TECHNICAL REPORT

ISO/IEC TR 10176

Second edition
1998-09-01

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

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

Reference number
ISO/IEC TR 10176:1998(E)

Contents

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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, but in exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types may be proposed:

— type 1, when the necessary 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 ISO/IEC Joint Technical Committee JTC 1, *Information technology,* Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This second edition cancels and replaces the first edition (ISO/IEC TR 10176:1991), which has been technically revised.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

Introduction

Background: Over the last three decades (1966-1998), 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 first edition of this Technical Report was produced during the 1980s to put together some of the experience that had been gained to that time, in a set of guidelines, designed to ease the task of drafting committees of programming language standards. This second edition enhances the guidelines to take into account subsequent experiences and developments in the areas of internationalization and character sets.

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 documented and made available, (e.g. in an informative annex of the programming language standard). This and the reason therefore can be taken into account at future revisions of the programming language standard or this technical report.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange.*

ISO/IEC 2022:1994, *Information technology — Character code structure and extension techniques.*

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

ISO/IEC 4873:1991, *Information technology — ISO 8-bit code for information interchange — Structure and rules for implementation.*

ISO/IEC 6937:1994, *Information technology — Coded graphic character set for text communication — Latin alphabet.*

ISO/IEC 8859-1:1998, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1.*

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.*

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.*

ISO/IEC TR 11017:1998, *Information technology — Framework for internationalization.*

ISO/IEC 11404:1996, *Information technology — Programming languages, their environments and system software interfaces — Language-independent datatypes.*

ISO/IEC 14977:1996, *Information technology — Syntactic metalanguage — Extended BNF.*

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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 difference. 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, 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

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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.

3.6 Terms related to character and internationalization

3.6.1 octet:

An ordered sequence of eight bits considered as a unit.

3.6.2 byte:

An individually addressable unit of data storage used to store a character, portion of a character or other data.

3.6.3 character:

A member of a set of elements used for the organization, control, or representation of data.

NOTE The definition above is that from the standard developed by ISO/IEC JTC 1/SC2. This ensures that the term “character” used in this TR is consistent with the coded character set standard. The composite sequence of ISO/IEC 10646 is not considered as a character. Each element of a composite sequence (as it is in ISO/IEC 10646) is considered as a “character” in this TR.

3.6.4 combining character:

A member of an identified subset of the coded character set of ISO/IEC 10646 intended for combination with the preceding non-combining graphic character, or with a sequence of combining characters preceded by a non-combining character.

3.6.5 composite sequence:

A sequence of graphic characters consisting of a non-combining character followed by one or more combining characters.

NOTES

1 A graphic symbol for a composite sequence generally consists of the combination of the graphic symbols of each character in the sequence.

2 A composite sequence is not a character and therefore is not a member of the repertoire of ISO/IEC 10646.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

3.6.7 coded character:
A character together with its coded representation.

3.6.8 basic character set:
A character set that is common across every execution environment of a programming language, e.g. the invariant set of ISO/IEC 646.

3.6.9 extended character set:
A character set that is used in an execution environment, e.g. ISO/IEC 10646-1. In most cases, the repertoire of the extended character set is larger than the basic character set.

3.6.10 character datatype:
Character datatype is a family of datatypes whose value spaces are character sets.

NOTE The value space of the character datatype should be wide enough to represent every member of extended character set, if the repertoire list of characters to be stored in the character datatype is not specified explicitly.

3.6.11 octet datatype:
Octet datatype is the datatype of 8-bit codes, as used for character sets and private encodings.

NOTE The value space of the octet datatype is wide enough to represent every member of basic character set, but may not be wide enough to every member of extended character sets.

3.6.12 octet string datatype:
Octet string datatype is the datatype of variable-length encoding using 8-bit codes.

NOTE The octet string datatype may be used to represent a member of extended character sets.

3.6.13 multi-byte representation of character:
A coded character represented by using a sequence of bytes (one-octet byte, two-octet byte, or four-octet byte).

NOTES

1 A character that is encoded by UTF-8 (UCS Transformation format) specified by ISO/IEC 10646-1/Amd.2 and stored in an octet-string datatype is an example of the multi-byte representation of a character. The size of a coded character encoded by UTF-8 is up to six octets, therefore it may occupy up to 6 one-octet bytes in the octet string datatype.

2 To handle the multi-byte representation of character correctly in an octet string datatype, the character boundary needs to be distinguished from the octet(s) boundary. Otherwise a multi-byte representation of character may be bisected as the result of octet base string manipulation, thus becoming no longer a character. In following reference the multi-byte representation of a character will be abbreviated as multi-byte character.

3.6.14 multi-octet representation of character:
A coded character stored in a character datatype that size is equal to or larger than two octets with whose values are multiple octets.

NOTES

1 A character that is encoded by UCS-2 stored in a character datatype is an example of the multi-octet representation of character. The size of a coded character encoded by UCS-2 is always two octets, therefore it can be considered as a coded character that is represented by single two-octet byte.

2 In following reference the multi-octet representation of a character will be abbreviated as the multi-octet character.

3 A coded character represented by UTF-16, which is specified by ISO/IEC 10646-1/Amd.1, is categorized in both multi-byte and multi-octet character, because the byte size of UTF-16 is two-octet, but a character may occupy 1 or 2 two-octet bytes in a octet string datatype.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

3.6.15 collation:
The logical ordering of strings according to defined precedence rules.

3.6.16 cultural convention:
A convention of an information system which is functionally equivalent between cultures, but may differ in presentation, operation behaviour or degree of importance.

NOTE Time zone, Summer time, Date and time format, Numeric format, Monetary format, Collation sequence, and Character classification, are examples of cultural convention.

3.6.17 cultural convention set:
A set of cultural conventions to be referred to by each programming language standard.

3.6.18 execution environment
An environment where a program is executed.

NOTES

1 An execution environment of program is not always the same as the compilation environment of the program.

2 Coded character sets supported by execution environment and input from the environment to program may vary from one to another. For example, ISO/IEC 8859-1 may be supported by an environment, and ISO/IEC 10646-1 may be supported by another environment.

3.7 Auxiliary verbs used in this TR
3.7.1 shall:
An indication of a requirement on programming language standard or processors.

3.7.2 should:
An indication of a recommendation to programming language standard or processors.

3.7.3 may:
An indication of an optional feature of programming language standard or processors. When this Technical Report provides a recommendation to the programming language standard that supports a specific optional feature, the auxiliary verb “may” is used in the sentence explaining the condition.

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.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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 a 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 ISO/IEC 14977:1996. 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 analyzers and syntax-directed editors.

2 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

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

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.

3 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.

