inlined. For example, the code in Figure 44 implements an addition operator on `float` arguments that rounds toward positive infinity. Table 18 lists the syntax of the new operators. The rounding mode operators also provide a convenient mechanism to specify the rounding mode of a few operations within a block of code influenced by a `rounding` declaration. Rounding versions of addition, subtraction, multiplication, and division are provided for all primitive floating point types.

```
static final float op+^(float a, float b)   // operands evaluated in proper order
{
    rounding Math.TO_POSITIVE_INFINITY;
    return a + b;
}
```

Figure 44 — Method body for a rounding mode operator.

Table 18 — Rounding mode operators for addition. Other rounding operators for subtraction, multiplication, and division are defined analogously.

Operator	Meaning
`a +@ b`	round toward 0
`a +^ b`	round toward +∞
`a +_ b`	round toward –∞
`a +% b`	round toward nearest

A consistent mnemonic scheme is needed to name the new rounding mode operators. The extra characters should imply the direction of rounding. The operators "`+/`" to round up, "`+\`" to round down, and "`+>`" to round to zero would have had a clear meaning, but are unworkable for various reasons.[32] The combination "`//`" conflicts with Java comment notation, "`\`" is involved with UTF escapes to encode Unicode characters, and "`->`" is the dereferencing operator in C. Instead, the operators in Table 18 are used. Caret "`^`" is near the top of the character vertical space implying "up," while underscore "`_`" is at the bottom implying "down." The at sign "`@`" resembles zero in a circle and "`%`" is meant to convey that the nearer of two choices is selected.

Figure 45 and Figure 46 show interval addition and multiplication written using the rounding mode operators.

---

[31] Interval arithmetic uses two rounding modes, round toward positive infinity and round toward negative infinity. The rounding mode changes in interval arithmetic can be eliminated by rewriting the expressions to use only one rounding mode, either to positive infinity or to negative infinity [82]. See section 6.12.4 for examples.

[32] The language FORTRAN-SC [97] uses "`+>`" to round toward positive infinity and "`+<`" to round toward negative infinity.

```
Interval add(Interval a, Interval b)
{
    double lower_bound, upper bound;

    lower_bound =   a.lower_bound +_ b.lower_bound;
    upper_bound =   a.upper_bound +^ b.upper_bound;

    return new Interval(lower_bound, upper_bound);
}
```

Figure 45 — Core computation for interval addition using rounding operators.

```
Interval multiply(Interval a, Interval b)
{
    double lower_bound, upper bound;

    lower_bound = min(   a.lower_bound *_ b.lower_bound,       a.lower_bound *_ b.upper_bound,
                         a.upper_bound *_ b.lower_bound,       a.upper_bound *_ b.upper_bound);

    upper_bound = max(   a.lower_bound *^ b.lower_bound,       a.lower_bound *^ b.upper_bound,
                         a.upper_bound *^ b.lower_bound,       a.upper_bound *^ b.upper_bound);

    return new Interval(lower_bound, upper_bound);
}
```

Figure 46 — Core computation for interval multiplication using rounding operators.

6.9.7. Interaction of Numeric Types

Java performs automatic widening promotions of numeric types in a number of contexts, including assignment statements and parameter passing. For example, a method taking a `double` parameter can be given a `float` argument and the `float` is automatically widened to `double`; an explicit cast is not needed. Figure 47 shows the Java numeric type width hierarchy.

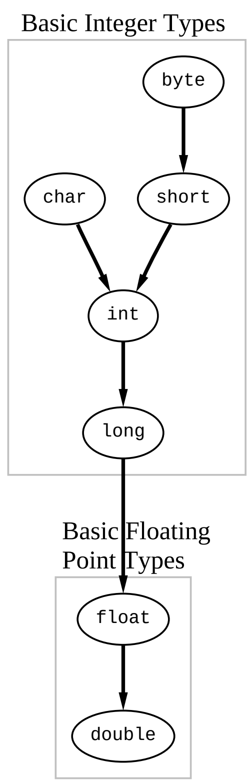


Figure 47 — Java numeric width hierarchy. Conversions for `indigenous` can easily be added.

`Value` classes are used to implement new numeric types. Figure 48 gives the desired numeric width relationships between primitive types and new Borneo numeric classes. Ideally, the Borneo compiler could perform the same sort of automatic conversions for `value` classes as performed for primitive types. An obvious candidate to encode the width information is to use the subtype relationship between classes, but this approach is not workable.

Borneo does not allow `value` classes to subclass one another. Putting that aside, the subtype relationship is not a suitable candidate for representing the width relationship between classes. If subtyping were used to guide compiler generated conversions, the automatic numeric widening conversions correspond to allowing a subclass to be used wherever the superclass can be used. Therefore, the widest numeric type must be at the top of the subtyping hierarchy. Java has single inheritance, implying a numeric type can only have one immediately wider type (since a class only has one immediate parent in the subtyping relationship). However, the width relationship in Figure 48 has several types that are immediately wider than another type. For example, `WideExpDouble` and `DoubledDouble` are both immediately wider than `double` (`double` in this instance can be thought of as a class instead of a primitive type). The constraints of the subtype relationship together with the desired width relationship imply a total ordering of all numeric types. Such a total ordering is not meaningful; which should be "wider" `WideExpDouble` with greater range or `DoubledDouble` with greater precision? How should an arbitrary precision class such as `Extended` compare to `Interval`? Since the existing subtype relationship is not suitable, some other mechanism is needed to implement useful conversions between `value` classes. Multiple inheritance could be used to better model the width relationship, but that would introduce great complexity into the language. A separate declaration could be used to indicate numeric width, but Borneo's existing operator overloading can implement the desired functionality.

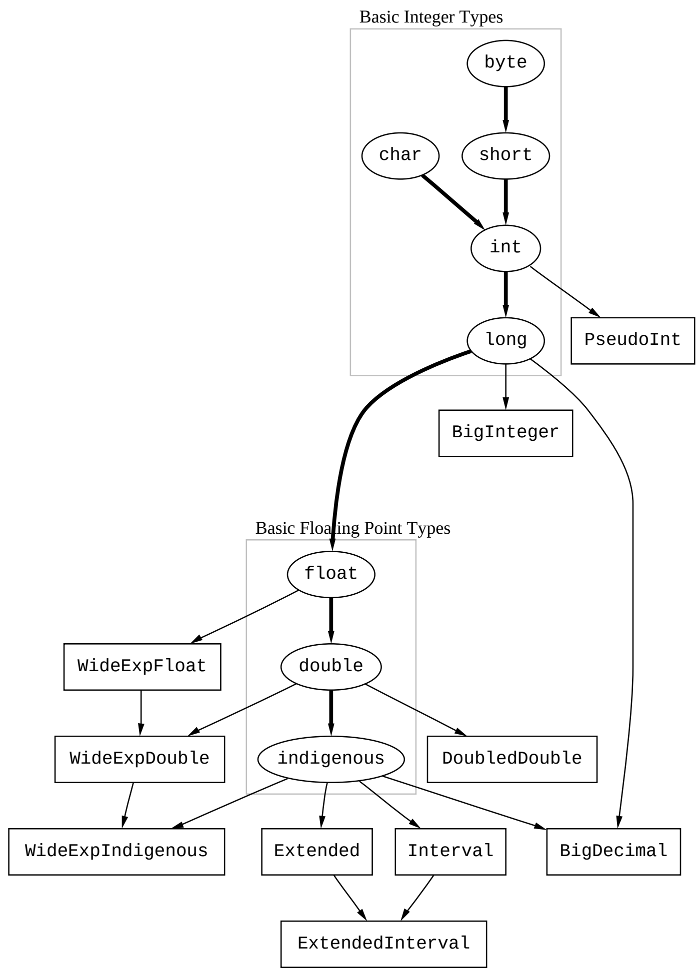


Figure 48— Desired width hierarchy for Borneo. Ovals represent primitive types and boxes represent value class. Bold arrows indicate existing widening conversion performed among primitive types. Conversions represented by narrow arrows need to be inferred or synthesized.

Overloaded operators and methods can provide to a user the same convenience as compiler-generated conversions (at the cost of more work for the class implementor). For example, the widening of `float` to `double` during addition can be implemented with the method signatures in Table 19. While the number of operators that need to be defined grows quadratically with the depth of the width hierarchy, Java's existing types form most of the levels of the width relationship in Figure 48. Therefore, the writer of a `value` class only has to provide methods interfacing with the top of the Borneo width hierarchy and with at most a few levels of other `value` classes.

Table 19 — Function signatures modeling Java arithmetic promotion rules (f = float, d = double).

Function Signatures	Operational Meaning
$+ : f \times f \rightarrow f$	$f + f \quad \rightarrow f$
$+ : f \times d \rightarrow d$	$(\text{double})\ f + d \quad \rightarrow d$
$+ : d \times f \quad \rightarrow d$	$d + (\text{double})\ f \quad \rightarrow d$
$+ : d \times d \quad \rightarrow d$	$d + d \quad \rightarrow d$

6.10. Anonymous Values

In the evaluation of numeric expressions of more than two terms, temporary values are needed to hold the intermediate results. Since these anonymous temporary values are not explicitly declared by the programmer, some convention is needed for determining the types of these locations. Borneo naturally extends the familiar Java rules to include the `indigenous` type. For primitive numeric types, if at least one of the operands of a numerical operation is of type `indigenous`, then the operation is carried out on `indigenous` values, and returns an `indigenous` result. If the other operand is not of type `indigenous`, it is first widened to `indigenous` via numeric promotion. Otherwise, the existing Java rules are used. However, to accommodate the x86, Borneo relaxes Java's requirement that intermediate results for `float` and `double` operands must be rounded to exactly `float` or `double` length. Borneo only requires that the significand bits be rounded to `float` or `double`.[33] To achieve exact `float` or `double` rounding, an explicit store of the intermediate result into a variable of the appropriate type can be performed. (Explicitly storing to temporaries still exhibits the double rounding problem on underflow discussed in section 6.1).

6.10.1. Widest Available and `anonymous` declarations

Augmented Java syntax
LocalVariableDeclarationStatement:
    AnonymousValueDeclaration `;`

New Borneo Productions
AnonymousValueDeclaration:
    `anonymous` *FloatingPointType*

Figure 49 — Modification to Java's grammar to support Borneo's `anonymous` declaration,

The convention Java and Borneo use to determine the width of anonymous values is called *strict evaluation*. Other rules for determining the size of anonymous values are preferable in some circumstances. One set of rules used in some older Fortran versions as well as in pre-ANSI C is *widest available*, where locations of type `indigenous` are used to store any intermediate results. Depending on the architecture the intermediate locations may be registers or actual memory locations.

The widest available strategy makes best use of the `double extended` registers on the x86 and can lead to more accurate results on all platforms. The programmer could certainly implement the widest available policy by casting all the operands to `indigenous`, but such programming is tedious and results in code difficult to read and maintain. To ease producing code employing the widest available policy, Borneo adds a new declaration `anonymous` *FloatingPointType* where *FloatingPointType* can be any of the primitive floating point types, `float`, `double`, or `indigenous`. An `anonymous` declaration is in effect until the end of the block or until the next `anonymous` declaration. As with `rounding` and `enable`/`disable` declarations, an `anonymous` declaration in the initialization of a `for` loop has influence over the rest of the loop construct.

All the intermediate results of floating point expressions in scope of an `anonymous` declaration have types at least as wide as the type in the `anonymous` declaration, not the type inferred from strict evaluation rules. If the type inferred from strict evaluation is wider than the type in an `anonymous` declaration, the strict evaluation rule is used instead (see Figure 50). Since `anonymous` declarations are intended to preserve precision, losing precision by implicitly narrowing operands is not supported. If all the operands in an expression are of the same type as the type given in an `anonymous` declaration, the declaration does not change the semantics of the expression. Since `anonymous` declarations cannot be used to implicitly narrow operands, `anonymous float` does not change the evaluation of floating point expressions. Therefore, `anonymous float` specifies to use strict evaluation. Such a declaration is useful when strict evaluation is desired in a portion of code under the influence of another `anonymous` declaration.

```
indigenous mac(indigenous a, indigenous b, indigenous c)
{
   anonymous double;
   return (a * b) + c;   //anonymous declaration ignored, type of operands wider than the anonymous type
}
```

Figure 50 — Ineffectual use of an `anonymous` declaration.

To preserve the precision of the result, instead of simply having the anonymous locations be of the (presumably) wider type, all the leaves of the expression tree are first converted to the target type before any

[33] The Limbo [20] programming language's only floating point type is a `double` precision `real` and Limbo only requires that the significand to be rounded to `double` precision.

arithmetic is performed. Figure 51 shows one possible desugaring of an expression under the influence of an `anonymous` declaration into equivalent Java code. Borneo's `anonymous` declarations alter Java's conventions concerning implicit narrowing conversions. In a block with an `anonymous` declaration, implicit narrowing between floating point types can occur when the type of the target of an assignment or return statement is no wider than the type given in the `anonymous` declaration. In other words, strict evaluation rules are used for floating point types wider than the type in an `anonymous` declaration. In Java, implicit narrowing only occurs across compound assignment statements. Implicit narrowing does not occur for method arguments since the types of a method's arguments select which method is called.[34] If a floating point variable or literal is given as an argument to a method, if the type of that argument is narrower than the type of an enclosing `anonymous` declaration, the argument is widened to the `anonymous` type. To avoid this behavior, an explicit cast can be used, see Figure 52 for examples.

Borneo Code[35]	**Equivalent Borneo Code**	**Equivalent Java code using explicit promotions**
static float mac(float a, float b, float c) { anonymous double; return a * b + c; }	static float mac(float a, float b, float c) { float f; anonymous double; f = a * b + c; return f; }	static float mac(float a, float b, float c) { float f; *//anonymous double* *//promote `float` values to `double`* f = (float)((double)a * (double)b + (double)c); return f; }

Figure 51 — Example desugaring of Borneo `anonymous` declarations into Java.

```
{
   float f=1.0f;
   double d = 2.0;
   indigenous n = 3.0n;

   anonymous double;
   Math.abs(f);                  //calls double abs(double)
   Math.abs(d);                  //calls double abs(double)
   Math.abs((float)f);           //calls float abs(float)
   Math.abs(((float)f));         //calls float abs(float)
   Math.abs(f + 0.0f);           //calls double abs(double)
   Math.abs((float)(f + 0.0f));  //calls float abs(float)
   Math.abs(n);                  //calls indigenous abs(indigenous), anonymous declaration has no effect
   Math.abs(n + 0.0);            //calls indigenous abs(indigenous), anonymous declaration has no effect
}
```

Figure 52 — Method resolution and `anonymous` declarations.

For an expression to have its types affected by an `anonymous` declaration, a built-in numeric operator does not have to be explicitly present or implicitly present through a compound assignment operator such as "`+=`". In particular, implicit narrowing can occur in simple assignments between variables of different widths. For example, for the assignment and return expressions in Figure 53 the compiler generates implicit narrowing conversions.

```
static float assign(float a, double b)
{
   anonymous double;
   a = b;      //implicit narrowing cast generated by the compiler, illegal without anonymous declaration
   return b;   //implicit narrowing cast generated by the compiler, illegal without anonymous declaration
}
```

Figure 53 — Implicit narrowing and the `anonymous` declaration.

---

[34] C++ can either widen or narrow numeric arguments during method resolution (widening is preferred to narrowing). However, Borneo does not aim to introduce all the complications of C++ into Java.

[35] The code in this example does not implement a `float` fused mac. While the result of multiplying two `float` numbers is exactly representable in a `double`, when the third `float` is added to the product and rounded back to `float`, a different answer can result than if the infinitely precise result was rounded to `float` [59].

Floating point to string conversions stemming from the "+" or "+=" operators are not affected by anonymous declarations. If an operator in a block with an anonymous declaration has floating point and integer operands, the integer operand is also promoted to the type specified by the anonymous declaration.

In the current version of Borneo, a user-defined numeric type cannot be used as the type for an anonymous declaration. In particular, there is no locution to conveniently express anonymous double extended to emulate the evaluation policy of machines where indigenous is double extended on machines where indigenous is double. Since only hardware supported formats can be indicated by the current anonymous declaration, the performance of code influenced by an anonymous declaration should not differ much from the same code without the anonymous declaration. However, once Borneo library's numeric types are better specified, the ability to declare "anonymous double extended" may be added. In general, if user-defined types were to be used for anonymous declarations, orthogonality requires automatic coercions from both primitive floating point types and "narrower" value class types. However, the current Borneo type system does not indicate the width of a numeric type, making generating automatic coercions problematic.

Although novel operators are added to the Math class, anonymous declarations have the same effect on expressions using novel operators as expressions using just built-in operators. The new operators added in the Math class have type signatures of the form $op{:}\alpha \times \alpha \rightarrow \alpha$, that is the operands and return value all have the same type. Therefore, the anonymous declarations will have the same effect on built-in and novel operators as long as the novel operators are provided for each precision and have homogeneous type signatures. Borneo's anonymous declarations are not without precedent; the language Numerical Turing [48] supports scoped precision declarations that can be varied at runtime among arbitrary decimal precisions.

The canonical example benefiting from the widest available evaluation is the dot product computation as shown in Figure 54. Using extra precision, even just double precision with float data, to accumulate the pairwise products eliminates intermediate overflow and underflow. Similarly, using double extended to accumulate a double dot product also removes the possibility of intermediate overflow or underflow. However, double extended does not have as much additional precision compared to double as double has to float.

Borneo Code

```
float dot(float[] a, float[] b)
{
    indigenous sum = 0.0n;

    if(a.length == b.length)
        {
            for(int i = 0; i < a.length; i++)
                {
                    anonymous indigenous;
                    sum += a[i] * b[i];
                }
            return sum;
        }
    else
        return NaNf;
}
```

Equivalent Borneo Code without an anonymous declaration

```
float dot(float[] a, float[] b)
{
    indigenous sum = 0.0n;

    if(a.length == b.length)
        {
            for(int i = 0; i < a.length; i++)
                {
                    // anonymous indigenous;
                    sum += (indigenous)a[i] * (indigenous)b[i];
                }
            return (float) sum;
        }
    else
        return NaNf;
}
```

Figure 54 — Dot product computation using anonymous declaration.

Heron's formula from section 6.7.1 also benefits from being evaluated with wider formats for the intermediate results. Table 20 (repeating some data from Table 6) shows that while Heron's formula with float input is unstable when evaluated using strict evaluation, Heron's formula is stable and correct when evaluated using anonymous double and rounding the final answer back to float.

Table 20 — Heron's formula evaluated using strict and widest available expression evaluation polices.

Rounding Mode	**Heron's Formula** $s = ((a+b)+c)/2$ $\sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$ (strict evaluation in `float` precision unstable)	**Heron's Formula** (`anonymous double` evaluation with `float` data stable)
a=12345679, *b*=12345678, *c*=1.01233995 > *a* – *b*		
to nearest	0.00	972730.06
to +∞	17459428.00	972730.06
to -∞	0.00	972730.00
to 0	-0.00	972730.00
a=12345679, *b*=1234567**9**, *c*=1.01233995 > *a* – *b*		
to nearest	12345680.00	6249012.00
to +∞	12345680.00	6249012.50
to -∞	0.00	6249012.00
to 0	0.00	6249012.00

6.10.2. Scan for widest

In addition to strict evaluation and widest available, a third expression evaluation policy, scan for widest, is advocated by some for giving better numeric results [24]. In scan for widest the expression (and the destination in case of an assignment) is scanned to find the widest numeric type used. Then, all the leaves of the expression are widened to the widest type in the entire expression. The intention of scan for widest is to relieve programmers from finding loss of precision bugs in their numeric codes.

Like the widest available policy, scan for widest is not needed for expressibility; the programmer can insert the necessary coercions. Scan for widest is primarily useful when many precisions running at varying speeds are combined; if only a single precision is used, scan for widest gives the same results as strict evaluation. While a limited version of scan for widest was added to a FORTRAN compiler [24], the type systems of Java and Borneo are very different from that of FORTRAN. Since Borneo allows new user-defined numeric types, for orthogonality, scan for widest should be used for both primitive and user-defined types, or not all. Scan for widest also complicates method resolution even when only primitive types are used. Due to these issues, investigating adding scan for widest to Borneo is left as future work. However, a user-defined class, such as arbitrary precision floating point numbers, can implement its own scan for widest policy by building up a data structure and delaying evaluation until assignment occurs. This technique is similar to *expression templates* in C++.

6.11. Threads

In addition to the state contained within each Java thread, a Borneo thread also contains the rounding mode, sticky flags, and enabled exceptions status. In a thread context switch, these values are saved as part of the outgoing thread's state. The incoming thread's values are then installed. New threads do not inherit the floating point state from another thread. A new thread starts in the default floating point state (flags cleared, round to nearest, non-trapping mode). When a thread dies, its floating point state is not merged with the any other thread; in particular, the sticky flag state is not merged with another thread's.

Saving and restoring the floating point state maintains the thread's integrity across context switches. If this thread state is not preserved, consistency problems can arise. For example, if one thread enabled floating point exceptions and then was context-switched out, the incoming thread might perform floating point operations expecting exceptions to be disabled. Since the new thread would not expect any exceptions to be thrown, it would not be written to perform exception recovery. Rounding mode, sticky flags, and trapping status *must* be changed when the context switch takes place.

6.12. Optimizations

> Java implementations must respect the order of evaluation as indicated explicitly by parentheses and implicitly by operator precedence. An implementation may not take advantage of algebraic identities such as the associative law to rewrite expressions into a more convenient computational order unless it can be proven that the replacement expression is equivalent in value and in its observable side effects, even in the presence of multiple threads of execution (using the thread execution model in §17), for all possible computation values that might be involved. [JLS §15.6.3]

Borneo extends Java's existing requirement of equivalent value and observable side effects to include IEEE 754 floating point state and floating point exceptions. Therefore, some optimizations legal in Java are forbidden in Borneo. Java's source code precise exception handling, well specified order of evaluation, and respecting of parentheses are all preserved in Borneo. Both JVM and native code generation issues are considered in the following discussion.

