ISO/IEC PDTR 10176

The second edition

Guidelines for preparation of programming language standards

November 19, 1995

Note: This text is for SC22 combining ballot of PDTR registration and PDTR consideration.

Contents

0 Introduction v

1 Scope 1

2 References 1

3 Definitions 1

4 Guidelines 5

4.1 Guidelines for the form and content of standards 5
4.1.1 Guideline: The general framework 5
4.1.2 Guideline: Definitions of syntax and semantics 6
4.1.3 Guidelines on the use of character sets 6
4.1.3.1 Guidelines on character sets used in program specifications 7
4.1.3.1.1 Guideline: Character sets used for program text 7
4.1.3.1.2 Guideline: Identification of character used for program text 7
4.1.3.1.3 Guideline: Character sets used in user-defined identifiers 8
4.1.3.1.4 Guideline: Character sets used in character literals 8
4.1.3.1.5 Guideline: Character sets used in comments 9
4.1.3.2 Guideline: Character sets used for data 9
4.1.3.3 Guidelines on character data types 9
4.1.3.3.1 Guideline: Basic character data type 9
4.1.3.3.2 Guideline: Extended character data type 9
4.1.3.3.3 Guideline: Subtype of character data type 9
4.1.3.4 Guidelines on character handling 10
4.1.3.4.1 Guideline: Presentation width 10
4.1.3.4.2 Guideline: Character classification 10
4.1.3.4.3 Guideline: Character mapping 10
4.1.3.5 Guideline: Collating sequences 10
4.1.3.6 Guideline: Multiple-octet coded character sets 10
4.1.3.7 Guideline: Multiple-byte coded character sets 11
4.1.4 Guideline: Error detection requirements 11
4.1.4.1 Checklist of potential errors 11
4.1.4.1.1 Errors of program structure 11
4.1.4.1.2 Transfer of control 12
4.1.4.1.3 Words and numbers 12
4.1.4.1.4 Procedures 12
4.1.4.1.5 Data structures 12
4.1.4.1.6 Lexical requirements 12
4.1.4.1.7 Assignments 12
4.1.4.1.8 Program element structure 13
4.1.5 Guideline: Exception detection requirements 13
4.1.5.1 Checklist of potential exceptions 13
4.1.5.1.1 Data operations 13
4.1.5.1.2 Violations of aggregate limits 14
4.1.5.1.3 Procedure calls 14
4.1.5.1.4 Input-output operations 14
4.1.5.1.5 System limitations and characteristics 15
4.1.6 Guideline: Static detection of exceptions 15
4.1.7 Guideline: Recovery from non-fatal errors and exceptions 15
4.1.8 Guideline: Requirements on user documentation 15
4.1.9 Guideline: Provision of processor options 15
4.1.9.1 Checklist of potential processor options 16
4.1.10 Guideline: Processor-defined limits 17
4.1.10.1 Checklist of potential processor-defined limits 17
4.1.10.2 Actual values of limits 17
4.2 Guidelines on presentation 18
4.2.1 Guideline: Terminology 18
4.2.2 Guideline: Presentation of source programs 18
4.3 Guidelines on processor dependence 18
4.3.1 Guideline: Completeness of definition 18
4.3.2 Guideline: Optional language features 19
4.3.3 Guideline: Management of optional language features 19
4.3.4 Guideline: Syntax and semantics of optional language features 19
4.3.5 Guideline: Predefined keywords and identifiers 19
4.3.6 Guideline: Definition of optional features 19

4.3.7 Guideline: Processor dependence in numerical processing . . . 20
4.4 Guidelines on conformity requirements . . . 20
4.5 Guidelines on strategy . . . 20
4.5.1 Guideline: Secondary standards . . . 20
4.5.2 Guideline: Incremental standards . . . 20
4.5.3 Guideline: Consistency of use of guidelines . . . 21
4.5.4 Guideline: Revision compatibility . . . 21
4.5.4.1 Classifications of types of change . . . 21
4.5.4.2 Difficulty of converting affected programs . . . 22
4.6 Guidelines on cross-language issues . . . 22
4.6.1 Guideline: Binding to functional standards . . . 22
4.6.2 Guideline: Facilitation of binding . . . 23
4.6.3 Guideline: conformity with multi-level functional standards . . . 23
4.6.4 Guideline: Mixed language programming . . . 23
4.6.5 Guideline: Common elements . . . 23
4.6.6 Guideline: Use of data dictionaries . . . 23
4.7 Guidelines on Internationalization . . . 23
4.7.1 Cultural convention switching mechanism . . . 24

Annexes

A Extended repertoire for identifier . . . 25

B Guidelines for preparation of Internationalized library . . . 26
B.1 Culture dependent services . . . 26
B.2 Multiple culture (locale) support . . . 26

C Issues on ISO/IEC 10646 handling . . . 27
C.1 Data type for ISO/IEC 10646 . . . 27
C.2 Coexistence of ISO/IEC 646 base single octet coded character set and ISO/IEC 10646-1 . . . 27
C.2.1 Source and execution character set . . . 27
C.2.2 Character manipulation between different execution coded character sets . . . 27
C.3 Identification and announcement mechanism of a specific coded character set . . . 27
C.4 Mapping between national variant of ISO/IEC 646 and ISO/IEC 10646-1 . . . 27

D Possible solution to ISO/IEC 10646-1 related issues . . . 28
D.1 File System Safe UTF (UTF-8) . . . 28
D.1.1 Introduction of FSS-UTF . . . 28
D.1.2 Possible usage of the FSS-UTF from the view point of programming language standard . . . 28
D.1.3 Recommendation . . . 28
D.2 A suggested mechanism to support extended repertoire for identifier . . . 29
D.2.1 Introduction . . . 29
D.2.2 First solution . . . 29
D.2.3 Second (quick and dirty) solution . . . 29
D.2.4 Conclusion . . . 30

E Multiple character sets support . . . 31

F Characterless programming . . . 32

G Comment instruction . . . 33

H Advanced data types for character and string handling . . . 34
H.1 Abstract Character Level . . . 34
H.1.1 Data type . . . 34
H.1.2 Literal resolution . . . 34
H.1.3 Encoded Character Level . . . 34
H.2 Text Level . . . 35

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) together form a system for worldwide standardization as a whole. 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 a technical committee is to prepare International Standards but in exceptional circumstances, 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, requiring wider exposure;

— 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.*

0 Introduction

Background: Over the last three decades (1966- 1995), 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 programming 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 technical report.

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.

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 processing – 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 systems – Vocabulary – Part 15: Programming languages*

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

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

ISO/IEC 8859-1:1987, *Information processing – 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, *Universal Multiple-Octet Coded Character Set (UCS)*

ISO/IEC DTR 11017:1995, *Information technology – Framework for internationalization*

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

3 Definitions

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

[Editor's note] Although the structure/hierarchy of definition clause in this draft is flat, the structure/hierarchy will be restored to the original one that is shown on the first edition of this technical report.

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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 2 See also 3.3.4

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

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

3.3.4 Relationship to other terminologyRelationship 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).

NOTE 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).

NOTE 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.

