On the Maintenance of Classic Modula-2 Compilers

Benjamin Kowarsch, Modula-2 Software Foundation

September 2018 (Final Draft)

Abstract

The classic Modula-2 language was specified in [Wir78] by N.Wirth at *ETH* Zürich in 1978. The last revision [Wir88] was published in 1988. Many computer science books of that era used Modula-2 in programming examples. Many of these are still valuable resources in computer science education today. To compile and run the examples therein, it is essential to have compilers available that follow the classic Modula-2 language definition and run on modern computer hardware and operating systems. Although most Modula-2 compilers of that era have disappeared, a few have since been re-released under open source licenses. Whilst the original authors have long ceased work on these compilers, new maintainers have stepped in. This paper gives recommendations for maintenance on classic Modula-2 compilers while balancing the aim to modernise with the need to maintain the capability to compile programming examples in the literature with minimal effort. Nevertheless, the principles, methods and conclusions presented are adaptable to maintenance on other languages.

1 Methodology

1.1 Design Principles

The following design principles strongly influenced the recommendations in this paper:

1.1.1 Single Syntax Principle (SSP)

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

1.1.2 Literate Syntax Principle (LSP)

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

1.1.3 Consistency of Syntax Principle (COSP)

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

1.1.4 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].

1.1.5 Single Responsibility Principle (SRP)

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

1.1.6 Principle of Information Hiding (POIH)

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

1.1.7 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].

1.2 Maintenance Objectives

The primary objectives for the recommendations in this paper are:

(1) to remove facilities that are harmful, outdated or violate any of [1.1]

(2) to resolve ambiguities in [Wir88] and incompatibilities between implementations

(3) to maintain the capability to compile programming examples in the literature with minimal effort

1.2.1 Weighing Objectives by Impact

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*** is recommended.

1.3 Mitigation Methods

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

1.3.1 Warning

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

1.3.2 Change

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

1.3.3 Deprecation

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

1.3.4 Removal

Support for the *offending facility* should 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.

1.3.5 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 recommendations given in this paper.

2 Lexis

2.1 Octal Literals

Support for octal literals should be ***removed***.

2.1.1 Rationale

The use of octal numbers has long been outdated. Further, the `B` and `C` suffixes used to denote octal literals are also legal digits within hexadecimal literals, which is confusing to human readers of the source code as it violates POLA [1.1.4] and it unnecessarily complicates lexing.

2.1.2 Substitution

The built-in `CHR()` function can be used instead without penalty as it is evaluated at compile time for constant arguments. The function accepts decimal and hexadecimal arguments. Code that uses the `CHR()` function instead of octal literals can always be compiled on any classic Modula-2 compiler, regardless of whether octal literals are recognised or not.

2.1.3 Backwards Compatibility

Preferably, ***transformation*** should be used to replace all occurences of octal literals in existing Modula-2 sources. Alternatively, octal literals could be ***deprecated*** instead.

2.2 Synonym Symbols

Support for synonym symbols `<>`, `&` and `~` should be ***removed***.

2.2.1 Rationale

The availability of alternative symbols violates SSP (1.1.1) and they are are notably ***absent*** from the grammar in [Wir88, pp.156-157].

The inequality operator symbol `#` is preferable to its synonym `<>` because it resembles the mathematical inequality symbol $\neq$ 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 caused ambiguity.

2.2.2 Substitution

Symbol `#` can be used in place of `<>`, while `AND` can be used in place of `&` and `NOT` in place of `~`.

2.2.3 Backwards Compatibility

Preferably, ***transformation*** should be used to replace all occurences of synonym symbols in existing Modula-2 sources. Alternatively, synonym symbols could be ***deprecated*** instead.

2.3 Non-Semantic Compiler Directives

Whilst *compiler directives* are implementation defined, the delimiters of *non-semantic compiler directives* should not be. They should be denoted by an opening `(*$` and a closing `*)` delimiter.

2.3.1 Rationale

Although [Wir88, p.18] mentions `(*` and `*)` as delimiters for both comments and directives, it does not specify their use in any normative manner. Some compiler implementors have taken this to mean that the delimiters are implementation defined. Several compilers still in use today do not follow the convention, for example the venerable MOCKA compiler [EV94].

As a result, source code with *compiler directives* is often not portable across different implementations. This is unfortunate, because non-semantic directives can safely be ignored in the event that they are not supported. Even where an implementation issues warnings about unrecognised directives, to be able to identify a directive as such in the first place, a common delimiter convention needs to be followed across implementations.

2.3.2 Backwards Compatibility

Non-compliant directives within existing source code should be replaceable with minimal effort using regular expressions and a filter program such as `sed` or `awk`.

