M Modula-2 interface

MODULA-2 . . . AN IMPROVEMENT ON PASCAL AND MODULA
— NIKLAUS WIRTH

Modula-2 is a middle member of a family of similar programming languages designed by the late Swiss computer scientist, Niklaus Wirth[1] (15 February 1934–1 January 2024), who was honored with the ACM Turing Award in 1984 for his influential works.[2]

Five years after his first book about the Pascal language, which we treat in Appendix P on page 989, Niklaus Wirth introduced a new language, Modula, in two journal articles in 1977.[3] He developed Modula to address some of the, by then, well-understood shortcomings of his Pascal language, but also to reduce the language size, and importantly, to introduce features for *multiprogramming*: subprocesses running in parallel within a single address space that can communicate via signals, as well as by the usual function and procedure calls. The design of Modula was influenced by the important earlier work on *Concurrent Pascal*, and the SOLO operating system, both created by the prominent Danish–American computer scientist, Per Brinch Hanson[4] (13 November 1938–31 July 2007), starting about 1975.

This appendix would have appeared in this book at the location suggested by its page numbers, but was necessarily written after book publication, when a portable compiler for Modula-2 made the language more widely available.

M.1 Modula history

Modula preserves most of the syntax of Pascal, making it familiar to programmers, and introduces the ideas of code *modules*, each of which is implemented in two separate parts: an *implementation* of constants, data, functions, and types, normally stored in a *private* file with extension `.mod`, and an *interface definition*, stored in a *public* file with extension `.def`. The two-part separation of library code allows the programmer to create a stable interface for user programs, while being able to alter internal details of the implementation, *without requiring changes to user code*. The public part is normally stored in a collection of such things in a filesystem location known to the

[1] See `https://www.math.utah.edu/pub/bibnet/authors/w/`.

[2] See `https://amturing.acm.org/award_winners/wirth_1025774.cfm`.

[3] *Modula: a Language for Modular Multiprogramming*, Software—Practice and Experience, **7**(1) 3–35 January/February (1977), doi:`10.1002/spe.4380070102`; *Design and Implementation of Modula*, Software—Practice and Experience, **7**(1) 67–84 January/February (1977), doi:`10.1002/spe.4380070104`.

[4] See `https://www.math.utah.edu/pub/bibnet/authors/h/`.

compiler, so that no special action is needed by the programmer to compile code that uses the module interface definition. Depending on the compiler, the `.def` file need not be compiled, but instead can be read directly by the compiler, much like C/C++ header files named in `#include` preprocessor directives. Some Modula-2 compilers convert `.def` files to faster loading equivalents, named with an extension such as `.sy`.

Among the Pascal features eliminated in Modula are the *infamous* `goto` *statement*, the `SET` *type*, and *input/output (I/O) statements*. In addition, conditional and loop statements are simplified, with minor syntax changes. I/O support is instead supplied by library modules that can be adapted for the needs of particular environments and operating systems.

The first compiler for Modula was written in Pascal, running on the same CDC 6400 where Pascal was originally developed. That allowed quick experimentation with proposed new language features.

Other systems that already had a Pascal compiler could soon have a working Modula compiler, provided that the low-level signal-handler support for multiprogramming could be supplied.

Five years of experience with Modula led to further language changes, introduced in a new book, *Programming in Modula-2*, that eventually appeared in four editions.[5]

Because Niklaus Wirth chose not to document it in a book, the original Modula language may have been used primarily at his academic institution, the *Eidgenössische Technische Hochschule* in Zürich, Switzerland, widely known by its initials, *ETH*, or in English, the *Swiss Federal Institute of Technology*.

The four editions of *Programming in Modula-2* indicate a much stronger interest in the language, and some computer science departments adopted it as their introductory programming language because it is clean, safe, and much simpler than major internationally standardized languages, such as Ada (1983), C (1972), C++ (1985), C# (2003), and the popular, but not standardized, Java (1991).

The ISO/IEC 10514-1:1996 Standard for the base language, and 10514-2:1998 for generics, recognize the importance of Modula-2, and make it easier for industry to adopt it for software development, because it now has a reliable definition that is independent of any particular compiler implementation.

Formalization of the language specification is not without problems, as recorded in *Recollections about the development of Pascal*, from a 1993 conference exchange with Niklaus Wirth:

> **Q**: "The current ISO draft of Modula-2 contains a full formal semantics. Do you consider this useful?"
> **A**: "Not really. It makes the report too voluminous for the reader. The Modula-2 report is 45 pages and the formal standardization document is about one thousand pages long. The definition of the empty statement alone takes a full page. The difference is intolerable."

As might be expected, Modula-2 underwent further major changes. Beginning in 1986, yet another language descendant, Modula-3, was produced by a multi-industry team outside ETH.[6] It adds support for *exception handling*, *garbage collection*, *generic programming* (similar to C++ *templates*), *marking of unsafe code*, *multithreading*, and *object-oriented programming*. However, despite the creation of several commercial compilers for Modula-3, the language seems not to have been widely used in industry, and compilers for it are now difficult to find.

Niklaus Wirth approved of the Modula-3 effort, but focused his own academic research on the last general-purpose programming language that he created: Oberon, first described in 1988.[7] We treat that language in Appendix O on page 989.

Some industrial projects continue to use Modula-2, and the lack of widely available and reasonably portable compilers for it on modern systems led to incorporation of support in the popular GNU `gcc` compiler family. The first stable release of `gm2`[8] was in `gcc-12` in 2021, and the compiler can also be built with later major versions. Some,

[5] Published by Springer-Verlag. First edition 1982: ISBN 0-387-11674-5, doi:10.1007/978-3-642-96717-7. Second edition 1983: ISBN 0-387-12206-0, doi:10.1007/978-3-642-96757-3. Third edition 1985: ISBN 0-387-15078-1, doi:10.1007/978-3-642-96878-5. Fourth edition 1988: ISBN 0-387-50150-9, doi:10.1007/978-3-642-83565-0. The last is available in HTML form at http://freepages.modula2.org/report4/modula-2.html.

[6] See https://en.wikipedia.org/wiki/Modula-3 and https://www.modula3.org/. Additional documentation is in the *Modula-3 Report* at https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-52.html and in *Some Useful Modula-3 Interfaces* at https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-113.html.

