Building Better ADaM Datasets Faster With If-less Programming

Lei Zhang, Celgene Corporation, Berkeley Height, NJ

Abstract

One of major tasks in building ADaM datasets for a clinical study is to create derived variables based on a set of ADaM dataset specifications. Programmers frequently find this task time-consuming and prone to error. The main reason is that a clinical study typically has dozens of variables to derive for the safety and efficacy analyses, and those variables often have to be implemented with numerous if-else statements that require more time and effort to write, understand, and validate. In order to alleviate this tedious process, this paper introduces the if-less programming that promotes using fewer or no if-else statements in the programs. With this approach, programmers can not only write ADaM derivations shorter and quicker, but also make them easier to understand, modify, and reuse. What's more, this approach can be applied to the creation of ADaM datasets from both SDTM, and non-SDTM datasets.

Introduction

The Analysis Data Model (ADaM) is one of the most important CDISC standards in clinical trial submission to FDA and other healthcare authorities. It describes the principles and ways to create analysis datasets and associated metadata required by a submission. Moreover, in the *Study Data Technical Conformance Guide* published in March 2016[1], FDA recommends that the software programs used to create all ADaM datasets should be provided along with the tables and figures in order to help reviewers to better understand how the datasets, tables and figures were created. The guide aims to help reviewers to further understand the process by which the variables for the respective analyses were created and to confirm the analysis algorithms adopted. Although FDA doesn't require the programs submitted to be run directly under their given environment, the programs for ADaM datasets are treated as part of important documents for a submission, and they have to be well-written, easy to read and understand at least.

However, writing good programs to build ADaM datasets is not an easy task. Like ADaM specification development, most of time programmers find it a tedious process. This is because each ADaM dataset in a clinical study often has ten, or more derived variables typically coded with if-else (including select-when) statements. Excessive use of if-else statements usually makes the program hard to read, test and update. Besides, since the ADaM dataset specification is a living document, if the corresponding program is full of conditional statements, it often becomes difficult to extend, modify, and reuse with the specification evolution.

In order to understand this issue further, let's take a look at some examples. Below are the abridged definitions of 7 ADSL variables from a mockup one treatment open-label study.

ID	Variable Name	Variable Label	Source/Derivation	Notes

1	TRT01A	Actual Treatment for Period 01	ENRLFL=Y then 'Drug A'
			else Blank.
				ENRLFL is enrollment flag.

2	TRT01AN	Actual Treatment for Period 01 (N)	ENRLFL=Y then 1 = 'Drug A'
			else 2 = BLANK	

3	SAFFL	Safety Population Flag	if ENRLFL=Y and TRTSDA is not missing, then set to Y; otherwise, set to N.	TRTSDA is treatment start date, or first dose date
4	DSFL	Discontinuation Flag	if DSDT is not missing then set to Y Otherwise set to N	
5	INVNAM	Investigator Name	Derive from SITEID - 002 = 'PARIS' 010 = 'NICE' 011 = 'NEW YORK" 013 = 'WASHINGTON' 021 = 'BEIJING'	SITEID is the site ID
6	COUNTRY	Country	Convert SITEID to numeric - if 1<= and < =10 then set to 'FRA'; if 10< and <=20 then set to 'USA'; if >20 then set to 'CHN'; otherwise set to BLANK.	
7	LSTCTDT	Last contact date	First non-missing date among the death date(DTHDT), discontinuation date(DISCDT), last dose date(LSTDDT) + 14 days, and last visit date(LSTVDT).	

As you may notice, each derived variable in the ADSL specific ation above is a function of one, or more variables under certain conditions.

The above 7 variables can be implemented within the following traditional if-else or select-when statements:

```
DATA ADSL;
       ...

       /* TRT01A */
    if ENRLFL='Y' then TRT01A='Drug A';
       else TRT01A=' ';

       /* TRT01AN */
    if ENRLFL='Y' then TRT01AN=1;
       else TRT01AN=2;

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

       /* DSFL */
       If not missing(DSDT) then DSFL='Y';
       else DSFL='N';

       /* INVNAM */
       if SITEID="002" then INVNAM='PARIS';
       else if SITEID="010" then INVNAM='NICE';
       else if SITEID="011" then INVNAM='NEW YORK';
       else if SITEID="013" then INVNAM='WASHINGTON';
       else if SITEID="021" then INVNAM='BEIJING';
       else INVNAM=' ';

       /* COUNTRY */
       if 1 <= input(SITEID,8.) <=10 then COUNTRY='FRA';
       else if 10 < input(SITEID,8.) <=20 then COUNTRY='USA';
       else if input(SITEID,8.) >20 then COUNTRY='CHN';
    else COUNTRY=' ';

       /* 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;
```

As you can see, lots of if-else statements are used in order to create the derived variables, and the program ends up with many branched verbose code. This approach has following issues:

- It makes the program hard to reason about its behavior because of numerous code branches introduced by if-else statements.
- It makes the program difficult to test or debug. You may have to test each of branches in the code.
- It makes the program hard to update or extend because there are chances that the difference in behavior between branches can be quite big.
- It may cause unintended side effects. This is because an if-else statement has to make change on the outside variables, which may cause unwanted or inconsistent changes on them.