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

3.5 Secondary, Incremental and supplementary standards

3.5.1 Secondary standards

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

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

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

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

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

3.6 Internationalization related terms

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 or portion of a character.

3.6.3 character: A member of a set of elements used for the origination, control, or representation of data.

NOTE — The above definition Originates from ISO/IEC 10646. This ensures that the term "character" used in this TR is consistent with the coded character set standard. The combining sequence of ISO/IEC 10646 is not considered as a character. Each element of a combining sequence, 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.

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

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

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

3.6.7 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.

3.6.8 basic character data type: A character data type whose constituent values include all the characters of the basic character set but not all the characters of the extended character set.

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 extended character data type: A character data type whose constituent values include all the characters of the extended character set.

3.6.11 multi-byte representation of character A coded character represented by using a sequence of values of the basic character data type. A character that is encoded by UTF-8 (UCS Transformation format) specified by a PDAM of ISO/IEC 10646-1 and stored in an array of the basic character data type is an example of the multi-byte representation of character. The size of a coded character encoded by UTF-8 is up to 6 octets, therefore it may occupy an array consists of 6 basic character data type. To handle the multi-byte form of character correctly in the basic character data type array, character boundary need to be detached. Otherwise a multi-byte representation of character may be bisected as the result of basic character data type string manipulation, then become no longer a character. Here after, The multi-byte representation of character will be abbreviated as multi-byte character.

3.6.12 multi-octet representation of character: A coded character stored in a unit of a character data type that size is equal to or larger than 16 bits length. A character that is encoded by UCS-2 stored in a unit of an extended character data type that size is equal to or larger than 16 bits length is an example of the multi-octet representation of character. Since the character boundary of the multi-octet character data type is coincide with the boundary of the basic character data type in an array, character boundary detection is not necessary to be considered. Here after the multi-octet representation will be abbreviated as multi-octet character.

[Editor's note] The character data types of most or all of programming language standards are designed independently from the bit representation of character nor the size of the character data type. However, implementation of the programming language standard may have assumption of the size of a data type. Some of programming language allow to allocate the same memory address to multiple data types by using EQU statement of FORTRAN or union statement of C, for example. From the view point of allocation portability, it might be difficult to change the size of basic data type to be large enough to store every element of a extended character set. So, to store an extended character into character data type or character data type array, two different ideas are presented in this TR, i.e. to store a multi-byte character in a basic character data type array and to store a multi-octet character in a extended character data type.

3.6.13 character cell terminal: The presentation device that character font is fixed width and fixed pitch. VGA display device that can present 80 columns by 25 lines, and fixed width font printer that can print 132 columns are examples of character cell terminals.

3.6.14 presentation width: The number of columns that is required to present a character on a character cell terminal. Although only one column is required to present a Latin alphabet, complicated shape character, e.g. Asian Ideographs, may take multiple columns on a character cell terminal.

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 behavior or degree of importance.

[Editor's note] The origin of the above definition is DTR 11017. The definition should be carefully reviewed and checked if it is appropriate from the programming language standard point of view.

3.6.17 cultural convention set: A set of cultural conventions of which support services are defined to be ideally located within the application Platform. (Services are normally available through Application Program Interfaces (API)).

3.7 Auxiliary verbs used in this TR

3.7.1 shall: An indication of a requirement on programming language standard.

NOTE — Since this document is not International Standard, this Technical Report does not have any requirement for conformity. Therefore, no "shall" is used in the body of this Technical report.

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

3.7.3 may: An indication of optional feature of programming language standard. 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:

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

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

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

d) 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.

e) One or more annexes containing an informal description of the language, a description of the metalanguage used in a) and any formal method used in b), 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.

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

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

h) 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 a) to d) above.

i) 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.

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

k) An annex that outlines the significant portability capabilities that are provided by the standard and indicates what user requirements are addressed by the standard.

l) An annex on "portability considerations" for interfaces, data and users.

[Editor's note] The last two guidelines, i.e. k) and l) are come from JTC1 resolution based on JTC1/TSG-1 recommendation on application portability. Those two guidelines are not directly related with internationalization.

NOTE 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.

NOTE 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 3 below).

NOTE 3 It is to be made clear that the annexes referred to in elements e) to l) 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 a), b), c) and d). It should be explicitly stated that, in any case of ambiguity or conflict, it is the standard as specified in elements a), b), c) and d) that is definitive. Note that, if a definition (as opposed to description) of any formal method used in elements a) and b) 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).

NOTE 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.

NOTE 5 Element i) 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 c) 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.

NOTE 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 BS 6154. The main ones can be summarized as conciseness, precision and elimination of ambiguity, and suitability for automatic processing for purposes like producing tools such as syntax analysers and syntax-directed editors.

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

NOTE 2 One of the purposes of including element e) 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.

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

4.1.3 Guidelines on the use of character sets

The guidelines in this clause cover two aspects of the use of character sets. The first guideline 4.1.3.1.1 is guideline relating to the need for international interchange of programs, and hence are based on the principle of using a minimal set of characters which can be expected to be common to all systems likely to use the programs. In general 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 guideline is based on the principle that standards should ensure that interchange of programs without such application dependence will be generally possible.

On the other hand, other guidelines relating to the need for handling capability of wide range of character sets by programming languages. 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 and non-English single octet character sets, e.g. ISO/IEC 10646.

NOTE 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.

NOTE 2 Some applications 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.

NOTE 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.

[Editor's note] The second paragraph of this clause and notes come from "Guideline: Use of other character set" in the first edition of this TR. Since one of the objectives of this revision the TR to provide guidelines for ISO/IEC 10646 support by programming language standard, and SC22 has resolved that every programming language standard should be able to support ISO/IEC 10646 where appropriate, the character set used or supported by programming language standards should no longer be limited within the minimal character sets as far as not to obstruct international portability or interchange of programs. Therefore, the guideline for supporting wider range of character sets should not be isolated in a sub clause but applied widely except the one for program specifications.

4.1.3.1 Guidelines on character sets used in program specifications

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.

NOTE 1 The motivation here is to provide a common basis for representing programs, which does not exist with current (published up to 1995) 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.

NOTE 2 The characters that are available in the eight-bit coded character sets ISO/IEC 4873 with ISO/IEC 8859-1, or ISO 6937-2, would be sufficient to represent programs in a way that, in the Western European and American cultures to most (but not APL) programmers. However, in 1995 these standards are not yet widely supported on printers and display terminals.

NOTE 3 The character sets that are available in the sixteen or thirty one-bit 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 1995 the standard is not yet widely supported on printers and display terminals.

NOTE 4 For advice on character set matters, committees should consult the ISO/IEC JTC1 subcommittee for coding.

4.1.3.1.2 Guideline: Identification of character 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, "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" has very similar to above. In addition to

that, the ISO/IEC 10646-1 specifies many "non-printing" characters that occupy a certain amount of space in the presentation of text. In some of programming languages, those "non-printing" characters works as token delimiter. 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 that have the identical or 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 in ISO/IEC 10646 are permitted, then the characters listed in annex A should be allowable.

NOTE 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 by not English.

NOTE 2 Using an extended character repertoire for user-defined identifiers may have an adverse effect on the portability of application concerned. An example of possible solution to this portability problem is proposed in the annex D.2.