[7] *From Modula to Oberon*, Software—Practice and Experience, **18**(7) 661–670 July (1988), doi:10.1002/spe.4380180706. *The Programming Language Oberon*, Software—Practice and Experience, **18**(7) 671–690 July (1988), doi:10.1002/spe.4380180707. *Programming in Oberon: Steps beyond Pascal and Modula*, ACM Press (1992), ISBN 0-201-56543-9.

[8] See https://gcc.gnu.org/onlinedocs/gm2.pdf.

but not yet all, operating system distributions that include the `gcc` compilers make `gm2` available as a separate package. Otherwise, it can be readily built from source code, as this author has done many times.

There is an ongoing effort with the LLVM compiler project to produce `m2lang`,[9] as a companion to `clang`, `clang++`, and `flang`, but in 2024, it is not yet in operating-system package distributions.

The `gm2` compiler accepts command-line options to enable features from the four editions of Wirth's books on Modula-2, as well as the ISO Standard. A large portion of the compiler itself is written in Modula-2, and a bootstrap compiler written in C and C++ translates that Modula-2 code to C++, which is then compiled to produce a native Modula-2 compiler. This complexity is entirely hidden from the programmer, who can just use the traditional recipe `gm2 hello.mod && ./a.out` to compile and run programs, as with other languages supported by the `gcc` family.

Pascal was designed to be *small* (as little as 8KB of memory for some compilers on microcomputers), and *fast to compile*. The language demanded that the entire program be present in a single source file, and that everything used must first be defined, forcing the `main()` program to appear at the end of the source file. Pascal programmers have to learn to read their code in the bottom-up direction. Both were important when computers were much slower than modern ones, and had limited, and expensive, memory. However, it meant that large software projects often simply could not be implemented in Pascal, unless nonstandard and nonportable extensions in some compilers for separate compilation and linking were employed.

By contrast, Modula allows, and encourages, separate compilation, and has more flexible code placement, such as alphabetical ordering of functions and procedures, so two or more passes are needed: the `gm2` bootstrap compiler has five passes, and the first Modula-2 compiler at ETH has seven, later reduced to five. On older systems with limited memory, each pass would often require moving data between memory and physical storage devices. Modern systems operate with much more memory, so the passes can generally be done without external storage access.

With the approval of Niklaus Wirth, a significant revision has been defined for release R10 of Modula-2, but `gm2` does not yet support that dialect. The frequently asked questions (FAQ) page at the project Web site[10] describes the extensions, and the reasons for basing the work on Modula-2, rather than on Modula-3 or Oberon. `gm2` already has additional support for a `VOLATILE` type qualifier, an interface to inline assembly code via an `ASM()` extension, and a `__BUILTIN__` attribute to allow convenient machine-independent access to features that might be provided by native hardware on some systems, but by software library functions on others, just as can be done with extended compilers for C and C++. Thus, specialized systems programming, device drivers, kernel modules, and so on can be written in Modula-2 instead of in the less safe, and more complicated, C and C++ languages.

Translators from Modula-2 to C, C++, C#, and Java have been reported on the Internet, and might still be found at their development sites.

Some compilers for the Modula and Oberon family produce C code as a high-level replacement for native assembly code. As long as the compilers themselves are programmed portably, they can be quickly implemented on many different operating systems.

M.2 Sample Modula-2 program

To give a flavor of Modula-2 programming, here is a program that does some numeric computation, reports sizes of some of the basic data types, and prints some numbers:

```
% cat hello-with-pi.mod
MODULE HelloWithPi;

FROM LongIO   IMPORT WriteFloat, WriteReal;
FROM NumberIO IMPORT WriteInt;
FROM StdChans IMPORT StdOutChan;
FROM StrIO    IMPORT WriteLn, WriteString;
FROM SYSTEM   IMPORT TSIZE;
```

[9] See `https://github.com/redstar/m2lang`.
[10] See `http://modula-2.net/m2r10`.

```
CONST
    PI = 3.14159265358979323846264338327950288;

VAR eps : LONGREAL;
      k : INTEGER;

BEGIN
    eps := 1.0;
    k := 0;

    WHILE  ((1.0 + 0.5 * eps) > 1.0)
    DO
        DEC (k);
        eps := 0.5 * eps
    END;

    WriteString ('TSIZE(LONGREAL)  = ');
    WriteInt (TSIZE(LONGREAL), 2);
    WriteLn;

    WriteString ('TSIZE(REAL)      = ');
    WriteInt (TSIZE(REAL), 2);
    WriteLn;

    WriteString ('TSIZE(SHORTREAL) = ');
    WriteInt (TSIZE(SHORTREAL), 2);
    WriteLn;

    WriteString ('TSIZE(LONGINT)   = ');
    WriteInt (TSIZE(LONGINT), 2);
    WriteLn;

    WriteString ('TSIZE(INTEGER)   = ');
    WriteInt (TSIZE(INTEGER), 2);
    WriteLn;

    WriteString ('TSIZE(SHORTINT)  = ');
    WriteInt (TSIZE(SHORTINT), 2);
    WriteLn;

    WriteString ('TSIZE(BOOLEAN)   = ');
    WriteInt (TSIZE(BOOLEAN), 2);
    WriteLn;

    WriteString ('TSIZE(CHAR)      = ');
    WriteInt (TSIZE(CHAR), 2);
    WriteLn;

    WriteString ('Machine epsilon  = 2**(');
    WriteInt (k, 0);
    WriteString ('), or about ');
```

```
      WriteFloat (StdOutChan(), eps, 4, 0);
      WriteLn;

      WriteString ('Hello from Modula-2: pi is about ');
      WriteFloat (StdOutChan(), PI, 36, 0);
      WriteLn

END HelloWithPi.
```

Modula-2 is case sensitive, and language keywords must be spelled in uppercase. Underscores are permitted in identifier names, but use of mixed case is traditional. As in Pascal, functions without arguments do not require an empty parenthesized list. The module name is repeated on the final `END` statement, which is terminated by a *period*, not a semicolon.

Following Pascal, semicolons in Modula-2 are statement *separators*, rather than *terminators*. However, Modula-2 has a null statement, so we could have used semicolons before the two `END` statements.

