A Proposal for a Revision of ISO Modula-2

Benjamin Kowarsch, Modula-2 Software Foundation

May 2020 (arXiv.org preprint) *

Abstract

The Modula-2 language was first specified in [Wir78] by N.Wirth at *ETH* Zürich in 1978 and then revised several times. The last revision [Wir88] was published in 1988. The resulting language reports included ambiguities and lacked a comprehensive standard library.

To resolve the ambiguities and specify a comprehensive standard library an ISO/IEC working group was formed and commenced work in 1987. A base standard was then ratified and published as IS 10514-1 in 1996 [JTC96]. Several standard compliant compilers have since been developed of which at least five remain available and at least three are being actively maintained. Meanwhile, various deficiencies of the standard have become apparent but since its publication, no revision and no maintenance has been carried out.

This paper discusses some of the deficiencies of IS 10514-1 and proposes a limited revision that could be carried out with moderate effort. The scope of the paper has been deliberately limited to the core language of the base standard and therefore excludes the standard library.

1 ISO/IEC Standardisation

[II20] describes ISO/IEC standardisation procedures, summarised in brief below.

1.1 Organisational Overview

The International Organisation for Standardisation (ISO) and the International Electrotechnical Commission (IEC) operate a joint technical committee (JTC1) to develop, publish and maintain information technology standards. JTC1 is organised into subcommittees (SC) and working groups (WG). The subcommittee for programming languages is JTC1/SC22. The working group that developed the ISO Modula-2 standard was JTC1/SC22/WG13. It was disbanded in 2002.

1.2 Working Groups and their Lifecycle

The development and maintenance of ISO/IEC standards is carried out within working groups by technical experts nominated by national standard bodies wishing to participate and having membership in the relevant technical committee or subcommittee.

A working group is established by the relevant committee or subcommittee upon request by a national standard body with voting membership in that committee. The request has to be supported by at least four other national standard bodies with voting membership by declaring their intent to participate in the new working group.

When a working group has completed all its work items, it is either put into stand-by status for future maintenance or disbanded. Once a working group has been disbanded it cannot be re-instated. For any maintenance on the standard a new working group must then be established.

1.3 Standard Development and Review Cycle

Once a working group has been formed, it is required to produce a standard or a revised edition of a standard within no more than 36 months. Once a standard has been published, it is subject to regular reviews every five years and may either be reconfirmed or withdrawn. Although JTC1/SC22 tends to reconfirm standards for which no working group exists to carry out maintenance, this cannot be taken for granted and such standards could in principle be withdrawn for lack of experts to carry out a proper review whenever a regular review is due.

*Parts of this paper are based on an earlier discussion paper by the same author circulated privately in 2015.

2 Practical Considerations

2.1 Challenges

The primary challenge with a revision of IS 10514-1 is not the technical work itself but the establishment of a new ISO/IEC working group, in particular the recruitment of collaborators to carry out the work. There are three detrimental factors.

2.1.1 Negative Perception

One factor is the common perception that participation in standard bodies is both time consuming and frustrating. Standardisation work is often characterised by re-opening of issues that had already been resolved, shifting goals and feature creep. This was certainly the case with JTC1/SC22/WG13. The root causes were collaborator turnover and a general lack of project management, in particular the absence of rigorous definition and guarding of scope.

2.1.2 Time and Cost of Attending Meetings

Another factor is the time and cost of attending face-to-face working group meetings. Such meetings are usually held on a rotational basis and require recurring international travel and hotel accommodation which participants have to cover out of their own pockets unless they participate as part of their employment for and on behalf of a larger corporation.

2.1.3 Membership Fees Charged by National Standard Bodies

Yet another factor are the prohibitively high membership fees most national standard bodies are now charging to individuals willing to participate in standardisation work. In many countries there are arrangements in place that permit individuals who are faculty members of universities to participate in their national standard body without being charged any membership fee. However, in the realm of programming languages, universities are no longer the primary actors. For a few mainstream programming languages, the actors are large corporations. And for the vast majority of programming languages and libraries the actors are small non-profit groups within the open source software movement. The majority of these groups and their collaborators cannot afford the membership fees charged by national standard bodies.

