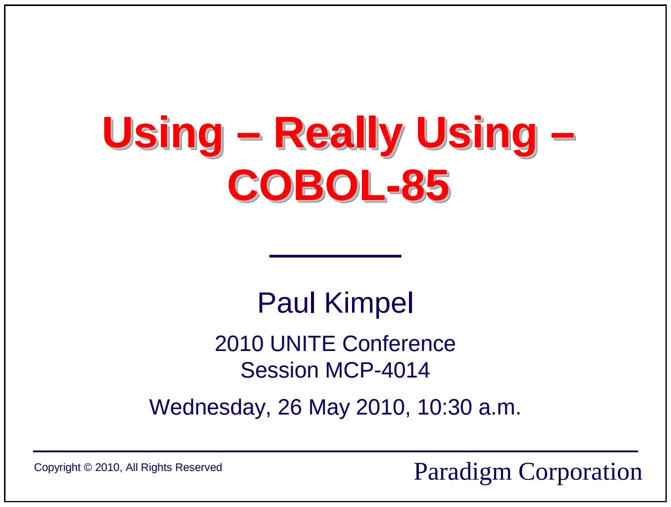


Using – Really Using – COBOL-85

Paul Kimpel

2010 UNITE Conference
Session MCP-4014

Wednesday, 26 May 2010, 10:30 a.m.

Copyright © 2010, All Rights Reserved

Paradigm Corporation

# Using – Really Using – COBOL-85

2010 UNITE Conference
Baltimore, Maryland

Session MCP-4014

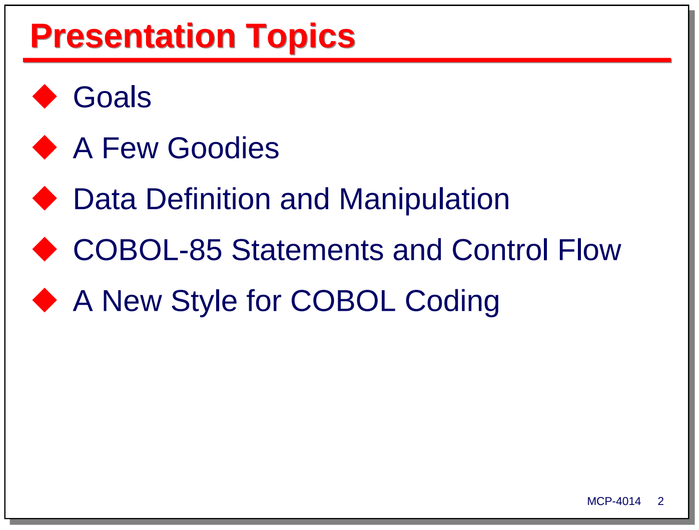
Wednesday, 26 May 2010, 10:30 a.m.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



## Presentation Topics

- Goals
- A Few Goodies
- Data Definition and Manipulation
- COBOL-85 Statements and Control Flow
- A New Style for COBOL Coding

MCP-4014 2

If you're like me, your site licensed COBOL-85 and you didn't miss a beat. You kept coding in COBOL-74 like you always had, just compiling it with the C85 compiler. Unisys has done a wonderful job making the C85 compiler upward compatible with C74 – perhaps too good a job – there just was not much incentive to use the newer features in C85.

A couple of years ago I started trying to use the new COBOL-85 language features in earnest. The initial impetus for this was writing a medium-size program to exercise the DMSQL Call Level Interface (CLI) in a general way.[1] The more of the new language constructs I used, the more I started to like it, and I found the style of coding I had developed over the years for COBOL-68 and -74 starting to change radically.

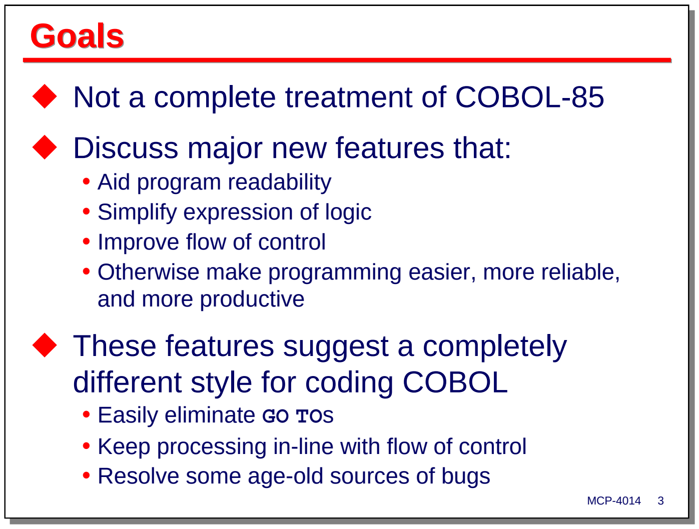
There have been UNITE presentations on the features of COBOL-85 before, particularly ones by Bob Morrow and Edward Reid at the 2002 conference (cited in the references at the end of this presentation), but I have been so taken with the new style of coding that has evolved, that I decided to approach the subject from that perspective. In order to discuss that new style, of course, we have to talk about the new[2] features in COBOL-85 that enable that style. The first, and largest, part of the presentation will be taken up with a discussion of those features, and you will see some of the new style in the accompanying examples. I'll talk about some general features, features for data definition and manipulation, and a really significant set of new features involving Procedure Division statements and program control flow.

The last dozen slides will discuss the style that has evolved for me, along with some general recommendations on how to apply COBOL-85 features to the writing of new code.

---

[1] See http://www.digm.com/UNITE/2008/ and particularly the file **`SOURCE\UNITE\DMSQL\QUERY\HARNESS.c85_m`** contained in http://www.digm.com/UNITE/2008/2008-MCP-4032-Resources.zip

[2] I'm not sure that "new" is the right word here, as most of this stuff has been around and available on the MCP for almost 25 years.



## Goals

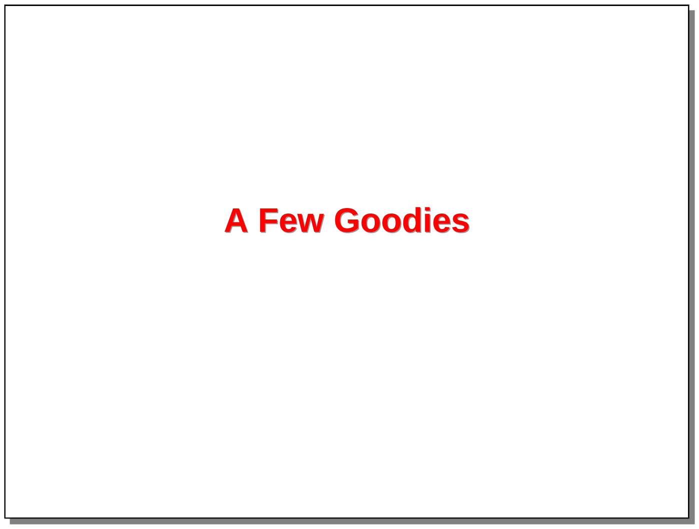
- Not a complete treatment of COBOL-85
- Discuss major new features that:
  - Aid program readability
  - Simplify expression of logic
  - Improve flow of control
  - Otherwise make programming easier, more reliable, and more productive
- These features suggest a completely different style for coding COBOL
  - Easily eliminate `GO TO`s
  - Keep processing in-line with flow of control
  - Resolve some age-old sources of bugs

MCP-4014 3

The goal in this presentation is not to do a complete overview of all of the features of COBOL-85. Instead, I want to discuss the major new features that aid program readability, simplify program logic, improve control flow, and otherwise make COBOL programs easier to generate, more reliable, and more productive.

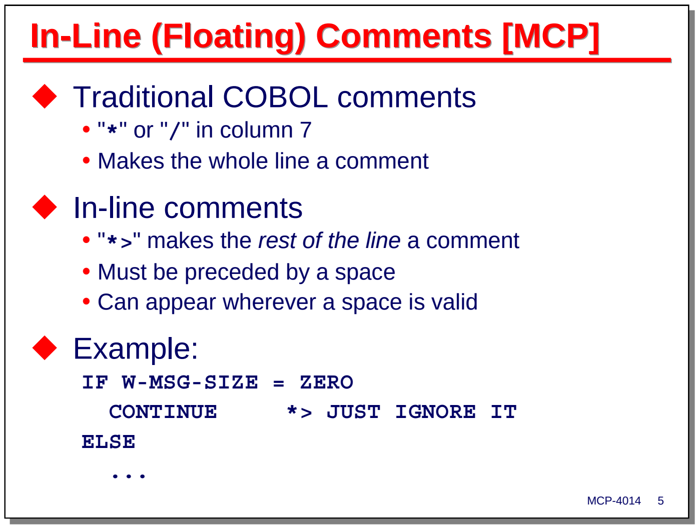
It is from these major features that my new style has evolved. Most of it is in the area of program control flow – finally having a good way to eliminate `GO TO`s, taking advantage of features for in-line coding within the control flow (I'll explain what this means later), and hopefully resolving some age-old sources of classic COBOL coding bugs.

Totally resolving sources of bugs simply with language features and coding style is too much to hope for, and I'll discuss at least one major gotcha with the new control flow features.



## A Few Goodies

First, let's start with a discussion of a few nice, but miscellaneous features in COBOL-85. Some of these are specific to the Unisys MCP environment rather than standard COBOL-85. I've tried to identify those by indicating "[MCP]" on the slides.



## In-Line (Floating) Comments [MCP]

- Traditional COBOL comments
  - "`*`" or "`/`" in column 7
  - Makes the whole line a comment
- In-line comments
  - "`*>`" makes the *rest of the line* a comment
  - Must be preceded by a space
  - Can appear wherever a space is valid
- Example:

```
IF W-MSG-SIZE = ZERO
  CONTINUE     *> JUST IGNORE IT
ELSE
  ...
```

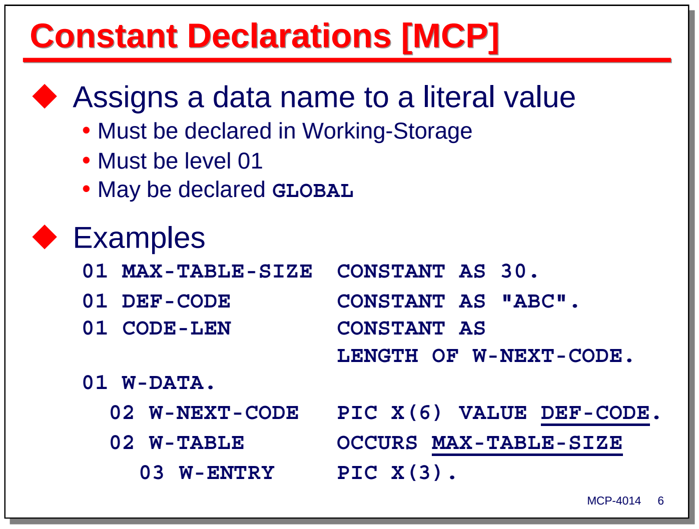
MCP-4014 5

Traditionally COBOL has had one way to indicate comments in a program,[1] and that is by placing an asterisk ("`*`") or slash ("`/`") in column 7 of the source record.

COBOL-2002 introduced a new method, similar to that found in many other languages, called the in-line (or floating) comment, which is signaled by the "`*>`" character pair. Unisys implemented this feature for the MCP COBL-85 compiler in MCP 8.0. The asterisk, unless in column 7, must be preceded by a space, and the comment can start anywhere a space is valid in the syntax.

I like this feature, and am using it more and more in my programs.

[1] Well, aside from some areas of the **`IDENTIFICATION DIVISION`** that are treated as comments, and the old **`NOTE`** sentence, which is long gone.



## Constant Declarations [MCP]

- Assigns a data name to a literal value
  - Must be declared in Working-Storage
  - Must be level 01
  - May be declared **GLOBAL**
- Examples

```
01 MAX-TABLE-SIZE  CONSTANT AS 30.
01 DEF-CODE        CONSTANT AS "ABC".
01 CODE-LEN        CONSTANT AS
                   LENGTH OF W-NEXT-CODE.
01 W-DATA.
  02 W-NEXT-CODE   PIC X(6) VALUE DEF-CODE.
  02 W-TABLE       OCCURS MAX-TABLE-SIZE
    03 W-ENTRY     PIC X(3).
```

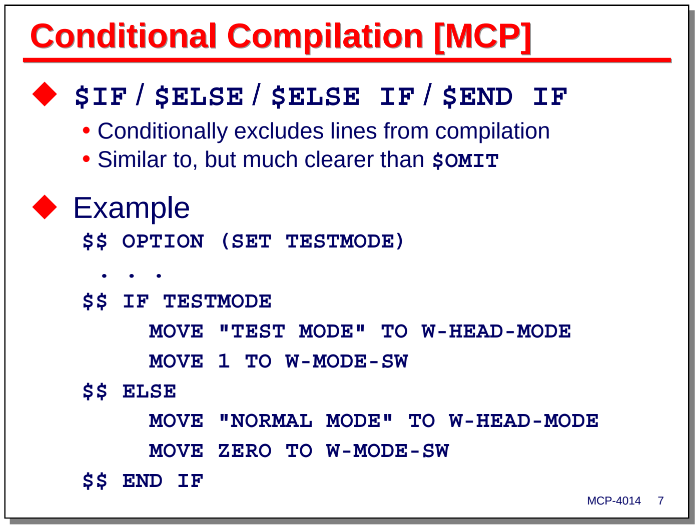
MCP-4014 6

Constant declarations are another MCP-specific feature introduced in MCP 8.0. I don't know what the genesis of this feature was, but I've wanted something like this for a long time and I now use it a lot.

In the simplest case, a constant declaration associates a data name with a literal value, similar to a **DEFINE** in Algol or a **CONSTANT** declaration in WFL. The declaration must be in Working-Storage and must be at level 01. I usually align the **CONSTANT AS** in column 40 (with **PIC** clauses). The literal value can be any of the forms that COBOL-85 supports, including hex and floating-point literals.

The data name can be used anywhere the corresponding literal is valid (except in picture strings, as the entire picture string is considered to be one syntax element). In particular the data name can be used in a **VALUE** clause, an **OCCURS** clause, or as a sending item in a Procedure Division statement.

You can also declare a constant as the **LENGTH OF** or **BYTE-LENGTH OF** some data name. The compiler will evaluate the constant name as the number of characters or bytes, respectively, contained in the specified data name. For the 8-bit character sets used with Western languages, **LENGTH** and **BYTE-LENGTH** are equivalent.



## Conditional Compilation [MCP]

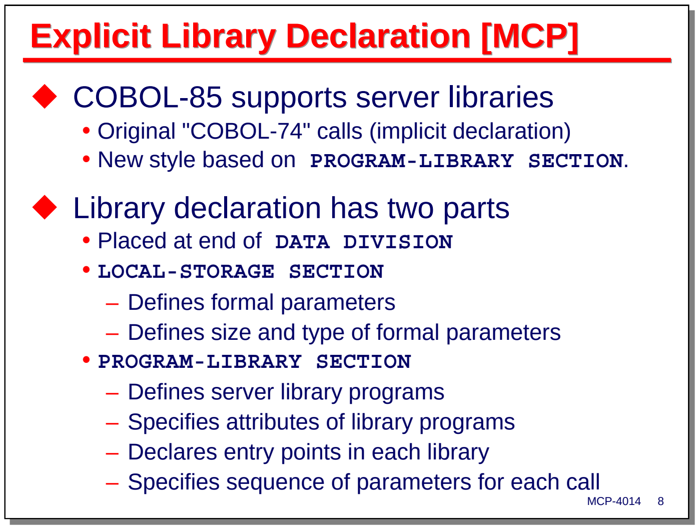
- **`$IF`** / **`$ELSE`** / **`$ELSE IF`** / **`$END IF`**
  - Conditionally excludes lines from compilation
  - Similar to, but much clearer than **`$OMIT`**
- Example

```
$$ OPTION (SET TESTMODE)
  .  .  .
$$ IF TESTMODE
     MOVE "TEST MODE" TO W-HEAD-MODE
     MOVE 1 TO W-MODE-SW
$$ ELSE
     MOVE "NORMAL MODE" TO W-HEAD-MODE
     MOVE ZERO TO W-MODE-SW
$$ END IF
```

MCP-4014 7

Yet another MCP-specific feature of COBOL-85 is a refinement of the **`$OMIT`** conditional compilation feature in COBOL-74 and several other MCP languages. **`$IF`** control statements allow you to define areas of the program that will or will not be compiled based on the value of a compile-time option (or a Boolean expression composed of compile-time options).

This is a lot easier to read and understand than the **`$OMIT`** approach, which requires you to unravel negative logic to determine whether the affected code gets compiled or not.



## Explicit Library Declaration [MCP]

- COBOL-85 supports server libraries
  - Original "COBOL-74" calls (implicit declaration)
  - New style based on **`PROGRAM-LIBRARY SECTION`**.
- Library declaration has two parts
  - Placed at end of **`DATA DIVISION`**
  - **`LOCAL-STORAGE SECTION`**
    - Defines formal parameters
    - Defines size and type of formal parameters
  - **`PROGRAM-LIBRARY SECTION`**
    - Defines server library programs
    - Specifies attributes of library programs
    - Declares entry points in each library
    - Specifies sequence of parameters for each call

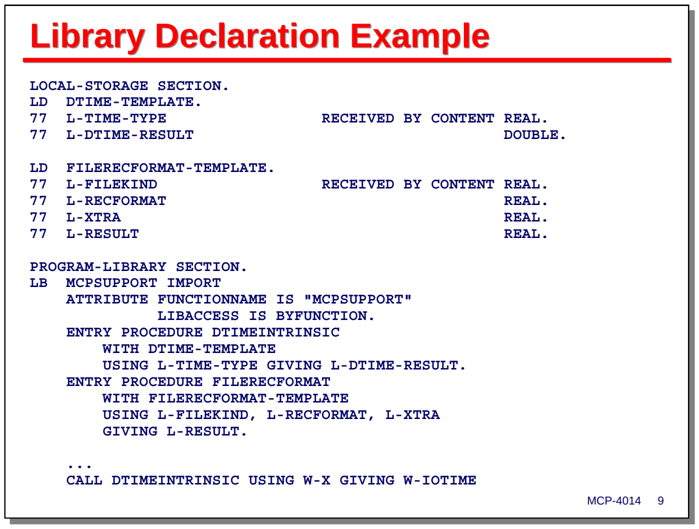
MCP-4014 8

When calling a server library routine, the compiler must build a system data structure called a library template that describes the routine's parameters and their data types. For COBOL-74 calls on server libraries, there is no declaration of the library or its routines. The compiler must deduce the parameter data types from the call. It does a good job of this, but all parameters must be passed by reference.

COBOL-85 fully supports the COBOL-74 style of library call, but also implements a new method that involves declaring the library, its entry point routines, and the sequence and types of the parameters for those routines. The syntax of the **`CALL`** verb is also slightly different. This method is based on two sections that go at the end of the Data Division, the **`LOCAL-STORAGE`** section and the **`PROGRAM-LIBRARY`** section.

**`LOCAL-STORAGE`** serves to associate data types with parameter names. It consists of a series of **`LD`** sections which contain 01- and 77-level declarations. These declarations do not allocate memory in the program, they merely provide metadata to describe the library entry point routine parameters.

**`PROGRAM-LIBRARY`** declares one or more server libraries, the entry point routines for each library, and the parameter sequences for each entry point. This section can also contain attributes for the libraries and alias names for the entry points. The overall capability is very similar to the **`LIBRARY`** declaration in Algol.



## Library Declaration Example

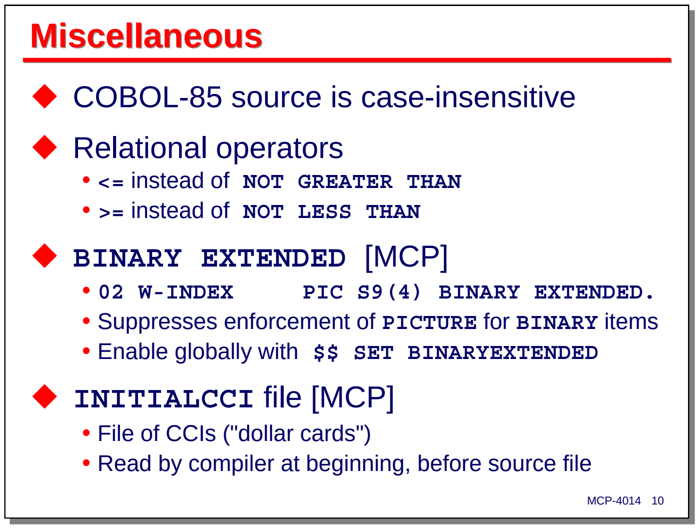
```
LOCAL-STORAGE SECTION.
LD  DTIME-TEMPLATE.
77  L-TIME-TYPE                 RECEIVED BY CONTENT REAL.
77  L-DTIME-RESULT                                  DOUBLE.

LD  FILERECFORMAT-TEMPLATE.
77  L-FILEKIND                  RECEIVED BY CONTENT REAL.
77  L-RECFORMAT                                     REAL.
77  L-XTRA                                          REAL.
77  L-RESULT                                        REAL.

PROGRAM-LIBRARY SECTION.
LB  MCPSUPPORT IMPORT
    ATTRIBUTE FUNCTIONNAME IS "MCPSUPPORT"
              LIBACCESS IS BYFUNCTION.
    ENTRY PROCEDURE DTIMEINTRINSIC
        WITH DTIME-TEMPLATE
        USING L-TIME-TYPE GIVING L-DTIME-RESULT.
    ENTRY PROCEDURE FILERECFORMAT
        WITH FILERECFORMAT-TEMPLATE
        USING L-FILEKIND, L-RECFORMAT, L-XTRA
        GIVING L-RESULT.

    ...
    CALL DTIMEINTRINSIC USING W-X GIVING W-IOTIME
```

MCP-4014 9

This slide shows an example of declaring two entry points for the **`MCPSUPPORT`** library. The **`LOCAL-STORAGE`** section defines the formal parameters and their data types; **`PROGRAM-LIBRARY`** defines the **`MCPSUPPORT`** library, the library's attributes, the entry points, and their parameter sequences. Note that the entry point declarations make use of the data names defined in **`LOCAL-STORAGE`**.

