Paper BB03-2013

Not All Equals are Created Equal: Nonstandard Statement Structures in the DATA Step

Arthur L. Carpenter
California Occidental Consultants, Anchorage, AK

ABSTRACT

The expression is a standard building block of logical comparisons and assignment statements. Most of us use them so commonly that we do not give them a second thought. But in fact they definitely do deserve that second thought. A more complete understanding of their construction and execution can greatly expand our ability to more fully take advantage of this fundamental component of the SAS® Language.

Once we understand the basic form of the expression and how it is used in various statements, we can use this understanding to create statement forms that would otherwise appear to be illegal or just plain wrong. Further and perhaps even more importantly this deeper understanding can help to prevent us from committing errors in logic.

INTRODUCTION

Expressions are made up of any number of components. These include constants, function calls, variables, and operators. These components can appear individually or in conjunction with other components.

Expressions produce a result and are said to resolve to a value. Since this value is either numeric or character, a given expression is often referred to as either a numeric or character expression.

The simplest expressions will have a single component and the simplest form of an expression is a constant. Here these expressions are used in assignment statements, most likely in a DATA step.

```
n_var = 5;
c_var='fred';
```

Operators are used to create associations among two or more expressions. These operators include the arithmetic operators as well as logical and Boolean operators. Operators are like verbs in the expression. They tell SAS what to do with things like constants and variables.

Comparison operators join expressions to form a logical expression. Regardless of whether the comparison is being made on character values or numeric values, these expressions always result in a numeric value (0 or 1), False or True. Notice that these are not full statements, but simply composite expressions joined by logical operators. Both of these expressions will resolve to True or False (1 or 0).

```
name = 'Fred'
1 le month(dob) le 3
```

In order to avoid confusion and ambiguity, operators are assigned a hierarchy or order in which they are applied. The hierarchy is formed by seven groups of operators, and within a group, operators of equal rank are applied from left to right (except Group 1 which is applied right to left).

At a simple level, we need to understand why the expression (5+6*2) is equal to 17 and not 22. But as we encounter expressions in non-standard form, such as some of those discussed in this paper, we need to have a solid understanding of this hierarchy, if we are to understand why the expressions evaluate the way that they do.

Group	Operators
Parentheses	Operations within parentheses are performed first
Group 1 (performed right to left)	Exponentiation (**) Prefix operators, such as, positive (+), negative (-), and *negation*
Group 2	Multiplication (*) and division (/)
Group3	Addition (+) and subtraction (-)
Group 4	Concatenation (||)
Group 5	Comparisons such as equal (=) and less than (<)
Group 6	AND - Boolean comparison (&)
Group 7	OR - Boolean comparison (|)

Since any of these operators can appear in any expression, whether in an assignment statement or an IF statement, we need to expand our perception of what an expression *should* contain.

Logical Comparisons

Logical comparisons, which include the use of Boolean and comparison operators, resolve to either true or false (1 or 0). Usually the evaluation of these comparisons is fairly straight forward, however there are some interesting aspects that deserve further discussion.

When you construct the logic of the expressions, especially when Boolean operators are involved, you need to be careful not to translate the idioms of speech into the syntax of code. In this expression we want to check for Sally's identification numbers, which are either 2 or 3. In English we might say "if the ID number is 2 or 3 then the name is Sally", and we might write the code shown to the right. The problem is that this logical expression is evaluated differently by SAS than we intended it to be. The Boolean comparison operator OR compares the result of two distinct expressions, shown here through the use of parentheses. Since the number 3 is always true, the overall expression is also always true regardless of the value of the variable ID. In the more precise language of SAS expressions we should have specified each of the two possible ID values separately.

```
if id=2 or 3 then name='Sally';
```

```
if (id=2) or (3) then name='Sally';
```

```
if id=2 or id=3 then name='Sally';
```

As an aside, the programmer may be tempted to group the two possible values of ID by constructing the expression with parentheses surrounding the (2 or 3). Unlike the first statement this expression will NOT always be true. The Boolean expression, inside the parentheses, is evaluated first and is of course true. Then ID is then compared to the result, which necessarily must be 1. The resulting expression, which completely misses the point of the comparison, is shown to the right.

```
if id=(2 or 3) then name='Sally';
```

```
if id=1 then name='Sally';
```

All of these comparisons are syntactically correct; none produce errors, but only one produces the desired comparison. It matters how you construct logical comparisons.

