Ada, C, C++, and Java vs. The Steelman

David A. Wheeler
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772
(703) 845-6662
dwheeler@ida.org

Abstract

This paper compares four computer programming languages (Ada95, C, C++, and Java) with the requirements of "Steelman", the original 1978 requirements document for the Ada computer programming language. This paper provides a view of the capabilities of each of these languages, and should help those trying to understand their technical similarities, differences, and capabilities.

Introduction

In 1975 the U.S. Department of Defense (DoD) established a "Common High Order Language" program with the goal of establishing a single high order computer programming language appropriate for DoD embedded computer systems. A High Order Language Working Group (HOLWG) was established to formulate the DoD requirements for high order languages, to evaluate existing languages against those requirements, and to implement the minimal set of languages required for DoD use. The requirements were widely distributed for comment throughout the military and civil communities, producing successively more refined versions from Strawman through Woodenman, Tinman, Ironman, and finally Steelman. Steelman was released in June 1978 [DoD 1978]. An electronic version of Steelman is available at "http://www.adahome.com/History/Steelman/intro.htm". The original version of the Ada computer programming language was designed to comply with the Steelman requirements.

Today there are a number of high order languages in commercial and military use, including Ada95, C, C++, and Java. I thought it would be an interesting and enlightening exercise to compare these languages to Steelman; this paper provides the results of this exercise. Other comparisons of these languages do exist (for example, [Sutherland 1996, Seidewitz 1996]), but none use the Steelman requirements as a basis.

This paper compares four computer programming languages (Ada95, C, C++, and Java) with the requirements of "Steelman". The paper first describes the rules used in this comparison of these four languages with Steelman. This is followed by conclusions summarizing how each language compares to Steelman. After the references is a large table, the "meat" of this paper, showing how each of these languages compare to each of the Steelman requirements.

This paper does not attempt to determine if the Steelman requirements are still relevant. In the author's opinion, most of the Steelman requirements are still desirable for general-purpose computer languages when both efficiency and reliability are important concerns. Steelman is no doubt incomplete; in particular, object-orientation is not a Steelman requirement but is widely considered desirable. However, desirability and completeness need not be true for this comparison to be useful, because this paper simply uses Steelman as a framework for examining technical issues of these computer languages. Steelman is a useful framework because it was widely reviewed and is technically detailed.

Rules for Comparison

The primary rule used in this paper is that a language provides a feature if:

1. that feature is defined in the documents widely regarded as the language's defining document(s), and
2. that feature is widely implemented by compilers typically used for that language with essentially the same semantics.

Features that are only provided by a single implementation, or are defined but cannot be depended upon across the most commonly used implementations, are not considered to be part of the language. Examples of features that cannot be depended on are features which are often unimplemented, implemented incorrectly on widely-used compilers, or implemented with significantly differing semantics. Subset compilers for research or student work are not considered unless they are widely used by users of that language. One area where some "benefit of the doubt" is given is for C++ templates. C++ templates are part of the C++ language definition, but current C++ compilers implement templates with different semantics [FSF 1995] and with different levels of quality [OMG 1995]. As a result, C++ templates are currently difficult to use in a portable way. Still, credit is given for C++ templates since the intent is clear and they can be used today (with trouble) on most compilers.

The defining documents used for each of these languages are as follows:

- Ada95: The "Ada 95 Reference Manual" (referred to as the Language Reference Manual, or LRM) [ISO 1995] is the official definition of Ada95. This document is freely available as a hypertext document at "http://www.adahome.com/rm95/". Ada95 is a revision of the original Ada language (now called Ada83), which added support for object-orientation and various other capabilities. The rest of this paper will use the term "Ada" to mean "Ada95".

- C: The official current definition of C is ISO/IEC 9899:1990, which is essentially the same as the definition developed by ANSI as ANSI X3.159-1989 [ANSI 1989]. This has been extended by Normative Addition 1 to support internationalization features, as described at "http://www.lysator.liu.se/c/na1.html". Programmers often don't have these costly standards documents; instead, they often use the book by C's developers, Kernighan and Ritchie [Kernighan 1988], so this book is also considered a defining document, along with its errata (see "http://www.lysator.liu.se/c/c-errata.html").

- C++: There is ongoing work to standardize C++ by 1998 but no final, approved standard. I have used the documents closest to a standard yet widely available, namely, the *Working Paper of ISO Working Group WG21 (April 28, 1995)*. WG21 is developing the International Standard for the C++ programming language. This document is available at "ftp://ftp.research.att.com/dist/c++std/WP/". and an HTML version is available at "http://www.cygnus.com/misc/wp/". There are more recent versions of this document, but access to these revisions is more restricted, and changes from this widely-available version are less likely to be fully implemented in a wide number of compilers. Two other key defining C++ documents are the older books *The Annotated C++ Reference Manual* (ARM) [Ellis 1990] and Stroustrup's *The C++ Programming Language* [Stroustrup 1991]. In July 1994 the ANSI/ISO C++ Standards committee voted to adopt the Standard Template Library (STL) as part of the C++ library, so the STL is considered part of C++ in this paper. Other useful C++ references can be found at "http://yoyodyne.tamu.edu/oop/oopcpp.html".

- Java: The current defining documentation on Java is the documentation set available from Javasoft (a Sun Microsystems business) at "http://java.sun.com/doc/language.html". Note that what is being evaluated here is the Java *language*, not the underlying Java Virtual Machine. There is already a compiler (by Intermetrics) that takes Ada source code and generates Java Virtual Machine code (see "http://www.adahome.com/Tutorials/Lovelace/java.htm").

Conclusions

The appendix shows how well each language supports each Steelman requirement.

The following table shows the number of Steelman requirements met by each language. The leftmost column shows the name of the language. The next columns are the number of requirements the language does not meet, only partially meets, mostly meets, and completely meets. The final column shows the percentage of Steelman requirements mostly or completely met by the language. Note that Ada meets the most, followed by Java, C++, and C in that order. The Java and C++ percentages are probably too close to be considered significantly different.

Language	"No"	"Partial"	"Mostly"	"Yes"	Percentage of Answers with "Mostly" or "Yes"
Ada	3	5	11	94	93%
C	32	21	16	44	53%
C++	19	17	23	54	68%
Java	20	12	22	59	72%

Caution is warranted, since differing percentages of "yes" values do not necessarily imply that a particular language is more suitable than another for a given specific task. Note that the original version of Ada was specifically designed to meet the Steelman requirements, while none of the other languages were specifically designed to do so, so it is expected that Ada would meet more of the Steelman requirements than the rest. Also, all of these languages have capabilities that are not Steelman requirements. For example, direct support for object-orientation is a feature of Ada, C++, and Java, but is not a Steelman requirement. Readers should use this comparison to gain additional understanding of each of these different languages, and determine which "yes" and "no" values are of importance to them.

The following are high-level remarks comparing each language to Steelman based on the table in the appendix, including remarks on the language support for reliability (requirement 1B):

- Ada has the most "yes" responses to Steelman; this is no surprise, since Ada83 was designed to meet the Steelman requirements, and Ada95 is a superset of Ada83. Ada has "no" responses to only 3 Steelman requirements: 3-3F, 5D, and 10F. Requirement 3-3F requires a specific syntax for constants and functions that Ada does not support. Requirement 5D restricts the kinds of values supported by the language. Ada83 complied more fully with requirement 5D (e.g. by not permitting access values to functions), but this Steelman requirement was found to be too restricting. Ada95 removed these restrictions, resulting in noncompliance with Steelman requirement 5D. Steelman requirement 10F requires simple assertion capabilities. Run-time assertions can be trivially implemented in Ada, but Ada does not provide a built-in construct for them. Some Steelman requirements are only supported by Ada, for example, fixed-point types (requirements 3-1G and 3-1H).

- C does not have a number of the Steelman capabilities. C does not have support for controlling concurrency - calls to the local operating system must be used instead. C also does not have an exception handling system nor generic processing (setjmp/longjmp and preprocessor commands can perform similar actions in trivial cases, but they are not practical substitutes in larger programs). Probably a more fundamental issue is that, while good programs can be written using C, C does not substantially aid in reliability nor maintainability, and does not try to maximize compile-time detection of errors. Those who believe otherwise need only compare C's error detection capabilities with C++, Java, and Ada.

