ISO/IEC JTC 1/SC 22/OWGV N 0121

Forms of language specifications: Examples from commonly used computer languages

Date	14 February 2008
Contributed by	Derek Jones
Original file name	langconform.pdf
Notes	Replaces N0078

ISO/IEC JTC 1/SC 22/OWGV N 0121

Forms of language specifications: Examples from commonly used computer languages

Date	14 February 2008
Contributed by	Derek Jones
Original file name	langconform.pdf
Notes	Replaces N0078

Forms of language specification

Examples from commonly used computer languages

Derek M. Jones
derek@knosof.co.uk

1 Introduction

A programming language specification has to define all of the constructs it supports and their possible behaviors. There are a variety of different structural forms that can be used in a specification to enumerate these constructs and express their behaviors.

This paper discusses the different structural forms that have been used to specify some of the programming languages of interest to the ISO language vulnerability working group (OWGV).

OWGV are working to produce a set of language independent guidelines (it is intended that these be adapted by users of various languages to suit their specific needs). Because no two language specifications have identical forms, often using different English phrases to express the same intended requirement, or giving different interpretations to the same phrase, people unfamiliar with a language are likely to find it difficult to grasp the full implications of wording contained in its specification. It is hoped that this paper will help by providing an introduction to some of the more common language specification techniques and a summary of the techniques and phrases used by various languages of interest to OWGV.

Any proposal for the creation a language specific version of the generic guidelines will need to understand the form of specification used and how it is possible to extract the necessary information from the language's specification. It is expected that those responsible for framing language specific guidelines will be very familiar with their chosen language and be able to map the OWGV guidelines into a form that is applicable.

Another factor that needs to be considered before creating a language specific version of the OWGV guidelines is the quality of its specification. This will also need to be evaluated to uncover any areas, relevant to the guidelines, where they may be uncertainty about the accuracy or completeness of the information.

2 Methods for specifying languages

Existing language specifications have been structured either in terms of an actual implementation or in some prose style listing the requirements.

- specification written in a form amenable to machine execution. Here the behavior of what is essentially a particular implementation constitutes the complete specification. An example of a language that uses this kind of definition is PERL.[8] Also included in this category are language definitions written using a formal mathematical notation; these are essentially implementations even if no program capable of executing them currently exists. This form of specification can be subdivided into two approaches:
  - model implementation. Here the primary intent is to clearly define the language and not be unduly concerned with runtime efficiency. These have been created for Pascal[9] and C[6]
  - production use implementation. Intended to be used to execute programs in a production environment and not be unduly concerned with making it easy for readers to extract language requirements from the source.

- specification by use of prose, often English. This form of specification can also be subdivided into two approaches:
  - specification of how an implementation is to behave (e.g., C++). The behavior of source code constructs has to be inferred from the specification of the implementation,
  - specification of the properties and behaviors of source code constructs (e.g., C, Fortran and Java). The behavior of an implementation has to be inferred from the specification of source code behavior.

As specification methods prose and implementation both have advantages and disadvantages. From the OWGV perspective languages specified using the prose approach are likely to require much less effort to deal with, because the necessary requirements will already have been abstracted out and be easily located.

However, OWGV is likely to be dealing with existing languages whose form of specification has already been chosen.

In the case of the Java language it is not clear whether its specification is the book whose title is "The Java Language Specification",[2] Sun Microsystems particular implementation, java.sun.com, or the associated compatibility test suite. The prose specification does not explicitly use any stylised phrases to express requirements, like those that appear in other language specifications (see the section "Some requirements wording" for requirements wording extracted from ISO standards).

Some languages were defined by specification where it was not expected that other implementations would be created. For instance, the source code of the PERL implementation has always been made available (even before the term open source existed). As such it was not expected that anybody else would want to produce a competing implementation.

The PHP group, www.php.net/credits.php, claim that they have the final say in the specification of (the language) PHP. This group's specification is an implementation, and there is no prose specification or agreed validation suite. There are alternative implementations (e.g., Quercus www.caucho.com/resin-3.0/quercus, which is written in Java, and Roadsend, www.roadsend.com, a native code compiler) that claim to be compatible (they don't say what this means) with some version of PHP.

2.1 Quality of a specification

A number of measures might be used to judge the quality of a specification, including the following:

- the ease with which a third party can create a viable conforming implementation based purely on the information contained in the specification. A viable conforming implementation is one that both conforms to the requirements contained in the specification and is capable of translating/executing the source code of many existing programs.
- the number of defects that have been found in the specification. The ISO Defect Reports provide a measure of the number of inconsistencies and unintended behaviors founds in language standards. In the case of the Ada95 Standard 215 DRs were filed of which 116 lead to changes appearing in Ada99, and for the C90 Standard 178 DRs were filed, although many contained multiple issues, of which around half lead to changes appearing in C99. There are factors driving the discovery of defects in a language other than the number of defects contained in the specification, , e.g., the number of people who carefully read the specification (the greater the number of people the greater the likelihood that an existing defect will be discovered). Some defects might be considered less important because they occur in obscure corners of a language that is rarely used. This issue is outside the scope of this discussion.
- the amount of effort that went into its production. This might also be a measure of commercial interest in the language, with the more commercial languages attracting more people to standard's meetings. Alternatively it might be a measure of the contention, resulting in lots of meetings to resolve issues, that a language generates (it is estimated[7] that 50 man years went into the creation of the first C Standard).

