Conditional Execution of the Drop Statement as well as Other Global and Declarative Statements

Brian O. Mullin, Idaho Division of Health, Center for Vital Statistics and Health Policy, Boise, Idaho

Abstract

The SAS® language contains three types of statements, two of which, declarative and global, cannot be executed conditionally. Three methods of conditionally implementing declarative and global statements are presented. Each of the methods focus on the Drop statement.

Introduction

Contained within the SAS language there are three types of statements: executable, declarative, and global. Executable statements are commands that can be implemented conditionally. Probably the two most commonly used executable statements consist of the assignment and the IF-THEN/ELSE statements. The assignment statement takes a form similar to

\[ x = 2; \]

While the IF-THEN/ELSE appears as follows.

\[ \text{if } x = 1 \text{ then } y = 2; \text{ else } y = .; \]

A complete list of executable statements appears on page 45 of the SAS Language: Reference, Version 6, First Edition. All executable statements can be issued based on some criteria established in the program, similar to the above IF-THEN/ELSE. However, there are many statements in the SAS language which cannot be executed conditionally; they are known as global and declarative statements (SAS Language: Reference).

The DROP and X (host command) statements are declarative and global, respectively. On the PC platform, the following statement is intended to produce a list of files in the appropriate directory when X is 2.

\[ \text{if } x = 2 \text{ then } x \ ' \text{dir c:\'} ; \]

Although syntactically correct (SAS will not explode, flooding the log and programer with demeaning error messages), the above statement does not produce the desired results. The dir command is passed to the operating system regardless of the value of X. In the CMS® environment, the X statement cannot follow a THEN statement. The following code in SAS 6.06 for CMS, intended to produce a list of all files, simply produces a syntactical error.

\[ \text{if } x = 2 \text{ then } x \ ' \text{filelist} ; \]

Perhaps an error message is better. At least, the programer is alerted to the inadequacies of the code, rather than living in blissful ignorance. Unfortunately, in SAS what you don't know can and will hurt you. However, the ability to conditionally pass a command to the host system is a desirable feature. Similarly, there are many other global and declarative statements which are useful when implemented based on some criteria. The LENGTH, DROP, and RENAME statements are all declarative. A complete list of global and declarative statements appears on pages 45 and 49, respectively, of the SAS Language: Reference.

The most obvious difference between the two groups is where the statements can be placed in a program. Global statements can be placed anywhere in a SAS program, while declarative statements must be placed within a DATA step. There are many other circumstances in which the conditional execution of declarative or global statements is useful. Some examples follow. 1) If the value of a variable falls into a certain range then the variable is dropped from, or kept in, the data set. 2) The length of a character variable is 200, but it is known that the longest observation has a length of less than 200. The length of each observation is evaluated, and the length of the variable is then set to the longest observation. 3) A host command is submitted based upon some condition in the DATA step. 4) The values of a variable X are compared to another variable Y, and if the two variables are identical one of them is renamed or dropped.

The following example is a faulty attempt to set the length of a variable according to the value of another variable.

\[ \text{data a; } \]
\[ x = 0 ; \]
\[ \text{if } x = 1 \text{ then length y } $ 10 ; \]
\[ \text{run; } \]

Once again, the above program does not explode suddenly making the user feel like the captain of the Titanic. However, regardless of whether X is 1, 10, or 1,000,000 the length of Y will always be 10. Similarly, the following, syntactically correct, program is intended to drop the variable Y if X and Y are the same. However, Y will always be dropped, regardless of what X is.

\[ \text{data one; } \]
\[ y = 0 ; \]
\[ x = 0 ; \]
\[ \text{if } x = y \text{ then drop y; } \]
\[ \text{run; } \]