[Editor's note] The intent of this clause come from JTC1/SC22/WG20 N227. Since the document is under review by NBs of ISO/IEC JTC1/SC22 and WGs under SC22, the contents of this clause also be a subject to be changed resulting of the review.

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 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 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 "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, within an extended character data type literal.

[Editor's note] Since character data types may be classified as basic character data type and extended character data type, the character literal also can have either data type. For the literal that has basic character data type attribute, at least printable characters in the minimal set should be allowed for a) type representation. But for the extended character data type literal, outside of the minimal set should be allowed.

NOTE 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.

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.

NOTE 2 The preceding paragraphs envisage that inside a program each character is storable as a single octet (8-bit byte). for discussion of handling of multi-octet characters.

[Editor's note] Jon Diamond commented that the above note is incorrect, then should be removed. I personally though that although programming language standard itself does not have the dependence with the size of data type explicitly, but many of programmer does have the assumption. Therefore, having the note could be problem.

NOTE 3 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.

NOTE 4 The character set in ISO 6937-2 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.

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 what way, support for ISO/IEC 10646-1 is required to be provided.

NOTE 1 The objective here is to provide a common basis for processing data. Many programs will assume that their data is expressed in ASCII or some other variant 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 set.

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

NOTE 3 See annex C for ISO/IEC 10646-1 issues.

4.1.3.3 Guidelines on character data types

4.1.3.3.1 Guideline: Basic character data type

A programming language standard may provide a character data type whose constituent values include all the characters of the basic character set but not all the characters of extended character set.

NOTE 1 Provision of the basic data type would be effective to keep portability of program that assume the size of character data type. For example, some program may share same memory area between character string and another data type. If the size of character data type becomes changed in order to contain extended character set, the alignment of those data become broken. In order not to impact to existing bad mannered program, programming language standard can provide the basic character data type in addition to the extended character data type for backward compatibility.

NOTE 2 The programming language standard may allow to use of the basic character data type to represent a wide range of characters, from outside of basic character set, by means of a sequence of values of the basic character data type, i.e. multi-byte character. In the case, the programming language standard should also provide functionality to detect character boundary in the array of basic character data type, and conversion from multi-byte character in the basic character data type array to extended character data type and vice vasa.

NOTE — See also 4.1.3.7.

4.1.3.3.2 Guideline: Extended character data type

The programming language standard should provide a data type that can store every character used in a execution environment in a unit of the data type.

4.1.3.3.3 Guideline: Subtype of character data type

A programming language standard may provide sub-types of the character data type or may provide multiple distinct character data types in order to restrict the character set that can be assigned into the particular the particular sub-type or particular character data type. **kind=n** of FORTRAN is an example of sub-type of character data type. If programming language standard support such

sub-types of character or multiple distinct character data types, inter character data type assignment and comparison permitted. For example, assignment from **kind=1** character data type to **kind=2** character data type when kind=1 is for ISO/IEC 646 IRV and **kind=2** is for ISO/IEC 8859-1, although current FORTRAN standard does not permit such inter-kinds assignment.

4.1.3.4 Guidelines on character handling

4.1.3.4.1 Guideline: Presentation width

A programming language standard may provide a character string formatting functionality that assumes character cell terminal. If a programming language standard provides such functionality, the standard should not assume that the presentation width of a character on a character cell terminal is always one or the same with the byte length of the character code. The programming language standard also should provide a functionality that test the number of cells will be occupied when the character is presented on a character cell terminal.

4.1.3.4.2 Guideline: Character classification

The programming language standard should provide means of testing whether a given character value belongs to subsets of the extended character set, likely to be of importance in applications, such as alphabetic, alphanumeric, upper case letters, lower case letter, decimal digit, hexadecimal digit, control character, punctuation character, printable character, graphic character, 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 that are culture-specific or natural language-specific.

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

4.1.3.4.3 Guideline: Character mapping

The programming language standard should provide a functionality to convert a character to another that has equivalent character semantics with the former character. For example, case conversion from upper case to lower case, and Latin alphabet to corresponding braille letter.

NOTE 1 This character mapping functionality will be used by programmer, but not necessarily applied when programming language processor parsing the program text

NOTE 2 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.

NOTE 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.

NOTE 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.

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

NOTE 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.)

NOTE 5 The international default ordering of character string that consists of characters defined by ISO/IEC 10646 and the switching mechanism of the ordering from the default to an alternative sequence is presently being standardized as ISO/IEC 14651. The project JTC1 22.30.02.02 is now on the stage of working draft consideration.

4.1.3.6 Guideline: Multiple-octet coded character sets

The programming language standard should provide extended character data type whose values form 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 is a value of this extended character data type.

NOTE 1 The programming language standard ought not to require that a composite sequence of ISO/IEC 10646 be recognized as a single character. Each character in a composite sequence should should be stored in extended character data type 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, but not so strongly recommended.

NOTE 2 If a programming language standard has a requirement to store a composite sequence in a data type, the programming language standard should provide a distinct data type from character, whose values include composite sequences of characters, and provide functionality to convert a character string to a value of this data type or to a string of this data type.

4.1.3.7 Guideline: Multiple-byte coded character sets

A programming language standard may support characters in the multi-byte representation. If programming language standard supports multi-byte representation of character, the standard should provide both or either of the following functionalities.

a) Convert the multi-byte character stored in a basic character data type array to the corresponding multi-octet character stored in an extended character data type, and vice versa.

b) Test the character boundary of a multi-byte character in a basic character data type array.

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.

NOTE 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.

NOTE 2 See 3.3.1 for a definition of the term "error" in this context.

NOTE 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.

NOTE 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);

[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;

[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 character data types (e.g. from extended character data type to basic character data type)

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.

NOTE 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.

NOTE 2 See 3.3.2 for a definition of the term "exception".

NOTE 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).

NOTE 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).

NOTE 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).

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;

[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, data type, 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.

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 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.

NOTE 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 code to "program around" restrictions.

NOTE 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).

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

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

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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".

NOTE 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;

— 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.

NOTE 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.

NOTE 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.

NOTE 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.

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

NOTE 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.

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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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 f) and g)], 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 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.

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

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.

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

NOTE 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 data types, 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.

NOTE 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.

NOTE 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.

NOTE 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.

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

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

NOTE 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.

NOTE 5 Drafting committees, and also implementors (through recommendations in element g) 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).

NOTE 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.

NOTE 2 See 3.5.1 for a definition of "secondary standard".

NOTE 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.

NOTE 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.

NOTE 2 See 3.5.2 for a definition of "incremental standard".

NOTE 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.

NOTE 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.

NOTE 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.

NOTE 2 For guidelines relevant to secondary, incremental and supplementary standards see 4.5.1, 4.5.2, 4.6.1 and 4.6.2.

4.5.4 Guideline: Revision compatibility

For each proposed addition, deletion or modification that represents a potential incompatibility from an earlier standard

— the rationale for the proposed change should be stated;

— the way in which the proposed change will affect the original language feature should be determined, in accordance with the classifications in 4.5.4.1 below;

— the difficulty of converting affected programs should be assessed, according to 4.5.4.2 below;

— an attempt should be made to determine how widely the affected feature is used;

— all the above should be documented, and conversion guidance should be provided in the relevant section of the standard [see element i) of 4.1.1].