From the OWGV's perspective the most important quality attribute is the ease with which it is possible to extract reliable information about those constructs addressed by the generic guidelines. If it is not possible to obtain this information, and be certain of its correctness, it is unlikely to be worthwhile creating a set of language specific guidelines for it.

2.2 Phrases in prose specifications

When a language specification is written in English its requirements invariable make use of English words and phrases that are intended to be interpreted as permissions, prohibitions, requirements, options, etc.

The English imperative mood is one way of expressing a prohibition or that some condition must be met. Experience has shown that different people can interpret the same sentences in different ways, with some considering it to be in the imperative mood and others not.

A common method of avoiding the ambiguities often present in conventionally phrased English sentences is to make use of stylised phrases having a specified meaning. For instance, *X is expected to be greater than zero*, *X is always greater than zero*, *X must be greater than zero*, or *X shall be greater than zero*.
The following subsections briefly discuss some of the standard's documents that have codified how these behavioral and requirements phrases are to be used and interpreted.

2.2.1 ISO rules

The ISO/IEC directives[5] (Part 2, Annex H "Verbal forms for the expression of provisions") list normative requirements on the phrases to be used to express various kinds of intent.

- Verbal forms: **shall**, **shall not**. "... requirements strictly to be followed in order to conform to the document and from which no deviation is permitted." ... "Do not use "must" as an alternative for "shall". (This will avoid any confusion between the requirements of a document and external statutory obligations.) Do not use "may not" instead of "shall not" to express a prohibition. To express a direct instruction, for example referring to steps to be taken in a test method, use the imperative mood in English."

- Verbal forms: **should**, **should not**. "... that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others, or that a certain course of action is preferred but not necessarily required, or that (in the negative form) a certain possibility or course of action is deprecated but not prohibited."

- Verbal forms: **may**, **may not**. "... a course of action permissible within the limits of the document." "Do not use "possible" or "impossible" in this context. Do not use "can" instead of "may" in this context. NOTE 1 "May" signifies permission expressed by the document, whereas "can" refers to the ability of a user of the document or to a possibility open to him/her."

- Verbal forms: **can**, **cannot**. "... for statements of possibility and capability, whether material, physical or causal."

The ISO Technical Report "Guidelines for the preparation of conformity clauses in programming language standards"[3] only provides a general overview of some of the issues that need to be considered (e.g., processor dependencies, errors, and extensions to the language). Its scope says "It was not considered practical to provide model statements that would be suitable for inclusion in all language standards." This TR does not attempt to defined terms such as *shall* or many of the other terms discussed in the paper.
Broader coverage on writing a programming language standard is provided by the ISO Technical Report "Guidelines for the preparation of programming language standards".[4] This gives guidelines for the form and content of standards, and for the specification of various error conditions.

2.2.2 RFC 2119

The subject of the Internet Task Force RFC 2119[1] is "Key words for use in RFCs to Indicate Requirement Levels". The relevant wording is:

RFC 2119

In many standards track documents several words are used to signify the requirements in the specification. These words are often capitalized. This document defines these words as they should be interpreted in IETF documents. Authors who follow these guidelines should incorporate this phrase near the beginning of their document:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

2.2.3 Document specific

While a language standard might (implicitly) claim to follow the requirements wording specified by some document, experience suggests that it is necessary to check what a document actually says.

For instance, the ISO C language Standard does not follow the exact requirements wording specified by ISO. The interpretation placed on phrases containing "shall" depends on the subsection in which they occur: "If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined." Within a constraint subsection these terms have the ISO meaning. This usage is a carry over from the original version of the standard which was created as a US National Standard.

Some ISO Standards were not originally written within a standard's body. For instance, while the current version of the Ada language specification is an ISO Standard it was originally written as a U.S. DOD document. The authors of this document invented their own terminology, e.g., *Legality Rules* subsections (see section 1.1.2 of the material quoted, in the annex to this document, from the Ada Standard) which specify rules that are enforced at compile time; it also defines other subheaders such as *Post-Compilation Rules* and *Bounded (Run-Time) errors*.[1]

3 Extracting information from a specification

Whichever form of specification is used a great deal of effort is likely to be needed to extract and understand all of the information applicable to the OWGV guidelines.

Extracting a list of requirements from an implementation requires that the behavior of that implementation be understood. This involves sorting out decisions made within the implementation (e.g., if statements) into those that relate to internal algorithmic details and those that relate to requirements driven by the language definition.

Extracting a list of requirements from a prose specification requires understanding the applicable terminology and how it is used to express requirements. The specification document can then be analysed to extract the necessary requirements.

3.1 When/Where does the behavior occur?

A number of different methods might be used to implement the requirements in a language specification. The most common is the sequence compile/link/execute, where each phase is completed before the next one starts and the order of the phases is fixed (i.e., no compilation occurs during program execution). A less commonly used method is one that translates and executes source code on an as needed basis (i.e., it is translated when encountered in the flow of control during program execution).