6.12.1. Preserving IEEE Floating Point semantics

Table 38 in Appendix 9.1 demonstrates that very few of the field axioms hold for floating point arithmetic. From an optimization standpoint, commutativity of addition and multiplication, and eliminating multiplying by `1.0` are the only useful field axioms valid for Borneo floating point arithmetic. As detailed in [94], even seemingly benign transformation such as replacing `x * 0.0` by zero and replacing `x - 0.0` by `x` are not correct under IEEE 754 arithmetic.[36] However, IEEE 754 arithmetic enjoys many useful properties not shared by other floating point arithmetics. For example, on IEEE machines multiplying by `1.0` never causes an overflow, unlike multiplication under Cray arithmetic.[37]

6.12.2. Order of evaluation

Java mandates left to right evaluation of operands, left to right evaluation of parameter lists, and that operands must be evaluated before an operation begins. In Java, these rules do not totally inhibit reordering floating point operations since no Java floating point operations throw exceptions and the flag state cannot be inspected. Although Borneo allows inspection of the flag state, this has minimal effect on code scheduling for non-trapping code. The flag effects of floating point operations are commutative and associative; as long as floating point operations are not moved across accesses to the flag state, any dependency-preserving ordering can be used. If floating point exceptions can be thrown, more care must be taken during code scheduling or additional work must be done by the trap handler and `catch` clause to modify the state so that it appears that a precise exception occurred. For Java, [17] recommends adapting the debugging optimized code techniques from [44] to the problem of maintaining the appearance of precise exceptions. If the sticky flag state must be maintained, floating point operations cannot be executed speculatively. For example, hoisting a division with loop invariant arguments outside of a loop cannot be performed; if the loop never executed, the division could raise spurious flags. However, this problem can be averted by peeling the loop instead of performing code hoisting. In loop peeling, one iteration of the loop is executed unoptimized and subsequent interations reuse shared values.

6.12.3. Constant Folding , Common Subexpression Elimination, and Dead Store Elimination

Constant folding is the evaluation of constant expressions at compile time instead of runtime. By default, without additional analysis, constant folding `float` and `double` values cannot be done inside a Borneo method due to possible changes to the method's flag effects or exceptions thrown. If floating point exceptions are enabled, constant folding must also preserve the appropriate semantics. If a floating point operation does not set any flags, including the inexact flag, that operation can be folded while preserving Borneo semantics. If the inexact flag is not raised, an operation has the same value under all rounding modes (except for `x – x` = `±0.0` where the sign of zero depends on the rounding mode). However, constant folding cannot be done on `indigenous` values since the size of

---

[36] `x * 0.0` is not `0.0` when `x` is a NaN. The expression `x - 0.0` does not have the same sign as `x` if `x` is `+0.0`.

[37] On a Cray, the floating point number `1.0` is represented as $0.5\cdot 2^1$. During a multiply, the exponents of the two factors are added and the resulting exponent is checked for overflow *before normalization*. Therefore, for all the floating point numbers with the maximum exponent, multiplying by `1.0` will overflow since the maximum exponent range will be exceeded before normalization would adjust the exponent back into representable range.

`indigenous` is not known until runtime.[38] If the inexact flag set by a floating point operation can be proved not to affect the outcome of the computation, some additional constant folding can be performed. However, if an operation sets the inexact flag, the runtime rounding mode must be a constant.[39] Since the flag effects of initializing `static` class variables are not seen by rest of the program, constant folding (under round to nearest) can be performed on those expressions.

Common subexpression elimination must be aware of the rounding mode and trapping status in effect when an expression is calculated; "common" subexpressions must be calculated under identical rounding modes and trapping status.

The standard allows copying a signaling NaN to signal invalid, implying that without further analysis dead stores cannot be eliminated. However, since Borneo does not include the semantics of signaling NaNs, such transformations can still take place.

6.12.4. Rounding Modes

Knowing the rounding mode statically at compile time enables the compiler to perform specialized optimizations to eliminate dynamic rounding mode changes. When compiling to native code, the Alpha architecture can specify in each floating point instruction which rounding mode to use. Therefore, when the rounding mode is known statically, the costs of changing rounding mode dynamically can be eliminated.[40] More generally, in a given section of code, if all operations round to either $\pm\infty$ (as is the case in interval arithmetic), changing rounding mode more than once can be avoided entirely by rewriting the expressions to use only a single rounding mode via the identities in Table 21 [82]. For a given machine, the compiler can determine if the cost of the extra negations outweighs the costs of changing rounding modes. Figure 55 and Figure 56 show the interval arithmetic skeletons from section 6.7.5.2 rewritten and optimized to run under a single rounding mode. If trapping on overflow or underflow, these identities cannot be used without additional analysis since a different value would be reported to the trap handler (and therefore a different value would be returned if a *`type`*`Value` method of the exception were called).

Table 21 — Identities relating arithmetic when rounding to ±infinity.

a +^ b	≡ –(–a +_ –b)	≡ –(–a –_ b)
a –^ b	≡ –(–a –_ –b)	≡ –(–a +_ b)
a *^ b	≡ –(–a *_ b)	≡ –(a *_ –b)
a /^ b	≡ –(–a /_ b)	≡ –(a /_ –b)
a +_ b	≡ –(–a +^ –b)	≡ –(–a –^ b)
a –_ b	≡ –(–a –^ –b)	≡ –(–a +^ b)
a *_ b	≡ –(–a *^ b)	≡ –(a *^ –b)
a /_ b	≡ –(–a /^ b)	≡ –(a /^ –b)

[38] If all the `indigenous` values fit without loss of information into `double` and the operations are also exact, constant folding can be done on `indigenous` values.

[39] Some additional constant folding may be possible if the set of possible rounding modes is known. For example, for positive results, rounding to zero and rounding to negative infinity give the same answer. For negative results, rounding to zero and rounding to positive infinity are equivalent.

[40] The Alpha devotes two bits in arithmetic opcodes to encoding the four possible rounding modes. However, since the standard calls for fully dynamic rounding modes, one of the four bit patterns is used to indicate the rounding mode is taken from the floating point control register. Therefore, one of the rounding modes (round to positive infinity) cannot be encoded in the instruction and can only be accessed by setting the dynamic rounding mode appropriately. But, all computations using round toward positive infinity can also be performed under round toward negative infinity at the cost of selectively negating operands and results.

As explained in section 6.7.3, the Alpha's static round to nearest encoding cannot be used by default in Borneo.

```
Interval add(Interval a, Interval b)
{
    double lower_bound, upper_bound;

    rounding Math.TO_NEGATIVE_INFINITY;
    lower_bound = a.lower_bound + b.lower_bound;

    // use only a single rounding mode
    upper_bound = –(–a.upper_bound – b.upper _bound);

    return new Interval(lower_bound, upper_bound);
}
```

Figure 55 — Core computation for interval addition using only one rounding mode (adapted from Figure 17).

```
Interval multiply(Interval a, Interval b)
{
    double lower_bound, upper_bound;

    rounding Math.TO_NEGATIVE_INFINITY;
    lower_bound = min(   a.lower_bound * b.lower_bound,    a.lower_bound * b.upper_bound,
                         a.upper_bound * b.lower_bound,    a.upper_bound * b.upper_bound);

    // use only a single rounding mode, move negations outside of reduction and switch from max to min
    upper_bound = –min( –a.lower_bound * b.lower_bound,   –a.lower_bound * b.upper_bound,
                        –a.upper_bound * b.lower_bound,   –a.upper_bound * b.upper_bound);

    return new Interval(lower_bound, upper_bound);
}
```

Figure 56 — Core computation for interval multiplication using only one rounding mode (adapted from Figure 18).

6.13. Compiler Implementation Issues

An initial Borneo implementation can be built on top of a modified Java infrastructure using `native` methods to access a processor's IEEE 754 floating point features. Optimizations legal in Java must be inhibited, otherwise a JVM or JIT could violate Borneo semantics. Better code could be generated if a Borneo specific bytecode were available as a target instead of JVM. For example, `native` methods can be used to implement operations on the `indigenous` type in current JVM systems, but only with a possibly significant performance penalty; `indigenous` opcodes could easily avoid this overhead. Additionally, a Borneo compiler is more naturally written in Borneo. For example, checking the inexact flag after an operation to aid constant folding is readily done in Borneo but only done slowly and with difficulty in Java (such as by simulating the floating point operations in integer arithmetic). Having a new extension for Borneo programs, "`.born`", avoids name clashes with existing Java programs. Files with a "`.java`" extension compiled under Borneo are compiled with Borneo's floating point semantics but do not have access to `rounding` declarations, operator overloading, and other Borneo features using new syntax.

6.13.1. Keywords

Borneo adds many new character sequences that are reserved by the compiler, either as keywords such as `rounding`, `enable`, and `disable`, or as floating point literals such as `NaNf`, and `infinityD` (see section 9.5). While the floating point literals probably would not usually conflict with identifiers in existing Java code, these new keywords are only available in Borneo programs..

6.13.2. Operator Overloading

The new operators are designed not to cause lexing and parsing difficulties, but operators tend to be only a few characters long so a simple typo could easily choose an unintended operator, potentially complicating syntactic error recovery. Allowing "`_`" in operator names introduces a minor syntactic disparity between Java and Borneo; variable names starting with underscores may be parsed differently by a Borneo compiler than a Java compiler. Operator methods have many rules and restrictions ordinary methods do not, complicating the language implementation. The semantics of `value` classes are somewhat contradictory, inviting certain kinds of programming errors.

6.13.3. Type System

In Borneo, certain `static` operators have different scoping rules than other methods, changing the set of possible methods that can be called. However, once all the candidate methods are identified, the existing Java method overloading resolution rules are used.

The `admits` and `yields` attributes are new to the type system and overriding methods must preserve the flag signature of the overridden method.

The `anonymous` declarations introduce additional circumstances where the Java compiler must generate implicit narrowing casts. The method resolution rules are also changed for certain situations involving `anonymous` declarations.

6.13.4. Borneo and Java Compilation differences

While a Java program cannot determine if another Java program has been compiled under Borneo floating point semantics, Borneo programs can in general determine if Java's more permissive floating point semantics have been utilized. For example, due to less aggressive constant folding, a Java program compiled under Borneo may set more sticky flags than the same program compiled under Java semantics. Therefore, compiling the same Java source program under Java and Borneo semantics can result in differing `class` files.

6.13.4.1. Floating point optimizations

As described in section 6.12.3, certain optimization legal in Java, such as floating point constant folding and common subexpression elimination, are applicable in fewer situations in Borneo.

6.13.4.2. Floating point equality

Borneo defines the floating point equality and inequality operations (`==` and `!=`) to not signal invalid on a NaN input. All other floating point comparisons in Borneo do signal invalid on a NaN input. Neither Java nor JVM include the IEEE 754 sticky flags and JVM uses a pair of instructions to implement all floating point comparisons. Since a Borneo program differentiates between floating point equality and other comparisons, different instructions must be used (or the flag state must be saved and restored around some floating point comparisons).

When Java source code is not available for recompilation to use Borneo comparison semantics, a `class` file converter could recognize the idioms for floating point `!=` and `==` and generate a new `class` file with the appropriate semantics. If a Borneo bytecode were available as a target, new quiet comparison instructions could be used. Otherwise, the comparison idiom has to be embedded in an instruction sequence that stores the current trapping status, turns off trapping on invalid, saves the current value of the invalid flag, performs the comparison, clears the invalid flag, restores the status of the invalid flag, and restores the previous trapping status.

6.13.4.3. Class initialization

To implement Borneo semantics, the class initialization process must save and restore the floating point state around initialization of a newly loaded class. This must be done so that the flag effects of initializing `static` fields are not visible to the rest of the program. To accomplish this, the `<clinit>` method ([66] §3.8) generated by a Borneo compiler must save and restore the flag state (`admits none yields none`). The `<clinit>` method must also always run under round to nearest. The class initialization process must initialize `value` class objects before they can be otherwise used.

7. Borneo Virtual Machine Specification

Nothing endures but change.
—Heraclitus

> The Java Virtual Machine is the cornerstone of Sun's Java programming language. It is the component of the Java technology responsible for Java's cross-platform delivery, the small size of its compiled code, and Java's ability to protect users from malicious programs.
>
> The Java Virtual Machine is an abstract computing machine. Like a real computing machine, it has an instruction set and uses various memory areas. [66]

The semantics of JVM, the Java Virtual Machine, are strongly coupled to those of Java. JVM is primarily intended as a portable intermediate format for Java. Since Borneo semantics differ from Java semantics, JVM might not be an ideal intermediate format for Borneo. Assuming a JVM implementation is running on a fully IEEE 754 compliant processor, existing Java compilers along with `native` methods to access IEEE 754 features should be able to implement much of Borneo's new functionality. Operations on `indigenous` values could be implemented as `native` method calls. While that would be rather expensive in an interpreter, an optimizing JIT (Just In Time) compiler could eliminate much of the overhead if the `indigenous` methods were recognized as deserving special treatment, such as aggressive inlining. A JIT could also recognize `native` methods that changed the rounding mode, sensed the sticky flags, and changed the trapping status as portions of the program that could affect other floating point operations. These changes would amount to adding IEEE 754 semantics to JVM, but the most natural manner to add IEEE 754 semantics in JVM is to add instructions implementing the relevant functionality.

> Attracted by a generally available, machine-independent platform, implementors of other languages are turning to the Java Virtual Machine as a delivery vehicle for their languages. In the future, we will consider bounded extensions to the Java Virtual Machine to provide better support for other languages. [66]

The JVM specification states future expansion is possible to better support other languages. Moving from Java 1.0 to Java 1.1 and onwards to Java 1.2 entails supporting new Java API classes that require changes to JVM implementations; a Java 1.0 interpreter cannot run all Java 1.1 programs. For example, Java 1.1 adds reflection, the ability to inspect what classes are currently loaded and extract information about the fields and methods of those classes.[41] Clearly a Java 1.0 interpreter/JIT would not necessarily be able to support such capabilities. A meaningful implementation of the *weak references* in Java 1.2 mandates changes to the garbage collection process.[42] While Java 1.2 supports these modifications without adding new opcodes, the implementation of the Java runtime must change. Introduced in Java 1.1, the Java Native Interface (JNI) for calling `native` methods also requires changes to a JVM environment.[43] Since the JVM already uses IEEE 754 encoding of floating point numbers, adding full IEEE 754 support is another reasonable extension.

It would be possible to support Borneo's new features by adding a "magic" class having methods that interface with the IEEE features of the underlying processor. Instead a new bytecode, the Borneo bytecode, is proposed to better integrate IEEE 754 features into the JVM. An implementation of the Borneo bytecode is referred to as a Borneo Virtual Machine, BVM. BVM offers a proper superset of the functionality of the JVM. The new bytecode must continue to support all the design goals supported by the existing JVM bytecode, including verification, compactness, and efficiency.[44] However, BVM is not just intended for Borneo: other languages

---

[41] Intended uses of reflection include writing debuggers and class browsers.

[42] Weak references (or weak pointers) are pointers that do not prevent an object from being garbage collected; if only weak pointers point to an object, the object can be deallocated. Weak pointers are useful for a variety of tasks. For example, weak pointers can be used to maintain a list of all objects representing files so the files' buffers can be flushed periodically [99]. Weak pointers are also used to implement the *hash consing* optimization for functional languages. If a new cons cell would have contents identical to an existing cell, a hash consing system returns a pointer to the existing cell instead of allocating a new cell. A hash table with weak pointers is used to find an existing cell with the given value [5].

[43] Microsoft's omission of JNI (among other Java 1.1 features) in its supposedly Java 1.1 compliant product Internet Explorer 4.0 prompted a lawsuit from Sun.

[44] This document will not actually provide a mapping of new instructions to new numeric opcodes; instead, several options for adding the instructions are offered.

wishing to use IEEE 754 features should be able to target BVM as well. Therefore, in some ways BVM has a more permissive floating point semantics than Borneo; for example, trapping status is not a static property in BVM although it is in Borneo.

7.1. `indigenous` Floating Point Type

The type `indigenous` is a new floating point type corresponding to the widest floating point type implemented with direct hardware execution on a particular processor. On many machines `indigenous` is the same as `double`, but on the x86 and 68000 series of processors, `indigenous` is the 80 bit `double extended` format. Each of the primitive floating point types currently in Java (`float` and `double`) is supported by its own set of typed bytecode instructions. Since `indigenous` is being added as a new basic type, it must also have its own set of typed instructions.

New comparison, stack manipulation, memory access, format conversion, and arithmetic operation instructions are needed to support `indigenous`. Table 22 through Table 26 enumerate the new opcode mnemonics and their stack effects, modeled after the format given in the JVM specification [66]. These new opcodes are analogous to the current instructions that support `float` and `double` and they should map easily to the appropriate underlying hardware instructions. In JVM, there are six comparison instructions for each integer type (equal, not equal, less than, less than or equal, etc.). However, each floating point type only has two comparison instructions (the instructions differ in their handling of NaN). A JVM floating point comparison returns –1, 0, or 1 depending on if the first operand is less than, equal to, or greater than the other. BVM has additional non-signaling comparison instructions. All BVM arithmetic instructions, except for remainder, follow the IEEE 754 standard.

By definition, the actual size of `indigenous` values is platform dependent. On platforms on which `indigenous` is equivalent to `double`, two words of stack space suffice to hold an `indigenous` value.[45] On platforms where `indigenous` is equivalent to `double extended`, three entries are needed. However, since the BVM bytecode must be portable across all platforms, `indigenous` entries in the constant pool are always three words (96 bits). While a naive implementation may always manipulate three stack entries for `indigenous`, on platforms where `indigenous` is actually `double`, `indigenous` values may safely be treated as two stack entries. Therefore, Table 22 through Table 26 use the notation *value1.words* to indicate where two to three stack entries are actually manipulated The size of `indigenous` is platform-dependent and fixed for a given BVM implementation; it would be neither meaningful nor correct to treat `indigenous` values sometimes as two words and sometimes as three words during execution of a single program. Since the size of `indigenous` values varies, separate stack manipulation instructions are needed for the `indigenous` type.

Table 22 — New comparison opcodes for the `indigenous` type.

Operation	Opcode Mnemonic	Stack
Compare `indigenous`	*ncmpg, ncmpl*	…,*value1.words,value2 .words*⇒ …,*result*

[45] JVM defines an abstract notion of "word." The lower bound on the size of a JVM word is 32 bits since a word must be able to hold a `float`. However since a reference or native pointer must also fit into one word ([66] §3.4), on 64 bit architectures a JVM word may actually be 64 bits. A JVM interpreter or JIT is not obliged to actually use a full 64 bit word to hold a 32 bit value (nor to use two 64 bit words to hold a single 64 bit value, such as a `double` or `long`). Only the program semantics need be maintained; alignment and padding are implementation details.

Table 23 — New memory opcodes for the `indigenous` type.

Operation	Opcode Mnemonic	Stack
Load `indigenous` from array	*naload*	…,*arrayref,index* ⇒ …,*value.words*
Store `indigenous` into array	*nastore*	…,*arrayref,index, value.words* ⇒ …
Load `indigenous` from local variable	*nload* index	… ⇒ …,*value.words*
Store `indigenous` into local variable	*nstore* index	…,*value.words* ⇒ …
Push `indigenous` from constant pool (wide index)[46]	*ldc_nw* index1 index2	… ⇒ …, *value.words*
Return `indigenous` from method	*nreturn*	…,*value.words* ⇒ *[empty]*

Table 24— New stack manipulation opcodes for the `indigenous` type.

Operation	Opcode Mnemonic	Stack
Duplicate `indigenous`[47]	*dupn*	…,*value.words*⇒ …,*value.words,value.words*
Pop `indigenous`	*popn*	…,*value.words*⇒ …

Table 25 — New conversion opcodes for the `indigenous` type.

Operation	Opcode Mnemonic	Stack
Convert `indigenous` to `double`	*n2d*	…,*value.words* ⇒ …,*result.word1,results.word2*
Convert `indigenous` to `float`	*n2f*	…,*value.words* ⇒ …,*result*
Convert `indigenous` to `int`	*n2i*	…,*value.words* ⇒ …,*result*
Convert `indigenous` to `long`	*n2l*	…,*value.words* ⇒ …,*result.word1,result.word2*
Convert `double` to `indigenous`	*d2n*	…,*value.word1,value.word2* ⇒ …,*result.words*
Convert `float` to `indigenous`	*f2n*	…,*value* ⇒ …,*result.words*
Convert `int` to `indigenous`	*i2n*	…,*value* ⇒ …,*result.words*
Convert `long` to `indigenous`	*l2n*	…,*value.word1,value.word2* ⇒ …,*result.words*

[46] Performs a runtime conversion discussed in section 7.1.1.

[47] Unlike other *dup* commands which operate on any type of data (as long as two-word values are not treated as single words), *dupn* is a typed *dup* instruction that only operates on `indigenous` values since the size of `indigenous` is variable across implementations. The other stack manipulation instruction for `indigenous` values, *popn*, is similarly typed.

Table 26 — New arithmetic opcodes for the `indigenous` type.

Operation	Opcode Mnemonic	Stack
Add `indigenous`	*nadd*	…,*value1.words,value2 .words*⇒ …,*result.words*
Divide `indigenous`	*ndiv*	…,*value1.words,value2 .words*⇒ …,*result.words*
Multiply `indigenous`	*nmul*	…,*value1.words,value2 .words*⇒ …,*result.words*
Negate `indigenous`	*nneg*	…,*value.words* ⇒ …,*result.words*
Subtract `indigenous`	*nsub*	…,*value1.words,value2 .words*⇒ …,*result.words*
Remainder `indigenous`	*nrem*	…,*value1.words,value2 .words*⇒ …,*result.words*

Besides adding opcodes, a new base type for `indigenous`, N, needs to be added so that field descriptors for methods and variables can indicate the new type ([66] §4.3.2). As shown in Figure 57, a new constant pool tag and entry are also needed ([66] §4.3.2). The *newarray* instruction is modified to enable the creation of `indigenous` arrays, indicated by using the new *atype* T_INDIGENOUS=12.

```
CONSTANT_Indigenous = 13

CONSTANT_Indigenous_info {
    u1 tag;
    u4 high_bytes;
    u4 middle_bytes;
    u4 low_bytes;
}
```

Figure 57 — Additions to JVM constant pool structures to support `indigenous`.

Part of the code attribute of a method in a `class` file is `max_stack`, "the maximum number of words on the operand stack at any point during the execution of the method" ([66] §4.7.4). Since the actual size of `indigenous` present at runtime is unknown, a conservative estimate of `max_stack` is computed assuming each `indigenous` value takes three stack words.

A number of existing JVM instructions must be modified to accommodate the `indigenous` type. Table 27 shows that the *getfield* and *getstatic* instructions are modified so that they may return up to three words of an `indigenous` value. Similarly, the *putfield* and *putstatic* instructions are modified so that they accept the two to three words of a `indigenous` value. Since the puts and gets of `static` values do not happen until runtime, `indigenous static` values to do not require the portable encoding used for `indigenous` literals. Objects can use the local size of `indigenous` as can `static` fields. The *wide* instruction modifies the behavior of *nload* and *nstore* in the same manner as other load and store instructions. Expressions involving the `indigenous` type are not considered constant.

Table 27 — Instructions with new stack signatures to operate on the `indigenous` type.