The `BOOLEAN` constants are `FALSE` and `TRUE`; on conversion to integers, they are 0 and 1.

Integer constants can be written in decimal, or as octal digits suffixed by `B`, or as uppercase hexadecimal digits suffixed by `H`. In the latter case, if the first digit would be a letter, then a leading 0 is required. Character constants can be written in decimal, or in octal with a suffix `C`. Examples are `0FACEH`, `7070B`, and `377C`. The `CHR()` and `ORD()` functions convert between small integers and characters.

String constants can be delimited either by apostrophes, as in this program, or by quotation marks, but there is no way to include the delimiter character in the string constant, as one can by its doubling in Fortran, `'Isn"t this nice!'`, or with backslash escapes in the C family: `"A quotation mark is \", right?"`.

`INC()` and `DEC()` functions for adding ± 1 to their arguments are conventional in Modula-2. The compiler knows how the two functions behave, and expands them inline, usually to just one or two instructions, instead of calling external functions.

Notice that the output functions are defined in several different modules, and that some default to using the standard output channel, while others need a function to supply that channel as their first argument.

Calls to external functions do not require a module prefix if their names are explicitly imported; otherwise, we would have to use the prefix, as in `StrIO.WriteString(...)`.

The final argument in numeric output functions is the minimum field width, and for floating-point output, the penultimate argument is the number of significant digits.

The test program intentionally uses the longest available floating-point type, which requires the `LongIO` module. It is available only with ISO Modula-2, so we need an extra compiler option to run the program:

```
% gm2 -fiso hello-with-pi.mod && ./a.out
TSIZE(LONGREAL)  = 16
TSIZE(REAL)      =  8
TSIZE(SHORTREAL) =  4
TSIZE(LONGINT)   =  8
TSIZE(INTEGER)   =  4
TSIZE(SHORTINT)  =  2
TSIZE(BOOLEAN)   =  1
TSIZE(CHAR)      =  1
Machine epsilon  = 2**(-63), or about 1.084E-19
Hello from Modula-2: pi is about 3.14159265358979323851280895940618620
```

Besides the eight basic data types whose sizes in bytes are shown here, there are three unsigned integer types (`SHORTCARD`, `CARDINAL`, and `LONGCARD`), and three complex floating-point types (`SHORTCOMPLEX`, `COMPLEX`, and `LONGCOMPLEX`). In Modula-2, the `BOOLEAN` type is not an integer type, but it can be converted with the `INTEGER8()` function.

The system-dependent module `SYSTEM` supplied with `gm2` provides several additional numeric types, including `CARDINAL8`, `CARDINAL16`, `CARDINAL32`, `CARDINAL64`, `COMPLEX32`, `COMPLEX64`, `COMPLEX128`, `INTEGER8`, `INTEGER16`, `INTEGER32`, `INTEGER64`, `REAL32`, `REAL64`, and `REAL128`. However, the types suffixed 128 are not necessarily IEEE

754 128-bit types — on common desktop platforms with the AMD64 processor family, they are instead the 80-bit format, but stored in 16-byte fields.

The long value for π in the `CONST` part does not carry a suffix to indicate its desired precision. The program output demonstrates that the constant was treated as an IEEE 754 80-bit value: comparison with a higher-precision value shows that it is correct to the 64-bit significand size of that format.

M.3 Initialization in Modula-2

Modula-2 declarations of variables do not permit initial values; as in many languages, that must be done separately. Although it is not evident from the complete programs in this appendix, modules provide a solution to the problem of guaranteed initialization of data: the *first time* that any function or procedure in a module is called from another module, code in the final top-level `BEGIN` block of the called module is executed.

A typical example of the need for one-time initialization is setting a default seed for a random number generator. Here is an implementation of such a generator:

```
% cat RandomNumbers.mod
IMPLEMENTATION MODULE RandomNumbers;

FROM StrIO         IMPORT WriteLn, WriteString;

CONST c = 5432109876;

VAR seed : INTEGER;

PROCEDURE NewSeed(newseed : INTEGER);
BEGIN
    seed := newseed;
END NewSeed;

PROCEDURE OldSeed() : INTEGER;
BEGIN
    RETURN seed
END OldSeed;

PROCEDURE RandomInteger() : INTEGER;
BEGIN
   seed := (c + seed) MOD 987654321; (* some calculation involving seed *)
   RETURN seed
END RandomInteger;

BEGIN
  seed := 1234567890;   (* module initialization: done exactly once *)
  WriteLn;
  WriteString('MODULE RandomNumbers is now initialized!');
  WriteLn;
  WriteLn
END RandomNumbers.
```

Procedures in such a module are automatically exported. We added output statements to the final block, so that we can see how often it is executed.

Here is the interface definition file:

```
% cat RandomNumbers.def
DEFINITION MODULE RandomNumbers;
```

```
PROCEDURE NewSeed(newseed : INTEGER);
PROCEDURE OldSeed() : INTEGER;
PROCEDURE RandomInteger() : INTEGER;

END RandomNumbers.
```

Here is the test program:

```
% cat TestRandomNumbers.mod
MODULE TestRandomNumbers;

FROM NumberIO      IMPORT WriteInt;
FROM RandomNumbers IMPORT RandomInteger;
FROM StrIO         IMPORT WriteLn, WriteString;

VAR k, r : INTEGER;
BEGIN
    FOR k := 1 TO 10 DO
        WriteString('k = ');
        WriteInt(k, 2);
        WriteString(' RandomInteger = ');
        r := RandomInteger();
        WriteInt(r, 9);
        WriteLn
    END;
END TestRandomNumbers.
```

To run the test, we have to compile the implementation separately, and then link its object file with the test program:

```
% gm2 -c RandomNumbers.mod

% gm2 TestRandomNumbers.mod RandomNumbers.o && ./a.out

MODULE RandomNumbers is now initialized!

k =  1 RandomInteger =  52051816
k =  2 RandomInteger = 201540075
k =  3 RandomInteger = 351028334
k =  4 RandomInteger = 500516593
k =  5 RandomInteger = 650004852
k =  6 RandomInteger = 799493111
k =  7 RandomInteger = 948981370
k =  8 RandomInteger = 110815308
k =  9 RandomInteger = 260303567
k = 10 RandomInteger = 409791826
```

There is only a single announcement from the initialization block, after which the generator returns a different number on each call.