3.1.1 Source code translation

The specifications of Ada, C, C++, Fortran and Java require that all of the source code making up a unit of compilation (whether it is called a package, a translation unit, or some other name) be checked for conformance to the appropriate requirements prior to program execution.

Some of the advantages of being able to check requirements prior to program execution are that there is no dependency on a program's flow of control and it is much easier to resolve violations (i.e., there are likely to be more options available from modifying the source than selecting a precompiled option at runtime).

3.1.2 Linking

Most languages only contain a small number of link time requirements. The most common link time requirement is that every symbol that is used resolves to a definition of that symbol. For instance, C requires that if a function is called then a definition of that function must be available during the link process.

3.1.3 Program execution

There are some language constructs which only have undesirable behavior when presented with certain values. For instance, the division operator when the second operand is zero, or when the result of an arithmetic operation is larger than can be represented in the number of bits available (i.e., it overflows).

[1] Given that the first Military Ada Standard was published in 1980 the authors did not have the opportunity to make use of any existing ISO terminology (e.g., *shall*), which appeared later.

In those cases where it is very difficult to detect a behavior statically language specifications often chose to label the situation as being undefined and may not require that instances of it be detected and diagnosed.

3.2 Tool information needs

In many cases the most cost effective way of checking that source adheres to guideline recommendations is to use some form of automated tool. The implementation of such a tool requires detailed and accurate information of a language's semantics, which in turn requires a detailed and accurate specification.

The following are some of the kinds of information that may need to be extracted from a language specification:

- The constructs an implementation of a language required to support. It makes no sense to create a guideline that covers a construct that is not supported by a language. Also guideline authors might be interested in analysing all of the constructs supported by a language.
- The extent to which a construct may exhibit different external behaviors. Constructs whose external behavior are permitted to vary between implementations are a common source of guideline recommendations. There is thus a need to be able to readily identify such constructs.
- The constructs which are required to be diagnosed by a conforming implementation. There is unlikely to be any benefit in a guideline covering constructs whose particular use will be cause a conforming implementation to issue a diagnostic.
- The terminology (i.e., phrases and their meaning) used to frame requirements and behaviors. A guideline will be interpreted using the norms applied within the community of a given programming language. As such it needs to use terminology in a way that is consistent with usage within the community of its users.

4 Language derived requirements

One class of guideline recommendations is often derived purely from the language specification itself, rather than being directly related to common developer driven. Most languages contain constructs whose behavior can vary between implementations or even within a single implementation. A section of code that depends on permissible variability always behaving in a particular way contains a latent fault which will suddenly appear when the implementation's behavior changes.

OWGV have an interest in understanding the kinds of ways in which language specifications classify the various behaviors that are permitted to vary between implementations. When creating language specific guidelines there is also the practical issue of being able to reliably enumerate all of the constructs whose behavior may vary.

4.1 Variation by method of specification

4.1.1 Specification by implementation

There are a number of reasons why an implementation based specification might exhibit different external behaviors for the same language construct. The following are some of the common reasons:

- the implementation was not carefully written to take into account possible differences in characteristics between different hosting environments. In this case any differences in behavior are unintentional,
- the implementation was designed to take advantage of the varying functionality that might be available to it on different host environments. In this case any differences in behavior are intentional,
- behavior is the result of a fault in the 'defining' implementation.

The de facto specifications of Perl and PHP are written in C and have been ported to a variety of 32- and 64-bit cpus, Unix like operating systems and Windows, using a variety of compilers.

4.1.2 Specification by prose

There are a number of reasons why a prose language specification might allow different implementations to produce different external behavior for the same language construct. The following are some of the common reasons:

- The designers of a language have to decide the extent to which the characteristics of the platform on which a program might be executed can cause differences in external behavior. For instance, the size of an integer datatype might be required to always have a fixed width or might be allowed to depend on the characteristics of the host processor. In the latter case the external output from a program might vary between processors, e.g., a 64-bit cpu will support a wider range of possible values than a 16-bit cpu.

- Languages are not always fully and unambiguously defined when they are first specified. This can lead to different implementations having different interpretations of the behavior of source code. Existing implementations will have been used to create source code. While it might be relatively easy to change an existing implementation, it may be very costly to change all of the existing code that has been developed using existing implementations. A later revision of a language specification that attempts to created a unified definition may have to include *implementation defined* behaviors (i.e., behavior that can vary between implementations) as the political and economic cost of unification.

- Simplifying the specification wording. Exactly specifying the desired behavior of a language construct in every situation might require a great deal of interrelated requirements. The volume and complexity of the specification may make it difficult to verify that the intended behavior has been correctly specified in all cases, and a great deal of effort may have to be invested by subsequent readers of the wording before they understand it. By simplifying the desired behavior it may be possible to simplify its specification.

4.2 Changes to the specification

Many language specifications sanction the creation of extensions, by an implementation, provided they do not change the behavior of standards conforming code.

In some markets a particular vendor is dominant, e.g., The Microsoft Visual languages in the Microsoft Windows market. Vendors often add extensions to their language products and customers make use of these extensions in their programs. Unfortunately experience shows that many vendors do not specify the behavior of their extensions as thoroughly as the language itself specifies its behavior.