- C++ supports exception handling and templates (generics), although at this time using C++ templates is problematic. C++ does not support concurrency directly, taking the same approach to this as C does. C++ does try to detect more errors at compile-time than C does by tightening up C's type system somewhat.

- Java supports exception handling and concurrency, and in general tries to detect errors at compile time. Java does not support enumerated types nor generics (templates). The former can be partly simulated with a long series of constants, and the latter with Java interfaces and the root Object type, but neither are very good mechanisms for simulating these capabilities. These weaknesses are well-known; for example, "Pizza" is superset of Java that adds templates and other capabilities (see "http://wwwipd.ira.uka.de/~odersky/papers.html#Pizza"). Java does not provide direct control of low-level hardware (such as the size and bit structure of types), since it was not designed for that purpose. Java does try to provide good compile-time and run-time protection; some of the "no" answers in the table are specifically because of this (for example, Java initializes all variable values which goes against requirement 5E and the efficiency issues of requirement 1D). Like Ada, Java does not have a built-in run-time assertion mechanism (requirement 10F), but this is trivially implemented in Java.

Again, users of this paper should apply caution; differing percentages of "yes" values and capabilities do not necessarily imply that a particular language is more suitable than another for a given specific task. Readers should examine each answer and determine which "yes" and "no" values are of importance to them. This information is intended to help readers gain additional understanding of each of these different languages.

References

- [ANSI 1989] ANSI C. 1989. ANSI X3.159-1989.
- [DoD 1978] U.S. Department of Defense. June 1978. ‘‘Department Of Defense Requirements for High Order Computer Programming Languages: "Steelman"’’ Electronically available at "http://www.adahome.com/History/Steelman/intro.htm".
- [Ellis 1990] Ellis, Margaret A., and Bjarne Stroustrup. 1990. "The Annotated C++ Reference Manual" (ARM). ISBN 0-201-51459-1 Reading, MA: Addison-Wesley.
- [FSF 1995] Free Software Foundation. 1995. "Where's the Template?". *Using and Porting GNU C*. Available at many locations, e.g. "http://www.delorie.com/gnu/docs/gcc/gcc_toc.html".
- [ISO 1995] ISO. January 1995. *Ada 95 Reference Manual*. ANSI/ISO/IEC-8652:1995. Available at "http://www.adahome.com/rm95/".
- [Kernighan 1988] Kernighan, Brian W., and Dennis M. Ritchie. 1988. "The C Programming Language". Second Edition. ISBN 0-13-110362-8. Englewood Cliffs, NJ: Prentice-Hall.
- [OMG 1995] Object Management Group. July 1995. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0. Section 15.1.2 says ‘‘Because C++ implementations vary widely in the quality of their support for templates, this mapping does not explicitly require their use ...’’.
- [Seidewitz 1996] Seidewitz, Ed. October 1996. "Another Look at Ada 95". *Object Magazine*. NY, NY: SIGS Publications Inc.
- [Stroustrup 1991] Stroustrup, Bjarne. 1991. The C++ Programming Language. Second Edition. ISBN 0-201-53992-6. Reading, MA: Addison-Wesley.
- [Sutherland 1996] Sutherland, Jeff. September 1996. "Smalltalk Manifesto". *Object Magazine*. NY, NY: SIGS Publications Inc.

This paper is not endorsed by, and does not necessarily represent the views of, the Institute for Defense Analyses (IDA), the U.S. Department of Defense (DoD), or the U.S. Government.

I am grateful to Michael Kappel, Magnus Kempe, and James Kuehn for their constructive comments.

Appendix: Table Comparing Four Languages to Steelman

In this table, the left-hand column gives the Steelman requirement. The next four columns show how well Ada, C, C++, and Java meet this requirement. An entry of "yes" indicates that the language and its major implementations generally meet the requirement, while a "no" indicates that requirement is generally not met. There are two intermediate entries: "partial" indicates some of the requirement is met, but a significant portion (or intent) of the requirement is not met, "mostly" indicates that the requirement is generally met, but some specific capability of the requirement is not fully met. Underneath the columns for each language is commentary explaining these entries.

I have tried to be fair to all of these languages. Nevertheless, some of these entries, particularly in section 1, are judgement calls. Readers are encouraged to revisit each entry (particularly in section 1), compare each language, and draw their own conclusions. Items which I felt are particularly questionable have been marked with a question mark ("?").

<table>
<tr><th>Requirement</th><th>Ada</th><th>C</th><th>C++</th><th>Java</th></tr>
<tr><td rowspan="2">1A. Generality. The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications involve real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing.</td><td>yes</td><td>yes</td><td>yes</td><td>**partial**</td></tr>
<tr><td colspan="4">Java can’t directly control hardware; Java programs must declare native methods and implement such operations in another language.</td></tr>
<tr><td rowspan="2">1B. Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to correct programming errors.</td><td>yes</td><td>**no**</td><td>**partial**?</td><td>mostly?</td></tr>
<tr><td colspan="4">Ada requires separate specifications for all modules other than stand-alone subprograms. C and C++ contain many well-known traps (= vs. ==, & vs. &&, premature semicolon in control structures, fall-through behavior in switch statements when "break" is omitted); some were removed or made less likely in Java but others were not. C permits separate specifications (through prototypes) but are optional; function names are globally accessible by default and can be incorrectly redefined. C++ supports separate specifications and has a slightly tighter type system than C. Also, good use of C++’s object-oriented features should increase the likelihood of compile-time detection of some kinds of errors. Java automatically generates specifications (as opposed to using redundant specifications). C and C++ do little checking at run-time. Both Ada and Java perform a number of run-time checks (e.g. bounds checking and checks for null values) to detect errors early.</td></tr>
<tr><td rowspan="2">1C. Maintainability. The language should promote ease of program maintenance. It should emphasize program readability (i.e., clarity, understandability, and modifiability of programs). The language should encourage user documentation of programs. It shall require explicit specification of programmer decisions and shall provide defaults only for instances where the default is stated in the language definition, is always meaningful, reflects the most frequent usage in programs, and may be explicitly overridden.</td><td>yes?</td><td>**partial**?</td><td>**partial**?</td><td>mostly?</td></tr>
<tr><td colspan="4">Ada was originally designed with readability in mind. C was not, and can easily be (ab)used to make impenetrable code. C (and hence C++ and Java) includes a great deal of terse notation which reduces readability (e.g. the "for" loop notation, using "&&" instead of "and", and operators such as "<<"). C++’s object-oriented features, if used, are likely to improve maintainability (because they force interfaces to be defined and used). Java’s document comments (//*) and standard documentation conventions aid in readability. Note that "readability" of a programming language is extremely subjective - well-structured programs can be read and maintained in any language by someone who knows the language, and no language can prevent all poor approaches. At issue in this requirement is how strongly the language encourages readable and maintainable code.</td></tr>
<tr><td rowspan="2">1D. Efficiency. The language design should aid the production of efficient object programs. Constructs that have unexpectedly expensive implementations should be easily recognizable by translators and by users. Features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs. Execution time support packages of the language shall not be included in object code unless they are called.</td><td>yes</td><td>yes</td><td>yes</td><td>**partial**?</td></tr>
<tr><td colspan="4">Ada functions returning unbounded size objects usually have hidden extra efficiency costs (access types can be used where this is important). C++ implicit conversion operations may be activated in situations not easily recognizable by its users. C/C++ pointer arithmetic and aliasing prohibit some optimizations. Java’s garbage collection raises questions about efficiency and guaranteed timing, especially in real-time systems.</td></tr>
</table>