Next section, I'll introduce the if-less programming that enables you to derive ADaM variables using short and concise expressions instead of the cumbersome and error-prone if-else (including select-when) statements.

If-less Programming

Like many other procedural programming language, the if-else statement is fundamental to SAS®, and we certainly don't want to get rid of it in all the programs, but at the same time we have to acknowledge that the if-else statement usually causes more time and effort to write, understand and validate[2][3][4]. There are at least three benefits when if-else statements can be avoided in a program:

- The program without if-else statements is easier to read
- The program without if-else statements is easier to test
- The program without if-else statements is easier to update or extend.

So the question becomes how we can use less if-else statements in our programs. Fortunately, SAS has already provided a couple of alternative ways to help us to do that.

One of approaches is to replace a list of if-else statements with built-in function PUT() and INPUT() under the help of informats or formats. For example, variable INVNAM and COUNTRY in the above example can be derived with following alternative code:

```
PROC FORMAT;
invalue $sitefmt
"002"='PARIS'
"010"='NICE'
"011"='NEW YORK'
"013"='WASHINGTON'
"021"='BEIJING'
Other=' '
;
Value cntyfmt
1 - 10 ='FRA'
10< - 20 ='USA'
20< - High='CHN'
OTHER=' '
;
Run;

DATA ADSL;
        ...

        /* INVNAM */
        INVNAM=input(SITEID, $SITEFMT.);

        /* COUNTRY */
        COUNTRY=PUT(input(SITEID,8.),CNTYFMT.);

        ...
RUN;
```

This is quite a popular method to replace if-else statements, especially when you have a large number of values that have to be directly mapped or transformed, but this method suffers from following drawbacks:

- It only works with limited cases. For example, variable SAFFL can't be derived with this method.
- The format or informat used has to be defined elsewhere first, either in a separate code segment, or even in another program file, which causes kinds of code fragments, and adds extra layer to the code understanding.
- It may be overkilled sometimes. For example, if variable TRT01A, and TRT01AN were coded with this method, the programmer would end up with writing longer pieces of code due to trivial formats.

The second method is to use the function IFN() and IFC() to replace if-else statements, and CHOOSEN() and CHOOSEC() to replace select-when statements. Those four functions have been introduced since SAS 9, for more information about them, please see [5]. For example, TRT01A and INVNAM can be implemented with following code:

```
DATA ADSL;
  ...
      /* TRT01AN */
  TRT01AN=IFN(ENRLFL='Y', 1, 2);

      /* TRT01A */
  TRT01A=IFC(TRT01AN=1, 'Drug A',' ');

      /* INVNAM */
  INVNAM=CHOOSEC(1+(SITEID="002")*1 + (SITEID="010")*2
         + (SITEID="011")*3 + (SITEID="013")*4
     + (SITEID="021")*5,
     ' ', 'PARIS', 'NICE', 'NEW YORK', 'WASHINGTON', 'BEIJING'
        );

  ...
RUN;
```

Although using those functions have some advantages over using if-else or select-when statements[6], there are some problems with them:

- IFN() and IFC() work well as a replacement of simple if-else statements, but not with nested if-else statements. For example, if using IFC(), COUNTRY variable will have to be coded with nested IFC() function calls as follows, which makes code much less readable and modifiable.

```
DATA ADSL;
       ...;
       /* COUNTRY */
  COUNTRY=IFC(1<= input(SITEID,8.) <=10, 'FRA'
   ,IFC(10 < input(SITEID,8.) <=20, 'USA'
   ,IFC(input(SITEID,8.) >20,'CHN',' ')));

       ...;
RUN;
```

- CHOOSEC() and CHOOSEN() sometimes can't replace the select-when statements straightforward. They often have to be coded with a little trick like the one in the alternative implementation of variable INVNAM. The code seems not easy to understand without a second thought.

Can we have a better way to replace if-else or select-when statements in both simple and complex ADaM variable derivations? In the next section, I'll describe the third method, which is accomplished by the mapping macros that are essentially the generalization of the function IFN(), IFC(), CHOOSEN(), and CHOOSEC().

Mapping Macros for If-less Programming

Since the function IFN(), IFC(), CHOOSEN(), and CHOOSEC() can perform if-else, or select-when logic and map a value into another in a succinct way in simple situations, it is very meaningful to make this method generic so that nested or complex logics can be coded in a succinct way as well. This can be accomplished with following six mapping macros that I have developed.

Inline Macro Function	Purpose
%N2WRD()	Maps a numeric value to a character value.
%N2N()	Maps a numeric value to another numeric value.
%COND2N()	Maps a condition or a logical expression to a numeric value.
%WRD2N()	Maps a character value to a numeric value.
%WRD2WRD()	Maps a character value to another character value.
%COND2WRD()	Maps a condition or a logical expression to a character value.