Also, note than when using this form of library declaration, the entry point in the **`CALL`** statement is a standard identifier – the one specified for the entry point in the **`PROGRAM-LIBRARY`** section. With a COBOL-74 library call, the entry point is a literal string of the form **`"`***procedurename* **`IN`** *libraryname***`"`** or **`"`***procedurename* **`OF`** *libraryname***`"`**.

See the *MCP System Interfaces Programming Reference Manual* (8600 2029) if you are interested in learning about the two routines used in this example.



## Miscellaneous

- COBOL-85 source is case-insensitive
- Relational operators
  - `<=` instead of `NOT GREATER THAN`
  - `>=` instead of `NOT LESS THAN`
- `BINARY EXTENDED` [MCP]
  - `02 W-INDEX      PIC S9(4) BINARY EXTENDED.`
  - Suppresses enforcement of `PICTURE` for `BINARY` items
  - Enable globally with `$$ SET BINARYEXTENDED`
- `INITIALCCI` file [MCP]
  - File of CCIs ("dollar cards")
  - Read by compiler at beginning, before source file

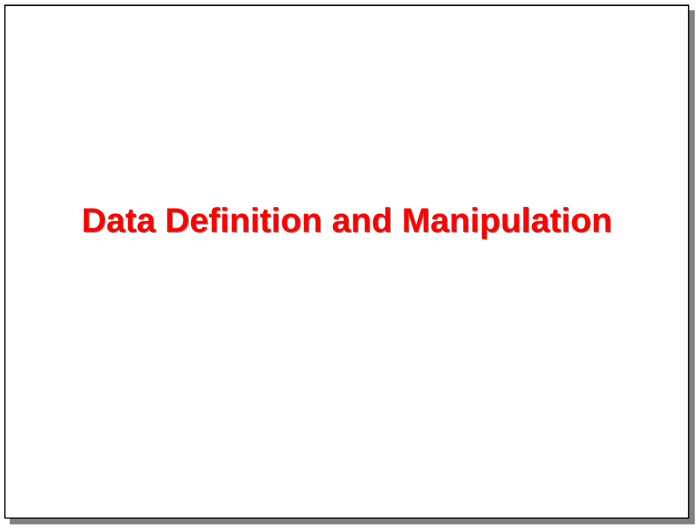
MCP-4014 10

Here are a few miscellaneous miscellaneous goodies in COBOL-85:

- COBOL-85 source programs are case-insensitive. You can now write programs in mixed case. "**MOVE**", "**move**", and "**Move**" all mean the same thing.
- Long overdue, you can now use "**<=**" instead of the excruciating "**NOT GREATER THAN**" (and the slightly less excruciating "**NOT >**"). Similarly you can use "**>=**" instead of "**NOT LESS THAN**" or "**NOT <**".
- MCP COBOL-85 supports **USAGE BINARY EXTENDED**. This allows the compiler to treat **USAGE BINARY** items as true hardware integers, ignoring the picture clause. Without the **EXTENDED** specification, moving a value to a **PIC S9(4)** data item forces the compiler to emit code to divide the value by 10,000 and store the remainder in the data item, thereby enforcing the picture clause, but destroying most, if not all, of the efficiency inherent in using binary integers. You can cause all **BINARY** items in the program to be treated as if **EXTENDED** had been specified by setting the **$BINARYEXTENDED** compiler option.
- MCP COBOL-85 has had a really nice feature since the beginning that can be used to standardize compiler options for a project or across a site. If you create a file of compiler options and file equate it to the compiler's **INITIALCCI** file, the compiler will read that file and apply the options to the compilation run before reading the source file to be compiled. The **INITIALCCI** file's default title is also **INITIALCCI**, so for most sites (and depending on how family substitution works at the site), creating a file with the title ***INITIALCCI ON DISK** will cause the options in that file to be applied by default to all COBOL-85 compilations at that site. The file can include specifiers that control whether the options are applied for all compilations, only for batch, or only for interactive compilations. See the documentation in the COBOL-85 reference manual for details.

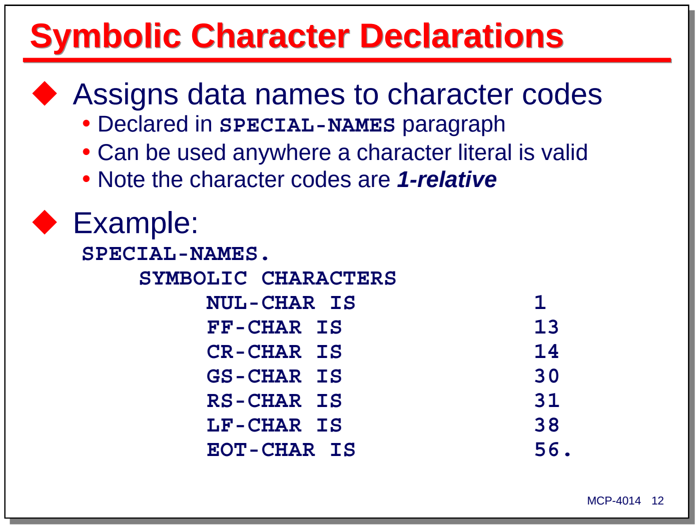
Due to space and time limitations, there were a few goodies that did not make the cut for this section of the presentation. You can read about them in the COBOL-85 reference manual:

- The keyword **FILLER** is no longer required. You can now write "**10 PIC X(3)**" in the Data Division and the compiler will consider that to be a filler item.
- COBOL-85 supports epilog procedures, which are declared in the **DECLARATIVES** section of the Procedure Division. An epilog procedure will be executed unconditionally at the termination of a program, whether the program terminates normally or not. It is typically used to clean up after an abort, closing files, freeing locks, etc.
- The MCP library **CALL** statement has an **ON EXCEPTION** clause that will trap linkage and parameter mismatches that would otherwise cause the caller to abort. Information about the failed linkage is available in the **LINKLIBRARY-RESULT** special register.



## Data Definition and Manipulation

That concludes the discussion of miscellaneous goodies in COBOL-85. Next we will discuss features for data definition and manipulation.



## Symbolic Character Declarations

- Assigns data names to character codes
  - Declared in **`SPECIAL-NAMES`** paragraph
  - Can be used anywhere a character literal is valid
  - Note the character codes are ***1-relative***
- Example:

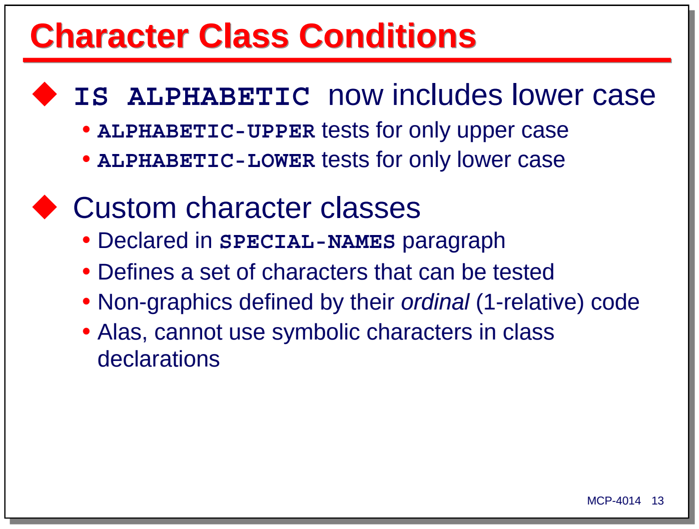
```
SPECIAL-NAMES.
    SYMBOLIC CHARACTERS
         NUL-CHAR IS             1
         FF-CHAR IS              13
         CR-CHAR IS              14
         GS-CHAR IS              30
         RS-CHAR IS              31
         LF-CHAR IS              38
         EOT-CHAR IS             56.
```

MCP-4014 12

A very nice, and from my experience, under-utilized feature of COBOL-85 is symbolic character declarations. These are placed in the **`SPECIAL-NAMES`** paragraph of the Configuration Section.

Once declared, the data names in this section can be used anywhere a character or hex literal can be used. These are generally more efficient than declaring special characters as items in Working-Storage, as the compiler can generate literal calls for the character values rather than having to load them from standard data areas.

**Gotcha Alert!** The original designers of COBOL never got the email that numbers start at zero, not one. Never mind that there was no such thing as email at the time. The character codes are specified as the character's *ordinal* position in the character set. Ordinal means one-relative (*__o__rdinal* ⇒ *__o__ne*, for those of you, like me, who have trouble remembering this). An EBCDIC **`NUL`** character has a *value* of zero, but an *ordinal position* of 1.



## Character Class Conditions

- **`IS ALPHABETIC`** now includes lower case
  - **`ALPHABETIC-UPPER`** tests for only upper case
  - **`ALPHABETIC-LOWER`** tests for only lower case
- Custom character classes
  - Declared in **`SPECIAL-NAMES`** paragraph
  - Defines a set of characters that can be tested
  - Non-graphics defined by their *ordinal* (1-relative) code
  - Alas, cannot use symbolic characters in class declarations

MCP-4014 13

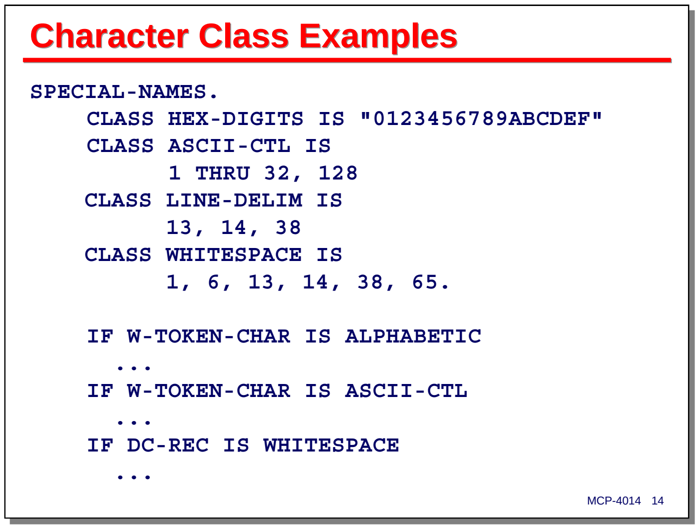
COBOL has traditionally had two intrinsic character class conditions, **`ALPHABETIC`** and **`NUMERIC`**, which are evaluated using the **`IS`** conditional operator (e.g., **`IF W-COUNTER IS NUMERIC`**).

Since COBOL-85 recognizes the existence of lower-case characters, the **`ALPHABETIC`** class now includes both upper- and lower-case letters. There are two new classes, **`ALPHABETIC-UPPER`** and **`ALPHABETIC-LOWER`**, that will test for all-upper or all-lower case letters.

COBOL-85 also allows you to define custom character classes (what Algol refers to as a **`TRUTHSET`**) and test the contents of data items against those, using the same **`IS`** operator. These custom character classes are declared in the **`SPECIAL-NAMES`** paragraph of the Configuration Section.

Non-graphic (i.e., "special") characters must be defined by a numeric code, and that numeric code, as with symbolic characters, must be specified as the ordinal (one-relative) position of the character in its character set.

Having gone to all of the trouble to create symbolic character declarations, you would think the designers of COBOL-85 would have allowed those symbolic characters to be used in the declaration of custom character classes, but no, they can't.

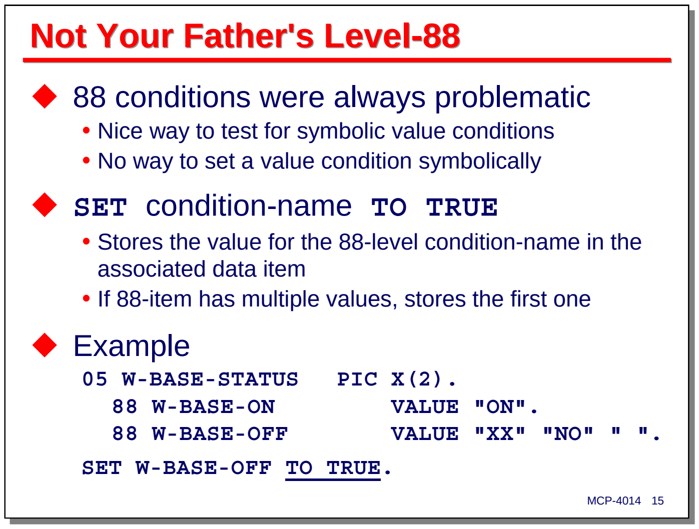


## Character Class Examples

```
SPECIAL-NAMES.
    CLASS HEX-DIGITS IS "0123456789ABCDEF"
    CLASS ASCII-CTL IS
          1 THRU 32, 128
    CLASS LINE-DELIM IS
          13, 14, 38
    CLASS WHITESPACE IS
          1, 6, 13, 14, 38, 65.

    IF W-TOKEN-CHAR IS ALPHABETIC
      ...
    IF W-TOKEN-CHAR IS ASCII-CTL
      ...
    IF DC-REC IS WHITESPACE
      ...
```

MCP-4014 14

This slide shows an example of several character class declarations and their use with class conditions in **`IF`** statements. Graphic (printable) characters can be used literally to define the class; non-graphic characters must be specified by their ordinal position in the character set.



## Not Your Father's Level-88

- 88 conditions were always problematic
  - Nice way to test for symbolic value conditions
  - No way to set a value condition symbolically
- **SET** condition-name **TO TRUE**
  - Stores the value for the 88-level condition-name in the associated data item
  - If 88-item has multiple values, stores the first one
- Example

```
05 W-BASE-STATUS   PIC X(2).
  88 W-BASE-ON         VALUE "ON".
  88 W-BASE-OFF        VALUE "XX" "NO" " ".
SET W-BASE-OFF TO TRUE.
```

MCP-4014 15

88-level condition names were a great idea in the original COBOL design, but they have always been problematic to use. They provide a very nice way to test a data item symbolically for a value or set of values, but the feature wasn't symmetric – there was no way to symbolically set a data name to one of its conditions.

COBOL-85 introduces an elegant solution to this problem using a variation of the **SET** verb. You can now **SET** an 88-level condition name to **TRUE**. This causes the compiler to store the value associated with the 88-level name in the associated data variable. If the 88-level has multiple values, the compiler stores the first declared value.

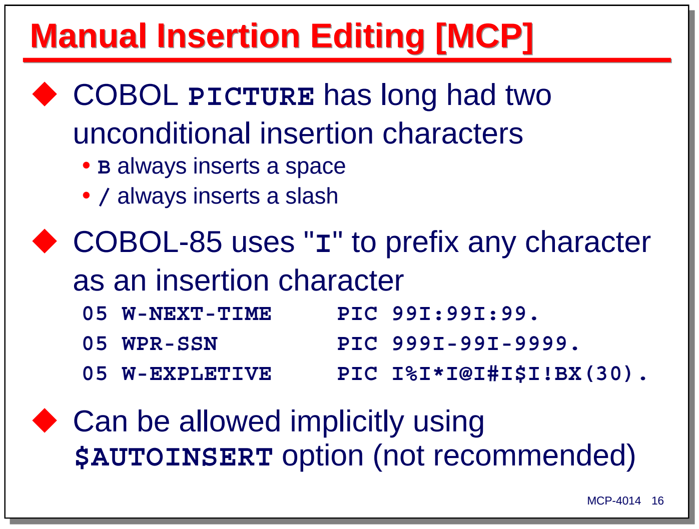
In the example on the slide,

```
SET W-BASE-OFF TO TRUE.
```

is equivalent to

```
MOVE "XX" TO W-BASE-STATUS.
```

Note that you cannot set an 88-level condition name to **FALSE**.



## Manual Insertion Editing [MCP]

- COBOL **PICTURE** has long had two unconditional insertion characters
  - **B** always inserts a space
  - / always inserts a slash
- COBOL-85 uses "**I**" to prefix any character as an insertion character

```
05 W-NEXT-TIME     PIC 99I:99I:99.
05 WPR-SSN         PIC 999I-99I-9999.
05 W-EXPLETIVE     PIC I%I*I@I#I$I!BX(30).
```

- Can be allowed implicitly using **$AUTOINSERT** option (not recommended)

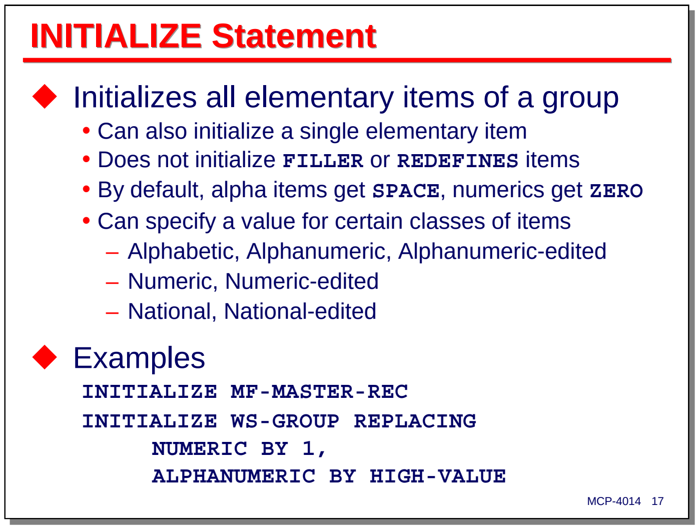
MCP-4014 16

The concept of **PICTURE** clauses was another great idea in the original COBOL design. Pictures format data, and part of their capability is to insert additional characters into the sending value in the process of formatting the receiving value.

**PICTURE** has long had the ability to unconditionally insert two characters into the receiving value, by specifying "**B**" (to insert a space) and "**/**" (to insert a slash). Earlier MCP compilers were quite liberal in interpreting picture strings, and generally allowed any character that was not otherwise defined as a picture character to be treated as an unconditional insertion character, similar to the "**/**". The actual behavior was highly context-sensitive and a little difficult to predict in some cases.

MCP COBOL-85 provides a much cleaner, if still non-standard, way to specify unconditional insertion characters in pictures. This was implemented in MCP 7.0. The character "**I**" in a picture clause signals that the next character in the string is to be inserted unconditionally in the receiving value. The nice thing about this feature is that you can insert *any* character unconditionally, and do it in a way that is unambiguous.

There is a compiler option, **$AUTOINSERT**, that will cause COBOL-85 to revert to the implied insertion behavior of older compilers, but using this is not recommended.



## INITIALIZE Statement

- Initializes all elementary items of a group
  - Can also initialize a single elementary item
  - Does not initialize **`FILLER`** or **`REDEFINES`** items
  - By default, alpha items get **`SPACE`**, numerics get **`ZERO`**
  - Can specify a value for certain classes of items
    - Alphabetic, Alphanumeric, Alphanumeric-edited
    - Numeric, Numeric-edited
    - National, National-edited
- Examples

```
INITIALIZE MF-MASTER-REC
INITIALIZE WS-GROUP REPLACING
     NUMERIC BY 1,
     ALPHANUMERIC BY HIGH-VALUE
```

MCP-4014 17

Initialization of record areas and group items is something we need to do all of the time in COBOL applications, but in the past there has not been a good way to do it. The classic method is to move **`SPACES`** to the record or group item, then individually move **`ZEROES`** to the numeric items.

Often we don't bother with the fix-up and just move the **`SPACES`**. MCP COBOL will treat a numeric **`DISPLAY`** field containing spaces as if it were zero, but probably everyone who has worked on MCP applications has run into situations where a packed-decimal field has the value **`404040`**… or **`040404`**… because spaces were moved to the record or group in which the field is contained. And we're the lucky ones – try that on an IBM 360/zSeries system and you'll get a data exception due to an invalid packed-decimal sign digit.