1E. Simplicity. The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications. It should have few special cases and should be composed from features that are individually simple in their semantics. The language should have uniform syntactic conventions and should not provide several notations for the same concept. No arbitrary restriction should be imposed on a language feature.	yes	yes	mostly	yes
	Ada includes both Ada 83’s discriminated records and the newer (OO) tagged types (these have many similarities). C is a very simple language (though not necessarily simple to use). C++ has C operations and its own operations (new/delete vs. malloc/free, cout vs. printf).			
1F. Implementability. The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.	yes	yes	yes	yes
	All of these languages have been reasonably implemented.			
1G. Machine Independence. The design of the language should strive for machine independence. It shall not dictate the characteristics of object machines or operating systems except to the extent that such characteristics are implied by the semantics of control structures and built-in operations. It shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. Nevertheless, there shall be a facility for defining those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.	yes	yes	yes	yes
	Approaches differ. Ada includes a number of mechanisms to query the underlying configuration (such as bit ordering conventions) and C/C++ include some querying mechanisms. Conditional compilation in Ada and Java is handled through "if (constant)" statements (this does not permit conditional compilation in cases where "if" statements are not permitted). Java has few mechanisms for querying the underlying configuration and imposes requirements on bit length and semantics of numeric types that must be supported. Java strives for machine independence by hiding the underlying machine.			
1H. Complete Definition. The language shall be completely and unambiguously defined. To the extent that a formal definition assists in achieving the above goals (i.e., all of section 1), the language shall be formally defined.	yes	yes	yes	yes

2A. Character Set. The full set of character graphics that may be used in source programs shall be given in the language definition. Every source program shall also have a representation that uses only the following 55 character subset of the ASCII graphics: %&'()*+,-./:;<=>? 0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ_ Each additional graphic (i.e., one in the full set but not in the 55 character set) may be replaced by a sequence of (one or more) characters from the 55 character set without altering the semantics of the program. The replacement sequence shall be specified in the language definition.	yes	mostly (trigraphs and digraphs)	mostly (trigraphs and digraphs)	**no**
	C, C++, and Java usually use a number of characters (such as {, }, [,], and #) that are not available on some European terminals (which only offer the seven-bit ISO 646 character set and use these positions for accented characters). C added trigraphs to help European users, but trigraphs are horrible to use in practice. Normative Addition 1 to C added digraphs and the <iso646.h> header in an attempt to make C easier to use in such cases; while somewhat improved, such programs are still more difficult to read. Note that supporting restricted character sets is becoming less important as old 7-bit European terminals disappear, and restrictions to support upper-case-only users are now irrelevant.			
2B. Grammar. The language should have a simple, uniform, and easily parsed grammar and lexical structure. The language shall have free form syntax and should use familiar notations where such use does not conflict with other goals.	yes	yes	**partial**	yes
	C has a few cases where parser state dependent feedback to the lexical analyzer is necessary (e.g. typedef, preprocessor tokenization). C++ is more difficult to parse because it isn't LALR(1). Java isn't really LALR(1) either, but known techniques make it so it can be handled as though it is LALR(1).			
2C. Syntactic Extensions. The user shall not be able to modify the source language syntax. In particular the user shall not be able to introduce new precedence rules or to define new syntactic forms.	yes	**no**	**no**	yes
	C/C++ preprocessor can be used to create some (obscure) syntactic forms. A preprocessor (such as cpp or m4) can be used with any language, including Ada and Java, but neither include a preprocessor in their definition.			
2D. Other Syntactic Issues. Multiple occurrences of a language defined symbol appearing in the same context shall not have essentially different meanings. Lexical units (i.e., identifiers, reserved words, single and multicharacter symbols, numeric and string literals, and comments) may not cross line boundaries of a source program. All key word forms that contain declarations or statements shall be bracketed (i.e., shall have a closing as well as an opening key word). Programs may not contain unmatched brackets of any kind.	yes	mostly	mostly	mostly
	C comments (also supported by C++) cross multiple lines. C,C++, and Java don't have "closing" key words, but use matching opening and closing braces. Matching braces have the advantage of being easy to type and support with text editors, but permit errors in maintenance when the "wrong" matching braces are used.			
2E. Mnemonic Identifiers. Mnemonically significant identifiers shall be allowed. There shall be a break character for use within identifiers. The language and its translators shall not permit identifiers or reserved words to be abbreviated. (Note that this does not preclude reserved words that are abbreviations of natural language words.)	yes	yes	yes	yes
	All support this. Note that Ada is case-insensitive, while C, C++, and Java identifiers are case-sensitive.			
2F. Reserved Words. The only reserved words shall be those that introduce special syntactic forms (such as control structures and declarations) or that are otherwise used as delimiters. Words that may be replaced by identifiers, shall not be reserved (e.g., names of functions, types, constants, and variables shall not be reserved). All reserved words shall be listed in the language definition.	yes	yes	yes	yes

<table>
<tr><td rowspan="2">2G. Numeric Literals. There shall be built-in decimal literals. There shall be no implicit truncation or rounding of integer and fixed point literals.</td><td>yes</td><td>mostly</td><td>mostly</td><td>yes</td></tr>
<tr><td colspan="4">All support numeric literals for integers. C and C++ permit implicit rounding, though many compilers will catch this. Only Ada directly supports fixed point numbers (see 3-1G).</td></tr>
<tr><td rowspan="2">2H. String Literals. There shall be a built-in facility for fixed length string literals. String literals shall be interpreted as one-dimensional character arrays.</td><td>yes</td><td>yes</td><td>yes</td><td>mostly</td></tr>
<tr><td colspan="4">Java String and String_Buffer are considered special types, not one-dimensional character arrays.</td></tr>
<tr><td rowspan="2">2I. Comments. The language shall permit comments that are introduced by a special (one or two character) symbol and terminated by the next line boundary of the source program.</td><td>yes</td><td>no</td><td>yes</td><td>yes</td></tr>
<tr><td colspan="4">Only C lacks comments automatically terminated by the end of line. Note that in practice, many C compilers share the preprocessor with C++ and can permit //-style comments with special compilation options, but this is not permitted portably by the C standard.</td></tr>
<tr><td rowspan="2">3A. Strong Typing. The language shall be strongly typed. The type of each variable, array and record component, expression, function, and parameter shall be determinable during translation.</td><td>yes</td><td>partial</td><td>mostly</td><td>yes</td></tr>
<tr><td colspan="4">Note that C and C++ have compile-time types but both are more weakly typed. Typedef does not define a new type, just a name synonym. A pointer to an array of objects is considered equivalent to a pointer to an object. In C, enumerations are considered identical to int, while in C++ enumerations are different types.</td></tr>
<tr><td rowspan="2">3B. Type Conversions. The language shall distinguish the concepts of type (specifying data elements with common properties, including operations), subtype (i.e., a subset of the elements of a type, that is characterized by further constraints), and representations (i.e., implementation characteristics). There shall be no implicit conversions between types. Explicit conversion operations shall be automatically defined between types that are characterized by the same logical properties.</td><td>yes</td><td>no</td><td>partial?</td><td>partial?</td></tr>
<tr><td colspan="4">C, C++, and Java do not have subtypes (types with additional constraints) for primitive types (such as int or float). Class structures can be used in C++ and Java to implement additional constraints on classes. C and C++ have some representation control using bitfield locations; this is not considered separate from the type. C and C++ have a number of implicit conversions.</td></tr>
<tr><td rowspan="2">3C. Type Definitions. It shall be possible to define new data types in programs. A type may be defined as an enumeration, an array or record type, an indirect type, an existing type, or a subtype of an existing type. It shall be possible to process type definitions entirely during translation. An identifier may be associated with each type. No restriction shall be imposed on user defined types unless it is imposed on all types.</td><td>yes</td><td>mostly</td><td>mostly</td><td>mostly</td></tr>
<tr><td colspan="4">C and C++ don’t have subtypes. Java doesn’t have subtypes or enumerated types; see 3B and 3-2A.</td></tr>
<tr><td rowspan="2">3D. Subtype Constraints. The constraints that characterize subtypes shall include range, precision, scale, index ranges, and user defined constraints. The value of a subtype constraint for a variable may be specified when the variable is declared. The language should encourage such specifications. [Note that such specifications can aid the clarity, efficiency, maintainability, and provability of programs.]</td><td>mostly</td><td>no</td><td>no</td><td>no</td></tr>
<tr><td colspan="4">Ada supports user definition of range, precision, scale, and index ranges, but does not directly support arbitrary user-defined constraints. With effort constructors and operations in C++ and Java could be used to enforce constraints.</td></tr>
</table>

