Paper SM05

Write Concise ADaM Programs with If-less Programming Techniques

Lei Zhang, Bristol Myers Squibb, USA
Wei Shao, Bristol Myers Squibb, USA

ABSTRACT

ADaM is a widely adopted CDISC standard for analysis datasets in clinical trials. Conditional statements, or blocks like IF-THEN-ELSE or SELECT-WHEN are commonly used to derive ADaM variables. However, excessive use of conditional code makes ADaM programs cumbersome, hard to read and modify. This paper examines methods to eliminate conditional blocks through if-less programming techniques. It first covers basic if-less programming approaches in SAS© and then focuses on how to develop new mapping macros to generalize them. The way to construct these mapping macros is fully explained along with practical code in ADaM scenarios such as flag creation, partial date imputation, and complex variable derivation. The paper underscores the benefits of if-less programming techniques for improving the readability and reusability of ADaM programs. Additionally, the techniques discussed are versatile enough to be adopted in many other programming situations, extending their use beyond ADaM programs.

INTRODUCTION

The Analysis Data Model (ADaM) is one of the key standards in clinical trials. It provides guidelines on how to create analysis datasets for regulatory reporting and submission purposes. Based on the *Study Data Technical Conformance Guide* [1], the ADaM dataset programs are part of components in FDA submission, and they are important and must be well-written, easy to read and understand.

However, writing good ADaM programs is not a trivial task. This is because an ADaM dataset often has a dozen variables to derive under varying conditions apart from dataset merging. When the derived variables are coded with numerous if-else and select-when blocks, the ADaM program becomes cumbersome and error-prone. Besides, since the ADaM dataset specification is a controlled document that is updated throughout the life of the trial, if the corresponding program is full of conditional blocks, it becomes difficult to modify and maintain along with the evolving specification.

To understand this issue better, let us look at some examples. Below are the abridged definitions of 7 variables in an annotated ADSL dataset specification of a randomized study.

Dataset Name	Variable Name	Variable Label	Source/Derivation
ADSL	ENRLFL	Enrolled Population Flag	Set to "Y" for all subjects who have signed informed consent (or RFICDT value is not missing); otherwise set to "N".
ADSL	RANDFL	Randomized Population Flag	Set to "Y" for all enrolled subjects (ENRLFL = "Y") who were randomized (not missing RANDDT); otherwise set to "N".
ADSL	SAFFL	Safety Population Flag	Set to "Y" for all enrolled subjects (ENRLFL = "Y") who received at least one dose of study drug (TR01SDT value is not missing); Otherwise set to "N".
ADSL	BRTHDT	Date of Birth	If birth date (DM. BRTHDTC) is not missing: (1) If birth month and year are provided then impute birth date to day 15th of the birth month and year. (2) If only birth year is provided: (a) If birth year is less than or equal to (informed consent year – 3) then impute birth date to July 1st of the birth year; (b) Else leave imputed birth date value missing.
ADSL	REGION1N	Geographic Region 1 (N)	Assign a unique numeric code corresponding to each value of REGION1. 1 = US/CANADA 2 = EUROPE 3 = REST OF WORLD 4 = NOT REPORTED
ADSL	AGEGR1	Pooled Age Group 1 (C)	Set to “<18” if AAGE < 18; Set to “18 <= and <65” if AAGE >= 18 and AAGE < 65; Set to “>=65” if AAGE >= 65

ADSL	AGEGR1N	Pooled Age Group 1 (N)	Set to 1 if AAGE < 18; Set to 2 if AAGE >= 18 and AAGE < 65; Set to 3 if AAGE >= 65
ADSL	LSTALVDT	Date Last Known Alive	If death date is not missing, set to death date (DTHDT); else if subject discontinued treatment, set to last dose date (DISCDT) + 14; else set to last visit date (LSTVDT)

As you see, each of the above variables is supposed to be derived with one or multiple conditional statements. One of the traditional implementations is shown below:

```
DATA ADSL;
...

    /* ENRLFL */
    if not missing (RFICDT) then ENRLFL = 'Y';
    else ENRLFL = 'N';

    /* RANDFL */
    RANDFL="N";
    if missing(RANDDT) then do;
       if ENRLFL = "Y" then RANDFL = "Y";
    end;

    /* SAFFL */
    If ENRLFL='Y' and not missing(TR01SDT) then SAFFL='Y';
    else SAFFL='N';

    /* BRTHDT */
    BRTHDT=.;
    if not missing(BRTHDTC) then do;
      if length(strip(BRTHDTC))=10 then do;
       BRTHDT=input(strip(BRTHDTC), yymmdd10.);
      end;
     else do;
      if length(strip(BRTHDTC))=7 then do;
       BRTHDT=input(strip(BRTHDTC)||'-15', yymmdd10.);
      end;
      else do;
       if length(strip(BRTHDTC))=4 then do;
          if input(BRTHDTC, 4.) <= (year(RFICDT) - 3)  then
                      BRTHDT=input(strip(BRTHDTC)||'-07-01', yymmdd10.);
       end;
      end;
     end;
    end;

     /* AGEGR1N */
    if not missing(AAGE) then do;
      if AAGE < 18 then AGEGR1N=1;
      else if 18 =< AAGE < 65 then AGEGR1N=2;
      else AGEGR1N=3;
    end;

    /* REGION1N */
    If REGION1="US/CANADA" then REGION1N=1;
    else if REGION1="EUROPE" then REGION1N=2;
    else if REGION1="REST OF WORLD" then REGION1N=3;
    else REGION1N=4;

    /* LSTCTDT */
    If nmiss(DTHDT)=0 then LSTCTDT=DTHDT;
    else if nmiss(DISCDT)=0 then LSTCTDT=DISCDT;
    else if nmiss(LSTDDT)=0 then LSTCTDT= LSTDDT+14;
    else LSTCTDT=LSTVDT;
...
RUN;
```

Compared with the assignment statement, the coding above reveals the following issues:

- The code is difficult to understand because a condition statement involves two or more branches.
- The code is difficult to change because the variable is essentially re-defined at each conditional branch.
- The code is difficult to test or debug, since each conditional branch must be read and tested.