4 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 standard should ensure that it is possible within the language to support the handling of a wide range of character sets, including multi-octet character sets, e.g. ISO/IEC 10646-1, and non-English single octet character sets, e.g. ISO/IEC 8859-1.

NOTES

1 For some applications, and for some classes of users for all applications, it is vital for the language to have the ability to accept and manipulate data from character sets other than the minimal character set needed for the basic purpose of specifying programs. For some users this need will be greater than the need for international interchange. An important task for any language standards committee is to ensure that it is possible for each of these needs to be met in a standard-conforming way.

2 Some programs will require both the ability to manipulate multi-octet and multi-byte characters and the capability of international interchange. This may imply two or more alternative representations of the same “character” (data object), one of which will be a representation (for interchange purposes) in the minimal character set defined in 4.1.3.1.1.

3 In general it should be possible to use non-English single-octet, multi-octet and multi-byte coded character sets in program text, character literals, comment, and data without recourse to the use of processors which are not standard-conforming. Programs using such characters in program text, literals or comments may not be standard-conforming and in general will be less portable internationally than those using only the minimal character set, but may still be portable within the applications community for those programs. Defined mappings from other character sets to the minimal character set of the language, and the presence of suitable processor options, are likely to maximize benefits and use-ability for differing requirements.

4.1.3.1 Guidelines on character sets used in program text

The guidelines in this clause covers the considerations on the character sets used in programming language source code, i.e. characters used for syntax of programming language, user-defined identifier, character literal, and comments.

4.1.3.1.1 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/IEC 646, avoiding the use of any that are in national use positions. If any symbols are used which are not included within ISO/IEC 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), in case of ISO/IEC 646 coded character set.

The guideline relates to the need for international interchange of programs, and hence is 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 this guideline is 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 larger character sets. The

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

guideline is based on the principle that standards should ensure that interchange of programs without such application dependence will be generally possible.

NOTES

1 The motivation here is to provide a common basis for representing programs, which does not exist with current (published up to 1998) standards. The characters that are available in all national variants of ISO/IEC 646 cannot represent programs in many programming languages in a way that is acceptable to programmers who are familiar with the International Reference Version of ISO/IEC 646 that is equivalent 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/IEC 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/IEC 646. For example, the pound symbol may not be available. The characters that are available in ISO/IEC 646 IRV (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/IEC 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 coded character sets ISO/IEC 4873 with ISO/IEC 8859-1, or ISO/IEC 6937-2, would be sufficient to represent programs in a way that, in the Western European and American cultures, looks familiar to most (but not APL) programmers.

3 The character sets that are available in the multi-octet coded character set of ISO/IEC 10646-1 would be sufficient to represent programs in a way that looks familiar to most programmers from most cultures. However, in 1998, the standard is not yet widely supported on printers and display terminals.

4 For advice on character set matters, committees should consult the ISO/IEC JTC 1 subcommittee for character coding.

4.1.3.1.2 Guideline: Identification of characters used for program text

The programming language standard should provide an annex containing a correspondence table between the graphic representation of the characters used for program text and character identifiers specified by ISO/IEC 10646.

NOTE It is possible to write program text using a character set that includes characters whose shapes are identical or very similar to one another. For example, in ISO/IEC 10646-1, “LATIN CAPITAL LETTER A”, “GREEK CAPITAL LETTER ALPHA”, and “CYRILLIC CAPITAL LETTER A” have identical shapes. Also the shape of “FULL WIDTH LATIN CAPITAL LETTER A” is very similar to these. In addition to that, ISO/IEC 10646-1 specifies many “non-printing” characters that occupy a certain amount of space in the presentation of text. In some programming languages, these “non-printing” characters act as token delimiters. Therefore, if a programming language standard specifies a character used for program text only by using its shape, it is ambiguous whether this shape means the identical or a similar shape (e.g. in the case of COBOL, character “A” means both “LATIN CAPITAL LETTER A” and “FULL WIDTH LATIN CAPITAL LETTER A” if the character appears in program text not in data) or a particular one of them (e.g. only “LATIN CAPITAL LETTER A” in the above example). Adoption of this guideline avoids such ambiguity.

4.1.3.1.3 Guideline: Character sets used in user-defined identifiers

The programming language standard should define which, and in what way, characters outside the “minimal” set defined in 4.1.3.1.1 can be used in user-defined identifiers. If characters outside of the minimal set are permitted, then the characters listed in annex A should be allowable.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

NOTES

1 It is important to allow characters from outside the minimal set to be used in user-defined identifiers in program text, to improve understandability for programmers whose native language is not English.

2 Using an extended character repertoire for user-defined identifiers may have an adverse effect on the portability of the program concerned.

3 As an alternative way to represent characters outside of the minimal set in a user-defined identifier by using the minimal character set for program portability, an escape character or an escape sequence followed by character short identifier standardized by ISO/IEC JTC 1/SC2, can be considered. For example, if *&u* is an escape sequence, *&u000000C1* represents LATIN CAPITAL LETTER A WITH ACUTE. The SC2 specified the code value of characters in ISO/IEC 10646, represented by 4 or 8 hexadecimal digits, for the character short identifier.

4 In the case that a programming language standard allows use of combining characters for user-defined identifier, the language standard need not require that a composite sequence is recognized as equivalent with the character which is pre-composed from the composite sequence.

4.1.3.1.4 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 an escape character or an escape sequence followed by character short identifier, e.g. if *&u* is an escape sequence, *&u000000C1* represents LATIN CAPITAL LETTER A WITH ACUTE.

d) A character is represented by three, five or nine characters: an escape character followed by two, four or eight hexadecimal digits that specify its internal value, e.g. if *&* is the escape character, the internal value of LATIN CAPITAL LETTER A can be represented by *&41* in the case of ISO/IEC 646, and can be represented by *&0041* or *&00000041* in the case of ISO/IEC 10646-1 depending on its forms, i.e. Two-octet Basic Multi-lingual Plane (BMP) form or Four-octet canonical form respectively.

Any conforming processor should be required to be able to accept “as themselves” [i.e. as in a)] at least all printable characters in the “minimal set” defined in 4.1.3.1.1, apart possibly from any special-purpose characters such as an escape character or those used to delimit literal character strings.

Any conforming processor should be required to be able to accept method c) to represent a character literal outside of “minimal set” defined in 4.1.3.1.1, any “non-printing character”, or any special-purpose character, in a way that is independent from any coded character set which is used to represent a source code in a machine readable format.

The programming language committee should consider to provide the means to accept “as themselves” [i.e. as in a)] all printable characters in the ISO/IEC 10646-1, apart possibly from any special-purpose characters such as an escape character or those used to delimit literal character strings, for character literal, e.g. a pre-processor to translate character literals represented by method a) to method c).