Our simple linear congruential generator is a poor one; many better methods are known for updating the seed, but we do not need them for this small test.

M.4 Arrays and strings in Modula-2

Pascal and Modula-2 use slightly different syntax for declarations of arrays with more than one dimension:

```
VAR a : array [1 .. MX,   1 .. MY] of Real;         (* Pascal *)
VAR a : ARRAY [1 .. MX], [1 .. MY] OF REAL;         (* Modula-2 *)
VAR a : ARRAY [1 .. MX] OF ARRAY [1 .. MY] OF REAL; (* variant Modula-2 *)
```

The last declaration shows that, as in C, two-dimensional arrays are stored in *row order*, with the last subscript increasing most rapidly. A reference to an element of that array can be written as `a[i][j]` or as `a[i, j]`.

Character strings are one-dimensional arrays:

```
VAR s : array [1 .. MAXSTR] of Char; (* Pascal *)
VAR s : ARRAY [1 .. MAXSTR] OF CHAR; (* Modula-2 *)
```

In both languages, array lengths are part of their types, but unlike Pascal, in Modula-2 it is legal to assign a shorter string to longer one, but not the reverse:

```
MODULE StringAssignmentTest;
VAR s : ARRAY [1 .. 5]  OF CHAR;
    t:  ARRAY [1 .. 10] OF CHAR;
BEGIN
  s := 'a';           (* legal *)
  s := 'abcde';       (* legal *)
  t := 'abcdeghijk';  (* legal *)
  t := s;             (* legal *)
  s := t              (* SYNTAX ERROR *)
END StringAssignmentTest.
```

In Modula-2, when a shorter string is assigned to a longer one, the next unused character is set to a `NUL`, represented by the character value `0C`. The string length is therefore determinable, and can vary up to the declared maximum during execution. However, implementation of the standard `Strings` module, which supplies functions for assignment, comparison, concatenation, copying, deletion, insertion, and length, is complicated by the need to check for a `NUL`, as well as a full array.

Functions in Modula-2 can be declared with dynamic array arguments, where the index ranges are omitted, allowing the restriction on matching lengths to be relaxed. Here is an example from the `gm2` library code that shows the coding technique:

```
PROCEDURE Length (a: ARRAY OF CHAR) : CARDINAL;

FROM ASCII IMPORT nul;

VAR
   l, h: CARDINAL;

BEGIN
   l := 0;
   h := HIGH (a);

   WHILE (l <= h) AND (a[l] # nul) DO
      INC (l)
   END;

   RETURN (l)
END Length;
```

The compiler passes the array either via a descriptor that has a pointer to the first element, and a count, or as an argument pair. The `gm2` compiler does the latter, passing the element count as an unsigned default integer, followed by a pointer to the first element of the array. The function then treats the array as if its subscript range is 0 to `HIGH(name)`, where `HIGH()` is a built-in function that returns the count, less one.

Multidimensional arrays can be handled as in this fragment of a test program:

```
MODULE TestArrayDimensions;
...
VAR x4 : ARRAY [10 .. 20] OF
         ARRAY [150 .. 170] OF
         ARRAY [1500 .. 1530] OF
         ARRAY [15000 .. 15040] OF CARDINAL32;
...
PROCEDURE show4(xx : ARRAY OF ARRAY OF ARRAY OF ARRAY OF CARDINAL32);
BEGIN
    WriteString('HIGH(xx)          = ');
    WriteCard(HIGH(xx), 2);
    WriteLn;

    WriteString('HIGH(xx[0])       = ');
    WriteCard(HIGH(xx[0]), 2);
    WriteLn;

    WriteString('HIGH(xx[0, 0])    = ');
    WriteCard(HIGH(xx[0, 0]), 2);
    WriteLn;

    WriteString('HIGH(xx[0, 0, 0]) = ');
    WriteCard(HIGH(xx[0, 0, 0]), 2);
    WriteLn;
    WriteLn
END show4;
...
BEGIN
    ...
    show4(x4)
END TestArrayDimensions.

% gm2 TestArrayDimensions && ./a.out
...
HIGH(xx)          = 10
HIGH(xx[0])       = 20
HIGH(xx[0, 0])    = 30
HIGH(xx[0, 0, 0]) = 40
```

The array indexes in calls to `HIGH()` do not refer to memory locations, and are not bounds checked: they are merely placeholders. Replacing the 0 indexes by a big number, like -9999999, produces identical output.

The omission of a `SET` type in Modula-2 was based on the observation that there are two main applications of sets in computer programs: groups of zero or more objects from a limited collection, such as characters, colors, digits, letters, months, . . . , and sets of bits in a machine word. Regrettably, compilers for these languages generally restrict the set size to the number of bits in a single word, making them useless for characters, and on small machines, also for letters.

Modula-2 provides limited bit access via a subterfuge of system-dependent `BITSETnn` types. Here is a sample program:

```
% cat bittest.mod
MODULE BitTest;

FROM NumberIO IMPORT WriteInt;
FROM StrIO    IMPORT WriteLn, WriteString;
FROM SYSTEM   IMPORT BITSET16, CARDINAL16, INTEGER8, TSIZE;

CONST wordsize = 16;

VAR
    bits : BITSET16;
    k : CARDINAL;
    n : CARDINAL16;

BEGIN
    WriteString('TSIZE(BITSET16) = ');
    WriteInt(TSIZE(BITSET16), 0);
    WriteLn;

    n := 64206;  (* 0xface = 0o175316 = 0b0011111101011001110 *)
    bits := BITSET16(n);

    FOR k := 0 TO wordsize + 4 DO
        WriteString('bits[');
        WriteInt(k, 2);
        WriteString('] = ');
        WriteInt(INTEGER8(k IN bits), 0);
        WriteLn;
    END

END BitTest.
```

Execution of that program looks like this:

```
% gm2 bittest.mod && ./a.out
TSIZE(BITSET16) = 2
bits[ 0] = 0
bits[ 1] = 1
bits[ 2] = 1
bits[ 3] = 1
bits[ 4] = 0
bits[ 5] = 0
bits[ 6] = 1
bits[ 7] = 1
bits[ 8] = 0
bits[ 9] = 1
bits[10] = 0
bits[11] = 1
bits[12] = 1
bits[13] = 1
bits[14] = 1
bits[15] = 1
bits[16] = 0
bits[17] = 0
```

```
bits[18] = 0
bits[19] = 0
bits[20] = 0
```

The index of a `BITSET` is required to be an unsigned number, and our code intentionally queried outside the range. The program comment on the assignment to `n` shows the expected 16-bit string, and the output demonstrates that on this system, bits are numbered from the right; other platforms might number bits from the left.

