K COBOL interface

THE USE OF COBOL CRIPPLES THE MIND; ITS TEACHING SHOULD, THEREFORE, BE REGARDED AS A CRIMINAL OFFENSE.
— EDSGER W. DIJKSTRA,
Selected Writings on Computing: a Personal Perspective (1972).

K.1 COBOL history and language overview K–1
K.2 COBOL code layout K–4
K.3 Free-format source code K–4
K.4 Control flow and modularization in COBOL K–4
K.5 Formatted output and rounding in COBOL K–5
K.6 IEEE 754 arithmetic deviations in COBOL K–7
K.7 Interfacing COBOL to C K–7

COBOL is the third major programming language still in use, after Fortran and Lisp. A multidisciplinary panel met in 1959 to work on the design of a new language intended for business data processing, and named it COBOL, an acronym of *COmmon Business-Oriented Language*. The first COBOL compiler was operational in the summer of 1960, and in the more than six decades since, it became one of the most widely used programming languages, with compiler implementations on most commercial computers, and multiple national and international standards from 1968 to 2023. ISO/IEC 1989:2023 is the third, and latest, edition.

Many modern programming languages share a common ancestry. The first version of the Algol language was designed by an international committee of computer scientists who met in Zurich, Switzerland, in early 1958, and the first Algol compiler was operational later that year. Significant extensions of Algol 58 were introduced in 1960 and 1968, but the language largely disappeared from use in the 1970s, and never saw wide interest outside of Europe. It was, however, replaced by numerous descendants, including the teaching language Pascal, and the C-language family, all of which borrow important ideas, and syntax, from Algol. The Algol designs were founded on rigorous minimal grammars, a practice followed in most modern programming languages. However, Algol appears to have had *no effect whatever* on the COBOL language design, which progressed in isolation from most other computer languages, and became large, complex, and idiosyncratic.

Lisp and Algol introduced the useful notion of recursive functions, most easily implemented with a run-time call stack, but few other languages of the time permitted recursion, and COBOL still does not, except in a restricted form introduced in the 2002 ISO COBOL Standard.

K.1 COBOL history and language overview

Although Fortran had a five-year head start on COBOL, and was already in wide use on computers from multiple vendors, Fortran lacked several critical features considered essential for business applications, including character strings, wide-precision decimal arithmetic for financial calculations, record structuring of program data and files, and sorting and merging of external files. For example, an inventory record might have character string fields containing things like catalog numbers, manufacturer names and addresses, part numbers, product descriptions, purchase dates, and so on, plus numeric fields for item quantities and unit costs.

In addition, the COBOL designers may have felt that the mathematical notation for expressions in Fortran would be foreign to people in the business community, so the language was defined with concepts like divisions, paragraphs, sections, sentences, and statements, and with verbose descriptions of expression evaluation that

resemble English sentences. Thus, COBOL programs contain statements like `MULTIPLY COST BY TAX-RATE GIVING BILLABLE-AMOUNT`, whereas a Fortran programmer might more succinctly write `BILAMT = TAXRTE * COST`. COBOL has another statement verb that shortens numeric expression evaluation, allowing `COMPUTE BILLABLE-AMOUNT = COST * TAX-RATE`.

Unlike most modern programming languages, there are no semicolons in COBOL to permit multiple short statements per line.

Assignments in COBOL can be written as `MOVE X TO Y`, or as `COMPUTE Y = X`. Assignments can have multiple targets, as in `MOVE E TO A, B, C, D` and `COMPUTE A, B, C, D = E`.

The `COMPUTE` verb can be used only for numeric expressions; character-string assignments require the `MOVE` form.

Because hyphens are valid characters in COBOL identifiers, surrounding spaces are essential if hyphen means subtraction: `COMPUTE A = B - C` does arithmetic, but `COMPUTE A = B-C` is an simple assignment of the value of the variable `B-C`.