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 to the minimum that is consistent with general practice and readability. On the other hand, programs themselves need to be able to represent and process as many different characters as possible.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

These two needs make it impossible to represent every character by itself in a literal character string if the language is to be suitable for general processing of character data.

2 A particular problem arises with the representation of a space in a character or string literal. It can be represented by a visible graphic character, the argument in favour being that blank spaces in program text should not affect the meaning. However, it can also be represented by itself, the argument in favour being that this is the most natural form of representation. The indistinguishability of a tabulation character from a sequence of spaces (in a printed representation) is a particular problem since a function that returns the length of a string, in characters, may give different results from two programs that appear identical. There can be further complications when using a "high quality" printer with variable-width characters. Drafting committees are recommended to pay particular attention to these points.

3 The character short identifier referred to by method c) is standardizedd by ISO/IEC JTC 1/SC2, and the SC2 uses the code value of characters in ISO/IEC 10646, represented by 4 or 8 hexadecimal digits, for the character short identifier.

4 The character set in ISO/IEC 6937 represents some graphic characters as a pair of octets. This is suitable for printing but is difficult to process in operations such as comparison and sorting.

4.1.3.1.5 Guideline: Character sets used in comments

The programming language standard should define the characters that are permitted in comments in a standard-conforming program. For comments, the programming language standard should permit as wide a repertoire of the characters as possible.

NOTE For publication in the pages of a journal, some languages make no restriction on permitted characters in comments, beyond making it clear where the comment finishes. For inclusion on a computer file, however, it is preferable to restrict the characters to those that are widely available, to help portability. Since comments are intended for human reading and hence escape mechanisms are unnecessary, there is no disadvantage in printing characters simply representing themselves (apart of course from any characters or sequences of characters marking the end of the comment), and in limiting non-printing characters to those (like carriage return and line feed) necessary for layout purposes.

4.1.3.2 Guideline: Character sets used for data

The programming language standard should be defined in such a way that it is not assumed that character data processed by a program is anything other than a sequence of octets whose meaning depends on the context. However, a conforming processor should be required at least to be able to input, manipulate and output characters from the minimal character set defined in 4.1.3.1.1 above.

The standard should also specify whether, and in what way, support for ISO/IEC 10646-1 is required to be provided.

NOTES

1 The objective here is to provide a common basis for processing data. Many programs will assume that their data is expressed in ISO/IEC 646 IRV (ASCII) or some other versions of ISO/IEC 646. But if the standard assumes that all data is expressed in any one particular character set, it will cause difficulties for some users of other coded character sets.

2 See also the guideline on collating sequences 4.1.3.5 below).

4.1.3.3 Guidelines on datatypes for character data

4.1.3.3.1 Guideline: Character datatype

The programming language standard should provide a character datatype whose value space is the entire repertoire of the extended character set in an execution environment.

NOTES

1 In the case that the value space of a character datatype is not specified explicitly, by using the repertoire list that enumerates allowable repertoire of characters for the datatype, the default value space of the character datatype should be the entire repertoire of the extended character set.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

2 The repertoire of the extended character set may be processor-defined, but the language standard should not restrict the repertoire.

The character datatype should be independent from any coded character set.

NOTE The character datatype may be sub-typed to restrict its value space specified by a character repertoire list (see 4.1.3.3.3), but it should not be sub-typed by an encoding scheme of character data. For example, a distinct or a subtype of the character datatype that is unique to the encoding scheme of ISO/IEC 10646-1 should not be provided. The characters in the ISO/IEC 10646-1 should be handled through a generic character datatype that is independent from any coded character set, as long as the programming language does not address the object code level portability. For the programming languages that address the object code level portability, such as Java, use of ISO/IEC 10646 encoding is recommended for the character datatype.

4.1.3.3.2 Guideline: Octet and octet string datatype

In addition to the character datatype (see 4.1.3.3.1), a programming language standard may use the octet or the octet string datatype for character data.

NOTES

1 The value space of the octet datatype is large enough to represent the entire repertoire of the basic character set, but may not represent the entire repertoire of the extended character set.

2 The use of octet or octet string datatype for character data would be effective to keep the portability of programs that assume the size of the datatype for character. For example, some program may share the same memory area between character string and data of another datatype, e.g. union statement of C language. If the size of a datatype for character becomes changed in order to contain an extended character set, the alignment of memory area assigned for the data becomes broken. In order not to impact on existing programs that assum the size of character datatype is an octet, the programming language standard could use the octet or the octet string datatype for character data, in addition to the character datatype for backward compatibility of such program.

3 The programming language standard may allow use of the octet string datatype to represent a wide range of characters, from outside the basic character set, by means of a sequence of values of the octet string datatype, i.e. multi-byte character (See also 4.1.3.7).

4.1.3.3.3 Guideline: Subtypes of character datatype

A programming language standard may provide sub-types of the character datatype or may provide multiple distinct character datatypes, by specifying a character repertoire list, in order to restrict the character set that can be assigned into the sub-type or the character datatype. An example of the sub-type of character datatype is **kind=n** of FORTRAN. If the programming language standard provides such sub-types of character character datatype or multiple distinct character datatypes, inter character datatype assignment and comparison should be processor-defined.

NOTE Assignment from a character datatype whose value space is ISO/IEC 646 IRV to another character datatype whose value space is ISO/IEC 10646 is an example of the inter character datatype assignment.

4.1.3.4 Guidelines on character handling

4.1.3.4.1 Guideline: Character classification

The programming language standard should provide the means of testing whether a character data belongs to subsets of the extended character set (character classes) likely to be of importance in programs, such as alphabetic, alphanumeric, upper case letters, lower case letter, decimal digit, hexadecimal digit, control character, punctuation character, printable character, graphic character, and space character. The programming language standard should require that the means supplied does not depend on a specific coded character set, and may require, or permit, the provision of such means of testing for further user-defined subsets (user-defined character class) that are culture-specific or natural language-specific.

NOTE For example, LATIN CAPITAL LETTER A could be classified in alphabetic, alphanumeric, uppercase, hexadecimal digit, printable, and graphic character subset, but not in lower case, decimal digit, punctuation nor space character subset.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

4.1.3.4.2 Guideline: Character transformation

The programming language standard should provide the means to transform a character to another. The means provided by the standard should not depend on any specific coded character set, any specific culture, nor any specific natural language.

NOTES

1 Transformation from an upper case letter to the corresponding lower case letter and from a full width letter to the corresponding half width (normal) letter are examples of character transformation.

2 This character transformation functionality should be usable by a programmer, but not necessarily applied when a language processor is parsing the program text.