The if-less programming is an approach aimed at minimizing the use of conditional statements. Research indicates that conditional statements often lead to more verbose code compared to assignment statements [2][3][4]. This paper will demonstrate how if-less programming can simplify ADaM coding by eliminating conditional statements. By utilizing these methods, a statistical programmer can reduce code complexity, decrease program size, and enhance both readability and reusability, particularly for the ADaM programs heavily laden with conditional blocks.

IF-LESS PROGRAMMING

Conditional statements are a fundamental aspect of SAS® and other procedural programming languages, yet they often demand more effort for writing and testing than straightforward assignment statements. When constructing an ADaM variable, assignment statements should be favored as they enhance the readability, testability, and reusability of the program. Fortunately, SAS offers a couple traditional techniques to facilitate the coding.

One of methods is to replace the if-else statements with informats or formats by using function PUT() or INPUT(). For example, variable REGION1N and AGEGR1 can be derived with format and informat respectively using following alternative code:

```
PROC FORMAT;
   Invalue region1n_infmt
            "US/CANADA"=1
            "EUROPE"=2
            "REST OF WORLD"=3
            Other=4
   ;
   value agegr1n_fmt
            1 = "<18"
            2 = ">=18 AND <65"
            3 = ">=65"
   ;
RUN;

DATA ADSL;
...

/* REGION1N */
REGION1N=INPUT(REGION1, REGION1N_INFMT.);
       . . .
/* AGEGR1 */
AGEGR1=PUT(AGEGR1N, AGEGR1N_FMT.);
...
RUN;
```

The method of using format and informat works very well with single input variable, especially when there are many values to map or convert for a variable, but this method bears following problems:

- It can't map multiple variables. For example, deriving the safety flag variable SAFFL from variable ENRLFL and TR01SDT.
- The format or informat must be defined somewhere else before use, either in a different part of the code, or even in a different file, which causes fragmented code.
- It may be used sometimes more than what is needed. For example, using this method to derive TRT01A and TRT01AN would result in longer code due to extra formats.

The second method is to use the function `IFN()` and `IFC()` to replace if-else blocks, and `WHICHN()`, `CHOOSEN()` `CHOOSEC()`, or their combinations to replace select-when blocks when deriving ADaM variables. For example, `ENRLFL`, `RANDFL` and `REGION1N` can be implemented with following code:

```
DATA ADSL;
  ...
/* ENRLFL */
  ENRLFL=IFC(NOT Missing(RFICDT), "Y", "N");

/* RANDFL */
```

```
  RANDFL=IFC(ENRLFL = "Y" and not missing(RANDDT), "Y", "N");

/* REGION1N */
  REGION1N=CHOOSEN(1+WHICHC (REGION1, "US/CANADA", "EUROPE", "REST OF WORLD"),4,1,2,3);

  ...
RUN;
```

Here are the advantages of using `IFN()/CHOOSEN()` functions and their cousins to replace if-else or select-when statements.

- A programmer can express conditional logic(s) with multiple input variables and the result in a functional way, that is, `y=IFN(X1, X2,. . .)`, making it more concise compared to writing out an entire if-else block.
- A programmer can change a conditional block into a character or numeric function and use it as one code-liner in the same or other functions, which also results in shorter code.

However, there exist some issues with this approach:

- `IFN()` or `IFC()` work well as a replacement of a simple if-else statement, but they become complex when multiple choices are needed. For example, when using `IFC()` to derive AGEGR1 variable, one would write the below code using nested `IFC()` function calls, which makes the code hard to write and read.

```
DATA ADSL;
  Length AGEGR1 $16;

  /* AGEGR1 */
  AGEGR1=IFC(nmiss(aage)=1," ",IFC(AAGE <18, '<18'
              ,IFC(18 <= AAGE <65, '>= 18 and <65'
              ,IFC(AAGE>=65,'>=65',' '))));

  . . .
RUN;
```

- Sometimes it can be challenging to use `IFN(), IFC(), CHOOSEC(),CHOOSEN()` and `WHICHN()` to replace select-when blocks, when additional coding tricks are required, like the one shown in the alternative implementation of variable REGION1N. The example code returns different values based on the value returned from `WHICHC()` call, which seems not easy to understand.

As we see, the built-in SAS functions, `IFN(), CHOOSEN(),` and similar ones can work together to derive variables with some complexity. The further enhancement is to simplify their usage with easier syntax and logic. This results in mapping macro functions that combine `IFN(), IFC(), CHOOSEN(), CHOOSEC(), WHICHN()`, and `COMPARE()`. The next section details each mapping macro's syntax with examples. Note their implementation code will be provided in the appendix.

MAPPING MACRO FUNCTIONS

As we see above, the SAS built-in functions such as `IFN(), IFC(), WHICHN(), CHOOSEN(),` and `CHOOSEC()` can replace conditional blocks to derive ADaM variables with assignment statements, but they become challenging when dealing with complex or nested conditions. SAS macro is a simple and efficient method to encapsulate these functions with new syntaxes and lexical transformations. Composing these functions in a macro allows us to handle conditions and actions as parameters (or mapping rules) with a simplified and uniform syntax, ensuring the mapped value is returned clearly and consistently. Since the derivation of ADaM variables often involves casting numeric, character, or logical expressions to numeric or character values, we refer to these as mapping macro functions.

Below is a table listing the common ones.

Mapping Macro Function	Purpose
%N2C()	Maps a numeric value to a character value based on a list of num-to-char mapping rules.
%C2N()	Maps a character value to a numeric value based on a list of char-to-num mapping rules.
%C2C()	Maps a character value to a character value based on a list of char-to-char mapping rules.
%COND2C()	Map a condition or a logical expression to a character value based on a list of condition-to-char mapping rules.

%COND2N()	Maps a condition or a logical expression to a numeric value based on a list of condition-to-num mapping rules.

Mapping macro functions enable programmers to define a series of mapping rules that consist of conditions and matching actions, execute them when called, and yield the result when the first condition is met. Since they are purely implemented with composite of functions, they are so called inline macro functions. They offer a simple, clear, and efficient way to define a variable under different conditions without using any conditional statements.

