TECHNICAL REPORT

ISO/IEC TR 10176

First edition
1991-04-15

Information technology — Guidelines for the preparation of programming language standards

Technologies de l'information — Lignes directrices pour la préparation des normes des langages de programmation

Reference number
ISO/IEC TR 10176 : 1991 (E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The main task of technical committees is to prepare International Standards. In exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/IEC TR 10176, which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology.*

Introduction

Background: Over the last 20 years and more (1966-1989), standards have been produced for a number of computer programming languages. Each has dealt with its own language in isolation, although to some extent the drafting committees have become more expert by learning from both the successes and the mistakes of their predecessors.

The time is now right to put together some of the experience that has been gained, in a set of guidelines, designed to ease the task of drafting committees of programming language standards.

This document is published as a Technical Report type 3 because the design of programming languages - and hence requirements relating to their standardization - is still evolving fairly rapidly, and because existing languages, both standardized and unstandardized, vary so greatly in their properties and styles that publication as a full standard, even as a standard set of guidelines, did not seem appropriate at this time.

The need for guidelines: While each language, taken as a whole, is unique, there are many individual features that are common to many, or even to most of them. While standardization should not inhibit such diversity as is essential, both in the languages and in the form of their standards, unnecessary diversity is better avoided. Unnecessary diversity leads to unnecessary confusion, unnecessary retraining, unnecessary conversion or redevelopment, and unnecessary costs. The aim of the guidelines is therefore to help to achieve standardization across languages and across their standards.

The existence of a guideline will often save a drafting committee from much discussion of detailed points all of which have been discussed previously for other languages.

Furthermore the avoidance of needless diversity between languages makes it easier for programmers to switch between one and another.

> NOTE - Diversity is a major problem because it uses up time and resources better devoted to the essential part, both by makers and users of standards. Building a language standard is very expensive in resources and far too much time and effort goes into "reinventing the wheel" and trying to solve again, from the beginning, the same problems that other committees have faced.
>
> However, a software writer faced with the task of building (say) a support environment (operating system facilities, utilities, etc.) for a number of different language processors is also faced with many problems from the eventual standards. Quite apart from the *essential* differences between the languages, there are to begin with the variations of layout, arrangement, terminology, metalanguages, etc. Much worse, there are the variations between requirements of basically the same kind, some substantial, some slight, some subtle - compounded by needless variations in the way they are specified. This represents an immense extra burden - as does the duplication in providing different support tools for different languages performing basically the same task.

How to use this Technical Report: This Technical Report does not seek to legislate on how programming languages should be designed or standardized: it would be futile even to attempt that. The guidelines are, as their name implies, intended for guidance only. Nevertheless, drafting committees are strongly urged to examine them seriously, to consider each one with care, and to adopt its recommendation where practicable. The guidelines have been so written that it will be possible in most cases to determine, by examination, whether a given programming language standard has been produced in accordance with a given guideline, or otherwise. However, the conclusions to be drawn from such an assessment, and consequent action to be taken, are matters for individual users of this Technical Report and are beyond its scope.

Reasons for not adopting any particular guideline should be recorded and made available (e.g. in the minutes of the meeting at which use of the guideline was considered but rejected) so that they may be taken into account when a new Technical Report on this subject is deemed necessary.

Of course, care must naturally be taken when following these guidelines to do so in a way which does not conflict with the ISO/IEC Directives, or other rules of the standards body under whose direction the standard is being prepared.

Further related guidelines: This Technical Report is concerned with the generality of programming languages and general issues concerning questions of standardization of programming languages, and is not claimed to be necessarily universally applicable to all languages in all circumstances. Particular languages or kinds of languages, or particular areas of concern, may need more detailed and more specific guidelines than would be appropriate for this Technical Report. At the time of publication, some specific areas are already the subject of more detailed guidelines, to be found in existing or forthcoming Technical Reports. Such Technical Reports may extend, interpret, or adapt the guidelines in this Technical Report to cover specific issues and areas of application. Users of this Technical Report are recommended to take such other guidelines into account, as well as those in this Technical Report, where the circumstances are appropriate. See, in particular, ISO/TR 9547 and ISO/IEC TR 10034.