3-1A. Numeric Values. The language shall provide distinct numeric types for exact and for approximate computation. Numeric operations and assignment that would cause the most significant digits of numeric values to be truncated (e.g., when overflow occurs) shall constitute an exception situation.	yes	**partial**?	**partial**?	**partial**?	All support integers and floats. C, C++, and Java don't raise exceptions on integer overflow. C/C++ implementations often define ways to handle IEEE floating point exceptions. Java does throw an exception for division by zero. The question marks are noted because it's not clear how much a penalty should be assessed for this.
3-1B. Numeric Operations. There shall be built-in operations (i.e., functions) for conversion between the numeric types. There shall be operations for addition, subtraction, multiplication, division, negation, absolute value, and exponentiation to integer powers for each numeric type. There shall be built-in equality (i.e., equal and unequal) and ordering operations (i.e., less than, greater than, less than or equal, and greater than or equal) between elements of each numeric type. Numeric values shall be equal if and only if they have exactly the same abstract value.	yes	mostly	mostly	mostly	In C, C++, and Java the exponentiation operator is pow(), not the usual infix operator, and the built-in operation only takes arguments of type double (not int). C provides an absolute value function for int but not double.
3-1C. Numeric Variables. The range of each numeric variable must be specified in programs and shall be determined by the time of its allocation. Such specifications shall be interpreted as the minimum range to be implemented and as the maximum range needed by the application. Explicit conversion operations shall not be required between numeric ranges.	yes	yes	yes	yes	Counting built-in types (such as "Integer" or "int") as specifying a range, all of these languages do so to some extent. C, C++ and Java do not support user-defined numeric ranges (see 3D).
3-1D. Precision. The precision (of the mantissa) of each expression result and variable in approximate computations must be specified in programs, and shall be determinable during translation. Precision specifications shall be required for each such variable. Such specifications shall be interpreted as the minimum accuracy (not significance) to be implemented. Approximate results shall be implicitly rounded to the implemented precision. Explicit conversions shall not be required between precisions.	yes	**partial**	**partial**	mostly	The standards for C and C++ define the minimum precision of double and float, but no control over actual precision. Java defines specific precisions for double and float, and no other control over precision.
3-1E. Approximate Arithmetic Implementation. Approximate arithmetic will be implemented using the actual precisions, radix, and exponent range available in the object machine. There shall be built-in operations to access the actual precision, radix, and exponent range of the implementation.	yes	yes	yes	**no**	Ada makes the precision, radix, and exponent range available through language-defined attributes. C and C++ make these available through <float.h>. Java defines these in the language itself. Java requires IEEE arithmetic semantics (with specific options) to be used, regardless of the underlying machine's floating point mechanisms.
3-1F. Integer and Fixed Point Numbers. Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.	yes	**partial**	**partial**	mostly	C, C++, and Java don't support fixed point numbers. C++ and Java classes could be used to build fixed point functionality. C and C++ permit implicit truncation in integer computations.
3-1G. Fixed Point Scale. The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable during translation. Scales shall not be restricted to powers of two.	yes	**no**	**no**	**no**	No built-in fixed point support in C, C++, or Java.

3-1H. Integer and Fixed Point Operations. There shall be integer and fixed point operations for modulo and integer division and for conversion between values with different scales. All built-in and predefined operations for exact arithmetic shall apply between arbitrary scales. Additional operations between arbitrary scales shall be definable within programs.	yes	**no**	**no**	**no**
	All support "modulo" operators; C, C++, and Java don't support fixed point numbers.			
3-2A. Enumeration Type Definitions. There shall be types that are definable in programs by enumeration of their elements. The elements of an enumeration type may be identifiers or character literals. Each variable of an enumeration type may be restricted to a contiguous subsequence of the enumeration.	yes	mostly	mostly	**no**
	Java doesn't support enumerations. C enumerations are weakly typed (integers can be freely assigned to them); C++ tightens this slightly (but still permits quiet conversions from enum to int). C and C++ only permit identifiers (not character constants) as enumeration elements. Neither C nor C++ support sub-sequences.			
3-2B. Operations on Enumeration Types. Equality, inequality, and the ordering operations shall be automatically defined between elements of each enumeration type. Sufficient additional operations shall be automatically defined so that the successor, predecessor, the position of any element, and the first and last element of the type may be computed.	yes	mostly?	mostly?	**no**
	C and C++ don't have operations to determine the first and last enumerated value.			
3-2C. Boolean Type. There shall be a predefined type for Boolean values.	yes	**no**	yes	yes
	C doesn't have a "bool" type; C++'s bool type is weakly typed. Both Ada's and Java's boolean type is fully distinct from their integer types.			
3-2D. Character Types. Character sets shall be definable as enumeration types. Character types may contain both printable and control characters. The ASCII character set shall be predefined.	yes	yes	yes	**partial**
	All languages have a predefined character type, though it's not necessarily considered an enumerated type. C and C++ enumeration types can be used to create "character sets", though this is rarely done. Java lacks enumeration types.			
3-3A. Composite Type Definitions. It shall be possible to define types that are Cartesian products of other types. Composite types shall include arrays (i.e., composite data with indexable components of homogeneous types) and records (i.e., composite data with labeled components of heterogeneous type).	yes	yes	yes	yes
	All have arrays and records (Java and C++ classes may be used as records).			
3-3B. Component Specifications. For elements of composite types, the type of each component (i.e., field) must be explicitly specified in programs and determinable during translation. Components may be of any type (including array and record types). Range, precision, and scale specifications shall be required for each component of appropriate numeric type.	yes	yes	yes	yes
	Range, precision, and scale specifications are included in numeric type definitions (with support varying, see 3-1).			

3-3C. Operations on Composite Types. A value accessing operation shall be automatically defined for each component of composite data elements. Assignment shall be automatically defined for components that have alterable values. A constructor operation (i.e., an operation that constructs an element of a type from its constituent parts) shall be automatically defined for each composite type. An assignable component may be used anywhere in a program that a variable of the component's type is permitted. There shall be no automatically defined equivalence operations between values of elements of a composite type.	yes	yes	yes	yes
	The "constructor" meant here is simply the ability to declare or allocate a value of the given type, which all support. Ada, C++, and Java provide more sophisticated control over construction.			
3-3D. Array Specifications. Arrays that differ in number of dimensions or in component type shall be of different types. The range of subscript values for each dimension must be specified in programs and may be determinable at the time of array allocation. The range of each subscript value must be restricted to a contiguous sequence of integers or to a contiguous sequence from an enumeration type.	yes	mostly	mostly	mostly
	C, C++, and Java array indexes may only start at zero and cannot use enumerations to define array subscripts. In C enumerations may be used to access array elements. In C++ enumerations can be cast into int's to access array values, while Java has no enumeration types.			
3-3E. Operations on Subarrays. There shall be built-in operations for value access, assignment, and catenation of contiguous sections of one-dimensional arrays of the same component type. The results of such access and catenation operations may be used as actual input parameter.	yes	**no**	**no**	**no**
	C and C++'s memcpy and memcmp can be used to do some of these operations using an extremely low-level interface. C, C++, and Java do not have array concatenation operators (Java has a string concatenator as a special case).			
3-3F. Nonassignable Record Components. It shall be possible to declare constants and (unary) functions that may be thought of as record components and may be referenced using the same notation as for accessing record components. Assignment shall not be permitted to such components.	**no**	**no**	yes	yes
	C++ and Java classes can include constants (and functions).			
3-3G. Variants. It shall be possible to define types with alternative record structures (i.e., variants). The structure of each variant shall be determinable during translation.	yes	yes	yes	yes
	Java and C++ class structures can be used to simulate at run time the typical uses of variants. C and C++ also permit "unions" to define types with alternative record structures without tag fields.			
3-3H. Tag Fields. Each variant must have a nonassignable tag field (i.e., a component that can be used to discriminate among the variants during execution). It shall not be possible to alter a tag field without replacing the entire variant.	yes	**no**	yes	yes
	Java operations (e.g. instanceof) and C++ RTTI can be used to simulate typical uses of tag fields. Note that C unions do not have automatic tag fields.			