5 Summary of characteristics

The number of occurrences of various phrases was obtained by counting all occurrences of them in a published pdf copy of the respective standard. In standards that contain an annex containing a list of conformance requirements, this counting method will sometimes result in a particular application of a phrase being counted twice.

All language constructs exhibiting a particular behavior can sometimes be found by locating all instances of the appropriate phrase within the specification. For some constructs some language specifications do not explicitly call out the behavior. For instance, the C Standard does not always explicitly call out a behavior as being undefined, in some cases this kind of behavior has to be inferred from other wording containing in the standard.

Table .1: Summary of specification characteristics of some languages.

	Ada	C	C++	Fortran	Java	Perl
Defined using English prose	Yes	Yes	Yes	Yes	Yes	No
Defined by implementation	No	No	No	No	Yes	Yes
Behavior of source code specified	Yes	Yes	No	Yes	???	No
Behavior of implementation specified	No	No	Yes	No	???	Yes
Some behavior specified to vary across implementations	Yes	Yes	Yes	Yes	Yes	Unknown
Requires runtime detection of some erroneous behavior	Yes	No	No	No	Yes	Yes
Validation suite for complete language available	Yes	Yes	Yes	Yes	Yes	No

This summary does not attempt to match or align what might be considered to be closely equivalent subsection header names (e.g., subsections headed *Legality Rules* in Ada perform a function that closely corresponds to *Constraints* subsections in C).

5.1 Ada 2005

In the Ada 2005 Standard there are:

- 36 occurrences of *bounded error*.
- 53 occurrences of the subsection header *Erroneous execution* and 116 occurrences of *erroneous*.
- 343 occurrences of *implementation-defined*.
- 373 occurrences of *may*, some of which describe optional behavior.
- 22 occurrences of *must* some of which that read as-if *shall* was intended.
- 38 occurrences of *optional*.
- zero occurrences of *processor dependent* and *processor-dependent*.
- 1018 occurrences of *shall* of which 131 have the form *shall not*.
- 452 occurrences of *should*.
- 8 occurrences of *undefined*, one referencing to an undefined range, three having the form *undefined range* and the rest occurring in annexes (also see *bounded error*).
- 89 occurrences of *unspecified*.

The phrase *bounded error* is not used by the any of the other languages discussed in this paper. It roughly corresponds to *undefined behavior* in C (however, the set of possible behaviors in C is not required to be bounded).

5.2 C

In the C99 Standard, after applying changes from the first two Technical Corrigenda, there are:

- zero occurrences of *bounded error*.
- 3 occurrences of *erroneous*.
- 163 occurrences of *implementation-defined*.
- 862 occurrences of *may*.
- 1 occurrence of *must*.
- 34 occurrences of *optional*.
- zero occurrences of *processor dependent* and *processor-dependent*.
- 596 occurrences of *shall* of which 71 have the form *shall not*.
- 63 occurrences of *should*.
- 183 occurrences of *undefined*.
- 98 occurrences of *unspecified*.

5.3 C++

In the C++ 2003 Standard there are:

- zero occurrences of *bounded error*.
- 5 occurrences of *erroneous*.
- 236 occurrences of *implementation-defined*.
- 371 occurrences of *may*.
- 111 occurrences of *must*.
- 30 occurrences of *optional*.
- zero occurrences of *processor dependent* and *processor-dependent*.
- 779 occurrences of *shall* of which 211 have the form *shall not*.
- 38 occurrences of *should*.
- 195 occurrences of *undefined*.
- 113 occurrences of *unspecified*.

5.4 Fortran

In the Fortran 2004 Standard there are:

- zero occurrences of *bounded error*.
- 1 occurrence of *erroneously*.
- 1 occurrence of *implementation-defined*. It occurs in informative annex C. Subsection 4.4.4.3 contains a phrase that implies the same kind of behavior: “Each implementation defines a collating sequence for the character set of each kind of character.”
- 887 occurrences of *may*, some of which describe optional behavior.
- 8 occurrences of *must* that read as-if *shall* was intended.
- 227 occurrences of *optional* to specify that some construct or behavior is optional.
- 217 occurrences of *processor dependent* and *processor-dependent*.
- 2298 occurrences of *shall* of which 663 have the form *shall not*.
- 19 occurrences of *should* of which a few apply to language constructs rather than behavior outside the scope of the Standard.
- 159 instances of the word *undefined*. All except one of these instances refer to an undefined value of undefined pointer association. The single exception occurs in subsection 5.1.2.5.3 “The size, bounds, and shape of an unallocated allocatable array or a disassociated array pointer are undefined.”
- 14 instances of the word *unspecified*. Of these, 12 occur in the phrase “unspecified storage unit”. The other two instances occur in subsection 9.4.5 “Execution of such an OPEN statement causes any new values of the specifiers for changeable modes to be in effect, but does not cause any change in any of the unspecified specifiers and the position of the file is unaffeected.” and subsection 9.4.5.11 “ASIS leaves the position unspecified if the file exists but is not connected.”

The phrases *processor dependent* and *processor-dependent* are not used by the any of the other languages discussed in this paper. It roughly corresponds to *implementation defined behavior* in C.