Information technology — Guidelines for the preparation of programming language standards

1 Scope

This Technical Report presents a set of guidelines for producing a standard for a programming language.

2 References

ISO 646:1983 *Information processing - ISO 7-bit coded character set for information interchange.*

ISO 2382-15:1985, *Data processing systems - Vocabulary - Part 15: Programming languages.*

ISO 4873:1986, *Information processing - ISO 8-bit code for information interchange - Structure and rules for implementation.*

ISO 6937-2:1983, *Information processing - Coded character sets for text communication - Part 2: Latin alphabetic and non-alphabetic graphic characters.*

ISO 8859:1987, *Information processing - 8-bit single byte coded graphic character sets.*

ISO/TR 9547:1988, *Programming language processors - Test methods - Guidelines for their development and acceptability.*

ISO/IEC TR 10034:1990, *Guidelines for the preparation of conformity clauses in programming language standards.*

BS 6154:1981, *Method of defining syntactic metalanguage.*

3 Definitions

This clause contains terminology which is used in particular specialized senses in this Technical Report. It is not claimed that all language standards necessarily use the terminology in the senses defined here; where appropriate, the necessary interpretations and conversions would need to be carried out when applying these guidelines in a particular case. Also, not all language standards use the terminology of ISO 2382-15; the terminology defined here, itself divergent in some cases from that in ISO 2382-15, has been introduced to minimize confusion which might result from such differences. Some remarks are made below about particular divergences from ISO 2382-15, for further clarification.

3.1 programming language processor (abbreviated where there is no ambiguity to **processor**): Denotes the entire computing system which enables the programming language user to translate and execute programs written in the language, in general consisting both of hardware and of the relevant associated software.

NOTES

1 A "processor" in the sense of this Technical Report therefore consists of more than simply (say) a "compiler" or an "implementation" in conventional terminology; in general it consists of a package of facilities, of which a "compiler" in the conventional sense may be only one. There is also no implication that the processor consists of a monolithic entity, however constituted. For example, processor software may consist of a syntax checker, a code generator, a link-loader, and a run-time support package, each of which exists as a logically distinct entity. The "processor" in this case would be the assemblage of all of these and the associated hardware. Conformity to the standard would apply to the assemblage as a whole, not to individual parts of it.

2 In ISO/TR 9547 the term "processor" is used in a more restricted sense. For the purposes of ISO/TR 9547, a differentiation is necessary between "processor" and "configuration"; that distinction is not necessary in this Technical Report. Those using both Technical Reports will need to bear this difference in terminology in mind. See 3.3.4 for another instance of a difference in terminology, where a distinction which is not necessary in ISO/TR 9547 has to be made in this Technical Report.

3.2 syntax and semantics: Denote the grammatical rules of the language. The term **syntax** refers to the rules that determine whether a program text is well-formed. The syntactic rules need not be exclusively "context-free", but must allow a processor to decide, solely by inspection of a program text, with a practicable amount of effort and within a practicable amount of time, whether that text conforms to the rules. An **error** (see 3.3.1) is a violation of the syntactic rules.

The term **semantics** refers to the rules which determine the behaviour of processors when executing well-formed programs. An **exception** (see 3.3.2) is a violation of a non-syntactic requirement on programs.

NOTE - In ISO 2382-15 the term **static** is defined (15.02.09) as "pertaining to properties that can be established before the execution of a program" and **dynamic** (15.02.10) as "pertaining to properties that can only be established during the execution of a program". These therefore appear to be close to the terms "syntax" and "semantics" respectively as defined in this Technical Report. ISO 2382-15 does not define "syntax" or "semantics", though these are terms very commonly used in the programming language community.