3 The mapping rule such as upper case to lower case mapping is culture and natural language specific.

4.1.3.5 Guideline: Collating sequences

The programming language standard should specify completely the default collating sequence to be provided by a conforming processor, and preferably that this should be that implied by the ordering of the characters in the minimal character set drawn from ISO/IEC 646 as defined in 4.1.3.1.1 above. If the default collating sequence is other than that implied by ISO/IEC 646, means should be provided whereby the user may optionally switch to the ISO/IEC 646 collating sequences, and consideration should be given to providing means for the user optionally to switch to alternative collating sequences, whether or not the defined default collating sequence is that based on ISO/IEC 646.

If a programming language standard provides the functionality to switch collating sequence from one to another, the cultural convention set switching mechanism described in 4.7.1 could be used for the purpose, since the collation sequence is a cultural convention.

NOTES

1 Programs which perform ordering of character data are in general not portable unless the collating sequence is completely defined. This guideline ensures that such programs will be portable at least where only those characters drawn from the minimal character set defined in 4.1.3.1.1 are used.

2 Drafting committees may wish to consider further guidance relating to characters not included in the minimal character set, especially where ordering of character data is a major anticipated use of the language.

3 Possible means of including alternative collating sequences are language features or processor options (see 4.1.9).

4 Possible reasons for wishing to provide such alternative means are to obtain maximum processing efficiency by use of a processor-defined internal character set, or to allow orderings more useful for particular purposes, e.g. a=A < b=B < ... < z=Z. (ISO/IEC 646 implies 0 < 1 < ... < 9 < A < B ... < Z < a < b ... < z, which is not always convenient.)

5 The international default ordering of character strings that consist of characters defined by ISO/IEC 10646, the switching mechanism of the ordering from the default to an alternative sequence, and language independent string comparison APIs, are presently being standardized towards ISO/IEC 14651.

4.1.3.6 Guideline: Multiple-octet coded character sets

The programming language standard should provide a character datatype whose value space is an extended character set representable by a multiple-octet code. The programming language standard should ensure that at least every character specified by ISO/IEC 10646 can be a value of the character datatype.

The programming language standard need not require that a composite sequence of ISO/IEC 10646 be recognized as a single character. Each character in a composite sequence should be stored in an extended character datatype and processed separately. The programming language standard may specify functionality to test the boundary of a composite sequence in a character string, and to convert the composite sequence into the corresponding pre-composed character, if it exists.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

If a programming language standard has a requirement to store a composite sequence in single value of a datatype, the programming language standard committee should consider the provision datatype distinct from other character datatypes, whose values include composite sequences of characters, and provide functionality to convert a character string to and from a value of this datatype or to and from a string of this datatype.

4.1.3.7 Guideline: Multiple-byte coded character sets

A programming language standard may support characters using the multi-byte representation. If the programming language standard supports a multi-byte representation of characters, the standard should provide both or either of the following functionality.

a) Convert the multi-byte character stored in an octet string datatype to the corresponding character stored in an character datatype, and vice versa.

b) Test or find out the character boundary of a multi-byte character in an octet string datatype.

4.1.4 Guideline: Error detection requirements

Requirements should be included covering error detection, reporting and handling, with appropriate conformity clauses. The standard should specify a minimum set of errors which a conforming processor must detect (in the absence of any masking errors); minimum level of accuracy and readability of error reports; whether an error is fatal or non-fatal; and, for non-fatal errors, the minimum recovery action to be taken.

NOTES

1 The objective of this guideline is to enhance the value of standards to users. The inclusion of requirements on error detection, reporting and handling provides a minimum level of assurance to the programmer of assistance from the processor in identifying errors.

2 See 3.3.1 for a definition of the term “error” in this context.

3 That an error is statically determinable (see 3.3.1) does not imply that the processor must necessarily determine it statically rather than dynamically.

4 It is recognized that requiring provision of specific error detection requirements within the standard entails a certain overhead in a conforming processor. It is a matter for each standards committee to determine how severely such overhead will affect the users of the language concerned, and consequently whether requiring detection is worthwhile. It is of course open to the committee to specify or recommend the provision of processor options which would permit the user to control the use of error detection (see 4.1.9).

4.1.4.1 Checklist of potential errors

The following is a list of typical errors which can arise in the submission of program text to a processor. Drafting committees should check all of the following for relevance to their language, and the standard produced should address all that are appropriate, plus others specific to the language concerned. This list is not to be considered either as exhaustive or as prescriptive.

In all cases the standard should specify whether the error concerned is fatal or non-fatal. Depending on the design and philosophy of the language, it may occur that a particular usage is not invalid (whereas it would be in another language) but that users would nevertheless benefit from the availability of a warning message within the processor.

4.1.4.1.1 Errors of program structure

a) unmatched brackets - either open without close, or vice versa.
NOTE This covers all sorts of bracket: (), [], {} etc.;

b) unmatched structure - similarly. (e.g. **begin-end**, **IF-ENDIF**, **repeat-until**, **ELSE without IF**, etc.);
NOTE In some languages, such as Algol 68, it is not meaningful to try to distinguish between this and a);

c) line number missing (e.g. in Basic);

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

d) absence of program heading (e.g. in Pascal);

e) constructs in disallowed order (e.g. parameter statement after data statement in FORTRAN, or **if...then for...do...else** in Algol 60);

f) program incomplete (e.g. no main program in FORTRAN);
NOTE In many languages this is a particular case of b);

g) program overcomplete (e.g. two main programs in FORTRAN);
NOTE In many languages this is a particular case of b);

h) section of program that cannot be accessed;
NOTE This is disallowed in (e.g.) FORTRAN, but is not a fault in many languages;

i) limitation on construct violated (e.g. too many continuation lines in FORTRAN, level 01 statement starting in incorrect margin in COBOL);

j) construct in disallowed context (e.g. declaration in Pascal statement-part).

4.1.4.1.2 Transfer of control

a) reference to non-existent or out-of-scope label;

b) transfer into a loop or procedure body;
NOTE In some languages this is included in a);

c) exit from function instead of normal return.

4.1.4.1.3 Words and numbers

a) unknown or misspelt keyword;

b) undeclared identifier;

c) duplicated identifier;

d) invalid syntax of numerical value (e.g. two decimal points).

4.1.4.1.4 Procedures

a) function that does not define its result (e.g. no assignment to function identifier in FORTRAN or Pascal);

b) call of unknown procedure or other named program segment (e.g. attempt to **PERFORM** non-existent paragraph in COBOL);

c) wrong number of arguments in procedure call;

d) wrong type of argument in procedure call.