Addition and subtraction operations can be applied to multiple variables with code like `SUBTRACT 3 FROM A, B, C`. Summing a list of variables can be done with `ADD A, B, C, D GIVING E` or `COMPUTE E = A + B + C + D`. There is similar syntax for `DIVIDE` and `MULTIPLY`, but that form is likely rare.

In its early years, COBOL's only numeric type was fixed-point decimal, where type declarations indicated whether values can be signed, and how many digits there are on both sides of a decimal point. Thus, a COBOL program might contain variable declarations like these:

```
WORKING-STORAGE SECTION.
01 K    PICTURE S9(3).
01 L    PICTURE S999.
01 COST PICTURE 9(7)V9(2).
```

They declare top-level variables `K` and `L` as signed 3-digit decimal integers, and `COST` as an `unsigned` decimal value with 7 digits before the decimal point, and 2 after. The letter `V` (possibly from French *virgule*, meaning comma) in the `PICTURE` specification represents the decimal point, which is implicit in the stored fixed-point value.

Identifiers in COBOL may be up to 30 characters long, and may contain letters, digits, hyphen, and underscore, but may not start or end with a hyphen, or begin with an underscore, For example, `X1`, `X-1`, and `X_1` are valid variable names, and surprisingly, so are `1X`, `1-X`, and `1_X`.

The maximum number of decimal digits in a number has varied with COBOL Standards. It was set at 18 in the 1960s, but was raised in 1985 to 23, and later, to 30. IBM mainframe compilers with a special option permit up to 31, and GnuCOBOL permits up to 38. The design of the IEEE 754 128-bit 34-digit decimal floating-point format was partly guided by the needs of COBOL, as well as ensuring the useful property that products of floating-point numbers are exactly representable in the next higher precision, and free from both underflow and overflow.

Characters string variables in COBOL are fixed length, blank padded on the right when assigned a shorter string, and truncated on the right when assigned a longer string. They are limited to at most 150 characters, and are declared and used with code like this:

```
WORKING-STORAGE SECTION.
01 S    PICTURE X(40).
...
    MOVE 'This is a string'                    TO S
    MOVE "This is also a string"               TO S
    MOVE 'This is a string, isn''t it?'        TO S
    MOVE "This is a string with ""quoted words""" TO S
```

As shown, either apostrophes or quotation marks may delimit strings. If the delimiter is needed inside the string, it is doubled, just as is done with apostrophe-delimited strings in Fortran. Unlike strings in the C-language family and many others in the Unix world, there is no provision in COBOL for embedded octal or hexadecimal or Unicode escape sequences to represent unprintable, or hard to input, characters.

Although most IBM mainframe operating systems use the 8-bit EBCDIC (Extended Binary-Coded Decimal Interchange Code) character set, most other operating systems and programming languages developed since the

late 1960s use the 7-bit ASCII (American Standard Code for Information Interchange) character set. Starting in the 1980s, work began to extend ASCII to handle all of the world's historical writing systems, for which more than 100,000 character glyphs are required. The new character set is called Unicode, and nominally requires 21-bit characters. However, multiple encodings of Unicode are possible, including UTF-8, UTF-16, and UTF-32. The UTF-8 encoding is widely used on Unix systems, because it automatically makes ASCII files valid UTF-8 files. Microsoft Windows, and IBM COBOL, prefer the UTF-16 encoding, where two bytes are needed for every character, and there is provision for an escape to higher character numbers beyond the $2^{16} = 65\,536$ slots in UTF-16. UTF-8 may need 1 to 4 bytes to encode every Unicode character. The GnuCOBOL compiler expects UTF-8 encoding in strings, and IBM mainframe COBOL compilers assume EBCDIC in ordinary quoted strings, or UTF-16 in national quoted strings of the form `N'...'`, or hexadecimal representations of UTF-16 in `NX'hhhh...'`, where the number of hexadecimal digits must be a multiple of 4. String variables to hold n UTF-16 characters are declared with `PICTURE N(`n`)`.

In support of communication with code in the C-language family, some recent COBOL compilers permit variable-length NUL-terminated string constants in the form `Z"This is a string"`. If that form is not supported, it can be simulated with an expression `FUNCTION CONCATENATE(StringName, X'00')`.