Operation	Opcode Mnemonic	Additional Stack Signature
fetch field from object	*getfield*	…,*objectref* ⇒ …,*value.words*
get `static` field from class	*getstatic*	… ⇒ …,*value.words*
set field in object	*putfield*	…,*objectref,value.words*⇒ …
set `static` field in class	*putstatic*	…,*value.words*⇒ …

7.1.1. Decoding `indigenous` Constant Pool Entries

As mentioned in section 6.1, when an `indigenous` value is used at runtime, some computation must be done since the size of `indigenous` varies from machine to machine. Simply rounding a `double extended` representation of a number to `double` on systems where `indigenous` is implemented as `double` is not sufficient since double rounding can occur. To minimize changes to JVM, BVM provides a single instruction to decode `indigenous` constant pool values at the cost of a somewhat elaborate encoding. The BVM *ldc_nw* instruction takes a constant pool representation of an `indigenous` value and computes the appropriate correctly rounded `double` or `double extended` value. Using a clever encoding for `indigenous` limits the amount of computation needed. Borneo pseudocode implementing the conversion algorithm is given in Figure 58 (the algorithm is not quite Borneo code since the algorithm must have dynamic trapping status). In the common case, a floating point addition and some integer operations need to be performed; the less common case requires some additional effort. The *ldc_nw* instruction can signal overflow and underflow if `indigenous` is implemented as `double` and can always signal inexact regardless of the format implementing `indigenous`. If the conversion does not signal, the value does not need to be recalculated for each access.[48] The corresponding encoding for `indigenous` literals in the constant pool is discussed in section 7.1.2.

```
indigenous constant_pool_convert(double component1, float component2)
   admits none
   yields overflow, underflow, inexact
{
   // For correct behavior, this method must run with the trapping environment of its caller.
   // This code works if indigenous is implemented as double or as double extended.
   rounding Math.TO_NEAREST;
   indigenous result;
   int test;

   // need to extract trailing 8 bits from the float component, bitwise convert float to integer and mask
   test = Float.floatToIntBits(component2) & 0x000000FF;

   if(test == 0)    // double only, float first, or double first encoding is being used (see section 7.1.2)
      {
         // Assume both components are scaled appropriately.
         // The addition may signal inexact and overflow, inexact and underflow, just inexact, or nothing at all
         return (indigenous)component1 + (indigenous)component2;
      }
   else  // scaled sum encoding is being used
      {
         // zero out the test bits before performing the add to avoid spurious inexact signals
         component2 = Float.intBitsToFloat(Float.floatToIntBits(component2) & 0xFFFFFF00);

         // this addition may signal inexact but not overflow or underflow
         result = (indigenous)component1 + (indigenous)component2;

         // set sign of test to be the same sign as the exponent of result
         if(Math.abs(result) < 1.0n)
            test *= –1;

         //scalb signals overflow or underflow with inexact and returns infinity or zero if appropriate
         return Math.scalb(result, 128 * test);
      }
}
```

Figure 58 — Pseudocode to convert `indigenous` constant pool entries to correctly rounded `double` or `double extended` values.

[48] Therefore loading `indigenous` literals is a suitable candidate for a new *_quick* instruction; see section 7.6 for an explanation of *_quick* instructions.

7.1.2. Encoding `indigenous` Constant Pool Entries

In general, storing `indigenous` values can require the full 15 exponent bits and 64 significand bits of the `double extended` format. Many other floating point values more likely to occur do not require the large range available in `double extended`; most floating point literals that occur in programs probably lie within the exponent range of `float`. Borneo uses this assumption to design an encoding for `indigenous` constant pool entries so that `indigenous` values within the range of normalized `float` numbers can be decoded quickly. The overall strategy is to construct the value of an `indigenous` literal from the sum of a `float` number and a `double` number. The layout of an `indigenous` constant pool entry is given in Figure 59; there are two components, a `double` component and a `float` component. Not all the significand bits of the `float` component are needed to hold the significand of the `indigenous` value; the low order bits of the `float` component are used as test bits to indicate if a slower decoding algorithm supporting a larger exponent range needs to be used. The remainder of this section presents the details of encoding `indigenous` literals in the constant pool; it is assumed that the decimal to binary conversion process can provide the necessary significand bits to encode a `double extended` value (along with additional guard, round, and sticky bits to provide proper rounding to `double`).

95 … 32	31 … 8	7 … 0
`double` value	`float` value	test bits

Figure 59 — 96 bit Encoding used to represent `indigenous` values in the constant pool (figure not drawn to scale).

The constant pool encoding for `indigenous` special values is given in Table 28. Adding two NaNs does not generate the invalid signal. Adding two like-signed zeros always produces a zero having the same sign as the inputs.

Table 28 — Representation of special `indigenous` values in the constant pool.

`indigenous` value	`double` component	`float` component
NaNn	NaNd	NaNf
+infinityN	+infinityD	+infinityF
–infinityN	–infinityD	–infinityF
+0.0n	+0.0d	+0.0f
–0.0n	–0.0d	–0.0f

For general `indigenous` floating point numbers, as summarized in Table 29, four different constant pool encodings are used depending on the range and precision of the floating point value being represented. The first encoding, called *`double` only*, is used when an `indigenous` number is exactly representable as a `double`; in that case, the `indigenous` number is represented by setting the value of the `double` component appropriately and setting the `float` component to zero. A different approach is needed when the full 64 significand bits of a `double extended` value need to be represented (`double` only has 53 significand bits). To ensure proper rounding to both `double` and `double extended`, at least 67 bits of precision must be present in any general `indigenous` encoding (64 bits for `double extended` plus guard, round, and sticky bits for proper rounding [59]). The total number of significand bits in a `double` and a `float` is more than adequate to encode a properly rounded `indigenous` value; therefore, where possible, an `indigenous` constant should be constructed by promoting a `float` component and a `double` component to `indigenous` and then adding. The *`float` first* and *`double` first* encoding schemes use this technique.

In the `float` first encoding, the `float` component of the `indigenous` constant pool entry holds the most significant bits of the number and the `double` component holds the trailing bits, as shown pictorially in Figure 61. The `float` first encoding can be used for `indigenous` values whose unbiased exponent lies between –127 and 127.[49] Somewhat larger values can be represented with the `double` first encoding, where the `double` component holds the leading bits and the `float` component the trailing bits (Figure 62). The exponent range of the `double` first encoding is constrained by a `float` value having to represent bits adjacent to the `double`'s least

[49] Although `float` values with a normalized exponent of –127 are subnormal, since these values are promoted to `indigenous` (which is at least as wide as `double`) unnecessary computation on subnormals is avoided.

significant bit. The `double` only encoding is not simply a special case of `double` first since the `double` only encoding can represent values over the full exponent range of `double`, –1022 to +1023.

Table 29 — Representations of normal `indigenous` values in the constant pool.

Encoding	**Properties of `indigenous` number *n***	**`double` component**	**`float` component**
`double` only	exactly representable as a `double`	`double` representation of *n*	0.0f
	Full precision needed		
`float` first	–127 ≤ `logbn`(*n*) ≤ 127 (see Figure 61)	trailing 50 to 49 significand bits of *n* with guard, round, and sticky	leading 14 to 15 significand bits of *n*
`double` first	–73 ≤`logbn`(*n*) ≤ 180 (see Figure 62)	leading 53 significand bits of *n*	trailing 11 significand bits of *n* with guard, round, and sticky
scaled sum	`logbn`(*n*) ≥ 128 or `logbn`(*n*) ≤ –128	Use test bits of `float` component to hold the 8 high order bits of *n*'s exponent. Use remaining bits of `float` and `double` components to hold significand bits of *n* with an exponent equal to the 7 low order bits of *n*'s exponent.	

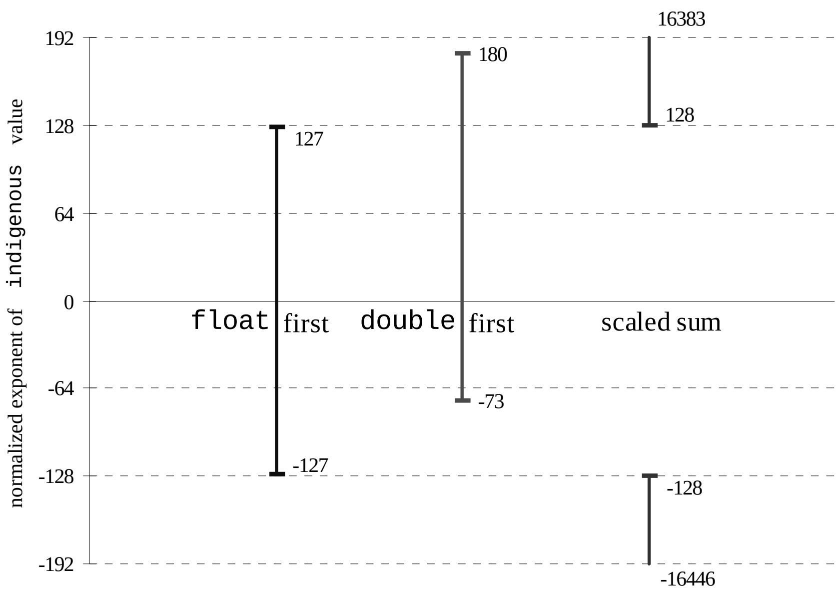


Figure 60 — Valid exponent ranges for different `indigenous` constant pool entry encodings.

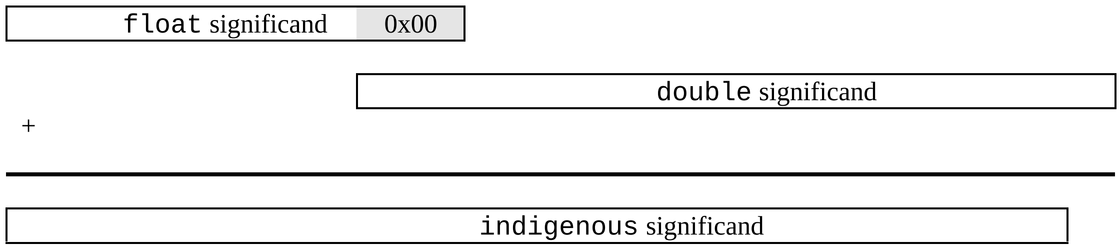


Figure 61 — `float` component holding the leading significand bits of an `indigenous` literal.

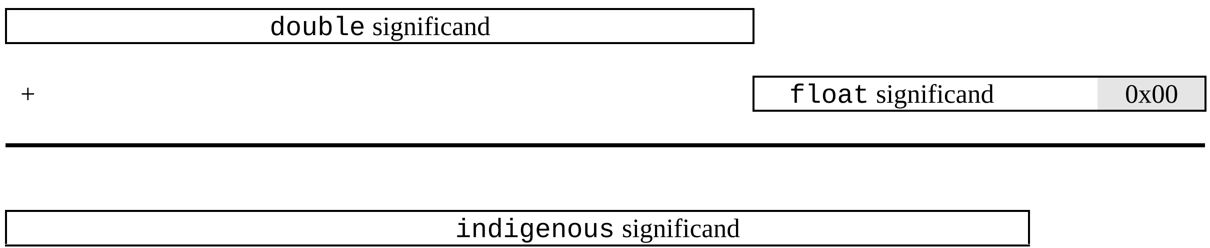


Figure 62 — `double` component holding the leading significand bits of an `indigenous` literal.

Figure 61 and Figure 62 could imply that for a given encoding there is a fixed difference in the exponents of the `float` and `double` components. The difference in exponents would be equal to the number of significand bits used for encoding the target significand in the leading component. However, that difference is only a lower bound on the difference in exponents. Since both leading and trailing components are normalized floating point numbers, if a component is non-zero, the most significant bit in that component must be one. But, the first bit position after the significand bit in the leading component may be zero. Therefore, due to zeros in some bit positions, the trailing component may have a smaller exponent than implied by the number of significand bits stored in the leading component.

The fourth *scaled sum* encoding is more elaborate than the `float` first or `double` first encoding. The scaled sum encoding can handle all `double extended` values that have exponents larger than 127 in magnitude, including `double extended` subnormals. Essentially, the scaled sum encoding splits the number's exponent into two parts; the high order bits of the exponent are stored in the test bits of the `float` component and the low order bits of the exponent are stored in the exponent of the `float` component. If the test bits are zeroed out, the `float` and `double` components form a number in the `float` first encoding; a number with the same significand as the intended value but with a different exponent. The exponent value of this `float` first number is equal to

((int)abs(logbn(*n*)) mod 128) * sign(logbn(*n*))

where *n* is the value of the number being encoded. After adding/subtracting out the `float` first exponent, the remaining value of the original exponent is a multiple of 128 that can be stored in an 8 bit unsigned value, such as the test bits (the eventual sign of the test bit portion of the exponent is the same as the sign of the exponent in the `float` first component). Using `scalb` on the `float` first value, the unaccounted for portion of the encoded value's exponent can be incorporated into the final value, as shown in Figure 58.

Not all possible combinations of `float` and `double` numbers are valid encodings of `indigenous` literals. In particular, no set of test bits with a value greater than 128 is legal. Catching such malformed constant pool entries is part of Borneo `class` file verification. As shown in Figure 60, the three full precision encoding schemes have overlapping ranges; for exponents between -73 and 127 either `float` first or `double` first can be used, for exponents between 128 and 180 either `double` first or scaled sum can be used. Using `double` first instead of `float` first has no performance impact, but using `double` first instead of scaled sum results in faster conversions.

7.2. Rounding Modes

Table 30 — New opcodes supporting IEEE 754 rounding modes.

Operation	Opcode Mnemonic	Stack
Get current rounding mode	*getrnd*	…, ⇒ …,*result*
Set rounding mode	*setrnd*	…,*value* ⇒ …

IEEE 754 dynamic rounding modes are supported in BVM. The instruction *getrnd* returns the rounding mode under which the current thread is running. The value returned is encoded according to the mapping in Table 31. The same values are used to set the rounding mode with the *setrnd* instruction. If an invalid rounding mode is given, an `UnknownRoundingModeException` is thrown. While Borneo requires that `rounding` declarations be lexically scoped, the BVM access to rounding modes is unstructured. Therefore, BVM verification does not check to see that rounding mode accesses are lexically scoped or that rounding modes are restored on method call and exit.

Table 31 —BVM encodings of rounding modes.

Integer Code	Rounding Mode
0	toward zero
1	to nearest (default)
2	toward positive infinity
3	toward negative infinity

7.3. Quiet Floating Point Comparisons

Table 32 — New opcodes supporting comparisons that do not signal invalid on a NaN input.

Operation	Opcode Mnemonic	Stack
Quiet compare `double`	*dcmplq, dcmpgq*	…,*value1.word1,value1.word2,* *value2.word1,value2.word2* ⇒ …,*result*
Quiet compare `float`	*fcmplq, fcmpgq*	…,*value1,value2* ⇒ …,*result*
Quiet compare `indigenous`	*ncmplq, ncmplq*	…,*value1.words,value2 .words*⇒ …,*result*

BVM defines the existing floating point comparison instructions in JVM to signal invalid if given a NaN operand (the JVM specification does not include the IEEE flags or traps). The quiet floating point comparison operators have the same behavior as the normal floating point comparison operators, except that the invalid is not signaled if any of the arguments is a NaN.

To see if two floating point numbers are unordered, a *cmpgq* can be done followed by a *cmplq* on the same operands. Since the *g* and *l* compare instructions only differ in their treatment of NaN's, the results of *cmpgq* and *cmplq* on the same input differ only if the numbers are unordered. Quiet comparisons are recommended in the IEEE 754 standard.

Comparing floating point numbers for equality (`==`) is likely to be rarer than other comparisons. Therefore, the current JVM comparison instructions are redefined to signal invalid when given a NaN input. This way more Java programs already compiled into JVM conform to Borneo (and IEEE 754) semantics. Borneo would compile floating point `==` and `!=` tests using the new quiet BVM comparison instructions.

7.4. Exception Handling and Traps

Table 33 — New opcodes supporting sticky flags and floating point exceptions.

Operation	Opcode Mnemonic	Stack
Get current exception trap mask	*gettrapmask*	…, ⇒ …,*value*
Set current exception trap mask	*settrapmask*	…,*value* ⇒ …
Get current sticky flags settings	*getsticky*	…, ⇒ …,*value*
Set current sticky flags settings	*setsticky*	…,*value* ⇒ …

settrapmask and *setsticky* pop the top word off of the stack and use it to set the exception trap mask and sticky flags, respectively. Conversely, *gettrapmask* and *getsticky* fetch the trap mask and sticky flags and push the value onto the top word of the stack. All four instructions use the same operand/result format, which is listed in Figure 63.

INV	Invalid
OVF	Overflow
UDF	Underflow
DBZ	Divide-by-Zero
INX	Inexact

n … 5	4	3	2	1	0
	INV	**OVF**	**UDF**	**DBZ**	**INX**

Figure 63 — Encodings for trapping status

settrapmask and *setsticky* ignore all but bits 4-0 of the operand when loading the trap mask or sticky flags. *gettrapmask* and *getsticky* return a word zeroed out except possibly for the lowest five bits.

The sticky flag signature of a method is encoded by two new method attributes, `AdmitsSignature` and `YieldsSignature` (Figure 64). The new attributes use the `flag_signature` field to encode what flags are admitted/yielded using the bit representation in Figure 63. JVM implementations are required to ignore novel attributes not defined in the JVM specification ([66] §4.7) so these new attributes should not affect existing JVM environments.

```
AdmitsSignature_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 flag_signature;
}

YieldsSignature_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 flag_signature;
}
```

Figure 64 — New attributes to record flag signatures.

For now, BVM does not require verification that a method has the claimed sticky flag signature, but that requirement may be added in the future. However, the verification process does check that an overriding method has the same declared flag signature as the overridden method. In the absence of flag attributes, the default flag signature is `admits all yields all`. A Borneo compiler should add `Exceptions` attribute entries ([66] §4.7.5) for any floating point exceptions that may be thrown due to enabling trapping mode. The current JVM specification does not verify `throws` clauses of methods. If an instruction throws a floating point exception, the corresponding sticky flag is not set. Table 34 lists what floating point exceptions BVM instructions can throw; different causes of invalid exceptions are distinguished, as in Table 7.

Table 34 —New exceptional conditions possibly generated by arithmetic bytecodes.

Operation	Opcodes	Exceptional Conditions INV	OVF	UDF	DBZ	INX
Floating point add, subtract	*fadd fsub dadd dsub nadd nsub*	X	X	X		X
Floating point multiply	*fmul dmul nmul*	X	X	X		X
Floating point divide	*fdiv ddiv ndiv*	X	X	X	X	X
Float point remainder[50]	*frem drem nrem*	X				
Floating point compare	*fcmpl fcmpg dcmpl dcmpg ncmpl ncmpg*	X				
	fcmplq fcmpgq dcmplq dcmpgq ncmplq ncmpgq					
Convert floating point to integer	*f2i f2l d2i d2l n2i n2l*	X				X
Convert integer to floating point	*i2f i2d i2n l2f l2d l2n*	X				X
Convert between floating point types	*n2f n2d d2f*		X	X		X
	f2d f2n d2n					
Load `indigenous` from constant pool[51]	*ldc_nw*		X	X		X

7.5. Threads

No bytecode changes are required to support threads. However, the JVM implementation must be modified so that the rounding mode, exception mask, and sticky flags are maintained as part of the thread state. When a thread context switch takes place, these three values must be saved with the outgoing thread, and the values associated with the incoming thread must be installed. New threads are started with the sticky flags cleared, rounding set to round to nearest, and trapping status set to non-trapping mode.

7.6. Overall

The JVM specification states that the JVM may rearrange bytecode instructions in order to improve performance as long as the reordering preserves reproducibility. In general, the BVM system may not move floating point instructions from one side of a *setrnd*, *settrapmask*, *setsticky*, or *getsticky* instruction to the other, since doing so could change the program's behavior. Such reorderings are valid if it can be determined that the given instruction movement could not change the program's behavior. For example, a comparison can be moved across a *setrnd* since the rounding mode does not effect the result of a comparison. If only the divide by zero flag is being tested, addition and multiplication instructions can be moved around a *getsticky* instruction since the add and multiply instructions cannot set the divide by zero flag.

The functionality of *setrnd*, *getrnd*, *settrapmask*, *gettrapmask*, *setsticky*, and *getsticky* must be known to BVM to support this kind of analysis and optimization. For example, currently one can set the rounding mode by calling a `native` method to access the hardware's floating point control registers directly. However, the JVM would not be aware that the `native` method did this. The JVM could be made aware that the method of a particular class changes the rounding mode; alternatively, BVM adds new instructions to implement this functionality.

[50] The JVM floating point remainder instructions do not implement the IEEE 754 remainder operation.
[51] When loading an `indigenous` value from the constant pool, overflow and underflow can only be signaled if `indigenous` is implemented with the `double` format.

Table 35 — Instruction counts in current JVM

Type of Opcode	Number
regular instructions	201
reserved opcodes	3
totally uncommitted opcodes	25
quick instructions	27
Total	**256**

Table 36 — Instruction counts for new Borneo opcodes.

Purpose of Opcode	Number
`indigenous` versions of existing instructions	25
rounding mode manipulation	2
quiet comparisons (includes 1 for `indigenous`)	3
flag and traps manipulation	4
Total	**34**

As shown in Table 35, most of the opcodes in the JVM bytecode are already assigned to designate either a real instruction or a *_quick* instruction variant. In Sun's implementation of the JVM interpreter, the bytecode sequence of a program can be dynamically rewritten at runtime with *_quick* pseudo-instruction variants ([66] §9). Instructions that reference the constant pool often need to perform some checking in addition to the actual execution of the instruction. When such instructions are first executed, the needed checking can be performed. If the checking is successful the instruction can then be replaced with a *_quick* pseudo-instruction variant that does not perform the checking on subsequent executions. The *_quick* pseudo-instructions are *not* part of the JVM specification; they are only an optimization used in Sun's JVM implementation. The opcodes currently used for *_quick* pseudo-instructions could be used to encode some of the new BVM instructions.

Unfortunately, Borneo proposes to add 34 new instructions (Table 36) while JVM only has 25 possible opcodes totally uncommitted. However, the additional nine instructions can be accommodated in various ways. The simplest way to add all the new opcodes is to designate one opcode as an escape for a two byte instruction. For example, the rounding mode, quiet comparison, and exception handling instructions could be placed within the existing opcode space while the `indigenous` instructions could be prefixed by an escape sequence. A second option is to use the *_quick* instruction opcodes. The *_quick* instruction opcodes and the uncommitted opcodes together can easily encode the Borneo extensions while keeping all instructions one byte. A third, more radical option is to reclaim existing JVM opcodes. As detailed in Appendix 9.2, nearly one quarter of the existing JVM instructions are dedicated to peephole optimized versions of more general instructions. Eliminating these redundant opcodes alone would free up enough opcode space to encode all the Borneo extensions. Since the redundant opcodes are exactly expressible in terms of their more general counterparts, a simple `class` file to `class` file transformer can be written to purge all uses of the redundant opcodes from existing class files. Since BVM `class` files have a different magic number[52] than JVM `class` files, an interpreter could dynamically recognize whether or not the redundant opcodes can be used in a given `class` file.

[52] The magic number is the first four bytes of the class file which identifies its format. While Java is a `ØxCAFEBABE`, Borneo is a `ØxCAFEDØØD`.