2.4 Semantic Compiler Directives

Semantic compiler directives should ***not*** be denoted by the same delimiters used for *non-semantic compiler directives*.

2.4.1 Rationale

Whilst *non-semantic compiler directives* can safely be ignored, *semantic compiler directives* cannot. A *semantic compiler directive* that is not supported by an implementation must be reported as an error. Consequently, an implementation needs to be able to distinguish between semantic and non-semantic directives. In order to do so, different delimiters must be used.

2.4.2 Substitution

Some Modula-2 compilers have used a `%` prefix to denote *compiler directives*, in particular for conditional compilation, for example the MOCKA compiler [EV94]. We recommend to follow that convention to denote *non-semantic compiler directives* if any are provided. The `%` symbol is not otherwise used in the language.

2.4.3 Backwards Compatibility

Non-compliant directives within existing source code should be replaceable with minimal effort using regular expressions and a filter program such as `sed` or `awk`.

3 Syntax

3.1 Multi-Dimensional Arrays

Inconsistent definition or declaration of multi-dimensional arrays should trigger ***warnings***.

3.1.1 Rationale

[Wir88, p.138] specifies alternative syntax forms 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 violates SSP [1.1.1] and COSP [1.1.3]. The more dimensions an array type has, the more preferable the abbreviated form becomes. However, it is more effort to ***remove*** or ***deprecate*** the long form. The impact of ***removal*** or ***deprecation*** on existing sources with multi-dimensional arrays would likely be high. It is less effort and sufficient to modify an implementation to detect and warn about mixed use in any given compilation unit.

3.1.2 Backwards Compatibility

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

3.2 Local Modules

Local Modules should be ***deprecated***.

3.2.1 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.

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

Finally, 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.

3.2.2 Substitution

Local modules should be removed from their enclosing module and provided as proper library modules with their own definition and implementation parts.

3.2.3 Backwards Compatibility

Backwards compatibility can be provided via compiler switch.

3.2.4 Private-Use Aspect

The private-use aspect of local modules is lost when they are removed from their host modules and converted into library modules. However, this aspect is useful when the local module's API is considered unstable and subject to frequent changes. A satisfactory solution can be implemented easily by introducing a *non-semantic compiler directive* that specifies a library's intended client modules and causes the compiler to issue warnings whenever such a library is imported by a module other than its designated client modules. An example is given below:

```
DEFINITION MODULE PrivateLib; (*$CLIENTS=FooLib, BarLib, BazLib*)
(* ATTENTION !!! This module is intended for private use only.
 * Its API is subject to frequent change without prior notice.
 * The compiler will issue a warning when it is imported from
 * any other than the designated client modules. *)
 ...
END PrivateLib.
```

3.3 Unary Minus

Unary minus should only be permitted before a factor.

```
simpleExpression :=
  ( '+' )? term ( addOp term )* | '-' factor
  ;
```

instead of

```
simpleExpression :=
  ( '+' | '-' )? term ( addOp term )*
  ;
```

3.3.1 Rationale

[Wir88] does not state the scope of the unary minus operator other than through the grammar.

An expression of the form

`– a * b + c`

could be interpreted ***mathematically*** correct

`(–a) * b + c`

or ***grammatically*** correct

`–(a * b + c)`

The former interpretation conforms to mathematical convention. The latter can be deduced from the grammar in [Wir88, pp.156-157] and is intended to be the correct interpretation[1].

[1] The author obtained clarification on the precedence of unary minus from Prof. Wirth by email.

However, this violates POLA [1.1.4] and some implementations therefore follow the mathematically correct interpretation, for example the ACK compiler [Jac85] and the MOCKA compiler [EV94]. Requiring a factor after a unary minus forces the use of parentheses whenever there is more than one factor to follow which makes programmer intent explicit.

3.3.2 Backwards Compatibility

Unary minus is an infrequently used operation. When it is used, it is usually in a form that is compliant with the syntax proposed in this paper. In the unlikely event that it is not in a compliant form, the source code is ambiguous and needed to be fixed anyway. It may well have been written for and tested with a compiler that uses a different interpretation than the compiler now used to compile the sources. The proposed mitigation will thereby help identify any non-compliant occurrences which can then be corrected with minimal effort.

4 Pervasives

4.1 Conversion

Pervasive function `VAL()` should be ***changed*** to cover all numeric conversions and numeric conversions only. Pervasive functions `FLOAT()` and `TRUNC()` should then be ***deprecated***. Any additional non-standard conversion functions such as `INT()`, `CARD()` and `LFLOAT()` should be ***removed***.