Because of its English-like syntax, COBOL has almost 1000 reserved words, many with abbreviated forms (such as `PIC` for `PICTURE`) and many of them candidates for program variables, such as `ADDRESS`, `LINE`, `TITLE`, `USER`, and `WINDOW`. In addition, the existence of several different language standards, and multiple independent compiler implementations, each with its own language extensions and additional reserved words, has been a barrier to portability of COBOL software. However, when compilers implement the language support offered by the several different COBOL compilers from the business industry leader, IBM, code portability is improved.

Beyond its standard reserved words, COBOL also recognizes more than 100 intrinsic function names that programmers should avoid, with names like `ABS`, `PI`, `SUM`, and `TRIM`. Most have a fixed number (0 to 4) of arguments, but several permit an unlimited number, such as `CONCATENATE`, `MAX`, and `MIN`. Intrinsic functions that take zero arguments can be used with, or without, an empty parenthesized argument list.

COBOL was later extended with support for floating-point arithmetic in both binary and decimal formats. Here are typical declarations of variables with those types:

```
WORKING-STORAGE SECTION.
01 D1  FLOAT-DECIMAL-16.
01 D2  FLOAT-DECIMAL-34.
01 X1  FLOAT-SHORT.
01 X2  COMPUTATIONAL-1.
01 Y1  FLOAT-LONG.
01 Y2  COMPUTATIONAL-2.
01 Y3  COMP-2.
```

The variables `D1` and `D2` have the types of the IEEE 754-2008 64-bit and 128-bit decimal formats, which hold 16 and 34 digits, respectively. Their types were introduced with the 2014 ISO COBOL Standard.

The variables `X1` and `X2` have the same type, corresponding to a single-word *binary* floating-point value. On IBM mainframe systems, that is the System/360 32-bit hexadecimal format, but on most other systems designed since the 1980s, it is the IEEE 754 32-bit binary format. However, compilers, including GnuCOBOL, on GNU/Linux systems on the IBM mainframe architecture use only IEEE 754 floating-point formats.

The variables `Y1`, `Y2`, and `Y3` also have the same type, corresponding to a platform-dependent double-word 64-bit format.

The existence of explicit digit counts in the `PICTURE` declarations of COBOL numeric variables means that the language has to define what happens when a value is assigned to a variable declared with fewer digits. The rule is that high-order digits are silently discarded (that is, arithmetic and assignment are modulo the target power of 10), but there is provision with the `OVERFLOW` keyword to catch such errors, as in this fragment:

```
WORKING-STORAGE SECTION.
01 A-2 PICTURE 99.
01 A-3 PICTURE 999.
...
```

```
MOVE 123 to A-3
MOVE A-3 to A-2 ON OVERFLOW GO TO HANDLE-OVERFLOW
```

COBOL has a set of mathematical functions similar to those available in Fortran, but the calling sequence requires an extra keyword:

```
COMPUTE A = FUNCTION SQRT(2.5)
```

Intrinsic function names in COBOL are *type generic*: the compiler arranges to call type-specific versions in the run-time library.

K.2 COBOL code layout

When COBOL, Fortran, and Lisp were first developed, the most common input source was 80-column punched cards. Fortran chose columns 1–5 for statement labels, a nonblank nonzero column 6 for a statement continuation indicator, columns 7–72 for program text, and columns 73–80 for an optional card sequence number. If sequence numbers were consistently supplied, a dropped card deck could be correctly reassembled in up to 8 passes in a punched-card sorter.

Fortran punched cards had vertical lines to indicate those boundaries, but when printing and video terminals became a source of input, it was easy to make mistakes, especially from text flowing into the sequence number columns. That was disastrous because Fortran does not require type declarations for variables, so a truncated variable name is undiagnosed. Unlike Fortran, COBOL has no default typing, and all variables must be declared before use.