%N2C

`%N2C()` is a mapping macro function that returns character value by comparing a numeric value against a set of values in the given mapping rules. It is implemented with the combination of `WHICHN()` and `CHOOSEC()`, and can take different numbers of mapping rules. It has the following syntax.

Syntax

CharVar=%**N2C**(
NumExp /* Numeric constant, variable, or expression*/
,Rule1 /* Num-to-char mapping rule 1 */
<,Rule2, Rule3, . . . Rulek> /* Mapping rule 2,3, . . . k */

,OTHER=' ' /* Character expression returned when no mapping rule works */
,SEP=# /* Separator between two items of a mapping rule */
)

`%N2C` takes a number (given by NumExp) and a list of mapping rules (Rule-1, Rule-2, ..., Rule-k) as inputs. Each mapping rule usually consists of two parts (separated by '#' or other symbol defined by SEP=). The first part (expressed as a numeric constant, variable, or expression) is a number that is used to compare with NumExp, and the second part (expressed as a character constant, variable, or expression) is a character value to return when the match occurs. If the first part of a mapping rule is dropped, then the order number (or index) of the mapping rule is used for the comparison. The %N2C checks each mapping rule in the order in which they are listed in the macro and returns the character value given by the first matching rule. If no rule matches, %N2C returns the character value given by OTHER=, or blank by default.

The following code demonstrates how %N2C() is used in a data step:

```
DATA N2C;
 Length  x1 x2 8 y1 y2 $20;
 input x1@@;
 x2=-1;
 y1=%N2C(x1,"Yes","No", Other="Missing");
 y2=%N2C(x1, (x2+3)~"Yes",x2~"No", Other="Missing", sep=~);
 put "Output: " x1= x2= y1= y2=;
datalines;
1 2 -1 5.1 .
;
RUN;
```

The results in the log:

```
Output: x1=1 x2=-1 y1=Yes y2=Missing
Output: x1=2 x2=-1 y1=No y2=Yes
Output: x1=-1 x2=-1 y1=Missing y2=No
Output: x1=5.1 x2=-1 y1=Missing y2=Missing
Output: x1= x2=-1 y1=Missing y2=Missing
```

%C2N

`%C2N()` maps a character value to a numeric value by checking the value against a list of char-to-num mapping rules. It is similar to `%N2C()` in both syntax and usage except that it returns a character value instead of numeric one. Below is its usage:

Syntax

NumVar=%**C2N**(
CharExp /* Character constant, variable, or expression*/
,Rule1 /* Char-to-num mapping rule 1 */
<,Rule2, Rule3, . . . Rulek> /* Mapping rule 2,3, . . . k */

,OTHER=. /* Numeric expression returned when no mapping rule works */
,SEP=# /* Separator between two items of a mapping rule */
)

The following code demonstrates its uses:

```
DATA C2N;
 Length y1 y2 8 x $16;
 input x $ 1-16;
 y1=%C2N(x,"Yes","No", Other=-9, MOD=":");
 y2=%C2N(x,"Y":2,"N":1, Other=-8, MOD=":i", SEP=:);
 put "Output: " x= y1= y2=;
datalines;
Yes
No
YES
NO
Y
N

;
RUN;
```

The results in the log:

```
Output: x=Yes y1=1 y2=2
Output: x=No y1=2 y2=1
Output: x=YES y1=-9 y2=2
Output: x=NO y1=-9 y2=1
Output: x=Y y1=1 y2=2
Output: x=N y1=2 y2=1
Output: x=  y1=-9 y2=-8
```

%C2C

`%C2C()` maps a character value to another character value by checking the value against a list of char-to-char mapping rules. It is like `%N2C()` in both syntax and usage. Below is its usage:

Syntax

CharVar=%**C2C**(
CharExp /* Character constant, variable, or expression*/
,Rule1 /* Char-to-char mapping rule 1 */
<,Rule2, Rule3, . . . Rulek> /* Mapping rule 2,3, . . . k */

,OTHER=' ' /* Character expression returned when no mapping rule works */
,SEP=# /* Separator between two items of a mapping rule */
)

The following example demonstrates its uses:

```
DATA C2C;
 Length y1 y2 $20 x $16;
 input x $ 1-16;
 y1=%C2C(x,"Yes","No", Other="UNK", mod="i");
 y2=%C2C(x,"Yes"#"Y","No"#"N", Other="UNK", mod=":");
 put "Output: " x= y1= y2=;
datalines;
Yes
No
Yes, Sir
Y
```

```
N

;
RUN;
```

The results in the log:

```
Output: x=Yes y1=1 y2=Y
Output: x=No y1=2 y2=N
Output: x=Yes, Sir y1=UNK y2=Y
Output: x=Y y1=UNK y2=Y
Output: x=N y1=UNK y2=N
Output: x=  y1=UNK y2=UNK
```

%COND2N

%COND2N() maps a condition or logical expression to a numeric value via a set of condition-to-num mapping rules. Unlike %N2C(), %C2N() and %C2C(), The mapping rules are listed in the positional arguments without the first expression for the comparison. Each rule contains two parts (separated by #). The first part is a logical expression, or condition to be evaluated, and the second part is the numeric expression with the value to be returned when the first part is evaluated to TRUE (or 1). Like %N2C() or %C2C(), it also has two key parameters OTHER= and SEP= for a user to change the default value to be returned and rule separator. Below is its usage:

Syntax

NumVar=%**COND2N**(
Rule1 /* Condition-to-num mapping rule 1 */
<,Rule2, Rule3, . . . Rulek> /* Mapping rule 2,3, . . . k */

,OTHER=. /* Numeric expression returned when no mapping rule works */
,SEP=# /* DLM between two parts of a mapping rule */
)

%COND2N() evaluates the first part of each mapping rule, from left to right, until it finds the first rule with the condition being evaluated to TRUE, and then returns the numeric value of the second part. If none of conditions are TRUE, then the numeric value provided by key parameter OTHER= is returned. If a rule is provided without the numeric value part, %COND2N() takes the order ID of the rule as the value to return. Below is the code that demonstrates its uses.

```
DATA COND2N;
 Length x y 8;
 input x@@;
 y=%COND2N(0<=x<5, 5<=x<10#110, (x=10)#120, x>=10, other=-1);
 put "Output: " x= y=;
datalines;
1.5 5 10 15.5 -10 .
;
RUN;
```