NOTE — Altering a standard in an incompatible manner during a revision may bring benefits but will also entail costs, and so should not be undertaken lightly. The rationale for a proposed change should include statements of

— Specific benefits, and how the benefits result from the change.

Benefits may fall into such categories as improved programming practice, better portability, better machine performance, elimination of ambiguity, or improved consistency and clarity of the language specification.

— Costs (other than those directly associated with compatibility, which are discussed below).

Costs may fall into such categories as use-ability, performance, or ease of learning.

4.5.4.1 Classifications of types of change

a) Change to semantics of well-defined feature. A change is made to the semantic specification of a feature for which the original document guarantees a reasonably precise result. The feature remains syntactically valid, but a program may now produce different results.

b) Deletion of semantically well-defined feature. A feature well-defined in the original document is rendered syntactically invalid by the new specification.

c) Deletion of semantically ill-defined feature. A feature which was not well-defined in the original document is rendered syntactically invalid by the new specification.

d) Clarification of semantically ill-defined feature. A feature which was not well-defined in the original document, so that its interpretation was open to question, is properly defined in the new specification. (This, strictly speaking, is not an incompatibility, since no guarantee has been withdrawn, but is included here for completeness since some past interpretations may not be compatible with that in the revised document.)

e) Change or deletion of obsolescent feature. A feature designated in the original document as obsolescent is deleted or changed in the new specification.

f) Change of level definition.

g) Change of processor defined limit.

h) Change of other processor requirement.

i) Change of conformity clause.

NOTE — Conversion problems (if any) in cases f) to i) are different from those in cases a) to e), where the language definition has been changed.

[Editor's note] The USA suggest the following items in the list, 1) different time format, 2) Change in collation sequence, and 3) Different character semantics. I'm not sure if addition of those detail is appropriate here. I think, all of above are processor defined behavior.

4.5.4.2 Difficulty of converting affected programs

At least four levels of difficulty may be distinguished. In doubtful cases use the more severe classification. From the standardization point of view, the following are listed in order of decreasing conversion effort:

a) **No possible translation**. There is no feasible way to implement the original function within the new standard.

b) **Semantic transformation**. The original function can still be performed using the language, but human translation, based upon knowledge of the purpose of the program, is required.

c) **Significant syntactic transformation.** A mechanical translation is feasible, but some analysis of the program structure as a whole may be required, or a significant amount of code may be generated.

d) **Simple syntactic transformation**. Old statements can be mechanically transformed to the new syntax with little or no knowledge of the rest of the program or its purpose.

NOTE 1 The extent of use of the affected feature may be estimated in terms of whether a high or low proportion of programs use the feature, or of frequency of use within programs. In making this estimate the drafting committee should consider the existing pattern of implementation. Thus, for example, even though many programs use the feature in question, few may actually be affected if the committee is simply ratifying existing practice. It is recognized that this estimate cannot be precise; the point is to distinguish at least between clearing up anomalous cases which are technically valid, but probably unused in practice, and changing features on which many programs may truly depend.

The inference is that a higher proportion of use would increase the total conversion cost. The conversion complexity, along with frequency of use, will provide a comparative measure of the conversion cost.

NOTE 2 Documentation will be needed under one or more of the following headings:

1) **Obsolescence**. A notification that the standard's support for the language feature in question is scheduled to be withdrawn in the next revision. This action allows users to plan a smooth evolution of their software base away from dependence upon the feature (or upon the old interpretation of the feature, if there is a semantic change).

2) **Documentation of transition semantics.** (see note 3)

3) Conversion guidance. This may be one or more of

i) A conversion program.

ii) An algorithm that is detailed enough to be understood by a reasonably informed user of the language.

iii) A commentary describing the conversion process.

iv) As much conversion information as possible, in cases when a well-defined conversion process is not feasible.

NOTE 3 **Transition semantics**: Transition between two interpretations of the same language feature may be provided in various ways:

1) The standard may require conforming implementations to make both the old and the new interpretations available to the user through the use of a user-controlled option (which itself may be part of the language, or provided as a processor option).

2) The standard may allow implementations to use either the old or the new interpretation, but with the old interpretation scheduled to be withdrawn in the next version. (The implementation should then be required to document which interpretation it is using.)

3) If it is judged that the costs of such measures outweigh the benefits, the standard may simply adopt the new interpretation and require implementations to provide a "flagging" capability which would detect and report cases of possible incompatibility. [Flagging may also be required in conjunction with case 2).] If this course of action is taken then the standard should be as specific as possible about the cases to be flagged and should provide appropriate guidance on the form of flagging, the user documentation which will be needed, and so on.

(In general, changing the interpretation of a language feature is to be avoided if possible, but may be essential in order to eliminate inconsistencies, or as enabling action to permit other desirable changes.)

NOTE 4 While this section applies primarily to revision of an existing standard, in cases where a new (initial) standard is based upon a previous informal, unofficial or "de facto" standard (for example a published and implemented language definition) the drafting committee may well find it appropriate to take into account at least some of the guidelines when preparing the formal standard.

4.6 Guidelines on cross-language issues

NOTE — At the time that this Technical Report is published, active work is in progress which is expect to result in another Technical Report giving guidelines for language bindings. The guidelines resulting from this work could lead to those described below in this section being modified or extended. Standards committees should therefore check on the progress of this work before applying these guidelines.

4.6.1 Guideline: Binding to functional standards

Where a binding is required between the programming language and a functional standard defined externally, the standards committee should ensure that this is specified in a supplementary (or incremental) standard to the functional specification, cross-referenced to the primary language standard.

NOTE — The objective here is to specify the location of the binding specification for a functional standard.

4.6.2 Guideline: Facilitation of binding

The standard should be designed so that it takes into account the existence of relevant existing or potential language-independent functional standards, in such a way that it facilitates binding (preferably by means of a supplementary standard), including the possibility of generic bindings for future functional standards.

NOTE 1 Many language processors currently obtain specialized functionality from language-independent subsystems provided by the host environment. Many users want and expect functionality to be provided in a uniform manner across language systems. Functional standards recognize both of these needs and it is beneficial for language standards to be so designed as to take account of them.

NOTE 2 Examples of existing (or emerging) functional standards are GKS (graphics), DBMS (database systems) and IRDS (information resource dictionary). Examples of potential functional standards are for communications facilities, screen management, mathematical library facilities, etc. It will be necessary for users to be able to invoke each of these from a variety of different languages.

4.6.3 Guideline: conformity with multi-level functional standards

The standards committee should ensure that the rules for conformity with multi-level functional standards are consistent with those for conformity with multi-level primary standards given above, both for programs and for processors.

NOTE 1 It is necessary, especially with the increasing number of functional standards, to ensure that the same criteria apply to conformity with external functional standards as to the primary language itself.

NOTE 2 The requirements for processors apply equally to subroutine packages (or their equivalent) which implement functional standards, since in the terms used in this Technical Report such a subroutine package will form part of the processor as far as binding to the functional standard is concerned.