2.2 Constraints

In order to recruit the expert collaborators required by [II20] to form a new ISO/IEC working group, collaborators must be given assurances that (a) the effort they will have to invest is reasonably limited both in terms of total work hours and elapsed calendar time to completion, (b) there will be no travel involved and (c) there will be no monetary cost to them.

2.3 Approach

To meet the aforementioned constraints, we propose the following management approach:

(1) the size of the working group shall be kept small, ideally no more than five or six collaborators

(2) it shall be attempted to negotiate fee waivers for collaborators who are not university staff

(3) all deliberations shall be undertaken via email, groupware and teleconferencing

(4) the scope of work shall be agreed upfront, once agreed no further items shall be added, it shall be strictly limited to 10-12 (target) plus 2-3 (reserve) work items

(5) work items shall be chosen to follow a pareto distribution by estimated work effort, 75-80% shall be lowest to moderate effort while 20-25% may be high effort

(6) any work item on which no consensus can be reached shall be removed from scope

(7) a register of potential future work items shall be maintained for any potential future revision, items removed from scope due to lack of time or consensus may be added to this register

(8) all work shall be completed and a draft revision shall be presented for ballot within one year

3 Methodology

[Kow18] describes design principles for the maintenance of classical Modula-2 compilers and how they should be applied, weighted and prioritised. This paper applies the same considerations to IS 10514-1. For clarity, the relevant section from [Kow18] is reproduced in this section.

3.1 Design Principles

The following design principles strongly influenced the proposed revisions in this paper:

3.1.1 Single Syntax Principle (SSP)

There should be one and ***only one*** syntax form to express any given concept [Dij78].

3.1.2 Literate Syntax Principle (LSP)

Syntax should be chosen for readability and comprehensibility by a human reader [Knu84].

3.1.3 Consistency of Syntax Principle (COSP)

Syntax should be consistent. Analogous concepts should be expressed by analogous syntax.

3.1.4 Principle of Least Astonishment (POLA)

Of any number of possible syntax forms or semantics, the one likely to cause the least astonishment for a human reader should be chosen and the alternatives should be discarded [Geo87].

3.1.5 Single Responsibility Principle (SRP)

Units of decomposition, such as modules, classes, procedures and functions should have a single focus and purpose [Mar09].

3.1.6 Principle of Information Hiding (POIH)

Implementation specific details should always be hidden, public access should be denied [Par72].

3.1.7 Safety Perimeter Principle (SPP)

Facilities that undermine the safety otherwise safeguarded within the language should be segregated from other facilities [Wir88, ch.29]. Their use should require an explicit expression of intent by the author and be syntactically recognisable so as to alert the author, maintainer and reader of the possible implications. This applies and extends the principle of least privilege [Sal74].

3.2 Maintenance Objectives

The primary objectives for the proposed revisions in this paper are:

(1) to remove facilities that are harmful, outdated or violate any of [3.1]

(2) to update IS 10514-1 with essential modern facilities

3.2.1 Weighting and Prioritising Objectives

The objectives given above may from case to case conflict with one another. Which objective should be given preference in the event of a conflict depends on the following factors:

(1) the severity of the *offending facility*

(2) the estimated frequency of use of the *offending facility*

(3) the effort to update sources impacted by change or removal of the *offending facility*

The greater the severity of an *offending facility*, the stronger is the case for ***change*** or ***removal***; the lower the estimated frequency of use and the less the effort to update impacted sources, the stronger the case for ***change*** or ***removal***. In the event that the estimated frequency of use is high and the effort to update impacted sources is significant, ***deprecation*** may be preferable.

3.3 Mitigation Methods

Terms of mitigation methods used in this paper have well defined meanings:

3.3.1 Warning

A warning shall be issued for each and every use of the *offending facility*.

3.3.2 Change

The *offending facility* shall be replaced with a proposed alternative.