Compound Comparisons

An expression which is made up of compound comparisons is evaluated as a series of expressions joined with an AND operator. In this comparison we would like to detect ages between 25 and 45 inclusively. This is equivalent to writing the compound expressions as two independent expressions joined with an AND.

```
if 25 le age le 45 then do;
```

```
if (25 le age) and (age le 45) then do;
```

From a programming standpoint there is no reason why we need to limit the composite comparison to just two comparisons. Indeed a series of comparisons can be made, each being effectively joined with an AND.

```
if 25 le age le 45 le maxage le 55 then do;
```

```
if (25 le age)
   &(age le 45)
   &(45 le maxage)
   &(maxage le 55) then do;
```

This interpretation of the composite comparison, as it is used in such places as the DATA step, is not correct for the macro language. Rewriting the first IF statement from above using macro language elements, could be expressed as shown to the right. This expression is evaluated from left to right and does not contain an implied Boolean operator. The left most expression is evaluated first and the result is then used with the next expression. In this example, regardless of the value contained in &AGE the overall expression will be true (both 0 and 1 are less than 45). Assume that &AGE contains 55, first the 25 is compared to 55 and a 1 is returned. The 1 is then compared to 45, and since it is less than 45, the expression seems to indicate that 55 is less than 45, but of course this is not true. We have in fact misused the composite comparison in the macro language, which requires us to fully specify the comparison explicitly using the Boolean operator as shown to the left.

```
%if 25 le &age le 45 %then %do;
```

```
%if (25 le &age) le 45 %then %do;
```

```
%if 25 le 55 le 45 %then %do;
```

```
%if 1 le 45 %then %do;
```

```
%if 25 le &age and &age le 45 %then %do;
```

EQUAL and PLUS SIGNS: TWO IN ONE

The assignment statement is characterized by a variable followed by an equal sign, which in turn is followed by an expression. The equal sign is not seen as an operator, but rather as a parsing character. The expression can be any numeric or character expression. This implies that the expression itself can contain operators, including an equal sign. Consider the assignment statement that creates the variable AGE14. This variable will be numeric and will take on the values of either 0 or 1 (AGE=14 is a logical comparison) depending on whether or not the variable AGE is 14. The parentheses are not needed and the expression could be rewritten without them. Although the statement may look like it is in a non-standard form, it is still actually in the form of the standard assignment statement, `varname=`*`expression`*`;` . Using an IF-THEN/ELSE construction this statement could be rewritten using logical comparisons, however the assignment statement will tend to be more efficient.

```
var_name = expression;
```

```
age14 = (age = 14);
```

```
age14 = age = 14;
```

```
if age=14 then age14=1;
else age14=0;
```

It should then be clear that these two equal signs are not being used in the same way by SAS. SAS will not be confused and neither will you, if you remember the form of the statement and keep track of the expression.

This is also true in the following seemingly more complex example. Here the programmer would like to take two actions (an assignment to the variables B and C) based on a logical comparison. The appropriate way to group these two actions is to use the DO block. Rather than use a DO block the code

```
if a=1 then b=2 and c=3;
```

```
if a=1 then do;
   b=2;
   c=3;
end;
```

Assuming that the expression a=1 is true let's look at the action, which in this case is an assignment statement, which is specified as
variable=expression;

```
        b=2 and c=3;
```

From the compiler's point of view this is the same as:

```
      b= (2 and c=3);
```

2 is always true, c=3 is either true or false. When C=3 is true:

```
      b=(1 and 1);
```

which evaluates to:

```
      b=(1);
```

which evaluates to:

```
      b=1;
```

When C = 3 is not true the variable B will be assigned the value of 0.

Actually this opens up other possibilities for us as well. The SUM statement, like the assignment statement, utilizes a special character following the variable name; in this case the plus sign. The plus sign which immediately follows the variable name is what makes this the SUM statement. You could not specify the SUM statement without it. If the number three is to be decremented from COUNT, the SUM statement must include the plus sign as well as the expression (the constant -3).

```
var_name + expression;
```

```
* this fails;
count - 3;
```

```
count + -3;
```

Since an expression, any numeric expression, can follow the plus sign, it stands to reason that the expression itself can contain a plus sign. Again SAS will not be confused, and now neither will you, since the statement has two different kinds of plus signs.

```
totaladj + budget + 500;
```

LOGIC WITHOUT THE LOGIC

In the following IF-THEN/ELSE statements we use the month of the date of birth to assign the season of birth.

```
if 1 le month(dob) le 3 then season = 1;
else if 4 le month(dob) le 6 then season=2;
else if 7 le month(dob) le 9 then season=3;
else if 10 le month(dob) le 12 then season=4;
```

Examine these IF-THEN/ELSE statements and notice the individual expressions and how they are used together to form other expressions. The composite expression `1 le month(dob) le 3`, is made up of three individual numeric expressions which are connected using the logical operator LE to form a single logical expression. The assignment statement `season=2;` has a constant numeric expression on the right side of the equal sign. Including the possibility of a missing value for DOB, these statements can yield a SEASON that varies from 1 to 4 and missing.