The results in the log:

```
Output: x=1.5 y=1
Output: x=5 y=110
Output: x=10 y=120
Output: x=15.5 y=4
Output: x=-10 y=-1
Output: x= y=-1
```

%COND2C

%COND2C() maps a condition or logical expression to a character value via a set of condition-to-char mapping rules. The mapping rules are listed in the positional arguments. Each rule contains two parts (separated by #). The first part is a logical expression, or condition to be evaluated, and the second part is the character expression with the value to be returned when the first part is evaluated to TRUE (or 1). Like %COND2C(), it also has two key parameters OTHER= and SEP= too. Below is its usage:

Syntax

CharVar=%COND2C(
Rule1 /* Condition-to-character mapping rule 1 */
<,Rule2, Rule3, . . . Rulek> /* Mapping rule 2,3, . . . k */

,OTHER=. /* Character expression returned when no mapping rule works */
,SEP=# /* DLM between two parts of a mapping rule */
)

`%COND2C()` evaluates the first part of each mapping rule, from left to right, until it finds the first rule with the condition being evaluated to TRUE, and then returns the character value. If none of conditions are TRUE, then the character value provided by key parameter OTHER= is returned. If a rule is provided without the second part, `%COND2C()` takes the order ID of the rule as the return value. Below is the code that demonstrates its uses.

```
DATA COND2C;
  Length x 8 y $20;
  input x@@;
  y=%COND2C(0<=x< 5#"LT 5", 5<=x<10#"5 GE & LT 10", x>=10#"GE 10", other="Missing");
  put "Output: " x= y=;
datalines;
1.5 5 10 15.5 -10 .
;
RUN;
```

The results in the log:

```
Output: x=1.5 y=LT 5
Output: x=5 y=5 GE & LT 10
Output: x=10 y=GE 10
Output: x=15.5 y=GE 10
Output: x=-10 y=Missing
Output: x= y=Missing
```

Rewriting the first example with mapping macro functions

With the above five mapping macros, the ADSL variables in the previous example can be re-implemented as follows:

```
DATA ADSL;
...
  /* ENRLFL */
  ENRLFL=IFC(NOT Missing(RFICDT), "Y", "N");

  /* RANDFL */
  RANDFL=IFC(ENRLFL = "Y" and not missing(RANDDT), "Y", "N","N");

  /* SAFFL */
  SAFFL=IFC(ENRLFL='Y' and not missing(TR01SDT), "Y", "N","N");

  /* REGION1N */
  REGION1N=%C2N(REGION1, "US/CANADA", "EUROPE", "REST OF WORLD", other=4);

  /* BRTHDDT */
  BRTHDT=input(%COND2C(lengthn(BRTHDTC)=10 # strip(BRTHDTC)
     ,lengthn(BRTHDTC)=7 # strip(BRTHDTC)||'-15'
     ,lengthn(BRTHDTC)=4 & input(BRTHDTC, 4.)<=(year(RFICDT) - 3)  # strip(BRTHDTC)||'-07-01'
     ,other=' '), ??yymmdd10.);


  /* AGEGR1N */
  AGEGR1N=%COND2N( 0 < AAGE<18 # 1 ,18<= AAGE <65 # 2,65 <=AAGE # 3);

  /* LSTCTDT */
  LSTCTDT=%COND2N(
       (nmiss(DTHDT)=0)#DTHDT
      ,(nmiss(DISCDT)=0)#DISCDT
      ,(nmiss(LSTDDT)=0)#LSTDDT+14
      , OTHER=LSTVDT);
...
RUN;
```

Obviously, the mapping macro functions enable a programmer to write much less code for ADaM variable derivation, and bring up the following additional benefits:

- Like any other SAS built-in functions, they can be used in the data steps, PROC SQL, or even PROC FCMP without any side effects, and they can be called within SAS functions and themselves.
- The code snippets written with the mapping macro calls can be conveniently stored in macro variables, and be combined to form new macros.
- The simple syntax of the mapping macro functions strings all rules (since each rule is consisted of the condition and code to be performed) in a sequence of conditional operations (or a mapping list) for variable derivation, eliminating the uses of complex and nesting function calls.
- The numeric or character expressions constructed are concise and compact, having all the required information gathered in one place. Therefore, it is easy to infer how the code works, and find out how to change, and reuse them.

CONSTRUCTING COMPLEX MAROS WITH MAPPING MACROS

The inline mapping macros defined above can also be used to construct more powerful macros and derive ADaM variables based on complex logic. For example, if you want to derive the variables ASTDT and AENDT with the specification below, you can complete the derivation with the following steps.

Dataset Name	Variable Name	Variable Label	Source/Derivation
ADAE	ASTDT	Analysis Start Date	Adverse Event Start Date derivation. If date part of reported adverse event start date (AESTDTC) is a full date (i.e. not partial and not missing), set ASTDT to numeric date value of AESTDTC, and set ASTDTF to missing. If date part of reported adverse event start date (AESTDTC) is missing or partial, impute ASTDT as follows: 1. Derive a temporary variable containing the imputed adverse event end date (AETMPED) as follows: • If reported adverse event end date (AEENDTC) is a full date (i.e., not partial and not missing), set AETMPED equal to the numeric date value of AEENDTC • If reported adverse event end date (AEENDTC) is a partial date with only year and month provided, impute AETMPED to the last day of the month and year • If reported adverse event end date (AEENDTC) is a partial date with year provided and month not provided, impute AETMPED to December 31st of the year 2. Use imputed adverse event end date (AETMPED) and first dose date (ADCORE.TRTSDT) to determine how to impute ASTDT: a. If the reported adverse event start date (AESTDTC) is missing: - If AETMPED >= ADCORE.TRTSDT or AETMPED value is missing, impute ASTDT to ADCORE.TRTSDT - Else impute ASTDT to January 1st of the year in AETMPED Set ASTDTF equal to "Y" b. Else if the reported adverse event start date (AESTDTC) is partial with only year and month provided: - If month and year of AESTDTC is the same as month and year of ADCORE.TRTSDT, and AETMPED >= ADCORE.TRTSDT or AETMPED value is missing, impute ASTDT to ADCORE.TRTSDT - Else impute ASTDT to the first day of the month of the year in AESTDTC Set ASTDTF equal to "D"