Both of the above programs attempt to conditionally submit a declarative or global statement, but none of them achieve the desired effect. In most cases, SAS does not even warn the unsuspecting user of their impending doom by placing errors or warnings in the log. Neither of the above examples can be accomplished using simple IF-THEN syntax. They must be executed using dynamic programming techniques. Dynamic code changes on the fly, in response to criteria established within the program. Dynamic code can be considered the antithesis of hard coding. Using dynamic code the global and declarative statements which are to be executed conditionally are generated only if the condition is true. Therefore, the SAS compiler only "sees" the statements if the condition is met. In the following case

\[ \text{if } x = 2 \text{ then } x \ ' \text{dir c:\'} ; \]

the

\[ x \ ' \text{dir c:\'} \]
portion of the code is submitted to the compiler only when \( X \) is 2. If \( X \) is not 2, the compiler never "sees" the host command. The event can potentially be produced using as many as five techniques. The methods presented in this paper have been developed under version 6.04 of the SAS System for DOS where only three of the five processes are available. Two procedures, both utilizing the call execute command, are not presented (Observations 1994). In addition to version 6.04, some of the code presented in the paper has also been tested under version 6.06 of the SAS System for CMS. The first method presented uses the PUT statement to write SAS code to an external file; the code is later recalled using a \%INCLUDE statement. The second technique utilizes the CALL SYMPUT routine, contained in the macro language, to assign the desired code to a macro variable which is then referenced. The final way to avoid running into an iceberg, combines the macro language with the PUT and \%INCLUDE statements.

The examples presented, for showcasing the different approaches to writing dynamic code, all focus on the conditional execution of the DROP statement. Often times a user will receive or create a data set with uninitialized variables. (For the purposes of this paper, an uninitialized variable will be defined as one which has missing values for every observation. The technical definition of an uninitialized variable is actually narrower; however, the final results are the same.) The creation of such variables probably falls into that nebulas zone known as sloppy code; something your mother might yell at you for. However, sloppy or not, they still creep into data sets. Anyone who has ever referenced an uninitialized variable, after seeing it listed in the VAR window or in the output of a CONTENTS procedure, understands the aggravation that can arise: threats to put a foot through the monitor, or the desire to see how well your computer flies through windows, and not ones built by Mr. Gates. The following examples have been developed specifically to remove the pesky uninitialized variable from a data set.

Method 1: PUT and \%INCLUDE Statements

The first method involves writing standard DATA step code to an external file using the FILE and PUT statements. The code is then retrieved into the program with a \%INCLUDE statement. First an external file is specified via a FILE statement. Next, the criteria for the event is established: the value of \( X \) is missing. If the condition is true then

\[ \text{drop } x; \]

is written to the external file by the PUT statement. The DATA step is then terminated with an implicit or explicit RUN. (An implicit run can be another procedure or DATA step.) The RUN must occur before the \%INCLUDE statement which calls the external file that was just written. In the DOS environment (the author assumes other environments are similar, CMS is not), SAS does not close the file until the completion of the DATA step. As a result, SAS cannot open the file, to read it, until the DATA step is completed, which is signaled by an explicit or implicit RUN. Finally, the DROP statement is included in the program through the use of the \%INCLUDE placed in another DATA step.

/* Turns the SOURCE2 option on. The contents of the external file, known as secondary code, will now be included in the SAS log. */
options source2;
/* The data we were given. We would never be so sloppy as to write code that produced an uninitialized variable. Oh no!! */
data one;
  y=1;
/* That pesky uninitialized variable. */
x=.;
/* Data step that evaluates the variables in the source data set and writes the DROP statement to an external file, if the condition is met. */
data _null_;
  set one;
/* Specifies the external file to write to. Exact syntax of the FILE statement is host dependent. */
  file 'd:\temp.cod';
/* If \( X \) is missing then the code drop x; is written to the external file specified above. Notice the semicolon after the \( X \) and inside the quotes. The code inside the quotes must be syntactically correct SAS code. */
  if x= . then put 'drop x;';
/* An explicit or implicit RUN must occur before the \%INCLUDE statement. SAS keeps the file d:\temp.cod open until an implicit or explicit run is encountered, and as long as the file is open SAS cannot open it again with the \%INCLUDE statement. */
run;
data fixed; set one;
/* Includes the code written to the external file. */
%include 'd:\temp.cod';
run;
One advantage to this method is that the external file can be viewed, allowing the user to easily scrutinize what is occurring. To make the task easier, turn the SOURCE2 system option on. The contents of the external file, known as secondary code, will then be included in the SAS log (SAS Language: Reference).