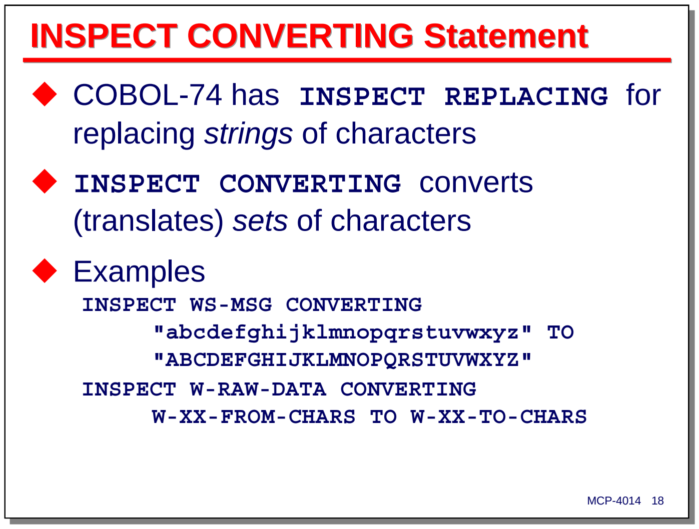
Enter the COBOL-85 **`INITIALIZE`** statement. This will initialize a record area or group item to specified values based on the classes of the elementary items in the record or group. **`FILLER`** items, and items subordinate to **`REDEFINES`** clauses, are not affected by **`INITIALIZE`**.

By default, alphabetic and alphanumeric items are initialized to spaces and numeric items are initialized to zero.

You can override the default by specifying values for seven different types of data classes:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- Numeric
- Numeric-edited
- National
- National-edited

The MCP implementation of this statement appears to be quite efficient. For small data areas (i.e., those having few subordinate items), the compiler just emits elementary moves. For areas with larger numbers of items, the compiler constructs a pre-initialized copy of the record or group area in its constant pool, and initializes the receiving area in one move.



## INSPECT CONVERTING Statement

- COBOL-74 has **`INSPECT REPLACING`** for replacing *strings* of characters
- **`INSPECT CONVERTING`** converts (translates) *sets* of characters
- Examples

```
INSPECT WS-MSG CONVERTING
     "abcdefghijklmnopqrstuvwxyz" TO
     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
INSPECT W-RAW-DATA CONVERTING
     W-XX-FROM-CHARS TO W-XX-TO-CHARS
```

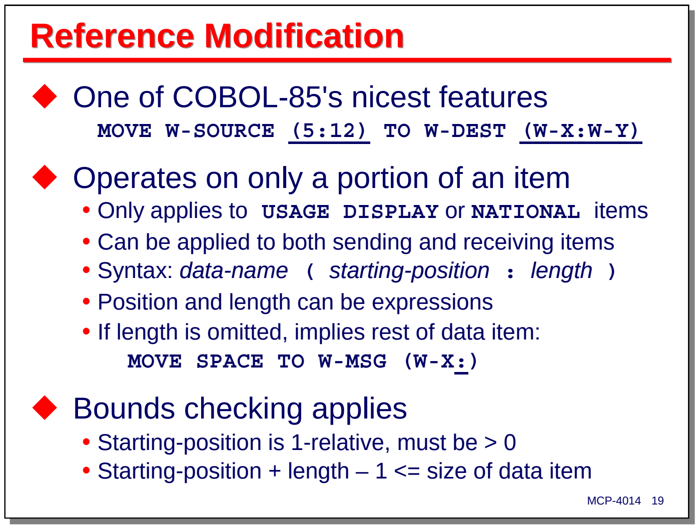
MCP-4014 18

The **`INSPECT`** statement has a number of forms, and all of those from COBOL-74 have been brought forward into COBOL-85. COBOL-85 adds a new variant of this statement, **`INSPECT CONVERTING`**.

**`INSPECT CONVERTING`** looks a lot like **`INSPECT REPLACING`**, but what it does is entirely different. Whereas the **`REPLACING`** form scans a string looking for substrings of a certain value and replaces those substrings by another substring, the **`CONVERTING`** form performs character translation.

The translation can be specified using either literals or data items. Using literals is generally more efficient, as the compiler can generate the necessary hardware translation tables at compile time and store them in the codefile. When using data items to define the translation, the tables must be constructed at run time.

The first example on the slide shows one way to up-case text. As we will see, there is another (and probably better) way to do this using intrinsic functions.



## Reference Modification

- One of COBOL-85's nicest features
  `MOVE W-SOURCE (5:12) TO W-DEST (W-X:W-Y)`
- Operates on only a portion of an item
  - Only applies to **USAGE DISPLAY** or **NATIONAL** items
  - Can be applied to both sending and receiving items
  - Syntax: *data-name* ( *starting-position* : *length* )
  - Position and length can be expressions
  - If length is omitted, implies rest of data item:
    `MOVE SPACE TO W-MSG (W-X:)`
- Bounds checking applies
  - Starting-position is 1-relative, must be > 0
  - Starting-position + length – 1 <= size of data item

MCP-4014 19

COBOL-85 also introduced the idea of "reference modification." This is simply a grand term for "substring."

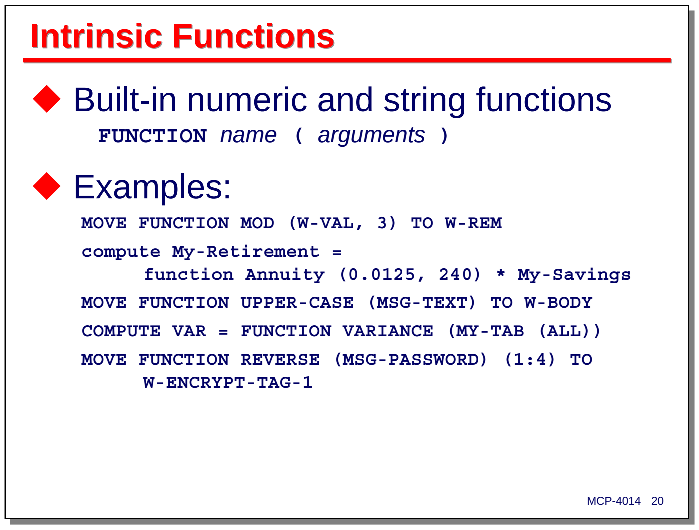
Reference modification allows you to address only a portion of a sending or receiving field. It can be used only with **USAGE DISPLAY** or **NATIONAL** items, but within that restriction can be used on both alphanumeric and numeric fields. As shown on the slide, the syntax for reference modification follows the data name with an opening parenthesis, an arithmetic expression specifying the one-relative character offset into the data, a colon ("**:**"), and a second arithmetic expression specifying the length of the substring in characters.

If the length is omitted (note that the colon must still be present), then the number of characters remaining after the starting offset is taken as an implicit length.

Reference modification can be used with subscripted data names. In that case it follows the subscripts, viz,

```
    MOVE W-TABLE-ENTRY (W-TX) (W-X:5) TO W-HOLD-AREA
```

If bounds checking is enabled (which it is by default – see the **$BOUNDS** compiler option), the compiler emits code to check that the starting position is one or greater, and that the starting position plus the length is within the valid length for the data item. Bounds violations result in an assertion failure fault.



## Intrinsic Functions

- Built-in numeric and string functions

  **FUNCTION** *name* ( *arguments* )

- Examples:

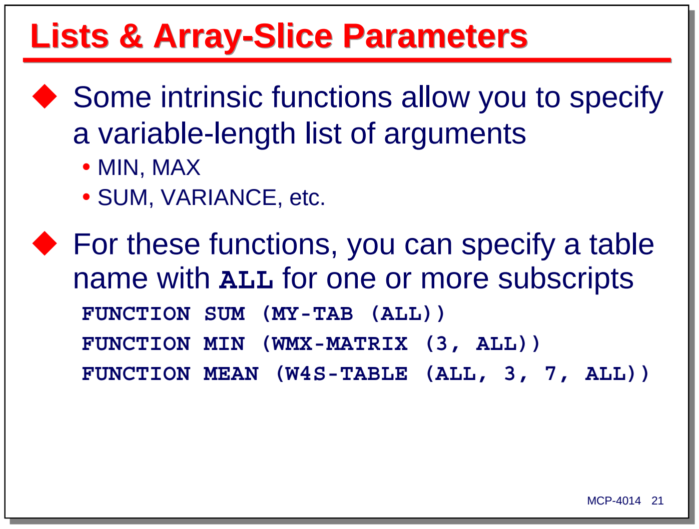
```
MOVE FUNCTION MOD (W-VAL, 3) TO W-REM
compute My-Retirement =
     function Annuity (0.0125, 240) * My-Savings
MOVE FUNCTION UPPER-CASE (MSG-TEXT) TO W-BODY
COMPUTE VAR = FUNCTION VARIANCE (MY-TAB (ALL))
MOVE FUNCTION REVERSE (MSG-PASSWORD) (1:4) TO
     W-ENCRYPT-TAG-1
```

MCP-4014 20

A follow-on to the original COBOL-85 standard implemented a rather amazing set of intrinsic functions for the language. These can be used as sending items in Procedure Division statements. The syntax begins with the keyword **FUNCTION**, followed by the name of the function, and optionally followed by arguments (parameters) in parentheses.

I'll give a brief overview of the intrinsic functions supported by the MCP compiler, arranged in the following categories:

- Numeric intrinsics
- Date intrinsics
- Character string intrinsics
- Multiple parameter/array intrinsics
- MCP-specific intrinsics



## Lists & Array-Slice Parameters

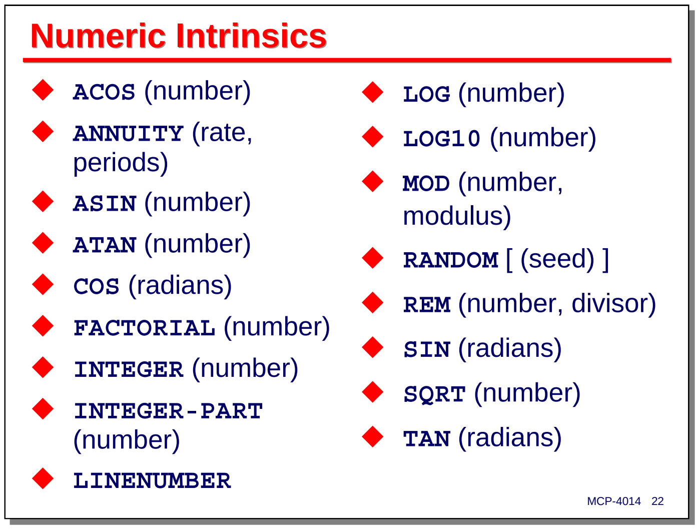
- Some intrinsic functions allow you to specify a variable-length list of arguments
  - MIN, MAX
  - SUM, VARIANCE, etc.
- For these functions, you can specify a table name with **ALL** for one or more subscripts

```
FUNCTION SUM (MY-TAB (ALL))
FUNCTION MIN (WMX-MATRIX (3, ALL))
FUNCTION MEAN (W4S-TABLE (ALL, 3, 7, ALL))
```

MCP-4014 21

Note that some intrinsic functions allow you to specify a list of arguments, and that the function is computed over the set of values represented by that list. Examples are **MIN**, **MAX**, **SUM**, and **VARIANCE**.

For these functions that accept an arbitrary number of arguments, you can also pass the name of a table or array with the keyword **ALL** specified for one or more of the dimensions. This effectively allows you to insert the entire contents of the array, or a slice of the array, into the argument list for the function – an extremely powerful capability.



## Numeric Intrinsics

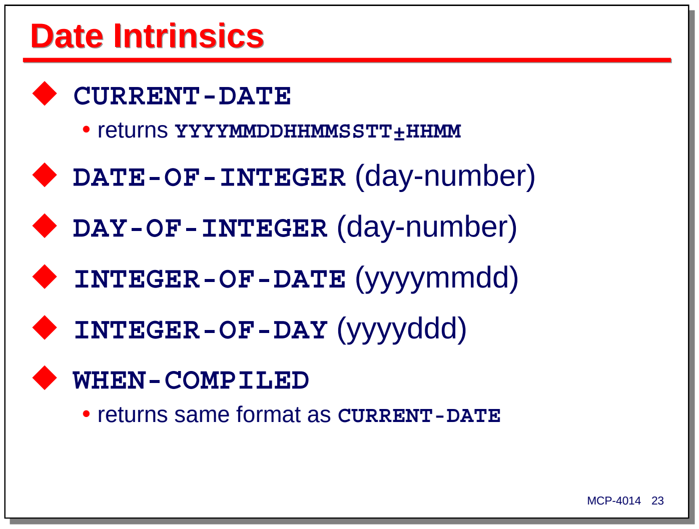
- **ACOS** (number)
- **ANNUITY** (rate, periods)
- **ASIN** (number)
- **ATAN** (number)
- **COS** (radians)
- **FACTORIAL** (number)
- **INTEGER** (number)
- **INTEGER-PART** (number)
- **LINENUMBER**
- **LOG** (number)
- **LOG10** (number)
- **MOD** (number, modulus)
- **RANDOM** [ (seed) ]
- **REM** (number, divisor)
- **SIN** (radians)
- **SQRT** (number)
- **TAN** (radians)

MCP-4014 22

COBOL-85 supports a number of numeric or arithmetic intrinsics, including the common transcendental functions, the factorial function, and mod and rem (remainder divide).

- **ANNUITY** approximates the value of an annuity for a given rate, paid at the end of each period, normalized to an initial principal value of 1.0.
- **INTEGER** returns the greatest integer value that is less than or equal to the argument value.
- **INTEGER-PART** strips any fractional part from the argument value and returns the integer part. This function is identical to **INTEGER** for non-negative argument values.
- The **LINENUMBER** intrinsic simply returns the sequence number of the source code line on which it appears.
- **RANDOM** returns a pseudo-random sequence of numbers. Calling **RANDOM** with an argument value (which must be a non-negative integer) initializes the pseudo-random sequence. Subsequent calls without an argument return successive values from the pseudo-random sequence. The result of the function is a non-negative fractional value less than one.

For details on the parameters and behavior of all the intrinsic functions, see the section devoted to them in the COBOL-85 reference manual.



## Date Intrinsics

- **CURRENT-DATE**
  - returns **YYYYMMDDHHMMSSTT±HHMM**
- **DATE-OF-INTEGER** (day-number)
- **DAY-OF-INTEGER** (day-number)
- **INTEGER-OF-DATE** (yyyymmdd)
- **INTEGER-OF-DAY** (yyyyddd)
- **WHEN-COMPILED**
  - returns same format as **CURRENT-DATE**

MCP-4014 23

Date computations have always been difficult in COBOL, since dates are not native data types in the language.

COBOL-85 supplies an intrinsic that will return the current date and time (plus the system's timezone offset) in one call using the **CURRENT-DATE** function.

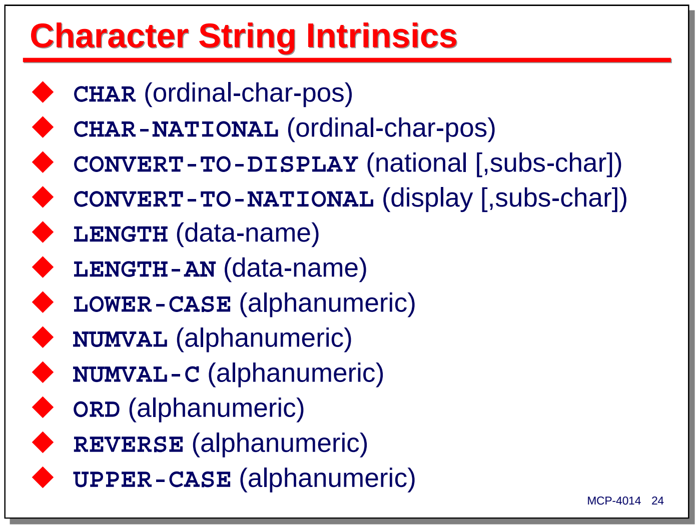
COBOL-85 also provides functions to convert between Gregorian (yyyymmdd) and Julian (yyyyddd) numeric date formats and a day number. Day number 1 (the "epoch" date) is 1 January 1601 (16010101); day number 149536 is 1 June 2010 (20100601).

Once a date is expressed as a number of days offset from an epoch date, it is easy to compute the number of days between two dates, compute a date so many dates before or after a given date, and to determine the day of the week. The following example shows how to compute a Gregorian date that is 30 days in the future.

```
77  W-DATE            PIC 9(8)   COMP.

    MOVE 20100601 TO W-DATE
    DISPLAY FUNCTION DATE-OF-INTEGER (
        FUNCTION INTEGER-OF-DATE (W-DATE) + 30)
```

This example will display a result of **20100701**.

The language also supports a **WHEN-COMPILED** function that returns the date and time the program was last compiled.



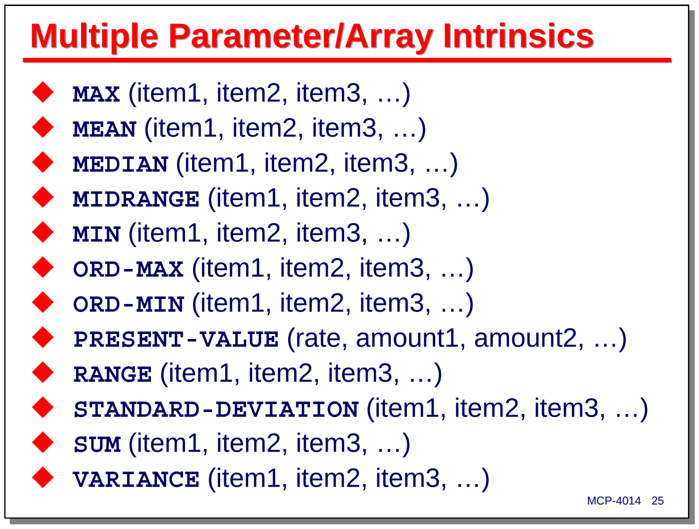
## Character String Intrinsics

- **CHAR** (ordinal-char-pos)
- **CHAR-NATIONAL** (ordinal-char-pos)
- **CONVERT-TO-DISPLAY** (national [,subs-char])
- **CONVERT-TO-NATIONAL** (display [,subs-char])
- **LENGTH** (data-name)
- **LENGTH-AN** (data-name)
- **LOWER-CASE** (alphanumeric)
- **NUMVAL** (alphanumeric)
- **NUMVAL-C** (alphanumeric)
- **ORD** (alphanumeric)
- **REVERSE** (alphanumeric)
- **UPPER-CASE** (alphanumeric)

MCP-4014 24

COBOL-85 also has a number of character string intrinsics that nicely complement the features of reference modification.

- **CHAR** and **CHAR-NATIONAL** convert the ordinal position of a character in a character set to an alphanumeric string of length one containing the character. Note that the ordinal character position is normally one greater than the character's binary value.
- **ORD** is the inverse of **CHAR**. It takes the character in the first position of its arguments and returns the corresponding ordinal position in the character set.
- **CONVERT-TO-DISPLAY** and **CONVERT-TO-NATIONAL** convert strings between 8- and 16-bit character sets.
- **LENGTH** returns the length in character positions of a data item. **LENGTH-AN** returns the length in bytes. There is no difference between the two for 8-bit Western character sets.
- **UPPER-CASE** and **LOWER-CASE** perform their eponymous translation functions.
- **NUMVAL** will convert an alphanumeric string to its numeric equivalent. **NUMVAL-C** does the same, but accepts currency formats (e.g., with dollar signs and commas). These functions appear to have limited value, as using them with invalid numeric data results in the program aborting.
- Finally, there is the **REVERSE** intrinsic, an interesting function which most modern languages support, and for which there is no known use. It simply returns the character string of its argument with the order of the characters reversed.



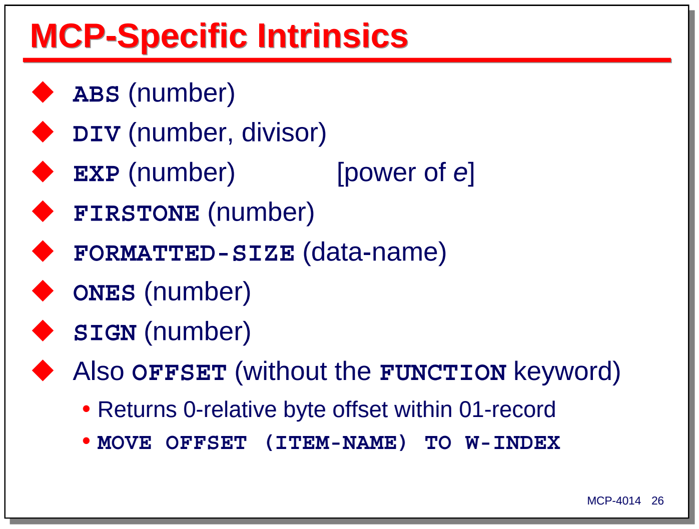
## Multiple Parameter/Array Intrinsics

- **`MAX`** (item1, item2, item3, …)
- **`MEAN`** (item1, item2, item3, …)
- **`MEDIAN`** (item1, item2, item3, …)
- **`MIDRANGE`** (item1, item2, item3, …)
- **`MIN`** (item1, item2, item3, …)
- **`ORD-MAX`** (item1, item2, item3, …)
- **`ORD-MIN`** (item1, item2, item3, …)
- **`PRESENT-VALUE`** (rate, amount1, amount2, …)
- **`RANGE`** (item1, item2, item3, …)
- **`STANDARD-DEVIATION`** (item1, item2, item3, …)
- **`SUM`** (item1, item2, item3, …)
- **`VARIANCE`** (item1, item2, item3, …)

MCP-4014 25

This slide shows the intrinsic functions which accept argument lists of arbitrary length. They will also accept arrays (tables) and array slices for arguments.