3-3I. Indirect Types. It shall be possible to define types whose elements are indirectly accessed. Elements of such types may have components of their own type, may have substructure that can be altered during execution, and may be distinct while having identical component values. Such types shall be distinguishable from other composite types in their definitions. An element of an indirect type shall remain allocated as long as it can be referenced by the program. [Note that indirect types require pointers and sometimes heap storage in their implementation.]	yes	yes	yes	yes	Ada access values, C/C++ pointers, and Java object references support this. Note that Java requires garbage collection and Ada permits garbage collection as an option (with a pragma for controlling it). C/C++ implementations usually do not include garbage collection, although conservative garbage collection systems for C/C++ are available. C and C++ permit pointers which reference deallocated storage. Ada programs using Unchecked_Deallocation may reference deallocated storage.
3-3J. Operations on Indirect Types. Each execution of the constructor operation for an indirect type shall create a distinct element of the type. An operation that distinguishes between different elements, an operation that replaces all of the component values of an element without altering the element's identity, and an operation that produces a new element having the same component values as its argument, shall be automatically defined for each indirect type.	yes	yes	yes	mostly	Note that the "constructor" mentioned here is "new" (in Ada, C++, and Java) or "malloc" (in C or C++). Note that Ada, C++, and Java (but not C) provide additional control over constructors. Copying isn't defined automatically in Java.
3-4A. Bit Strings (i.e., Set Types). It shall be possible to define types whose elements are one-dimensional Boolean arrays represented in maximally packed form (i.e, whose elements are sets).	yes	**partial**?	yes	yes	In Ada, declare a packed array of boolean values. In C, short bit strings can be handled using "int" or "long", but longer bit strings are best handled through user-defined functions or macros. In C++, use the STL template class bitset (for short ones, use bitmask). In Java, use class BitSet in package "java.util".
3-4B. Bit String Operations. Set construction, membership (i.e., subscription), set equivalence and nonequivalence, and also complement, intersection, union, and symmetric difference (i.e., component-by-component negation, conjunction, inclusive disjunction, and exclusive disjunction respectively) operations shall be defined automatically for each set type.	yes	**partial**?	yes	mostly	In C, such operations can be easily created with the built-in operations when there are sizeof(long) or fewer bits; longer bit strings are typically handled by user-defined functions or macros. Java BitSet doesn't have a group negation ("not") operation.
3-5A. Encapsulated Definitions. It shall be possible to encapsulate definitions. An encapsulation may contain declarations of anything (including the data elements and operations comprising a type) that is definable in programs. The language shall permit multiple explicit instantiations of an encapsulation.	yes	yes	yes	yes	Ada's unit of encapsulation is the package. C's is the ".h" file. C++'s are classes and ".h" files. Java's is the class.
3-5B. Effect of Encapsulation. An encapsulation may be used to inhibit external access to implementation properties of the definition. In particular, it shall be possible to prevent external reference to any declaration within the encapsulation including automatically defined operations such as type conversions and equality. Definitions that are made within an encapsulation and are externally accessible may be renamed before use outside the encapsulation.	yes	**partial**	yes	yes	C encapsulation requires extreme discipline using the "static" keyword (the default is to make everything globally accessible). C++ more strongly supports encapsulation when classes and the private modifier are used.

3-5C. Own Variables. Variables declared within an encapsulation, but not within a function, procedure, or process of the encapsulation, shall remain allocated and retain their values throughout the scope in which the encapsulation is instantiated.	yes	yes	yes	yes
4A. Form of Expressions. The parsing of correct expressions shall not depend on the types of their operands or on whether the types of the operands are built into the language.	yes	yes	yes	yes
4B. Type of Expressions. It shall be possible to specify the type of any expression explicitly. The use of such specifications shall be required only where the type of the expression cannot be uniquely determined during translation from the context of its use (as might be the case with a literal).	yes	yes	yes	yes
	Ada qualifiers do this. C, C++, and Java "casts" can do this, but may also quietly invoke a conversion operation.			
4C. Side Effects. The language shall attempt to minimize side effects in expressions, but shall not prohibit all side effects. A side effect shall not be allowed if it would alter the value of a variable that can be accessed at the point of the expression. Side effects shall be limited to own variables of encapsulations. The language shall permit side effects that are necessary to instrument functions and to do storage management within functions. The order of side effects within an expression shall not be guaranteed. [Note that the latter implies that any program that depends on the order of side effects is erroneous.]	mostly	**partial**	mostly	mostly
	All permit side effects beyond their own variables of encapsulation (e.g. global variables or other objects can be affected). C encourages this, and combined with macros can cause unexpected results (e.g. "putchar(*p++)"). Such use is not necessarily considered erroneous in C/C++. C++ permits the same effects, though due to other language features (such as OO features) they tend to receive less use.			
4D. Allowed Usage. Expressions of a given type shall be allowed wherever both constants and variables of the type are allowed.	yes	yes	yes	yes
4E. Translation Time Expressions. Expressions that can be evaluated during translation shall be permitted wherever literals of the type are permitted. Translation time expressions that include only literals and the use of translation time facilities (see 11C) shall be evaluated during translation.	yes	yes	yes	yes
	C, C++, and Java specifications do not require all literals to be evaluated at compile time, but compilers typically do so.			
4F. Operator Precedence Levels. The precedence levels (i.e., binding strengths) of all (prefix and infix) operators shall be specified in the language definition, shall not be alterable by the user, shall be few in number, and shall not depend on the types of the operands.	yes	mostly	mostly	mostly
	C, C++, and Java have a large number of precedence levels.			

4G. Effect of Parentheses. If present, explicit parentheses shall dictate the association of operands with operators. The language shall specify where explicit parentheses are required and shall attempt to minimize the psychological ambiguity in expressions. [Note that this might be accomplished by requiring explicit parentheses to resolve the operator-operand association whenever a nonassociative operator appears to the left of an operator of the same precedence at the least-binding precedence level of any subexpression.]	yes	yes	yes	yes	
5A. Declarations of Constants. It shall be possible to declare constants of any type. Such constants shall include both those whose values-are determined during translation and those whose value cannot be determined until allocation. Programs may not assign to constants.	yes	yes	yes	yes	
5B. Declarations of Variables. Each variable must be declared explicitly. Variables may be of any type. The type of each variable must be specified as part of its declaration and must be determinable during translation. [Note, "variable" throughout this document refers not only to simple variables but also to composite variables and to components of arrays and records.]	yes	mostly	mostly	yes	C and C++ permit "void *" as a type, which is really a pointer to an unknown type and subverts the type system.
5C. Scope of Declarations. Everything (including operators) declared in a program shall have a scope (i.e., a portion of the program in which it can be referenced). Scopes shall be determinable during translation. Scopes may be nested (i.e., lexically embedded). A declaration may be made in any scope. Anything other than a variable shall be accessable within any nested scope of its definition.	yes	**partial**	mostly	mostly	All support nested scopes of variable declarations. Ada supports hierarchical packages, nested packages, and nested subprograms. C only provides two scope levels for functions: static and non-static. Java and C++ support class scoping mechanisms (private, protected, and public) and larger structuring mechanisms (C++ namespaces and Java nested packages). Neither Java nor C++ support nested functions (functions declared in other functions).
5D. Restrictions on Values. Procedures, functions, types, labels, exception situations, and statements shall not be assignable to variables, be computable as values of expressions, or be usable as nongeneric parameters to procedures or functions.	**no**	**no**	**no**	**no**	Ada, C, and C++ support the use of access/pointer to subprograms/functions. The original Ada83 did not permit access to subprograms; this was later found to be too limiting. Ada and C++ permit passing of exceptions as values. Java does not support pointers to functions, though Java interfaces can be used as a somewhat clumsy workaround. Java permits types and exceptions to be assigned and used as nongeneric parameters.
5E. Initial Values. There shall be no default initial-values for variables.	**partial**	yes	yes	**partial**	Ada defines an initial value for access values. Java defines initial values for all types, though recommends against using them and Java compilers attempt to warn of such use. In both Java and Ada, this is to support reliability.
5F. Operations on Variables. Assignment and an implicit value access operation shall be automatically defined for each variable.	yes	yes	yes	yes	