M.5 Calling practice in Modula-2

As in Pascal, Modula-2 has two types of procedure parameters: variable, and default. The first type is qualified by the keyword `VAR`, which tells the compiler that changes in the argument inside the procedure must be reflected in the caller. That can be implemented quickly with *call-by-reference*, where the address of the caller's variable is passed to the callee: any changes to it are immediately visible in both. Alternatively, it can be implemented by *copy-in, copy-out*: the callee makes a local copy of the argument, and then on return, overwrites the variable in the caller.

Without the `VAR` keyword, the caller passes a *copy* of the argument: that is termed *call-by-value*. Any changes in the callee affect that copy, but not the original argument. For scalar arguments, this carries low overhead. However, for large arrays and records, the allocation and freeing of argument copies involves substantial run-time overhead. For that reason, programmers of Pascal and Modula-2 might intentionally add a `VAR` qualifier, even when they know that the callee does not modify the argument.

By contrast, in the C family, a function declaration can prefix a structure or array argument with the `const` attribute, and the compiler then prevents any changes to the argument. No large argument copying is ever needed. The language behavior is then *call-by-value* for scalars, and *call-by-reference* for composite objects.

M.6 Type transfers in Modula-2

A numeric type must sometimes be converted to another type while preserving the value, such as with the C typecast `(double)n`. The Modula-2 equivalent looks like this: `VAL(REAL, n)`. The first argument is the result type, and the second is the value to be converted.

However, we sometimes just want to reinterpret the bit patterns of a value, *without* doing any data conversion. For example, in order to access the bits of a floating-point number, we need a way to view it as a different type. Modula-2 makes this possible by using a type name as a pseudo-function call:

```
% cat ShortRealToInteger.mod
MODULE ShortRealToInteger;

FROM NumberIO IMPORT WriteBin, WriteHex, WriteInt, WriteOct;
FROM RealIO   IMPORT WriteFloat;
FROM StdChans IMPORT StdOutChan;
FROM StrIO    IMPORT WriteLn, WriteString;
FROM SYSTEM   IMPORT INTEGER32;

VAR x : SHORTREAL;
    n : INTEGER32;

BEGIN
    x := 1.25;
    WriteString ('x = ');
    WriteFloat (StdOutChan(), x, 9, 0);
    WriteLn;
```

```
    n := INTEGER32(x);    (* <== here is the type transfer *)

    WriteString ('n = ');
    WriteInt(n, 0);
    WriteString (' = 0x');
    WriteHex(n, 0);
    WriteString (' = 0o');
    WriteOct(n, 0);
    WriteString (' = 0b');
    WriteBin(n, 0);
    WriteLn
END ShortRealToInteger.
```

We can run that program like this:

```
% gm2 ShortRealToInteger.mod && ./a.out
x = 1.25000000
n = 1067450368 = 0x3FA00000 = 0o7750000000 = 0b11111110100000000000000000000000
```

A similar program handles the type `REAL` using the `INTEGER64` type, but it cannot output octal and binary strings because `WriteLongOct()` and `WriteLongHex()` are not in the standard modules — another example of the lack of orthogonality in the Modula-2 library.

We cannot use this technique to inspect the bits of a `LONGREAL` value because Modula-2 lacks support for an `INTEGER128` type. The next section provides a solution.

M.7 Bit field access in Modula-2

Pascal and Modula-2 both have a variant `RECORD` type that is an analogue of the `union` type in the C language family. We can use it to access the fields of a floating-point number in memory.

The details are hidden from the programmer who need only `IMPORT` the definition module to access the sign, exponent, and significand of the IEEE 754 80-bit format used for the `LONGREAL` data type, as well as to get and set the payload of a NaN. We prefix function names with `F80` to emphasize the restriction to that format:

```
% cat F80Fields.def
DEFINITION MODULE F80Fields;

FROM SYSTEM IMPORT CARDINAL64;

PROCEDURE F80GetSignBit          (x : LONGREAL) : INTEGER;
PROCEDURE F80GetBiasedExponent   (x : LONGREAL) : INTEGER;
PROCEDURE F80GetUnbiasedExponent (x : LONGREAL) : INTEGER;
PROCEDURE F80GetExponentBias     (x : LONGREAL) : INTEGER;
PROCEDURE F80GetPayload          (x : LONGREAL) : CARDINAL64;
PROCEDURE F80GetSignificand      (x : LONGREAL) : CARDINAL64;

PROCEDURE F80SetPayload          (x : LONGREAL; payload : CARDINAL64) : LONGREAL;

END F80Fields.
```

The implementation module is lengthy, and it illustrates several features of the Modula-2 language that we have not yet encountered. We exhibit it here in code fragments, each preceded by commentary.