**Method 2: Call Symput**

The second method employs the CALL SYMPUT routine to pass the desired DROP statement to a macro variable. In the _NULL_ DATA step, X is once again evaluated to determine if it is missing. If it is, then
drop x;
is assigned to a macro variable through the CALL SYMPUT routine. The macro variable is then referenced in the DATA step FIXED. When the macro variable is referenced, the appropriate code is generated for the SAS compiler.

```sas
/*The MPRINT system option displays
the code generated by the macro
variable MACROVAR in the SAS log.*/
options mprint;

/*The data set we created. X is uninitialized. OK, maybe we do
call sloppy code.*/
data one;
y=1;
x=.;

/*If X is missing then the value
DROP X is assigned to the macro
variable MACROVAR through the use of
the CALL SYMPUT routine.*/
data _null_
set one;
if x=. then call
symput('macrovar', 'drop x');
else call symput ('macrovar', '');

/*The DATA statement serves as the
implicit run. The original data set is
read. Then, the macro variable is
called, dropping the variable
X.*/
data fixed;
set one;
&macrovar;
run;
```

The _NULL_ DATA step reads the original data set and evaluates the variable X. If X is missing, the CALL SYMPUT routine is implemented. The first argument to the CALL SYMPUT routine is the name of the macro variable. The second argument is the value to be assigned to the macro variable. Both arguments are enclosed in quotes because the macro variable is to literally be called MACROVAR and the value of the variable is to literally be DROP X. If MACROVAR had not been enclosed in quotes and had been a variable in the data set with the value BOB, the name of the macro variable would have been the value of MACROVAR: BOB. For further information see the SAS Guide to Macro Processing, Second Edition. The macro variable MACROVAR is then referenced with an ampersand followed by the name of the macro variable. There is no space between the ampersand and the name of the macro variable. Finally, turning on the MPRINT system option causes the code generated by the macro variable MACROVAR to be printed in the log, which allows the code to be more easily scrutinized for those creepy crawly things (bugs).

**Method 3: Macro with FILE, PUT, and %INCLUDE**

The third method uses the macro facility combined with the PUT and %INCLUDE statements. First a macro is created which ultimately takes on the value DROP X. The statement which calls the macro is then written to an external file. The macro call is then retrieved into the program through a %INCLUDE.

```sas
/*Options to help discover those
creepy crawly things.*/
options source2 mprint mtrace
symbolgen;

/*Creates macro named DROPX. The
macro has one parameter: VALUE X.
The value of X is fed into the macro
parameter, on each iteration of the
_Null_ DATA step, which occurs
later.*/
%macro dropx (value_x);

/*Evaluates macro variable
VALUE X to determine if it is
missing. If VALUE X has a null
value (it is missing), then the
macro resolves to the value DROP
X. Note that the statement DROP
X is not terminated with a
semicolon.*/
%if &value_x = %or &value_x =.
%then %do;
  drop x
%end;

/*End of the macro.*/
%mend;

/*The data set.*/
data one;
y=1;
x=.;

/*The DATA _NULL_ statement reads
the data set ONE, and writes the
macro call
&DROPX (:);
to an external file, specified by
the FILE statement. The missing
```

1122
value, or period, contained in the parentheses is the value of X. If the value of X is 1, the macro call becomes:

```sas
%DROP (1); /*
data _null_;
set one;
file 'c:\temp\temp';
put %dropx('x ');
*/
```

The data set FIXED reads the data in the original data set. The macro call which was written to the external file in the previous _NULL_DATA step is included. The macro call then invokes the macro DROPX. If X is missing, it is dropped from the data set FIXED. */

data fixed;
set one;
%include 'c:\temp\temp';
run;