- **`MAX`**, **`MEAN`**, **`MEDIAN`**, **`MIN`**, **`STANDARD-DEVIATION`**, **`SUM`**, and **`VARIANCE`** perform their standard function across the list of argument values.
- **`ORD-MAX`** and **`ORD-MIN`** are similar to **`MAX`** and **`MIN`**, but instead of returning the maximum or minimum *value*, they return the one-relative *position* of the max or min value in the argument list.
- **`MIDRANGE`** is similar to **`MEAN`**, but returns the average of the maximum and minimum values in the argument list instead of the average across the whole list.
- **`RANGE`** returns the difference between the maximum and minimum values in the argument list.
- **`PRESENT-VALUE`** computes an approximation of the present value of a future series of period-end payments. The first argument is the discount rate specified as a decimal fraction (not a percentage). The remaining arguments in the list are the period-end payments.



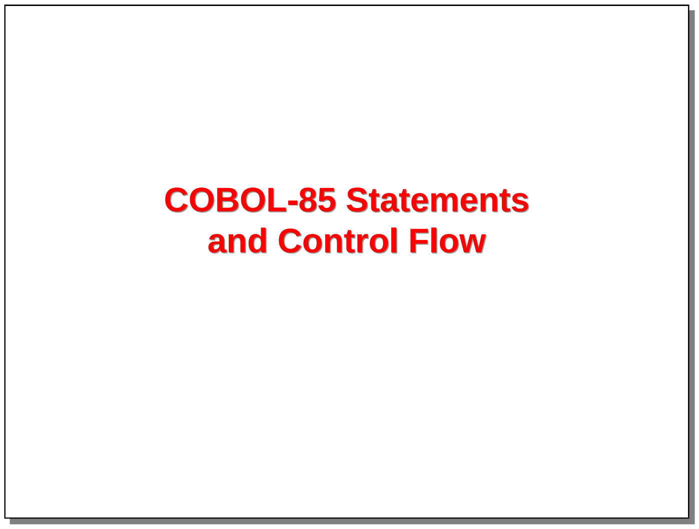
## MCP-Specific Intrinsics

- **ABS** (number)
- **DIV** (number, divisor)
- **EXP** (number) [power of *e*]
- **FIRSTONE** (number)
- **FORMATTED-SIZE** (data-name)
- **ONES** (number)
- **SIGN** (number)
- Also **OFFSET** (without the **FUNCTION** keyword)
  - Returns 0-relative byte offset within 01-record
  - **MOVE OFFSET (ITEM-NAME) TO W-INDEX**

MCP-4014 26

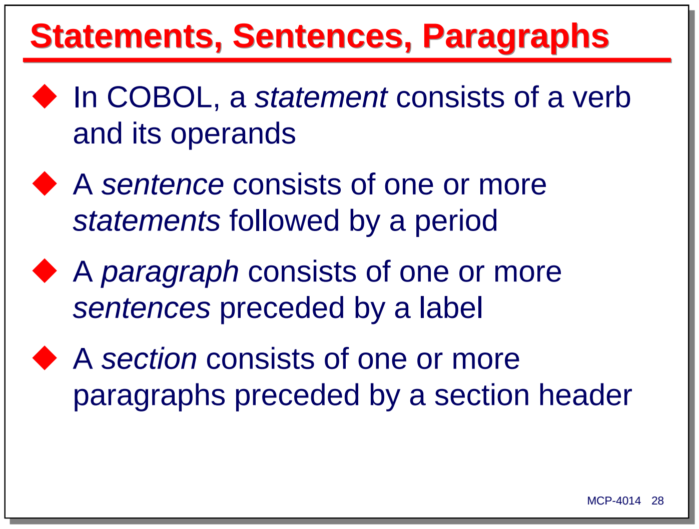
MCP COBOL-85 adds a number of additional intrinsics to the standard set.

- **ABS** strips the sign of the argument value and always returns a non-negative number
- **DIV** performs an integer division and returns an integer result
- **EXP** computes the exponential function (power of *e*)
- **FIRSTONE** implements the E-mode LOG2 instruction, which returns the bit number plus one of the first non-zero bit in the argument's numeric value. The argument is converted to a single-precision binary value prior to determining the leading bit number.
- Similarly, **ONES** implements the E-mode CBON instruction, which returns a count of the number of one bits in the argument's single- or double-precision binary value.
- **FORMATTED-SIZE** is similar to the **LENGTH-AN** function. It returns the number of bytes in the argument's data area. It is a carry-over from MCP COBOL-74.
- **SIGN** returns a value indicating the argument's sign, -1, 0, or +1.
- The **OFFSET** function is used without the **FUNCTION** keyword. It is another carry-over from MCP COBOL-74, and returns the number of bytes that precede the argument's data are in its 01 record area. Note that unlike most COBOL offsets, this one is *zero*-relative.



## COBOL-85 Statements and Control Flow

That concludes the discussion of COBOL-85 features for data definition and manipulation. Next, I'll discuss the very significant changes COBOL-85 has made in the area of program control flow.



## Statements, Sentences, Paragraphs

- In COBOL, a *statement* consists of a verb and its operands
- A *sentence* consists of one or more *statements* followed by a period
- A *paragraph* consists of one or more *sentences* preceded by a label
- A *section* consists of one or more paragraphs preceded by a section header

MCP-4014 28

Before talking about COBOL-85 statements and control flow, there are some terms that we need to define and use consistently. These terms have always been used with COBOL, but in the following discussion, the distinction between a *statement* and and *sentence* is particularly important. Note, for example, that

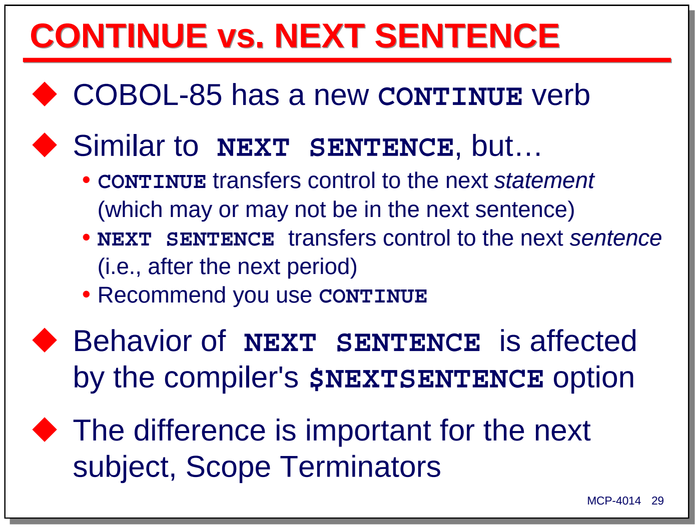
```
    MOVE A TO B
```

is a statement, while

```
    MOVE A TO B.
```

is a sentence that contains a statement.

Oh, that COBOL period.



## CONTINUE vs. NEXT SENTENCE

- COBOL-85 has a new **CONTINUE** verb
- Similar to **NEXT SENTENCE**, but…
  - **CONTINUE** transfers control to the next *statement* (which may or may not be in the next sentence)
  - **NEXT SENTENCE** transfers control to the next *sentence* (i.e., after the next period)
  - Recommend you use **CONTINUE**
- Behavior of **NEXT SENTENCE** is affected by the compiler's **$NEXTSENTENCE** option
- The difference is important for the next subject, Scope Terminators

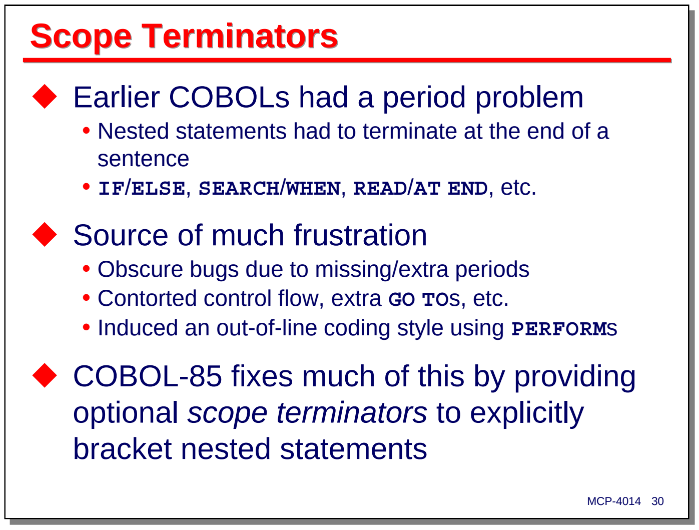
MCP-4014 29

COBOL-85 has a new verb (also a statement), **CONTINUE**. This statement does nothing. It is typically used as the object of a predicate where we want no action to take place, e.g.,

```
IF A > B
    CONTINUE
ELSE
    ...
```

**CONTINUE** is very similar to **NEXT SENTENCE**, but differs in that it transfers control to the next *statement*, while **NEXT SENTENCE** is intended to transfer control to the next *sentence* in the program. The general consensus these days seems to be that you should use **CONTINUE** and avoid **NEXT SENTENCE**, especially since the behavior of **NEXT SENTENCE** can be affected by the MCP compiler's **$NEXTSENTENCE** option.

The idea of **CONTINUE** and its difference with **NEXT SENTENCE** is important for the next discussion on Scope Terminators.



## Scope Terminators

- Earlier COBOLs had a period problem
  - Nested statements had to terminate at the end of a sentence
  - **IF**/**ELSE**, **SEARCH**/**WHEN**, **READ**/**AT END**, etc.
- Source of much frustration
  - Obscure bugs due to missing/extra periods
  - Contorted control flow, extra **GO TO**s, etc.
  - Induced an out-of-line coding style using **PERFORM**s
- COBOL-85 fixes much of this by providing optional *scope terminators* to explicitly bracket nested statements

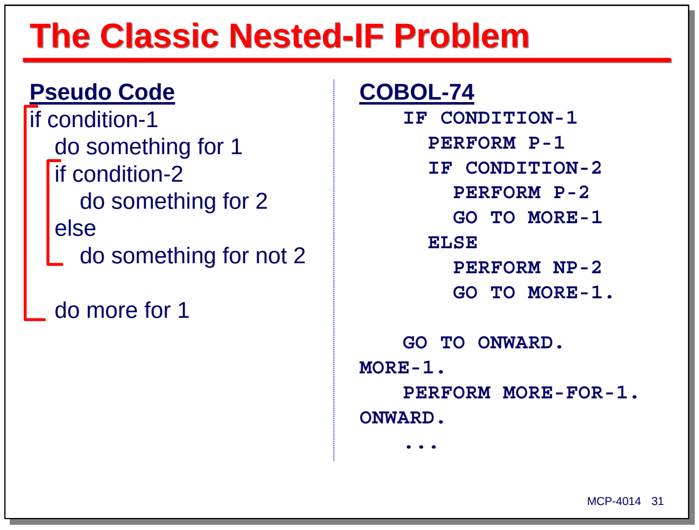
MCP-4014 30

COBOL has a problem with the period that ends a sentence, and everyone who has programmed in COBOL is no doubt painfully familiar with it. The problem is that nested statements that are the object of a predicate (e.g., the statements subordinate to an **IF** statement or an **AT END** clause) must be at the end of a sentence. Another way of stating this is that the *scope* of that **IF** or **AT END** is terminated by the next period in the source file.

This has been the source of must frustration to COBOL programmers. It is also the source of any number of obscure bugs due to missing or extra periods. It leads, in all but the simplest cases, to contorted control flow, as we write labels and **GO TO**s to branch around blocks of code that can't be nested the way they are in other, block-structured languages.

In part, using the period as a scope terminator has lead to what I call an "out-of-line" style for **GO TO**-less programming. You have probably seen this, and may have been forced to practice it. The idea is that the bodies of **IF**s and **ELSE**s, exception clauses such as **AT END**, and the like, are removed to a separate block of code and **PERFORM**ed from the predicate that they are subordinate to. Similarly, all loops are coded as **PERFORM**s of a separate block of code. Personally, I find this style to be almost as bad as coding with **GO TO**s, and often less readable.

COBOL-85 fixes much of the problem with periods by providing optional *scope terminators* for each of the statements that serve as predicates for a nested set of subordinate statements. You don't have to use scope terminators, and you can mix their use with the classic style of COBOL coding, but they are a significant (if long overdue) advance in COBOL syntax. They are also one of the main influences for a new style of COBOL coding.



## The Classic Nested-IF Problem

**Pseudo Code**

```
if condition-1
   do something for 1
   if condition-2
      do something for 2
   else
      do something for not 2

   do more for 1
```

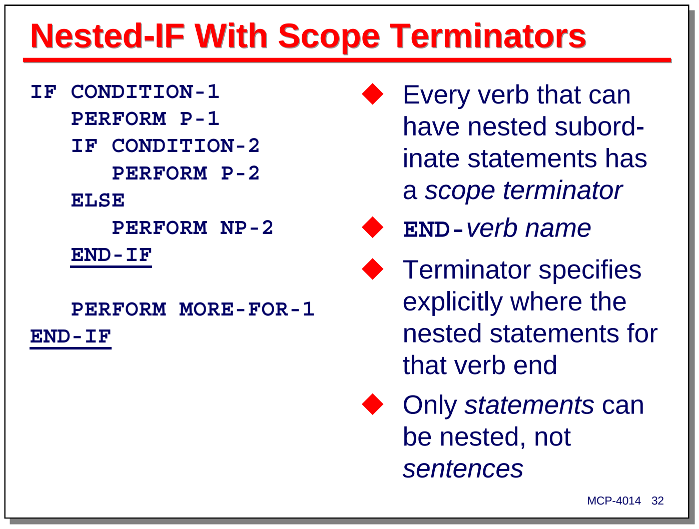
**COBOL-74**

```
    IF CONDITION-1
      PERFORM P-1
      IF CONDITION-2
        PERFORM P-2
        GO TO MORE-1
      ELSE
        PERFORM NP-2
        GO TO MORE-1.

    GO TO ONWARD.
MORE-1.
    PERFORM MORE-FOR-1.
ONWARD.
    ...
```

MCP-4014 31

To illustrate the use of scope terminators, consider the classic nested-**`IF`** problem. We test for a condition, and if that condition is true, want to execute a series of subordinate statements. Within the subordinate statements, we have another **`IF`** statement that itself has subordinate statements. The inner **`IF`** statement may or may not have an **`ELSE`** clause. Regardless of the outcome of that inner **`IF`** statement, however, we want to execute some additional code after that, but which is still conditioned by the first **`IF`** statement.

In classic COBOL, there is no way to code this by simply nesting statements under the **`IF`** statements. Alas, the indentation that makes the meaning of the pseudo code clear doesn't count in real code, unless you are programming in Python. In COBOL, you need to either repeat the original **`IF`** statement or resort to **`GO TO`**s and labels. The slide shows one common way of doing this. It's not exactly obvious that the code under the **`MORE-1`** label is actually conditioned by the first **`IF`** statement. What's worse, it becomes less obvious the farther that code is removed from the **`IF`**.



## Nested-IF With Scope Terminators

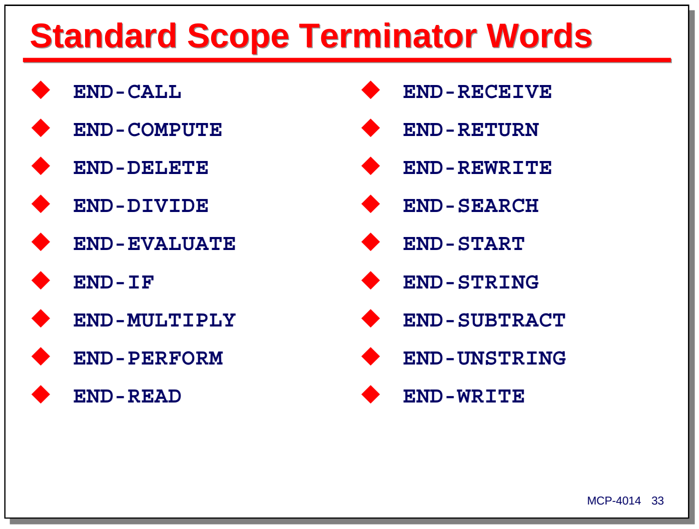
```
IF CONDITION-1
   PERFORM P-1
   IF CONDITION-2
      PERFORM P-2
   ELSE
      PERFORM NP-2
   END-IF

   PERFORM MORE-FOR-1
END-IF
```

- Every verb that can have nested subord-inate statements has a *scope terminator*
- `END-`*verb name*
- Terminator specifies explicitly where the nested statements for that verb end
- Only *statements* can be nested, not *sentences*

MCP-4014 32

This slide shows how this problem can be solved quite easily and clearly in COBOL-85 using the **`END-IF`** scope terminator keyword. **`END-IF`** tells us (and the compiler) where the scope of the immediately preceding **`IF`** statement occurs.

In COBOL-85, every verb that can have nested subordinate statements has a corresponding scope terminator keyword, formed as **`END-`***verb name*. Other languages use standard statement brackets like **`BEGIN`** and **`END`** or curly braces, **`{}`**, as scope terminators. This is just the way the designers of COBOL-85 chose to solve the problem. It may not be pretty, but it sure beats the **`FI`** and **`ESAC`** approach that ALGOL-68 tried to foist on the world.

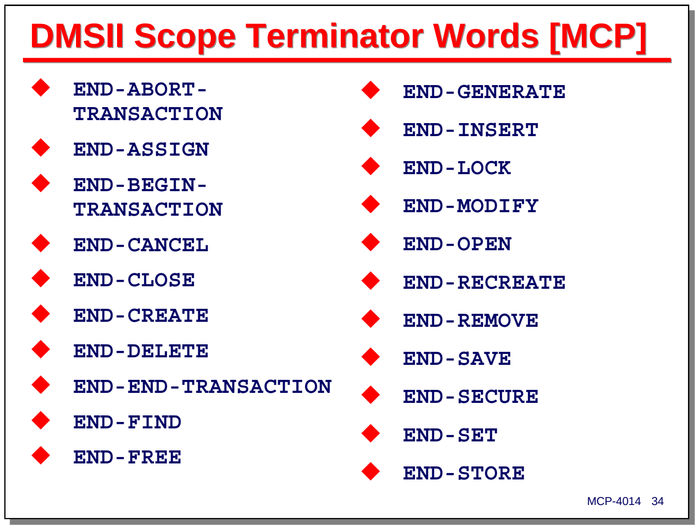


## Standard Scope Terminator Words

- END-CALL
- END-COMPUTE
- END-DELETE
- END-DIVIDE
- END-EVALUATE
- END-IF
- END-MULTIPLY
- END-PERFORM
- END-READ
- END-RECEIVE
- END-RETURN
- END-REWRITE
- END-SEARCH
- END-START
- END-STRING
- END-SUBTRACT
- END-UNSTRING
- END-WRITE

MCP-4014 33

This slide shows all of the scope terminator keywords for standard COBOL-85 statements.

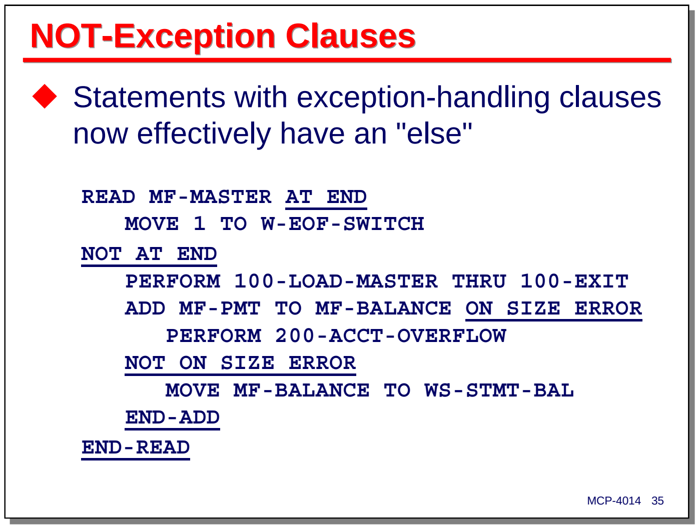


## DMSII Scope Terminator Words [MCP]

- END-ABORT-TRANSACTION
- END-ASSIGN
- END-BEGIN-TRANSACTION
- END-CANCEL
- END-CLOSE
- END-CREATE
- END-DELETE
- END-END-TRANSACTION
- END-FIND
- END-FREE
- END-GENERATE
- END-INSERT
- END-LOCK
- END-MODIFY
- END-OPEN
- END-RECREATE
- END-REMOVE
- END-SAVE
- END-SECURE
- END-SET
- END-STORE

MCP-4014 34

MCP COBOL-85 has a number of additional verbs for the DMSII host language interface. These also have their scope terminator keywords. The COMS **RECEIVE** statement also has **END-RECEIVE**.