Like Fortran, COBOL statements also have a fixed format, but with different boundaries: columns 1–6 are an optional sequence number, a nonblank column 7 indicates statement continuation, columns 8–11 are called *Area A*, where certain declarations must begin, and columns 12–72 are called *Area B*, where most statements reside. The precise rules for use of columns 7–72 are complex. Although columns 73–80 exist on punched cards, they are not defined in COBOL, and any text that extends beyond column 72 is lost.

K.3 Free-format source code

Our COBOL examples have shown only uppercase code and the fixed format of the punched-card era, but some modern COBOL compilers ignore lettercase in identifiers, and allow free-format input, similar to the practice of modern Fortran. In both languages, there are still compiler-dependent limits on the length of a line, and compile-time options, or in-code directives, or source file naming, are required to indicate use of the free format.

The free format appears not to be supported by IBM's mainframe COBOL compilers, so its use is likely to compromise portability to other systems, and we consequently avoid it in this Appendix.

K.4 Control flow and modularization in COBOL

Although it has numerous intrinsic functions, COBOL lacks the concept of user-defined functions and procedures, each with its own variables. Instead, it uses level numbers in variable declarations, and has `PERFORM` statements that jump to a labeled code `SECTION`, and then return after the last statement in that `SECTION`. It also has the ability to turn complete programs into subroutines that can be called with arguments.

The limited control structures of COBOL mean that `GO TO` statements are common, although the `PERFORM` verb has extensions that allow creation of loops with termination tests at the beginning or end. Here is a code fragment with `PERFORM` that determines and reports the machine epsilon in decimal floating-point arithmetic:

```
WORKING-STORAGE SECTION.
01 K   PIC S9(2)        VALUE IS 0.
01 X   FLOAT-DECIMAL-16 VALUE IS 1.
01 Y   FLOAT-DECIMAL-16 VALUE IS 2.
```

```
PROCEDURE DIVISION.

    PERFORM WITH TEST AFTER UNTIL Y EQUALS 1
        COMPUTE Y = 1 + X / 10
        IF Y IS GREATER THAN 1
            COMPUTE X = X / 10
            COMPUTE K = K - 1
        END-IF
    END-PERFORM

    DISPLAY "FLOAT-DECIMAL-16 machine epsilon = ", X,
             " = 10**(", K, ")"
```

Notice that type declarations permit run-time initialization of variables with scalars following the `VALUE IS` keywords, and the keyword `IS` can be omitted. The long relational phrase in the `IF` statement can be shortened to the keyword `GREATER`, or to the operator `<`. The keyword `EQUALS` can be reduced to `EQUAL` or the operator `=`.

Using `PERFORM` with remote code blocks that themselves contain `PERFORM` statements is a serious source of implementation-dependent issues, and has been called a *minefield*.[1]

As in Fortran, uninitialized variables in COBOL are usually not reported by the compiler, and have unpredictable values at run time.

A counted loop, similar to what the Fortran `DO` statement provides, looks like this example in COBOL that prints a table of factorials using a standard intrinsic function:

```
PERFORM WITH TEST AFTER VARYING K FROM 1 BY 1 UNTIL K > 40
    DISPLAY  K, "! = ", FUNCTION FACTORIAL(K)
END-PERFORM
```

Each `DISPLAY` statement produces a complete output line, but partial lines can be produced with code like this:

```
DISPLAY "  X = ", X WITH NO ADVANCING
DISPLAY "  Y = ", Y WITH NO ADVANCING
DISPLAY "  Z = ", Z WITH NO ADVANCING
DISPLAY " "
```

COBOL does not permit empty strings, so the last `DISPLAY` statement prints a single space. To avoid that, omit that statement, and drop `WITH NO ADVANCING` from the preceding statement.

K.5 Formatted output and rounding in COBOL

The COBOL language has nothing like C's `printf()` and `scanf()` family, or Fortran's formatted input/output statements, and the `DISPLAY` statement always outputs full precision for its numeric values, including leading zeros if they are decimal integers.