8. Changes to the Package `java.lang`

The structure and wording of this portion of the Borneo specification is strongly modeled after JLS chapter 20. Specifications are not given for the new numeric classes discussed in section 6.4, only for changes to existing classes in the package `java.lang` and for the new `Indigenous` class.

8.1. Changes to `java.lang.Class`

8.1.1. `public String` **`getName`**`()`

The behavior of this method is the same as in Java except that an element type name encoding is added to represent the new basic type `indigenous`:

```
N       indigenous
```

8.1.2. `public Class` **`getSuperclass`**`()`

The behavior of this method is the same as in Java except that `null` is returned for `value` classes.

8.2. Changes to `java.lang.Number`

8.2.1. `public abstract indigenous` **`indigenousValue`**`()`

The general contract of the `indigenousValue` method is that it returns the numeric value represented by this `Number` object after converting it to type `indigenous`.

Overridden by `Integer`, `Long`, `Float`, `Double`, and `Indigenous`.

8.3. Changes to `java.lang.Integer`

8.3.1. `public indigenous` **`indigenousValue`**`()`

The `int` value represented by this `Integer` object is converted to type `indigenous` and the result of the conversion is returned.

Overrides the `indigenousValue` method of `Number`.

8.4. Changes to `java.lang.Long`

8.4.1. `public indigenous` **`indigenousValue`**`()`

The `long` value represented by this `Long` object is converted to type `indigenous` and the result of the conversion is returned.

Overrides the `indigenousValue` method of `Number`.

8.5. Changes to `java.lang.Float`

8.5.1. `public static final float` **`MIN_VALUE`** `= 1.4e-45f;`

The constant value of this field is the smallest positive nonzero value of type `float`. It is equal to the value returned by `Math.nextAfter(0.0f, infinityF)`.

8.5.2. `public static final float` **`MIN_NORMAL`** `= 1.17549435e-38f;`

The constant value of this field is the smallest positive normalized value of type `float`. It is equal to the value returned by `Math.nextAfter(0.0f, infinityF)/(Math.nextAfter(1.0f, infinityF)-1.0f)`.

8.5.3. `public static final float MAX_VALUE = 3.4028235e+38f;`

The constant value of this field is the largest positive finite value of type `float`. It is equal to the value returned by `Math.nextAfter(infinityF, 0.0f)`.

8.5.4. `public static final float NEGATIVE_INFINITY = -infinityF;`

The constant value of this field is the negative infinity of type `float`.

8.5.5. `public static final float POSITIVE_INFINITY = infinityF;`

The constant value of this field is the positive infinity of type `float`.

8.5.6. `public static final float NaN = NaNf;`

The constant value of this field is the Not-a-Number value of type `float`.

8.5.7. `public static final float ROUNDING_THRESHOLD = 5.960465e-8f;`

The constant value of this field is the smallest positive `float` value such that under the round to nearest rounding mode `1.0f` plus this value is not equal to `1.0f`. It is equal to the value returned by `Math.nextAfter((Math.nextAfter(1.0f, infinityF)-1.0f)/2.0f, infinityF)`.

8.5.8. `public static final int SIGNIFICAND_WIDTH = 24;`

The constant value of this field is the number of bits in the significand of a value of type `float`, including the implicit bit. It is equal to the value returned by `–(int)Math.logb(Math.nextAfter(1.0f, infinityF) -1.0f)+ 1`.

8.5.9. `public static final int MIN_EXPONENT = -126;`

The constant value of this field is the smallest exponent of a normalized value of type `float`. It is equal to the value returned by `(int)Math.logb(Float.MIN_NORMAL)`.

8.5.10. `public static final int MAX_EXPONENT = 127;`

The constant value of this field is the largest exponent of a normalized value of type `float`. It is equal to the value returned by `(int)Math.logb(Math.nextAfter(infinityF, 0.0f))`.

8.5.11. `public static final int BIAS_ADJUST = 192;`

The constant value of this field is the absolute value of the amount by which the exponent is adjusted when a value of type `float` overflows or underflows when trapping on those conditions is enabled. It is equal to the value returned by

```
(int)(3.0f * Math.scalb(2.0f,
                         (int)(Math.ceil(Math.log(
                              Math.logb(Float.MAX_VALUE))/
                                   Math.log(Math.E))-2) )).
```

8.5.12. `public Float(indigenous value)`

This constructor initializes a newly created `Float` object so that it represents the result of narrowing the argument from type `indigenous` to type `float`.

8.5.13. `public indigenous indigenousValue()`

The `float` value represented by this `Float` object is converted to type `indigenous` and the result of the conversion is returned.

Overrides the `indigenousValue` method of `Number`.

8.5.14. `public static String toString(double d, int rm) throwns UnknownRoundingModeException`

This method has the same requirements as the `toString` of one argument method except that the decimal string is rounded under the rounding mode represented by `rm`. If `rm` does not specify a valid rounding mode, an `UnknownRoundingModeException` is thrown.

8.5.15. `public static float valueOf(String s, int rm) throws NullPointerException, NumberFormatException, UnknownRoundingModeException`

The string `s` is interpreted as the representation of a floating point value and a `Double` object representing that value is created and returned.

If `s` is `null`, then a `NullPointerException` is thrown.

Leading and trailing whitespace characters in `s` are ignored. To be interpreted as a number, the rest of `s` must have the same lexical structure as a Borneo floating point literal. If `s` does not have that structure, a `NumberFormatException` is thrown. The value of the returned `Float` object is the decimal floating point value of `s` correctly rounded and converted to `float` using the rounding mode specified by `rm`. If `rm` does not specify a valid rounding mode, an `UnknownRoundingModeException` is thrown.

8.6. Changes to `java.lang.Double`

8.6.1. `public static final double MIN_VALUE = 4.94065645841246544e-324;`

The constant value of this field is the smallest positive nonzero value of type `double`. It is equal to the value returned by `Math.nextAfter(0.0, infinity)`.

8.6.2. `public static final double MIN_NORMAL = 2.2250738585072014E-308;`

The constant value of this field is the smallest positive normalized value of type `double`. It is equal to the value returned by `Math.nextAfter(0.0, infinity)/(Math.nextAfter(1.0, infinity)-1.0)`.

8.6.3. `public static final double MAX_VALUE = 1.79769313486231570E+308;`

The constant value of this field is the largest positive finite value of type `double`. It is equal to the value returned by `Math.nextAfter(infinity, 0.0)`.

8.6.4. `public static final double NEGATIVE_INFINITY = -infinity;`

The constant value of this field is the negative infinity of type `double`.

8.6.5. `public static final double POSITIVE_INFINITY = infinity;`

The constant value of this field is the positive infinity of type `double`.

8.6.6. `public static final double NaN = NaNd;`

The constant value of this field is the Not-a-Number value of type `double`.

8.6.7. `public static final double ROUNDING_THRESHOLD = 1.1102230246251568E-16d;`

The constant value of this field is the smallest positive `double` value such that under the round to nearest rounding mode `1.0` plus this value is not equal to `1.0`. It is equal to the value returned by `Math.nextAfter((Math.nextAfter(1.0, infinity)-1.0)/2.0, infinity)`.

8.6.8. `public static final int` **`SIGNIFICAND_WIDTH`** `= 53;`

The constant value of this field is the number of bits in the significand of a value of type `double`, including the implicit bit. It is equal to the value returned by `–(int)Math.logb(Math.nextAfter(1.0, infinity) -1.0)+ 1`.

8.6.9. `public static final int` **`MIN_EXPONENT`** `= -1022;`

The constant value of this field is the smallest exponent of a normalized value of type `double`. It is equal to the value returned by `(int)Math.logb(Double.MIN_NORMAL)`.

8.6.10. `public static final int` **`MAX_EXPONENT`** `= 1023;`

The constant value of this field is the largest exponent of a normalized value of type `double`. It is equal to the value returned by `(int)Math.logb(Math.nextAfter(infinity, 0.0))`.

8.6.11. `public static final int` **`BIAS_ADJUST`** `= 1536;`

The constant value of this field is the absolute value of the amount by which the exponent is adjusted when a value of type `double` overflows or underflows when trapping on those conditions is enabled. It is equal to the value returned by

```
(int)(3.0 * Math.scalb(2.0,
          (int)(Math.ceil(Math.log(
                Math.logb(Double.MAX_VALUE))/
                Math.log(Math.E))-2) )).
```

8.6.12. `public` **`Double`**`(indigenous value)`

This constructor initializes a newly created `Double` object so that it represents the result of narrowing the argument from type `indigenous` to type `double`.

8.6.13. `public indigenous` **`indigenousValue`**`()`

The `double` value represented by this `Double` object is converted to type `indigenous` and the result of the conversion is returned.

Overrides the `indigenousValue` method of `Number`.

8.6.14. `public static String` **`toString`**`(double d, int rm) throwns UnknownRoundingModeException`

This method has the same requirements as the `toString` of one argument method except that the decimal string is rounded under the rounding mode represented by `rm`. If `rm` does not specify a valid rounding mode, an `UnknownRoundingModeException` is thrown.

8.6.15. `public static double` **`valueOf`**`(String s, int rm) throws NullPointerException, NumberFormatException, UnknownRoundingModeException`

The string `s` is interpreted as the representation of a floating point value and a `Double` object representing that value is created and returned.

If `s` is `null`, then a `NullPointerException` is thrown.

Leading and trailing whitespace characters in `s` are ignored. To be interpreted as a number, the rest of `s` must have the same lexical structure as a Borneo floating point literal. If `s` does not have that structure, a `NumberFormatException` is thrown. The value of the returned `Double` object is the decimal floating point value of `s` correctly rounded and converted to `double` using the rounding mode specified by `rm`. If `rm` does not specify a valid rounding mode, an `UnknownRoundingModeException` is thrown.

8.7. The Class `java.lang.Indigenous`

Where `indigenous` is implemented as `double`, corresponding fields in `java.lang.Indigenous` and `java.lang.Double` have the same value. The behavior of corresponding methods is also analogous.

```
public final class Indigenous extends Number {
      public static final indigenous MIN_VALUE =
            Math.nextAfter(0.0n, infinityN);
      public static final indigenous MIN_NORMAL =
            Math.nextAfter(0.0n, infinityN)/
            (Math.nextAfter(1.0n, infinityN)-1.0n);
      public static final indigenous MAX_VALUE =
            Math.nextAfter(infinityN, 0.0n);
      public static final indigenous NEGATIVE_INFINITY = -infinityN;
      public static final indigenous POSITIVE_INFINITY = infinityN;
      public static final indigenous NaN = NaNN;
      public static final indigenous ROUNDING_THRESHOLD =
            Math.nextAfter((Math.nextAfter(1.0n, infinityN)-1.0n)/
                              2.0n, infinityN);
      public static final int SIGNIFICAND_WIDTH =
            –(int)Math.logb(nextAfter(1.0n, infinityN) -1.0n)+ 1;
      public static final int MIN_EXPONENT =
            (int)Math.logb(Indigenous.MIN_NORMAL)
      public static final int MAX_EXPONENT =
            (int)Math.logb(Math.nextAfter(infinityN, 0.0n))
      public static final int BIAS_ADJUST =
            (int)(3.0n * Math.scalb(2.0n,
                              (int)(Math.ceil(Math.log(
                              Math.logb(Indigenous.MAX_VALUE))/
                              Math.log(Math.E))-2) ))

      public Indigenous(indigenous value);
      public Indigenous(String s);
            throws NumberFormatException;
      public String toString();
      public boolean equals(Object obj);
      public int hashCode();
      public int intValue();
      public int longValue();
      public int floatValue();
      public int doubleValue();
      public int indigenousValue();
      public static String toString(indigenous n);
      public static String toString(indigenous n, int rm)
            throwns UnknownRoundingModeException
      public static Indigenous valueOf(String s)
            throws NullPointerException, NumberFormatException;
      public static Indigenous valueOf(String s, int rm)
            throws NullPointerException, NumberFormatException,
                  UnknownRoundingModeException;

      public boolean isNaN();
      public static boolean isNaN(indigenous n);
      public boolean isInfinite();
      public static boolean isInfinite(indigenous n);
      public static double[] decompose(indigenous n);
      public static indigenous compose(double d, float f)
            throws NumberFormatException
            admits none yields inexact, overflow, underflow;
      }
```

8.7.1. `public static final indigenous` **`MIN_VALUE`** `= Math.nextAfter(0.0n, infinityN);`

The value of this field is the smallest positive nonzero value of type `indigenous`.

8.7.2. `public static final indigenous` **`MIN_NORMAL`** `= Math.nextAfter(0.0n, infinityN)/(Math.nextAfter(1.0n, infinityN)-1.0n);`

The value of this field is the smallest positive normalized value of type `indigenous`.

8.7.3. `public static final indigenous` **`MAX_VALUE`** `= Math.nextAfter(infinityN, 0.0n);`

The value of this field is the largest positive finite value of type `indigenous`.

8.7.4. `public static final indigenous` **`NEGATIVE_INFINITY`** `= -infinityN;`

The value of this field is the negative infinity of type `indigenous`.

8.7.5. `public static final indigenous` **`POSITIVE_INFINITY`** `= infinityN;`

The value of this field is the positive infinity of type `indigenous`.

8.7.6. `public static final indigenous` **`NaN`** `= NaNN;`

The value of this field is the Not-a-Number of type `indigenous`.

8.7.7. `public static final indigenous` **`ROUNDING_THRESHOLD`** `= Math.nextAfter((Math.nextAfter(1.0n, infinityN)-1.0n)/ 2.0n, infinityN);`

The value of this field is the smallest positive `indigenous` value such that under the round to nearest rounding mode `1.0n` plus this value is not equal to `1.0n`.

8.7.8. `public static final int` **`SIGNIFICAND_WIDTH`** `= -(int)Math.logb(nextAfter(1.0n, infinityN) -1.0n)+ 1;`

The value of this field is the number of bits in the significand of a value of type `indigenous`, including an implicit bit, if present.

8.7.9. `public static final int` **`MIN_EXPONENT`** `= (int)Math.logb(Indigenous.MIN_NORMAL);`

The value of this field is the smallest exponent of a normalized value of type `indigenous`.

8.7.10. `public static final int` **`MAX_EXPONENT`** `= (int)Math.logb(Math.nextAfter(infinityN, 0.0n));`

The value of this field is the largest exponent of a normalized value of type `indigenous`.

8.7.11. `public static final int` **`BIAS_ADJUST`** `= (int)(3.0n * Math.scalb(2.0n,(int)(Math.ceil(Math.log( Math.logb( Indigenous.MAX_VALUE))/ Math.log(Math.E))-2) ));`

The value of this field is the absolute value of the amount by which the exponent is adjusted when a value of type `indigenous` overflows or underflows when trapping on those conditions is enabled.

8.7.12. `public` **`Indigenous`**`(indigenous value)`

This constructor initializes a newly created `Indigenous` object so that it represents the primitive value that is the argument.

8.7.13. `public` **`Indigenous`**`(String s) throws NumberFormatException`

This constructor initializes a newly created `Indigenous` object so that it represents the floating point value of type `indigenous` represented by the string. The string is converted to an `indigenous` value in exactly the manner used by the `valueOf` method.

8.7.14. `public String` **`toString`**`()`

The primitive `indigenous` value represented by this `Indigenous` object is converted to a string exactly as if by the method `toString` of one argument.

Overrides the `toString` method of `Object`.

8.7.15. `public boolean` **`equals`**`(Object obj)`

The result is `true` if and only if the argument is not `null` and is an `Indigenous` object that represents the same bitwise `indigenous` value as this `Indigenous` object. For the purposes of the `equals` method, +0 and –0 are not the same. If the argument and this `Indigenous` object both have the same NaN value, `equals` will return true.

8.7.16. `public int` **`hashCode`**`()`

If `indigenous` is implemented as `double`, return the same result as `Double.hashCode`. Otherwise, follows the general contract for the `hashCode` method.

8.7.17. `public int` **`intValue`**`()`

The `indigenous` value represented by this `Indigenous` object is converted to type `int` and the result of the conversion is returned.

Overrides the `intValue` method of `Number`.

8.7.18. `public int` **`longValue`**`()`

The `indigenous` value represented by this `Indigenous` object is converted to type `long` and the result of the conversion is returned.

Overrides the `longValue` method of `Number`.

8.7.19. `public int` **`floatValue`**`()`

The `indigenous` value represented by this `Indigenous` object is converted to type `float` and the result of the conversion is returned.

Overrides the `floatValue` method of `Number`.

8.7.20. `public int` **`doubleValue`**`()`

The `indigenous` value represented by this `Indigenous` object is converted to type `double` and the result of the conversion is returned.

Overrides the `doubleValue` method of `Number`.

8.7.21. `public indigenous` **`indigenousValue`**`()`

The `indigenous` value represented by this `Indigenous` object is returned.

Overrides the `indigenousValue` method of `Number`.

8.7.22. `public static String` **`toString`**`(indigenous n)`

On platforms where `indigenous` is implemented as `double`, this method returns the same string as `Double.toString((double) n)`. On platforms where `indigenous` is `double extended`, this method has the same specification as `Double.toString` except that the number of decimal digits printed out is the least number of decimal digits necessary to uniquely distinguish the argument value from adjacent `double extended` values (at least one digit is needed for the fractional part of the number).

8.7.23. `public static String` **`toString`**`(indigenous n, int rm) throwns UnknownRoundingModeException`

This method has the same requirements as the `toString` of one argument method except that the decimal string is rounded under the rounding mode represented by `rm`. If `rm` does not specify a valid rounding mode, an `UnknownRoundingModeException` is thrown.

8.7.24. `public static Indigenous` **`valueOf`**`(String s) throws NullPointerException, NumberFormatException`

The string `s` is interpreted as the representation of a floating point value and an `Indigenous` object representing that value is created and returned.

If `s` is `null`, then a `NullPointerException` is thrown.

Leading and trailing whitespace characters in `s` are ignored. To be interpreted as a number, the rest of `s` must have the same lexical structure as a Borneo floating point literal. If `s` does not have that structure, a `NumberFormatException` is thrown. The value of the returned `Indigenous` object is the decimal floating point value of `s` correctly rounded and converted to `indigenous` using round to nearest.

8.7.25. `public static Indigenous` **`valueOf`**`(String s, int rm) throws NullPointerException, NumberFormatException, UnknownRoundingModeException`

The string `s` is interpreted as the representation of a floating point value and an `Indigenous` object representing that value is created and returned.

If `s` is `null`, then a `NullPointerException` is thrown.

Leading and trailing whitespace characters in `s` are ignored. To be interpreted as a number, the rest of `s` must have the same lexical structure as a Borneo floating point literal. If `s` does not have that structure, a `NumberFormatException` is thrown. The value of the returned `Indigenous` object is the decimal floating point value of `s` correctly rounded and converted to `indigenous` using the rounding mode specified by `rm`. If `rm` does not specify a valid rounding mode, an `UnknownRoundingModeException` is thrown.

8.7.26. `public boolean` **`isNaN`**`()`

The result is `true` if and only if the value represented by this `Indigenous` object is NaN.

8.7.27. `public static boolean` **`isNaN`**`(indigenous n)`

The result is `true` if and only if the value of the argument is NaN.

8.7.28. `public boolean` **`isInfinite`**`()`

The result is `true` if and only if the value represented by this `Indigenous` object is positive or negative infinity.

8.7.29. `public static boolean` **`isInfinite`**`(indigenous n)`

The result is `true` if and only if the value of the argument is positive or negative infinity.

8.7.30. `public static double[]` **`decompose`**`(indigenous n) admits none yields none`

Takes the `indigenous` argument and returns an array of values suitable for creating `indigenous` constant pool entries using the encodings defined in section 7.1.2. The first element of the returned array is the `double` component, the second element is a `float` value widened to `double`. If a number can be represented by more than one encoding (see section 7.1.2), `double` only is used if possible, `float` first takes precedence over `double` first, and `double` first takes precedence over scaled sum.

8.7.31. `public static indigenous compose(double d, float f) throws NumberFormatException admits none yields inexact, overflow, underflow`

Takes a `double` argument and a `float` argument, interprets them as encoding an `indigenous` literal as described in section 7.1.2, and returns the appropriate `indigenous` value. If the `double` and `float` values do not comprise a valid encoding, a `NumberFormatException` is thrown. The inexact flag is raised if the floating point value is not exactly representable as an `indigenous` value. The overflow and underflow flags are raised appropriately if the floating point value is too large or too small to be represented as an `indigenous` value on the current platform.

8.8. Changes to `java.lang.Math`

Borneo adds transcendental functions acting on `indigenous` values to the `Math` class. However, since Java currently specifies those functions acting on `double` values in terms of `fdlibm` algorithms, Borneo provides no detailed specification of those functions acting on `indigenous` values. Providing necessary and sufficient conditions for transcendental functions is left as future work.

The `getRound` and `setRound` methods represent rounding modes as indicated by the `TO_NEAREST`, `TO_ZERO`, `TO_POSITIVE_INFINITY`, and `TO_NEGATIVE_INFINITY` `static final` fields. The four rounding modes occur in the range [0, 3]. The four rounding modes should be represented as an enumerated type, but Java does not directly support enumerated types.

The `getFlags`, `setFlags`, and `setFlag` methods manipulate the sticky flag state using an integer representation of the sticky flags. Each of the low order five bits an integer represents a different condition, as given by the `*_FLAG` fields. The `setFlag` and `setFlags` methods ignore all other bits; the `getFlag` method always returns 0 in bits 31-5.

Since Borneo has many features intended for robust numerical programs, the Borneo implementation of the IEEE recommended functions should be robust as well. Although not strictly required by the language semantics, it is suggested that the IEEE recommended functions work properly even under “malicious” dynamic rounding mode and trapping status settings. Additionally, for many of the recommended functions, if a NaN is given as input, a NaN must be returned. Borneo suggests the same NaN be returned.

The family of `fpClass` methods return the kind of floating point number given to them (infinite, normal, subnormal, etc.). This information is encoding as indicated by the `FP_*` fields.

The Borneo `Math` library does not include any methods to query or set the trapping status. Borneo language semantics have trapping status as a static property. Having a method to explicitly set the trapping status could foil the compiler’s analysis of a program; therefore, no such method is provided.

To support the new operators, Borneo makes the `Math` class a `value` class.

8.8.1. Exponentiation operators

Borneo has an exponentiation operation for each primitive floating point type. The operator is named “`**`” as in FORTRAN. The exponentiation operators have the same behavior as the `pow` method on the same type of argument.

```
public static float op**(float base, float power)
public static double op**(double base, double power)
public static indigenous op**(indigenous base, indigenous power)
```

8.8.2. Directed Rounding operators

Sixteen directed rounding operators are provided for each primitive floating point type. There is one directed rounding operator for each combination of type, rounding mode, and operator affected by rounding mode. The directed rounding operators have the same flag behavior as the corresponding built-in Java operator. The signatures of the rounding mode operators are of the form

`public static τ opαβ(τ a, τ b)`

where $\tau \in$ {`float`, `double`, `indigenous`}, $\alpha \in$ {`+`, `–`, `*`, `/`}, and $\beta \in$ {`@`, `^`, `_`, `%`}. The elements of the set {`@`, `^`, `_`, `%`} indicate respectively round to zero, round toward $+\infty$, round toward $-\infty$, and round to nearest.