4.1.1 Rationale

[Wir88, p.150] specifies pevasive conversion function `VAL()` which covers all possible use cases for safe type conversion between whole number types. There is no reason why `VAL()` could not also cover all possible use cases for conversions between whole and real number types. By contrast, `CHR()` and `ORD()` are not conversion functions[2] and they should not be duplicated by `VAL()`. Duplication violates SSP [1.1.1].

Furthermore, function `FLOAT()` is confusingly named since the type it converts to is `REAL` and there is no type `FLOAT`. [Wir88] does not specify how real number types are to be implemented. They need not be implemented as floating point numbers but could be implemented as binary coded decimals, for instance. The naming of `FLOAT()` is thus misleading and violates COSP [1.1.3] and POLA [1.1.4].

Finally, function `TRUNC()` violates SRP [1.1.5] as it represents both a conversion function and the mathematical function $trunc(x)$. Its name is misleading as it does not suggest conversion, violating POLA [1.1.4]. Conversion should be the responsibility of function `VAL()` and truncation the responsibility of a math function `trunc()` to be provided in a library such as `MathLib` and return the same type as its argument, it should not perform any type conversion.

4.1.2 Substitution

Once function `VAL()` has been extended to provide type conversion between whole and real number types, any invocation of functions `FLOAT()` and `TRUNC()` within existing source code may be safely replaced with an invocation of `VAL()`.

4.1.3 Backwards Compatibility

Backwards compatibility can be provided via compiler switch.

4.1.4 Implementation

It should be noted that function `VAL()` is not generally implemented as a single function, nor should it be. Instead, it is usually and preferably implemented as a built-in macro, resolved at compile time to a function internal to the compiler that is specific to the source and target types of its arguments, since the types are known at compile time. `VAL()` thereby simplifies the user interface without incurring the penalty of increasing the complexity of its implementation. The specific conversion functions need to be implemented within the compiler anyway, but they should not be individually exposed in the user interface.

[2] `CHR()` performs a lookup while `ORD()` returns a property.

5 Semantics

5.1 Exported Variables

Variables defined in definition parts should be exported ***read-only***.

5.1.1 Rationale

Allowing a client module to write to imported variables violates POIH [1.1.6]. And indeed, [Wir88, p.88] explicitly states that "imported variables should be treated as 'read-only' objects".

5.1.2 Substitution

A setter procedure can be defined and exported by the same module if needed.

```
DEFINITION MODULE Foo;

VAR bar : Bar; (* read-only *)

PROCEDURE SetBar ( value : Bar );

END Foo.
```

5.1.3 Backwards Compatibility

Considering that Wirth explicitly stated to treat imported variables as "read-only" objects, noone should have written any code that treats them as mutable objects. Unfortunately, there will be code written by careless programmers who didn't follow his advice. The cautious maintainer may therefore consider providing a compiler switch to enable and disable write-access to imported variables, which should then be disabled by default.

5.2 Pointer Variables

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

5.2.1 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.

5.2.2 Backwards Compatibility

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

5.3 Unsafe Facilities

Any facility that bypasses the type safety provided by the language should be disabled by default.

5.3.1 Unsafe Facilities provided by SYSTEM

[Wir88, p.121] defines pseudo-module `SYSTEM` as a container for unsafe facilities. The facilities provided therein are only enabled by import. This is desirable as it sensitises programmers to the fact that the facilities are unsafe and thereby discourages their use.

5.3.2 Unsafe Facilities NOT provided by SYSTEM

Unfortunately, not all unsafe facilities are provided through pseudo-module `SYSTEM`. Two unsafe facilities are provided in the language core and are enabled by default without import:

(1) Unsafe type transfers [Wir88, p.119], also known as ***type casts***

(2) Records with variant parts [Wir88, p.71, p.138], also known as ***variant records***

This is a violation of SPP [1.1.7] and given its implications for program safety it is unacceptable. In order to mitigate this situation, support for type casts and variant records should be disabled by default and require enabling by compiler switch.

5.3.3 Ideal World Scenario

In an ideal world, the type cast syntax would be replaced with a `CAST()` function provided by `SYSTEM` as in ISO Modula-2 [JTC96], and variant records with type safe extensible records as in Oberon [Wir90]. However, this would constitute a substantial language revision and as such go beyond the scope of the maintenance aspect of this paper. It would also run counter to the objective to allow the compilation of programming examples in the literature with minimal effort.

6 Language Extensions

6.1 Availability

All language extensions should be disabled by default and only enabled by compiler switch.

6.1.1 Rationale

Disabling language extensions by default aids and promotes writing of portable source code.