This set of macros are written as inline macro functions used primarily by data steps. They accept varying number of parameters and are implemented as the composites of IFN(), IFC(), CHOOSEN(), and CHOOSEC(), therefore they can be used like any built-in data step function. Each of the mapping macros enables you to apply a list of inline mapping rules against all values in one, or more variables and returns a new variable that contains all mapped values, thus offering programmers an elegant way to create a variable based on the result of multiple comparisons. As you will see later, the six mapping macros provide a clear, expressive, and efficient way to encode derived variables without using any if-else statements.

%N2WRD

%N2WRD() maps a numeric value to a character one. As an extension of function CHOOSEC(), it accepts varying number of arguments and has following syntax.

Syntax

```
%N2WRD(
 nvalue /* Numeric expression to be searched in the mapping list*/
 ,<n-identifier-1#>c-value-1 /* Mapping item 1 */
 ,<n-identifier-1#>c-value-2 /* Mapping item 2 */
 , , ,
 ,<n-identifier-k#>c-value-k /* Mapping item k */
 ,OTHER=' ' /* Character value returned if no match */
 ,SEP=#     /* Separator between identifier and value part of a mapping item */
)
```

The first positional parameter of this mapping macro (that is, `nvalue`) is a required numeric value to be searched in the inline mapping list which is consisted of the rest of positional arguments. Each mapping item in the mapping list contains two parts: the first part is the numeric identifier of the item, and the second part is the character value of the item to be returned when the identifier equals `nvalue`. The character # is used as a separator between the two parts by default. You can change it to a different separator with key parameter `SEP=`. If the identifier of a mapping item is not provided, then the sequence number of the item (or the item number) will be used as the default identifier. %N2WRD() checks the inline mapping list from left to right until it finds the first item with its identifier equal to `nvalue`. If none of identifiers equals `nvalue`, it returns the value provided in the key parameter OTHER=.

The following example demonstrates the uses of %N2WRD():

```
DATA N2WRD;
 Length phase1 phase2 $20;
 input n@@;
 phase1=%N2WRD(n,"Yes","No", Other="Missing");
 phase2=%N2WRD(n,2:"Yes",-1:"No", Other="Missing", sep=:);
 put "Output: " n= phase1= phase2=;
datalines;
1 2 -1 5.1 .
;
RUN;
```

The results in the log:

```
Output: n=1 phase1=Yes phase2=Missing
Output: n=2 phase1=No phase2=Yes
Output: n=-1 phase1=Missing phase2=No
Output: n=5.1 phase1=Missing phase2=Missing
Output: n= phase1=Missing phase2=Missing
```

%N2N

%N2N() maps a numeric value to another numeric value and is an extension of function CHOOSECN(). It is similar to %N2WRD() in both syntax and usage except that it returns a numeric value instead of character one. Below is its syntax:

Syntax

```
%N2N(
 nvalue /* Numeric expression to be searched in the mapping list*/
 ,<n-identifier-1#>n-value-1 /* Mapping item 1 */
 ,<n-identifier-1#>n-value-2 /* Mapping item 2 */
 , , ,
 ,<n-identifier-k#>n-value-k /* Mapping item k */
 ,OTHER=.  /* Numeric value returned if no match */
 ,SEP=#    /* Separator between identifier and value part of a mapping item */
)
```

The following example demonstrates its uses:

```
DATA N2N;
       input n@@;
       m1=%N2N(n,20,35,50,100,other=-99);
       m2=%N2N(n,5:20,4:35,3:50,2:100,other=-88, sep=:);
       put "Output: " n= m1= m2=;
datalines;
1 2 3 4 5 .
;
RUN;
```

The results in the log:

```
Output: n=1 m1=20 m2=-88
Output: n=2 m1=35 m2=100
Output: n=3 m1=50 m2=50
Output: n=4 m1=100 m2=35
Output: n=5 m1=-99 m2=20
Output: n= m1=-99 m2=-88
```

%COND2N

%COND2N() maps a condition or logical expression to a numeric value. Different from %N2WRD() and %N2N(), all of its arguments are treated as a logical mapping list. Each item in the logical mapping list contains two part parts (separated by # by default). The first part is the identifier that is a logical expression, or condition, and the second part is the numeric value to be returned when the associated logical expression, or condition is evaluated to TRUE. Like %N2WRD() or %N2N(), it also has two key parameters OTHER= and SEP=. %COND2N() has following syntax:

Syntax

```
%COND2N(
  logical-identifier-1<#n-value-1> /* Mapping item 1 */
 ,logical-identifier-2<#n-value-2> /* Mapping item 2 */
 , , ,
 ,logical-identifier-k<#n-value-k> /* Mapping item k */
 ,OTHER=. /* Value-returned when all identifiers are evaluated to FALSE */
 ,SEP=#   /* Separator between identifier and value part of a mapping item */
)
```

%COND2N() checks the logical mapping list, from left to right, until it finds the first item with its logical identifier being evaluated to TRUE, and then returns the value part. If none of identifiers are evaluated to TRUE, then the numeric value in key parameter OTHER= is returned. If a logical mapping item is provided without the value part, %COND2N() takes the item number as the value to be returned. Below is an example that demonstrates its uses.