5G. Scope of Variables. The language shall distinguish between open scopes (i.e., those that are automatically included in the scope of more globally declared variables) and closed scopes (i.e., those in which nonlocal variables must be explicitly Imported). Bodies of functions, procedures, and processes shall be closed scopes. Bodies of classical control structures shall be open scopes.	yes	yes	yes	yes
	The languages differ significantly in their notions of "importing" and how scoping is handled, but all support the essence of this requirement.			
6A. Basic Control Facility. The (built-in) control mechanisms should be of minimal number and complexity. Each shall provide a single capability and shall have a distinguishing syntax. Nesting of control structures shall be allowed. There shall be no control definition facility. Local scopes shall be allowed within the bodies of control statements. Control structures shall have only one entry point and shall exit to a single point unless exited via an explicit transfer of control (where permitted, see 6G), or the raising of an exception (see 10C).	yes	yes	yes	yes
	C, C++, and Java have both "continue" and "break" operations, which could be viewed as two "exit" points from a control structure. Java has a multi-level break statement, but these goes to specific exit points in specific control structures.			
6B. Sequential Control. There shall be a control mechanism for sequencing statements. The language shall not impose arbitrary restrictions on programming style, such as the choice between statement terminators and statement separators, unless the restriction makes programming errors less likely.	yes	yes	yes	yes
	All use statement terminators.			
6C. Conditional Control. There shall be conditional control structures that permit selection among alternative control paths. The selected path may depend on the value of a Boolean expression, on a computed choice among labeled alternatives, or on the true condition in a set of conditions. The language shall define the control action for all values of the discriminating condition that are not specified by the program. The user may supply a single control path to be used when no other path is selected. Only the selected branch shall be compiled when the discriminating condition is a translation time expression.	yes	yes	yes	yes
6D. Short Circuit Evaluation. There shall be infix control operations for short circuit conjunction and disjunction of the controlling Boolean expression in conditional and iterative control structures.	yes	yes	yes	yes
6E. Iterative Control. There shall be an iterative control structure. The iterative control may be exited (without reentry) at an unrestricted number of places. A succession of values from an enumeration type or the integers may be associated with successive iterations and the value for the current iteration accessed as a constant throughout the loop body.	yes	mostly	mostly	mostly
	In C, C++, and Java, the loop control variable is not considered a constant.			

6G. Explicit Control Transfer. There shall be a mechanism for control transfer (i.e., the go to). It shall not be possible to transfer out of closed scopes, into narrower scopes, or into control structures. It shall be possible to transfer out of classical control structures. There shall be no control transfer mechanisms in the form of switches, designational expressions, label variables, label parameters, or alter statements.	yes	yes	yes	**partial**	The Java language does not include the "goto" (though it does reserve the keyword). Java does have a multi-level break and continue statement which can serve the role of "goto" in many cases. K&R does not list any restrictions on C's goto statement, so some implementations may permit entry into control structures.
7A. Function and Procedure Definitions. Functions (which return values to expressions) and procedures (which can be called as statements) shall be definable in programs. Functions or procedures that differ in the number or types of their parameters may be denoted by the same identifier or operator (i.e., overloading shall be permitted). [Note that redefinition, as opposed to overloading, of an existing function or procedure is often error prone.]	yes	**no**	yes	yes	C does not permit multiple functions to share the same name even when the parameter signatures differ.
7B. Recursion. It shall be possible to call functions and procedures recursively.	yes	yes	yes	yes	
7C. Scope Rules. A reference to an identifier that is not declared in the most local scope shall refer to a program element that is lexically global, rather than to one that is global through the dynamic calling structure.	yes	yes	yes	yes	
7D. Function Declarations. The type of the result for each function must be specified in its declaration and shall be determinable during translation. The results of functions may be of any type. If a result is of a nonindirect array or record type then the number of its components must be determinable by the time of function call.	mostly	mostly	mostly	mostly	Ada permits any type as a function return value, and does not require the number of components to be determined by function call time (which gives flexibility at the cost of efficiency when using this capability). Java also permits a function to return an array without knowning the number of components at call time; Java's approach to arrays is different than that implied by this requirement.
7F. Formal Parameter Classes. There shall be three classes of formal data parameters: (a) input parameters, which act as constants that are initialized to the value of corresponding actual parameters at the time of call, (b) input-output parameters, which enable access and assignment to the corresponding actual parameters, either throughout execution or only upon call and prior to any exit, and (c) output parameters, whose values are transferred to the corresponding actual parameter only at the time of normal exit. In the latter two cases the corresponding actual parameter shall be determined at time of call and must be a variable or an assignable component of a composite type.	yes	**partial**	**partial**	**no**	C, C++, and Java do not identify in, out, and in-out parameters. C and C++ can identify in-only parameters using "const" or by using non-pointer types. C++ supports passing by reference as well as passing by value (which implies that the item already exists).

7G. Parameter Specifications. The type of each formal parameter must be explicitly specified in programs and shall be determinable during translation. Parameters may be of any type. The language shall not require user specification of subtype constraints for formal parameters. If such constraints are permitted they shall be interpreted as assertions and not as additional overloading. Corresponding formal and actual parameters must be of the same type.	yes	mostly	mostly	yes
	C and C++ permit re-specification in other places, permitting the specifications to go "out of sync". C and C++ also permit recasting of pointers that can subvert the type specification.			
7H. Formal Array Parameters. The number of dimensions for formal array parameters must be specified in programs and shall be determinable during translation. Determination of the subscript range for formal array parameters may be delayed until invocation and may vary from call to call. Subscript ranges shall be accessible within function and procedure bodies without being passed as explicit parameters.	yes	**no**	**no**	**partial**?
	Subscript ranges are not accessible in C and C++. C, C++, and Java don’t support multidimension arrays, though arrays of arrays permit some similar operations (particularly in Java).			
7I. Restrictions to Prevent Aliasing. The language shall attempt to prevent aliasing (l.e., multiple access paths to the same variable or record component) that is not intended, but shall not prohibit all aliasing. Aliasing shall not be permitted between output parameters nor between an input-output parameter and a nonlocal variable. Unintended aliasing shall not be permitted between input-output parameters. A restriction limiting actual input-output parameters to variables that are nowhere referenced as nonlocals within a function or routine, is not prohibited. All aliasing of components of elements of an indirect type shall be considered intentional.	yes	**no**	**no**	**no**
	Aliasing is a well-known problem when trying to optimize C and C++ code. Java defines all non-primitives as references and does not permit "internal" references, so in some sense all aliases are intended.			
8A. Low Level Input-Output. There shall be a few low level input-output operations that send and receive control information to and from physical channels and devices. The low level operations shall be chosen to insure that all user level input-output operations can be defined within the language.	mostly?	**partial**	**partial**	**no**
	Ada, C, and C++ permit access to memory-mapped locations but do not have standard I/O channel operations. Ada’s machine code insertion capability can perform I/O channel operations in the language.			
8B. User Level Input-Output. The language shall specify (i.e., give calling format and general semantics) a recommended set of user level input-output operations. These shall include operations to create, delete, open, close, read, write, position, and interrogate both sequential and random access files and to alter the association between logical files and physical devices.	yes	yes	yes	yes
	Ada and Java applets may be restricted further by the environment, but this is determined by the local applet security manager and user, not by the language.			
8C. Input Restrictions. User level input shall be restricted to data whose record representations are known to the translator (i.e., data that is created and written entirely within the program or data whose representation is explicitly specified in the program).	yes	yes	yes	yes