5.5 Java

In the third edition of the Java Language Specification there are:

- zero occurrences of *bounded error*.
- 421 occurrences of *compile-time error*.
- 7 occurrences of *erroneous*.
- 2 occurrences of *implementation-dependent* or *implementation dependent*. There are 13 occurrences of *the/an implementation may*. The one occurrences of *implementation-defined* refers to a specification in the C Standard.
- 3 occurrences of *link-time error* or *linkage time error*.
- 597 occurrences of *may*, of which 59 have the form *may not*.
- 420 occurrence of *must*, of which 32 have the form *must not*.
- 19 occurrences of *optional*.
- zero occurrences of *processor dependent* and *processor-dependent*.
- 2 occurrences of *run-time error*.
- 6 occurrences of *shall* none of which have the form *shall not*.
- 107 occurrences of *should*.
- 3 occurrences of *undefined*.
- 2 occurrences of *unspecified* (one of which refers to behavior specified in the C/C++ Standard).

6 Some requirements wording

The following subsections contain requirements related wording that has been copied verbatim from the respective language standard.

6.1 Ada

1.1.2 Structure This International Standard contains thirteen sections, fourteen annexes, and an index.

The core of the Ada language consists of:

```
Sections 1 through 13
Annex A, "Predefined Language Environment"
Annex B, "Interface to Other Languages"
Annex J, "Obsolescent Features"
```

The following Specialized Needs Annexes define features that are needed by certain application areas:

```
Annex C, "Systems Programming"
Annex D, "Real-Time Systems"
Annex E, "Distributed Systems"
Annex F, "Information Systems"
Annex G, "Numerics"
Annex H, "High Integrity Systems"
```

The core language and the Specialized Needs Annexes are normative, except that the material in each of the items listed below is informative:

Text under a NOTES or Examples heading.

Each clause or subclause whose title starts with the word "Example" or "Examples".

All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Specialized Needs Annexes.

The following Annexes are informative:

```
       Annex K, "Language-Defined Attributes"
       Annex L, "Language-Defined Pragmas"
       M.2, "Implementation-Defined Characteristics"
       Annex N, "Glossary"
       Annex P, "Syntax Summary"
```

Each section is divided into clauses and subclauses that have a common structure. Each section, clause, and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

Syntax

Syntax rules (indented).

Name Resolution Rules

Compile-time rules that are used in name resolution, including overload resolution.

Legality Rules

Rules that are enforced at compile time. A construct is legal if it obeys all of the Legality Rules.

Static Semantics

A definition of the compile-time effect of each construct.

Post-Compilation Rules

Rules that are enforced before running a partition. A partition is legal if its compilation units are legal and it obeys all of the Post-Compilation Rules.

Dynamic Semantics

A definition of the run-time effect of each construct.

Bounded (Run-Time) Errors

Situations that result in bounded (run-time) errors (see 1.1.5).

Erroneous Execution

Situations that result in erroneous execution (see 1.1.5).

Implementation Requirements

Additional requirements for conforming implementations.

Documentation Requirements

Documentation requirements for conforming implementations.

Metrics

Metrics that are specified for the time/space properties of the execution of certain language constructs.

Implementation Permissions

Additional permissions given to the implementer.

Implementation Advice

Optional advice given to the implementer. The word "should" is used to indicate that the advice is a recommendation, not a requirement. It is implementation defined whether or not a given recommendation is obeyed.

NOTES

1 Notes emphasize consequences of the rules described in the (sub)clause or elsewhere. This material is informative.

Examples

Examples illustrate the possible forms of the constructs described. This material is informative.

1.1.3 Conformity of an Implementation with the Standard

Implementation Requirements

A conforming implementation shall:

- *Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation;*
- *Identify all programs or program units that are so large as to exceed the capacity of the implementation (or raise an appropriate exception at run time);*
- *Identify all programs or program units that contain errors whose detection is required by this International Standard;*
- *Supply all language-defined library units required by this International Standard;*
- *Contain no variations except those explicitly permitted by this International Standard, or those that are impossible or impractical to avoid given the implementation's execution environment;*
- *Specify all such variations in the manner prescribed by this International Standard.*

The external effect of the execution of an Ada program is defined in terms of its interactions with its external environment. The following are defined as external interactions:

- *Any interaction with an external file (see A.7);*
- *The execution of certain code_statements (see 13.8); which code_statements cause external interactions is implementation defined.*
- *Any call on an imported subprogram (see Annex B), including any parameters passed to it;*
- *Any result returned or exception propagated from a main subprogram (see 10.2) or an exported subprogram (see Annex B) to an external caller;*
- *Any read or update of an atomic or volatile object (see C.6);*
- *The values of imported and exported objects (see Annex B) at the time of any other interaction with the external environment.*

A conforming implementation of this International Standard shall produce for the execution of a given Ada program a set of interactions with the external environment whose order and timing are consistent with the definitions and requirements of this International Standard for the semantics of the given program.

An implementation that conforms to this Standard shall support each capability required by the core language as specified. In addition, an implementation that conforms to this Standard may conform to one or more Specialized Needs Annexes (or to none). Conformance to a Specialized Needs Annex means that each capability required by the Annex is provided as specified.