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

The results in the log:

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

%WRD2N

%WRD2N() is the reverse of %N2WRD(), but a little more complicated. It maps a character value to a numeric one with following syntax:

Syntax

```
%WRD2N (
 cvalue   /* Character value to be searched in the mapping list */
 ,c-identifier-1<#n-value-1> /* Mapping Item 1 */
 ,c-identifier-2<#n-value-2> /* Mapping item 2 */
 , , ,
 ,c-identifier-k<#n-value-k> /* Mapping item k */
 ,OTHER=.  /* Numeric value-returned if no match */
 ,SEP=#    /* Separator between identifier and value part of a mapping item */
 ,MOD=’ ’  /* Comparison modifiers similar to those defined in compare()    */
 )
```

The first positional parameter (that is, `cvalue`) is a required character value to be searched in the inline mapping list that is consisted of the rest of positional arguments in the macro. Each item in the mapping list contains two parts (separated by # by default) : the first part is the character identifier of the item, and the second part is the numeric value of the item to be returned when its associated identifier matches with `cvalue`. Like %N2WRD(), if the value part of a mapping item is not provided, then the item number is used as the value to be returned. %WRD2N() checks the inline mapping list from left to right and returns the value of the first identified mapping item.

The following example demonstrates its uses:

```
DATA WRD2N;
 Length s $16;
 input s $ 1-16;
 n1=%WRD2N(s,"Yes","No", Other=-9, MOD=":");
 n2=%WRD2N(s,"Y":2,"N":1, Other=-8, MOD=":i", SEP=:);
 put "Output: " s= n1= n2=;
datalines;
Yes
No
YES
NO
Y
N

;
RUN;
```

The results in the log:

```
Output: s=Yes n1=1 n2=2
Output: s=No n1=2 n2=1
Output: s=YES n1=-9 n2=2
Output: s=NO n1=-9 n2=1
Output: s=Y n1=1 n2=2
Output: s=N n1=2 n2=1
Output: s= n1=-9 n2=-8
```

Please note if none of item identifiers matches with `cvalue`, the macro returns the numeric value provided in key parameter OTHER=, like other mapping macros. What is more, %WRD2N() allows a programmer to compare the character identifier of a mapping item with `cvalue` using the modifiers provided by key parameter `MOD=`, which are similar to those defined in the function COMPARE(). For example, in the above example, MOD=’:’ is used to instruct

%WRD2N() to truncate the longer of two character values to be compared to the length of the shorter one. You can also use other modifiers in a comparison if needed, such as ‘I’, ‘N’, ,‘L’, or their combinations.

%WRD2WRD

%WRD2WRD() maps a character value to another character one. It has following syntax:

Syntax

```
%WRD2WRD(
 cvalue  /* Character value to be searched in the mapping list*/
 ,c-identifier-1<#c-value-1> /* Mapping Item 1 */
 ,c-identifier-2<#c-value-2> /* Mapping Item 2 */
 , , ,
 ,c-identifier-k<#c-value-k> /* Mapping Item k */
 ,OTHER=’ ’ /* Character value to be returned when no match */
 ,SEP=#     /* Separator between identifier and value part of a mapping item */
 ,MOD=’ ’   /* Comparison modifiers similar to those defined in compare()    */
)
```

Like %WRD2N(), %WRD2WRD() takes a varying number of arguments. The first positional argument (that is `cvalue`) of the mapping macro provides the character value to be searched in the inline mapping list that is consisted of the rest of positional arguments. Each item in the mapping list has two parts: the first part is a character identifier to be compared with `cvalue`, and the second part is the character value to be returned when its identifier matches with `cvalue`. The character # is used by default as a separator between the two parts, and can be replaced with other character using key parameter `SEP=`. Similar to %WRD2N(), %WRD2WRD() allows you to control how to compare an identifier with `cvalue` using key parameter `MOD=`. In case the value part of a mapping item is not provided, then the item number (in character) is used as the value part.