Furthermore, the uses of "static" and "dynamic" (and other terms) in ISO 2382-15 seem designed for use within a single language rather than across all languages, but while that terminology can mostly be applied consistently within a single language, it becomes much harder to do so across the generality of languages, which is the need in this Technical Report. This problem is not totally absent with "syntax/ semantics" but is much less acute.

3.3 Errors, exceptions, and conditions

3.3.1 errors: The incorrect program constructs which are statically determinable solely from inspection of the program text, without execution, and from knowledge of the language syntax. A **fatal error** is one from which recovery is not possible, i.e. it is not possible to proceed to (or continue with) program execution. A **non-fatal error** is one from which such recovery is possible.

NOTE - A fatal error may not necessarily preclude the processor from continuing to process the program, in ways which do not involve program execution (for example, further static analysis of the program text).

3.3.2 exceptions: The instances of incorrect program functioning which in general are determinable only dynamically, through execution of the program. A **fatal exception** is one from which recovery is not possible, i.e. it is not possible to continue with (or to proceed to) program execution. A **non-fatal exception** is one from which recovery is possible.

NOTES

1 In case of doubt, "possible" within this section should be interpreted as "possible without violating definitions within or requirements of the standard". For example, the hardware element of a language processor may have the technical capability of continuing program execution after division by zero, but in terms of a language standard which defines division by zero as a fatal exception, the consequences of such continued execution would not be meaningful.

2 See also 3.3.4

3.3.3 conditions: Occurrences during execution of the program which cause an interruption of normal processing when detected. A condition may be an exception, or may be some language-defined or user-defined occurrence, depending on the language.

NOTE - For example, reaching end-of-file on input may always be an exception in one language, may always be a condition in another, while in a third it may be a condition if action to be taken on detection is specified in the program, but an exception if its occurrence is not anticipated.

3.3.4 Relationship to other terminology

In ISO/TR 9547 the term "error" is used in a more general sense to encompass what this Technical Report terms "exceptions" as well as "errors". For the purposes of ISO/TR 9547, the differentiation made here is not necessary. Those using both Technical Reports will need to bear this difference in terminology in mind. See note 2 of 3.1 for another instance of a difference in terminology, where a distinction has to be made in ISO/TR 9547 which is not necessary in this Technical Report.

ISO 2382-15 does not define "error" but does define "exception (in a programming language)" (15.06.12). The definition reads "A special situation which may arise during execution, which is considered abnormal, which may cause a deviation from the normal execution sequence, and for which facilities exist in the programming language to define, raise, recognize, ignore and handle it". ON-conditions in PL/I and exceptions in Ada are cited as examples.

The reason for not using this terminology in this Technical Report, which deals with the generality of existing and potential standardized languages rather than just a single one, is that it makes it difficult to distinguish (as this Technical Report needs to do) between "pure" exceptions, more general conditions, and processor options for exception handling which are built into the language (all in the senses defined in this Technical Report). It also does not aid making sufficient distinction between ON-conditions being enabled or disabled (globally or locally), nor whether the condition handler is the system default or provided by the programmer.

3.4 Processor dependence

For the purposes of this Technical Report, the following definitions are assumed.

If this Technical Report refers to a feature being left **undefined** in a standard (though referred to within the standard), this means that no requirement is specified concerning its provision and the effect of attempting to use the feature cannot be predicted.

If this Technical Report refers to a feature being **processor-dependent**, this means that the standard requires the processor to supply the feature but that there are no further requirements upon how it is provided.

If this Technical Report refers to a feature being **processor-defined**, this means that its definition is left processor-dependent by the standard, but that the definition shall be explicitly specified and made available to the user in some appropriate form (such as part of the documentation accompanying the processor, or through use of an environmental enquiry function).