3.3.3 Deprecation

A compiler switch to enable and disable the *offending facility* shall be provided and it shall be disabled by default. When it is enabled, a deprecation warning shall be issued for each and every use of the *offending facility*.

3.3.4 Removal

Support for the *offending facility* shall be removed altogether.

From an educational perspective, the availability of *offending facilities* promotes bad habits. ***Replacement*** or ***removal*** is therefore generally preferable to ***warning*** or ***deprecation***.

3.3.5 Transformation

To be used in combination with any of change, deprecation or removal. A conversion program should be provided that transforms source code that uses *offending facilities* into semantically equivalent source code that complies with the proposed revisions in this paper.

4 Lexis

4.1 Lexical Alternatives

IS 5014-1 specifies lexical alternatives `!`, `@`, `<>`, `&` and `~` as synonyms for `|`, `^`, `#`, `AND` and `NOT`.

4.1.1 Revision

Synonym symbols `!`, `@`, `<>`, `&` and `~` shall be ***removed*** from IS 10514-1.

4.1.2 Rationale

The availability of alternative symbols is unnecessary and violates SSP [3.1.1].

Symbols `!` and `@` were not part of the original Modula-2 language, they were added in IS 10514-1 in the mistaken belief that computer systems with character sets that lacked a vertical bar and caret, i.e. mainframe computers with 6-bit character sets would still be in use. In reality 6-bit character sets had already been obsolete for 30 years when IS 10514-1 was published and there had never been any need for these alternatives in Modula-2.

The inequality operator symbol `#` is preferable to its synonym `<>` because it resembles the mathematical inequality symbol ≠ and as a single character symbol it simplifies lexing. Reserved words `AND` and `NOT` are preferable to their respective synonyms `&` and `~` because of consistency: While there are synonyms for `AND` and `NOT`, there is none for `OR`. Although `|` could have been used to denote `OR`, it is already used to separate case labels and would have introduced ambiguity.

4.1.3 Backwards Compatibility

The proposed revision may render existing source code incompatible with the revised standard. However, affected code may be brought into compliance through lexical ***transformation***, using regular expressions and a filter program such as `sed` or `awk` which would be virtually effortless.

4.2 Octal Literals

IS 10514-1 specifies octal literals suffixed with `B` for numeric values and `C` for character values.

4.2.1 Revision

Octal literals shall be ***removed*** from IS 10514-1.

4.2.2 Rationale

The use of octal numbers in programming languages had already been outdated for 20 years when IS 5014-1 was published. Moreover, the `B` and `C` suffixes used to denote octal literals in Modula-2 are also legal digits within hexadecimal literals. This unnecessarily complicates lexing and it violates POLA [3.1.4] as it is confusing to human readers of the source code.

4.2.3 Substitution

The built-in `CHR()` function can be used instead of octal character code literals. It evaluates constant arguments at compile time and accepts both decimal and hexadecimal arguments.

4.2.4 Backwards Compatibility

The proposed revision will likely render existing source code incompatible with the revised standard. However, affected code may be brought into compliance through lexical ***transformation***.

4.3 Set Difference Operator

IS 5014-1 specifies the minus symbol as set difference operator.

4.3.1 Revision

The set difference operator shall be ***changed*** to the backslash symbol.

4.3.2 Rationale

Using the minus symbol for set difference is mathematically incorrect, or at best ambiguous. The proper mathematical symbol for set difference is a reverse solidus, which closely resembles the backslash symbol.

(1) $A \setminus B = \{x \in A \mid x \notin B\}$

(2) $A - B = \{a - b \mid a \in A \wedge b \in B\}$

(3) $A \setminus B \neq A - B$

Furthermore, the use of the minus symbol as set difference operator is a violation of ISO/IEC 80000-2 which states that the minus symbol should not be used as set difference operator [II19].

4.3.3 Backwards Compatibility

The proposed revision will likely render existing source code incompatible with the revised standard. However, affected code may be brought into compliance through lexical ***transformation***.