Below is a code example that demonstrates its uses.

```
DATA WRD2WRD;
 Length code1 code2 $20 s $16;
 input s $ 1-16;
 code1=%WRD2WRD(s,"Yes","No", Other="UNK", mod="i");
 code2=%WRD2WRD(s,"Yes"#"Y","No"#"N", Other="UNK", mod=":");
 put "Output: " s= code1= code2=;
datalines;
Yes
No
Yes, Sir
Y
N

;
RUN;
```

The results in the log:

```
Output: s=Yes code1=1 code2=Y
Output: s=No code1=2 code2=N
Output: s=Yes, Sir code1=UNK code2=Y
Output: s=Y code1=UNK code2=Y
Output: s=N code1=UNK code2=N
Output: s= code1=UNK code2=UNK
```

%COND2WRD

%COND2WRD() maps a logical or conditional expression to a character value. It takes a varying number of arguments that form a logical mapping list. Each item in the logical mapping list has two part parts (separated by # by default). The first part is the identifier consisting of a logical or conditional expression, and the second part is the character value to be returned when the logical or conditional expression is evaluated to TRUE. It has following syntax:

Syntax

```
%COND2WRD(
  logical-identifier-1<#c-value-1> /* Mapping Item 1 */
 ,logical-identifier-2<#c-value-2  /* Mapping Item 2 */
 , , ,
 ,logical-identifier-k<#c-value-k> /* Mapping Item k */
 ,OTHER=' ' /* Character value to be returned when all are evaluated to FALSE */
 ,SEP=#     /* Separator between the logical-identifier-k and c-value-k       */
)
```

%COND2WRD() checks the inline mapping list, from left to right, until it finds the first mapping item whose associated logical expression is evaluated to TRUE, and then return the value part. If none of logical identifiers are evaluated to TRUE, it then returns the value provided in the key parameter OTHER=. In addition, %COND2WRD() allows you to provide a logical expression without the value part in an item. In that case, the item number (in character) is returned if the logical expression is evaluated to TRUE. Like other mapping macros, %COND2WRD() use # as default separator between the logical expression and value part. When needed, it allows you to choose a different character as a separator with key parameter `SEP=`. Below is a code example that demonstrates its uses.

```
DATA COND2WRD;
       Length code $20;
       input x@@;
 code=%COND2WRD(0<=x< 5#"LT 5", x<10#"5 GE, LT 10"
     ,x>=10#"GT 10", other="Missing");
 put "Output: " x= code=;
datalines;
1.5 5 10 15.5 -10 .
;
RUN;
```

The results in the log:

```
Output: x=1.5 code=LT 5
Output: x=5 code=5 GE, LT 10
Output: x=10 code=GT 10
Output: x=15.5 code=GT 10
Output: x=-10 code=5 GE, LT 10
Output: x= code=5 GE, LT 10
```

Rewriting the Example with Mapping Macros

With the above 6 mapping macros, let's see how the ADSL variables previously defined can be re-implemented:

```
DATA ADSL;
       ...

       /* TRT01AN */
       TRT01AN=%WRD2N(ENRLFL,'Y',OTHER=2);
       /* TRT01A */
       TRT01A=%N2WRD(TRT01AN,1#'Drug A',OTHER=' ');
       /* SAFFL */
       SAFFL=%COND2WRD((ENRLFL='Y' and not missing(TRTSDA))#'Y'
         ,OTHER='N');
       /* DSFL */
      DSFL=%COND2WRD(NMISS(DISCDT)=0#'Y', OTHER='N');
       /* INVNAM */
       INVNAM=%WRD2WRD(SITEID
         ,"002"#'PARIS',"010"#'NICE'
         ,"011"#'NEW YORK',"013"#'WASHINGTON'
         ,"021"#'BEIJING', OTHER=' ');
       /* COUNTRY */
       COUNTRY=%COND2WRD(1<= input(SITEID,8.) <=10#'FRA'
            ,10 < input(SITEID,8.) <=20#'USA'
            ,input(SITEID,8.) >20#'CHN'
              ,OTHER=' ');
```

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

Clearly, using the mapping macros allows a programmer to write much shorter code to derive a ADaM variable. It has following benefits:

- The code is clear, concise and expressive, having all the required information collected in one place.
- It is easy to infer how the code works with all the information.
- Unnecessary code blocks and branching are eliminated, or reduced to their most refined forms.
- The code is easy to test and validate because the mapping macros are stateless and have no side effects.

Other If-less Programming Techniques

The mapping macros introduced above help ADaM programmers avoid roughly 70% of if-else and select-when statements when coding ADaM derived variables. However, there are more coding techniques that enable you to reduce the use of if-else or select-when statements in a program. One is to use other built-in SAS functions, such as COALESCE, PRX, MIN and MAX functions. For example, the function COALESCEN() can be used to select the first non-missing numeric value among several ones, you therefore can derive the variable LSTCTDT with following simple code:

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

Using logical expressions as numeric values is another technique. For example, if you want to derive the common variable ASTDY defined below,

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

you can write a piece of code by combination with a logical expression like this

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

Another technique worthy to mention is to create user-defined functions with PROC FCMP first to replace multiple if-else statements and other code pieces, and then call them from data steps. This technique is especially useful when you have a common algorithm to be shared by more than one ADaM variables. Here is an example. Assuming that we want to create an analysis start date variable ASTDT and the correspond flag ASTDTF for ADAE dataset with following imputation method:

Derivation ID	Variable Name	Variable Label	Source/Derivation	Notes
1	ASTDT	Analysis Start Date	Derive from AE.AESTDTF - impute partial dates using the following rules: <u>Missing Year</u>: No imputation, return missing value <u>Missing day and month</u>: impute the day and month with 'June 15', return the numeric date. <u>Missing day only</u>: impute the day with '15', return the numeric date	In ISO8601
2	ASTDTF	Analysis End Date Imputation Flag	Derived from AE.AESTDTF. If it is completely missing or no missing then ASTDTF=' '; otherwise, if start date has day and month missing then ASTDTF='M' else if start date has day missing then ASTDTF='D'.	In ISO8601

The imputation method involves a few of if-else statements logically, and it can be addressed with %COND2WRD() as follows.

```
DATA ADAE;
```

```
      . . .
   /* ASTDTF */
      ASTDTF=%COND2WRD(
     lengthn(AESTDTF)=6#'D'
    ,lengthn(AESTDTF)=4#'M'
        ,OTHER=’ ’
    );

      /* ASTDT */
      ASTDT=input(
     %WRD2WRD(ASTDTF,
        ,”D”#cats(AESTDTF,"15")
        ,”M”#cats(AESTDTF,"0701")
        ,OTHER=AESTDTF)
     ,B8601DA.) ;
 . . .
Run;
```

However, since the imputation flag variable ASTDTF are also needed in other ADaM datasets, such as ADLB, ADCM, it is better to implement it as a user-defined function to further simplify ADaM programming across multiple datasets. Below is the sample code implemented with a user-defined function in PROC FCMP:

```
PROC FCMP Outlib=WORK.usrfunc.string;
 /* Get the ADaM date imputation flag */
 Function ImpFlag(
  CDate $ /* Character expression in ISO8601 */
 )$1;
 Return(%COND2WRD(
     lengthn(CDATE)=6#'D'
    ,lengthn(CDATE)=4#'M'
    ,OTHER=' '
   )
 );
 Endsub;
RUN;
options cmplib=(WORK.usrfunc);

DATA ADAE;
  Length ASTDTF $1 AESTDTF $8;
  . . .
  /* ASTDTF */
  ASTDTF=ImpFlag(AESTDTF);
  . . .
RUN;
```

A user defined function encapsulates multiple if-else and/or select-when statements, and can handle one or more derived variables at one time, thus making the code much shorter and simpler across multiple ADaM dataset programs.

Conclusion

This paper describes the mapping macros along with several other if-less programming techniques that enable programmers to replace if-else or select-when statements with clear, concise and expressive code pieces in a program. As more and more clinical studies require ADaM datasets for the regulatory submissions, the coding techniques in this paper will enable programmers to write better ADaM programs with a more efficient way in a short time. It is expected that more programmers will adopt them in the ADaM dataset creation and other programming adventures to gain maximum benefits.

Reference

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*

Acknowledgements

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.

Contact Information

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

Lei Zhang
Celgene Corporation.
400 Connell Dr.
Berkeley Heights NJ 07922
Phone: (908) 673-9000