NOTES

1 The term "feature" is used here to encompass both language features (syntactic elements a change to which would change the text of a program) and processor features (e.g. processor options, or accompanying documentation, a change to which would not change the text of a program). Examples of features which are commonly left undefined, processor-dependent or processor-defined are the collating sequence of the supported character set (a language feature) and processor action on detection of an exception (a processor feature).

2 In any particular instance the precise effect of the use of any of these terms may be affected by the nature of the feature concerned and the context in which the term is used.

3 None of the above terms specifically covers the case where reference to a feature is omitted altogether from the standard. While in general this might be regarded as "implicit undefined", it is possible that an unmentioned feature might necessarily have to be supplied for the processor to be usable (and would hence be processor-dependent) and that some aspects of the feature might in turn have to be processor-defined for the feature to be usable.

3.5 Secondary, Incremental and supplementary standards

3.5.1 Secondary standards

In this Technical Report, a secondary standard is one which requires strict conformity with another ("primary") standard - or possibly more than one primary standard - but places further requirements on conforming products (e.g. in the context of this Technical Report, on language processors or programs).

NOTE - A possible secondary standard for conforming programs might specify additional requirements with respect to use of comments and indentation, provision of documentation, use of conventions for naming user-defined identifiers, etc.

A possible secondary standard for conforming processors might specify additional requirements with respect to error and exception handling, range and accuracy of arithmetic, complexity of programs which can be processed, etc.

3.5.2 Incremental standards

In this Technical Report, an incremental standard adds to an existing standard without modifying its content. Its purpose is to supplement the coverage of the existing standard within its scope (e.g. language definition) rather than (as with a secondary standard, see 3.5.1) to add further requirements upon products conforming with an existing standard which are outside that scope. It is recognized that in some cases it might be desirable to produce a standard additional to an existing one which was both "incremental" (in terms of language functionality) and "secondary" (in terms of other requirements upon products).

3.5.3 Supplementary standards

In this Technical Report, a supplementary standard adds functionality to an existing standard without extending its range of syntactic constructs; such as by the binding of a language to a specific set of functions. Supplementary standards are expected to be expressed in terms of the base language which they supplement, but do not replace any elements of the primary standard.

4 Guidelines

4.1 Guidelines for the form and content of standards

4.1.1 Guideline: The general framework

The standard should be designed so that it consists of at least the following elements:

1) The specification of the syntax of the language, including rules for conformity of programs and processors.

2) The specification of the semantics of the language, including rules for conformity of programs and processors.

3) The specification of all further requirements on standard-conforming programs, and of rules for conformity.

4) The specification of all further requirements on standard-conforming processors (such as error and exception detection, reporting and processing; provision of processor options to the user; documentation; validation; etc.), and of rules for conformity.

5) One or more annexes containing an informal description of the language, a description of the metalanguage used in 1) and any formal method used in 2), a summary of the metalanguage definitions, a glossary, guidelines for programmers (on processor-dependent features, documentation available, desirable documentation of programs, etc.), and a cross-referenced index to the document.

6) An annex containing a checklist of any implementation-defined features.

7) An annex containing guidelines for implementors, including short examples.

8) An annex providing guidance to users of the standard on questions relating to the validation of conformity, with particular reference to ISO/IEC TR 10034, and any specific requirements relating to validation contained in 1) to 4) above.

9) In the case where a language standard is a revision of an earlier standard, an annex containing a detailed and precise description of the areas of incompatibility between the old and the new standard.

10) An annex which forms a tutorial commentary containing complete example programs that illustrate the use of the language.

NOTES

1 The objective of this guideline is to provide a framework for use by drafting committees when producing standards documents. This framework ensures that users of the standard, whether programmers, implementors or testers, will find in the standards document the things that they are looking for; in addition, it provides drafting committees with a basis for organizing their work.

2 The elements referred to above are concerned only with the technical content of the standard, and are to be regarded as logical elements of that content rather than necessarily physical elements (see note 4 below).