5 Syntax

5.1 Multi-Dimensional Arrays

IS 10514-1 specifies two alternative syntax forms for multi-dimensional array type declaration.

5.1.1 Revision

The long syntax form for multi-dimensional array declaration shall be ***deprecated***.

5.1.2 Rationale

The short syntax form for multi-dimensional array type declaration

```
TYPE Matrix = ARRAY [0 .. Cols], [0 .. Rows] OF REAL;
```

is an abbreviation of and thus equivalent to

```
TYPE Matrix = ARRAY [0 .. Cols] OF ARRAY [0 .. Rows] OF REAL;
```

The availability of alternative syntax forms is unnecessary and violates SSP [3.1.1] and COSP [3.1.3]. The long form becomes increasingly impractical when declaring array types of more than two or three dimensions. The short form is therefore preferable to the long form.

5.1.3 Backwards Compatibility

The proposed revision does not impact the compatibility of legacy sources.

5.2 Universal Type Conversion Syntax

IS 10514-1 lacks a universal type conversion syntax.

5.2.1 Revision

A universal type conversion operator `::` shall be ***added*** to IS 10514-1. Its precedence shall be one level above that of operator `NOT` and one level below that of parentheses and function calls.

It shall permit safe type conversions (a) between character types, (b) between whole number types, (c) between real number types and (d) between whole and real number types.

5.2.2 Proposed Syntax

```
typeConvExpr :=
  factor '::' typeIdent ;
```

5.2.3 Fallback

Alternatively – failing to reach consensus – a pervasive universal safe type conversion function `CONV()` analogous to function `CAST()` but with the above semantics shall be ***added*** instead.

5.2.4 Rationale

In a type safe language with name equivalence such as Modula-2, the assignment of values between variables of different types is severely restricted by the type regime. It follows that type conversion becomes an important and frequent operation. A universal type conversion facility is preferable over type specific conversion functions as it is consistent and simplifies the core language. It thereby reduces mental load for authors and readers of source code.

5.3 Local Modules

IS 10514-1 permits the lexical nesting of module declarations within the implementation parts of modules. A module declared within another module is known as a local module.

5.3.1 Revision

Local modules shall be ***deprecated***.

5.3.2 Rationale

If there is sufficient reason to delegate certain responsibilities of a library module to a local module, then there is also sufficient reason to delegate those responsibilities to a separate library module. There is no reason why a local module should be chosen over a separate library.

A local module within a program or library module unnecessarily increases the line count of the module and thus reduces its readability and maintainability. It runs counter to the very rationale of decomposing source code into separate modules in the first place.

Moreover, a test environment for testing a local module must necessarily be provided within the hosting module. This further increases clutter and poses the question whether to release the module with or without the embedded test environment. By contrast, a proper library module can be imported into any number of lexically independent test environments.

5.3.3 Substitution

Local modules should be removed from their enclosing module and provided as proper library modules with their own definition and implementation parts. Such library modules may then be marked for private use [5.4], exclusive to the module they were removed from.

5.3.4 Backwards Compatibility

The proposed revision does not impact the compatibility of legacy sources.

5.4 Private-Use Modules

IS 10514-1 lacks a facility to mark a library module for private use.

5.4.1 Revision

A *compiler directive* shall be ***added*** that marks a library module for private use. A warning shall be issued whenever a module so marked is imported into any scope other than that of an implementation part of any of its designated client modules.

5.4.2 Proposed Syntax

The directive shall be recognised after the module header of a definition module.

```
privModPragma :=
  '<*' 'PRIVATETO' '=' clientModuleList '*>' ;

clientModuleList :=
  identList ;
```

5.4.3 Rationale

This facility restores the private-use aspect of local modules which is lost when they are removed from their host modules and converted into library modules, thereby supporting the deprecation of local modules. It is further useful when the *API* of a lower-level library module is considered unstable and subject to frequent changes.

5.5 Foreign Definition Modules

IS 10514-1 lacks a facility to mark a definition module as a *foreign definition module*.