The macro, named DROPX, establishes the criteria and evaluates the value of X to determine if it should be dropped. Of course, it should be; if it is after all just a pesky uninitialized variable. The value of the variable X is passed to the macro with the PUT statement in the _NULL_DATA step. The _NULL_DATA step writes the line %DROPX(). The name of the macro is DROPX. To call a macro, the macro name is preceded by a percent sign. The value of the variable x: missing is then inserted within the parentheses, so it may be passed to and ultimately evaluated by the macro facility. The macro itself begins with a macro definition of %MACRO followed by the macro name, DROPX. The parentheses then contain the name of the variable to which a value can be passed. The macro variable VALUE_X is then evaluated just as any variable in a DATA step. However, within a macro, variable names are preceded by an ampersand, and what are normally executable statements are preceded by a percent sign. In the macro language, there are no numeric variables, only character variables. As a result, if a numeric variable, with a missing value, is passed to a macro variable, the macro facility "sees" a period which character variables do not consider missing. One way to correct the problem is to first change the variable type to character, or to allow both spaces or periods to be treated as missing. The latter approach is taken by using the expression

```sas
%or &value_x = .;
```

A Useful Application

Three methods of conditionally issuing a DROP statement have been presented. However, each of the methods have evaluated only a single record. Issuing the DROP statement based on the value of a variable in a single record is not very useful. Greater power accrues from conditionally executing a declarative statement when the condition is based upon the collective values of all observations in the DATA step. The following code demonstrates how every record in a data set can be evaluated before issuing the DROP statement.

```sas
/*CREATES 40 observations. I and A have values 1 to 40. X is uninitialized */
data one;
do I = 1 to 40;
a=I;
x=.;output;
end;

data _null_;
/*The END option of the SET statement creates a variable, in this case named last, which is set to 1 when the last observation in the data set being read: ONE is read. See the SET statement in the Sas Language: Reference for more information.*/
set one end=last;

/*If X is NOT missing then X_1 is set to 1.*/
if x ne . then x_1 = 1;

/*The value of X_1 is retained throughout all future iterations of the DATA step. Therefore, if X is greater than missing, X_1 becomes 1 and is 1 for all subsequent observations.*/
retain x_1;

/*On the last observation (specified by IF LAST), X_1 is evaluated. If it is missing, the value of X has never taken on a value other than missing, and the value DROP X is assigned to the macro variable DROPX, via the CALL SYMPUT statement.*/
if last and x_1 = . then call symput ('dropx', 'drop x');
else call symput ('dropx', ' ');;

/*Data set FIXED calls the macro variable DROPX to drop the variable X, if it is missing.*/
data fixed;
set one ;
&dropx;
run;
```

DATA step ONE creates a data set with 40 observations, where X is always missing. In the _NULL_DATA step, a new variable X_1 is created. If X is ever greater than missing, X_1 is set to 1. The value of X_1 is retained throughout all future iterations of the DATA step. Its value will only be reset if X is greater than missing, at which time X_1 will be set to
Posters

one. Therefore, at the end of the DATA step, if X is ever greater than missing, X_1 will equal 1. At the end of the DATA step the variable LAST is set to one, or true (SAS Language: Reference). When the last observation is read from data set ONE, X_1 is evaluated. If it is missing, X has never been greater than missing and the statement

\texttt{drop x}

is assigned to the macro variable DROPX by using the CALL SYMPUT routine. The macro variable is then called, dropping the variable X.

Conclusion

Three techniques for conditionally executing statements which cannot normally be implemented conditionally have been presented. The ability to conditionally execute global and declarative statements is especially powerful when all values of a data set are first evaluated.

References


SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. CMS is a registered trademark or trademark of International Business Machines Corporation. © indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Brian O. Mullin
450 W. State Street
1st floor, P. O. Box 83720
Boise, Idaho 83720-0036
mull104w@wonder.em.cdc.gov