4.6.4 Guideline: Mixed language programming

When specifying requirements upon conforming processors, possible needs of users for mixed language programming should be taken into account. These may include the incorporation of modules or segments in programs mainly written in another language, as well as the use of modules or segments written in another language. The committee should consider whether it is appropriate to require conforming processors to provide means to facilitate this, or to provide guidance to implementors on such provision.

NOTE — Standards are commonly designed as if the language and its community operate in an isolated world in which other languages do not exist. However, in practice many users of many languages find the need to invoke in some way facilities written in other languages. The growth of libraries of reusable software packages which are not necessarily available in a specific language, and the risk of decreased reliability in the configuration management of multiple copies of those packages, reinforces the need to give attention to this topic.

4.6.5 Guideline: Common elements

Those elements and properties within the language which may be held in common with those of other languages should be defined in the standard; the hierarchy of the elements should be specified; and the functionality of the common holding of definitions to be performed internal and external to the language should be specified.

NOTE — This guideline is aimed at ensuring that common elements may be used consistently across languages.

4.6.6 Guideline: Use of data dictionaries

Where a standard exists for a data dictionary and users of the language standard require access to that dictionary through their language, the semantics and the matching of the structures, elements and properties of those data elements in the language which are associated with the dictionary standard should be specified, preferably in a supplementary standard.

NOTE — The aim of this guideline is to remove any possible ambiguity between the data descriptors in the language and those in the dictionary, provide an additional check on the functionality, semantics and structure of the dictionary, and provide some commonality within the dictionary for the use of alternative languages.

The holding of elements within the dictionary requires definition of properties for each element type which, if commonly defined for instances, provides the necessary means for the validation of format, content, and relationship with other instances.

4.7 Guidelines on Internationalization

[Editor's note] Guidelines for how to internationalize programming language standards is described here.

4.7.1 Cultural convention switching mechanism

The programming language standard should provide the functionality to dynamically switch from one cultural convention set to another (e.g. setlocale function in C language). If the programming language supports multiple threads in a process, the cultural convention set binding should be done by thread, not by process.

The programming language standard should provide interfaces to the internationalization related functions described in annex B.

A Extended repertoire for identifier

The programming language standards should allow characters outside the repertoire of the Latin non-accented upper and lower case letters in identifiers. The repertoire recommended consists of characters that are used to write different natural languages of the world, and consists of letters, ideographic and syllabic characters.

Attention is drawn to that using extended repertoire for identifiers may impact source code portability. It is reasonable to assume that there exist work-arounds to this problem.

The recommended characters consists of the following characters of ISO/IEC 10646-1:1993 implementation level 1, given as their code points in UCS-2:

[Editor's note] The proposed characters that should be able to be used for identifier is subject to be changed, resulting of the review of WG20 N227 by NBs of SC22 and WGs under SC22.

```
Latin: 0041-005a,0061-007a,00c0-00d6,00d8-00f6,00f8-01f5,01fa-0217,
     0250-02a8,1e00-1e9a,1ea0-1ef9
Greek: 0384,0388-038a,038c,038e-03a1,03a3-03ce,03d0-03d6,03da,03dc,03de,
     03e0,03e2-03f3,
     1f00-1f15,1f18-1f1d,1f20-1f45,1f48-1f4d,1f50-1f57,1f59,1f5b,1f5d,
     1f5f-1f7d,1f80-1fb4,1fb6-1fbc,1fc2-1fc4,1fc6-1fcc,1fd0-1fd3,
     1fd6-1fdb,1fe0-1fec,1ff2-1ff4,1ff6-1ffc,
Cyrilic: 0401-040d,040f-044f,0451-045c,045e-0481,0490-04c4,04c7-04c8,
         04cb-04cc,04d0-04eb,04ee-04f5,04f8-04f9
Armenian: 0531-0556,0561-0587
Hebrew: 05d0-05ea,05f0-05f4
Arabic: 0621-063a,0640-0652,0670-06b7,06ba-06be,06c0-06ce,06e5-06e7,
Devanagari: 0905-0939,0958-0962
Bengali: 0985-098c,098f-0990,0993-09a8,09aa-09b0,09b2,09b6-09b9,
         09dc-09dd,09df-09e1,09f0-09f1
Gurmukhi: 0a05-0a0a,0a0f-0a10,0a13-0a28,0a2a-0a30,0a32-0a33,
          0a35-0a36,0a38-0a39,0a59-0a5c,0a5e
Gujarati: 0a85-0a8b,0a8d,0a8f-0a91,0a93-0aa8,0aaa-0ab0,0ab2-0ab3,
       0ab5-0ab9,0ae0,
Oriya: 0b05-0b0c,0b0f-0b10,0b13-0b28,0b2a-0b30,0b32-0b33,0b36-0b39,
       0b5c-0b5d,0b5f-0b61,
Tamil: 0b85-0b8a,0b8e-0b90,0b92-0b95,0b99-0b9a,0b9c,0b9e-0b9f,0ba3-0ba4,
       0ba8-0baa,0bae-0bb5,0bb7-0bb9,
Telugu: 0c05-0c0c,0c0e-0c10,0c12-0c28,0c2a-0c33,0c35-0c39,0c60-0c61,
Kannada: 0c85-0c8c,0c8e-0c90,0c92-0ca8,0caa-0cb3,0cb5-0cb9,0ce0-0ce1,
Malayalam: 0d05-0d0c,0d0e-0d10,0d12-0d28,0d2a-0d39,0d60-0d61,
Thai: 0e01-0e30,0e32-0e33,0e40-0e46,0e4f-0e5b,
Lao: 0e81-0e82,0e84,0e87,0e88,0e8a,0e0d,0e94-0e97,0e99-0e9f,0ea1-0ea3,
     0ea5,0ea7,0eaa,0eab,0ead-0eb0,0eb2,0eb3,0ebd,0ec0-0ec4,0ec6,
Georgian: 10a0-10c5,10d0-10f6,
Hiragana: 3041-3094,309b-309e
Katakana: 30a1-30fe,
Bopmofo: 3105-312c,
Hangul: 1100-1159,1161-11a2,11a8-11f9
CJK Unified Ideographs: f900-fa2d,
     fb1f-fb36,fb38-fb3c,fb3e,fb40-fb41,fb42-fb44,fb46-fbb1,fbd3-fd3f,
     fd50-fd8f,fd92-fdc7,fdf0-fdfb,fe70-fe72,fe74,5e76-fefc,
     ff21-ff3a,ff41-ff5a,ff66-ffbe,ffc2-ffc7,ffca-ffcf,ffd2-ffd7,
     ffda-ffdc,4e00-9fa5
```

B Guidelines for preparation of Internationalized library

B.1 Culture dependent services

[Editor's note] This clause of annex intends to refer to the internationalized library standard comes from JTC1 22.30.02.01 project. If the standard text does not become an stable stage, i.e. agreed by CD consideration ballot and the DIS text becomes available, the clause will be removed from this TR text and will postpone for the third edition of this TR. Input from the project JTC1 22.30.02.01 Functionality of the Internationalization of application is expected.