5.5.1 Revision

A *compiler directive* to mark a definition module as a *foreign definition module* shall be ***added***. Implementations that provide means to interface to *foreign APIs* shall implement the directive.

5.5.2 Proposed Syntax

The directive shall be recognised after the module header of a foreign definition module.

```
ffiPragma :=
  '<*' 'FFI' '=' '"' foreignAPI '"' '*>' ;

foreignAPI :=
  'ASM' | 'C' | 'Fortran' | 'Pascal' | ... ;
```

5.5.3 Rationale

A standardised syntax for marking *foreign definition modules* is a requirement for source code compatibility across different implementations.

6 Pervasives

6.1 Large Unsigned Type

IS 10514-1 lacks a large unsigned type.

6.1.1 Revision

A pervasive large unsigned type `LONGCARD` shall be ***added***. The range of the type shall be large enough to express the highest address of the addressable memory of the underlying architecture.

6.1.2 Rationale

For indices of modern filesystems and databases, a large unsigned type is an absolute requirement. Many existing Modula-2 implementations provide a `LONGCARD` type as a language extension.

6.2 Unicode Character Type

IS 10514-1 lacks a dedicated Unicode character type.

6.2.1 Revision

A dedicated pervasive Unicode character type `UNICHAR` shall be ***added***. The type shall be able to represent all Unicode code points and its internal representation shall use UCS-4 encoding.

6.2.2 Rationale

Universal international character sets were in their infancy when IS 10514-1 was developed. Meanwhile use of the Unicode standard [Uni19] has become ubiquitous. Today, most programming languages provide both a conventional 7-bit or 8-bit character type and an extended 16-bit or 32-bit Unicode character type.

6.3 Unicode Character Constructor Function

IS 10514-1 lacks a constructor function for values of type `UNICHAR`.

6.3.1 Revision

A pervasive function `UCHR()` to return a value of type `UNICHAR` for a given code point of type `LONGCARD` shall be ***added***. This is analogous to existing function `CHR()` in support of type `CHAR`.

6.3.2 Rationale

The introduction of type `UNICHAR` necessitates a constructor function in support of the new type.

6.4 Conversion Functions

IS 10514-1 specifies conversion functions `INT()`, `CARD()`, `FLOAT()`, `LFLOAT()`, `TRUNC()` and `VAL()`.

6.4.1 Revision

Pervasive functions `INT()`, `CARD()`, `FLOAT()`, `LFLOAT()`, `TRUNC()` and `VAL()` shall be ***removed***.

6.4.2 Rationale

The universal type conversion facility proposed in [5.2] replaces type specific conversion functions. Moreover, there are serious problems with the aforementioned functions. Their naming and purpose is inconsistent and confusing, violating COSP [3.1.3], POLA [3.1.4] and SRP [3.1.5].

While functions `INT()` and `CARD()` are named for their return types, functions `FLOAT()`, `LFLOAT()`, `TRUNC()` and `VAL()` are not. Functions `FLOAT()` and `LFLOAT()` are confusingly named since their return types are `REAL` and `LONGREAL`. IS 10514-1 does not mandate how the real number types are to be implemented. An implementation may or may not implement them as floating point numbers. The use of identifiers that suggest a floating point implementation is therefore incorrect and misleading.

Function `TRUNC()` represents both a conversion function and mathematical function $trunc(x)$. Its primary purpose is conversion, not truncation. However, its name indicates truncation, not conversion. The name of a function whose purpose is conversion should indicate conversion and the function should ideally perform conversion only.

Conversely, the name of a function whose purpose is truncation should indicate truncation and the function should strictly perform truncation only. Such a function would then be more appropriately provided by a math library. It is worthwhile noting that $trunc(x)$ is a relic of early one's complement hardware, since replaced by $entier(x)$[1] on modern two's complement hardware where truncation is less efficient and leads to the accumulation of rounding errors [Tuc04]. Thus, if any pervasive function of this kind is desired, it should be `ENTIER()` but not `TRUNC()`.