The module begins with the keyword `IMPLEMENTATION`, indicating its purpose. It is followed by `IMPORT` statements that identify the needed library functions, and three constants that we use later.

```
% cat F80Fields.mod
IMPLEMENTATION MODULE F80Fields;

FROM BitWordOps  IMPORT WordAnd, WordOr, WordShl;
FROM SYSTEM      IMPORT CARDINAL32, CARDINAL64;

CONST F80EMAX = 7FFFH;          (* 32767 *)
      F80ExponentBias = 3FFFH;  (* 16383 *)
      NF80Words = 4;            (* 80-bit format stored in 4 32-bit words on AMD64 *)
```

To clarify the bit operations that we need to perform, we first document the floating-point storage layout in this comment block:

```
(*
** Layout of IEEE 754 80-bit LONGREAL in little-endian 128-bit field:
** bit num = 3322 2222 2222 1111 1111 1100 0000 0000
**           1098 7654 3210 9876 5432 1098 7654 3210
** word[0] = bbbb_bbbb_bbbb_bbbb_bbbb_bbbb_bbbb_bbbb -- low  significand 0..31
** word[1] = bbbb_bbbb_bbbb_bbbb_bbbb_bbbb_bbbb_bbbb -- high significand 32..63
** word[2] = ????_????_????_????_seee_eeee_eeee_eeee -- unset 80..95, sign 79, exponent 64..78
** word[3] = ????_????_????_????_????_????_????_???? -- unset 96..127
*)
```

The first two 32-bit words contain the significand, and the bottom half of the third word contains the sign and the biased exponent. There is *no hidden bit* in the 80-bit format, and the implicit binary point follows the first stored significand bit. The remaining bits of the 128-bit field are unused, but are zero during execution. The layout and bit numbering are *architecture dependent*. Our code assumes the common AMD64 family, which uses *little-endian* addressing and bit numbering.

The storage layout is mapped onto a variant `RECORD` whose views are defined by enumeration constants in the definition of the `F80Choice` type. Notice the peculiar vertical bars that separate members of a Modula-2 `CASE` statement:

```
TYPE F80Choice = ( Flt, UInt32, UInt64 );

TYPE F80Type =
     RECORD
          CASE which : F80Choice OF
          UInt32 :
              w32 : ARRAY [0 .. NF80Words - 1] OF CARDINAL32
          | UInt64 :
              w64 : ARRAY [0 .. (NF80Words DIV 2) - 1] OF CARDINAL64
          | Flt:
              f : LONGREAL
          END                   (* case *)
     END;                       (* record *)
```

We also need to be able to view a 64-bit unsigned integer as a pair of 32-bit values, so two more `TYPE` definitions are needed:

```
TYPE U64Choice = ( Card32, Card64 );

TYPE U64Type =
    RECORD
          CASE which : U64Choice OF
          Card32:
```

```
            w32 : ARRAY [0 .. 1] OF CARDINAL32;
          | Card64:
            v : CARDINAL64;
          END                   (* case *)
     END;                       (* record *)
```

The Modula-2 library has functions for bitwise operations on 32-bit words, but regrettably, not on 64-bit words, so our first two functions, `BitAnd64()` and `BitOr64()`, supply them. There are several assignments to the `which` member of the `U64Type RECORD` that clutter the code, but they are actually not needed, and could be removed. However, Modula-2 documentation recommends that they be used to identify the variant currently in use. Compile-time optimization removes them from the assembly code, so they have no run-time overhead.

```
PROCEDURE BitAnd64 (a, b : CARDINAL64) : CARDINAL64;
VAR aa, bb, cc : U64Type;
BEGIN
    aa.which := Card64;
    aa.v := a;
    bb.which := Card64;
    bb.v := b;
    aa.which := Card32;
    bb.which := Card32;
    cc.which := Card32;
    cc.w32[0] := WordAnd(aa.w32[0], bb.w32[0]);
    cc.w32[1] := WordAnd(aa.w32[1], bb.w32[1]);
    cc.which := Card64;
    RETURN cc.v
END BitAnd64;

PROCEDURE BitOr64 (a, b : CARDINAL64) : CARDINAL64;
VAR aa, bb, cc : U64Type;
BEGIN
    aa.which := Card64;
    aa.v := a;
    bb.which := Card64;
    bb.v := b;
    aa.which := Card32;
    bb.which := Card32;
    cc.which := Card32;
    cc.w32[0] := WordOr(aa.w32[0], bb.w32[0]);
    cc.w32[1] := WordOr(aa.w32[1], bb.w32[1]);
    cc.which := Card64;
    RETURN cc.v
END BitOr64;
```

We now have the internal functions that are needed to implement the user-visible functions of the module.

The first task is to extract the sign bit, which is found in middle of the third word of the field. We declared our integer arrays with ranges that start at 0, so the required index is 2. The surprise here is that the left- and right-shift functions in the `gm2` module `BitWordOps` do the *opposite* of what their documentation says, so to shift the word right by 15 bits, we have to call `WordShl()`, not the expected `WordShr()`! Before doing that, we have to mask off all of the unwanted bits in the word by a call to `WordAnd()`:

```
PROCEDURE F80GetSignBit (x : LONGREAL) : INTEGER;
VAR u : F80Type;
BEGIN
```

```
    u.which := Flt;
    u.f := x;
    u.which := UInt32;
    RETURN WordShl(WordAnd(u.w32[2], 8000H), 15) (* NB: RIGHT shift here! *)
END F80GetSignBit;
```

The 15-bit biased exponent is found in the same 32-bit word as the sign, so all that is needed is to clear the top 17 bits:

```
PROCEDURE F80GetBiasedExponent (x : LONGREAL) : INTEGER;
VAR u : F80Type;
BEGIN
    u.which := Flt;
    u.f := x;
    u.which := UInt32;
    RETURN WordAnd(u.w32[2], 7FFFH)
END F80GetBiasedExponent;
```

Programmers should not have to remember the size of the exponent bias, so we supply two functions to report the true exponent, and to return the bias:

```
PROCEDURE F80GetUnbiasedExponent (x : LONGREAL) : INTEGER;
BEGIN
    RETURN F80GetBiasedExponent(x) - F80ExponentBias
END F80GetUnbiasedExponent;

PROCEDURE F80GetExponentBias (x : LONGREAL) : INTEGER;
BEGIN
    RETURN F80ExponentBias
END F80GetExponentBias;
```

The last component of the 80-bit floating-point value is the 64-bit significand, stored in the first 64-bit word:

```
PROCEDURE F80GetSignificand (x : LONGREAL) : CARDINAL64;
VAR u : F80Type;
BEGIN
    u.which := Flt;
    u.f := x;
    u.which := UInt64;
    RETURN u.w64[0]
END F80GetSignificand;
```

IEEE 754 NaN and Infinity values share the highest biased exponent, `EMAX`. They are distinguished by a zero fraction for Infinity, and a nonzero fraction for a NaN. On many machines, there are two types of NaNs: quiet and signaling. The IEEE 754 Standard forbids the use of the sign bit for that identification, and all CPU vendors seem to have adopted the convention that the *highest-order fraction bit* has that job, but they disagree on its meaning. In the AMD64 family, a 0-bit there means a signaling NaN, and a 1-bit, a quiet NaN. Thus, on that family, an all-bits-1 pattern is a negative quiet NaN, and could be a useful storage initializer in the C family with a call to `memset(&x, 0xff, sizeof(x))`.