An implementation conforming to this International Standard may provide additional attributes, library units, and pragmas. However, it shall not provide any attribute, library unit, or pragma having the same name as an attribute, library unit, or pragma (respectively) specified in a Specialized Needs Annex unless the provided construct is either as specified in the Specialized Needs Annex or is more limited in capability than that required by the Annex. A program that attempts to use an unsupported capability of an Annex shall either be identified by the implementation before run time or shall raise an exception at run time.

Documentation Requirements

Certain aspects of the semantics are defined to be either implementation defined or unspecified. In such cases, the set of possible effects is specified, and the implementation may choose any effect in the set. Implementations shall document their behavior in implementation-defined situations, but documentation is not required for unspecified situations. The implementation-defined characteristics are summarized in M.2.

The implementation may choose to document implementation-defined behavior either by documenting what happens in general, or by providing some mechanism for the user to determine what happens in a particular case.

Implementation Advice

If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise Program_Error if feasible.

If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit.

NOTES

2 The above requirements imply that an implementation conforming to this Standard may support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities.

1.1.4 Method of Description and Syntax Notation

The form of an Ada program is described by means of a context-free syntax together with context dependent requirements expressed by narrative rules.

The meaning of Ada programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs.

The context-free syntax of the language is described using a simple variant of Backus-Naur Form. In particular:

Lower case words in a sans-serif font, some containing embedded underlines, are used to denote syntactic categories, for example:

case_statement

Boldface words are used to denote reserved words, for example:

array

Square brackets enclose optional items. Thus the two following rules are equivalent.

```
simple_return_statement ::= return [expression];
simple_return_statement ::= return; | return expression;
```

Curly brackets enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.

```
term ::= factor {multiplying_operator factor}
term ::= factor | term multiplying_operator factor
```

A vertical line separates alternative items unless it occurs immediately after an opening curly bracket, in which case it stands for itself:

```
constraint ::= scalar_constraint | composite_constraint
discrete_choice_list ::= discrete_choice {| discrete_choice}
```

If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example subtype_name and task_name are both equivalent to name alone.

The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code position is between 16#20# and 16#7E#, inclusively. The special characters for which names are *defined in this International Standard (see 2.1) belong to the same range. For example, the character E in the definition of exponent is the character whose name is "LATIN CAPITAL LETTER E", not "GREEK CAPITAL LETTER EPSILON".*

When this International Standard mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using locale-independent full case folding, as defined by documents referenced in the note in section 1 of ISO/IEC 10646:2003.

A syntactic category is a nonterminal in the grammar defined in BNF under "Syntax." Names of syntactic categories are set in a different font, like_this.

A construct is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax".

A constituent of a construct is the construct itself, or any construct appearing within it.

Whenever the run-time semantics defines certain actions to happen in an arbitrary order, this means that the implementation shall arrange for these actions to occur in a way that is equivalent to some sequential order, following the rules that result from that sequential order. When evaluations are defined to happen in an arbitrary order, with conversion of the results to some subtypes, or with some run-time checks, the evaluations, conversions, and checks may be arbitrarily interspersed, so long as each expression is evaluated before converting or checking its value. Note that the effect of a program can depend on the order chosen by the implementation. This can happen, for example, if two actual parameters of a given call have side effects.

NOTES

3 The syntax rules describing structured constructs are presented in a form that corresponds to the recommended paragraphing. For example, an if_statement is defined as:

```
if_statement ::=
    if condition then
       sequence_of_statements
    {elsif condition then
       sequence_of_statements}
    [else
       sequence_of_statements]
    end if;
```

4 The line breaks and indentation in the syntax rules indicate the recommended line breaks and indentation in the corresponding constructs. The preferred places for other line breaks are after semicolons.

1.1.5 Classification of Errors

Implementation Requirements

The language definition classifies errors into several different categories:

- *Errors that are required to be detected prior to run time by every Ada implementation; These errors correspond to any violation of a rule given in this International Standard, other than those listed below. In particular, violation of any rule that uses the terms shall, allowed, permitted, legal, or illegal belongs to this category. Any program that contains such an error is not a legal Ada program; on the other hand, the fact that a program is legal does not mean, per se, that the program is free from other forms of error.*

  The rules are further classified as either compile time rules, or post compilation rules, depending on whether a violation has to be detected at the time a compilation unit is submitted to the compiler, or may be postponed until the time a compilation unit is incorporated into a partition of a program.
- *Errors that are required to be detected at run time by the execution of an Ada program; The corresponding error situations are associated with the names of the predefined exceptions. Every Ada compiler is required to generate code that raises the corresponding exception if such an error situation arises during program execution. If such an error situation is certain to arise in every execution of a construct, then an implementation is allowed (although not required) to report this fact at compilation time.*

- *Bounded errors; The language rules define certain kinds of errors that need not be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. The errors of this category are called bounded errors. The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception Program_Error.*
- *Erroneous execution. In addition to bounded errors, the language rules define certain kinds of errors as leading to erroneous execution. Like bounded errors, the implementation need not detect such errors either prior to or during run time. Unlike bounded errors, there is no language-specified bound on the possible effect of erroneous execution; the effect is in general not predictable.*

Implementation Permissions

An implementation may provide nonstandard modes of operation. Typically these modes would be selected by a pragma or by a command line switch when the compiler is invoked. When operating in a nonstandard mode, the implementation may reject compilation_units that do not conform to additional requirements associated with the mode, such as an excessive number of warnings or violation of coding style guidelines. Similarly, in a nonstandard mode, the implementation may apply special optimizations or alternative algorithms that are only meaningful for programs that satisfy certain criteria specified by the implementation. In any case, an implementation shall support a standard mode that conforms to the requirements of this International Standard; in particular, in the standard mode, all legal compilation_units shall be accepted.