			c. Else if the reported adverse event start date (AESTDTC) is partial with year provided and month not provided: - If year of AESTDTC is the same as year of ADCORE.TRTSDT, and AETMPED >= ADCORE.TRTSDT or AETMPED value is missing, impute ASTDT to ADCORE.TRTSDT - Else impute ASTDT to January 1st of the year in AESTDTC Set ASTDTF equal to "M"
ADAE	AENDT	Analysis End Date	Adverse event end date. If date part of reported adverse event end date (AEENDTC) is a full date (i.e. not partial and not missing), set AENDT equal to the numeric date value of AEENDTC. If the reported adverse event end date (AEENDTC) is missing, set AENDT to missing value. If the reported adverse event end date (AEENDTC) is partial, impute AENDT as follows: 1. If reported end date (AEENDTC) is partial with only year and month provided, impute AENDT to the last day of the month and year. If month and year of imputed AENDT is the same as month and year of <last reference date>, set AENDT to <last reference date variable>. 2. Else if reported end date (AEENDTC) is partial with year provided and month not provided, set AENDT to missing value.

First, write a macro called %IMP_YMD10 to impute a date string in YYYY-MM-DD format, such as 2024-08-19, 2024-07, or 2024. The macro can be written as follows.

```
%Macro IMP_YMD10(
  CDATE      /* Date string in YYYY-MM-DD format, such as 2024-08-19, 2024-07, or 2024 */
 ,MODE=B     /* If  &CDATE is a partial date, then impute it with the following mode:  */
             /* Mode = B, M, or E, same as INTNX(). Default is B */
              /*  B = impute the cdate as the beginning of year, or month */
             /*  M = impute the cdate as the middle of year, or month */
              /*  E = impute the cdate as the end of year, or month */
);

%COND2N(
   lengthn(&CDATE) EQ 10 # input(strip(&CDATE), ??YYMMDD10.)
  ,lengthn(&CDATE) EQ 7  # intnx('MONTH',input(strip(&CDATE)||"-01",??YYMMDD10.)
           , 0, %sysfunc(quote(&MODE)) )
  ,lengthn(&CDATE) EQ 4  # intnx('YEAR', input(strip(&CDATE)||"-01-01", ??YYMMDD10.)
           , 0,%sysfunc(quote(&MODE)))
)

%Mend;
```

Second, implement the macro %IMP_ENDT to impute the end date using %IMP_YMD10 as follows.

```
%Macro IMP_ENDT(
   CDATE      /* Date string in YYYY-MM-DD format (full or partial) */
  ,LastRefDT= /* Last reference date for the event (num)  */
);

%COND2N(
            lengthn(&CDATE) EQ 10 # input(strip(&CDATE), ??YYMMDD10.)
          , lengthn(&CDATE) EQ 7  and substrn(strip(&CDATE),1,4)=Year(&LASTREFDT)
            and substrn(strip(&CDATE),6,2)=MONTH(&LASTREFDT) #  &LASTREFDT
          , lengthn(&CDATE) EQ 7 # %imp_ymd10(&CDATE,MODE=E)
)

%Mend;
```

Third, implement the macro %IMP_STDT with %IMP_ENDT and %IMP_YMD10 as follows.

```
%Macro IMP_STDT(
      STDTC      /* Character start date variable in YYYY-MM-DD format (full or partial)  */
     ,ENDTC      /* Character end date variable in YYYY-MM-DD format (full or partial)   */
     ,TRTSDT     /* Treatment start date (num)                                           */
);

%local AETMPED STDT IMP_DAY IMP_MONTH IMP_YEAR;
%let AETMPED=%IMP_YMD10(&ENDTC, mode=E);
%let STDT=%IMP_YMD10(&STDTC, mode=B);

%let IMP_YEAR=%COND2N(nmiss(&TRTSDT)=0 and (&AETMPED >= &TRTSDT or nmiss(&AETMPED)=1) # &TRTSDT
                     ,Other=intnx('YEAR', &AETMPED, 0, "B") );

%let IMP_DAY=%COND2N(nmiss(&TRTSDT)=0 and year(&TRTSDT)=substrn(&STDTC,1,4)
             and month(&TRTSDT)=substrn(&STDTC,6,2) and (&AETMPED > &TRTSDT
             or nmiss(&AETMPED)=1 ) # &TRTSDT,Other=&stdt);

%let IMP_MONTH=%COND2N(nmiss(&TRTSDT)=0 and year(&TRTSDT)=substrn(&STDTC,1,4)
             and (&AETMPED > &TRTSDT  or nmiss(&AETMPED)=1) # &TRTSDT, Other=&stdt);

%COND2N( cmiss(&STDTC)=1 # &imp_year
        ,lengthn(&STDTC)=7 # &imp_day
        ,lengthn(&STDTC)=4 # &imp_month
        ,Other=&stdt
        )
%Mend;
```

With the above three macros, the variables ASRDT and AENDT can simply be derived by calling %IMP_STDT and %IMP_ENDT separately. Here is an illustrative example.

```
Data ADAE;
   . . .
   trtsdt="02JUL2024"D;
   lastrefdt="025JUL2024"D;

   aestdtc="2024-07";
   aeendtc="2024-07-17";

   astdt=%IMP_STDT(aestdtc, aeendtc, trtsdt);

   aendt=%IMP_ENDT(aeendtc, LastRefDT=lastrefdt);

   put  aestdtc= aeendtc= trtsdt= astdt=;
   format astdt aendt trtsdt lastrefdt yymmdd10.;
   . . .
Run;
```

MORE IF-LESS PROGRAMMING TECHNIQUES

The mapping macros introduced above allow a programmer to replace most of conditional blocks when creating ADaM variables. In case more code reduction is needed in certain situations, please consider the following extra techniques.