Appendix 1: Sample Mapping Macros For Illustrative Purposes

```
/* NOTE: the sample mapping macros only accept a mapping list with up to 8 items */

/*$ N2WRD
 Purpose: Mapping numeric value to character value.
*/
%Macro N2WRD(
   nValue /* Numeric value to be mapped */
  ,ITEM1, ITEM2, ITEM3, ITEM4  /* Mapping items (1 - 8)  */
  ,ITEM5, ITEM6, ITEM7, ITEM8
  ,OTHER=' ' /* Value to be returned if no items are identified */
  ,SEP=#   /* Separator between identifier and value part of a mapping item */
);

%LOCAL IDX N K XITEM CMPFN;
%LOCAL ENDFLAG;

%let N=8;
%LET IDX= 1;
%let endflag=0;
%let xitem=&&ITEM&IDX;
%DO %WHILE ((%length(&xitem) ne 0) and (&endflag=0));
  %LET IDX=%EVAL(&IDX + 1);
  %if &IDX=%eval(&N+1) %then %let endflag=1;
  %else %let xitem=&&ITEM&IDX;
%End;
```

```
%let K=%eval(&idx-1);
%if &K=0 %then %do;
  %PUT ERROR: Empty conditional mapping list.;
  %return;
%end;

%let xitem=&&ITEM&K;
%let n_exp=%sysfunc(scan(&xitem,1,&sep,Q));
%let n_sel=%sysfunc(scan(&xitem,2,&sep,Q));
%if %length(&n_sel)=0 %then %do;
       %let n_sel=&n_exp;
       %let n_exp=&K;
%end;
%let CMPFN=IFC((&nValue)=(&n_exp),&n_sel,&OTHER);

%DO idx=%eval(&K-1) %to 1 %by -1;
  %let xitem=&&ITEM&IDX;
  %let n_exp=%sysfunc(scan(&xitem,1,&sep,Q));
  %let n_sel=%sysfunc(scan(&xitem,2,&sep,Q));
       %if %length(&n_sel)=0 %then %do;
              %let n_sel=&n_exp;
              %let n_exp=&idx;
       %end;
  %let CMPFN=IFC((&nValue)=(&n_exp),&n_sel,&CMPFN);
%END;
&CMPFN
%Mend;


/*$ N2N
 Purpose: Mapping numeric value to numeric value.
*/
%Macro N2N(
   nValue /* Numeric value to be mapped */
  ,ITEM1, ITEM2, ITEM3, ITEM4  /* Mapping items (1 - 8)  */
  ,ITEM5, ITEM6, ITEM7, ITEM8
  ,OTHER=. /* Value to be returned if no items are identified */
  ,SEP=#  /* Separator between identifier and value part of a mapping item */
);

%LOCAL IDX N K XITEM CMPFN;
%LOCAL ENDFLAG;

%let N=8;
%LET IDX= 1;
%let endflag=0;
%let xitem=&&ITEM&IDX;
%DO %WHILE ((%length(&xitem) ne 0) and (&endflag=0));
  %LET IDX=%EVAL(&IDX + 1);
  %if &IDX=%eval(&N+1) %then %let endflag=1;
  %else %let xitem=&&ITEM&IDX;
%End;

%let K=%eval(&idx-1);
%if &K=0 %then %do;
  %PUT ERROR: Empty conditional mapping list.;
  %return;
%end;

%let xitem=&&ITEM&K;
%let n_exp=%sysfunc(scan(&xitem,1,&sep,Q));
%let n_sel=%sysfunc(scan(&xitem,2,&sep,Q));
```

```
%if %length(&n_sel)=0 %then %do;
       %let n_sel=&n_exp;
       %let n_exp=&K;
%end;
%let CMPFN=IFN((&nValue)=(&n_exp),&n_sel,&OTHER);

%DO idx=%eval(&K-1) %to 1 %by -1;
  %let xitem=&&ITEM&IDX;
  %let n_exp=%sysfunc(scan(&xitem,1,&sep,Q));
  %let n_sel=%sysfunc(scan(&xitem,2,&sep,Q));
       %if %length(&n_sel)=0 %then %do;
              %let n_sel=&n_exp;
              %let n_exp=&idx;
       %end;
  %let CMPFN=IFN((&nValue)=(&n_exp),&n_sel,&CMPFN);
%END;
&CMPFN
%Mend;

/*$ COND2N
 Purpose: Mapping a logical expression, or condition to a numeric value.
*/

%Macro COND2N(
   ITEM1, ITEM2, ITEM3, ITEM4 /* Mapping items (1 - 8) */
  ,ITEM5, ITEM6, ITEM7, ITEM8
  ,OTHER=. /* Value to be returned if no items are identified */
  ,SEP=#  /* Separator between identifier and value part of a mapping item */
);

%LOCAL IDX N K XITEM ENDFLAG;
%LOCAL CMPFN L_EXP N_EXP;

%let N=8;
%LET IDX= 1;
%let endflag=0;
%let xitem=&&ITEM&IDX;
%DO %WHILE ((%length(&xitem) ne 0) and (&endflag=0));
  %LET IDX=%EVAL(&IDX + 1);
  %if &IDX=%eval(&N+1) %then %let endflag=1;
  %else %let xitem=&&ITEM&IDX;
%End;

%let K=%eval(&idx-1);
%if &K=0 %then %do;
  %PUT ERROR: Empty conditional mapping list.;
  %return;
%end;

%let xitem=&&ITEM&K;
%let L_EXP=%sysfunc(scan(&xitem,1,&sep,Q));
%let N_EXP=%sysfunc(scan(&xitem,2,&sep,Q));
%if %length(&N_EXP)=0 %then %let N_EXP=&K;
%LET CMPFN=IFN(&L_EXP,&N_EXP,&OTHER);

%DO idx=%eval(&K-1) %to 1 %by -1;
  %let xitem=&&ITEM&IDX;