The ISO/IEC JTC1/SC22/WG20 is now developing the standard of the formal definition of cultural conventions. The standard intends to includes the formal definition of the following cultural conventions to be used by programming language for internationalization. Therefore, the programming language standard should provide the following cultural convention set support facility, where appropriate, or provide function call interface to the internationalization API that the WG20 may establish.

a) Calendar

b) Case Mapping

c) Currency symbols

d) Date format

e) Number format

f) String ordering and Comparison

g) Time format

[Editor's note] The above items are copied from Annex B of DTR 11017.

In addition to the above cultural convention set support facility, the programming language standard should provide a mechanism to bind or switch the cultural convention set dynamically at program execution time.

B.2 Multiple culture (locale) support

[Editor's note] In ISO/IEC JTC1/SC2, language tagging in a text has been discussing. When the standard is established, programming language standard may have to support dynamic cultural convention set switching in regard to the language/culture tag. This subclause is the place holder for the functionality. Input from Danish national body is expected.

C Issues on ISO/IEC 10646 handling

This annex provides guidelines that programming language standard should follow to support ISO/IEC 10646.

C.1 Data type for ISO/IEC 10646

There are several methods of handling ISO/IEC 10646 coded characters in character data types of a programming language. One is to use an extended character data type and another is to use multi-byte character representation, e.g. UTF-8 and UTF-16, in a basic character data type array. The following guideline should be applied to provide character data types to store ISO/IEC 10646 characters.

a) The programming language standard should provide extended character data type to store every character of ISO/IEC 10646.

b) The programming language standard does not need to recognize a composite sequence as an individual character. Therefore, it is not needed to normalize a composite sequence as the corresponding pre-composed character and store it a unit of extended character data type.

c) The programming language standard may provide basic character data type and store a ISO/IEC 10646 character by using multi-byte character representation. UTF-8 and UTF-16 would be examples to represent a ISO/IEC 10646 in a multi-byte character representation. In this case, programming language standard should provide a means to test character boundary in the basic character string, in order to enable application program to handle the character string by character by character.

d) The programming language standard may provide basic character data type and store combining or conjoining character in ISO/IEC 10646 in a multi-byte character form. In this case, programming language standard should provide a means to test character boundary in the basic character string, in order to enable application program to handle the character string by character by character.

e) The programming language standard should not provide character data type that is unique to ISO/IEC 10646, in order to programming language standard be coded character set independent.

C.2 Coexistence of ISO/IEC 646 base single octet coded character set and ISO/IEC 10646-1

C.2.1 Source and execution character set

The coded character set that is used in an execution environment might be different from the coded character set that was used to write the source code of a program. For example, source code is written using ISO/IEC 10646-1 and the program handles ISO/IEC 8859-1 as character data on an execution environment. If a programming language supports character literals as a constant value, then those literals should be evaluated appropriately for the expected execution coded character set. In other words, if a literal value "A" is compared with character data "A" in an execution environment, the answer should be equal regardless of the difference between source code coded character set and execution environment coded character set.

NOTE — Since the expected execution character set might not be known when the source code is processed it may be necessary for the source code character set to be known at program load/initiation/execution time.

C.2.2 Character manipulation between different execution coded character sets

A programming language may store different coded characters into character data type. Some of programming language standards have subtypes of character data type, e.g. kind=n in FORTRAN, to identify the difference of coded character sets. To perform character manipulation operation, e.g. compare or concatenate, between different coded character sets, code conversion to wider coded character set may be required, just like calculation between different arithmetical data types, e.g. integer and real. If programming language provide such code conversion or casting functionality, that functionality should be invoked automatically and be transparent from programmer's view point.

C.3 Identification and announcement mechanism of a specific coded character set

[Editor's note] This clause is the place-holder to contain guideline on announcement mechanisms of ISO/IEC 10646-1. This issue was assigned by SC22 plenary meeting. A flat character text of ISO/IEC 10646-1 may contains several control sequences, e.g. UCS coded representation form with implementation level, subsets of graphic characters, and signatures.

C.4 Mapping between national variant of ISO/IEC 646 and ISO/IEC 10646-1

[Editor's note] Some of programming language standards use national variants set of ISO/IEC 646 as syntactic characters of the programming language, for example back-slash character in the C language. So, there might be any portability issue to port a program source code that is written by a national variant set of ISO/IEC 646 to ISO/IEC 10646-1 environment. This clause is the place-holder for the issue.

D Possible solution to ISO/IEC 10646-1 related issues

D.1 File System Safe UTF (UTF-8)

[Editor's note] Since File System Safe UTF described in this annex is being standardized as UTF-8 by an amendment of ISO/IEC 10646-1, this annex will not be appeared in the final text of this TR, unless specific recommendation for programming language standard to use the UTF-8 will be provided.

This annex intends to propose applicability of the FSS-UTF into programming languages.

This annex intends to contain:

D.1.1 Introduction of FSS-UTF

ISO/IEC 10646-1, the Universal Multiple-Octet Coded Character Set (UCS), provides the capability to encode multi-script text within a single coded character set. However, because UCS is fully encoded, null octets and the code values of the other ASCII characters, including the code value of the ASCII slash ("/") character, are not protected. This makes the UCS character encoding incompatible with many existing ASCII based operating system implementations.

There are several challenges presented by UCS which must be dealt with by historical operating systems and programming languages. The most significant of these challenges is the encoding scheme used by UCS. More precisely, the challenge is the marrying of the UCS standard with existing programming languages and existing operating systems. Prominent among the operating system UCS handling concerns is the representation of the data within the file system. An underlying assumption is that there is an absolute requirement to maintain the existing operating system software investments while at the same time taking advantage of the use the large number of characters provided by UCS.

The UTF-8 transformation format was originally conceived as a file system safe transformation format of UCS to allow historically ASCII based operating systems to cope with representation and handling of the large number of characters that are possible to be encoded by UCS.

The UTF-8 transformation format encoding is intended to be used as a file code and an intermediate step towards full UCS support. However, since many historical operating system and programming language implementations face similar obstacles in supporting UCS, UTF-8 provides a common and compatible encoding during this transition stage.

The objective of UTF-8 is to provide an UCS transformation format which also meets the requirement of being usable on historical operating system file systems in a non-disruptive manner.

The UTF-8 transformation format encodes both UCS-2 and UCS-4 in a compatible format using multi-byte characters of lengths 1, 2, 3, 4, 5, and 6 bytes:

```
 Bits Hex Min Hex Max  Byte Sequence in Binary
1  7 00000000 0000007F 0vvvvvvv
2 11 00000080 000007FF 110vvvvv 10vvvvvv
3 16 00000800 0000FFFF 1110vvvv 10vvvvvv 10vvvvvv
4 21 00010000 001FFFFF 11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
5 26 00200000 03FFFFFF 111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
6 31 04000000 7FFFFFFF 1111110v 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
```

The UCS value is the concatenation of the v-bits in the multibyte encoding, where the v-bits are the 0's and 1's that constitute the UCS value.

D.1.2 Possible usage of the FSS-UTF from the view point of programming language standard

[Editor's note] T.B.D.

D.1.3 Recommendation

[Editor's note] T.B.D.

D.2 A suggested mechanism to support extended repertoire for identifier

[Editor's note] This annex intend to explain to the reviewers of this TR that there is a solution to support extended repertoire of identifier, and does not intend to recommend this mechanism to implementation of programming language. Therefore, this annex will not be appeared in the final text of this TR. This annex will be removed at provision of DTR text.