## NOT-Exception Clauses

- Statements with exception-handling clauses now effectively have an "else"

```
READ MF-MASTER AT END
   MOVE 1 TO W-EOF-SWITCH
NOT AT END
   PERFORM 100-LOAD-MASTER THRU 100-EXIT
   ADD MF-PMT TO MF-BALANCE ON SIZE ERROR
      PERFORM 200-ACCT-OVERFLOW
   NOT ON SIZE ERROR
      MOVE MF-BALANCE TO WS-STMT-BAL
   END-ADD
END-READ
```

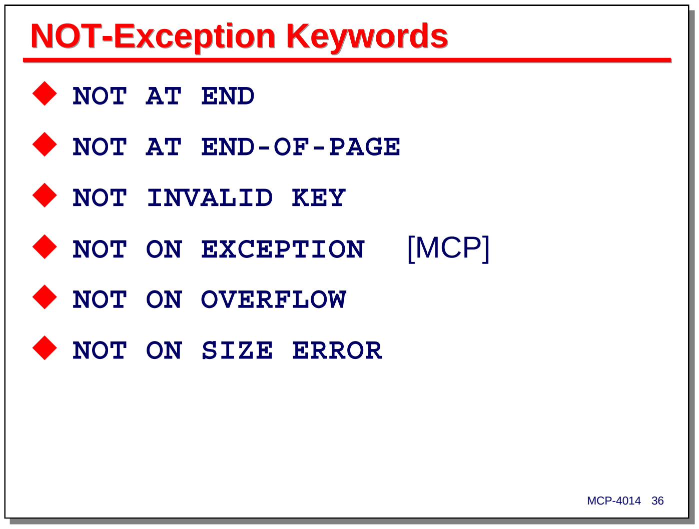
MCP-4014 35

Closely related to scope terminators is another COBOL-85 syntax feature, **`NOT`**-exception clauses. Effectively, these are **`ELSE`** clauses for the exception clauses that some COBOL verbs have.

As shown on the slide, in addition to handling the case of an **`AT END`** condition on a **`READ`** statement, you can now handle the case of a successful read by coding **`NOT AT END`** and nesting a series of statements under that clause.

Similarly, you can code a **`NOT ON SIZE ERROR`** to balance the **`ON SIZE ERROR`** clause of an arithmetic statement.

The real value of scope terminators and **`NOT`**-exception clauses is that they allow you to nest statements arbitrarily without the need to use **`GO TO`**s to branch around the sentences and positive-result cases that the classic COBOL syntax required you to write.

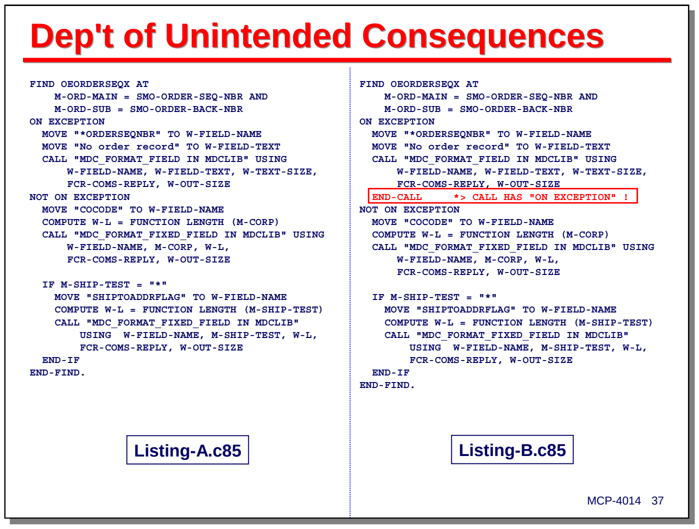


## NOT-Exception Keywords

- **`NOT AT END`**
- **`NOT AT END-OF-PAGE`**
- **`NOT INVALID KEY`**
- **`NOT ON EXCEPTION`** [MCP]
- **`NOT ON OVERFLOW`**
- **`NOT ON SIZE ERROR`**

MCP-4014 36

This slide shows all of the **`NOT`**-exception clauses that COBOL-85 supports. Note that **`ON EXCEPTION`** and **`NOT ON EXCEPTION`** are MCP extensions.



## Dep't of Unintended Consequences

```
FIND OEORDERSEQX AT
    M-ORD-MAIN = SMO-ORDER-SEQ-NBR AND
    M-ORD-SUB = SMO-ORDER-BACK-NBR
ON EXCEPTION
  MOVE "*ORDERSEQNBR" TO W-FIELD-NAME
  MOVE "No order record" TO W-FIELD-TEXT
  CALL "MDC_FORMAT_FIELD IN MDCLIB" USING
      W-FIELD-NAME, W-FIELD-TEXT, W-TEXT-SIZE,
      FCR-COMS-REPLY, W-OUT-SIZE
NOT ON EXCEPTION
  MOVE "COCODE" TO W-FIELD-NAME
  COMPUTE W-L = FUNCTION LENGTH (M-CORP)
  CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB" USING
      W-FIELD-NAME, M-CORP, W-L,
      FCR-COMS-REPLY, W-OUT-SIZE

  IF M-SHIP-TEST = "*"
    MOVE "SHIPTOADDRFLAG" TO W-FIELD-NAME
    COMPUTE W-L = FUNCTION LENGTH (M-SHIP-TEST)
    CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB"
        USING  W-FIELD-NAME, M-SHIP-TEST, W-L,
        FCR-COMS-REPLY, W-OUT-SIZE
  END-IF
END-FIND.
```

Listing-A.c85

```
FIND OEORDERSEQX AT
    M-ORD-MAIN = SMO-ORDER-SEQ-NBR AND
    M-ORD-SUB = SMO-ORDER-BACK-NBR
ON EXCEPTION
  MOVE "*ORDERSEQNBR" TO W-FIELD-NAME
  MOVE "No order record" TO W-FIELD-TEXT
  CALL "MDC_FORMAT_FIELD IN MDCLIB" USING
      W-FIELD-NAME, W-FIELD-TEXT, W-TEXT-SIZE,
      FCR-COMS-REPLY, W-OUT-SIZE
  END-CALL     *> CALL HAS "ON EXCEPTION" !
NOT ON EXCEPTION
  MOVE "COCODE" TO W-FIELD-NAME
  COMPUTE W-L = FUNCTION LENGTH (M-CORP)
  CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB" USING
      W-FIELD-NAME, M-CORP, W-L,
      FCR-COMS-REPLY, W-OUT-SIZE

  IF M-SHIP-TEST = "*"
    MOVE "SHIPTOADDRFLAG" TO W-FIELD-NAME
    COMPUTE W-L = FUNCTION LENGTH (M-SHIP-TEST)
    CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB"
        USING  W-FIELD-NAME, M-SHIP-TEST, W-L,
        FCR-COMS-REPLY, W-OUT-SIZE
  END-IF
END-FIND.
```

Listing-B.c85

MCP-4014 37

All is not sweetness and light with **NOT**-exception clauses, however. You really have to be aware of any implicit scope termination the compiler may be providing for you. This slide illustrates a very nasty gotcha that bit me a few months ago. I finally had to use TADS and trace the flow of the program, and then look at the generated object code to understand what was going on.

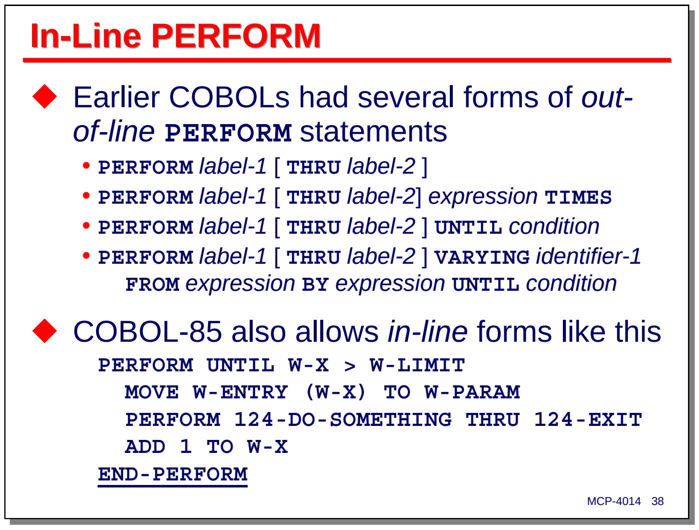
The symptom was that neither the **ON EXCEPTION** or **NOT ON EXCEPTION** clauses of the DMSII **FIND** statement were being executed. The record being searched for was present in the database, and tracing showed that the **ON EXCEPTION CLAUSE** was not being executed, but the **NOT ON EXCEPTION** clause was skipped over as well. Finally after looking at the code the compiler generated using EDITOR, I realized what the problem was.

The MCP library **CALL** syntax also has an **ON EXCEPTION** clause, and just as with **ELSE** clauses for nested IF statements, the compiler was associating the **ON EXCEPTION** with the most immediate verb that accepted that clause, which in this case was the **CALL** statement on the seventh line. The **NOT ON EXCEPTION** clause was being compiled as subordinate to the **FIND** statement's **ON EXCEPTION** clause, which was not being executed. Doh!

The solution to this was really simple – just add an **END-CALL** keyword before the **FIND** statement's **NOT ON EXCEPTION** clause to properly bracket the scope of the **CALL** statement.

This particular case is an MCP-specific problem, as MCP COBOL uses **ON EXCEPTION** in a large number of extensions to the standard language, but the potential exists for it to occur with purely standard coding as well, say, with nested **READ** statements.

If you find the text on this slide too small to read, there is a companion document with listings from this presentation in a less eye-watering font size.



## In-Line PERFORM

- Earlier COBOLs had several forms of *out-of-line* **PERFORM** statements
  - **PERFORM** *label-1* [ **THRU** *label-2* ]
  - **PERFORM** *label-1* [ **THRU** *label-2*] *expression* **TIMES**
  - **PERFORM** *label-1* [ **THRU** *label-2* ] **UNTIL** *condition*
  - **PERFORM** *label-1* [ **THRU** *label-2* ] **VARYING** *identifier-1* **FROM** *expression* **BY** *expression* **UNTIL** *condition*
- COBOL-85 also allows *in-line* forms like this

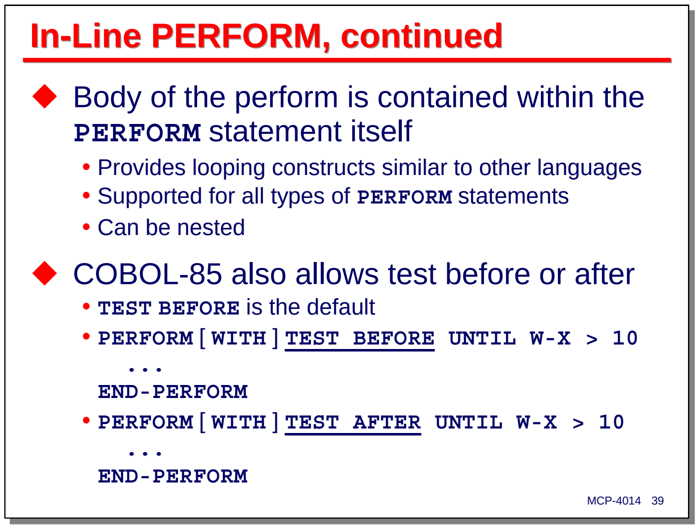
```
PERFORM UNTIL W-X > W-LIMIT
  MOVE W-ENTRY (W-X) TO W-PARAM
  PERFORM 124-DO-SOMETHING THRU 124-EXIT
  ADD 1 TO W-X
END-PERFORM
```

MCP-4014 38

The next subject is another major advance in the syntax of control flow that brought COBOL coding practices kicking and screaming into the '70s.

COBOL-74 and earlier versions had several forms of **PERFORM** statements where the object of the **PERFORM** was an out-of-line block of code, i.e., the statements being performed were elsewhere in the source file and identified by paragraph or section labels. In addition to the simple subroutine-like **PERFORM**, there are variants that implement loop constructs.

COBOL-85 has done a very simple thing – it allows you to move that out-of-line block of code under the **PERFORM** verb and terminate it with the **END-PERFORM** scope terminator. This construct is termed an *in-line* **PERFORM**.



## In-Line PERFORM, continued

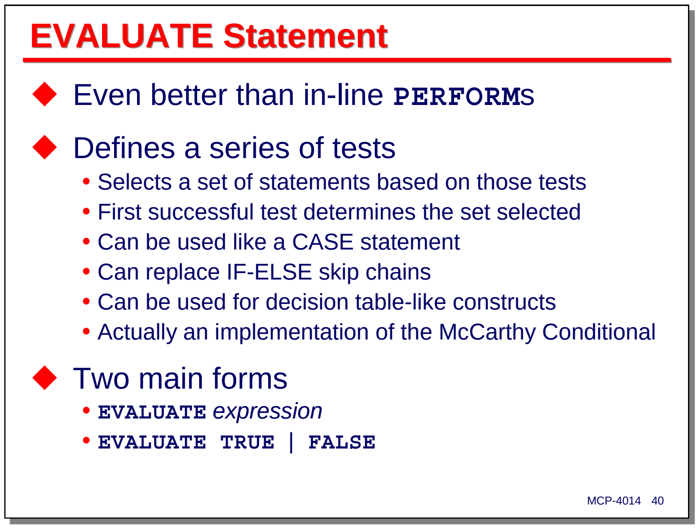
- Body of the perform is contained within the **PERFORM** statement itself
  - Provides looping constructs similar to other languages
  - Supported for all types of **PERFORM** statements
  - Can be nested
- COBOL-85 also allows test before or after
  - **TEST BEFORE** is the default
  - `PERFORM [ WITH ] TEST BEFORE UNTIL W-X > 10`
    ```
       ...
    END-PERFORM
    ```
  - `PERFORM [ WITH ] TEST AFTER UNTIL W-X > 10`
    ```
       ...
    END-PERFORM
    ```

MCP-4014 39

The in-line **PERFORM** is supported for all variants of the **PERFORM** verb, although it is most often used with the variants that do looping. In-line **PERFORM**s can be nested arbitrarily. Finally we have a way to code loops without **GO TO**s or out-of-line **PERFORM**s.

COBOL-85 also introduced a new option for the **PERFORM** variants that use the **UNTIL** clause. By default (and in earlier COBOLs the only option), the **UNTIL** condition is tested at the beginning of the loop. This implies that if the condition is initially true, the loop will not be executed at all.

COBOL-85 allows you to specify [ **WITH** ] **TEST BEFORE** (the default) or [ **WITH** ] **TEST AFTER** following the **PERFORM** verb to indicate whether the condition should be tested at the beginning or end of the loop. Thus, **TEST BEFORE** behaves like a **WHILE** loop in other languages, while **TEST AFTER** behaves like a **DO** or **REPEAT** loop.



## EVALUATE Statement

- Even better than in-line **`PERFORM`**s
- Defines a series of tests
  - Selects a set of statements based on those tests
  - First successful test determines the set selected
  - Can be used like a CASE statement
  - Can replace IF-ELSE skip chains
  - Can be used for decision table-like constructs
  - Actually an implementation of the McCarthy Conditional
- Two main forms
  - **`EVALUATE`** *expression*
  - **`EVALUATE TRUE | FALSE`**

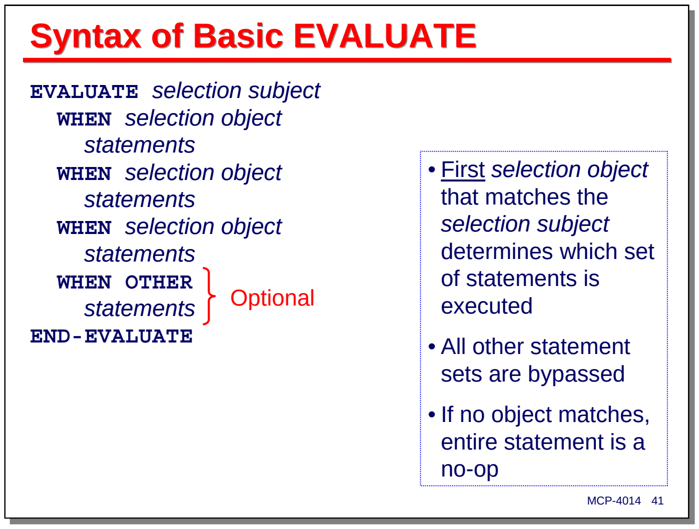
MCP-4014 40

Possibly the greatest syntactic advance in COBOL-85 is the **`EVALUATE`** statement. This is an extremely versatile construct. I love it, and use it wherever I can.

Basically, **`EVALUATE`** defines a series of tests and associates a set of statements with each of those tests. The tests are evaluated one at a time, in the order written. For the first test that succeeds, the corresponding set of statements is executed. All remaining tests are skipped and all other sets of statements are not executed. **`EVALUATE`** can be used like a **`CASE`** statement, to replace **`IF-ELSE`** skip chains, and even to implement decision table-like constructs. It is form of programming language construct known as a McCarthy Conditional.

There are two main forms of **`EVALUATE`**:

- One where you evaluate an expression and construct tests against the value of that expression
- The other where you indicate whether you are looking for a test that evaluates to true or false, and then specify the tests as Boolean conditions.



## Syntax of Basic EVALUATE

```
EVALUATE  selection subject
   WHEN  selection object
      statements
   WHEN  selection object
      statements
   WHEN  selection object
      statements
   WHEN  OTHER  }
      statements }  Optional
END-EVALUATE
```

- First *selection object* that matches the *selection subject* determines which set of statements is executed
- All other statement sets are bypassed
- If no object matches, entire statement is a no-op

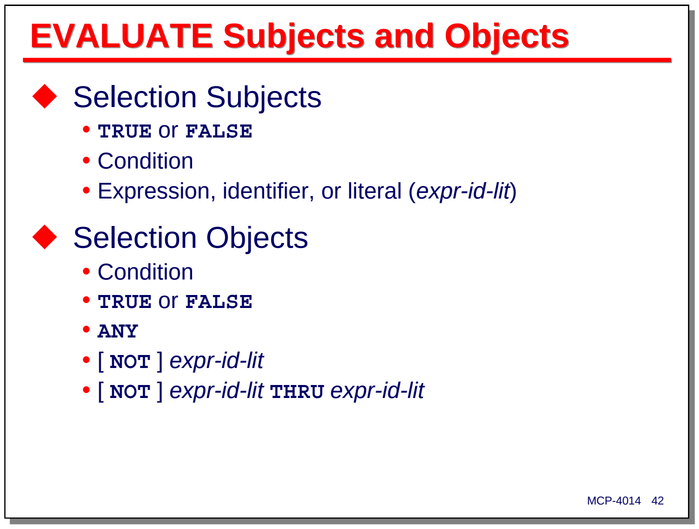
MCP-4014 41

This slide illustrates the general idea of the basic **EVALUATE** statement. The statement specifies a *selection subject*, which can take the forms shown on the next slide. Subordinate to the **EVALUATE** are a series of **WHEN** clauses that specify *selection objects*. These are the tests to be evaluated. Subordinate to each **WHEN** clause is a series of statements. The first **WHEN** clause whose selection object matches the selection subject has its statements executed.

If you have several tests that, if successful, would cause the same set of statements to be executed, you can stack the **WHEN** clauses one after the other without any intervening statements. All of the contiguously-written **WHEN** clauses will be associated with the next set of subordinate statements. If any of the **WHEN** clauses matches the selection subject, the associated set of statements will be executed. An example a couple of slides ahead illustrates this.

If none of the selection objects in the **WHEN** clauses matches the selection subject, the **EVALUATE** statement is effectively a no-op.

The last **WHEN** clause in an **EVALUATE** statement can specify **WHEN OTHER**. This is a catch-all test, and its statements will be executed if no other **WHEN** clause was successful. Use of **WHEN OTHER** is entirely optional.