If full output precision is not desired, programmers must construct any needed shortened numeric output with intermediate variables, as in this example:

```
WORKING-STORAGE SECTION.
01 K1 PIC S9(1).
01 K2 PIC S9(2).
01 K3 PIC S9(3).
01 K4 PIC S9(4).
01 K5 PIC S9(5).
```

[1] See Niels Veerman and Ernst-Jan Verhoeven, *Cobol minefield detection*, Software — Practice and Experience, **36**(14) 1605–1642, 25 November 2006. doi:10.1002/spe.745

```
...
    MOVE 12345 TO K1, K2, K3, K4, K5
    DISPLAY "  K1 = ", K1 WITH NO ADVANCING
    DISPLAY "  K2 = ", K2 WITH NO ADVANCING
    DISPLAY "  K3 = ", K3 WITH NO ADVANCING
    DISPLAY "  K4 = ", K4 WITH NO ADVANCING
    DISPLAY "  K5 = ", K5
```

The output of a program with that code is

```
K1 = +5  K2 = +45  K3 = +345  K4 = +2345  K5 = +12345
```

When fractional values are assigned to variables with fewer fractional digits, the default is silent truncation (*round towards zero*) of the values. However, it is possible to request rounding in assignments, as illustrated by this code block that shows both practices:

```
WORKING-STORAGE SECTION.
01 F1 PIC S9V9.
01 F2 PIC S9V99.
01 F3 PIC S9V999.
...
    COMPUTE F1, F2, F3 = 1.255
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1, F2, F3 = -1.255
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1 ROUNDED, F2 ROUNDED, F3 = 1.255
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1 ROUNDED, F2 ROUNDED, F3 = 1.240
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1 ROUNDED, F2 ROUNDED, F3 = 1.250
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1 ROUNDED, F2 ROUNDED, F3 =  1.260
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1 ROUNDED, F2 ROUNDED, F3 = -1.240
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1 ROUNDED, F2 ROUNDED, F3 = -1.250
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3

    COMPUTE F1 ROUNDED, F2 ROUNDED, F3 = -1.260
    DISPLAY "F1 = ", F1, " F2 = ", F2, " F3 = ", F3
```

The output of that code looks like this:

```
F1 = +1.2 F2 = +1.25 F3 = +1.255
F1 = -1.2 F2 = -1.25 F3 = -1.255
F1 = +1.3 F2 = +1.26 F3 = +1.255
F1 = +1.2 F2 = +1.24 F3 = +1.240
F1 = +1.3 F2 = +1.25 F3 = +1.250
```

```
F1 = +1.3 F2 = +1.26 F3 = +1.260
F1 = -1.2 F2 = -1.24 F3 = -1.240
F1 = -1.3 F2 = -1.25 F3 = -1.250
F1 = -1.3 F2 = -1.26 F3 = -1.260
```

The first two lines illustrate the default truncation toward zero, while the remaining lines demonstrate that COBOL rounding follows the biased rule *round to nearest, with ties away from zero*.

The IBM *Enterprise COBOL for z/OS Version 6.1 Language Reference* manual of 29 December 2020 describes the rounding operation like this:

> When the size of the fractional result exceeds the number of places provided for its storage, truncation occurs unless `ROUNDED` is specified. When `ROUNDED` is specified, the least significant digit of the resultant identifier is increased by 1 whenever the most significant digit of the excess is greater than or equal to 5.

If numeric values are computed as binary or decimal floating-point values, output precision control must be done by assignment to intermediate variables declared with the traditional fixed-point decimal formats.

The COBOL Standard limit of 30 decimal digits in fixed-point numbers means that it is difficult to output large and small floating-point numbers with fewer digits than the default.

The Matula rules for correct round-trip conversions call for 9 and 17 decimal digits in the IEEE 754 32-bit and 64-bit binary formats. The `DISPLAY` statement compiled by GnuCOBOL instead outputs 8 and 16 digits.