[Editor's note] Canadian contribution on an easy mechanism to support extended repertoire for identifier in programming language is intended to be described here. The following is row text of the Canadian contribution, it has not yet been edited for adaptation to this TR.

D.2.1 Introduction

It has been said that source code portability is seriously affected when characters outside of the ISO/IEC 646 invariant repertoire are used in programs. This argument is often used to influence to threaten programmers not to use richer character sets, which is somewhat affecting productivity of non-English speakers, an issue that is more subtle than non-tariff barriers in international trade, but that is nevertheless real, even if it is said that most programmers have a working knowledge of English (which is less true than technical forums generally believe).

Now, it is reasonable to believe that the programmers should not be threatened, annoyed or concerned by such a caveat. It should be the responsibility of compilers, or compiler providers, to provide a solution to this problem, and it should not be the responsibility of users.

This being said, here are briefly proposed 2 solutions to the reflection of SC22 and of those who absolutely want to influence to threaten.

D.2.2 First solution

Simply said, the first solution would solve the problem for a good while: it would be to standardize, for compilers being conformant to a language standard, a mandatory "character set editor" which would take care of code issues: in a word, it would be to state in future language standards that all characters of ISO/IEC 10646 are allowed for variable names of programming languages with the realistic recognition that the display and printing environment could use a less rich character set. A fallback would then be left to the discretion of the implementers to display and print characters that are not know to the target coded character set. But essentially, once and for all, source code would essentially be transmitted using ISO/IEC 10646-1. This measure is not so revolutionary when one thinks that for text, street people are used to exchange WP documents in an user-unreadable format such as WordPerfect format, MS-Word format and so on. So why not for program texts?

D.2.3 Second (quick and dirty) solution

But for those who absolutely want to lead dinosaurs at least into the 21st century, without changing things, there is another quick solution, very dirty, that is now left into the hands of the end-users (programmers, as end-users of compilers) but that should be provided by compiler makers.

While some will say it affects portability (and I will give an example to emphasize they are slightly right), what follows is a reasonable "work-about" (a word that I did not find in my English-French dictionaries, and that I learned in Tokyo from our convenor, himself not an English-speaking native).

Given variable the strictest name restrictions to the invariant character repertoire of ISO/IEC 646 (26 Latin uppercase [or lower case, according to the programming language] letters plus 10 digits, the first character restricted to the letters, and 8 characters per variable name), it is possible to design a preprocessor to convert variables using n-character variable names (n¿8) with each character being a 2- or 4-octet character from ISO/IEC 10646-1 (or a non ISO/IEC 646 character set which contains more than 96 characters that have to be downgraded in the same way) to preprocessor-generated names, of the form AXXXXXXX (where A is a base Latin letter, and X is either a base Latin letter or a digit). The preprocessor would make all replacements necessary (except in literals, the main area where portability is affected: but in this case using literals in a program is a bad programming practice for localisable or internationalized software anyway), including in comments. This gives a possibility of 26 x (36 to the 7th power) or roughly 2.000.000.000.000 variables in a given program, a reasonable limit within the current technology of human-generated programs (see example in figure 1)

Furthermore, since it is logical to think that in at least the system on which such program was entered, it was possible to display and print the original characters, the "preprocessor" could have an option to list equivalences of the original variable names with the generated ones.

Now, as I promised I will talk about where portability has chances to be broken with such a scheme: of course, as I said, if literals use rich characters (a bad practice but let's suppose it), the source program will not be portable as, if it is expected that comments will be parsed in such a scheme, literals should not. In this case, if variable names are displayed by the resulting compiled program, they will no longer correspond to the preprocessor- generated variable names. But this is not a big issue

because in such a case, it is very likely that the whole object program itself will not be portable because it assumes a given coding that has nothing to do with the target character set. This is but a single example of non-portable source code, strict sense.

But another portability issue is more important: the resulting program will be less readable by the recipients (who did not write it) and will have to understand it. Here, though, it is arguable that anyway, comments of such a program would be in a language that the recipient reader would not understand anyway, so it is not a big issue, unless, in order to go to the limit, the gentle intellectual terrorists push their arguments to say that a program which uses comments not written in English is not portable. But I assume they would not go that far. Nevertheless, if the resulting program is modified by the recipients, the author will have to use a "Compare" program to have himself notified of the changes and modify his original source accordingly. A very dirty situation indeed, but a ransom of "environmentalists" who want to preserve dinosaurs into the 21st century, again. Why not? After all, Jurassic Park was a terrible success, as we know!

```
+-----------------------------------------------------------------+
| Figure 1                                                        |
|                                                                 |
| Example of not dinosaur-portable source program (in an          |
| hypothetical language):                                         |
|                                                                 |
| ABCDEFGHIJKL=0                                                  |
| A0000001=ABCDEFGHIJKL+1 /* ABCDEFGHIJKL is not a portable name */|
|                                                                 |
| This would be preprocessed, after recognition that variable     |
| A0000001 is already conformant to dinosaur-portable restrictions,|
| into:                                                           |
|                                                                 |
| A0000002=1                                                      |
| A0000001=A0000002+1 /* A0000002 is not a portable name */       |
|                                                                 |
|      (btw you will have remarked that the comment is no longer  |
|       true, so is the source program not portable for that? Do  |
|       absolutely have to believe comments)                      |
+-----------------------------------------------------------------+
```

D.2.4 Conclusion

This was just a manifest to help the reflection of those who proclaim that portable programs should not use more than a repertoire of about 40 characters in variable names not to affect portability.

There are good-will solutions to that problem, two of which have been presented here. Instead of proclaiming caveats or of nit-picking proposals which try to advance the technology, if there is any flaw in these two solutions, it would be better to improve them or come with better ones. This should not be too difficult. This took a few minutes of the author's time. A few hours more might be worth it.

E Multiple character sets support

[Editor's note] The issues and proposed solution on multiple character sets support is intended to be discussed here. Danish contribution for this annex is expected.

This annex may contain:

a) Stateful encoding such as ISO 2022

b) Encoding announcement mechanism of ISO/IEC 10646-1

c) Coded character sets announcement mechanism

d) e.t.c.

F Characterless programming

[Editor's note] This annex was removed based on the discussion in the WG20 meeting in U.S.A 1994. The reason of this removal is that the idea presented in this annex is too advanced comparing with other guidelines discussed in this TR.

G Comment instruction

This annex describes kinds of information that should be provided for the comments to this proposed text for the second edition of TR 10176. This annex will not be appeared in the final text of this TR.

At the top of comments to this proposed text, each commenter should provide the following information:

Source: (National body's name, Organization name, or contributor's name)
Status: (National body contribution, Liaison contribution, or an expert contribution)
Date: (yyyy–mm–dd)

At the each comment item, the following information should be provided:

Seq. no. (Source name – sequential number in the comments. e.g. **. JAPAN-1**)
Clause no. & title: (Clause number indicated by k.l.m.n, and its title
e.g. **4.1.3.4 Guideline: Character sets used in comments**)
Line: (Number of lines from the top of the subject clause)
Kind: (The indication of **Major technical**, **Minor technical**, or **Editorial**)
Problem: (Problem description)
Proposed text: (Proposed modification of text or editing instruction)

H Advanced data types for character and string handling

[Editor's note] In this annex, new idea of data types that may be useful for developing internationalized application is presented. Since the idea is quite new, every reviewer of this PDTR is requested to review this annex carefully, and comments to it. The idea may conflict with the other guidelines presented in this TR, and may not be implementable for some programming language standard.