4.1.4.1.5 Data structures

a) array declared with too many dimensions;

b) attempt to select element of non-existent structure (e.g. **A[i]** where **A** is not an array);

c) array variable unsubscripted (in context where subscript necessary);

d) incorrect number of subscripts;

e) use of unknown field selector;

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

f) incorrect type of subscript or selector;

g) invalid use of structure element (e.g. in many languages, array variable used as control variable of loop);

h) empty structure in disallowed context (e.g. character string in FORTRAN).

4.1.4.1.6 Lexical requirements

a) symbol not in character set.

4.1.4.1.7 Assignments

a) type incompatibility (e.g. **int** j; **real** x;...; j:=x; in Algol 68);

b) assignment to loop control variable (not a fault in some languages);

c) assignment to constant (e.g. **const** k=2; ... k:=4 in Pascal).

d) assignment between different datatypes (e.g. from character datatype to octet string datatype)

4.1.4.1.8 Program element structure

a) expression incorrectly formed (e.g. **A*-B** in FORTRAN);

b) incorrect statement syntax (e.g. **IF(A.EQ.B) 12, 15** in FORTRAN);

c) reference incorrectly formed;

d) declaration incorrectly formed.

4.1.5 Guideline: Exception detection requirements

Requirements should be included covering exception detection, reporting and handling, with appropriate conformity clauses. A minimum set should be specified of exceptions which a conforming processor must be capable of detecting (possibly by the user invoking a processor option). Conforming processors should be required to be capable of accurately reporting the occurrence of exceptions; whether an exception is fatal or non-fatal; and, for non-fatal exceptions, the recovery action to be taken.

NOTES

1 The objective of this guideline is to enhance the value of standards to users by the inclusion of requirements on exception detection, reporting and handling. This ensures a minimum level of “safety” to the user, e.g. in executing a program with incorrect data.

2 See 3.3.2 for a definition of the term “exception”.

3 That an exception is in general determinable only dynamically (see 3.3.2) does not imply that the processor is precluded from determining it statically rather than dynamically if the nature of the language itself and the processor concerned makes static detection feasible (see 4.1.6).

4 It is recognized that languages exist which do not in themselves recognize the concept of “exception” in the sense that any syntactically correct program is regarded as executable even if the consequent output may be empty or meaningless. Nevertheless it is recommended that in such cases standards committees consider requiring processors to provide an appropriate amount of detection and reporting of specified conditions (chosen to suit the particular language, see 3.3.3) which can arise during program execution, as a processor option (see 4.1.9).

5 It is recognized that requiring provision of specific requirements within the standard for the detection of exceptions entails a certain overhead in a conforming processor. It is a matter for each standards committee to determine how severely such overhead will affect the users of the language concerned, and consequently whether requiring detection is worthwhile. It is of course open to the committee to specify or recommend the provision of processor options which would permit the user to control the use of exception handling (see 4.1.9).

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

4.1.5.1 Checklist of potential exceptions

The following is a list of typical exceptions which can arise during execution of a program by a processor. Drafting committees should check all of the following for relevance to their language, and the standard produced should address all that are appropriate, plus others specific to the language concerned. This list is not to be considered either as exhaustive or as prescriptive.

In all cases the standard should specify whether the exception concerned is fatal or non-fatal. Depending on the design and philosophy of the language, it may occur that the occurrence of a particular event is not invalid (whereas it would be in another language) but that users would nevertheless benefit from the availability of a warning message within the processor.

When considering requirements in this area, drafting committees may well need to take execution overhead into account, which for some languages, some processors or some applications could be considerable. A possible way of dealing with conflicting priorities (e.g. between speed and safety) for differing applications could be to specify that processor options (see 4.1.9) should be available to allow the level and extent of checking to be controlled.

4.1.5.1.1 Data operations

a) attempt to divide by zero;

b) numeric overflow on arithmetic (floating-point or fixed-point, including integer) operation;

c) numeric underflow on floating-point operation;
NOTE It is recommended that a processor option be specified, to permit the user to treat such an exception as non-fatal, replacing the underflow value by zero and continuing, or as fatal, which would be the default;

d) attempt to raise a negative value to a non-integral power (where a real arithmetic result rather than a complex arithmetic result is expected);

e) attempt to raise zero to a negative or zero power;
NOTE Even where the language accepts and defines the result of such an operation it is recommended that the processor be capable of treating such a condition as a non-fatal exception;

f) overflow upon string or list concatenation;

g) attempt to perform an operation undefined for an empty string or list (e.g. **car(L)** in Lisp, where **L** is empty);

h) operation undefined for value (e.g. **succ(last)** in Pascal, or ordering operation attempted on item of (unordered) set type);

i) attempt to perform operation on an undefined value;

j) attempt to dereference a nil pointer value;

k) attempt to delete a non-existent item;

l) overlapping assignment (e.g. **A[2:5]=A[m:n]** where **m=1** and **n=4** - valid in some languages);

m) operation requiring dynamic storage allocation (not a fault in many languages).

n) truncation of a multi-byte character

o) data (code value) is not in repertoire

4.1.5.1.2 Violations of aggregate limits

a) subscript out of range;

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

b) substring reference out of range;

c) incorrect dimensionality in array reference;

d) unrecognized dynamically generated field selector of record;

e) index of control flow switch out of range;
NOTE For example, index out of implied range in “computed GOTO” statements; while this may not be an exception in the language - the default being to proceed to the next statement - the possibility of a warning or non-fatal exception message being available should be considered;

f) value of case selector not allowed for.
NOTE Similar remarks apply as for e).

4.1.5.1.3 Procedure calls

a) unable to execute call (e.g. named procedure unavailable);

b) mismatch between actual and formal parameters (in number, datatype, or other attributes);

c) recursive call of procedure in disallowed context (e.g. where the language does not support recursion, or a recursive procedure must specifically be declared as such);
NOTE Though some such cases can be detected as errors, the possibility of indirect recursion, including through the use of procedure parameters, means that consideration must also be given to detecting them as exceptions;

d) argument out of defined range for intrinsic function (e.g. **sqrt(x)** where **x** is negative).

4.1.5.1.4 Input-output operations

a) attempt to open file which cannot be found;

b) attempt to open file which is already open;
NOTE Perhaps non-fatal though it may indicate incorrect file naming;

c) illegal file name;
NOTE File names may be generated dynamically;

d) attempt to access (for input or output) file to which access is unauthorized;
NOTE It is advisable not to require in the standard the provision of an unnecessary amount of information or lower levels of security than provided by the host environment. Any message should be aimed at a legitimate user who has merely omitted to unlock a protected file for read or write access, and who will be able to obtain the needed information and take the necessary action without direct assistance from the processor;

e) unexpected end of file during input;
NOTE May be fatal, non-fatal or condition-raising, depending on the language;

f) required record not found on input (in random-access input);

g) attempt to input from output-only file (e.g. printer stream);

h) attempt to output to input-only file (e.g. keyboard);

i) attempt to create a record which already exists;

j) attempt to replace a non-existing record;

k) attempt to close file already closed.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

4.1.5.1.5 System limitations and characteristics

a) insufficient memory available for specified operation;

b) time limit exceeded;

c) limit on depth of nesting (e.g. of recursion) exceeded;

d) use of non-standard dynamic processor-defined extension;

e) language/culture dependent service is not available;

4.1.6 Guideline: Static detection of exceptions

The standard should specify that, where a processor will detect, solely by inspection of the program text, that an exception may (or will) occur if an otherwise well-formed program is executed, a processor option (see 4.1.9) is to be provided whereby the user may choose how the anticipated exception is to be handled.