Finally, function `VAL()` has the worst possible naming of all. Instead of indicating conversion in its name, it indicates that it returns a value. However, this is is entirely meaningless since all functions return values. The name carries no information about the function's purpose at all.

6.4.3 Excluded Functions

It should be noted that we do not propose to remove functions `CHR()` and `ORD()` because they are are not conversion functions. Character types do not represent numeric values and consequently there is no conversion between character types and numeric types. Instead, `CHR()` is a lookup function that looks up a character by numeric index. Function `ORD()` performs a reverse lookup.

6.4.4 Backwards Compatibility

The proposed revision will likely render existing source code incompatible with the revised standard. Affected code may be brought into compliance through syntactical ***transformation***.

7 Semantics

7.1 Compatibility of NIL

`NIL` is not compatible with opaque pointer types and procedure types.

7.1.1 Revision

`NIL` shall be compatible with opaque pointer types and procedure types.

7.1.2 Rationale

The definition of type specific nil values for every opaque pointer type and procedure type unnecessarily increases code clutter and mental load. No type safety is gained by this restriction.

7.1.3 Backwards Compatibility

The proposed revision does not impact the compatibility of legacy sources.

[1] $entier(x)$ rounds towards $-\infty$ while $trunc(x)$ rounds towards 0.

7.2 Casting Constant Values

Function `CAST()` does not accept constant arguments.

7.2.1 Revision

Function `CAST()` shall accept constant arguments.

7.2.2 Rationale

The definition of intermediate variables and assignment of constant values for the sole purpose of casting the values unnecessarily increases code clutter and mental load. No type safety is gained.

7.2.3 Backwards Compatibility

The proposed revision does not impact the compatibility of legacy sources.

7.3 Pointer Initialisation

IS 10514-1 does not require pointer variables to be initialised.

7.3.1 Revision

All pointer variables shall be initialised to `NIL` and deallocation should reset them to `NIL`.

7.3.2 Rationale

The two primary paradigms of Modula-2 are (a) program decomposition through data encapsulation and information hiding, and (b) reliability through type safety. Opaque pointer types are the primary instrument through which the former is achieved. The ability to test whether an opaque pointer type has been allocated or deallocated is central to achieving the latter.

7.3.3 Backwards Compatibility

The proposed mitigation does not impact the compatibility of legacy sources.

7.4 Write Access to Imported Variables

IS 10514-1 permits write access to imported variables.

7.4.1 Revision

Write access to imported variables shall be ***deprecated***.

7.4.2 Rationale

Write access to to imported variables violates POIH [3.1.6]. The original Modula-2 language report [Wir88, p.88] states that “imported variables should be treated as ‘read-only’ objects”.

7.4.3 Backwards Compatibility

The proposed mitigation does not impact the compatibility of legacy sources.

7.5 Shifting Non-Bitset Values

Function `SHIFT()` is restricted to operate on values of `BITSET` types.

7.5.1 Revision

Function `SHIFT()` shall operate on values of any 8-, 16-, 32- and, if available 64-bit type.

7.5.2 Rationale

The restriction is entirely impractical because it is most likely to require casting from the original type to a `BITSET` type before the shift operation and then casting the result back to the original type afterwards. This unnecessarily increases clutter and makes the source code difficult to read and maintain. Let us consider the example of a real world hash function below:

```
PROCEDURE valueForNextChar ( hash : Key; ch : CHAR ) : Key;
BEGIN
    RETURN CAST(Key, ORD(ch)) +
           CAST(Key, SHIFT(CAST(BITSET32, (hash)), 6)) +
           CAST(Key, SHIFT(CAST(BITSET32, (hash)), 16)) - hash;
END valueForNextChar;
```

This function body would be far more readable without the casting:

```
RETURN ORD(ch) + SHIFT(hash, 6) + SHIFT(hash, 16) - hash;
```

The `SHIFT()` function has been placed in pseudo-module `SYSTEM` for a reason: Facilities provided by `SYSTEM` are by their very definition unsafe. Any import and use of function `SHIFT()` thereby constitutes a deliberate decision by the developer to break type safety. No type safety is gained by restricting the argument types for what is already an unsafe operation.