8.8.3. Quiet Comparison operators

Borneo adds four new quiet comparison operators for each primitive floating point type. The quiet comparison operators do not set the invalid flag when given a NaN argument and return `true` for the unordered relation. The quiet comparison operators have signatures of the form

`public static boolean op`γ`?(`τ `a,` τ `b)`

where $\tau \in$ {`float`, `double`, `indigenous`} and $\gamma \in$ {<, <=, >, >=}.

8.8.4. `public static final int` **`TO_NEAREST`** `= 0;`

The constant value of this field represents round to nearest for `rounding` expressions and the `getRound` and `setRound` methods.

8.8.5. `public static final int` **`TO_ZERO`** `= 1;`

The constant value of this field represents round to zero for `rounding` expressions and the `getRound` and `setRound` methods.

8.8.6. `public static final int` **`TO_POSITIVE_INFINITY`** `= 2;`

The constant value of this field represents round to positive infinity for `rounding` expressions and the `getRound` and `setRound` methods.

8.8.7. `public static final int` **`TO_NEGATIVE_INFINITY`** `= 3;`

The constant value of this field represents round to negative infinity for `rounding` expressions and the `getRound` and `setRound` methods.

8.8.8. `public static int` **`getRound`**`()`

Returns the current dynamic rounding mode encoded using `TO_NEAREST`, `TO_ZERO`, `TO_NEGATIVE_INFINITY`, and `TO_POSITIVE_INFINITY`.

8.8.9. `public static void` **`setRound`**`(int rm) throws UnknownRoundingModeException`

Sets the dynamic rounding mode according to the value of the arguement. If the argument is not equal to `TO_NEAREST`, `TO_ZERO`, `TO_NEGATIVE_INFINITY`, nor `TO_POSITIVE_INFINITY`, an `UnknownRoundingModeException` is thrown.

8.8.10. `public static final int` **`NONE`** `= 0x0;`

The constant value of this field represents the empty set of flags for the various sticky flag manipulation methods.

8.8.11. `public static final int` **`INEXACT_FLAG`** `= 0x1;`

The constant value of this field represents the inexact flag for the various sticky flag manipulation methods.

8.8.12. `public static final int` **`DIVIDEbyZERO_FLAG`** `= 0x2;`

The constant value of this field represents the divide by zero flag for the various sticky flag manipulation methods.

8.8.13. `public static final int` **`UNDERFLOW_FLAG`** `= 0x4;`

The constant value of this field represents the underflow flag for the various sticky flag manipulation methods.

8.8.14. `public static final int` **`OVERFLOW_FLAG`** `= 0x8;`

The constant value of this field represents the overflow flag for the various sticky flag manipulation methods.

8.8.15. `public static final int INVALID_FLAG = 0x10;`

The constant value of this field represents the invalid flag for the various sticky flag manipulation methods.

8.8.16. `public static final int ALL = 0x1F;`

The constant value of this field represents the entire set of flags for the various sticky flag manipulation methods.

8.8.17. `public static void setFlag(int flags, boolean value)`

The sticky flags indicated by the `flags` argument are set to the value of the `value` argument; `false` meaning 0 or clear and `true` meaning 1 or set.

8.8.18. `public static void setFlags(int flagState)`

The current sticky flag state is replaced by the state represented by the `flagState` argument.

8.8.19. `public static int getFlags()`

Returns the current sticky flag state is as represented as an integer.

8.8.20. `public static indigenous E()`

This method returns the `indigenous` value closer than any other to *e*, the base of the natural logarithms. On a platform where `indigenous` is implemented with the `double` format, the result of the method `E` is equal to the `double` class variable `Math.E`.

8.8.21. `public static indigenous PI()`

This method returns the `indigenous` value closer than any other to π, the ratio of the circumference of a circle to its diameter. On a platform where `indigenous` is implemented with the `double` format, the result of the method `PI` is equal to the `double` class variable `Math.PI`.

8.8.22. Transcendental functions

```
public static indigenous sin(indigenous n)
public static indigenous cos(indigenous n)
public static indigenous tan(indigenous n)
public static indigenous asin(indigenous n)
public static indigenous acos(indigenous n)
public static indigenous atan(indigenous n)
public static indigenous atan2(indigenous n)
public static indigenous exp(indigenous n)
public static indigenous log(indigenous n)
public static indigenous pow(indigenous a, indigenous b)
```

8.8.23. `public static indigenous sqrt(indigenous n)`

This method computes an approximation to the square root of the argument.

Special cases:

- If the argument is NaN or less than zero, then the result is NaN
- If the argument is positive infinity, than the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is the same as the argument.

Otherwise, the result is the `indigenous` value closest to the true mathematical square root.

8.8.24. `public static indigenous IEEEremainder(indigenous x, indigenous y) admits none yields invalid`

This method computes the remainder operation on two arguments as prescribed by the IEEE 754 standard: the remainder value is mathematically equal to $x - y \times n$ where n is the mathematical integer closet to the exact mathematical value of the quotient *x* / *y*; if two mathematical integers are equally close to *x* / *y* then *n* is the integer that is even. If the remainder is zero, its sign is the same as the sign of the first argument.

Special cases:

- If either argument is NaN, or the first argument is infinite, or the second argument is positive zero or negative zero, then the result is NaN and invalid is signaled.
- If the first argument is finite and the second argument is infinite, then the result is the same as the first argument.

8.8.25. `public static indigenous` **`ceil`**`(indigenous n)`

The result is the smallest (closest to negative infinity) `indigenous` value that is not less than the argument and is equal to a mathematical integer.

Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive or negative zero, then the result is the same as the argument.
- If the argument is less than zero but greater than `-1.0`, then the result is negative zero.

8.8.26. `public static indigenous` **`floor`**`(indigenous n)`

The result is the largest (closest to positive infinity) `indigenous` value that is not greater than the argument and is equal to a mathematical integer.

Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive or negative zero, then the result is the same as the argument.

8.8.27. `public static indigenous` **`rint`**`(indigenous n)`

The result is the `indigenous` value that is closest in value to the argument and is equal to a mathematical integer. If two `indigenous` values that are mathematical integers are equally close to the value of the argument, the result is the integer value that is even.

Special cases:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NaN or an infinity or positive or negative zero, then the result is the same as the argument.

8.8.28. `public static long` **`round`**`(indigenous n)`

The result is rounded to an integer by adding 1/2, taking the floor of the result, and casting the result to type `long`.

In other words, the result is equal to the value of the expression:

```
(long) Math.floor(n + 0.5n)
```

Special cases:

- If the argument is NaN, the result is 0.
- If the argument is negative infinity, or indeed any value less than or equal to the value of `Long.MIN_VALUE`, the result is equal to the value of `Long.MIN_VALUE`.
- If the argument is positive infinity, or indeed any value greater than or equal to the value of `Long.MAX_VALUE`, the result is equal to the value of `Long.MAX_VALUE`.

8.8.29. `public static indigenous` **`abs`**`(indigenous n)`

The argument is returned with its sign changed to be positive.

Special case:

- If the argument is positive zero of negative zero, the result is positive zero.
- If the argument is infinite, the result is positive infinity.
- If the argument is NaN, the result is NaN

8.8.30. `public static indigenous` **`min`**`(indigenous a, indigenous b)`

The result is the smaller of the two arguments—that is, the one closer to negative infinity. If the arguments have the same value, the result is that same value.

Special cases:

- If one of the arguments is positive zero and the other is negative zero, the result is negative zero.
- If either argument is a NaN, the result is NaN.

8.8.31. `public static indigenous max(indigenous a, indigenous b)`

The result is the larger of the two arguments—that is, the one closer to positive infinity. If the arguments have the same value, the result is that same value.

Special cases:

- If one of the arguments is positive zero and the other is negative zero, the result is positive zero.
- If either argument is a NaN, the result is NaN.

8.8.32. `public static float fmac(float a, float b, float c) admits none yields overflow, underflow, inexact, invalid`

The result is equal to the product of `a` and `b` calculated to infinite precision, added to `c` and rounded to `float`.

If the final answer cannot be represented exactly the inexact flag is set. Tininess and loss of accuracy (defined in IEEE 754) are necessary for the underflow flag to be raised. The overflow flag is raised when the rounded `float` result would be larger in magnitude than `Float.MAX_VALUE`, in which case an appropriately signed infinity is returned.

The invalid flag is set when one of `a` and `b` is infinite and the other is zero. If the product of `a` and `b` is infinite and `c` is an opposite signed infinity, the invalid flag is also set. In both cases a NaN is returned.

8.8.33. `public static double fmac(double a, double b, double c) admits none yields overflow, underflow, inexact, invalid`

The result is equal to the product of `a` and `b` calculated to infinite precision, added to `c` and rounded to `double`.

If the final answer cannot be represented exactly the inexact flag is set. Tininess and loss of accuracy (defined in IEEE 754) are necessary for the underflow flag to be raised. The overflow flag is raised when the rounded `double` result would be larger in magnitude than `Double.MAX_VALUE`, in which case an appropriately signed infinity is returned.

The invalid flag is set when one of `a` and `b` is infinite and the other is zero. If the product of `a` and `b` is infinite and `c` is an opposite signed infinity, the invalid flag is also set. In both cases a NaN is returned.

8.8.34. `public static indigenous fmac(indigenous a, indigenous b, indigenous c) admits none yields overflow, underflow, inexact, invalid`

The result is equal to the product of `a` and `b` calculated to infinite precision, added to `c` and rounded to `indigenous`.

If the final answer cannot be represented exactly the inexact flag is set. Tininess and loss of accuracy (defined in IEEE 754) are necessary for the underflow flag to be raised. The overflow flag is raised when the rounded `indigenous` result would be larger in magnitude than `Indigenous.MAX_VALUE`, in which case an appropriately signed infinity is returned.

The invalid flag is set when one of `a` and `b` is infinite and the other is zero. If the product of `a` and `b` is infinite and `c` is an opposite signed infinity, the invalid flag is also set. In both cases a NaN is returned.

8.8.35. `public static float copySign(float value, float sign) admits none yields none`

Returns the first floating point argument with the sign of the second floating point argument. If either argument is a NaN, NaN is returned.

8.8.36. `public static double copySign(double value, double sign) admits none yields none`

Returns the first floating point argument with the sign of the second floating point argument. If either argument is a

NaN, NaN is returned.

8.8.37. `public static indigenous` **`copySign`**`(indigenous value, indigenous sign)` `admits none yields none`

Returns the first floating point argument with the sign of the second floating point argument. If either argument is a NaN, NaN is returned.

8.8.38. `public static float` **`scalb`**`(float value, int scale_exponent)` `admits none yields overflow, underflow, inexact`

Return `value * 2`$^{\texttt{scale\_exponent}}$. If the exponent of the result is between the E_{min} and E_{max} for `float`, the answer is calculated exactly and no signal is generated.
Special cases:

- If `value` is a NaN the result is a NaN.
- If `value` is an infinity, the returned value is an infinity with the same sign.
- If the exponent of the infinitely precise result is larger than E_{max}, infinity is returned and overflow and inexact are signaled
- If the result is a subnormal number, inexact and underflow are signaled if tininess and loss of accuracy occur.

8.8.39. `public static double` **`scalb`**`(double value, int scale_exponent)` `admits none yields overflow, underflow, inexact`

Return `value * 2`$^{\texttt{scale\_exponent}}$. If the exponent of the result is between the E_{min} and E_{max} for `double`, the answer is calculated exactly and no signal is generated.
Special cases:

- If `value` is a NaN the result is a NaN.
- If `value` is an infinity, the returned value is an infinity with the same sign.
- If the exponent of the infinitely precise result is larger than E_{max}, infinity is returned and overflow and inexact are signaled
- If the result is a subnormal number, inexact and underflow are signaled if tininess and loss of accuracy occur.

8.8.40. `public static indigenous` **`scalb`**`(indigenous value, int scale_exponent)` `admits none yields overflow, underflow, inexact`

Return `value * 2`$^{\texttt{scale\_exponent}}$. If the exponent of the result is between the E_{min} and E_{max} for `indigenous`, the answer is calculated exactly and no signal is generated.
Special cases:

- If `value` is a NaN the result is a NaN.
- If `value` is an infinity, the returned value is an infinity with the same sign.
- If the exponent of the infinitely precise result is larger than E_{max}, infinity is returned and overflow and inexact are signaled
- If the result is a subnormal number, inexact and underflow are signaled if tininess and loss of accuracy occur.

8.8.41. `public static float` **`logb754`**`(float value)` `admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the reuslt is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, $E_{min} - 1$ (in accord with IEEE 754).

8.8.42. `public static double logb754(double value) admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the reuslt is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, $E_{min} - 1$ (in accord with IEEE 754).

8.8.43. `public static indigenous logb754(indigenous value) admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the reuslt is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, $E_{min} - 1$ (in accord with IEEE 754).

8.8.44. `public static float logb(float value) admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the reuslt is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, E_{min} is returned (as specified in IEEE 854). This allows subnormals to be indentified as when `scalb(x, -logb(x))` is less than 1.0 in magnitude.

8.8.45. `public static double logb(double value) admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the reuslt is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, E_{min} is returned (as specified in IEEE 854). This allows subnormals to be indentified as when `scalb(x, -logb(x))` is less than 1.0 in magnitude.

8.8.46. `public static indigenous logb(indigenous value) admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the reuslt is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, E_{min} is returned (as specified in IEEE 854). This allows subnormals to be indentified as when `scalb(x, -logb(x))` is less than 1.0 in magnitude.

8.8.47. `public static float logbn(float value)` `admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the result is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, returns an exponent as if the number were normalized.

8.8.48. `public static double logbn(double value)` `admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the result is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, returns an exponent as if the number were normalized.

8.8.49. `public static indigenous logbn(indigenous value)` `admits none yields divideByZero`

Returns the unbiased exponent of `value`.
Special cases:

- If `value` is a NaN the result is a NaN
- If `value` is infinite the result is +infinity.
- If `value` is zero the result is –infinity and the divide by zero flag is set.
- If `value` is subnormal, returns an exponent as if the number were normalized.

8.8.50. `public static float nextAfter(float base, indigenous direction)` `admits none yields overflow, underflow, inexact`

Returns the floating point number adjacent to `base` in the direction of the `direction`. When the result is subnormal, underflow and inexact are signaled. If the result is infinity from a finite `base`, overflow and inexact are signaled. If both arguments are equal, the first argument is returned (this preserves the sign of zero appropriately). Except possibly when `base` is zero, the sign of the result is the same as the sign of `base`. The direction parameter is `indigenous` so that `base` can be perturbed in relation to any floating point value.

8.8.51. `public static double nextAfter(double base, indigenous direction)` `admits none yields overflow, underflow, inexact`

Returns the floating point number adjacent to `base` in the direction of the `direction`. When the result is subnormal, underflow and inexact are signaled. If the result is infinity from a finite `base`, overflow and inexact are signaled. If both arguments are equal, the first argument is returned (this preserves the sign of zero appropriately). Except possibly when `base` is zero, the sign of the result is the same as the sign of `base`. The direction parameter is `indigenous` so that `base` can be perturbed in relation to any floating point value.

8.8.52. `public static indigenous nextAfter(indigenous base, indigenous direction)` `admits none yields overflow, underflow, inexact`

Returns the floating point number adjacent to `base` in the direction of the `direction`. When the result is subnormal, underflow and inexact are signaled. If the result is infinity from a finite `base`, overflow and inexact are signaled. If both arguments are equal, the first argument is returned (this preserves the sign of zero appropriately). Except possibly when `base` is zero, the sign of the result is the same as the sign of `base`.

8.8.53. `public static boolean unordered(float comparand1, float comparand2) admits none yields none`

Returns `true` if and only if the unordered relation holds between the two floating point arguments. For the unordered relation to be `true`, at least one argument must be a NaN.

8.8.54. `public static boolean unordered(double comparand1, double comparand2) admits none yields none`

Returns `true` if and only if the unordered relation holds between the two floating point arguments. For the unordered relation to be `true`, at least one argument must be a NaN.

8.8.55. `public static boolean unordered(indigenous comparand1, indigenous comparand2) admits none yields none`

Returns `true` if and only if the unordered relation holds between the two floating point arguments. For the unordered relation to be `true`, at least one argument must be a NaN.

8.8.56. `public static final int FP_NAN = 0;`

The constant value of this field is the result of `Math.fpClass` when given a NaN argument.

8.8.57. `public static final int FP_NEGATIVE_INFINITY = -4;`

The constant value of this field is the result of `Math.fpClass` when given an argument equal to `NEGATIVE_INFINITY` of the appropriate type.

8.8.58. `public static final int FP_NEGATIVE_NORMAL = -3;`

The constant value of this field is the result of `Math.fpClass` when given an argument that is a negative nonzero normalized value of the appropriate type.

8.8.59. `public static final int FP_NEGATIVE_SUBNORMAL = -2;`

The constant value of this field is the result of `Math.fpClass` when given an argument that is a negative nonzero subnormal value of the appropriate type.

8.8.60. `public static final int FP_NEGATIVE_ZERO = -1;`

The constant value of this field is the result of `Math.fpClass` when given an argument that is a negative zero of the appropriate type.

8.8.61. `public static final int FP_POSITIVE_ZERO = 1;`

The constant value of this field is the result of `Math.fpClass` when given an argument that is a positive zero of the appropriate type.

8.8.62. `public static final int FP_POSITIVE_SUBNORMAL = 2;`

The constant value of this field is the result of `Math.fpClass` when given an argument that is a positive nonzero subnormal value of the appropriate type.

8.8.63. `public static final int FP_POSITIVE _NORMAL = 3;`

The constant value of this field is the result of `Math.fpClass` when given an argument that is a positive nonzero normalized value of the appropriate type.

8.8.64. `public static final int` **`FP_POSITIVE_INFINITY`** `= 4;`

The constant value of this field is the result of `Math.fpClass` when given an argument equal to `POSITIVE_INFINITY` of the appropriate type.

8.8.65. `public static int` **`fpClass`**`(float value) admits none yields none`

Returns the classification of a floating point number according to the `FP_*` fields. The sign of the returned value is the same as the sign of the argument; 0 is returned for NaN. The magnitude of the values returned by `fpClass` for different kinds of floating point numbers is given in Table 37.

Table 37 — Behavior of `fpClass`.

Kind of floating point number	Absolute value of result of `fpClass`
infinite	4
normal	3
subnormal	2
zero	1

8.8.66. `public static int` **`fpClass`**`(double value) admits none yields none`

Returns the classification of a floating point number according to the `FP_*` fields. The sign of the returned value is the same as the sign of the argument; 0 is returned for NaN. The magnitude of the values returned by `fpClass` for different kinds of floating point numbers is given in Table 37.

8.8.67. `public static int` **`fpClass`**`(indigenous value) admits none yields none`

Returns the classification of a floating point number according to the `FP_*` fields. The sign of the returned value is the same as the sign of the argument; 0 is returned for NaN. The magnitude of the values returned by `fpClass` for different kinds of floating point numbers is given in Table 37.

8.9. Changes to `java.lang.String`

8.9.1. `public static String` **`valueOf`**`(indigenous n)`

A string is created and returned. The string is computed exactly as if by the method `Indigenous.toString` of one argument.

8.10. Changes to `java.lang.StringBuffer`

8.10.1. `public StringBuffer` **`append`**`(indigenous n)`

The argument is converted to a string as if by the method `String.valueOf` and the characters of that string are then appended to this `StringBuffer` object. A reference to this `StringBuffer` object is returned.

8.10.2. `public StringBuffer` **`insert`**`(int offset, indigenous n) throws IndexOutOfBoundsException`

The argument is converted to a string as if by the method `String.valueOf` and the characters of that string are then inserted into this `StringBuffer` object at the position indicated by `offset`. A reference to this `StringBuffer` object is returned.

8.11. Changes to `java.lang.System`

As listed in Table 5, Borneo has two new floating point related systems properties not found in Java.

8.12. New Subclasses of `java.lang.Exception`

Borneo adds a number of new exception classes to Java. The `UnknownRoundingModeException` class is a direct subclass of `RuntimeException` and is therefore an unchecked exception. `UnknownRoundingModeException` is thrown by `rounding` declarations and `Math.setRound` when an invalid rounding mode is used.

As shown in Figure 19, Borneo adds many checked exception classes to model IEEE 754 exceptional conditions. Of these, `OverflowException` and `UnderflowException` have additional constructors and methods (Figure 65). The new constructors take `float`, `double`, and `indigenous` arguments that are returned, respectively, when the `floatValue`, `doubleValue`, and `indigenousValue` methods are called. The `float`, `double`, and `indigenous` values held in floating point overflow and underflow exceptions generated as a result of an arithmetic operation have the relationships described in Table 9. A programmer can create overflow/underflow exceptions with any combination of `float`, `double`, and `indigenous` values.

```
public class OverflowException extends FloatingPointException
  {
    private float f;
    private double d;
    private indigenous n;

    public OverflowException()
      {
        super();
        n = d = f = 0.0f;
      }
    public OverflowException(float f, double d, indigenous n)
      {
        super();
        this.f = f; this.d = d; this.n = n;
      }
    public OverflowException(String message)
      {
        super(message);
        n = d = f = 0.0f;
      }
    public OverflowException(String message, float f, double d, indigenous n)
      {
        super(message);
        this.f = f; this.d = d; this.n = n;
      }

    public float floatValue(){return f};
    public double doubleValue(){return d};
    public indigenous indigenousValue(){return n};
  }

public class UnderflowException extends FloatingPointException
  {
    private float f;
    private double d;
    private indigenous n;

    public OverflowException()
      {
        super();
        n = d = f = 0.0f;
      }
    public OverflowException(float f, double d, indigenous n)
      {
        super();
        this.f = f; this.d = d; this.n = n;
      }
    public OverflowException(String message)
      {
        super(message);
        n = d = f = 0.0f;
      }
    public OverflowException(String message, float f, double d, indigenous n)
      {
        super(message);
        this.f = f; this.d = d; this.n = n;
      }

    public float floatValue(){return f};
    public double doubleValue(){return d};
    public indigenous indigenousValue(){return n};
  }
```

Figure 65 — New constructors and methods for `OverflowException` and `UnderflowException`.