## EVALUATE Subjects and Objects

- Selection Subjects
  - **TRUE** or **FALSE**
  - Condition
  - Expression, identifier, or literal (*expr-id-lit*)
- Selection Objects
  - Condition
  - **TRUE** or **FALSE**
  - **ANY**
  - [ **NOT** ] *expr-id-lit*
  - [ **NOT** ] *expr-id-lit* **THRU** *expr-id-lit*

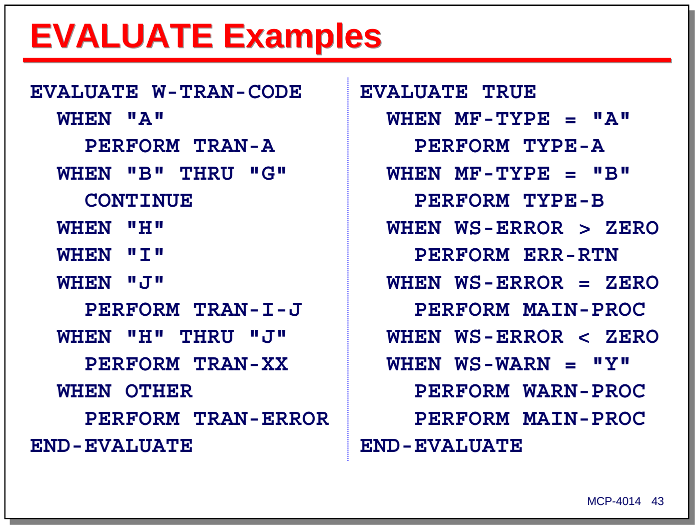
MCP-4014 42

The **EVALUATE** statement is very flexible, and this makes its semantics a little difficult to understand at first. The basic idea is that it attempts to match a selection subject to one of several selection objects.

A selection subject can be the keyword **TRUE** or **FALSE**, a Boolean condition, or an expression (including data-name identifiers and literals).

A selection object can be a Boolean condition, the keyword **TRUE** or **FALSE**, the keyword **ANY**, an expression (including a data-name identifier or literal, and optionally preceded by **NOT**), or a pair of expressions separated by the keyword **THRU**. This latter case is a shorthand for specifying a range of values starting with the first expression and ending with the second one.

The object keyword **ANY** is a don't-care value – it will match any subject value.



## EVALUATE Examples

```
EVALUATE W-TRAN-CODE
  WHEN "A"
    PERFORM TRAN-A
  WHEN "B" THRU "G"
    CONTINUE
  WHEN "H"
  WHEN "I"
  WHEN "J"
    PERFORM TRAN-I-J
  WHEN "H" THRU "J"
    PERFORM TRAN-XX
  WHEN OTHER
    PERFORM TRAN-ERROR
END-EVALUATE
```

```
EVALUATE TRUE
  WHEN MF-TYPE = "A"
    PERFORM TYPE-A
  WHEN MF-TYPE = "B"
    PERFORM TYPE-B
  WHEN WS-ERROR > ZERO
    PERFORM ERR-RTN
  WHEN WS-ERROR = ZERO
    PERFORM MAIN-PROC
  WHEN WS-ERROR < ZERO
  WHEN WS-WARN = "Y"
    PERFORM WARN-PROC
    PERFORM MAIN-PROC
END-EVALUATE
```

MCP-4014 43

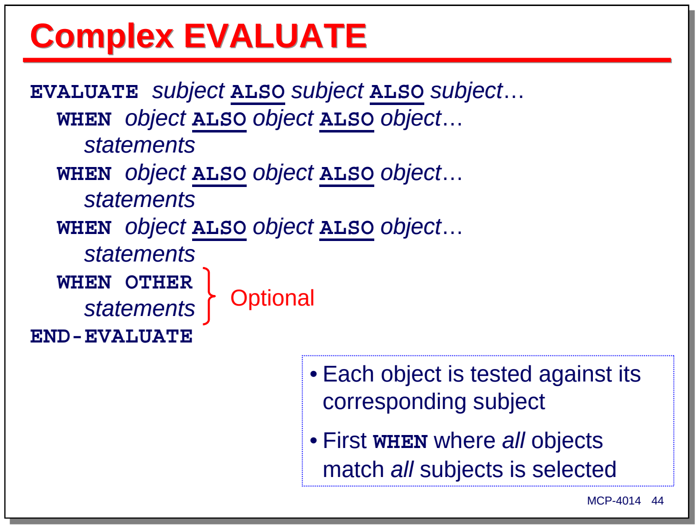
This slide illustrates the two primary ways of using **EVALUATE**.

In the left panel, the selection subject is an expression or data-name identifier that evaluates to a value. The selection objects in the **WHEN** clauses are values that are tested against the subject's value. The first object value that is equal to the subject value determines which set of statements gets executed. This form of **EVALUATE** is similar to a **CASE** statement.

Note that the **WHEN** clauses for values "**H**", "**I**", and "**J**" are written together. All of these are associated with the **PERFORM TRAN-I-J** statement. If any of those **WHEN** clauses matches the value of **W-TRAN-CODE**, that **PERFORM** will be executed.

Also note that the next **WHEN** clause for **"H" THRU "J"** is effectively the same as the three **WHEN** clauses written separately above it. This **WHEN** will never be selected, however, as the **WHEN** clauses are evaluated in the order written (or rather the result is as if they had been evaluated in the order written), and the preceding individual **WHEN** clauses would always be selected before the one with the **THRU** object.

The right panel shows the second general form. The selection subject is **TRUE** or **FALSE** (although **TRUE** seems to be used far more often), and the selection objects in the **WHEN** clauses are written as Boolean conditions. The first of those conditions that evaluates to the truth value of the subject determines which **WHEN** clause is selected. This form of **EVALUATE** is effectively a skip-chain test, and is a much nicer way of writing long sequences of tests than **IF**/**ELSE IF** constructs.



## Complex EVALUATE

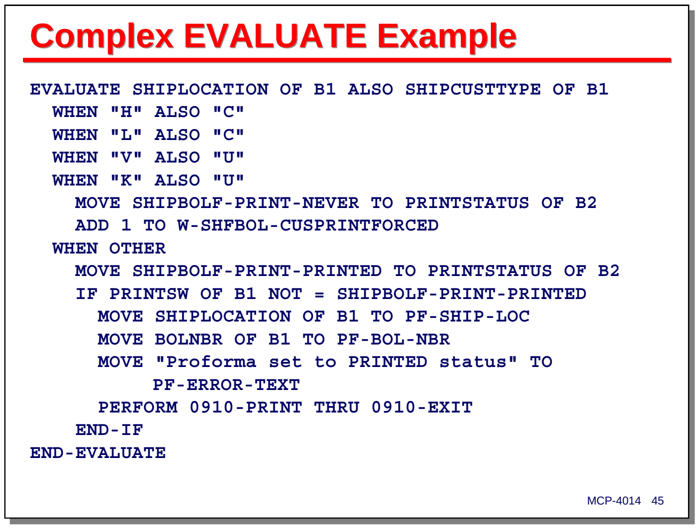
```
EVALUATE  subject ALSO subject ALSO subject…
   WHEN  object ALSO object ALSO object…
      statements
   WHEN  object ALSO object ALSO object…
      statements
   WHEN  object ALSO object ALSO object…
      statements
   WHEN OTHER  } Optional
      statements }
END-EVALUATE
```

- Each object is tested against its corresponding subject
- First **WHEN** where *all* objects match *all* subjects is selected

MCP-4014 44

Now for some serious stuff: **EVALUATE** has a more advanced form, which involves multiple selection subjects and selection objects. You write the multiple subjects and objects with the keyword **ALSO** separating them. This form works the same as the basic **EVALUATE**, except that for a **WHEN** clause to be selected, all of its objects much match all of the subjects.

You can use this form to more or less directly implement a decision table. The big problem with this form is trying to make it readable in the 61 columns available with the COBOL source record format for Margin B.



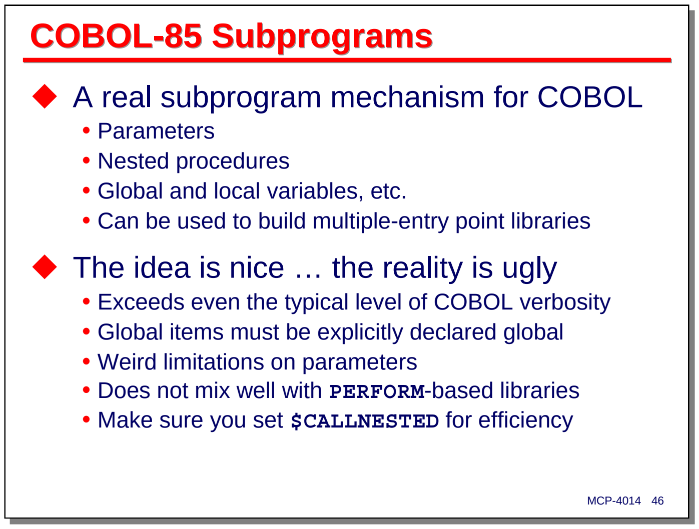
## Complex EVALUATE Example

```
EVALUATE SHIPLOCATION OF B1 ALSO SHIPCUSTTYPE OF B1
  WHEN "H" ALSO "C"
  WHEN "L" ALSO "C"
  WHEN "V" ALSO "U"
  WHEN "K" ALSO "U"
    MOVE SHIPBOLF-PRINT-NEVER TO PRINTSTATUS OF B2
    ADD 1 TO W-SHFBOL-CUSPRINTFORCED
  WHEN OTHER
    MOVE SHIPBOLF-PRINT-PRINTED TO PRINTSTATUS OF B2
    IF PRINTSW OF B1 NOT = SHIPBOLF-PRINT-PRINTED
      MOVE SHIPLOCATION OF B1 TO PF-SHIP-LOC
      MOVE BOLNBR OF B1 TO PF-BOL-NBR
      MOVE "Proforma set to PRINTED status" TO
           PF-ERROR-TEXT
      PERFORM 0910-PRINT THRU 0910-EXIT
    END-IF
END-EVALUATE
```

MCP-4014 45

I have only encountered one case thus far where I thought the multiple-selection form of **`EVALUATE`** was appropriate, although I've probably had some others and simply didn't recognize them. If you find you are nesting **`EVALUATE`** basic statements, you might think about whether this more advanced form would be a better choice.

This code snippet came from a database conversion program I wrote earlier this year.



## COBOL-85 Subprograms

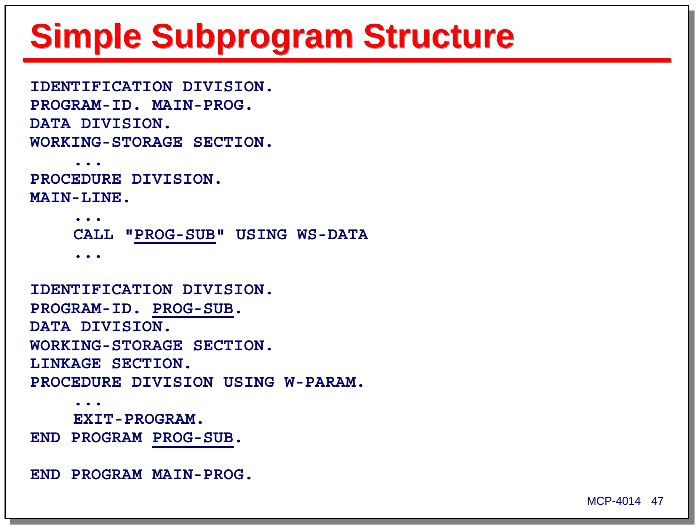
- A real subprogram mechanism for COBOL
  - Parameters
  - Nested procedures
  - Global and local variables, etc.
  - Can be used to build multiple-entry point libraries
- The idea is nice … the reality is ugly
  - Exceeds even the typical level of COBOL verbosity
  - Global items must be explicitly declared global
  - Weird limitations on parameters
  - Does not mix well with **`PERFORM`**-based libraries
  - Make sure you set **`$CALLNESTED`** for efficiency

MCP-4014 46

The final subject in this section on statements and program control flow concerns the new subprogram or "nested program" capability of COBOL-85. This seems like such a really great idea – real subroutines, with parameters and local storage, and global scoping, and all of the other stuff we're used to in block-structured languages.

The idea is nice, but the implementation and the coding necessary to invoke it is pretty ugly. The COBOL-85 designers came up with a syntax that not only imposed a COBOL-like flavor to subroutine declaration, but set a new standard for coding verbosity that was not exceeded until COBOL-2002 provided object oriented constructs. The slide lists some of the limitations and weirdness with global variables and parameters. **`PERFORM`**s can only exist and be accessed within a subprogram, which makes this facility difficult to use with established libraries of **`COPY`** routines. Also, the **`CALL`** syntax for subprograms is identical to that for ANSI IPC calls to externally bound routines, which generates really inefficient code unless you set the **`$CALLNESTED`** compiler option and forego ANSI IPC altogether.

I've tried to use this feature of COBOL-85, and frankly, I'm turned off by the verbosity of the syntax and all of the weird constraints and limitations. I've concluded that it's more trouble that it's worth. The only thing I found nested programs good for is constructing COBOL server libraries with multiple entry points.



## Simple Subprogram Structure

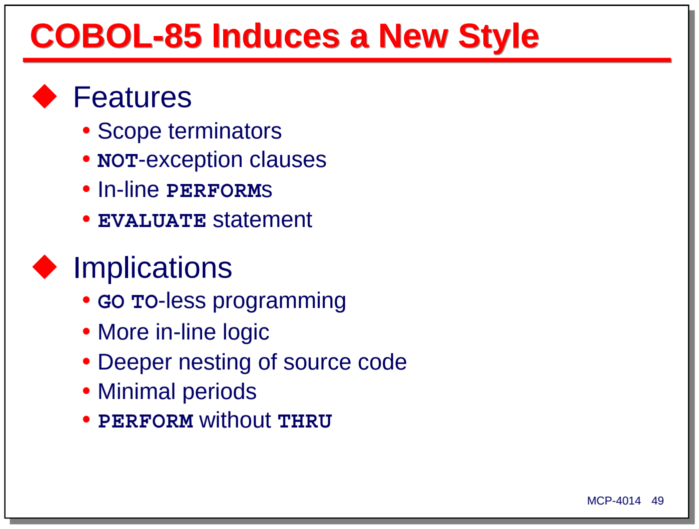
```
IDENTIFICATION DIVISION.
PROGRAM-ID. MAIN-PROG.
DATA DIVISION.
WORKING-STORAGE SECTION.
    ...
PROCEDURE DIVISION.
MAIN-LINE.
    ...
    CALL "PROG-SUB" USING WS-DATA
    ...

IDENTIFICATION DIVISION.
PROGRAM-ID. PROG-SUB.
DATA DIVISION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
PROCEDURE DIVISION USING W-PARAM.
    ...
    EXIT-PROGRAM.
END PROGRAM PROG-SUB.

END PROGRAM MAIN-PROG.
```

MCP-4014 47

This slide shows a simple outline of a nested program, **`PROG-SUB`**, contained within an outer program, **`MAIN-PROG`**, and called from that main program.

## A New Style for COBOL Coding

This concludes the discussion on new features in COBOL-85. Next, I want to talk about how these features have influenced the way that I think about coding in COBOL and how they have, for me, induced a new style of coding.



## COBOL-85 Induces a New Style

- Features
  - Scope terminators
  - **NOT**-exception clauses
  - In-line **PERFORM**s
  - **EVALUATE** statement
- Implications
  - **GO TO**-less programming
  - More in-line logic
  - Deeper nesting of source code
  - Minimal periods
  - **PERFORM** without **THRU**

MCP-4014 49

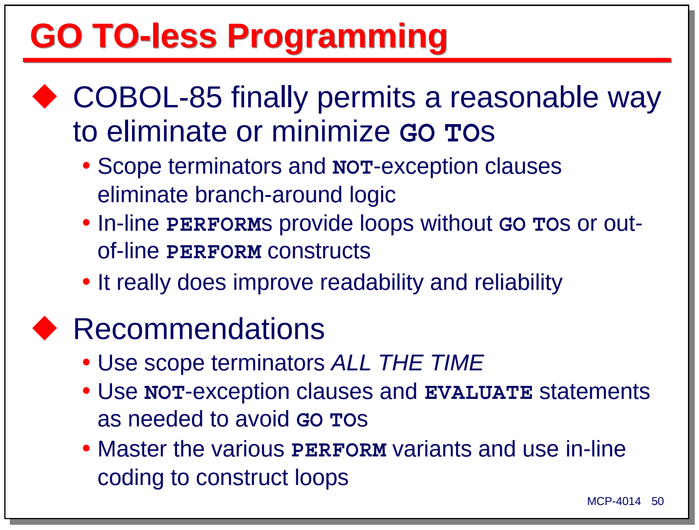
All of the new features in COBOL-85 have their influence, but four in particular have really affected my style of coding:

- Scope terminators (**END-IF**, **END-READ**, **END-SEARCH**, etc.)
- **NOT**-exception clauses (**NOT AT END**, **NOT ON EXCEPTION**, etc.)
- In-line **PERFORM**s
- **EVALUATE** statements.

Using these on a consistent basis has resulted in the following style implications:

- **GO TO**-less programming. Using the same techniques I use in Algol and other block-structured languages, I have been able to eliminate essentially all **GO TO**s from the new code that I write while retaining a highly readable and maintainable result.
- More in-line logic. All four features have allowed me to move more subordinate code in line with the predicate statements they go with, rather than keeping them out of line and branching to or **PERFORM**ing them.
- More in-line logic results in deeper nesting of source code. This is mostly good, but as we will see shortly, there are some reasonable limits that need to be applied.
- Minimal periods. I find that I don't need to include periods very often, so I have gotten rid of as many as I can. In other words, I’m writing longer sentences, and fewer of them.
- **PERFORM** without **THRU**. Largely as a consequence of **GO TO**-less programming and minimizing the number of periods, I am finding that it is possible to **PERFORM** single paragraphs, so I don't really need the **THRU** clause (and the occasionally nasty problems it causes).

I'll elaborate on these points over the next several slides.



## GO TO-less Programming

- COBOL-85 finally permits a reasonable way to eliminate or minimize **GO TO**s
  - Scope terminators and **NOT**-exception clauses eliminate branch-around logic
  - In-line **PERFORM**s provide loops without **GO TO**s or out-of-line **PERFORM** constructs
  - It really does improve readability and reliability
- Recommendations
  - Use scope terminators *ALL THE TIME*
  - Use **NOT**-exception clauses and **EVALUATE** statements as needed to avoid **GO TO**s
  - Master the various **PERFORM** variants and use in-line coding to construct loops

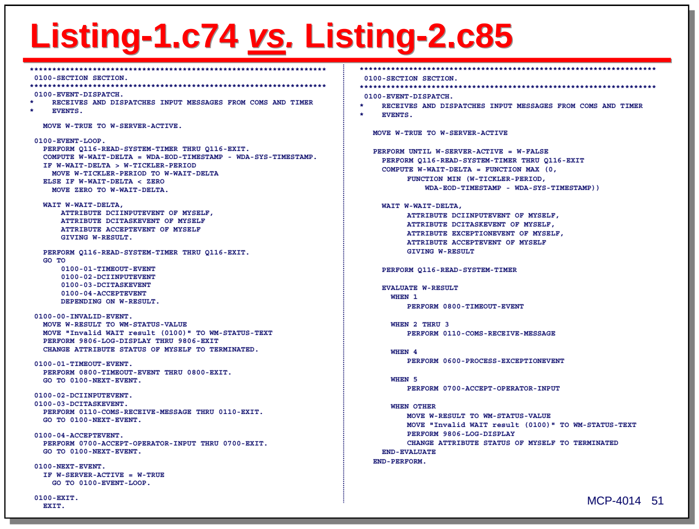
MCP-4014 50

The new control-flow features of COBOL-85 finally permit us to eliminate (or at least minimize) **GO TO**s in a reasonable way, without resorting to extraneous out-of-line **PERFORM**s. Scope terminators, **NOT**-exception clauses, and in-line **PERFORM**s are the chief enablers of this, although the **EVALUATE** statement can also be a big help. It took a while to break my old style and get comfortable with a new one, but I've found it really does improve readability and reliability of my code, and I think I may be coding more productively as well.

My recommendation is to use scope terminators all of the time, especially with **IF**, **SEARCH**, **EVALUATE**, and all I/O statements (including DMSII verbs) that have exception clauses. Enforcing a consistent style in this area will help minimize confusion and eliminate bugs. It will also help with the minimization of periods, as discussed a little later.

Also use **NOT**-exception clauses to avoid **GO TO**s. These clauses are a wonderful addition to the language, and can really help in eliminating tortuous branch-around logic. Similarly, **EVALUATE** statements can make long skip chain tests easier to understand and maintain. **EVALUATE** statements are great for implementing **CASE**-like constructs without the need for computed **GO TO**s and **GO TO**s for "branch around the rest of the cases" logic.

Finally, if you haven't mastered the various forms of iterative **PERFORM** statements, do so. They allow you to implement loop constructs in a consistent way and without **GO TO**s.