Output from the `DISPLAY` verb can be redirected from its default stream with the `UPON` qualifier, like this:

```
DISPLAY "This goes in a separate file" UPON SYSPUNCH
```

The name `SYSPUNCH` is one of a small number of standard names, and with GnuCOBOL, it corresponds to the setting of an environment variable whose value is a pathname, used at run time like this:

```
% cobc -x myprog.cob
% env COB_DISPLAY_PUNCH_FILE=/path/to/punch/file ./myprog
```

K.6 IEEE 754 arithmetic deviations in COBOL

Test programs compiled with the GnuCOBOL compiler and run reveal surprises in the handling of computed IEEE 754 Infinity and NaN values: both are mapped to zero! A NaN *can* be returned from a C function, and printed correctly by a `DISPLAY` statement.

The sign of zero is lost in both 32-bit and 64-bit binary, and 64-bit and 128-bit decimal, floating-point values. However, tests show that a negative zero can be returned from a C function, and printed with the proper sign by a `DISPLAY` statement, but any arithmetic in COBOL with that value quickly loses the sign.

Tests of underflow in the two binary formats, starting with a value 1.0 and repeatedly halving it until the result is 0.0, show that subnormals are supported, and the smallest nonzeros reached are the correct values 2^{-149} and 2^{-1073}.

Similar underflow tests with decimal arithmetic, however, show unexpected results: when the subnormal underflow thresholds of 10^{-398} and 10^{-6176} are reached, the next iteration that reduces the test value by 10 does not underflow to zero, but instead to the *smallest subnormal*. The code then had an infinite loop until an additional exit test on the power-of-ten loop counter was added.

The irregularities of floating-point arithmetic in COBOL suggest that use of floating-point variables should be carefully restricted to working only with values that are known to be finite, and where all intermediate results are also provably finite.

K.7 Interfacing COBOL to C

Documentation of several modern COBOL compilers shows that their designers have already responded to the need to communicate with code written in other languages, including C, Fortran, and Java.

Issues that must be addressed include the mapping of fundamental character and numeric data types in the languages, how character strings are passed, and what character sets are supported. The GnuCOBOL compiler used for the interface illustrated in this Appendix makes this reasonably easy, although the COBOL language makes the calling sequence unnecessarily verbose, and error prone.

The first issue is that most functions in the C mathematical library return their results as function values, whereas COBOL expects results to be supplied via the argument list, with any function value either discarded, or else expected to be a small integer indicating success or failure.

The second issue is that COBOL has nothing like C's header files that can be used to define argument prototypes that permit compile-time checking and, if needed, type coercion of argument types and return values.

A third issue is that, because of the lack of prototypes, C function names must be looked up dynamically.

Here is a short C function that makes error function computations accessible to COBOL programs, which lack that function in the COBOL run-time library:

```
#include <math.h>
void
myerf(double * result, double x)
{
    *result = erf(x);
}
```

A COBOL program that needs that function must call it with code similar to that in our only complete example of a main program:

```
      * Sample COBOL program to call erf() in C interface code

       IDENTIFICATION   DIVISION.
       PROGRAM-ID.      erf-comp-2.

       DATA             DIVISION.

       WORKING-STORAGE  SECTION.
       01 Y        COMP-2.
       01 RESULT   COMP-2.

       PROCEDURE        DIVISION.
       MOVE -1.5 TO Y

       CALL "myerf" USING
           BY REFERENCE RESULT
           BY VALUE Y
           RETURNING NOTHING

       DISPLAY "erf(-1.5) = ", RESULT
       DISPLAY "Expect:     -0.96610514647531072706697626164594785"
       STOP RUN.
```

The C function name `myerf` must be supplied as a character string and is found dynamically at run time. The `BY REFERENCE` qualifier can be omitted, because it is the default in COBOL.

A sample compilation and run of the program looks like this:

```
% cobc -x erf-comp-2.cob myerf.c -lm && ./erf-comp-2
erf(-1.5) = -0.9661051464753108
Expect:     -0.96610514647531072706697626164594785
```

TO DO: Prepare an interface library with wrappers for binary and decimal formats that are recognized in COBOL, using a library name like `-lcobolmcw` and function names like `mcw_erf`, with the result as the first argument.