8D. Operating System Independence. The language shall not require the presence of an operating system. [Note that on many machines it will be necessary to provide run-time procedures to implement some features of the language.]	yes	yes	yes	yes
8E. Resource Control. There shall be a few low level operations to interrogate and control physical resources (e.g., memory or processors) that are managed (e.g., allocated or scheduled) by built-in features of the language.	mostly	**no**	**partial**	**partial**
	Ada supports memory storage pool management (managed by the allocation/deallocation language features), task scheduling policies, and task priorities. The C++ "new" operator can be overridden to support memory storage pool management. Java supports thread priorities.			
8F. Formating. There shall be predefined operations to convert between the symbolic and internal representation of all types that have literal forms in the language (e.g., strings of digits to integers, or an enumeration element to its symbolic form). These conversion operations shall have the same semantics as those specified for literals in programs.	yes	**partial**	**partial**	**partial**
	C and C++ don't have built-in enumeration reading and writing; Java doesn't have enumerated types.			
9A. Parallel Processing. It shall be possible to define parallel processes. Processes (i.e., activation instances of such a definition) may be initiated at any point within the scope of the definition. Each process (activation) must have a name. It shall not be possible to exit the scope of a process name unless the process is terminated (or uninitiated).	yes	**no**	**no**	yes
	C and C++ do not have have built-in thread or process facilities - the assumption is that these are to be provided by operating system dependent libraries (such as the POSIX p-threads library). Java's parallel processing facilities differ in approach from the wording of this requirement, but can provide these facilities.			
9B. Parallel Process Implementation. The parallel processing facility shall be designed to minimize execution time and space. Processes shall have consistent semantics whether implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor.	mostly	**no**	**no**	mostly
	Both Ada and Java leave some semantics open to permit efficient implementation on different operating systems.			
9C. Shared Variables and Mutual Exclusion. It shall be.possible to mark variables that are shared among parallel processes. An unmarked variable that is assigned on one path and used on another shall cause a warning. It shall be possible efficiently to perform mutual exclusion in programs. The language shall not require any use of mutual exclusion.	**partial**	**no**	**no**	**partial**
	Neither Ada nor Java require shared variables to be marked with an attendant warning. Ada 83's obsolete pragma shared doesn't really meet this requirement. Both Ada and Java support shared variables and high-efficiency locking (using protected types/synchronized guards), and both permit the circumventing of such if the programmer determines it to be necessary. Ada, C, and C++ support marking variables as "volatile", and Ada supports marking variables as "atomic".			

9D. Scheduling. The semantics of the built-in scheduling algorithm shall be first-in-first-out within priorities. A process may alter its own priority. If the language provides a default priority for new processes it shall be the priority of its initiating process. The built-in scheduling algorithm shall not require that simultaneously executed processes on different processors have the same priority. [Note that this rule gives maximum scheduling control to the user without loss of efficiency. Note also that priority specification does not impose a specific execution order among parallel paths and thus does not provide a means for mutual exclusion.]	yes	**no**	**no**	mostly	Java in general runs the highest priority thread, but permits occasional running of lower priority threads (see "http://java.sun.com/Series/Tutorial/java/threads/priority.html"). Java does not guarantee first-in first-out within a priority.
9E. Real Time. It shall be possible to access a real time clock. There shall be translation time constants to convert between the implementation units and the program units for real time. On any control path, it shall be possible to delay until at least a specified time before continuing execution. A process may have an accessible clock giving the cumulative processing time (i.e., CPU time) for that process.	yes	**no**	**no**	yes	C/C++ provide some functions for handling local and calendar time; real-time calls are operating system dependent. Java supports delays in centiseconds and clock access in milliseconds.
9G. Asynchronous Termination. It shall be possible to terminate another process. The terminated process may designate the sequence of statements it will execute in response to the induced termination.	**partial**	**no**	**no**	yes	C/C++ programs can call an OS-dependent library to perform this task. Ada and Java permit asynchronous termination (via the abort statement and stop() call respectively). Ada does not permit statement sequences to be run on termination in general (though asynchronous transfer of control and the terminate alternative can permit this in some cases). Java can do this by catching Error ThreadDeath.
9H. Passing Data. It shall be possible to pass data between processes that do not share variables. It shall be possible to delay such data transfers until both the sending and receiving processes have requested the transfer.	yes	**no**	**no**	yes	Ada rendezvous and Java synchronized calls permit controlled passing of data.
9I. Signalling. It shall be possible to set a signal (without waiting), and to wait for a signal (without delay, if it is already set). Setting a signal, that is not already set, shall cause exactly one waiting path to continue.	mostly	**no**	**no**	mostly	Java objects can be used as synchronized guards. Ada does not have a "signal" type, but protected types can trivially implement them (see Ada LRM D.12 for an example). C and C++ have a file "signal.h" but this file does not provide this functionality.
9J. Waiting. It shall be possible to wait for, determine, and act upon the first completed of several wait operations (including those used for data passing, signalling, and real time).	mostly	**no**	**no**	mostly?	Ada select statement supports such capabilities. Java does not have an equivalent structure, but intermediate structures could be used to easily implement such functionality.

10A. Exception Handling Facility. There shall be an exception handling mechanism for responding to unplanned error situations detected in declarations and statements during execution. The exception situations shall include errors detected by hardware, software errors detected during execution, error situations in built-in operations, and user defined exceptions. Exception identifiers shall have a scope. Exceptions should add to the execution time of programs only if they are raised.	yes	**no**	yes	yes
	C's "signal.h" and setjmp/longjmp can be used to handle some exceptions, but don't really satisfy these requirements.			
10B. Error Situations. The errors detectable during execution shall include exceeding the specified range of an array subscript, exceeding the specified range of a variable, exceeding the implemented range of a variable, attempting to access an uninitialized variable, attempting to access a field of a variant that is not present, requesting a resource (such as stack or heap storage) when an insufficient quantity remains, and failing to satisfy a program specified assertion. [Note that some are very expensive to detect unless aided by special hardware, and consequently their detection will often be suppressed (see 10G).]	mostly	**partial**	**partial**	mostly
	None normally detect uninitialized variables access. Ada's Normalize_Scalars pragma aids in detecting uninitialized variables. Java attempts to detect uninitialized variables at compile-time. Ada and Java don't have assertions built into the language, while C and C++ can detect assertion errors. C and C++ don't detect out-of-bound array accesses. C, C++, and Java don't detect range errors (of either kind) nor overflow. All can detect out-of-memory errors.			
10C. Raising Exceptions. There shall be an operation that raises an exception. Raising an exception shall cause transfer of control to the most local enclosing exception handler for that exception without completing execution of the current statement or declaration, but shall not of itself cause transfer out of a function, procedure, or process. Exceptions that are not handled within a function or procedure shall be raised again at the point of call in their callers. Exceptions that are not handled within a process shall terminate the process. Exceptions that can be raised by built-in operations shall be given in the language definition.	yes	**no**	yes	yes
	C has an assert macro, but it doesn't have the kind of enclosure described here.			
10D. Exception Handling. There shall be a control structure for discriminating among the exceptions that can occur in a specified statement sequence. The user may supply a single control path for all exceptions not otherwise mentioned in such a discrimination. It shall be possible to raise the exception that selected the current handler when exiting the handler.	yes	**no**	yes	yes
10E. Order of Exceptions. The order in which exceptions in different parts of an expression are detected shall not be guaranteed by the language or by the translator.	yes	**no**	yes	yes