7.5.3 Backwards Compatibility

The proposed revision does not impact the compatibility of legacy sources.

7.6 Replacing Variant Records with Extensible Records

IS 10514-1 specifies records with variant fields, also known as variant records.

7.6.1 Revision

Variant record types shall be ***removed*** from IS 10514-1 and ***replaced*** with type safe extensible record types described in [Wir90]. Slightly deviating from [Wir90], a record type shall only be extensible if its declaration includes a base type specifier. The declaration of an extensible record type that does not extend another type shall specify `NIL` as its base type.

Also deviating from [Wir90], there shall be no special type guard syntax. Instead, the `CASE` statement alone shall be used to implement type guards.

```
CASE record OF
| TypeA : (* access to fields unique to TypeA *)
| TypeB : (* access to fields unique to TypeB *)
(* etc *)
```

7.6.2 Proposed Syntax

```
recordType :=
  TYPE Ident '=' RECORD ( '(' baseType ')' )? fieldListSeq END ;

baseType :=
  'NIL' | typeIdent ;
```

7.6.3 Rationale

Variant records permit writing a variant field as one variant and then reading it back as another. This constitutes a hidden type cast and is thus an unsafe operation. The availability of unsafe operations without prior import from pseudo-module `SYSTEM` is a violation of SPP [3.1.7].

Furthermore, variant record types are fragile. Whenever another variant is added to a variant field list, all existing client modules have to be recompiled. Variant records are therefore not an adequate facility for software extensibility.

By contrast, extensible record types are both type safe and non-fragile.

7.6.4 Backwards Compatibility

The proposed revision will likely render existing source code incompatible with the revised standard. Affected code may be brought into compliance through syntactical ***transformation***.

Definitions

API

Application programming interface.

compiler directive

A directive within the source to instruct a language processor how to process the input.

ETH

Eidgenössisch Technische Hochschule – Swiss Federal Institute of Technology.

foreign API

An API implemented in a language other than Modula-2.

foreign definition module

A definition module that specifies an interface to a *foreign API*.

offending facility

A language facility that is (a) outdated, harmful or bad habit forming in general, or (b) violates any of the design principles in section 3.1 in particular.

References

[Par72] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comunications of the ACM*, 15(12), 1972.

[Sal74] Jerome H. Saltzer. The Protection of Information in Computer Systems. *Comunications of the ACM*, 17(7), 1974.

[Dij78] Edsger W. Dijkstra. *On the GREEN Language submitted to the DoD.* E.W.Dijkstra Archive, originally circulated privately, 1978.

[Wir78] Niklaus Wirth. *Modula-2.* Technical report, ETH Zürich, 1978.

[Knu84] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2), 1984.

[Geo87] James Geoffrey. *The Tao of Programming.* InfoBooks, 1987.

[Wir88] Niklaus Wirth. *Programming in Modula-2.* Springer, Heidelberg, 4th edition, 1988.

[Wir90] Niklaus Wirth. *Oberon Language Report.* Technical report, ETH Zürich, 1990.

[JTC96] ISO/IEC JTC1/SC22/WG13. Information Technology – Programming Languages – Part 1: Modula-2, Base Language. *International Standard 10514-1*, ISO/IEC, 1996.

[Tuc04] Allan B. Tucker. *Computer Science Handbook.* Chapman & Hall, 2nd Edition, 2004.

[Mar09] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2009.

[Kow18] Benjamin Kowarsch. *On the Maintenance of Classic Modula-2 Compilers.* Technical report, Modula-2 Software Foundation, 2018.

[II19] ISO/IEC. Quantites and units – Part 2: Mathematics. *International Standard ISO/IEC 80000-2,* ISO/IEC, 2019.

[Uni19] The Unicode Consortium. *The Unicode Standard.* ISBN 978-1-936213-25-2. Unicode Consortium, Version 12.1, 2019.

[II20] ISO/IEC. *ISO/IEC Directives.* ISO/IEC, 16th edition, 2020.