One is to use functions COALESCE, PRX, MIN and MAX functions to replace conditional blocks. For example, the function `COALESCEN()` can be used to get the first non-missing numeric value among several ones, the variable LSTCTDT thus can be derived with following one-line code:

```
LSTCTDT=COALESCE(DTHDT,DISCDT,LSTDDT+14,LSTDT);
```

Treating a logical expression as a numeric one is another technique to avoid conditional blocks.

Dataset Name	Variable Name	Variable Label	Source/Derivation
ADAE	ASTDY	Analysis Start Relative Day	ASTDT – ADSL.TRTSDT + 1 if ASTDT is on or after TRTSDT, else ASTDT – ADSL.TRTSDT if ASTDT precedes TRTSDT

For example, the variable ASTDY defined above can be derived with one- line assignment statement when we treat a

logical expression as a part of numeric expression.

```
ASTDY = ASTDT – TRTSDT + (ASTDT >= TRTSDT);
```

The third technique worthy to mention is to create user-defined functions with PROC FCMP so the conditional blocks and associated actions can be encapsulated and called by name from data steps when needed. The user-defined functions can perform the same functionalities as those inline mapping macros and further modularize your ADaM programs if you would like to spend extra time and effort to define and manage them. Below is the PROC FCMP code that re-implements %IMP_YMD10, %IMP_STDT and %IMP_ENDT in the form of user-defined function.

```
PROC FCMP Outlib=WORK.usrfunc.string;

 Function IMP_YMD10(
          CDATE $ /* Character date expression in YYMMDD10. */
         ,MODE $  /* Impution mode: B, M, and E           */
 );
 Length date 8;

 If lengthn(CDATE) EQ 10 then date= input(strip(CDATE), YYMMDD10.);
 else if lengthn(CDATE) EQ 7  then date=intnx('MONTH',input(strip(CDATE)
                                      ||"-01",YYMMDD10.), 0, MODE);
 else if lengthn(CDATE) EQ 4 then date=intnx('YEAR', input(strip(CDATE)
                                      ||"-01-01", YYMMDD10.), 0, MODE);
 return(date);
Endsub;

 Function IMP_ENDT(
          ENDTC $   /* End Date in character expression in YYMMDD10. */
         ,LASTREFDT /* Last reference date for the event             */
 );
 Length date 8;

 If lengthn(ENDTC) EQ 10 then date= input(strip(ENDTC), YYMMDD10.);
 else if lengthn(ENDTC) EQ 7 then do;
           if substrn(strip(ENDTC),1,4)=Year(LASTREFDT)
              and substrn(strip(ENDTC),6,2)=MONTH(LASTREFDT) then date=LASTREFDT;
           else date=imp_ymd10(ENDTC,"E");
 end;
 return(date);
Endsub;

 Function IMP_STDT(
          STDTC $ /* Start Date in character expression in YYMMDD10. */
         ,ENDTC $ /* End Date in character expression in YYMMDD10. */
         ,TRTSDT  /* Treatment start date */
 );
         Length AETMPED STDT IMP_DAY IMP_MONTH IMP_YEAR 8;

         AETMPED=IMP_YMD10(ENDTC, "E");
         STDT=IMP_YMD10(STDTC, "B");

         IMP_YEAR=%COND2N(nmiss(TRTSDT)=0 and (AETMPED >= TRTSDT or nmiss(AETMPED)=1) # TRTSDT
                                             , OTHER=intnx('YEAR', AETMPED, 0, "B"));

         IMP_DAY=%COND2N(nmiss(TRTSDT)=0 and year(TRTSDT)=substrn(STDTC,1,4)
            and month(TRTSDT)=substrn(STDTC,6,2) and (AETMPED > TRTSDT
            or nmiss(AETMPED)=1 ) # TRTSDT
                  , OTHER=STDT);

         IMP_MONTH=%COND2N(nmiss(TRTSDT)=0 and year(TRTSDT)=substrn(STDTC,1,4)
            and (AETMPED > TRTSDT  or nmiss(AETMPED)=1) # TRTSDT
                   , OTHER=STDT);

         STDT=%COND2N( cmiss(STDTC)=1 # imp_year
                      ,lengthn(STDTC)=7 # imp_day
                      ,lengthn(STDTC)=4 # imp_month
                      ,Other=STDT
       );

    return(STDT);
```

```
 Endsub;

RUN;

options cmplib=(WORK.usrfunc);

Data ADAE;
   . . .
   trtsdt="02JUL2024"D;
   lastrefdt="025JUL2024"D;

   aestdtc="2024-07";
   aeendtc="2024-07-17";

   astdt=IMP_STDT(aestdtc, aeendtc, trtsdt);

   aendt=IMP_ENDT(aeendtc, LastRefDT=lastrefdt);

   put  aestdtc= aeendtc= trtsdt= astdt=;
   format astdt aendt trtsdt lastrefdt yymmdd10.;
   . . .
Run;
```

Since the derivation of start and end dates occurs in other ADaM datasets (such as ADLB and ADCM) apart from ADAE, implementing them in user-defined function can greatly facilitate coding across many ADaM datasets, making the ADaM programs much shorter and simpler across study projects.

CONCLUSION

Adopting if-less programming practices is vital for developing ADaM programs that are maintainable, flexible, and efficient. Implementing the strategies explored in this paper enables statistical programmers to minimize dependency on conditional statements, enhancing code clarity. Furthermore, these techniques facilitate the easy modification and reuse of ADaM programs and macros, thereby reducing development time and effort. Embracing these approaches and principles leads to the creation of more robust and scalable software solutions for ADaM programs as well as other clinical applications

REFERENCES

1. U.S. Food and Drug Administration, “Study Data Technical Conformance Guide v3.0.” March 2016. Available at http://www.fda.gov/downloads/ForIndustry/DataStandards/StudyDataStandards/UCM384744.pdf
2. Lisnic, Andrei “If-less programming”. Available at http://alisnic.github.io/posts/ifless/
3. Jouan, Matthias “if-else trees vs SOLID principles”. Available at http://blog.mjouan.fr/if-else-trees-vs-solid-principles/
4. McCabe, THOMAS J., “A Complexity Measure”, *IEEE Transactions on Software Engineering*, VOL. SE-2, NO.4, DECEMBER 1976
5. SAS Institute, Inc. “SAS® 9.2 Language Reference: Dictionary, Fourth Edition”. Available at http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000245852.htm
6. Eberhardt, Peter, Shao, Lucheng. “Functioning at a Higher Level: Using SAS® Functions to Improve Your Code”, *Proceedings of the PharmaSUG 2014 Conference*
7. Zhang, Lei “Building Better ADaM Datasets Faster with If-less Programming”, *Proceedings of the WUSS 2016 Conference*

ACKNOWLEDGMENTS

The authors are grateful to Allison Covucci and Nicole Thorne for their kind review and editorial comments in the preparation of this paper.