The sign of a NaN is never significant, and NaNs have the distinguishing property that they never compare equal to anything, *including themselves*. By contrast, Infinity values of the same sign compare equal. The conditional test `x <> x` is true only if `x` is a NaN, *provided that* the compiler does not incorrectly optimize it away.

In the 32-, 64-, and 128-bit IEEE 754 formats, the significand has a hidden integer bit before the stored fraction. However, in the 80-bit format, the integer bit is explicitly stored at the left of the 64-bit significand, and is a 1-bit for both Infinity and NaN. Thus, only the rightmost 62 bits of the significand are available to hold a user-defined

payload. Hardware never generates a nonzero payload for a NaN, but tends to preserve the payload of a NaN operand. The recent RISC-V architecture, however, *always* zeros the payload of a computed NaN.

To recover the payload from a floating-point number, we adopt the convention that the payload is zero for everything but a NaN, and otherwise is the rightmost 62 bits of a NaN. Our code therefore has to identify a NaN, and then mask off the payload bits:

```
PROCEDURE F80GetPayload(x : LONGREAL) : CARDINAL64;
VAR e : CARDINAL32;
    s : CARDINAL64;
BEGIN
    e := F80GetBiasedExponent(x);
    s := F80GetSignificand(x);

    IF e <> F80EMAX THEN
        s := 0          (* x is finite *)
    ELSIF s = 8000000000000000H THEN
        s := 0          (* x = Infinity *)
    ELSE                (* x is QNaN or SNaN *)
        s := BitAnd64(s, 3FFFFFFFFFFFFFFFH) (* clear top 2 bits *)
    END;

    RETURN s
END F80GetPayload;
```

To allow the user to set a payload in a floating-point number, our function simply returns the input value when it is not a NaN. Otherwise, it clears all but the top two bits of the input significand, and then ORs in the bottom 62 bits of the user-supplied payload, taking care to ensure that it is nonzero:

```
PROCEDURE F80SetPayload(x : LONGREAL; payload : CARDINAL64) : LONGREAL;
VAR e : CARDINAL32;
    p : CARDINAL64;
    s : CARDINAL64;
    xx : F80Type;
BEGIN
    e := F80GetBiasedExponent(x);
    s := F80GetSignificand(x);
    xx.which := Flt;
    xx.f := x;

    IF (e = F80EMAX) AND (s <> 8000000000000000H) THEN          (* x is QNaN or SNaN *)
        (* top integer significand bit: 1 (NaN or Inf);
           highest fraction bit: 1 (QNaN), 0 (SNaN); rest: at least one 1-bit *)
        xx.which := UInt64;
        xx.w64[0] := BitAnd64(xx.w64[0], 0C000000000000000H);   (* keep original top 2 bits *)
        p := BitAnd64(payload, 3FFFFFFFFFFFFFFFH);              (* clear top 2 bits *);

        IF xx.w64[0] = 0C000000000000000H THEN
            ;                                                   (* canonical quiet NaN *)
        ELSIF p = 0 THEN
            p := 1                                              (* payload must be nonzero *)
        END;

        xx.w64[0] := BitOr64(xx.w64[0], p);
        xx.which := Flt
```

```
    END;

    RETURN xx.f
END F80SetPayload;

END F80Fields.
```

That completes our presentation of the module `F80Fields`. Examination of the compiler-generated optimized assembly code shows that the function bodies are generally short, except that the bitwise operations, which would be single inline instructions in the C family, result in calls to external functions.

M.8 Programming the Modula-2 MathCW interface

Modula-2 code can use the MathCW library via an interface definition, fragments of which look like this:

```
% cat MathCW.def
DEFINITION MODULE FOR "C" MathCW;

EXPORT UNQUALIFIED
    acosdegf, acosdeg, acosdegl,
    acosf, acos, acosl,
    acoshf, acosh, acoshl,
    ...
    zetam1f, zetam1, zetam1l,
    zetnm1f, zetnm1, zetnm1l,
    zetnumf, zetnum, zetnuml;

PROCEDURE acosdegf      (x : SHORTREAL)        : SHORTREAL;
PROCEDURE acosf         (x : SHORTREAL)        : SHORTREAL;
PROCEDURE acoshf        (x : SHORTREAL)        : SHORTREAL;
...
PROCEDURE acosdeg       (x : REAL)             : REAL;
PROCEDURE acos          (x : REAL)             : REAL;
PROCEDURE acosh         (x : REAL)             : REAL;
...
PROCEDURE acosdegl      (x : LONGREAL)         : LONGREAL;
PROCEDURE acosl         (x : LONGREAL)         : LONGREAL;
PROCEDURE acoshl        (x : LONGREAL)         : LONGREAL;
...
PROCEDURE zetam1l       (x : LONGREAL)         : LONGREAL;
PROCEDURE zetnm1l       (n : INTEGER)          : LONGREAL;
PROCEDURE zetnuml       (n : INTEGER)          : LONGREAL;
...
END MathCW.
```

The `EXPORT` statement lists the function names that are to be externally accessible, and the `UNQUALIFIED` keyword suppresses *name mangling* that would normally occur to incorporate module names, and possibly argument types, into the function names.

Here is a small test program that calls one of the MathCW functions and reports its value:

```
MODULE Terfl;

FROM FpuIO     IMPORT WriteLongReal;
```

```
FROM MathCW    IMPORT erfl;
FROM StrIO     IMPORT WriteLn, WriteString;

VAR x, y : LONGREAL;

BEGIN
    x := 0.5;
    y := erfl(x);

    WriteString('erfl(');
    WriteLongReal(x, 0, 1);
    WriteString(') = ');
    WriteLongReal(y, 0, 19);
    WriteLn

END Terfl.
```

We can compile, link, and execute it like this:

```
% gm2 terfl.mod -lmcw && ./a.out
erfl(0.5) = 0.5204998778130465377
```

Comparison with a higher precision value shows that the result is correct to its 64-bit significand.