## Listing-1.c74 *vs.* Listing-2.c85

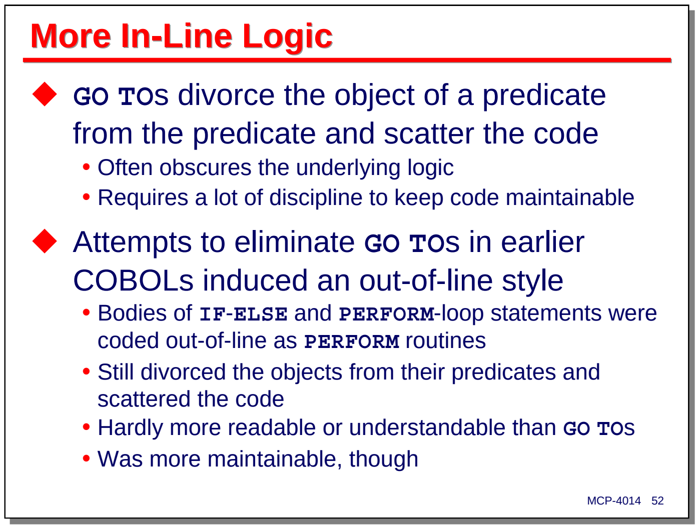
```
*****************************************************************
 0100-SECTION SECTION.
*****************************************************************
 0100-EVENT-DISPATCH.
*    RECEIVES AND DISPATCHES INPUT MESSAGES FROM COMS AND TIMER
*    EVENTS.

   MOVE W-TRUE TO W-SERVER-ACTIVE.

 0100-EVENT-LOOP.
   PERFORM Q116-READ-SYSTEM-TIMER THRU Q116-EXIT.
   COMPUTE W-WAIT-DELTA = WDA-EOD-TIMESTAMP - WDA-SYS-TIMESTAMP.
   IF W-WAIT-DELTA > W-TICKLER-PERIOD
     MOVE W-TICKLER-PERIOD TO W-WAIT-DELTA
   ELSE IF W-WAIT-DELTA < ZERO
     MOVE ZERO TO W-WAIT-DELTA.

   WAIT W-WAIT-DELTA,
       ATTRIBUTE DCIINPUTEVENT OF MYSELF,
       ATTRIBUTE DCITASKEVENT OF MYSELF
       ATTRIBUTE ACCEPTEVENT OF MYSELF
       GIVING W-RESULT.

   PERFORM Q116-READ-SYSTEM-TIMER THRU Q116-EXIT.
   GO TO
       0100-01-TIMEOUT-EVENT
       0100-02-DCIINPUTEVENT
       0100-03-DCITASKEVENT
       0100-04-ACCEPTEVENT
       DEPENDING ON W-RESULT.

 0100-00-INVALID-EVENT.
   MOVE W-RESULT TO WM-STATUS-VALUE
   MOVE "Invalid WAIT result (0100)" TO WM-STATUS-TEXT
   PERFORM 9806-LOG-DISPLAY THRU 9806-EXIT
   CHANGE ATTRIBUTE STATUS OF MYSELF TO TERMINATED.

 0100-01-TIMEOUT-EVENT.
   PERFORM 0800-TIMEOUT-EVENT THRU 0800-EXIT.
   GO TO 0100-NEXT-EVENT.

 0100-02-DCIINPUTEVENT.
 0100-03-DCITASKEVENT.
   PERFORM 0110-COMS-RECEIVE-MESSAGE THRU 0110-EXIT.
   GO TO 0100-NEXT-EVENT.

 0100-04-ACCEPTEVENT.
   PERFORM 0700-ACCEPT-OPERATOR-INPUT THRU 0700-EXIT.
   GO TO 0100-NEXT-EVENT.

 0100-NEXT-EVENT.
   IF W-SERVER-ACTIVE = W-TRUE
     GO TO 0100-EVENT-LOOP.

 0100-EXIT.
   EXIT.
```

```
*****************************************************************
 0100-SECTION SECTION.
*****************************************************************
 0100-EVENT-DISPATCH.
*    RECEIVES AND DISPATCHES INPUT MESSAGES FROM COMS AND TIMER
*    EVENTS.

   MOVE W-TRUE TO W-SERVER-ACTIVE

   PERFORM UNTIL W-SERVER-ACTIVE = W-FALSE
     PERFORM Q116-READ-SYSTEM-TIMER THRU Q116-EXIT
     COMPUTE W-WAIT-DELTA = FUNCTION MAX (0,
           FUNCTION MIN (W-TICKLER-PERIOD,
               WDA-EOD-TIMESTAMP - WDA-SYS-TIMESTAMP))

     WAIT W-WAIT-DELTA,
           ATTRIBUTE DCIINPUTEVENT OF MYSELF,
           ATTRIBUTE DCITASKEVENT OF MYSELF,
           ATTRIBUTE EXCEPTIONEVENT OF MYSELF,
           ATTRIBUTE ACCEPTEVENT OF MYSELF
           GIVING W-RESULT

     PERFORM Q116-READ-SYSTEM-TIMER

     EVALUATE W-RESULT
       WHEN 1
           PERFORM 0800-TIMEOUT-EVENT

       WHEN 2 THRU 3
           PERFORM 0110-COMS-RECEIVE-MESSAGE

       WHEN 4
           PERFORM 0600-PROCESS-EXCEPTIONEVENT

       WHEN 5
           PERFORM 0700-ACCEPT-OPERATOR-INPUT

       WHEN OTHER
           MOVE W-RESULT TO WM-STATUS-VALUE
           MOVE "Invalid WAIT result (0100)" TO WM-STATUS-TEXT
           PERFORM 9806-LOG-DISPLAY
           CHANGE ATTRIBUTE STATUS OF MYSELF TO TERMINATED
     END-EVALUATE
   END-PERFORM.
```

MCP-4014 51

This slide shows an example of the kind of transformation that can take place by using the newer COBOL-85 flow-of-control constructs. It is the main-line loop of a COMS transaction processor that does a multiple wait on timeouts, COMS message events, and the task's **`ACCEPTEVENT`**. Note the use of a computed **`GO TO`** to select what should be done as a result of the wait, and the extra **`GO TO`**s to branch around the cases that are not to be executed.

The code on the right was manually converted from the code on the left using COBOL-85 constructs. Note that the in-line **`PERFORM`** makes the loop explicit. The **`EVALUATE`** statement that replaces the computed **`GO TO`** also makes it much clearer what is going on in this routine. This version actually has an extra feature – the addition of a wait on the task's **`EXCEPTIONEVENT`**. Despite the additional functionality, the COBOL-85 version is shorter than the original one.



## More In-Line Logic

- **GO TO**s divorce the object of a predicate from the predicate and scatter the code
  - Often obscures the underlying logic
  - Requires a lot of discipline to keep code maintainable
- Attempts to eliminate **GO TO**s in earlier COBOLs induced an out-of-line style
  - Bodies of **IF-ELSE** and **PERFORM**-loop statements were coded out-of-line as **PERFORM** routines
  - Still divorced the objects from their predicates and scattered the code
  - Hardly more readable or understandable than **GO TO**s
  - Was more maintainable, though

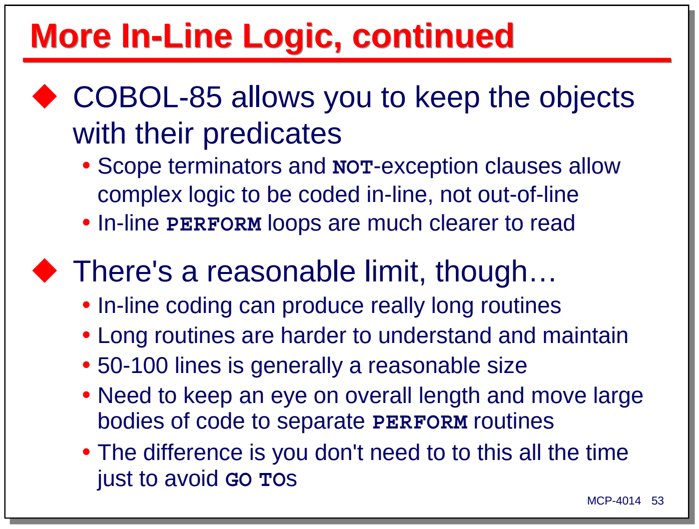
MCP-4014 52

The next implication that comes from using COBOL-85 flow-of-control constructs is that you tend to code more of the logic in line. Earlier versions of COBOL essentially forced you to code quite a bit of code out-of-line, either by branching to it, or by means of **PERFORM** constructs. This separates the predicate of a construct (an **IF** statement, say) from its object (the statements nested under that **IF**).

The problem with out-of-line coding and the separation of predicates from their objects is that it can obscure the underlying logic. You can minimize this effect by using a consistent style and keeping an eye on the complexity of the code, but it takes a lot of discipline to preserve that consistency across multiple updates and keep the code maintainable.

Attempts to enforce a **GO TO**-less style in earlier versions of COBOL induced an out-of-line coding style, where the object of a predicate was almost always written as a **PERFORM** statement, and the statements that were really the object were located elsewhere in the program. Similarly, all loops were written as an iterative form of **PERFORM**, with the body of the loop residing in another part of the source.

I have always thought that this style was self-defeating, because it eliminated **GO TO**s at the cost of separating predicates from objects and scattering the code. This always seemed to me to be more confusing than a disciplined use of **GO TO** statements, although it could keep you away from the kind of rat's nest that undisciplined branching can produce.



## More In-Line Logic, continued

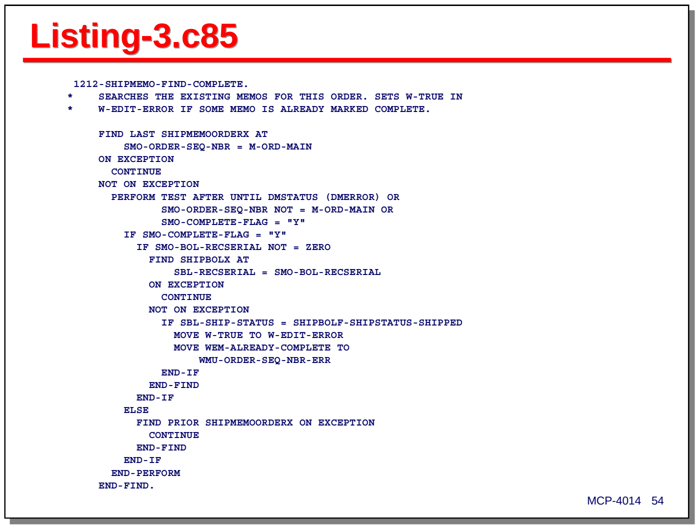
- COBOL-85 allows you to keep the objects with their predicates
  - Scope terminators and **NOT**-exception clauses allow complex logic to be coded in-line, not out-of-line
  - In-line **PERFORM** loops are much clearer to read
- There's a reasonable limit, though…
  - In-line coding can produce really long routines
  - Long routines are harder to understand and maintain
  - 50-100 lines is generally a reasonable size
  - Need to keep an eye on overall length and move large bodies of code to separate **PERFORM** routines
  - The difference is you don't need to to this all the time just to avoid **GO TO**s

MCP-4014 53

COBOL-85 solves this problem by allowing you to code the objects in line with their predicates. Scope terminators and **NOT**-exception clauses permit you to write complex logic in line, without the need for branch-around logic or out-of-line **PERFORM**s. Using in-line **PERFORM**s for loops makes the loops explicit and much easier to read and understand.

There's a reasonable limit to how much you can code in line, however. Moving to a more in-line style tends to make your routines longer, and as routines get longer, they generally become harder to understand and maintain. The upper limit on the size of a routine is generally accepted to be on the order of 50-100 lines, although routines with straight-line logic (i.e., streams of code without significant branching or looping) often can be longer without reducing readability or maintainability.

You simply need to keep an eye on how long your routines are getting and, at an appropriate point, break the more deeply-nested code out into a separate **PERFORM**. The big difference here from the out-of-line style discussed earlier is that you get to choose when to break out the lower-level logic – you don't need to do it all of the time – just when it makes sense to do so.



## Listing-3.c85

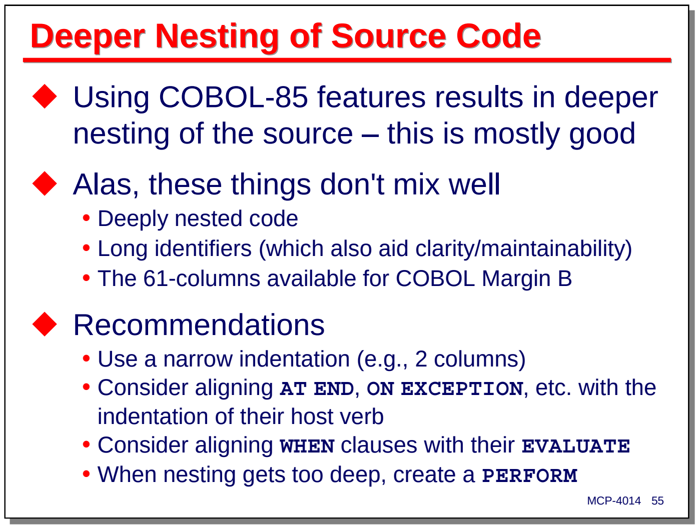
```
 1212-SHIPMEMO-FIND-COMPLETE.
*    SEARCHES THE EXISTING MEMOS FOR THIS ORDER. SETS W-TRUE IN
*    W-EDIT-ERROR IF SOME MEMO IS ALREADY MARKED COMPLETE.

     FIND LAST SHIPMEMOORDERX AT
         SMO-ORDER-SEQ-NBR = M-ORD-MAIN
     ON EXCEPTION
       CONTINUE
     NOT ON EXCEPTION
       PERFORM TEST AFTER UNTIL DMSTATUS (DMERROR) OR
               SMO-ORDER-SEQ-NBR NOT = M-ORD-MAIN OR
               SMO-COMPLETE-FLAG = "Y"
         IF SMO-COMPLETE-FLAG = "Y"
           IF SMO-BOL-RECSERIAL NOT = ZERO
             FIND SHIPBOLX AT
                 SBL-RECSERIAL = SMO-BOL-RECSERIAL
             ON EXCEPTION
               CONTINUE
             NOT ON EXCEPTION
               IF SBL-SHIP-STATUS = SHIPBOLF-SHIPSTATUS-SHIPPED
                 MOVE W-TRUE TO W-EDIT-ERROR
                 MOVE WEM-ALREADY-COMPLETE TO
                     WMU-ORDER-SEQ-NBR-ERR
               END-IF
             END-FIND
           END-IF
         ELSE
           FIND PRIOR SHIPMEMOORDERX ON EXCEPTION
             CONTINUE
           END-FIND
         END-IF
       END-PERFORM
     END-FIND.
```

MCP-4014 54

This slide shows an example of some code that would have required extensive use of **`GO TO`**s or out-of-line **`PERFORM`**s had it been coded for an earlier version of COBOL. I have found this particular type of routine to be one that recurs often, especially when doing DMSII programming.

The general problem is that we need to retrieve a series of records and perform some operation on them. Note that the entire routine is one DMSII **`FIND`** statement.

- It attempts to locate the first record in the sequence: if there isn't one, it simply exits; otherwise the sequence is processed as part of the **`NOT ON EXCEPTION`** clause.
- Within the **`NOT ON EXCEPTION`** clause, there is a **`PERFORM`** loop that will terminate when we get a DMSII exception (which will occur on the **`FIND PRIOR`** statement at the end of the loop), when the key that identifies the sequence of records (**`SMO-ORDER-SEQ-NBR`**) no longer matches, or an early-exit condition (**`SMO-COMPLETE-FLAG = "Y"`**) is encountered.
- Within the loop, there are a couple of nested **`IF`** statements and another **`FIND`** statement that checks the status of a related record
- If none of the terminating conditions is found, the **`FIND PRIOR`** statement at the end of the loop fetches the next record in sequence to be tested in the next iteration of the loop.



## Deeper Nesting of Source Code

- Using COBOL-85 features results in deeper nesting of the source – this is mostly good
- Alas, these things don't mix well
  - Deeply nested code
  - Long identifiers (which also aid clarity/maintainability)
  - The 61-columns available for COBOL Margin B
- Recommendations
  - Use a narrow indentation (e.g., 2 columns)
  - Consider aligning **AT END**, **ON EXCEPTION**, etc. with the indentation of their host verb
  - Consider aligning **WHEN** clauses with their **EVALUATE**
  - When nesting gets too deep, create a **PERFORM**

MCP-4014 55

Another outcome of using the COBOL-85 features, and one closely related to more in-line coding, is that the source code tends to become more deeply nested. This is mostly a good thing.

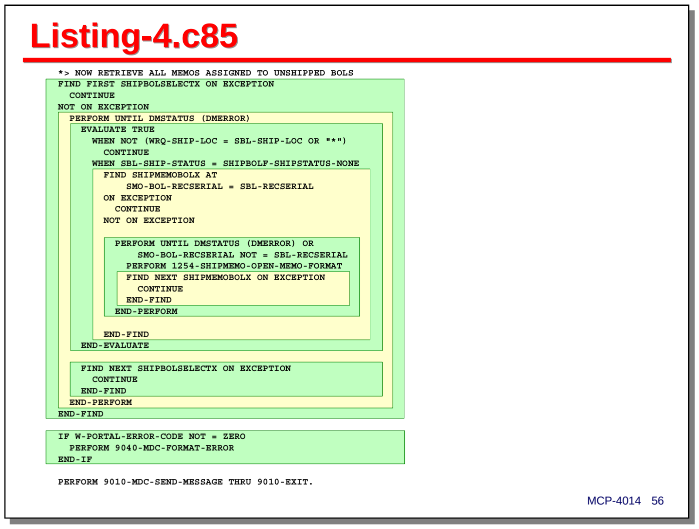
Where it's not so good is that deeply-nested code does not mix well with longer identifiers (which are good for clarity and maintainability of the code) and the 61 columns available in the standard COBOL source record format for Margin B. You tend to run of out room for coding statements on one line fairly quickly, and breaking statements across lines can reduce the readability of the code.

The solution, once again, is to apply some reasonableness tests as you are coding. There are a number of things you can do to preserve the space that is available for indentation and nesting:

- Use a narrow indentation increment. Two columns seems to work best.
- Consider aligning exception clauses, such as **AT END** and **ON EXCEPTION**, with their host verb, thus:

```
READ MF-MASTER-FILE
AT END
  ...
END-READ
```

- Similarly, consider aligning **WHEN** clauses in the same column as their **EVALUATE** verb.
- Keep an eye on the depth of nesting, and especially on the frequency of statements breaking across lines. When statement breaks become too frequent, create a **PERFORM** routine to hold the more deeply-nested statements.

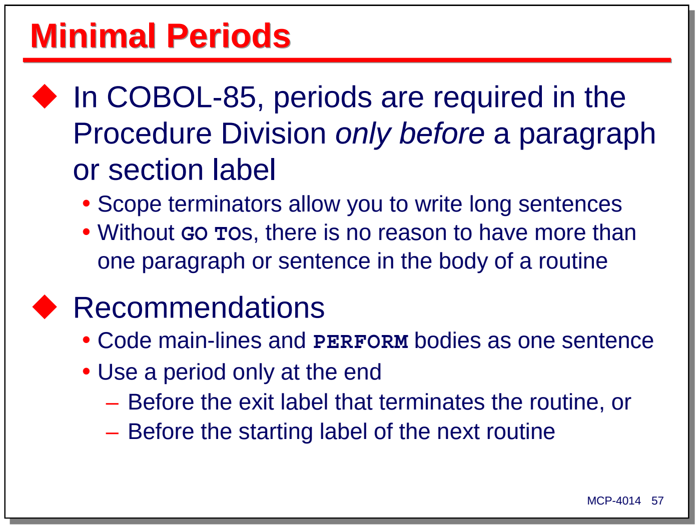


## Listing-4.c85

```
*> NOW RETRIEVE ALL MEMOS ASSIGNED TO UNSHIPPED BOLS
FIND FIRST SHIPBOLSELECTX ON EXCEPTION
  CONTINUE
NOT ON EXCEPTION
  PERFORM UNTIL DMSTATUS (DMERROR)
    EVALUATE TRUE
      WHEN NOT (WRQ-SHIP-LOC = SBL-SHIP-LOC OR "*")
        CONTINUE
      WHEN SBL-SHIP-STATUS = SHIPBOLF-SHIPSTATUS-NONE
        FIND SHIPMEMOBOLX AT
            SMO-BOL-RECSERIAL = SBL-RECSERIAL
        ON EXCEPTION
          CONTINUE
        NOT ON EXCEPTION

          PERFORM UNTIL DMSTATUS (DMERROR) OR
              SMO-BOL-RECSERIAL NOT = SBL-RECSERIAL
            PERFORM 1254-SHIPMEMO-OPEN-MEMO-FORMAT
            FIND NEXT SHIPMEMOBOLX ON EXCEPTION
              CONTINUE
            END-FIND
          END-PERFORM

        END-FIND
    END-EVALUATE

    FIND NEXT SHIPBOLSELECTX ON EXCEPTION
      CONTINUE
    END-FIND
  END-PERFORM
END-FIND

IF W-PORTAL-ERROR-CODE NOT = ZERO
  PERFORM 9040-MDC-FORMAT-ERROR
END-IF

PERFORM 9010-MDC-SEND-MESSAGE THRU 9010-EXIT.
```

MCP-4014 56

This slide shows a routine with several levels of nesting and how the structure of the routine is revealed by the degree of indentation. A consistent indentation style is critical to readability – and reliability – of the code. The eye is very good at inferring logic structure from physical structure, even when the two aren't related. Thus, it's very important that your indentation be consistent and follow the logical structure of your code.