DISCLAIMER: The opinions expressed in this presentation and on the following slides are solely those of the presenter and not necessarily those of Bristol Myers Squibb. Bristol Myers Squibb does not guarantee the accuracy or reliability of the information provided herein.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact author at:

Lei Zhang
Bristol Myers Squibb
lei.zhang2@bms.com

Wei Shao
Bristol Myers Squibb
wei.shao2@bms.com

Appendix 1: Sample Mapping Macro Functions (for illustrative and exercise purpose)

```
/* NOTE: the sample mapping macros only accept mapping rules with up to 8 */

/*$ N2C
 Purpose: Mapping a numeric value to a character value.
*/
%Macro N2C(
      NUMEXP   /* Numeric expression */
     ,RULE1, RULE2, RULE3, RULE4, RULE5, RULE6, RULE7, RULE8 /* Mapping RULEs */
     ,OTHER=" " /* Value to be returned if no RULEs are satisfied */
     ,SEP=# /* DLM between the first and second part of a mapping RULE */
);
        %LOCAL IDX N K MAXK XRULE;
        %LOCAL RULE_L_EXP RULE_R_EXP COND_LST EXP_LST;

        %let N=8;
        %Let IDX=0;
        %DO %WHILE (&IDX LT &N);
                %LET IDX=%EVAL(&IDX + 1);
                %if %length(&&RULE&IDX) EQ 0 %then %goto leave;
        %End;
        %leave:
        %if &IDX EQ &N %then %let MAXK=&N; %else %let MAXK=%eval(&IDX-1);

        %if &MAXK=0 %then %do;
                        %PUT ERROR: Empty numeric-to-character mapping list.;
                        %return;
        %end;

        %let K=&MAXK;

        %DO %UNTIL(&K=0);
                %let xRULE=&&RULE&K;
                %let RULE_L_EXP=%sysfunc(scan(&xRULE, 1, &sep, Q));
                %let RULE_R_EXP=%sysfunc(scan(&xRULE, 2, &sep, Q));

                %if %length(&RULE_R_EXP)=0 %then %do;
                        %let RULE_R_EXP=&RULE_L_EXP;
                        %Let RULE_L_EXP=&K;
                %end;

                %if &K=&MAXK %then %do;
                        %Let COND_LST=(&NUMEXP EQ &RULE_L_EXP);
                        %Let EXP_LST=&RULE_R_EXP;
                %end; %else %do;
                        %Let COND_LST=(&NUMEXP EQ &RULE_L_EXP)%str(,) &COND_LST;
                        %Let EXP_LST=&RULE_R_EXP%str(,) &EXP_LST;
                %end;

                %let K=%EVAL(&K-1);
        %End;

Choosec(1+whichn(1, &COND_LST),&OTHER,&EXP_LST)
%Mend;


/*$ C2N
```

```
 Purpose: Mapping a character value to a numeric value.
*/
%Macro C2N(
   CHAREXP   /* Character expression */
   ,RULE1, RULE2, RULE3, RULE4, RULE5, RULE6, RULE7, RULE8 /* Mapping rules (1-8) */
   ,OTHER=. /* Value to be returned if no rule is satisfied */
   ,SEP=# /* DLM between the first and second part of a mapping rule */
   ,MOD=" " /* Comparison modifiers, same as those used by COMPARE()    */

);
          %LOCAL IDX N K MAXK XRULE ENDFLAG;
          %LOCAL RULE_L_EXP RULE_R_EXP COND_LST EXP_LST;

          %let N=8;
          %Let IDX=0;
          %DO %WHILE (&IDX LT &N);
                    %LET IDX=%EVAL(&IDX + 1);
                    %if %length(&&RULE&IDX) EQ 0 %then %goto leave;
          %End;
          %leave:
          %if &IDX EQ &N %then %let MAXK=&N; %else %let MAXK=%eval(&IDX-1);

          %if &MAXK=0 %then %do;
                              %PUT ERROR: Empty character-to-numeric mapping list.;
                              %return;
          %end;

          %let K=&MAXK;
          %DO %UNTIL(&K=0);
                    %let xRULE=&&RULE&K;
                    %let RULE_L_EXP=%sysfunc(scan(&xRULE, 1, &sep, Q));
                    %let RULE_R_EXP=%sysfunc(scan(&xRULE, 2, &sep, Q));

                    %if %length(&RULE_R_EXP)=0 %then %let RULE_R_EXP=&K;

                    %if &K=&MAXK %then %do;
                              %Let
COND_LST=(compare(trimn(&CHAREXP),trimn(&RULE_L_EXP),&MOD)=0);
                              %Let EXP_LST=&RULE_R_EXP;
                    %end; %else %do;
                              %Let
COND_LST=(compare(trimn(&CHAREXP),trimn(&RULE_L_EXP),&MOD)=0)%str(,) &COND_LST;
                              %Let EXP_LST=&RULE_R_EXP%str(,) &EXP_LST;
                    %end;

                    %let K=%EVAL(&K-1);
          %End;

Choosen(1+whichn(1, &COND_LST),&OTHER,&EXP_LST)
%Mend;

/*$ C2C
 Purpose: Mapping a character value to a character value.
*/
%Macro C2C(
   CHAREXP   /* Character expression */
  ,RULE1, RULE2, RULE3, RULE4, RULE5, RULE6, RULE7, RULE8 /* Mapping rules */
  ,OTHER=" " /* Value to be returned if no rule is satisfied */
  ,SEP=# /* DLM between the first and second part of a mapping rule */
  ,MOD=" " /* Comparison modifiers, same as those used by COMPARE() */