9. Appendixes

9.1. Field Axioms

Arithmetic on real numbers forms a field. Fields include the properties of rings; rings in turn are an extension of quasirings. Many local optimizations compilers perform are valid because they are derived from the field axioms. For example, for all integers values i, $i + 0 = i$. However, IEEE floating point numbers do not form even a quasiring. As Table 38 shows, very few of the field axioms hold for IEEE arithmetic. For two expressions to be equivalent, they must have identical values (same sign bit, significand, and exponent) and raise the same exceptions. The field axioms can fail due to limited range, limited precision, or because of IEEE 754 special values. Table 38 assumes the round to nearest rounding mode is in effect, the signaling properties of signaling NaNs are ignored, and that all NaN values are regarded as equivalent. If signaling NaNs are included, multiplicative identity no longer holds. If different NaN bit patterns are differentiated, multiplication and addition are no longer (necessarily) commutative.

The following abbreviations are used in Table 38:

Ω = nextAfter(∞, 0) *//The largest finite value*

δ = nextAfter(ROUNDING_THRESHOLD, 0.0) *//The largest positive number less than the rounding threshold*

α = {finite floating point numbers}

Table 38 — Field axiom validity for IEEE 754 floating point arithmetic.

Field Axiom over Reals	Example	Status of IEEE floating point
Quasiring Properties		
Closed under addition		*true*, if ∞ and NaN are included
Associative addition	$(a+b) + c = a + (b+c)$	• $(\Omega + \Omega) + -\Omega = \infty \not\equiv \Omega + (\Omega + -\Omega) = \Omega$ • $(1.0 + \delta) + \delta = 1.0 \not\equiv 1.0 + (\delta + \delta) > 1.0$ • $(\infty + -\infty)$+NaN signals invalid, $\infty+(-\infty$ +NaN) does not
Identify element for addition	$\forall a$, $a + 0 = 0 + a = a$	*false* if a is -0, $a + (+0) = +0$, $+0$ is distinguishable from -0
Closed under multiplication		*true*, if ∞ and NaN are included
Associative multiplication	$(a*b)*c = a*(b*c)$	*false* • roundoff and loss of precision on underflow • $(\Omega * 2.0) * (0.5) = \infty \not\equiv \Omega * (2.0 * 0.5) = \Omega$ • $(\infty*0.0)$*NaN signals invalid, $\infty*(0.0*$NaN) does not
Identity element for multiplication	$\forall a$, $a*1 = 1*a = a$	*true*
Zero annihilator	$\forall a$, $a*0 = 0*a = 0$	*false* • 0 * NaN and NaN * 0 are NaN, NaN is not 0 • $0 * \infty$ and $\infty * 0$ are NaN, NaN is not 0
Commutative addition	$\forall a\ \forall b$, $a + b = b + a$	*true*
Distributivity	$a*(b + c) = a*b + a*c$ and $(b + c)*a = b*c + c*a$	*false* • roundoff • overflow/underflow thresholds • differences signaling invalid with ∞, NaN, and 0.0
Ring Properties		
Additive inverse	$\forall a\ \exists b$, $a + b = b + a = 0$	*false* • $\infty + -\infty$ is NaN, NaN is not 0; $\infty + \alpha = \infty$, $\infty \not\equiv 0$ • NaN + α is NaN and NaN + ∞ is NaN, NaN is not 0
Field Properties		
Commutative multiplication	$\forall a\ \forall b$, $a*b = b*a$	*true*
Multiplication inverse	except for a=0, $\forall a\ \exists b$, $a*b = b*a = 1$	*false* • many ordinary floating point values lack exact inverses • $\infty * 0$ is NaN, ∞ * NaN is NaN and $\forall\ \alpha \neq 0$, $\infty * \alpha = \pm\infty$

9.2. Redundant JVM Instructions

Table 39 — 47 truly redundant JVM opcodes.

Redundant Opcode	Equivalent Sequence
aload_0, aload_1, aload_2, aload_3	*aload* n
astore_0, astore_1, astore_2, astore_3	*astore* n
dload_0, dload_1, dload_2, dload_3	*dload* n
dstore_0, dstore_1, dstore_2, dstore_3	*dstore* n
fload_0, fload_1, fload_2, fload_3	*fload* n
fstore_0, fstore_1, fstore_2, fstore_3	*fstore* n
iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5	*bipush* n
iload_0, iload_1, iload_2, iload_3	*iload* n
istore_0, istore_1, istore_2, istore_3	*istore* n
lload_0, lload_1, lload_2, lload_3	*lload* n
lstore_0, lstore_1, lstore_2, lstore_3	*lstore* n

Table 40 — JVM bytecodes which could be expressed in terms of other opcodes with some modification to other portions of the `class` file.

Unnecessary Opcode	Equivalent Sequence
dconst_0, dconst_1	*bipush* n, *i2d*
f_const_0, f_const_1, fconst_2	*bipush* n, *i2f*
lconst_0, lconst_1	*bipush* n, *i2l*
dneg	*bipush* 0, *bipush* 1, *isub, i2d, dmul*
fneg	*bipush* 0, *bipush* 1, *isub, i2f, fmul*
ineg	*bipush* 0, *swap, isub*
lneg	*bipush* 0, *i2l, dup2_x2, pop2, lsub*
dup instructions *dup_x1, dup_x2, dup2_x1, dup2_x2*	More complicated *dup*'s can be replaced by sequences of simple *dup*'s and loads and stores to extra local variables
ifnonull/ifnull	Only one of these instructions is needed, the order of branches can be reversed instead.
iinc m n	*iload* m, *bipush* n, *iadd, istore* m

The 11 rows of Table 39 contain 47 obviously redundant opcodes that could be replaced by their more general counterparts. The redundant opcodes were included to save space and execution time by making some common instruction sequences fit into a single byte. The opcodes in Table 40 can also be replaced by an equivalent sequence of other JVM instructions, but at the cost of increasing stack utilization, requiring more local variables, or performing additional runtime conversions between numeric types. Since there are approximately 200 instructions defined in JVM (excluding the *_quick* pseudo-instruction variants), nearly one quarter of the opcode space is devoted to peephole optimizations while precious few free opcodes remain.

A simple translator could be created to purge existing `class` files of the 47 opcodes in Table 39, freeing those opcodes for other uses. The translator must properly update all control transfer targets in the code portion of the `class` file. Except for the *ret* instruction, all JVM control transfers take their target from the `class` file; a branch target cannot be computed at runtime. The *ret* instruction takes its target from a `returnAddress` value stored in a local variable. However, a return address can only be generated by a *jsr* or *jsr_w* instruction pushing a return address on the stack (the *astore* instructions are used to store a `returnAddress` in a local variable). Since no computation can be done on a `returnAddress` value, the translator does not need to modify the uses of *jsr* and *ret* instructions. When the new `class` file is generated, the padding for the *lookupswitch* and *tableswitch* instructions may need to be changed. Besides changing the code attribute of the `class` file, such a utility also needs to modify the exceptions table to update the code offsets handled by different `catch` blocks. The largest offset of the code attribute that can be covered by an exception table is 65,534 bytes. Since the translator lengthens the code attribute, it may not be possible to directly convert a very long code attribute to one that does not use the redundant instructions. However, such a long method could be split into parts.

9.3. A Brief History of Programming Language Support for Floating Point Computation

In roughly chronological order, the following sections discuss the floating point support provided by a number of programming languages. Computational peculiarities of particular languages are also described.

9.3.1. FORTRAN

9.3.1.1. Early FORTRAN

[Six months] was to remain the interval-to-completion until it was actually completed over two years later.
—John Bachus on the first FORTRAN compiler

The purpose of the original FORTRAN project, started in approximately January 1954, was to provide a practical "automatic programming system" for the new IBM 704 computer. The 704 is a member of the first generation of commercial computers to have index registers and floating point arithmetic in hardware. On earlier machines, interpreters for virtual architectures provided index registers and floating point, slowing the real machine down by a factor of five to ten. FORTRAN aimed to generate code for scientific programs that were nearly as efficient as hand generated assembly, a task made more challenging by the improved hardware capabilities of the 704; there was no longer an order of magnitude slowdown to hide behind [7]!

In a preliminary report late in 1954, FORTRAN has an unusual rule for mixed integer-floating point expression evaluation: the arithmetic used to evaluate an expression is either all integer or all floating point, determined by the type of the variable being assigned to. The preliminary report also proposes new facilities to be added to the language, including complex and double precision arithmetic. Parentheses are used to help the compiler perform common subexpression elimination [7].

By the release of the programmer's reference manual for the completed FORTRAN I language, the language designers felt mixed integer-floating point expressions should not have implicit type coercions generated by the compiler. Only integer exponents and inter function arguments are allowed in expressions having other floating point components [7].

9.3.1.2. FORTRAN II

FORTRAN II is an evolution of earlier FORTRAN dialects. Besides subroutines and improved debugging, FORTRAN II has a different treatment of arithmetic than earlier dialects. Integer to floating point and floating point to integer conversions occur across assignment statements, but mixed expressions are still quite restricted. When converting floating point to integer, round to zero is used; library methods also provide explicit conversion capabilities. By a naming convention, single precision floating point variables can be used (the names of integer variables start with the letters I, J, K, L, M, or N; variables starting with other letters are floating point). To use double precision numbers for a widest available style expression evaluation, a "D" can be placed in the first card column (punch cards were used to input programs to the computer). However, even if a double variable was being stored into, integer to floating point conversion would only occur over the single precision range. Only single precision decimal floating point numbers are read in or printed out; to preserve full double precision binary or octal input and output must be used. Language primitives are provided to test for various overflow conditions. While requiring parentheses to be respected, FORTRAN II assumes that "*mathematically* equivalent expressions are computationally equivalent." Instead of warning programmers of the vagaries of floating point computation, the FORTRAN II manual discusses the problems of integer arithmetic! "Although the assumption concerning mathematical and computation equivalence is virtually true for floating point expressions, special care must be taken to indicate the order of fixed point multiplication and division..." [49].

9.3.1.3. FORTRAN IV

Coming a few years after FORTRAN II, FORTRAN IV continued FORTRAN's development and refinement. Variables can be explicitly declared instead of relying on the naming convention. Both single and double precision are available; single values are promoted to double in mixed-mode expressions using strict evaluation. Implicit conversions between integer and floating point in expressions do not occur. More extensive library support is provided for the double type than in FORTRAN II. FORTRAN IV preserves the FORTRAN II assumption that "mathematically equivalent expressions are computationally equivalent." When a floating point literal found in the source program and a string representing a floating point number read in at runtime are converted to binary, “there may be a difference in the low-order bits of the same constant arising from these two sources” [50].

9.3.1.4. FORTRAN 77

FORTRAN 77 is a replacement for FORTRAN 66, which is in turn based on FORTRAN IV. The FORTRAN 77 standard [3] describes the behavior of legal programs; a given implementation is free to provide extensions to the standard, including adding new features and removing limitations. For example, while a standard FORTRAN 77 program cannot have a statement with more than 1320 characters, a FORTRAN 77 compiler has various options for dealing with such a statement: report an error, accept and compile correctly, or even issue no warning and compile such a long statement incorrectly. FORTRAN 77’s major changes from FORTRAN 66 include a character data type and additional control flow constructs.

FORTRAN 77 does not give requirements for decimal to binary conversion. Two base floating point types, single and double, are supported although “the range or precision of numeric quantities and the method of rounding of numeric results” is not specified beyond that double must have more precision than single. A complex type with single precision real and imaginary portions is also included. Signed zeros are not supported in the FORTRAN 77 standard; if two zeros are present they must be treated identically. Even if represented internally, a negative zero must be printed without a minus sign [3].

Strict evaluation is used for numeric expressions. If an operation has mixed integer and floating point operands, the integer operand is converted to the floating point type. Implicit coercions between integer and floating point types also occur across assignment statements; assigning a floating point value to an integer rounds toward zero. Various library functions return floating point numbers rounded to integer in various ways; `INT` rounds to zero and `NINT` rounds to nearest [3].

While FORTRAN 77 requires that explicit parentheses be respected, the compiler has some leeway to evaluate “any mathematically equivalent arithmetic expression.” Explicit license is given to use commutativity of addition and multiplication, associativity of addition, and distributivity of multiplication over subtraction. Intermediate expressions may have implementation-dependent types depending on the expression evaluation used by the compiler [3].

9.3.2. APL

The design of APL (A Programming Language) began around 1956. Initially created for data processing tasks, APL was influenced by mathematical notation and has numerous built-in operators acting on both arrays and scalar values. There is no operator precedence; expressions are evaluated from left to right. Implementation was not seriously attempted until 1964. Unlike early FORTRAN, optimal speed was not the primary implementation goal. Flexibility for language experimentation and limiting concessions to machine-specific considerations were more important [29].

Both integer and floating point numbers are used in APL. When values exceed the range of integers, floating point numbers are automatically introduced. Arbitrary precision arithmetic is not provided, so to allow comparisons of very close numbers to return true, a comparison tolerance, or “fuzz” was introduced into the language [29]. For example, after executing

$$Y \leftarrow 2 \div 3 \text{ and } X \leftarrow 3 \times Y$$

the intention is to allow $2 = X$ to be true even though X is not exactly equal to 2 [29]. In general, APL’s operators are designed to preserve various identities [28]. However, these fuzzy comparison semantics break many desirable properties of floating point comparison. For example, with a fuzzy comparison, floating point equality is not transitive. Consider three numbers A, B, and C where B is A + *comparison tolerance* and C is B + *comparison tolerance*. $A = B$ and $B = C$ are true, but $A = C$ is false since the difference between A and C is greater than the comparison tolerance. At first the comparison tolerance was fixed at 10^{-13} but was later adjustable. Setting the comparison tolerance to zero yields exact comparisons.

APL defines the value of $0 \div 0$ to be 1. This definition preserves the identity $X / X = 1$ for all values of X. However, this definition breaks other equally valid and useful identities such as $0 / X = 0$ [71]. In IEEE arithmetic, 0/0 is a NaN.

The *system variables* of APL provide an interface to various aspects of the APL environment and the underlying processor. Besides setting the comparison tolerance, system variables can also define *latent expressions* to be executed when exceptional conditions occur [30]. System variables can be localized in APL, giving them copy-on-write semantics. A program, as it is invoked, can set a system variable and the variable's previous value is restored when the program exits.

9.3.3. ALGOL 60

Here is a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors.
—C. A. Hoare on ALGOL 60

ALGOL 60 is a watershed language, introducing many precedents and structures still visible in languages over thirty years later. Designed by a multi-national committee, ALGOL 60's goals include supporting numeric processing. The ALGOL 60 report [75] does not have accuracy requirements for decimal to binary conversion stating, "the actual numerical value of a primary is obvious in the case of numbers." However, the hardware variability of floating point is acknowledged. There is a single `real` type; an `integer` value is converted to `real` in mixed mode expression evaluation. The transfer function from `real` to `integer`, `entier`, rounds toward zero. If a expression of type `real` is assigned to a variable of `integer` type, the real value is rounded to nearest [75].

9.3.4. Algol 68

Developed several years after Algol 60, the design of Algol 68 stresses orthogonality and extensibility. Algol 68 supports the creation of user defined data types and operators. Algol 68 has a family of `real` types: `real`, `long real`, `long long real`, and so on. However, the sizes and precisions of these types is implementation dependent. The functions `round` and `entier` are provided to convert `real` to `integer` by rounding to nearest and truncating, respectively. Integers are automatically widened to `real` (and $\texttt{long}^{n}$ `real` to $\texttt{long}^{n+1}$ `real`) in assignment statements and parameter passing. The standard arithmetic operators are included via a general purpose operator mechanism and are not a special language construction. In addition to defining operators for user-defined types, programmers can redefine existing operators, such as making + on integers execute subtraction. Ten operator precedence levels are available and the precedence of built-in operators can also be modified so that 1+2*3 evaluates to 9 instead of 7 [92].

9.3.5. C

In the early 1970's, C was created to be the systems programming language for the UNIX operating system. Therefore, C is a fairly low level language designed to expose system dependent details and generate fast executables. C was ported along with UNIX to a wide variety of platforms with various floating point formats. In 1983, an ANSI (American National Standards Institute) committee was formed to standardize the C language. Both versions of C are discussed.

9.3.5.1. Pre-ANSI C

No constraints are given for base conversion. Early C supports two floating point data types, `float` for single precision and `double` for double precision numbers. Pre-ANSI C uses the `double` type for all floating point expression evaluation; all floating point literals are also of type `double`. Floating point to integer conversion is truncated. Across assignment statements, both implicit floating point to integer and integer to floating point conversions may occur depending on the type of the variable or array location being assigned to. However, automatic coercions do not occur for function arguments; explicit casts are required. C allows wide compiler latitude in expression evaluation: "the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects" [64]. The grouping of parentheses does not need to be respected; the compiler is allowed to treat mathematically associative operations (such as multiplication) as computationally associative. To force a particular evaluation order, the intermediate results must be stored into explicit temporaries.

9.3.5.2. ANSI C

Among other changes to the language, ANSI C modifies the floating point expression evaluation rules. Instead of using `double` for widest available expression evaluation, ANSI C uses strict evaluation. A new type, `long double` is introduced. Suffixes are added to floating point numbers to create `float` and `long double` numeric literals. Automatic integer to floating point conversion also occurs for function arguments in ANSI C. The compiler can no longer treat mathematically associative operations as computationally associative and explicit parentheses must be respected. In both flavors of C, floating point exception handling is implementation defined [65].

9.3.6. Pascal and Modula-2

I have made this letter longer than usual, because I lack the time to make it short.
—Blaise Pascal

The programming language Pascal, designed by Niklaus Wirth around 1970, was a relativity simple alternative to the growing complexity of other languages in the Algol family, such as Algol 68. Pascal includes many facilities to create new user defined types, including subrange types, nested structures, and unions.

Pascal has one floating point type, `Real`, an "implementation-defined subset of real numbers" [53]. Other built-in types in Pascal are ordinal types; these types can be used in a variety of contexts where `Real` cannot. For example, ordinal types have `prec` and `succ` functions which provide the next smaller and next larger value. These functions are not provided for the `Real` type, even though floating point numbers do have a well-defined concept of next larger and next smaller value. In IEEE arithmetic, the functionality of `prec` and `succ` for floating point numbers is provided by the recommended function `nextafter`. Also, unlike for the `Integer` type, Pascal does not support declaring subranges of `Real`.

Decimal to binary conversion in Pascal does not explicitly require correct rounding, only that the value denoted by a character sequence be assigned to a variable. However, a decimal string may denote a value not representable in binary. No explicit requirements are given for correctly rounded binary to decimal conversion either [53].

Since Pascal only has one floating point type, all floating point operations are performed in that type. For arithmetic operands, an implicit `Integer` to `Real` conversion occurs if one of the operands is of type `Real`. The same coercion occurs if an `Integer` argument is given for a `Real` value parameter. `Integer` values can be assigned to `Real` variables without an explicit coercion. The standard function `Trunc` takes a `Real` and returns an `Integer` rounded to zero while `Round` takes a `Real` value and returns an `Integer` rounded to nearest.

Approximately ten years after designing Pascal, Niklaus Wirth designed Modula-2 [100] based on experience from Pascal and Modula. Modula-2 has many features absent in Pascal, such as modules and separate compilation, but the floating point support is quite similar. Base conversion is implementation dependent. Only a single implementation dependent floating point type `REAL` is required. Modula-2 removes Pascal's implicit integer to floating point conversions. Instead, the transfer function `FLOAT` must be used to convert non-negative integers to `REAL`. `Round` is not included in Modula-2's standard library, although particular implementations can include such transfer functions that are not required by the standard. The document specifying Modula-2 warns of the non-associativity of floating point addition, suggesting "the correct way [to sum a series of terms] is evidently to start with the small terms."

9.3.7. Ada

Starting in 1974, in order to consolidate the large number of programming languages being used in embedded applications, the United States Department of Defense funded the specification and development of the Ada programming language [95]. Ada's design goals include support for managing large programs as well as interfacing to low-level hardware capabilities.

Ada does not specify any accuracy constraints for base conversion. The base floating point type in Ada is `FLOAT`. From `FLOAT`, other floating point types can be specified as subranges or as a floating point type with a minimum precision of some number of decimal digits. These derived types can use all the existing floating point literals and all built-in operators. Function names can be overloaded in Ada. Each derived floating point type does not need its own set of library functions; the Ada type system can determine the appropriate library function to call for a derived floating point type.

Ada floating point numbers follow the Brown model [9] of floating point numbers. The Brown model distinguishes between *model numbers*, which adhere to Brown's axioms, and machine numbers, which may not. Essentially, in order to represent a variety of floating point arithmetics with different computational peculiarities, the model numbers have reduced range and/or precision compared to the actual machine floating point numbers. By only considering well-behaved model numbers, the Brown model aims to allow programs to be proved portable across various floating point architectures. Unfortunately, Brown's model is not very useful for creating portable numerical programs.

Brown's model (or any other such universal model) describes a hypothetical machine that simultaneously exhibits the union of the arithmetic irregularities of *all* possible machines described by the model. Suppose a numerical analyst wants to prove a program works under Brown's model. If he is unable to construct a proof, the program might be scrutinized for a bug. If no bug is found, the numerical analyst may insert additional tests to try to strengthen the program against possible threats. However, the program may actually work on any given actual architecture, even though a general proof cannot be constructed showing the program works on all (possibly perverse) architectures. A well defined standard, such as IEEE 754, allows much stronger assumptions to be made about the arithmetic, easing proof construction [61].

Arithmetic operations in Ada can throw exceptions. The conditions OVERFLOW and DIVIDE_ERROR are collapsed into the single exception NUMERIC_ERROR since it is claimed "their distinction is rarely helpful for recovery purposes" [10]. Optimizations are allowed to change the result of arithmetic expressions. For example, expressions can be rewritten so that the modified version does not throw an exception the original expression would throw. Extra precision can be used in expression evaluation and, within some limits, expressions can be rewritten using associativity. Floating point to integer conversion rounds to nearest [95].

Ada has operator overloading. Many existing operators can be overloaded, but new operators cannot be introduced. Unlike other languages with overloading, Ada uses *result overloading* to determine which version of an operator or function to call. Instead of just using the types of the arguments to determine which function to call, result overloading also considers the return type of the function [95].

9.3.8. Common Lisp

Introduced in 1984, Common Lisp intended to span the functionality of several diverging Lisp dialects, providing a common, portable, expressive, and stable base for further development. The designers of Common Lisp were cognizant of IEEE 754 and the language provides extensive numerical primitives and specifies many details of the behavior of trigonometric functions on complex numbers [88].

Common Lisp has a variety of numerical data types. Besides arbitrary length integers and exact ratios of integers (together integers and ratios are rationals), Common Lisp has four floating point data types, `short-float`, `single-float`, `double-float`, and `long-float`. For each floating point type, the Common Lisp standard has recommended minimum precision and exponent size. Depending on the number of hardware floating point formats, various mappings of language type to hardware format are permitted. Complex numbers are also included; the real and imaginary parts of a Common Lisp complex number are not necessarily floating point numbers. For numeric operators, strict evaluation is used; if a rational and a floating point number are mixed, the rational is converted to the floating point type of the other operand. Mathematically associative operations can be carried out in any order, possibly affecting which automatic conversions occur (and what answer is delivered). Common Lisp has four functions to convert floating point values to integer, including rounding to $\pm\infty$, truncating to zero, and rounding to nearest [86].