  %let L_EXP=%sysfunc(scan(&xitem,1,&sep,Q));
  %let N_EXP=%sysfunc(scan(&xitem,2,&sep,Q));
  %if %length(&N_EXP)=0 %then %let N_EXP=&IDX;
  %LET CMPFN=IFN(&L_EXP,&N_EXP,&CMPFN);
%END;
&CMPFN
```

```
%Mend;

/*$ WRD2N
 Purpose: Mapping a character value to a numeric value.
*/
%Macro WRD2N(
  cValue  /* Character value to be mapped */
  ,ITEM1, ITEM2, ITEM3, ITEM4 /* Mapping items (1 - 8) */
  ,ITEM5, ITEM6, ITEM7, ITEM8
  ,OTHER=' ' /* Value to be returned if no items are identified */
  ,SEP=# /* Separator between identifier and value part of a mapping item */
  ,MOD=' ' /* comparison modifiers that is same as those */
       /* used by built-in function COMPARE().    */
);

%LOCAL XITEM IDX LOC LOC1;
%LOCAL EXPLST1 EXPLST2;
%LOCAL PART1 PART2;
%LOCAL C_EXPS N_SELS;

%let C_EXPS=;
%let N_SELS=&OTHER;

%LET IDX=1;
%LET XITEM=&&ITEM&IDX;
%DO %WHILE(%LENGTH(&XITEM));

  %let part1=%sysfunc(scan(&xitem,1,&SEP,Q));
  %let part2=%sysfunc(scan(&xitem,2,&SEP,Q));

  %if &idx = 1 %then %DO;
              %let C_EXPS=%str(compare(trimn(&cValue),trimn(&part1),&MOD)=0);
  %end; %else %do;
              %let
C_EXPS=&C_EXPS%str(,)%str(compare(trimn(&cValue),trimn(&part1),&MOD)=0);
       %end;

  %if %length(&Part2)=0 %then %let Part2=&IDX;
  %let N_SELS=&N_SELS%str(,)&part2;

  %LET IDX=%EVAL(&IDX+1);
  %LET XITEM=&&ITEM&IDX;
%END;

%if %length(&C_EXPS) %then %do;
Choosen(1+%COND2N(%unquote(&C_EXPS),OTHER=0),%unquote(&N_SELS))
%end; %else %do;
&OTHER
%end;
%Mend;

/*$ WRD2WRD
 Purpose: Mapping a character value to a character value.
*/
%Macro WRD2WRD(
  cValue /* Character value to be mapped */
  ,ITEM1, ITEM2, ITEM3, ITEM4 /* Mapping items (1 - 8) */
  ,ITEM5, ITEM6, ITEM7, ITEM8
  ,OTHER=' ' /* Value to be returned if no items are identified */
  ,SEP=# /* Separator between identifier and value part of a mapping item */
  ,MOD=' ' /* comparison modifiers that is same as those */
       /* used by built-in function COMPARE().    */
);
```

```
%LOCAL XITEM IDX LOC LOC1;
%LOCAL EXPLST1 EXPLST2;
%LOCAL PART1 PART2;
%LOCAL C_EXPS C_SELS;

%let C_EXPS=;
%let C_SELS=&OTHER;

%LET IDX=1;
%LET XITEM=&&ITEM&IDX;
%DO %WHILE(%LENGTH(&XITEM));

  %let part1=%sysfunc(scan(&xitem,1,&SEP,Q));
  %let part2=%sysfunc(scan(&xitem,2,&SEP,Q));

  %if &idx = 1 %then %DO;
              %let C_EXPS=%str(compare(trimn(&cValue),trimn(&part1),&MOD)=0);
  %end; %else %do;
              %let
C_EXPS=&C_EXPS%str(,)%str(compare(trimn(&cValue),trimn(&part1),&MOD)=0);
       %end;

  %if %length(&Part2)=0 %then %let Part2="&IDX";
  %let C_SELS=&C_SELS%str(,)&part2;

  %LET IDX=%EVAL(&IDX+1);
  %LET XITEM=&&ITEM&IDX;
%END;

%if %length(&C_EXPS) %then %do;
Choosec(1+%COND2N(%unquote(&C_EXPS),OTHER=0),%unquote(&C_SELS))
%end; %else %do;
&OTHER
%end;
%Mend;

/*$ COND2WRD
 Purpose: Mapping a logical expression or condition to a character value.
*/
%Macro COND2WRD(
   ITEM1, ITEM2, ITEM3, ITEM4 /* Mapping items (1 - 8) */
  ,ITEM5, ITEM6, ITEM7, ITEM8
  ,OTHER=' ' /* Value to be returned if no items are identified */
  ,SEP=# /* Separator between identifier and value part of a mapping item */
);

%LOCAL XITEM IDX LOC LOC1;
%LOCAL COND_EXPS C_SELS;
%local explst1 explst2;
%local part1 part2;

%let COND_EXPS=;
%let C_SELS=&OTHER;

%LET IDX=1;
%LET XITEM=&&ITEM&IDX;
%DO %WHILE(%LENGTH(&XITEM));

  %let part1=%sysfunc(scan(&xitem,1,&SEP,Q));
  %let part2=%sysfunc(scan(&xitem,2,&SEP,Q));

  %if &idx = 1 %then %let COND_EXPS=&part1;
  %else %let COND_EXPS=&COND_EXPS%str(,)&part1;

  %if %length(&C_SELS)=0 %then %let C_SELS="&idx";
  %let C_SELS=&C_SELS%str(,)&part2;

  %LET IDX=%EVAL(&IDX+1);
  %LET XITEM=&&ITEM&IDX;
%END;

%if %length(&COND_EXPS) %then %do;
Choosec(1+%cond2n(%unquote(&COND_EXPS),OTHER=0),%unquote(&C_SELS))
%end; %else %do;
&OTHER
%end;
%Mend;
```