NOTES

1 In a particular case the most appropriate form of handling will depend on the nature of the exception in the context of the application and the stage of development of the program. This cannot be foreseen either by the standard or by the designer of the processor if the action is left processor-dependent. Provision of a user-controlled processor option reduces the need for the user to include devious codes to "program around" restrictions.

2 In the case of a fatal exception, it is recommended that the default option be to treat the statically-detected exception as if it were a fatal error, an alternative option being to treat it as a non-fatal error and to continue processing (until, unless some other action intervenes, the anticipated fatal exception is encountered).

3 In the case of a non-fatal exception, it is recommended that the default option be to treat the statically-detected exception as if it were a non-fatal error and to continue processing (until, unless some other action intervenes, the anticipated non-fatal exception is encountered, and thereafter as if the non-fatal error had not been anticipated), an alternative being to treat it as a non-fatal error but not to proceed to execution.

4 The recommendations in notes 2 and 3 above do not preclude the provision of further alternative options.

4.1.7 Guideline: Recovery from non-fatal errors and exceptions

Where the standard permits recovery mechanisms from error or exception conditions, the required results of the actions to be taken by the processor (when such a recovery mechanism is invoked) should be defined as fully as are defined the normal semantic features of the language.

NOTE The objective of this guideline is to improve the predictability of processor action in the case of recoverable faults. Users of standard-conforming processors should be able to expect a similar degree of consistency of behaviour in such circumstances as they do with normal programs.

4.1.8 Guideline: Requirements on user documentation

Requirements on the documentation which is to be provided with a standard-conforming processor should be included. Some particular requirements of this kind may be found in ISO/IEC TR 10034. Committees may wish to extend the documentation requirements which those guidelines recommend.

NOTES

1 The value of standards to users is enhanced by the inclusion of requirements on documentation, since to make effective use of a processor it is necessary that adequate documentation is available to explain its use. Specific examples will be found in ISO/IEC TR 10034.

2 This guideline does not specify the form in which the documentation is to be provided; this is also the case with ISO/IEC TR 10034. Some language committees may specify conventional manuals, others may specify "on-line" help systems, yet others may require both, or leave the question open, depending on the nature of the languages. However, it is envisaged that all should specify a reasonable level of minimal provision, in some form, in this area, at least to the level recommended in ISO/IEC TR 10034.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

3 Whatever form of documentation is required by the standard, it should be specified in such a way that the user of the processor can check by inspection that the processor conforms with such requirements. By the very nature of documentation this should be possible. Validation services should not be expected, and should not feel it necessary, to check conformity with requirements related to this guideline, except as envisaged in ISO/IEC TR 10034 and in ISO TR 9547.

4.1.9 Guideline: Provision of processor options

The standard should specify processor options required to be provided within a standard-conforming processor, including in each case a specification of standard default settings of the option and the form or forms in which the processor options are to be made available to the user.

NOTES

1 The aim here is to widen the range of facilities guaranteed to the user by standard conformity of a processor. When a processor is being used, almost always some facilities are needed in addition to the ability to process standard-conforming programs and to detect programs which are not standard-conforming, depending on the particular application; this guideline assures the user that a standard-conforming processor will provide at least a minimum set of such facilities.

2 “Processor option” in this context means an option for the user which the processor is required to supply, not a facility which the processor may optionally provide.

3 Options may be provided, for example, as “switches” set when the processor is invoked by the user, or as “processor directives” embedded in a standard-conforming program.

4 Default settings of an option could possibly vary between different types of processor, such as compilers or interpreters.

5 In some cases it will be appropriate to require the option to be provided both statically - e.g. processor option - and dynamically - e.g. processor directive or interactive session command.

6 In general the form of provision of a required option can be left processor-dependent, though where it is invoked by a directive embedded in the program text, a program invoking it will not be standard-conforming or (e.g. if the directive is embedded in “pragmatic comments”) will not be fully portable unless the form is specified in the standard.

7 A checklist of appropriate options is given in 4.1.9.1. The choice from these or others to be covered in a particular standard is a matter for the individual language committee to determine in the light of the nature of their particular language.

8 Provision of processor options is sufficiently common that this guideline, and many of the specific items listed in 4.1.9.1, can be regarded as recommending standardization of “existing practice”.

9 It should be noted that, for purposes of validation of conformity, e.g. by a registered validation service or agency, each possible combination of settings of options produces, in general, a different processor requiring validation. It is not reasonable to expect that the effect on conformity of all possible combinations of settings can be checked and validated. Rather than, as a consequence, limiting the number of options or removing them from the standard, drafting committees are recommended to ensure that

— checking that the provision of options is in accordance with the standard can, as far as possible, be performed by the user;

— the requirements upon provision of options are so designed as to limit the validation overhead, e.g. by making as many as possible checkable independently without interaction with the effects of other options.

4.1.9.1 Checklist of potential processor options

Drafting committees should consider all of the following features as potential areas for specifying standard processor options, and the standard produced should address all that are appropriate for the language and types of processor covered:

— the handling of non-standard features;

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

— the use of machine-dependent or processor-dependent features;

— the type(s) of optimization;

— the use of overlays;

— the selection of debugging, profiling and trace options, including post-mortem dumps;

— the handling of errors, exceptions and warning messages;

— the handling of array bound, overflow and similar range checking;

— the control of output listing and pagination, including any listing of variable attributes and usage and listing of object or intermediate code;

— operating modes, such as execution automatically following compilation;

— the mapping of relevant language elements (such as files or input-output channels into corresponding elements of the host environment);

— the use of preconnected files and their status on termination;

— the rounding or truncation of arithmetic operations;

— the precision and accuracy of representation and of arithmetic, as appropriate;

— the default setting of uninitialized variables;

— in the case where a language standard is a revision of an earlier standard, the detection within programs, and reporting, of usage incompatible with the old standard.

NOTES

1 It may well be appropriate in many cases to specify several different settings of a given option, or a hierarchy of combinations of settings, though see note 9 of 4.1.9 above.