The Modula-2 handling of array arguments means that functions in the MathCW library that take character string arguments cannot be called directly: they need an interface function, such as in this code fragment:

```
(* Modula-2 portion *)
PROCEDURE gm2_qnanf (s : ARRAY OF CHAR) : SHORTREAL;

FROM MathCW IMPORT gm2_qnanf;

VAR x : REAL;
...
x := gm2_qnanf('0FEEDCAFEH');

(* C portion *)
#include <mathcw.h>
...
float gm2_qnanf(unsigned int n, const char * s) { return qnanf(s); }
```

It is regrettable that `gm2` does not provide a cleaner way of defining interlanguage communication, to avoid the need for naming the C function differently in Modula-2. Some Fortran implementations pass character string lengths as additional arguments at the *end* of the declared argument list, allowing a direct interlanguage call.

M.9 Limitations of Modula-2 and its math library

The `gm2` implementation of Modula-2 provides additional modules `Builtins`, `ComplexMath`, `LongComplexMath`, `ShortComplexMath`, `RealMath`, `LongRealMath`, and `libm` that provide access to some, but not all, of the C mathematical function repertoire. The expected module `ShortRealMath` is curiously absent. While functions from the higher-precision companions can be used, that introduces the problem of *double rounding*.

The modules `LowShort`, `LowReal`, and `LowLong` provide functions for getting the sign, exponent, and significand of floating-point numbers, and for doing exact scaling. However, there are no property test functions, such as `isinf()`, `isnan()`, `isnegzero()`, and `issubnormal()`, nor are there predefined constants for many special floating-point values, as provided by the Standard C header file, `<float.h>`.

There are no native Modula-2 operators for bitwise operations on integers, but the modules `BitBlockOps`, `BitWordOps`, and `BitByteOps` supply some of the needed functions. Bit field access can be programmed with those functions, but seems to be absent from standard modules.

There is no support for hexadecimal floating-point constants, but that can be worked around with suitable library module support with run-time conversions of numeric strings, as in this snippet for getting a correctly rounded value of π:

```
FROM FpuIO           IMPORT WriteLongReal;
FROM RealConversions IMPORT StringToLongReal;
FROM StrIO           IMPORT WriteLn, WriteString;

VAR ok  : BOOLEAN;
    xxx : LONGREAL;
...
    StringToLongReal('+0x1.921fb54442d18469898cc51701b839a2p+1', xxx, ok);

    IF ok THEN
        WriteString ('LONGREAL xxx = ');
        WriteLongReal (xxx, 0, 36)
    ELSE
        WriteString ('ERROR: LONGREAL: no conversion')
    END;

    WriteLn;
```

Test programs that contain assignments, and printing, of the Matula-mandated decimal representations of the largest floating-point numbers in the IEEE 754 32-, 64-, and 80-bit formats produced the expected values in the first two cases, but Infinity for the `LONGREAL` case. That result persisted even when the constant was reduced to just four leading digits, calling into question the precision of the compiler's decimal-to-binary conversion. A test program that assigns the value 0.1 to a `LONGREAL` variable, and then prints it, shows that the constant was first converted with a 53-bit significand, then extended to the longer format. If the assignment is written in two steps, `x := 10.0; x := 1.0 / x;`, then the conversion is correct for a 64-bit significand. That defect makes it impossible to code `LONGREAL` constants that are out of the range of `REAL` values, unless they are wrapped in a call to an accurate string conversion function.

The only exponent letter permitted in floating-point constants is `E`: a lowercase letter is rejected. This is a minor nuisance when importing numerical data from other programming languages, most of which ignore lettercase of exponent letters, and some use additional exponent letters, such as `D`, `Q`, `W`, and `X`, that identify the precision of the constants.

The lack of any escape representation of nonprintable characters in strings forces messy use of functions for string and character concatenation. If `\n` could represent a newline in strings, most of the `WriteLn()` calls in Modula-2 software could be eliminated.

Further tests with negative zero constants, subnormals, compile-time generation of Infinity from `1.0 / 0.0`, NaN from `0.0 / 0.0`, and comparison of NaNs, show that are all handled correctly by `gm2`.

The output functions for floating-point types are incomplete, inconsistent, and irregularly supplied by different modules. The alert reader may have noticed the different argument order in `WriteLongReal()` compared to `WriteFloat()`. Indeed, the single function `WriteFloat()` has different argument counts and types, depending on the module that it is defined in!

As with almost all strongly typed programming languages, producing output in Modula-2 requires tedious and verbose coding, and gives considerably less control over formatting than is possible with the `printf()` functions in the C family. The more recent languages Go and Rust solve that problem: see Appendix G on page 946 and Appendix R on page 994. The University of Ulm Modula-2 compiler provides several library module functions that are modeled on `printf()`, but those extensions are not in the ISO Standard, or in the `gm2` implementation.

M.10 Summary

The preceding section identified several problems with using Modula-2 for numeric programming. Because of the uncertain precision of numeric constants, programmers need to take special care if they expect to produce code versions for all three supported floating-point types.

The requirement of different library functions for different data types, and the lack of any reliable way to specify the precision of numeric constants, as we do in the MathCW library code with `FP()` wrappers, substantially increase the maintenance work of Modula-2 code that uses different floating-point types.

The presence of digit-count arguments in many output functions could be repaired by adopting the convention that a zero count means the minimal number of digits for *exact round-trip conversion*, using the Matula formula. At present, programmers with little experience in floating-point arithmetic and numerical software are likely to get those counts wrong!

The modest function repertoire offered by standard modules for access to the mathematical library strongly suggests that the future library developers *must* consult experts in numerical programming for help in enhancing those facilities: the Modula-2 language itself need not be changed at all. Similar problems afflict almost all programming languages, but numerical experts have known better for decades.

The interface to the MathCW library described in this appendix is a substantial enhancement for writing numerical software in Modula-2. That interface is accompanied by others that fill in some of the gaps in the standard library, such as `WriteLongBin()` and `WriteLongOct()` in module `MoreLongIO`, and more bit operations supplied by the `BitOps32` and `BitOps64` modules.