10F. Assertions. It shall be possible to include assertions in programs. If an assertion is false when encountered during execution, it shall raise an exception. It shall also be possible to include assertions, such as the expected frequency for selection of a conditional path, that cannot be verified. [Note that assertions can be used to aid optimization and maintenance.]	**no**, not built-in	mostly	mostly	**no**, not built-in
	C and C++ include a simple assert() facility. Neither Ada nor Java have a built-in assert checking facility, though they can be trivially implemented. GNAT Ada compiler has pragma assert, but this is compiler-specific. None permit simple assertions of frequency.			
10G. Suppressing Exceptions. It shall be possible during translation to suppress individually the execution time detection of exceptions within a given scope. The language shall not guarantee the integrity of the values produced when a suppressed exception occurs. [Note that suppression of an exception is not an assertion that the corresponding error will not occur.]	yes	**no**	**no**	**no**
11A. Data Representation. The language shall permit but not require programs to specify a single physical representation for the elements of a type. These specifications shall be separate from the logical descriptions. Physical representation shall include object representation of enumeration elements, order of fields, width of fields, presence of "don't care" fields, positions of word boundaries, and object machine addresses. In particular, the facility shall be sufficient to specify the physical representation of any record whose format is determined by considerations that are entirely external to the program, translator, and language. The language and its translators shall not guarantee any particular choice for those aspects of physical representation that are unspecified by the program. It shall be possible to specify the association of physical resources (e.g., interrupts) to program elements (e.g., exceptions or signals).	yes	**partial**	**partial**	**no**
	C and C++ bitfields provide some data representation control, but don't control big-endian/little-endianness. Lack of endianness control makes control over portable representation of lower-level constructs very difficult. C and C++ also don't provide mechanisms to control the exact bit size of basic types, nor hooks to interrupts. Java does not provide representation control.			
11C. Translation Time Facilities. To aid conditional compilation, it shall be possible to interrogate properties that are known during translation including characteristics of the object configuration, of function and procedure calling environments, and of actual parameters. For example, it shall be possible to determine whether the caller has suppressed a given exception, the callers optimization criteria, whether an actual parameter is a translation time expression, the type of actual generic parameters, and the values of constraints characterizing the subtype of actual parameters.	**partial**	**partial**	**partial**	**no**
	Ada, C, and C++ all provide some mechanisms to query the compilation environment, though not to the extent given in this requirement.			

11D. Object System Configuration. The object system configuration must be explicitly specified in each separately translated unit. Such specifications must include the object machine model, the operating system if present, peripheral equipment, and the device configuration, and may include special hardware options and memory size. The translator will use such specifications when generating object code. [Note that programs that depend on the specific characteristics of the object machine, may be made more portable by enclosing those portions in branches of conditionals on the object machine configuration.]	**partial**	**no**	**no**	**no**?	Ada requires that separately compiled modules be compatible when linked together, implying some of the requirements here. Java is designed to make this generally unnecessary, by translating to system-independent bytecodes first, so it’s arguable if this requirement applies to Java.
11E. Interface to Other Languages. There shall be a machine independent interface to other programming languages including assembly languages. Any program element that is referenced in both the source language program and foreign code must be identified in the interface. The source language of the foreign code must also be identified.	yes	**partial**	yes	**partial**	Ada has standard interfaces to C, Fortran, COBOL, and machine language, and standard pragmas Import, Export, and Convention for interfacing to other languages. Some C implementations support the "asm" keyword. C has little support for interfacing to other languages, but on many systems it is the "standard" host language and serves as a common interface standard for other languages. C++ has a general external linkage system using extern "language", though often only C is supported as the external language. Java has an external link to C.
11F. Optimization. Programs may advise translators on the optimization criteria to be used in a scope. It shall be possible in programs to specify whether minimum translation costs or minimum execution costs are more important, and whether execution time or memory space is to be given preference. All such specifications shall be optional. Except for the amount of time and space required during execution, approximate values beyond the specified precision, the order in which exceptions are detected, and the occurrence of side effects within an expression, optimization shall not alter the semantics of correct programs, (e.g., the semantics of parameters will be unaffected by the choice between open and closed calls).	yes	**partial**?	**partial**?	**no**?	Ada programs can specify whether to optimize for speed or space. C and C++ code cannot make such specifications, but do provide the "register" keyword to provide optimization hints. Most compilers support external optimization flags.
12A. Library. There shall be an easily accessible library of generic definitions and separately translated units. All predefined definitions shall be in the library. Library entries may include those used as input-output packages, common pools of shared declarations, application oriented software packages, encapsulations, and machine configuration specifications. The library shall be structured to allow entries to be associated with particular applications, projects, and users.	yes	yes	yes	yes	All provide mechanisms to make reusable components available to a library.
12B. Separately Translated Units. Separately translated units may be assembled into operational systems. It shall be possible for a separately translated unit to reference exported definitions of other units. All language imposed restrictions shall be enforced across such interfaces. Separate translation shall not change the semantics of a correct program.	yes	**partial**	mostly	yes	All support separate compilation. C separately compiled units need not agree on their interface. This is true for C++ as well, but C++ programs using classes and header files in normal ways obtain most such protection.

12D. Generic Definitions. Functions, procedures, types, and encapsulations may have generic parameters. Generic parameters shall be instantiated during translation and shall be interpreted in the context of the instantiation. An actual generic parameter may be any defined identifier (including those for variables, functions, procedures, processes, and types) or the value of any expression.	yes	**no**	mostly	**no**
	Ada types cannot be directly made generic, but can be generic via encapsulation. C/C++'s #define preprocessor is not sufficiently powerful to meet these requirements. C++'s templates provide this capability, though weaknesses and different semantics render C++ templates less useful at this time. Java lacks this ability.			
13A. Defining Documents. The language shall have a complete and unambiguous defining document. It should be possible to predict the possible actions of any syntactically correct program from the language definition. The language documentation shall include the syntax, semantics, and appropriate examples of each built-in and predefined feature. A recommended set of translation diagnostic and warning messages shall be included in the language definition.	yes	mostly	mostly	yes
	The C and C++ defining documents include a very large number of undefined results.			
13B. Standards. There will be a standard definition of the language. Procedures will be established for standards control and for certification that translators meet the standard.	yes	yes	**partial**	**no**
	Official standards are available for Ada and C. There is no official C++ standard, but work to develop one is ongoing. Work to standardize Java is at an extremely early stage.			
13C. Completeness of Implementations. Translators shall implement the standard definition. Every translator shall be able to process any syntactically correct program. Every feature that is available to the user shall be defined in the standard, in an accessible library, or in the source program.	mostly?	yes in practice	mostly?	yes
	Some Ada compilers have not completed their transition to Ada95; the Ada validation process (including the ACVC test suite) helps to ensure that Ada compilers implement the entire Ada language. C implementations need not implement the entire language, but production compilers generally do so. Since the definition of C++ is changing, C++ compilers are mostly complete as of some version of the C++ standard. Note that all compilers have bugs, so none can truly process "any" correct program.			
13D. Translator Diagnostics. Translators shall be responsible for reporting errors that are detectable during translation and for optimizing object code. Translators shall be responsible for the integrity of object code in affected translation units when any separately translated unit is modified, and shall ensure that shared definitions have compatible representations in all translation units. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and should provide warnings where constructs will be dangerous or unusually expensive in execution and shall attempt to detect exceptions during translation. If the translator determines that a call on a routine will not terminate normally, the exception shall be reported as a translation error at the point of call.	yes	**partial**	**partial**?	yes
	All provide at least some error reporting and warnings, all have optimizing compilers, and all permit separate compilation. Translator characteristics for detecting errors in the presence of separate compilation are undefined by C and C++. Shared definitions are not required in C; they are generally used in C++. Many C compilers do not do full type checking; lint (where available) can supplement checking.			

13E. Translator Characteristics. Translators for the language will be written in the language and will be able to produce code for a variety of object machines. The machine independent parts of translators should be separate from code generators. Although it is desirable, translators need not be able to execute on every object machine. The internal characteristics of the translator (i.e., the translation method) shall not be specified by the language definition or standards.	mostly	mostly	mostly	mostly
	Many compilers are implemented in their own language, though not all. Java implementations may have two code production stages, one that generates virtual machine code and a second that converts virtual machine code to a specific machine's code.			
13F. Restrictions on Translators. Translators shall fail to translate otherwise correct programs only when the program requires more resources during translation than are available on the host machine or when the program calls for resources that are unavailable in the specified object system configuration. Neither the language nor its translators shall impose arbitrary restrictions on language features. For example, they shall not impose restrictions on the number of array dimensions, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels.	yes	**partial**	yes	yes
	The C language definition still permits externally-visible identifiers to be considered identical if the first 6 characters are equal ignoring case (as a concession to old linkers). C++ recommends large maximums (see "Implementation Quantities"), though these are not mandated. Credit is given if the limits are sufficiently large that encountering them is very unlikely.			
13G. Software Tools and Application Packages. The language should be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and will include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them.	yes	yes	yes	yes
	Many tools exist for all of these languages (particularly for C and C++) from a wide variety of vendors.			