6.2 Smallest Addressable Unit

Under no circumstances should any implementation change the definition of type `SYSTEM.WORD`. However, an implementation targeting an architecture where the smallest addressable storage unit is eight bits wide should provide an alias type `BYTE` in module `SYSTEM` as follows:

```
TYPE BYTE = WORD;
```

6.2.1 Rationale

[Wir88, p.153] specifies `SYSTEM.WORD` as the smallest addressable unit[3]. Whilst the report permits provision of ***additional*** facilities in `SYSTEM`, it does not permit ***alterations*** of `SYSTEM` facilities specified in the report.

It would thus be permissible to provide an additional type `SYSTEM.MACHINEWORD` that represents a machine word larger than the smallest addressable unit, but it is not permissible to change `SYSTEM.WORD` to represent a machine word that is not the smallest addressable unit.

6.3 Foreign Identifiers

An implementation that provides a means to interface to *foreign APIs*, should also allow the use of *foreign identifiers*. Such an identifier may contain one or more dollar `$` and/or lowline `_` characters. The use of *foreign identifiers* for other purposes should be discouraged. Support should be disabled by default and enabled by compiler switch when needed.

To avoid any collission with name mangled symbols generated by the Modula-2 compiler, no consecutive and no trailing dollar signs should be permitted; no leading, no consecutive and no trailing lowlines should be permitted. Further, dollar signs and lowlines should not be permitted within module identifiers.

[3] When the first report on Modula-2 was written it was still common to call the smallest addressable storage unit a ***word***. Since then the terminology has changed and it has become common to use the term ***byte*** instead.

6.3.1 Rationale

Operating system *APIs* and other *foreign APIs* often include variables and procedures with identifiers that include dollar `$` (predominantly on the VMS operating system) and lowline `_` (predominantly on Unix systems and C *APIs* in general).

6.4 Foreign Definition Modules

An implementation that provides a means to interface to *foreign APIs*, should use a *non-semantic compiler directive* to mark a definition module as a *foreign definition module*.

6.4.1 Rationale

[Wir88] does not mention *foreign definition modules*. As a result, implementors have invented their own syntax to mark a definition module as foreign, and this varies between implementations.

However, the corresponding implementation could in principle be done in Modula-2. The marking of a definition module as foreign does not alter its semantics. Consequently, any such marking constitutes de-facto a *non-semantic compiler directive*.

6.4.2 Proposed Syntax

The directive should be placed after the module header, its proposed syntax is as follows:

```
ffiPragma :=
  '(*$' ffiPragmaKey '=' '"' foreignAPI '"' '*)'
  ;

ffiPragmaKey :=
  'F' | /* if implementation uses single-letter keys */
  'FFI' /* if implementation uses multi-letter keys */
  ;

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

7 Miscellaneous

7.1 Filename Suffixes

Modula-2 implementations should ***only*** recognise input files with suffixes `def` and `mod`.

7.1.1 Rationale

Neither [Wir78] nor [Wir88] mention filename suffixes for Modula-2 source files. A de-facto standard has been established by the compilers distributed by *ETH* Zürich: Suffix `def` is used for definition module files and suffix `mod` for implementation and program module files.

The absence of any mentioning of filename suffixes in [Wir78] and [Wir88] has been taken by some implementors as an invitation to define their own non-standard filename suffixes.

The use of non-standard suffixes leads to unnecessary effort when compiling sources across different implementations. Moreover, it has led to conflicts with other notations in and inconsistent support for Modula-2 by various text editors and source code renderers.

7.2 Pre-Revision Compilers

An implementation that follows the original monograph [Wir78] or the second edition [Wir83] should be updated to follow the third [Wir85] or, preferably the fourth [Wir88] edition.

7.2.1 Rationale

Prior to [Wir85] a definition module was required to have an export list. This was unnecessary duplication since the purpose of a definition module is to export all its contents.

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*.

foreign identifier
An identifier of a foreign API.

non-semantic compiler directive
A compiler directive that does not alter the semantics of the source code.

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 1.1 in particular.

semantic compiler directive
A compiler directive that alters the semantics of the source code.

References

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

[EV94] H. Emmelmann and J. Vollmer. *GMD Modula-2 System MOCKA User Manual*, 1994.

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

[Jac85] Ceriel J.H. Jacobs. *The ACK Modula-2 Compiler*, 1985.

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

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

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

[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.

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

[Wir83] Niklaus Wirth. *Programming in Modula-2*. Springer, Heidelberg, 2nd edition, 1983.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Springer, Heidelberg, 3rd edition, 1985.

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

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