Implementation Advice

If an implementation detects a bounded error or erroneous execution, it should raise Program_Error.

6.2 C

3.4 behavior

external appearance or action

3.4.1 implementation-defined behavior

unspecified behavior where each implementation documents how the choice is made

EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

3.4.2 locale-specific behavior

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

EXAMPLE An example of locale-specific behavior is whether the islower function returns true for characters other than the 26 lowercase Latin letters.

3.4.3 undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

EXAMPLE An example of undefined behavior is the behavior on integer overflow.

3.4.4 unspecified behavior

behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated.

...

3.8 constraint

restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

...

3.10 diagnostic message

message belonging to an implementation-defined subset of the implementation's message output

...

3.12 implementation

particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

3.13 implementation limit

restriction imposed upon programs by the implementation

...

3.16 recommended practice

specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations

...

3.17.1 implementation-defined value

unspecified value where each implementation documents how the choice is made

3.17.2 indeterminate value

either an unspecified value or a trap representation

3.17.3 unspecified value

valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance

NOTE An unspecified value cannot be a trap representation.

...

4. Conformance

In this International Standard, "shall" is to be interpreted as a requirement on an implementation or on a program; conversely, "shall not" is to be interpreted as a prohibition.

If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words "undefined behavior" or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe "behavior that is undefined".

A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.

The implementation shall not successfully translate a preprocessing translation unit containing a #error preprocessing directive unless it is part of a group skipped by conditional inclusion.

A strictly conforming program shall use only those features of the language and library specified in this International Standard.2) It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.

The two forms of conforming implementation are hosted and freestanding. A conforming hosted implementation shall accept any strictly conforming program. A conforming freestanding implementation shall accept any *strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers <float.h>, <iso646.h>, <limits.h>, <stdarg.h>, <stdbool.h>, <stddef.h>, and <stdint.h>. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.3)*

A conforming program is one that is acceptable to a conforming implementation.4)

An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

4) Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs may depend upon nonportable features of a conforming implementation.

6.3 C++

1.3.2 diagnostic message

a message belonging to an implementation-defined subset of the implementation's output messages.

1.3.4 ill-formed program

input to a C++ implementation that is not a well-formed program (1.3.14).

1.3.5 implementation-defined behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation shall document.

1.3.6 implementation limits

restrictions imposed upon programs by the implementation.

1.3.7 locale-specific behavior

behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

1.3.12 undefined behavior

behavior, such as might arise upon use of an erroneous program construct or erroneous data, for which this International Standard imposes no requirements. Undefined behavior may also be expected when this International Standard omits the description of any explicit definition of behavior. [Note: permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed.]

1.3.13 unspecified behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation. The implementation is not required to document which behavior occurs. [Note: usually, the range of possible behaviors is delineated by this International Standard.]

1.3.14 well-formed program

a C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).

1.4 Implementation compliance

The set of diagnosable rules consists of all syntactic and semantic rules in this International Standard except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "undefined behavior."

Although this International Standard states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:

-- If a program contains no violations of the rules in this International Standard, a conforming implementation shall, within its resource limits, accept and correctly execute3) that program.

-- If a program contains a violation of any diagnosable rule, a conforming implementation shall issue at least one diagnostic message, except that

-- If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.

For classes and class templates, the library clauses specify partial definitions. Private members (clause 11) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library clauses.

For functions, function templates, objects, and values, the library clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library clauses. The names defined in the library have namespace scope (7.3). A C++ translation unit (2.1) obtains access to these names by including the appropriate standard library header (16.2).

The templates, classes, functions, and objects in the library have external linkage (3.5). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (2.1).

Two kinds of implementations are defined: hosted and freestanding. For a hosted implementation, this International Standard defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (17.4.1.3).

A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this International Standard. Having done so, however, they can compile and execute such programs.

3) "Correct execution" can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.

6.4 Fortran

1.5 Conformance

A program (2.2.1) is a standard-conforming program if it uses only those forms and relationships described herein and if the program has an interpretation according to this standard. A program unit (2.2) conforms to this standard if it can be included in a program in a manner that allows the program to be standard conforming.