3 It is to be made clear that the annexes referred to in elements 5) to 10) above are informative annexes (i.e. descriptive or explanatory only), and not normative, i.e. do not qualify or amend the specific requirements of the standard given in elements 1), 2), 3) and 4). It should be explicitly stated that, in any case of ambiguity or conflict, it is the standard as specified in elements 1), 2), 3) and 4) that is definitive. Note that, if a definition (as opposed to description) of any formal method used in elements 1) and 2) cannot be established by reference, then the standard may need to incorporate that definition, insofar as is allowed by the rules of the responsible standards body (see also 4.1.2).

4 Given the requirements of note 3 above, a drafting committee has the right to interleave the various elements of the standard it is producing if it feels that this has advantages of clarity and readability, provided that precision is not compromised thereby, and that the distinction between the normative (specification) elements and the informative (informal descriptive) elements is everywhere made clear.

5 Element 9) will be empty if the standard is not a revision of an earlier standard. No specific guidelines or recommendations are included in this Technical Report concerning requirements on programs other than conformity with the syntactic and semantic rules of the language, and if this is the case in a standard, element 3) will be empty; however, it is recommended that in such a case an explicit statement be included that the only rules for conformity of programs are those for conformity with the language definition. It is recommended that none of the other elements should be left empty.

4.1.2 Guideline: Definitions of syntax and semantics

Consideration should be given to the use of a syntactic metalanguage for the formal definition of the syntax of the language, and the current "state of the art" in formal definition of semantics should be investigated, to determine whether the use of a formal method in the standard is feasible; the current policies on the use of formal methods within the standards body responsible for the standard should also be taken into account.

NOTES

1 Traditionally some language standards have not used a full metalanguage (with production rules) for defining language syntax; some have used a metalanguage for only part of the syntax, leaving the remainder for natural-language explanation; some have used notation which is not amenable to automatic processing. The advantages of a true syntactic metalanguage are given in the introduction to BS6154:1981. The main ones can be summarized as conciseness, precision and elimination of ambiguity, and suitability for automatic processing for purposes like producing tools such as syntax analysers and syntax-directed editors.

At the time of publication of this Technical Report, formal semantic definition methods suitable for programming languages form an active research area, making it impractical to provide any definite guidelines concerning whether to adopt a particular method, or any method at all, hence the recommendation to drafting committees to look at the position current when they begin work on their standard.

2 One of the purposes of including element 5) in 4.1.1 is to ensure that the standard as a whole is accessible to non-specialist readers while still providing the exact definitions required by those who are to implement the language processors.

3 Any formal method used may be specified by reference to an external standard or other definitive document, or may need to be specified in the standard itself (e.g. an annex providing a complete definition). In either case an informal *description* of the formal method should be included [element 5) of 4.1.1] so that for many purposes the standard can be read as a self-contained document even by those unfamiliar with the particular formal method concerned. As this guideline itself indicates, in deciding on matters of this kind, the current policies governing use of formal methods will need to be observed.

4.1.3 Guidelines on the use of character sets

The guidelines in this clause cover two aspects of the use of character sets. The first four (4.1.3.1 to 4.1.3.4 inclusive) are guidelines relating to the need for international interchange of programs, and hence are based on the principle of using a minimal set of characters which can be expected to be common to all systems likely to use the programs. In general these four guidelines are based on the default assumption that the form of representation of the program is not critical for the application concerned. In some cases, however (such as a program to convert text from one alphabet to another), interchange cannot be general but limited to processors capable of handling wider character sets. The guidelines are based on the principle that standards should ensure that interchange of programs without such application dependence will be generally possible.

The fifth guideline, 4.1.3.5, deals with the issue of collating sequences.

The sixth guideline, 4.1.3.6, deals with the issue not of interchange, but ensuring that users of standard-conforming processors will be able to handle wider ranges of character sets than that needed for international interchange.