2 See also 4.1.6 and 4.4.

4.1.10 Guideline: Processor-defined limits

Minimum levels should be specified of guaranteed translation time and run-time support to be supplied by conforming processors in appropriate circumstances, namely where

a) it is probable that programs in the language may encounter processor-defined limits in the implementation of the language, and

b) such limits can be expressed in terms of the logical behaviour of programs (rather than implementation issues such as storage capacity);

and provide advice on choice of actual levels.

NOTES

1 Users should be able to feel assured of a guaranteed minimal level of support from a conforming processor. Severe processor restrictions (e.g. inability to handle **SET OF CHAR** in Pascal) impede portability; at a minimum, all such restrictions should be documented. In all the cases listed above, it is desirable that programmers be able to rely on a specified minimum, while allowing processors to supply additional capability if they so choose.

2 The limits specified in the standard may be semantic or syntactic, depending on the language.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

3 As can be seen from the checklist below, it is clear that some of these requirements upon processors may be interdependent, and drafting committees are advised to pay particular attention to ensuring mutual consistency between them. Attention also needs to be paid to the implications of having to meet all the limits on provision simultaneously; for example, it may be relatively simple for a processor to meet any individual one of these limits, but meeting them all at once places a much greater demand upon the resources of the underlying system supporting the processor.

4.1.10.1 Checklist of potential processor-defined limits

Examples of features for which it may be appropriate to specify minimal limits in standards are

— length of character strings;

— range of integers;

— internal precision of real numbers;

— magnitude of real numbers;

— number of files which can be open simultaneously;

— number of dimensions for arrays;

— number of array elements;

— length of external names;

— length of records which can be read or written;

— length of keys in keyed files;

— length in characters of a line of source text;

— length in items of a list-structured object;

— depth of nesting of various constructs (e.g. lists, records, procedure calls, loop constructs);

— number of items in various program constructs (e.g. declarations or statements in a block or compilation unit, procedures or modules in a package) and the accumulated length of such items.

Particular care is needed where limit requirements impinge on the external world, for example in the context of mixed language processing (see 4.6.4).

4.1.10.2 Actual values of limits

When advising implementors on considerations involved in setting the actual values of processor-defined limits, note that such advice may do one or more of

— recommending specific values;

— recommending minimum useful values;

— recommending maximum useful values;

— recommending that limits should depend on processor thresholds where efficiency changes sharply (such as word size, or memory size);

— recommending that limits should depend on resource availability, which may fluctuate during processing;

— setting forth other criteria appropriate to the specific language.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

In each case the reasons for the recommendations should be explained. Different recommendations may be appropriate for different limits.

It should be noted that appropriate processor-defined limits need to be made accessible to users, in particular for those performing conformity testing, as well as being documented. Where this is not available through language facilities (such as environmental enquiry functions), appropriate guidance to implementors should be provided.

4.2 Guidelines on presentation

4.2.1 Guideline: Terminology

As far as possible, the standard document should use the terminology given in the appropriate parts of ISO 2382, taking into account common practice in the language community concerned and possible costs of transfer to new terminology (see 4.5.4). Additional terms not covered by ISO 2382 should be defined in a specific section of the standard, and these additional terms should be registered with the appropriate subcommittee of ISO/IEC JTC 1.

NOTES

1 The objective of this guideline is to avoid unnecessary variations in terminology between standards for different languages. In general, the same word should be used for the same concept in all language standards; this aids “programmer portability” between languages, mutual understanding, and promotion of commonality between languages, and also strengthens the credibility of standards generally by making sure that one standard recognizes the existence and validity of other related standards.

2 Any divergence from standard terminology should be explicitly documented in the glossary section of the standard. Where for historical reasons a different word is commonly used, the standard should record this fact in an appropriate way, and could use that different word in any informal language definition included as an annex. Similarly the same word should not be used for different concepts in different language standards, and explanations should similarly be incorporated.

4.2.2 Guideline: Presentation of source programs

A consistent format should be adopted for textual presentation of source programs, and should be used in the relevant programming language standards documents for examples of language constructs, program fragments, and complete programs; when determining this format, such matters as indentation, how to break up long statements into lines, etc. should be taken into account.

NOTES

1 Guidance from standards committees on matters of source program presentation is useful to implementors trying to determine how to present source code listings, to those developing utilities (e.g. prettyprinters) which transform syntactically correct programs into programs formatted in a universally recognized way, to those publishing programs, and more generally to the community of language users who read and maintain programs.

2 In recommending consistency of appearance of programs in standards documents, there is no suggestion that standards, or drafting committees, should specify style.

4.3 Guidelines on processor dependence

4.3.1 Guideline: Completeness of definition

The number of aspects within its scope that the standard leaves not completely defined should be minimized (and preferably eliminated altogether). Where full definition is impracticable, in general such aspects should be required to be processor-defined, subject where appropriate to specified minimal or other limits, rather than left as processor-dependent or undefined. In this case, a complete checklist should be provided of all such processor-defined features [see 4.1.1, elements 6) and 7)], guidance should be provided for implementors, required limits (see 4.1.10), as appropriate, should be specified, and the documentation accompanying the processor should be required to provide for the user a full specification of the processor definitions used.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

NOTES

1 Though in particular cases counter-arguments to this guideline may exist on the grounds of “flexibility”, everything within the scope of a standard which is left undefined, processor-dependent or processor-defined weakens the standard and harms portability. Flexibility may sensibly be provided within the standard itself in the form of guaranteed ranges of facility for the user, but not as unguaranteed variations in provision which are outside the control of the user.

2 This guideline applies to matters within the scope of the standard and it is important that the definition of scope is itself sufficiently precise that it is clear when a matter is outside the scope. Where genuine doubt can exist - or simply as an aid to the user of the standard, to avoid misunderstanding - it may be appropriate to state explicitly that something is undefined by the standard. However, the scope of a standard should not be given contrived precision by the use of exclusion clauses which remove from its definition aspects which, given the objective of the standard, fall naturally within it.

4.3.2 Guideline: Optional language features

Inclusion within the standard of optional language features, whether as optional additions or as optional alternatives, should be minimized.

NOTES

1 The argument here is similar to note 1 under 4.3.1. Language options provided for the user within the standard are acceptable provided the choice is with the user. Language options which may or may not be available and are out of the control of the user are not acceptable.

2 Ideally, the aim should be to have no optional features at all.

4.3.3 Guideline: Management of optional language features

Where complete avoidance of language options is impracticable, they should be organized in levels so that each level is a pure subset of all higher levels, and the number of different levels should be minimized.

NOTES

1 If a standard contains N optional features (whether separate facilities, or modules containing several facilities), this implies the existence of 2 to the power N different possible combinations and hence different processor configurations. This severely harms portability and greatly increases the problems of validation.