While the Common Lisp specification includes formulas for the complex trigonometric functions, the implementor is repeated warned

> These formulae are mathematically correct, assuming completely accurate computation. They may be terrible methods for floating-point computation! Implementors should consult a good text on numerical analysis. [86]

A "floating point cookbook" [19] is suggested as being possibly useful for implementing the irrational and transcendental functions but [19] does not discuss functions on complex arguments. In [55] Kahan gives a consistent scheme for the behavior of complex elementary functions at "branch cuts."

9.3.9. C++

C++ [89] was initially designed in the early and mid 1980's as an extended version of C that included classes. Over the years, C++ has grown to include additional features such as templates, exception handling, and operator overloading. The floating point types, expression evaluation rules, and compiler generated conversions in C++ are the same as those in ANSI C. The C++ draft standard [102] includes headers and classes to determine information about the floating point capabilities of a machine, such as if IEEE 754 floating point is available. C++ throws no floating point exceptions.

Most of the built-in C++ operators can be overloaded, including the arithmetic, comparison, and array access operators. Programmers cannot define operators not already in the language. Both binary and unary operators can be overloaded; at least one argument must be a user-defined type. Operators taking a built-in type as the first argument cannot be a member function of a class.

9.3.10. ML

ML originally arose during the mid 1970's as a system for constructing logical proofs and eventually evolved into a full-fledged programming language. The exception mechanism of ML has served as a model for the exception mechanisms of a number of other languages, including C++.

ML does not place constraints on decimal to binary conversion. ML has a single floating point type and all conversions between integer and floating point numbers must be explicit. The `floor` function returns the integer value of a floating point number rounded toward negative infinity [72].

ML provides a rich operator overloading mechanism. Any identifier, either an alpha-numeric identifier or an identifier composed of symbols (#,$,!, etc.), can be treated as an infix operator. The user defined operators can be right or left associative and have one of 10 predefined precedence levels. The text of an operator does not carry any indication of its precedence. Therefore, since the precedence of operators is not known to the parser, abstract syntax trees cannot be easily built until after typechecking, somewhat complicating the compiler's front end [14].

9.3.11. Haskell

The Haskell language is a modern lazy functional language designed in the late 1980's. Haskell aims to provide a language suitable for teaching, research, and applications while also reducing unnecessary diversity in functional programming languages [79].

Haskell's numeric types are heavily influenced by Common Lisp and Scheme. The numeric types include arbitrary and fixed length integers, rational numbers, two floating point types (`Float` and `Double`), and complex. Explicit coercions are used to convert between numeric types.

The designers of Haskell are aware of but have not fully embraced IEEE 754 arithmetic: "Some, but not all, aspects of the IEEE standard floating point standard [*sic*] have been accounted for in the class `RealFloat`" [79]. Although not included in the final version, early drafts of Haskell version 1.4 have an optional library of new arithmetic operators to perform IEEE directed rounding.

The Haskell infix operator facilities are similar to those in ML. Any function may be treated as an infix operator by enclosing the function identifier in backquotes; infix operators can be treated as prefix functions by surrounding the operator in parentheses. There are ten precedence levels and operators can be declared to be right, left, or non-associative [79]. The standard infix operators are just predefined symbols and may be rebound. Functions may also be overloaded, unlike ML.

9.3.12. Matlab

Matlab (matrix laboratory) was originally written in the late 1970's to provide a convenient interface to the linear algebra algorithms from the LINPACK and EISPACK efforts. Over the years Matlab has become very widely used in academia and industry and now includes support for sparse matrices, graphics, and symbolic expressions [39].

The fundamental datatype in Matlab is a matrix whose elements are double precision real or complex numbers. Matlab provides an interpreted environment with extensive libraries to manipulate matrices in addition to a small imperative language and interfaces to other languages such as C and FORTRAN. No explicit requirements are given for correctly rounded base conversion. Functions are provided to round floating point values to integers under all four rounding modes (in case of a tie, the round to nearest function rounds out instead of IEEE 754-style round to nearest even) [39].

Since basic Matlab uses the floating point formats native to a given platform, numbers other than IEEE 754 `double`, such as the VAX D or G formats, may also be used. Matlab has functions that return the rounding threshold and the smallest and largest positive floating point numbers. In newer versions of Matlab, special variables hold NaN and infinity values. However, sticky flags are not supported beyond warning messages issued for some invalid operations. More recent Matlab releases also support variable precision arithmetic for symbolic computation. To control the amount of precision used for symbolic computation, a global `Digits` parameter can be set or the `vpa` function can be used to evaluate a single expression to a given precision without changing the global precision [39].

9.3.13. Mathematica

Since 1988 the Mathematica software system has featured arbitrarily precise arithmetic, symbolic expression manipulation, and graphical output. Mathematica symbolically manipulates expressions by applying a series of re-write rules until a fixed point is reached. To prevent infinite loops, the call stack depth can be limited. Programmers can declare their functions to be associative and commutative to control how expressions are simplified. The language also include pattern matching and lambda expressions [101].

Mathematica numeric types include arbitrarily large integers, exact integer ratios, arbitrarily high precision real numbers, and complex versions of the above. The Mathematica function `N[]` explicitly specifies the precision to use for a calculation; it can cause the available hardware floating point arithmetic to be used instead of arbitrarily high precision. Mathematica's number system includes infinities and `Indeterminate`, which is analogous to NaN. IEEE 754 style sticky flags are not supported. Mathematica has functions to round real numbers to integers in various ways. Correctly rounded base conversion is not explicitly required [101].

When operating on arbitrarily high precision real values, Mathematica uses a form of *significance arithmetic* to determine the precision of the result. Significance arithmetic is a scheme where unnormalized numbers are used to provide an indication of the accuracy of the number; the accuracy of a result is a function of the magnitudes and accuracies of the inputs. Since only the bits believed to be significant are stored, depending on the details of the implementation, significance arithmetic constrains the expressed uncertainty in the result to be 1 or ½ of a unit in the last place (ULP). A number in significance arithmetic can be thought of as representing an interval between the two adjacent numbers. The error inferred from the inputs to an operation may not be exactly expressible in a significance arithmetic result. Therefore, a policy is needed for returning either a result more or less precise than deserved. Given such a policy, a "folk theorem" states it is always possible to construct a sequence of n operations such that the width of the interval grows or shrinks faster than it should by a factor of $2^{n/2}$ [59]. For this reason, significance arithmetic cannot be entirely trustworthy.

9.3.14. SANE, Standard Apple Numeric Environment

SANE [6] (Standard Apple Numeric Environment) is an interface to IEEE 754 features defined and supported by Apple Computer, Inc. on various hardware platforms and in several programming languages. In [6], the interface to SANE is given in terms of a Pascal dialect, but C bindings are discussed as well.

SANE requires binary to decimal and decimal to binary conversion to meet or exceed the correctly rounded requirements in IEEE 754. SANE defines four datatypes, three floating point types corresponding to IEEE 754 formats (single, double, and double extended) and one integer type, `comp`, implemented with double extended arithmetic. For all SANE datatypes, expression evaluation uses the widest available strategy with double extended precision. Floating point constants are stored in extended format. Conversions from double extended to integer or `comp` are influenced by the dynamic rounding mode. For Apple's SANE Pascal, `INF` is a predefined constant for infinity.

SANE provides the IEEE recommended functions, but due to a historical accident the arguments of `copysign` are reversed. The SANE `logb` function follows the recommendations in IEEE 854 instead of IEEE 754. It is not explicitly stated whether or not `scalb` is sensitive to dynamic rounding modes.

Functions that get and set the rounding mode, get and set the rounding precision, get and set the sticky flags, and get and set the trapping status are included in SANE. If a condition is being trapped on, the computation presumably aborts; SANE does not have a mechanism for providing user-defined trap handlers. Floating point state is treated as a global variable; functions inherit the dynamic floating point environment of their caller and the caller's environment can be changed by the callee. To code algorithms that do not signal unnecessarily, SANE has `ProcEntry` and `ProcExit` procedures. `ProcEntry` saves the current environment and installs a default environment (rounding to nearest, non-trapping mode, sticky flags cleared, etc.). `ProcExit` takes a environment to be restored, saves the current exception state, and signals the exceptions raised in the saved exception state. For example, if the environment to be restored was trapping on overflow, calling `ProcExit` after an overflowing operation had occurred would cause an overflow trap to occur. SANE disallows optimizations that would change the value or observable side effects of floating point operations [6].

9.3.15. LIA

The 1994 LIA-1 standard (Language independent arithmetic, Part 1 ISO/IEC 10967-1) [52] aims to enhance the portability of programs by providing a parameterized machine model of floating point and integer arithmetic where properties of the arithmetic can be queried (possibly at runtime). The LIA-1 standard sets out to make "as few presumptions as possible about the underlying machine architecture," but the floating point of Cray architectures does not meet LIA-1 requirements and the LIA-1 standard has frequent comments about the relationship of LIA-1 requirements to IEEE 754. While LIA-1 claims that "the floating point requirements of this part of ISO/IEC 10967 [LIA-1] are compatible with (and enhance) IEC 559 [IEEE 754]," only a subset of IEEE 754 features are included in the base LIA-1 standard. Many of IEEE 754's features including special NaN and infinity values, signed zero, rounding to ±infinity, rounding to nearest even, square root operation, and the inexact flag are not directly covered in LIA-1. Full IEEE 754 conformance (indicated by a boolean `iec_559` value) is an option under LIA-1, with a few caveats. To conform to LIA-1, a program with a raised underflow, overflow, or invalid flag cannot be allowed to "complete successfully," an error must be reported in a "hard to ignore" manner. Additionally, LIA-1 defines a "undefined" flag which is (almost) the union of the IEEE 754 invalid and divide by zero flags [52].[53]

LIA-1 is primarily concerned with the precise definitions of basic arithmetic operations; many other language features relevant to floating point computation are not addressed in LIA-1. For example, various options for arithmetic on operands of different types are not given nor are requirements for binary to decimal conversion listed. Many LIA-1 requirements concern documentation for language standards or compiler implementations, such as what kinds of floating point optimizations are permissible or if extended precision can be used in expression evaluation. Each language can have different interfaces to LIA-1 features as long as the features are documented.

The annex of the LIA-1 standard says "It makes no sense to write code intended to run on all machines describable with the LIA model — the model covers too wide a range for that." Instead, LIA-1 facilities can be used to determine if a platform meets an algorithm's needs and expectations. LIA-1 does provide a framework for documenting IEEE 754 conformance; but since the entirety of IEEE 754 is not required, many corner cases are not addressed by LIA-1. For example, many of the IEEE 754 recommended functions have counterparts in LIA-1 required functions. However, since LIA-1 does not include infinity or NaN, the signaling behavior of the IEEE 754 recommended functions on those inputs is outside the scope of LIA-1. LIA-1 does not provide any guidance for the undefined cases of IEEE recommended functions such as `scalb`.

While it is claimed that "The documentation required by LIA-1 will highlight the differences between 'almost IEEE' systems and fully IEEE conforming ones," the parameters in LIA-1 do not even fully characterize the behavior of existing "almost IEEE" machines. For example, by default the Alpha architecture does not operate on subnormal numbers, in violation of the standard. Many other processor, such as the UltraSPARC, have similar non-conforming flush to zero modes. Therefore, in a LIA-1 environment with a processor flushing to zero, the `iec_559` value must be `false`. Such a processor performs rounding to nearest even for normal operations. However, the LIA-1 characterization of rounding modes does not distinguish between IEEE 754's round to nearest even and other round to nearest schemes, such as the VAX policy of always rounding away from zero in case of a tie. Therefore, information about the arithmetic is lost; an otherwise IEEE 754 compliant machine flushing to zero cannot be distinguished in LIA-1 from an arithmetic that is similar to IEEE 754 except that flush to zero and VAX round to nearest (or round to nearest odd) is used.

9.3.16. C9X

The C9X floating point proposal addresses two concerns for numeric programming in the C language; namely providing more predictable floating point arithmetic and a standard language binding for IEEE 754 features [93], [94]. Since C still aims to run on non-IEEE machines, full IEEE 754 support is not mandatory in C9X. By default, a C9X implementation is not required to respect all IEEE 754 floating point semantics, even on IEEE 754 compliant hardware. The following discussion focuses on the C9X requirements for fully IEEE 754 compliant implementations.

C9X does not require correctly rounded decimal to binary conversion over the entire range of possible values; however, correctly rounded conversion must occur if a string has less than or equal to `DECIMAL_DIG` digits. `DECIMAL_DIG` is large enough to ensure that all binary floating point numbers can be expressed via a decimal string. These requirements are more stringent than the IEEE 754 conversion requirements. To specify exact floating point values more easily, C9X supports hexadecimal floating point literals. To create special IEEE 754 values, the C9X `strtod` function (**`str`**`ing` **`to d`**`ouble`) recognizes "`inf`" and "`infinity`" as representing infinity and "`nan`" and "`nan(`*`character sequence`*`)`" as representing quiet NaNs. The *`character sequence`* is intended to allow additional information to be encoded in a NaN's significand in a platform-dependent manner. For

[53] If the undefined flag is cleared or set, both the invalid and divide by zero flags must be cleared or set.

backwards compatibility, instead of a NaN, strtod returns 0.0 if given a string argument that does not denote a floating point number. Other input and output library functions, such as printf, are also modified to handle the new floating point features. Runtime decimal to binary conversion is influenced by the dynamic rounding mode while compile time conversion always occurs under round to nearest [94].

In C9X, the float type corresponds to the single format and the double type to the double format. A third type, long double is platform dependent, corresponding to a floating point type at least as precise as double. An IEEE extended format should be used if available in preference to a non-IEEE format or to double. Additionally, C9X has two other platform dependent types defined as typedef's in <math.h>, float_t and double_t. float_t is at least as wide as float and double_t is at least as wide as double. The type float_t may map to float, double, or long double and double_t may map to double or long double. The mapping used is implementation dependent and is related to the expression evaluation policy. The mapping of *type*_t should be to the most efficient type at least as wide as *type* [94].

Table 41 — Type mappings used in C9X for different evaluation methods.

FLT_EVAL_METHOD	float_t mapping	double_t mapping
0	float	double
1	double	double
2	long double	long double
any other value	implementation defined	implementation defined

C9X allows a number of conventions for expression evaluation, depending on what is most efficient for a given platform. The mapping used is indicated by the constant macro FLT_EVAL_METHOD; the various possibilities are listed in Table 41. If FLT_EVAL_METHOD is 0, strict evaluation is being used. If FLT_EVAL_METHOD is 1 or 2, a variant of widest available is being used. The programmer cannot request a particular evaluation strategy; the strategy used by the compiler can only be queried, not set. If FLT_EVAL_METHOD is outside [0, 2] some other implementation defined evaluation strategy is in use. Early drafts of C9X supported scan for widest evaluation, but it was removed from later versions due to lack of prior art [93].

A variety of integer to floating point conversions are provided in C9X. The functions nearbyint and rint return the floating point number with the integral value of the floating point argument using the current rounding direction (nearbyint raises inexact but rint does not). The round function returns the nearest integral value in floating point, rounding ties away from zero while trunc rounds its argument to the integer value in floating point no larger in magnitude than the argument. roundtol rounds a floating point argument to the nearest integer value, rounding away from zero in ties; if the result is outside the range of long int, the result is unspecified. Casting from floating point to integer always rounds toward zero.

The result of converting a floating point NaN or infinity to integer is unspecified. Whether or not casting from floating point to integer can signal inexact is unspecified. C9X keeps ANSI C's rules for implicit integer to floating point and floating point to integer conversions.

The new header file <fenv.h> provides routines to access and control the floating point state. Functions are provided to get and set the dynamic rounding mode and sticky flags. C9X only directly supports non-trapping execution but a few of the library functions interact with floating point traps that change control flow. For example, the feraiseexcept function raises the conditions represented by its argument. If trapping mode is being used, feraiseexcept causes a trap to occur; feraiseexcept does not merely set the sticky flags [94]. However, C9X has no library function to install custom trap handlers.

Additional functions in <fenv.h> treat the floating point environment, rounding modes, sticky flags, and possibly other information, as an entity that can be saved and restored (similar to the ProcEntry and ProcExit in SANE). For example, on the x86, the precision control (controlling whether arithmetic operations are rounded to float, double, or double extended precision) is part of the floating point environment.

While C9X has library functions to manipulate the floating point state, using those functions is not sufficient information to make the compiler respect IEEE 754 floating point semantics. The macro FENV_ACCESS_ON must be used to ensure the compiler does not violate floating point semantics due to overly aggressive optimization. If IEEE 754 features (such as changing the rounding mode) occur when FENV_ACCESS_OFF is true, the results are undefined. FENV_ACCESS_ON acts as a lexically scoped declaration. While C9X defines FENV_ACCESS_DEFAULT, the value for this default is undefined. Similarly, FP_CONTRACT macros control in a lexically scoped manner whether or not a fused mac can be used. FP_CONTRACT_DEFAULT also has an implementation defined value [94]. While structured access is provided for controlling the use of fused mac and for informing the compiler when IEEE 754 semantics must be followed, other information, such as whether a function changes the rounding mode, or what flags a function may set, is only provided in comments and not in any language structure.

Constant expressions setting static variables and initializer lists for unions and arrays are defined to be unaffected by dynamic modes and to not set the sticky flags. Therefore, such initializers may be evaluated at compile time. If `FE_ACESSS` is on, all other floating point expressions must be evaluated as if at runtime.

C9X does not add function overloading to C, however, but some of the convenience of overloaded library functions can be achieved with macros, as shown below [93].

```
#define fpclassify(x)  (sizeof(x) == sizeof(float))? __feclassifyf(x) :\
                                      (sizeof(x) == sizeof(double)) ?   __feclassifyd(x): \
                                                                        __feclassifyl(x)
```

The math library of C9X includes some specifications for how functions should operate under exceptional conditions. However, domain errors are allowed to return implementation defined values and `errno` may or may not be set. Range errors also may or may not set `errno`. Some functions, such as the transcendental, exponential, and gamma functions may or may not set the inexact flag if the exact result is not representable. Whether or not the standard library functions honor different dynamic rounding directions is implementation defined. The behavior of the IEEE recommend functions in the C9X library is not fully specified and occasionally contradicts the standard. It is not stated if `scalb` is sensitive to dynamic rounding modes. In case the two arguments are equal, the `nextafter` function in C9X returns the second argument instead of the first argument as called for in IEEE 754. This convention alters the result of `nextafter` for some combinations of signed zero inputs. The C9X `logb` follows the IEEE 854 recommendations.

9.3.17. Modula-3

Starting in 1986, Modula-3 was designed to be a successor to Modula-2 suitable for systems programming while maintaining safety and strong typing. The IEEE recommended functions as well as functions to control IEEE 754 features are included in the Modula-3 standard library.

Modula-3 does not place constraints on decimal to binary or decimal to binary conversion. There are three floating point types, `REAL`, `LONGREAL`, and `EXTENDED`. A standard interface to find the size of a floating point type is provided. In floating point expression evaluation, no implicit promotion occurs, the two operands to an arithmetic operation must have the same type and the result has the same type as the operands. The `FLOAT` function takes an integer or floating point argument and converts it to the specified floating point type, rounding according to the current rounding mode. For converting floating point to integer, functions that round in all four directions are provided: `FLOOR`, `CEILING`, `ROUND`, and `TRUNC`. A Modula-3 compiler is not free to rearrange computations in a manner that could affect the result. For example, floating point addition cannot be regarded as associative [77].

The Modula-3 library supports, but not does mandate, IEEE 754 arithmetic. For example, besides returning IEEE 754 rounding modes, the `GetRounding` function may also return `Vax`, `IBM370`, or `Other`. Functions are provided to get and set the sticky flags. The trapping behavior can also be queried and set; if a condition is trapped on, a Modula-3 exception can be thrown. The floating point state is preserved across thread switches [77].

9.3.18. RealJava

RealJava [23] is a Java dialect modified to express all the required features of IEEE 754 in a portable manner (sticky flags are required, but floating point exceptions are not). RealJava is designed to allow full utilization of the available floating point hardware, avoiding performance problems caused by Java.

RealJava has many similarities with C9X. For example, RealJava's basic floating point types `floatN` and `doubleN` are analogous to C9X's `float_t` and `double_t`. While RealJava's binary to decimal conversion is correctly rounded, the type of unsuffixed floating point literals is `doubleN`, not `double` as in Java. Therefore, Java's compile time range checks on floating point literals are not always performed in RealJava. Additionally, RealJava defines decimal to binary to conversion to occur at runtime with no visible side effects (since there are no side effects, conversions within a loop can be hoisted outside of the loop). While `floatN` and `doubleN` are always IEEE formats, RealJava's built-in `longDouble` type may or may not be an IEEE style format and may or may not have direct hardware support [23].

Strict expression evaluation is used by default in RealJava. By defining a special class variable, a form of widest available evaluation, called natural evaluation in RealJava, is used in all methods of that class. A block level declaration is not provided. Natural evaluation promotes `float` values to `floatN` and `double` values to `doubleN` [23]. Since the `float` and `floatN` types may actually both map to the same format on some platforms, there is no easy method to guarantee an expression is evaluated in higher precision than the input data (of course explicit casts to a wider type could be added to the expression).

RealJava treats the rounding mode and sticky bits as global variables visible in all methods. If a method changes the dynamic rounding mode and does not restore the old value, the new rounding mode is in affect in the method's caller. Library methods are provided to sense and change the rounding mode and sticky flags [23].

9.3.19. Proposal for Extension of Java™ Floating Point Semantics

> Except for timing dependencies or other non-determinisms and given sufficient time and sufficient memory space, a Java program should compute the same result on all machines and in all implementations.
>
> —Preface to *The Java™ Language Specification* [38]

Since its public release, Java has promised "write once, run anywhere" portability. For many reasons, both intrinsic and practical, Java has not achieved its goal of freeing developers from porting their program from one system to another. However, by defining the exact sizes of its primitive types and by specifying expression evaluation order, Java is much more *predictable* than other contemporary languages such as C and C++. In a reversal of philosophy, Sun's proposal to modify Java's floating point semantics [91], (abbreviated PEJFPS), destroys Java's floating point predictability by allowing the compiler to arbitrarily use or not use extended precision values.

PEJFPS aims to grant limited access to extended precision hardware, where available, and to lessen the performance impact of an x86 processor strictly conforming to the original Java floating point semantics. In certain contexts, PEJFPS allows `float` and `double` local variables, parameters, and return values to be stored as and operated on as extended precision floating point numbers. However, neither arrays nor object fields can use extended formats. New class and method qualifiers `widefp` and `strictfp` control where the new semantics can be used. By default, existing Java source code and `class` files are subject to the revised semantics, breaking existing code that depends on Java's tight floating point specification.