At the time of publication of this Technical Report, much work is in progress relevant to these guidelines, and standards committees are strongly urged to ascertain the current status of such work before applying these guidelines and drafting character set requirements in standards.

4.1.3.1 Guideline: Character sets used for program text

As far as possible, the language should be defined in terms only of the characters included within ISO 646, avoiding the use of any that are in national use positions. If any symbols are used which are not included within ISO 646 or are in national use positions, an alternative representation for all such symbols should be specified. A conforming processor should be required to be capable of accepting a program represented using only this minimal character set. Great care should be taken in specifying how "non-printing" characters are to be handled, i.e. those characters that correspond to integer values 0 to 32 inclusive and 127, i.e. *null* (0/0) to *space* (2/0) and *delete* (7/15).

NOTES

1 The motivation here is to provide a common basis for representing programs, which does not exist with current (published up to 1990) standards. The characters that are available in all national variants of ISO 646 cannot represent programs in many programming languages in a way that is acceptable to programmers who are familiar with the U.S. national variant (usually referred to by its acronym "ASCII"). In particular, square brackets, curly brackets and vertical line are unavailable.

Further, the characters that are available in the International Reference Version of ISO 646 cannot represent programs in many programming languages in a way that is acceptable to programmers who are familiar with a particular national variant of ISO 646. For example, neither the pound nor dollar symbol may be available. The characters that are available in ASCII cannot represent programs in many programming languages in a way that is acceptable to programmers because their terminals support some other national variant of ISO 646.

Consideration needs also to be given to the use of upper and lower case (roman) letters. If only one case is required it should be made clear whether the other case is regarded as an alternative representation (so that, for example, *TIME*, *time*, *Time*, *tImE* are regarded as identical elements) or its use is disallowed in a standard-conforming program. Where both cases are required or allowed, the rules governing their use should be as simple as possible, and exactly and completely specified.

Of the non-printing characters, nearly all languages allow *space* (2/0), and *carriage return* (0/13) *line feed* (0/10) as a pair, though they differ as to whether these characters are meaningful or ignored. How *carriage return* without *line feed* (or vice versa) is to be treated needs consideration, as do constructions such as *carriage return, carriage return, line feed*. If characters are disallowed that do not show themselves on a printed representation, the undesirable situation may arise where a program may be incorrect though its printout shows no fault. If a tabulation character (0/9) is disallowed, this can cause trouble, since it appears to be merely a sequence of spaces; if allowed, the effect on languages such as Fortran, having a given length of line, has to be considered.

2 The characters that are available in the eight-bit character sets ISO 4873 with ISO 8859, or ISO 6937/2, would be sufficient to represent programs in a way that looks familiar to most (but not APL) programmers. However, in 1990 these standards are not yet widely supported on printers and display terminals.

3 For advice on character set matters, committees should consult the ISO/IEC JTC1 subcommittee for coding.

4.1.3.2 Guideline: Character sets used in character literals

Character literals permitted to be embedded in program text in a standard-conforming program should be defined in such a way that each character may be represented using one or more of the following methods:

a) The character represents itself, e.g. *A, B, g, 3, +, (*.

b) A character is represented by a pair of characters: an escape character followed by a graphic character, e.g. if *&* is the escape character, *&'* to represent apostrophe, *&&* to represent ampersand, *&n* to represent newline.

c) A character is represented by three characters: an escape character followed by two hexadecimal digits that specify its internal value.

Any conforming processor should be required to be able to accept "as themselves" [i.e. as in 1)] at least all printable characters in the "minimal set" defined in 4.1.3.1, apart possibly from any special-purpose characters such as an escape character or those used to delimit literal character strings.

NOTES

1 For reasons of portability it is necessary to provide a common basis for representing character literals in programs, in addition to the characters used for the program text itself. The required character set could be wider than (and for general purpose text handling would need to be wider than) that which is necessary for representation of program statements. Programs must be representable on as many different peripherals and systems as possible; the number of characters required to represent a program therefore needs to be reduced