2 Drafting committees will always have to balance the arguments against levels and subsets, the arguments against making the language and its implementations too large, and the dangers of leaving extensions to provide further functionality outside the standard and hence liable to be provided in incompatible ways.

3 Revision of an existing standard offers an opportunity to reduce the number of options and levels, including by migration of optional features to mandatory features.

4.3.4 Guideline: Syntax and semantics of optional language features

Whenever a language feature is made optional in a standard, whether by inclusion in a level higher than a minimal level, or otherwise, and if a processor accepts, syntactically, a standard-conforming program beyond the level or subset for which standard conformity is claimed, then the standard should require that, nevertheless, the processor must process that program in the way described by the standard.

NOTES

1 The aim of this guideline is to ensure consistency of semantics. It must be possible to be sure that any syntax defined in the standard, whether optional or not, means the same thing in any standard-conforming implementation, and that if a feature is described in the standard, whether optional or not, it is provided in the same way in all standard-conforming implementations.

There can also be the problem that a processor claiming conformity only at a lower level may still provide equivalent functionality to some language feature at a higher level, but provide it with different syntax. Any program using that functionality will not be standard-conforming. Standards committees may wish to consider whether this is a likely scenario with their language which might cause serious problems, and whether some further conformity statement or at least warning might be appropriate.

2 Detailed consequences of this general guideline are provided below (see 4.3.5, 4.3.6).

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

4.3.5 Guideline: Predefined keywords and identifiers

The standard should specify that any standard keyword or identifier defined in any section of a language standard, whether optional or not, retains the same standard-defined meaning throughout the whole standard and applies to all standard-conforming processors, at whatever level, even if, when optional, the keyword or identifier is not directly supported by the processor.

NOTES

1 In line with 4.3.4, this guideline ensures consistency of use of standard-defined words.

2 This applies, for example, to COBOL reserved words, FORTRAN keywords, Pascal word-symbols and required identifiers, and predefined identifiers such as the names of standard datatypes, and to the names of optional built-in functions; but it does not preclude redefinition within a program of the meaning of a standard-defined identifier if the language (and the standard) permits this (e.g. by application of scope rules).

4.3.6 Guideline: Definition of optional features

As far as possible, any optional (or higher level) features should be defined functionally in terms of mandatory (or lower level) features.

NOTES

1 This guideline enhances portability because a user of (say) a lower level processor but who needs higher level features can implement those features individually in a (functionally) standard-conforming way.

2 The purpose of including such higher level features in the standard is often to relieve the user of the need to implement them individually, and (very often) so that the implementor can provide them more efficiently than can a user with only the lower level language features available. (A simple example is that of the standard intrinsic functions commonly required to be supplied by a standard-conforming processor, many of which - like the common trigonometric or arithmetic functions - can be programmed in the language itself.) On the other hand the purpose of providing them as options or higher level features is often so that users will not have to “pay” in some way to get features they will never or will rarely use. This guideline simply recognizes this and suggests a means whereby it can be taken into account without impairing portability.

It is recognized that some optional or higher level features are intrinsically incapable of being treated in this way and it is not suggested that they should therefore be avoided. However, it may be felt appropriate to point out in the standard that their use has a greater impact on portability than those which are expressible in terms of mandatory or lower level features.

4.3.7 Guideline: Processor dependence in numerical processing

Where a major anticipated use of the language is for arithmetic processing, means whereby the user may specify and interrogate the range, precision and accuracy of arithmetic operations should be included in the standard.

NOTES

1 Because of the wide variety of data processing equipment with which languages are used, these features of numerical work are commonly left processor-defined or processor-dependent. While for many uses it is adequate for the default ranges, precisions and accuracy of arithmetic to be processor-defined, such variations severely inhibit the production of portable numerical software, and specifying lower limits (see 4.1.10) is only a partial solution.

2 Suitable means of providing such facilities may be specific language features, processor options, or binding of a language-independent facility.

3 Processor limits, as in 4.1.10, should still also be specified for processor-defined defaults.

4 It is recommended that processor (or language-independent facility) documentation be required to include a specification of the means (including algorithms for controlling accuracy) used to achieve requirements under this heading.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998

5 Drafting committees, and also implementors (through recommendations in element 7) of the standard, see 4.1.1) should seek guidance from professional numerical analysts on how to draw up and how to meet requirements under this heading.

4.4 Guidelines on conformity requirements

Guidelines on requirements for conformity to the standard may be found in ISO/IEC TR 10034. Particular attention is drawn to the need for consistency between requirements for different levels or options, if the standard permits subsets or optional modules.

4.5 Guidelines on strategy

4.5.1 Guideline: Secondary standards

Where existing standards do not address all of the issues proposed in these guidelines, standards committees should consider producing secondary standards to cover such matters (e.g. requirements upon processors).

NOTES

1 The advantage of the use of secondary standards is that they make it possible, in effect, to improve the content of the corresponding primary standards without introducing unnecessary delay, such as by having to wait for the next full revision.

2 See 3.5.1 for a definition of “secondary standard”.

3 This procedure could also be considered for standards not yet in existence but in an advanced stage of processing, where delay in order to introduce further requirements would be undesirable.

4.5.2 Guideline: Incremental standards

Standards committees should, in general, use incremental standards to add new constructs to existing languages rather than incorporate them in a complete revision.

NOTES

1 The advantage of incremental standards is that they make it possible, in effect, to augment the content of existing standards without introducing unnecessary delay, e.g. while waiting for the next full revision.

2 See 3.5.2 for a definition of “incremental standard”.

3 Consideration should always be given to producing a revised standard (to correct errors but not change the language except perhaps to extend existing constructs) and an incremental standard in parallel, rather than attempt to do the two together, though perhaps in such a way that the two could be merged at a later revision, after gaining experience of the new standard.

4 For an example of the incremental standards approach see ISO 1989/AMD1.

4.5.3 Guideline: Consistency of use of guidelines

Where guidelines in this Technical Report are applied in a primary standard, they should be applied, as appropriate, to related secondary, incremental and supplementary standards, in the same manner.

NOTES

1 The concept of secondary, incremental and supplementary standards will provide a mechanism whereby additions and corrections can be made to primary standards without the need to reconsider and reapprove those standards immediately. Standards committees should consider utilizing these mechanisms to revise portions of primary standards on a more frequent basis than is possible for the complete standard. To maintain stylistic compatibility, secondary, incremental and supplementary standards should follow the same form as the primary standard. This will enhance the ability of the committee to integrate any changes or modifications into the primary standard when that standard is updated as a whole.

STANDARDSISO.COM : Click to view the full PDF of ISO/IEC TR 10176:1998