## Minimal Periods

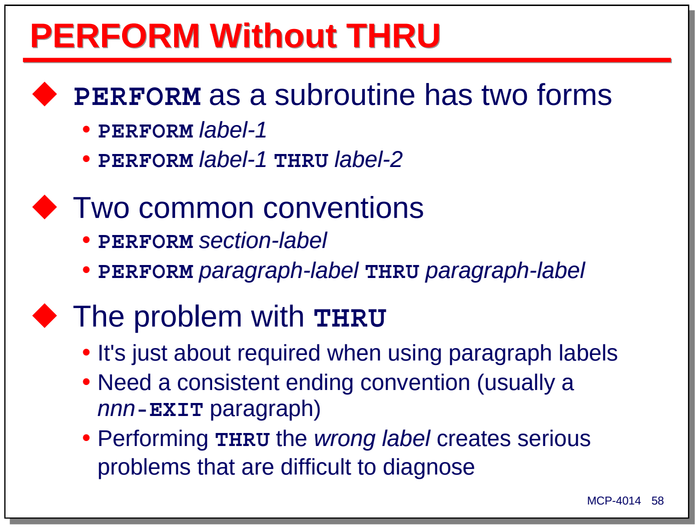
- In COBOL-85, periods are required in the Procedure Division *only before* a paragraph or section label
  - Scope terminators allow you to write long sentences
  - Without **GO TO**s, there is no reason to have more than one paragraph or sentence in the body of a routine
- Recommendations
  - Code main-lines and **PERFORM** bodies as one sentence
  - Use a period only at the end
    - Before the exit label that terminates the routine, or
    - Before the starting label of the next routine

MCP-4014 57

One of the things that surprised me when I started to code with the new COBOL-85 constructs is that you don't need periods nearly as much anymore. In COBOL-85, as with earlier COBOLs, periods are required in the Procedure Division only before a paragraph or section label. Since scope terminators and the other COBOL-85 features allow you to write longer sentences, you don't need as many periods. Further, without **GO TO** statements, you don't need labels within a routine, so there is no reason to break the body of a routine into multiple sentences – you can write the routine as one long sentence.

Writing routines as one long sentence turns out to have a number of advantages, especially if you need to move code around or change the indentation (say, to add an intervening **IF** statement). Since a period only needs to be at the end of the routine, you don't need to fix up any periods in the middle of the text when you make these kinds of changes.

I strongly recommend that you adopt this style aggressively and eliminate as many periods as you can, attempting to write your routines without **GO TO**s and internal labels, resulting in the routine being a single sentence. Then you only need place a period at the end of the routine, before the exit label that terminates the routine (if you use one) or before the starting label of the next routine.



## PERFORM Without THRU

- **PERFORM** as a subroutine has two forms
  - **PERFORM** *label-1*
  - **PERFORM** *label-1* **THRU** *label-2*
- Two common conventions
  - **PERFORM** *section-label*
  - **PERFORM** *paragraph-label* **THRU** *paragraph-label*
- The problem with **THRU**
  - It's just about required when using paragraph labels
  - Need a consistent ending convention (usually a *nnn*-**EXIT** paragraph)
  - Performing **THRU** the *wrong label* creates serious problems that are difficult to diagnose

MCP-4014 58

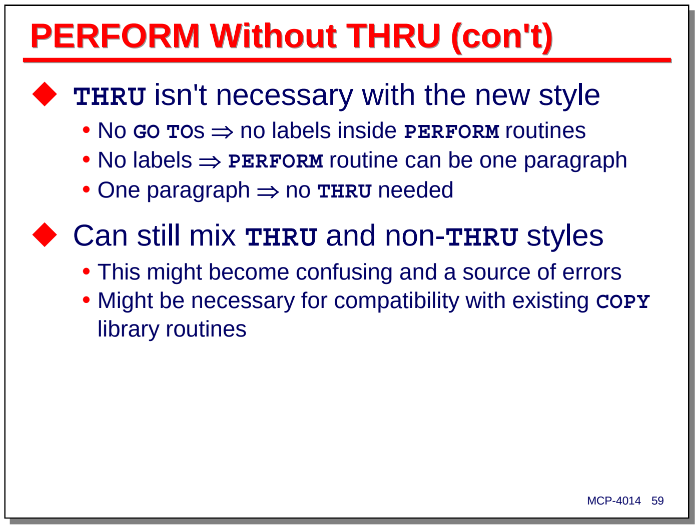
The final implication for my new style using the COBOL-85 constructs is that it is now possible to reliably construct **PERFORM** statements without a **THRU** clause.

**PERFORM** has two forms – addressing a single label, which will cause the sentences subordinate to that label to be executed, or addressing a range of labels using the **THRU** clause, which will cause the range of sentences starting with the first label, through and including the sentences subordinate to the second label. The labels can be either paragraph or section labels, and when using **THRU**, both labels must be of the same type.

In the classic COBOL style, there are two common conventions: performing a single section label and performing a range of paragraph labels. In my experience, the second one is the more common.

The **THRU** clause has some problems. First, it's just about required when performing paragraph labels. Performing a single paragraph label won't work if the routine being performed has internal labels. Since keeping straight in all cases whether the routine does or doesn't require the **THRU** can be both confusing and error prone, most people end up using **THRU** all the time, whether it's strictly necessary or not. This also implies that you need a consistent ending label for your routines, so most people code an ending paragraph with an *nnn*-**EXIT** label followed by an **EXIT** statement.

The really big problem with **THRU** is that it's quite easy to code the wrong ending label, especially when copying or cloning routines. Performing **THRU** the wrong label generates nasty bugs that are very difficult to analyze – the error usually becomes apparent some distance from where the bad **THRU** is coded, the program often exhibits *very* strange behavior just before the program completely messes up, and the final result is often a stack overflow fault, for which you can't get a dump. A bad **THRU** can ruin your whole day, along with a few of the ones immediately following.



## PERFORM Without THRU (con't)

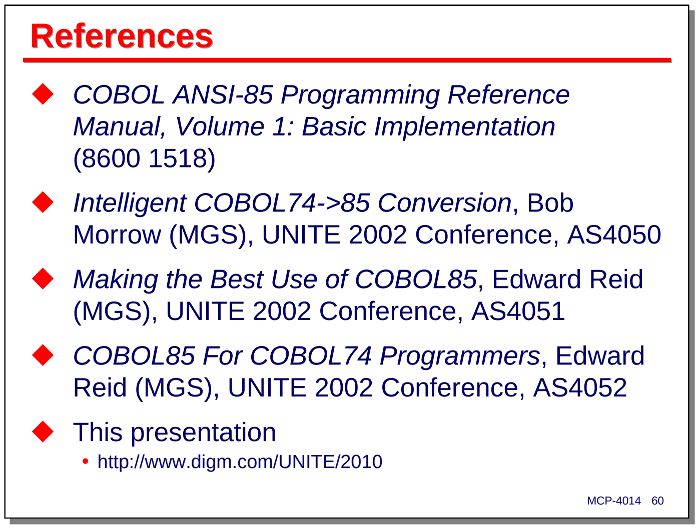
- **THRU** isn't necessary with the new style
  - No **GO TO**s ⇒ no labels inside **PERFORM** routines
  - No labels ⇒ **PERFORM** routine can be one paragraph
  - One paragraph ⇒ no **THRU** needed
- Can still mix **THRU** and non-**THRU** styles
  - This might become confusing and a source of errors
  - Might be necessary for compatibility with existing **COPY** library routines

MCP-4014 59

The good news with COBOL-85 is that **PERFORM** with **THRU** should no longer be necessary. If you don't use **GO TO**s, you don't need to have any internal labels in your routines. If you don't have any internal labels, your routines can consist of one paragraph (and, as pointed out earlier, just one sentence). If your routines are always just one paragraph, you don't need an exit paragraph, and don't need to **PERFORM** them with **THRU**.

You can mix the **THRU** and non-**THRU** styles, but that generally isn't a good idea. You can easily lose track whether a routine requires a **THRU** or not. That can be another nasty source of bugs.

I must confess that I have not yet made the transition to **THRU**-less **PERFORM**s. The reason is that the applications that I work on, while being compiled all with COBOL-85, are still mainly in the older style. This is especially true for the libraries common routines embedded in **COPY** modules. Most of those still need to be performed with **THRU**, and thus to keep things simple and consistent, I'm still stuck using the **THRU** clause.



## References

- *COBOL ANSI-85 Programming Reference Manual, Volume 1: Basic Implementation* (8600 1518)
- *Intelligent COBOL74->85 Conversion*, Bob Morrow (MGS), UNITE 2002 Conference, AS4050
- *Making the Best Use of COBOL85*, Edward Reid (MGS), UNITE 2002 Conference, AS4051
- *COBOL85 For COBOL74 Programmers*, Edward Reid (MGS), UNITE 2002 Conference, AS4052
- This presentation
  - http://www.digm.com/UNITE/2010

MCP-4014 60

The primary reference for the topics discussed in this presentation is Volume 1 of the MCP COBOL-85 reference manual. This is available without charge from the Unisys support site, http://support.unisys.com.

I want to recognize three excellent presentations from the 2002 UNITE conference on COBOL-85 and its usage, by Bob Morrow and Edward Reid, both at the time presenting on behalf of MGS, Inc. Much of what they say aligns with my comments in this presentation, but they have a different take on things in some areas, and you might find their viewpoint interesting and informative.

Finally, a copy of this presentation is available on our web site under the URL shown on the slide.

If you are interested in examining some larger examples using the newer COBOL-85 constructs and the style I've evolved from them, check out the sample code resources of these presentations from earlier UNITE conferences:

- *DMSQL Query Capabilities and Performance* (2008 UNITE, MCP-4032/4033) http://www.digm.com/UNITE/2008/
- *Using Application Data Access* (2009 UNITE, MCP-4021) http://www.digm.com/UNITE/2009/

## End

## Using – Really Using – COBOL-85

2010 UNITE Conference

Session MCP-4014

## Listing-A.c85

```
 1282-SHIPMEMO-DISPLAY-ORDER.
*    LOADS AND FORMATS FIELDS FROM THE RELATED OEFORDM RECORDS
*    FOR THE CURRENT MEMO.

     FIND OEORDERSEQX AT
         M-ORD-MAIN = SMO-ORDER-SEQ-NBR AND
         M-ORD-SUB = SMO-ORDER-BACK-NBR
     ON EXCEPTION
       MOVE "*ORDERSEQNBR" TO W-FIELD-NAME
       MOVE "No order record" TO W-FIELD-TEXT
       CALL "MDC_FORMAT_FIELD IN MDCLIB" USING
           W-FIELD-NAME, W-FIELD-TEXT, W-TEXT-SIZE,
           FCR-COMS-REPLY, W-OUT-SIZE
     NOT ON EXCEPTION
       MOVE "COCODE" TO W-FIELD-NAME
       COMPUTE W-L = FUNCTION LENGTH (M-CORP)
       CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB" USING
           W-FIELD-NAME, M-CORP, W-L,
           FCR-COMS-REPLY, W-OUT-SIZE

       IF M-SHIP-TEST = "*"
         MOVE "SHIPTOADDRFLAG" TO W-FIELD-NAME
         COMPUTE W-L = FUNCTION LENGTH (M-SHIP-TEST)
         CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB" USING
             W-FIELD-NAME, M-SHIP-TEST, W-L,
             FCR-COMS-REPLY, W-OUT-SIZE
       END-IF
     END-FIND.
```

## Listing-B.c85

```
 1282-SHIPMEMO-DISPLAY-ORDER.
*    LOADS AND FORMATS FIELDS FROM THE RELATED OEFORDM RECORDS
*    FOR THE CURRENT MEMO.

     FIND OEORDERSEQX AT
         M-ORD-MAIN = SMO-ORDER-SEQ-NBR AND
         M-ORD-SUB = SMO-ORDER-BACK-NBR
     ON EXCEPTION
       MOVE "*ORDERSEQNBR" TO W-FIELD-NAME
       MOVE "No order record" TO W-FIELD-TEXT
       CALL "MDC_FORMAT_FIELD IN MDCLIB" USING
           W-FIELD-NAME, W-FIELD-TEXT, W-TEXT-SIZE,
           FCR-COMS-REPLY, W-OUT-SIZE
       END-CALL     *> CALL ALSO HAS "ON EXCEPTION" !
     NOT ON EXCEPTION
       MOVE "COCODE" TO W-FIELD-NAME
       COMPUTE W-L = FUNCTION LENGTH (M-CORP)
       CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB" USING
           W-FIELD-NAME, M-CORP, W-L,
           FCR-COMS-REPLY, W-OUT-SIZE

       IF M-SHIP-TEST = "*"
         MOVE "SHIPTOADDRFLAG" TO W-FIELD-NAME
         COMPUTE W-L = FUNCTION LENGTH (M-SHIP-TEST)
         CALL "MDC_FORMAT_FIXED_FIELD IN MDCLIB" USING
             W-FIELD-NAME, M-SHIP-TEST, W-L,
             FCR-COMS-REPLY, W-OUT-SIZE
       END-IF
     END-FIND.
```

## Listing-1.c74

```
*******************************************************************
 0100-SECTION SECTION.
*******************************************************************
 0100-EVENT-DISPATCH.
*    RECEIVES AND DISPATCHES INPUT MESSAGES FROM COMS AND TIMER
*    EVENTS.

     MOVE W-TRUE TO W-SERVER-ACTIVE.

 0100-EVENT-LOOP.
     PERFORM Q116-READ-SYSTEM-TIMER THRU Q116-EXIT.
     COMPUTE W-WAIT-DELTA = WDA-EOD-TIMESTAMP - WDA-SYS-TIMESTAMP.
     IF W-WAIT-DELTA > W-TICKLER-PERIOD
       MOVE W-TICKLER-PERIOD TO W-WAIT-DELTA
     ELSE IF W-WAIT-DELTA < ZERO
       MOVE ZERO TO W-WAIT-DELTA.

     WAIT W-WAIT-DELTA,
         ATTRIBUTE DCIINPUTEVENT OF MYSELF,
         ATTRIBUTE DCITASKEVENT OF MYSELF
         ATTRIBUTE ACCEPTEVENT OF MYSELF
         GIVING W-RESULT.

     PERFORM Q116-READ-SYSTEM-TIMER THRU Q116-EXIT.
     GO TO
         0100-01-TIMEOUT-EVENT
         0100-02-DCIINPUTEVENT
         0100-03-DCITASKEVENT
         0100-04-ACCEPTEVENT
         DEPENDING ON W-RESULT.

 0100-00-INVALID-EVENT.
     MOVE W-RESULT TO WM-STATUS-VALUE
     MOVE "Invalid WAIT result (0100)" TO WM-STATUS-TEXT
     PERFORM 9806-LOG-DISPLAY THRU 9806-EXIT
     CHANGE ATTRIBUTE STATUS OF MYSELF TO TERMINATED.

 0100-01-TIMEOUT-EVENT.
     PERFORM 0800-TIMEOUT-EVENT THRU 0800-EXIT.
     GO TO 0100-NEXT-EVENT.

 0100-02-DCIINPUTEVENT.
 0100-03-DCITASKEVENT.
     PERFORM 0110-COMS-RECEIVE-MESSAGE THRU 0110-EXIT.
     GO TO 0100-NEXT-EVENT.

 0100-04-ACCEPTEVENT.
     PERFORM 0700-ACCEPT-OPERATOR-INPUT THRU 0700-EXIT.
     GO TO 0100-NEXT-EVENT.

 0100-NEXT-EVENT.
     IF W-SERVER-ACTIVE = W-TRUE
       GO TO 0100-EVENT-LOOP.

 0100-EXIT.
     EXIT.
```

## Listing-2.c85

```
*******************************************************************
 0100-SECTION SECTION.
*******************************************************************
 0100-EVENT-DISPATCH.
*    RECEIVES AND DISPATCHES INPUT MESSAGES FROM COMS AND TIMER
*    EVENTS.

     MOVE W-TRUE TO W-SERVER-ACTIVE

     PERFORM UNTIL W-SERVER-ACTIVE = W-FALSE
       PERFORM Q116-READ-SYSTEM-TIMER THRU Q116-EXIT
       COMPUTE W-WAIT-DELTA = FUNCTION MAX (0,
           FUNCTION MIN (W-TICKLER-PERIOD,
               WDA-EOD-TIMESTAMP - WDA-SYS-TIMESTAMP))

       WAIT W-WAIT-DELTA,
           ATTRIBUTE DCIINPUTEVENT OF MYSELF,
           ATTRIBUTE DCITASKEVENT OF MYSELF,
           ATTRIBUTE EXCEPTIONEVENT OF MYSELF,
           ATTRIBUTE ACCEPTEVENT OF MYSELF
           GIVING W-RESULT

       PERFORM Q116-READ-SYSTEM-TIMER

       EVALUATE W-RESULT
         WHEN 1
           PERFORM 0800-TIMEOUT-EVENT

         WHEN 2 THRU 3
           PERFORM 0110-COMS-RECEIVE-MESSAGE

         WHEN 4
           PERFORM 0600-PROCESS-EXCEPTIONEVENT

         WHEN 5
           PERFORM 0700-ACCEPT-OPERATOR-INPUT

         WHEN OTHER
           MOVE W-RESULT TO WM-STATUS-VALUE
           MOVE "Invalid WAIT result (0100)" TO WM-STATUS-TEXT
           PERFORM 9806-LOG-DISPLAY
           CHANGE ATTRIBUTE STATUS OF MYSELF TO TERMINATED
       END-EVALUATE
     END-PERFORM.
```

## Listing-3.c85

```
 1212-SHIPMEMO-FIND-COMPLETE.
*    SEARCHES THE EXISTING MEMOS FOR THIS ORDER. SETS W-TRUE IN
*    W-EDIT-ERROR IF SOME MEMO IS ALREADY MARKED COMPLETE.

     FIND LAST SHIPMEMOORDERX AT
         SMO-ORDER-SEQ-NBR = M-ORD-MAIN
     ON EXCEPTION
       CONTINUE
     NOT ON EXCEPTION
       PERFORM TEST AFTER UNTIL DMSTATUS (DMERROR) OR
               SMO-ORDER-SEQ-NBR NOT = M-ORD-MAIN OR
               SMO-COMPLETE-FLAG = "Y"
         IF SMO-COMPLETE-FLAG = "Y"
           IF SMO-BOL-RECSERIAL NOT = ZERO
             FIND SHIPBOLX AT
                 SBL-RECSERIAL = SMO-BOL-RECSERIAL
             ON EXCEPTION
               CONTINUE
             NOT ON EXCEPTION
               IF SBL-SHIP-STATUS = SHIPBOLF-SHIPSTATUS-SHIPPED
                 MOVE W-TRUE TO W-EDIT-ERROR
                 MOVE WEM-ALREADY-COMPLETE TO
                     WMU-ORDER-SEQ-NBR-ERR
               END-IF
             END-FIND
           END-IF
         ELSE
           FIND PRIOR SHIPMEMOORDERX ON EXCEPTION
             CONTINUE
           END-FIND
         END-IF
       END-PERFORM
     END-FIND.
```

## Listing-4.c85

```
     *> NOW RETRIEVE ALL MEMOS ASSIGNED TO UNSHIPPED BOLS
     FIND FIRST SHIPBOLSELECTX ON EXCEPTION
       CONTINUE
     NOT ON EXCEPTION
       PERFORM UNTIL DMSTATUS (DMERROR)
         EVALUATE TRUE
           WHEN NOT (WRQ-SHIP-LOC = SBL-SHIP-LOC OR "*")
             CONTINUE
           WHEN SBL-SHIP-STATUS = SHIPBOLF-SHIPSTATUS-NONE
             FIND SHIPMEMOBOLX AT
                 SMO-BOL-RECSERIAL = SBL-RECSERIAL
             ON EXCEPTION
               CONTINUE
             NOT ON EXCEPTION

               PERFORM UNTIL DMSTATUS (DMERROR) OR
                   SMO-BOL-RECSERIAL NOT = SBL-RECSERIAL
                 PERFORM 1254-SHIPMEMO-OPEN-MEMO-FORMAT THRU
                         1254-EXIT
                 FIND NEXT SHIPMEMOBOLX ON EXCEPTION
                   CONTINUE
                 END-FIND
               END-PERFORM

             END-FIND
         END-EVALUATE

         FIND NEXT SHIPBOLSELECTX ON EXCEPTION
           CONTINUE
         END-FIND
       END-PERFORM
     END-FIND

     IF W-PORTAL-ERROR-CODE NOT = ZERO
       PERFORM 9040-MDC-FORMAT-ERROR THRU 9040-EXIT
     END-IF

     PERFORM 9010-MDC-SEND-MESSAGE THRU 9010-EXIT.
```