A processor conforms to this standard if

(1) It executes any standard-conforming program in a manner that fulfills the interpretations herein, subject to any limits that the processor may impose on the size and complexity of the program;

(2) It contains the capability to detect and report the use within a submitted program unit of a form designated herein as obsolescent, insofar as such use can be detected by reference to the numbered syntax rules and constraints;

(3) It contains the capability to detect and report the use within a submitted program unit of an additional form or relationship that is not permitted by the numbered syntax rules or constraints, including the deleted features described in Annex B;

(4) It contains the capability to detect and report the use within a submitted program unit of an intrinsic type with a kind type parameter value not supported by the processor (4.4);

(5) It contains the capability to detect and report the use within a submitted program unit of source form or characters not permitted by Section 3;

(6) It contains the capability to detect and report the use within a submitted program of name usage not consistent with the scope rules for names, labels, operators, and assignment symbols in Section 16;

(7) It contains the capability to detect and report the use within a submitted program unit of intrinsic procedures whose names are not defined in Section 13; and

(8) It contains the capability to detect and report the reason for rejecting a submitted program.

However, in a format specification that is not part of a FORMAT statement (10.1.1), a processor need not detect or report the use of deleted or obsolescent features, or the use of additional forms or relationships.

A standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. However, a standard-conforming processor may allow additional intrinsic procedures even though this could cause a conflict with the name of a procedure in a standard-conforming program. If such a conflict occurs and involves the name of an external procedure, the processor is permitted to use the intrinsic procedure unless the name is given the EXTERNAL attribute (5.1.2.6) in the scoping unit (16). A standard-conforming program shall not use nonstandard intrinsic procedures or modules that have been added by the processor.

Because a standard-conforming program may place demands on a processor that are not within the scope of this standard or may include standard items that are not portable, such as external procedures defined by means other than Fortran, conformance to this standard does not ensure that a program will execute consistently on all or any standard-conforming processors.

In some cases, this standard allows the provision of facilities that are not completely specified in the standard. These facilities are identified as processor dependent. They shall be provided, with methods or semantics determined by the processor.

NOTE 1.1

The processor should be accompanied by documentation that specifies the limits it imposes on the size and complexity of a program and the means of reporting when these limits are exceeded, that defines the additional forms and relationships it allows, and that defines the means of reporting the use of additional forms and relationships and the use of deleted or obsolescent forms. In this context, the use of a deleted form is the use of an additional form.

The processor should be accompanied by documentation that specifies the methods or semantics of processor-dependent facilities.

1.7 Notation used in this standard

In this standard, "shall" is to be interpreted as a requirement; conversely, "shall not" is to be interpreted as a prohibition. Except where stated otherwise, such requirements and prohibitions apply to programs rather than processors.

1.7.1 Informative notes

Informative notes of explanation, rationale, examples, and other material are interspersed with the normative body of this publication. The informative material is nonnormative; it is identified by being in shaded, framed boxes that have numbered headings beginning with "NOTE."

1.7.2 Syntax rules

Syntax rules describe the forms that Fortran lexical tokens, statements, and constructs may take. These syntax rules are expressed in a variation of Backus-Naur form (BNF) in which:

(1) Characters from the Fortran character set (3.1) are interpreted literally as shown, except where otherwise noted.

...

1.7.3 Constraints

Each constraint is given a unique identifying number of the form Csnn, where s is a one- or two-digit section number and nn is a two-digit sequence number within that section.

Often a constraint is associated with a particular syntax rule. Where that is the case, the constraint is annotated with the syntax rule number in parentheses. A constraint that is associated with a syntax rule constitutes part of the definition of the syntax term defined by the rule. It thus applies in all places where the syntax term appears.

Some constraints are not associated with particular syntax rules. The effect of such a constraint is similar to that of a restriction stated in the text, except that a processor is required to have the capability to detect and report violations of constraints (1.5). In some cases, a broad requirement is stated in text and a subset of the same requirement is also stated as a constraint. This indicates that a standard-conforming program is required to adhere to the broad requirement, but that a standard-conforming processor is required only to have the capability of diagnosing violations of the constraint.

1.7.4 Assumed syntax rules

In order to minimize the number of additional syntax rules and convey appropriate constraint information, the following rules are assumed; an explicit syntax rule for a term overrides an assumed rule. The letters "xyz" stand for any syntactic class phrase:

```
R101  xyz-list             is   xyz [ , xyz ] ...
R102  xyz-name             is   name
R103  scalar-xyz           is   xyz
C101     (R103) scalar-xyz shall be scalar.
```

References

1. S. Bradner. *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*. IETF, 1997.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, 1996.
3. ISO. *Implementation of ISO/IEC TR 10034:1990 Guidelines for the preparation of conformity clauses in programming language standards*. ISO, 1990.
4. ISO. *Implementation of ISO/IEC TR 10176:1997 Information technology —Guidelines for the preparation of programming language standards*. ISO, 1997.
5. ISO. *ISO/IEC Directives, Part 2 Rules for the structure and drafting of International Standards*. ISO, fifth edition, 2004.
6. D. M. Jones. Who guards the guardians? `www.knosof.co.uk/whoguard.html`, 1992.
7. D. M. Jones. The new C Standard: An economic and cultural commentary. Knowledge Software, Ltd, 2005.
8. L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc, 3rd edition, 2000.
9. J. Welsh and A. Hay. *A Model Implementation of Standard Pascal*. Prentice-Hall, Inc, 1986.