Under PEJFPS, the virtual machine has wide latitude in choosing when and where to use extended precision. Even in `widefp` contexts on a processor with extended formats, before and after every floating point operation the virtual machine can promote ordinary floating point values to extended formats and can round expended values to narrower formats. This laxity in the proposal sanctions many perverse implementations. For example, assigning a `float` variable to a `double` variable can overflow. Calls to the same method with the same arguments can give different results depending on if the method is interpreted or compiled under a JIT. The proposal also allows values stored to memory during register spilling to be of a narrower format than the expression being evaluated. (On the recent x86 processors, even ignoring reduced memory traffic, the instruction to store a 64 bit `double` value executes faster than the instruction to store an 80 bit `double extended` value [51].) Spilling registers at a reduced width breaks referential transparency.

PEJFPS introduces much needless non-determinism into Java. The goals of PEJFPS can be met while preserving Java's predictability, such as by Borneo's `indigenous` type and `anonymous` declarations. PEJFPS does not address Java's other floating point failings, such as lack of support for the IEEE 754-required rounding modes and sticky flags.

9.3.20. Synopsis

In the past, few languages properly acknowledged issues related to floating point support. Either details were overlooked (such as omitting correctly rounded base conversion) or inconsistencies were tolerated (such as allowing compile-time and runtime base conversion of the same literal to give different values). FORTRAN has left a legacy of loosely specified floating point semantics.

Programming languages have been slow to provide support for IEEE 754 floating point and the languages that do support IEEE 754 floating point have usability problems. Java mandates IEEE 754 numbers and correctly rounded base conversions, but Java does not support all of the standard's required features. SANE and C9X have incomplete support for trapping mode; the trapping status can be set, but no portable mechanism is provided to install or write a trap handler. While Modula-3 integrates trapping mode into an existing exception mechanism, all changes to the floating point state are unstructured. Like SANE, C9X, and RealJava, Modula-3 provides the hardware model of rounding modes, sticky flags, and trapping status: a single global value inherited and modifiable by all functions. C9X has lexically scoped declaration to indicate if IEEE semantics must be followed but lacks lexically scoped declarations to actually use IEEE features. Many applications that take advantage of floating point features use them in a structured manner; therefore, structured control should be supported at the language level. Borneo maintains Java's base conversion requirements and mandates use of IEEE 754 while adding new language declarations to provide structured control over IEEE 754 features.

9.4. IEEE 754 Conformance

Table 42 — IEEE 754 Conformance in Java and Borneo.

Feature	Java	Borneo
directed rounding	explicitly forbidden (JLS §4.2.4)	`rounding` declarations, `Math.setRound`, `getRound`
sticky flags	not supported	another attribute of a method's signature, library methods to sense and alter flags, new control structure
floating point exceptions	explicitly forbidden (JLS §4.2.4)	`enable`/`disable` declarations, new exception classes
extended formats for primitive floating point types	only `float` and `double` are supported ([15] §5.15.3)	`indigenous` maps to `double extended` on machines supporting that format
non-signaling comparison operators	not supported	new operators added
fused mac (not part of IEEE 754)	not supported	library call provided

9.5. New Borneo keywords and textual literals

BorneoKeywordNotInJava: one of

```
all             enable          invalid         underflow
anonymous       flag            none            value
disable         indigenous      overflow        waved
divideByZero    inexact         rounding
```

BorneoTextualFloatingPointLiteral: one of

```
infinity        infinityD       nanf            nann
infinityf       infinityn       nanF            nanN
infinityF       infinityN       nand
infinityd                       nanD
```

9.6. Changes to the Java Grammar

My own good judgment tells me not to try to parse the statement.
—Mike McCurry, White House Press secretary, January 21, 1998

Borneo makes two kinds of changes to the Java grammar: new alternatives for existing Java grammar productions (JLS §19) and new productions for Borneo features.

9.6.1. Changes for the `indigenous` type

Augmented Java Syntax
FloatingPointType: one of

```
float double indigenous
```

FloatTypeSuffix: one of

```
f F d D n N
```

9.6.2. New method and constructor declarations

Augmented Java Syntax
ConstructorDeclaration:
    Modifiers$_{opt}$ *ConstructorDeclarator Throws*$_{opt}$ *Admits*$_{opt}$ *Yields*$_{opt}$ *ConstructorBody*

MethodHeader:
    Modifiers$_{opt}$ *Type MethodDeclarator Throws*$_{opt}$ *Admits*$_{opt}$ *Yields*$_{opt}$
    Modifiers$_{opt}$ `void` *MethodDeclarator Throws*$_{opt}$ *Admits*$_{opt}$ *Yields*$_{opt}$

New Borneo Productions
Admits:
    `admits` *TrappingConditions*

Yields:
    `yields` *TrappingConditions*

TrappingConditions:
    TrappingCondition
    TrappingConditions `,` *TrappingCondition*

TrappingCondition: one of
    `overflow underflow divideByZero invalid inexact all none`

9.6.3. New Block Declarations

Augmented Java syntax
LocalVariableDeclarationStatement:
    FloatingPointRoundingDeclaration `;`
    FloatingPointTrappingDeclaration `;`
    AnonymousValueDeclaration `;`

New Borneo Productions
FloatingPointRoundingDeclaration:
    `rounding` *Expression*

FloatingPointTrappingDeclaration:
    `enable` *TrappingConditions*
    `disable` *TrappingConditions*

AnonymousValueDeclaration:
    `anonymous` *FloatingPointType*

9.6.4. `flag-waved` Statement

Augmented Java Syntax
StatementWithoutTrailingSubstatement:
    FlagStatement

New Borneo Productions
FlagStatement:
    `flag` *Admits*$_{opt}$ *Yields*$_{opt}$ *Block Waves*$_{opt}$

Waves:
WaveClause
Waves WaveClause

WaveClause:
`waved` *TrappingConditions Admits*$_{opt}$ *Yields*$_{opt}$ *Block*

9.6.5. Operator Overloading

Borneo's operator overloading introduces many changes to the Java grammar. The definition of *Identifier* is modified to include names such as `op+`. However, these new names can only be used to name and explicitly call operator methods; they cannot be used as variables or class field names.

Augmented Java Syntax
Modifier: one of
Existing Java *Modifer*
`value`

MethodDeclarator:
`op` *Identifier (FormalParameterList*$_{opt}$*)*

Identifier:
Existing Java definition of an *Identifier*
`op` concatenated by the text of any of the overloadable operators in Table 13
`op` concatenated by a sequence of characters recognized by the regular expressions in Table 16
`op[]=`

UnaryExpression:
AdditiveOperator UnaryExpression

UnaryExpressionNotPlusMinus:
BitwiseComplementOperator UnaryExpression

MultiplicativeExpression:
MultiplicativeExpression MultiplicativeOperator UnaryExpression

AdditiveExpression:
AdditiveExpression AdditiveOrNovelOperator MultiplicativeExpression

ShiftExpression:
ShiftExpression ShiftOperator AdditiveExpression

RelationalExpression:
RelationalExpression RelationalOperator ShiftExpression

AndExpression:
AndExpression AndOperator EqualityOperator

ExclusiveOrExpression:
ExclusiveOrExpression ExclusiveOrOperator AndExpression

InclusiveOrExpression:
InclusiveOrExpression InclusiveOrOperator ExclusiveOrExpression

New Borneo Productions

AdditiveOperator:

Any operator matched by the Flex regular expression `[+-][‘@%^&*_|<>?]+`

BitwiseComplementOperator

Any operator matched by the Flex regular expression `"~"[‘@%^&*_|<>?]+`

MultiplicativeOperator

Any operator matched by the Flex regular expression `"*"[‘@%^&*_|<>?]+`

AdditiveOrNovelOperator:

AdditiveOperator
Any operator matched by the Flex regular expression `[‘@][‘@%^&*_|<>?]*`

ShiftOperator:

Any operator matched by the Flex regular expression `"<<"[‘@%^&*_|<>?]+ |`
`">>"[‘@%^&*_|>?][‘@%^&*_|<>?]*`

RelationalOperator:

Any operator matched by the Flex regular expression `’<’[‘@%^&*_|>?][‘@%^&*_|<>?]* |`
`’>’[‘@%^&*_|>?][‘@%^&*_|<>?]* |`
`"<="[‘@%^&*_|<>?]+ |`
`">="[‘@%^&*_|<>?]+`

AndOperator:

Any operator matched by the Flex regular expression `’&’[‘@%^*_|<>?][‘@%^&*_|<>?]*`

ExclusiveOrOperator:

Any operator matched by the Flex regular expression `’^’[‘@%^&*_|<>?]+`

InclusiveOrOperator:

Any operator matched by the Flex regular expression `’|’[‘@%^&*_<>?][‘@%^&*_|<>?]*`

10. References

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] Götz Alefeld and Jürgen Herzberger, *Introduction to Interval Computations*, Academic Press, New York 1983.

[3] ANSI, New York, American National Standard Programming Language FORTRAN, ANSI X3.9-1978, 1978.

[4] ANSI/IEEE, New York, IEEE Standard for Binary Floating Point Arithmetic, Std 754-1985 ed., 1985.

[5] Andrew Appel and Marcelo J. R. Gonçalves, "Hash-consing Garbage Collection," CS-TR-412-93, Princeton University, February 1993.

[6] *Apple Numerics Manual, Second Edition*, Apple, Addison-Wesley Publishing Company, Inc., 1988.

[7] John Backus, "The History of FORTRAN I, II, and III," History of Programming Languages, pp. 25-45, 1981.

[8] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.W. Chin, "Optimizing Matrix Multiply using PHiPAC: a Portable, High Performance, ANSI C Coding Methodology," LAPACK Working Note 111.

[9] W. S. Brown, "A Simple but Realistic Model of Floating-Point Computation," ACM Transactions on Mathematical Software, vol. 7, no. 4, pp. 445-480, 1981.

[10] J. G. P. Barnes, "An Overview of ADA," Software Practice and Experience, vol. 10, pp. 851-887, 1980.

[11] Robert G. Burger, R. Kent Dybvig, "Printing Floating-Point Numbers Quickly and Accurately," Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, May 21-24, 1996, pp. 108-116.

[12] William D Clinger, "How to Read Floating Point Numbers Accurately," Proceedings of the ACM Conference on Programming Language Design and Implementation, June 20-22 1990, pp. 92-101.

[13] E. Anderson, Z. Bia, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Gennbaum, S. Hammerling, A. McKenny, S. Ostourchov, and D. Sorensen, LAPACK Users' Guide, Release 2.0, SIAM, Philadelphia, 1995.

[14] Andrew W. Appel and David B. MacQueen, "Standard ML of New Jersey," Third International Symposium on Programming Language Implementation and Logic Programming, August 1991, pp. 1-13.

[15] Ken Arnold and James Gosling, *The Java™ Programming Language*, Addison-Wesley, 1996.

[16] L. S. Blackford, et. al. "LAPACK Working Note 112: Practical Experience in the Dangers of Heterogeneous Computing," http://www.netlib.org/lapack/lawns/lawn112.ps.

[17] Zoran Budimlic and Ken Kennedy, "Optimizing Java, Theory and Practice," submitted to "Concurrence: Practice and Experience," http://www.cs.rice.edu/~zoran/

[18] W.J. Cody et. al, "A Proposed Radix-and Word-length-independent Standard for Floating-Point Arithmetic," IEEE Micro vol. 4, no.4, August 1984, pp 86-100.

[19] William J. Cody and William Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, 1980.

[20] Computer Sciences Research Center, Bell Laboratories, *The Limbo Programming Language*, http://inferno.bell-labs.com/inferno/limbo.html.

[21] Jerome Coonen, *Contributions to a Proposed Standard for Binary Floating-Point Arithmetic*, Ph.D. Thesis, University of California, Berkeley 1984.

[22] Jerome Coonen, "A Note On Java Numerics," Numeric Interest Mailing list, Jan. 25, 1997.

[23] Jerome Coonen, “A Proposal for RealJava, Draft 1.0,” July 4, 1997, Numeric Interest Mailing list, http://www.validgh.com/java/realjava.

[24] Robert Paul Corbett, “Enhanced Arithmetic for Fortran,” SIGPLAN Notices, vol. 17, no. 12, December, 1982, pp. 41-48.

[25] James W. Demmel and Xiaoye Li, “Faster Numerical Algorithms via Exception Handling”, IEEE Transactions on Computers, vol. 43, no. 8, August 1994, pp. 983-992.

[26] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. LINPACK User’s Guide. SIAM, Philadelphia PA, 1979.

[27] Margaret A. Ellis, Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, 1990.

[28] Adin D. Falkoff and Kenneth E. Iverson, “The Design of APL,” IBM Journal of Research and Development, July 1973, pp. 324-334.

[29] Adin D. Falkoff and Kenneth E. Iverson, “The Evolution of APL,” History of Programming Languages, pp. 661-673, 1981.

[30] Adin D. Falkoff, “Some Implications of Shared Variables,” Proceedings of APL 76, pp. 141-148.

[31] Charles Farnum, “Compiler Support for Floating-Point Arithmetic,” Software Practice and Experience, vol. 18, no. 7, 1988, pp. 701-9.

[32] Richard J. Fateman, “High-Level Language Implications of the Proposed IEEE Floating-Point Standard,” ACM Transactions on Programming Languages and Systems, vol. 4, no. 2, April 1982, pp. 239-257.

[33] Stuart Feldman, “Language Support for Floating Point,” *IFIP TC2 Working Conference on the Relationship between Numerical Computation and Programming Languages*, J.K. Reid ed., 1982, pp. 263-273.

[34] David M. Gay, “Correctly Rounded Binary-Decimal and Decimal-Binary Conversions,” Numerical Analysis Manuscript No. 90-10, AT&T Bell Laboratories, Murray Hill NJ, 1990.

[35] David Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic,” Computing Surveys, vol. 23, no. 1, March 1991, pp. 5-48.

[36] Roger A. Golliver, “First-implementation artifacts in Java™”

[37] James Gosling, “Java Intermediate Bytecodes,” ACM SIGPLAN Workshop on Intermediate Representations (IR ‘95) also in SIGPLAN Notices vol. 30, no. 3, March 1995, pp 111-118.

[38] James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison-Wesley, 1996.

[39] Duane Hanselman and Bruce Littlefield, *The Student Edition of MATLAB: version 5 user’s guide*, Prentice Hall, 1997.

[40] John R. Hauser, “Handling floating-point exceptions in numeric programming,” ACM Transactions on Programming Languages and Systems, vol. 18, no. 2, March 1996, pp. 139-174.

[41] John R. Hauser, *Programmed exception handling*. M.S. Thesis, University of California, Berkeley, CA 1994.

[42] John R. Hauser, SoftFloat, http://www.cs.berkeley.edu/~jhauser/arithmetic/softfloat.html.

[43] Roger Hayes, *100% Pure Java Cookbook*, Sun Microsystems, Inc., http://java.sun.com/100percent

[44] John L. Hennessy, “Symbolic debugging of Optimized Code,” ACM Transactions on Programming Languages and Systems, vol. 4, no. 3, July 1982, pp. 323-344.

[45] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach, Second Edition*. Morgan Kaufmann Publishers, Inc., 1996.

[46] Hewlett-Packard Company, *PA–RISC 1.1 Architecture and Instruction Set Reference Manual*, Third Edition, 1996.

[47] Paul N. Hilfinger, Titanium Language Working Sketch, http://www.cs.berkeley.edu/projects/Titanium/lang-ref.ps.

[48] T. E. Hull, A. Abrham, M. S. Cohen, A. F. X. Curley, C. B. Hall, D. A. Penny, J. T. M. Sawchuk, "Numerical Turing," SIGNUM Newsletter, vol. 20, no. 3, 1985, pp. 26-34.

[49] IBM, FORTRAN II Programming, IBM Corporation, Programming Systems Publications, 1964.

[50] IBM, FORTRAN IV Language, IBM Corporation, Programming Systems Publications, 1964.

[51] Intel Corporation, *Pentium Pro Family Developer's Manual*, 1996.

[52] ISO, *Information technology — Language independent arithmetic — Part 1: Integer and Floating Point Arithmetic*. ISO/IEC 10967-1:1994(E). International Standards Organization, Geneva, 1994.

[53] Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report, Fourth Edition*, revised by Andrew B. Mickel and James F. Miner, Springer-Verlag, 1991.

[54] Bill Joy, "Proposal for Enhanced Numeric Programming Facilities in Java, Draft #3," March 16, 1997.

[55] William Kahan, "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit," Chapter 7 in *The State of the Art in Numerical Analysis*, ed. A. Iserles and M.J.D. Powell, Clarendon Press, Oxford, 1987.

[56] William Kahan, Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, http://HTTP.CS.Berkeley.EDU/~wkahan/ieee754status/ieee754.ps.

[57] William Kahan, "Miscalculating Area and Angles of a Needle-like Triangle," http://www.cs.berkeley.edu/~wkahan/triangle.ps.

[58] William Kahan, "A More Complete Interval Arithmetic," Lecture notes prepared for a summer course at the University of Michigan, June 17-21, 1968.

[59] William Kahan, personal communication.

[60] William Kahan, "Presubstitution and Continued Fractions," July 1995.

[61] William Kahan, "Why do we need a floating-point arithmetic standard?," 1981.

[62] William Kahan and Melody Y. Ivory, "Roundoff Degrades an Idealized Cantilever," http://www.cs.berkeley.edu/~wkahan/Cantilever.ps.

[63] William Kahan and J.W. Thomas, "Augmenting A Programming Language with Complex Arithmetic," Report No. UCB/CSD 91/667, December 1991.

[64] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language*, PTR Prentice Hall, 1978.

[65] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language, Second Edition*. PTR Prentice Hall, 1988.

[66] Tim Lindholm and Frank Yellin, *The Java™ Virtual Machine Specification*, Addison-Wesley, 1996.

[67] Alex Liu, personal communication, August 8, 1997.

[68] Brian Livingston, "Don't count on the Calculator for dollars and sense," Infoworld vol. 16, no. 48, Novemeber 28, 1994, p. 38.

[69] David W. Matula, "In-and-out conversions," Communications of the ACM, vol. 11, no. 1, January 1968, pp. 47-50.

[70] David W. Matula, "A formalization of floating-point numeric base conversion," IEEE Transactions on Computers, vol. C-19, no. 8, August 1970, pages 681-692.

[71] E. E. McDonnell, "Zero Divided by Zero." Proceedings of APL 76, pp. 295-296.

[72] Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, The MIT Press, 1990.

[73] Ramon E. Moore, *Methods and Applications of Interval Analysis*, SIAM Philadelphia, 1979.

[74] Motorola, *MC68881/882 Floating-Point Coprocessor User's Manual, Second edition*, Prentice Hall, 1989.

[75] Naur, et al, "Report on the Algorithmic Language ALGOL 60," Communications of the ACM, vol. 6, no. 1, 1963, pp. 1-17.

[76] Nedialko Stoyanov Nedialkov, *Precision Control and Exception Handling in Scientific Computing*, M.S. Thesis, Department of Computer Science, University of Toronto, 1994.

[77] Greg Nelson, ed., *Systems Programming with Modula-3*, Prentice Hall, 1991.

[78] J. Michael O'Conner and Marc Tremblay, "PicoJava-I: The Java Virtual Machine in Hardware," IEEE Micro March 1997, pp. 45-53.

[79] John Peterson et. al., "Report on the Programming Language Haskell, A Non-strict, Purely Functional Language Version 1.4," April 7, 1997, http://www.haskell.org/report/.

[80] Evgenija D. Popova, "Interval Operations Involving NaNs," Reliable Computing, vol. 2, no. 2, 1996, pp. 161-166.

[81] Dietmar Ratz, "Inclusion Isotone Extended Interval Arithmetic," Report No. D-76128 Karlsruhe, Universität Karlsruhe, May 1996.

[82] Fred N. Ris, *Interval Analysis and Applications to Linear Algebra*, Doctoral thesis, Wadam College, Oxford University, 1972.

[83] Richard L. Sites, Richard T. Witek, *Alpha AXP Architecture Reference Manual, Second Edition*, Digital Press, 1995.

[84] Jonathan Richard Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates in C," CMU-CS-96-140, May 17, 1996.

[85] Richard Stallman, *Using and Porting GNU CC*, Free Software Foundation, 1995.

[86] Guy L. Steele, Jr., *Common Lisp, The Language*, Digital Press, 1984.

[87] Guy L. Steele, Jr., Jon L White, "How to Print Floating-Point Numbers Accurately," Proceedings of the ACM Conference on Programming Language Design and Implementation, June 20-22 1990, pp. 112-123.

[88] Guy L. Steele, Jr. and Richard P. Gabriel, "The Evolution of Lisp," ACM SIGPLAN Notices, vol. 28, no. 3 March 1993, pp. 231-270.

[89] Bjarne Stroustrup, *The C++ Programming Language*. Addison-Wesley Publishing Company, 1991.

[90] Charles-Francois Sturm, "Memoire sur la resolution des equations numeriques," 1845.

[91] Sun Microsystems Inc., "Proposal for Extension of Java™ Floating Point Semantics," Revision 1, May 1998, http://java.sun.com/feedback/fp.html.

[92] A. S. Tanenbaum, "A Tutorial on Algol 68," Computing Surveys vol. 8, no. 2, June 1976.

[93] Jim Thomas, C9X Floating Point, WG14/N546 X3J11/96-010 (Draft 2/26/96).

[94] Jim Thomas, C9X Floating Point, WG14/N595 X3J11/96-059 (Draft 9/12/96).

[95] United States Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983.

[96] Dennis Verschaeren, Annie Cuyt, and Brigitte Verdonk, "On the need for predictable floating-point arithmetic in the programming languages Fortran 90 and C / C++," ACM SIGPLAN Notices, vol. 32, no. 3, March 1997, pp. 57-64.

[97] Wolfgang Walter, FORTRAN-SC: A FORTRAN Extension for Engineering/Scientific Computation with Access to ACRITH," printed in *Reliability in Computing*, ed. Ramon E. Moore, Academic Press, Inc., 1988, pp. 43-62.

[98] David L. Weaver and Tom Germond, ed., *The SPARC Architecture Manual, version 9*, Prentice Hall, 1994.

[99] Paul R. Wilson, "Uniprocessor Garbage Collection Techniques," to appear in ACM Computing Surveys.

[100] Niklaus Wirth, *Programming in MODULA-2, Third Corrected Edition*, Springer-Verlag, 1985.

[101] Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer, Second Edition*, Addison-Wesley Publishing Company, 1991.

[102] Working Paper for the Draft Proposed International Standard for Information Systems—Programming Language C++, Doc No: X3J16/95-0087.

[103] Kathy Yelick, et. al, "Titanium: A High-Performance Java Dialect," ACM 1998 Workshop on Java for High-Performance Network Computing, Stanford California, February 1998.

Hofstadter's Law:
It always takes longer than you think it will take, even if you take into account Hofstadter's Law.
—Douglas R. Hofstadter