);
        %LOCAL IDX N K MAXK XRULE ENDFLAG;
        %LOCAL RULE_L_EXP RULE_R_EXP COND_LST EXP_LST;

        %let N=8;
        %Let IDX=0;
        %DO %WHILE (&IDX LT &N);
                %LET IDX=%EVAL(&IDX + 1);
                %if %length(&&RULE&IDX) EQ 0 %then %goto leave;
        %End;
        %leave:
        %if &IDX EQ &N %then %let MAXK=&N; %else %let MAXK=%eval(&IDX-1);

        %if &MAXK=0 %then %do;
```

```
                        %PUT ERROR: Empty numeric-to-character mapping list.;
                        %return;
        %end;

        %let K=&MAXK;
        %DO %UNTIL(&K=0);
                %let xRULE=&&RULE&K;
                %let RULE_L_EXP=%sysfunc(scan(&xRULE, 1, &sep, Q));
                %let RULE_R_EXP=%sysfunc(scan(&xRULE, 2, &sep, Q));

                %if %length(&RULE_R_EXP)=0 %then %let RULE_R_EXP="&K";

                %if &K=&MAXK %then %do;
                        %Let COND_LST=(compare(trimn(&CHAREXP),trimn(&RULE_L_EXP),&MOD)=0);
                        %Let EXP_LST=&RULE_R_EXP;
                %end; %else %do;
                        %Let
COND_LST=(compare(trimn(&CHAREXP),trimn(&RULE_L_EXP),&MOD)=0)%str(,) &COND_LST;
                        %Let EXP_LST=&RULE_R_EXP%str(,) &EXP_LST;
                %end;

                %let K=%EVAL(&K-1);
        %End;

Choosec(1+whichn(1, &COND_LST),&OTHER,&EXP_LST)
%Mend;


/*$ COND2N
 Purpose: Mapping a condition to a numeric value.
*/
%Macro COND2N(
    RULE1, RULE2, RULE3, RULE4, RULE5, RULE6, RULE7, RULE8 /* Mapping rules */
  , OTHER=. /* Value to be returned if no rule is satisfied */
  , SEP=# /* DLM between the first and second part of a mapping rule */
);
          %LOCAL IDX N K MAXK XRULE ENDFLAG;
          %LOCAL RULE_L_EXP RULE_R_EXP COND_LST EXP_LST;

          %let N=8;
          %Let IDX=0;
          %DO %WHILE (&IDX LT &N);
                    %LET IDX=%EVAL(&IDX + 1);
                    %if %length(&&RULE&IDX) EQ 0 %then %goto leave;
          %End;
          %leave:
          %if &IDX EQ &N %then %let MAXK=&N; %else %let MAXK=%eval(&IDX-1);

          %if &MAXK=0 %then %do;
                              %PUT ERROR: Empty character-to-numeric mapping list.;
                              %return;
          %end;

          %let K=&MAXK;
          %DO %UNTIL(&K=0);
                    %let xRULE=&&RULE&K;
                    %let RULE_L_EXP=%sysfunc(scan(&xRULE, 1, &sep, Q));
                    %let RULE_R_EXP=%sysfunc(scan(&xRULE, 2, &sep, Q));

                    %if %length(&RULE_R_EXP)=0 %then %let RULE_R_EXP=&K;

                    %if &K=&MAXK %then %do;
                              %Let COND LST=&RULE L EXP;
                              %Let EXP_LST=&RULE_R_EXP;
                    %end; %else %do;
                              %Let COND_LST=&RULE_L_EXP%str(,) &COND_LST;
                              %Let EXP_LST=&RULE_R_EXP%str(,) &EXP_LST;
                    %end;

                    %let K=%EVAL(&K-1);
          %End;

Choosen(1+whichn(1, &COND_LST),&OTHER,&EXP_LST)
%Mend;
```

```
/*$ COND2C
 Purpose: Mapping a condition to a character value.
*/
%Macro COND2C(
    RULE1, RULE2, RULE3, RULE4, RULE5, RULE6, RULE7, RULE8 /* Mapping rules */
   ,OTHER=" " /* Value to be returned if no rule is satisfied */
   ,SEP=# /* DLM between the first and second part of a mapping rule */
);
        %LOCAL IDX N K MAXK XRULE;
        %LOCAL RULE_L_EXP RULE_R_EXP COND_LST EXP_LST;

        %let N=8;
        %Let IDX=0;
        %DO %WHILE (&IDX LT &N);
                %LET IDX=%EVAL(&IDX + 1);
                %if %length(&&RULE&IDX) EQ 0 %then %goto leave;
        %End;
        %leave:
        %if &IDX EQ &N %then %let MAXK=&N; %else %let MAXK=%eval(&IDX-1);

        %if &MAXK=0 %then %do;
                        %PUT ERROR: Empty character-to-numeric mapping list.;
                        %return;
        %end;

        %let K=&MAXK;
        %DO %UNTIL(&K=0);
                %let xRULE=&&RULE&K;
                %let RULE_L_EXP=%sysfunc(scan(&xRULE, 1, &sep, Q));
                %let RULE_R_EXP=%sysfunc(scan(&xRULE, 2, &sep, Q));

                %if %length(&RULE_R_EXP)=0 %then %let RULE_R_EXP=&K;

                %if &K=&MAXK %then %do;
                        %Let COND_LST=&RULE_L_EXP;
                        %Let EXP_LST=&RULE_R_EXP;
                %end; %else %do;
                        %Let COND_LST=&RULE_L_EXP%str(,) &COND_LST;
                        %Let EXP_LST=&RULE_R_EXP%str(,) &EXP_LST;
                %end;

                %let K=%EVAL(&K-1);
        %End;

Choosec(1+whichn(1, &COND_LST),&OTHER,&EXP_LST)
%Mend;
```