The character data types support should be done at 3 levels: abstract character level, encoded character level, and text level (composite sequences). The sequence of three levels indicates how WG20 sees the importance of the data type support in the programming language design.

A set of APIs needs to be defined to transform between the 3 levels.

H.1 Abstract Character Level

H.1.1 Data type

The programming language standard should facilitate a character data type design that is independent of the actual encoding of characters. This abstract character level should be the main form of the national character data type in programming language as it facilitates the portability among application programs across platforms. This level corresponds to the "character" term in ISO/IEC JTC1/SC2. The specification of the encoding should be hidden and transparent to application programs, thus the encoding is implementation defined. The character is presented in exactly one integral unit, therefore the indexing on the character array is permissible.

H.1.2 Literal resolution

There is a need to have literal in the programming language design. In the scope of the internationalization, the literal design should be on the abstract level, i.e. the abstract character level. For example, a specification of literal `"This is a literal"` should be in abstract data type and the encoding of th literal string should be determined according the runtime codeset not the codeset at the complication time. This may possibly be dependent on the actual language specification to transform such literal strings into the appropriate data type accordingly. For languages not having such specific requirement, the encoded representation can be specified explicitly. For example, in pseudo programming language, (`encoded "UCS-2" "This is a literal"`).

[Editor's note] The "Abstract character data type" described above implies to prohibits to read the character code into the "abstract character data type" as it is where the code in a file system. For example, POUND SIGN may has code point 2/3 in a national version of ISO/IEC 646, but the same character has code point 10/03 in ISO/IEC 8859-1. Since the code point (value) of a character, e.g. POUND SIGN, must be transparent in the "abstract character data type" regardless of encoding scheme of character code in a file system, the code of a character need to be converted to an implementation defined internal value of the "abstract character data type", e.g. ISO/IEC 10646, at the I/O boundary of program and file. In other word, an implementation of a programming language standard can only support coded character sets what code conversion facility of the implementation supports between the code and internal representation of "abstract character data type". Currently, most of implementation of programming languages read the character data in a file system as it is into character data type, therefore the programming language can support any coded character set without code conversion. I mean, the implementation does not care about the representation of character except its character boundary. But the proposal of this "abstract character data type" requires implementation of programming languages to know the encoding scheme. The requirement may be too tough for most of implementation.

H.1.3 Encoded Character Level

The programming language standard may provide a data type to support the encoded character level, where the encoding storage requirement of the abstract characters is known. This level corresponds to the "coded character" term from ISO/IEC JTC1/SC2. This form of encoding can be used to meet the explicit storage requirements, and is useful in programming with multiple coded character sets.

One multi-octet data type is sufficient, where the storage requirement is not determined by the string termination delimiter defined by programming language. The actual data storage can cater for the coded character sets support by the implementation and for a set of ISO character sets based the nature of the coded character set such as single byte single octet (i.e. ISO/IEC 8859-1), single byte multi-octet (i.e. UCS-2 form of ISO/IEC 10646 UCS-2), multi-byte single octet (i.e. UTF-8 of ISO/IEC 10646), and multi-byte multi-octet (i.e. UTF-16 of ISO/IEC 10646), where the encoding and date storage may be defined by POSIX charmap definitions or by other means. Only one set of the multi-octet APIs needs to be defined. For example, a data definition could be done in the following pseudo statement:

```
encoded "UCS-2" character_string = (encoded "UCS-2") "literal string"
```

Here `encoded` is a data type, and `"UCS-2"` is a reference to the actual data storage requirement defined in the charmap or by the standards to be found in the file system or similar places. The list of reserved keywords on the coded character sets which are based on the ISO/IEC JTC1/SC2/WG3 and ISO/IEC JTC1/SC2/WG2 should be provided to ensure the minimum portability among the ISO character sets.

Once the data type with referenced storage requirement (such as `encoded` and `"UCS-2"`) is specified, the programming language standard should provide the necessary code conversion between the machine coded character set and storage coded character set.

[Editor's note] This idea of "encoded" data type intends to guarantee the encoding scheme of characters in the "encoded" data type is the one indicated by the identifier immediate after the "encoded" keyword. It implies that code conversion may be required at I/O boundary between program and file, if required, and assignment or comparison operation between difference encoded data (sub) types. This idea also requests implementation of programming languages to understand encoding scheme of character in both "encoded" data type and the one in a file. Also, when a coded character set standard referred to by the identifier of "encoded" data type has been amended, the distinct new identifier should be assigned for the amended code standard. This requirement may force implementation of programming language standard to chase amendments of code standard.

H.2 Text Level

The programming language standard may provide the data type support for the text level, which corresponds to the "composite sequences" term from SC2. The behavior of this level is currently not so well developed, and it is difficult to advise on required functionality.

A new data type for string handling (compare, sort, search)

In addition to the need to provide containers for character data with different character data types, ISO/IEC 14651 introduces a series of APIs, one of which transforms character data into directly usable numeric data which can be used for direct, parse-free and context-free, numerical comparisons.

So that this data can be reused within an application without constant conversions to occur on the same character string, and exchanged between different applications, an additional container, a new data type, ideally language-independent, is desirable.

Explanation:

Analogy

Machines have to make numerical manipulations, not on the character representation of numbers as entered with different input methods, but on digested numerical data associated with that input. For this, various conceptual containers have been provided by different programming languages for ease of use under the form of as many different data types (such as INTEGER, REAL, and so on).

In the same way, ISO/IEC 14651 introduces APIs which allow comparisons of character data at different levels of precision and equivalences. These various levels are not deduceable by direct operation on character data types but need a transform into these different levels, according to localization parameters. For further manipulation of this data by programming languages, it is desirable to have a new conceptual container, a new data type, which is a structure of the following form:

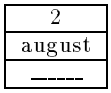
N (number of levels)
numerical string representing equivalence to a string at level 1
numerical string representing equivalence to a string at level 2
numerical string representing equivalence to a string at level 3
. . .
. . .
numerical string representing equivalence to a string at level n

For example, let's say the localization parameters support two levels of equivalence on English words, and we have the word "AUGUST" to manipulate, the new data type will translate this into a structure whose first element says 2 levels, second element is direct numerical equivalence at level 1, and third and last element direct equivalence at level 2:

2
august
******

`where * represent upper case representation`

This will allow comparisons to be made, say with "august", whose occurrence of the same new data type will give:




```
where _ represent upper case representation
```

We can see here that at level 1, both strings are equal (meaning equivalence at level 1). At level 2, the original higher level presentation forms of the character strings, are not equal. The CHARACTER data type does not allow such direct manipulations, not more that one can do additions on character strings "2" and "3" directly. Internationalization of character comparisons require such a new data type to ease manipulations.