Rather than employ a series of IF-THEN/ELSE statements you could use the assignment statement shown here to create the variable SEASON. Not exactly equivalent to the statements above, this statement can result in the assignment of numeric values from 0 thru 4 depending on the month of the date of birth. The 'less-than-or-equal-to' comparison operators (Group 5) return a zero or one which is multiplied against the constants. The comparison operators are just another form of expression operators and are perfectly suited to assignment statements as well as to logical expressions. This form of logical assignment tends to be more efficient than the IF-THEN/ELSE statements.

```
season = 1*(1 le month(dob) le 3)
       + 2*(4 le month(dob) le 6)
       + 3*(7 le month(dob) le 9)
       + 4*(10 le month(dob) le 12);
```

In fact there is no reason why any of the logical and comparison operators cannot appear in an assignment statement. The key to their use is to remember that logical expressions will yield either TRUE or FALSE, which is represented by 1 or 0 respectively. For a date of birth in May the previous equation is evaluated as is shown on the left. The expression results in a value of 2 for SEASON. When you are generating a numeric value based on a logical determination, such as this one, you should be able to write the assignment statement in a form similar to the one above rather than the less efficient series of IF-THEN/ELSE statements.

```
season = 1*(0)
       + 2*(1)
       + 3*(0)
       + 4*(0);
```

Although the previous example could have also been made using a user defined format and a PUT function, the assignment of a value to GROUP using the series of IF-THEN/ELSE statements, such as the one shown here, does not so easily lend itself to a solution involving a format. The value can, however, be determined with an assignment statement containing the same logic as was used in these IF-THEN/ELSE statements.

```
if sex = 'M' and year(dob) >  1949 then group=1;
else if sex = 'M' and year(dob) le  1949 then group=2;
else if sex = 'F' and year(dob) >  1949 then group=3;
else if sex = 'F' and year(dob) le  1949 then group=4;
```

Since assignment statements tend to be processed faster than IF-THEN/ELSE statements, it is likely that the use of assignment statements can decrease processing time. This type of assignment statement will also generally out perform a look up using a PUT function.

```
group = 1*(sex = 'M' and year(dob) >  1949)
      + 2*(sex = 'M' and year(dob) le  1949)
      + 3*(sex = 'F' and year(dob) >  1949)
      + 4*(sex = 'F' and year(dob) le  1949);
```

BUILDING BINARY AND OTHER FLAG VARIABLE COMBINATIONS

It is not uncommon to need to create binary (0,1) variables that are to be used in subsequent analyses. There are a number of ways to create these variables, and depending on the result that is required, these variations can include the way that missing and negative values are to be treated. In addition to 0/1 flags, it is common to create flags that contain values such combinations as (./0/1) and (-1/0/1).

Logical Assignment

Since True/False determinations always result in either a 0 or a 1, logical expressions can be especially useful if assigning a numeric 0,1 value to a variable. In the following DATA step we would like to create a flag that indicates whether or not the date of birth is before 1950. Three equivalent variables (BOOMER, BOOMER2, and BOOMER3) have been created to demonstrate three different methods.

```
data flags;
   set advrpt.demog (keep=lname fname dob sex);
   if year(dob) >  1949 then boomer=1; ❶
   else boomer=0;
   boomer2 = year(dob) >  1949; ❷
   boomer3 = ifn(year(dob) >  1949, 1, 0); ❸
   run;
```

❶ Very often IF-THEN/ELSE statements are used. These statements tend to process slower than assignment statements.
❷ The logical expression appears on the right of the equal sign. ❸ The IFN function can be used to assign the result. This function has added value when a result other than just 0 or 1 is to be returned. For each of these determinations a missing DOB will result in a flag of 0.

A similar flag assignment can be made through the macro language. Here we want to flag values of &AGE that are greater than 25. The macro variable &AGEFLG will take on the value of 0 for values of age =< 25 (including a null value) and 1 for values greater than 25. Notice the use of the %EVAL function to force the comparison. As in the above DATA step example, the IFN function can also be used in this type of situation. The %SYSEVALF function also supports an optional second option. When this option takes on the value of BOOLEAN the function returns a binary value. Here &VFLAG will be 1 for all values of &VAL except 0 (the function fails if &VAL is null).

```
%let ageflg = %eval(&age > 25);
```

```
%let flg = %sysfunc(ifn(%eval(&age>25),1,0));
```

```
%let vflag=%sysevalf(&val,boolean);
```

Using a Double Negation

The logical NOT operator can be used to build binary variables by using a double negation. Negation of any false value will be true. Since 0 and missing are false negating either of these values will result in a 1. Conversely any non-false value (any value that is not 0 or .) will be negated to a 0. The use of a double negation will therefore result in a 0 or 1. Consider the value 5.6 (which is evaluated as true), negating it will result in 0 (false), negating 0 results in a 1 (true), therefore ^^5.6 maps to 1. This is demonstrated in the following example.

In this example, which was suggested by Chang Chung and Mike Rhodes, we want to create a binary flag which will indicate whether or not a specific number is stored in any of a number of variables in an array. Here we need to determine if the number 3 is contained in one or more of the variables X1 through X4.

```
data _null_;
input x1-x4;
array a {*} x1-x4;
flag1 = (3 in a); ❹
flag2 = ^^whichn(3,of x:); ❺
put flag1= flag2=;
datalines;
1 3 5 7
5 6 7 8
run;
```

❹ FLAG1 is created by using a logical expression `(3 in a)`, which checks to see if the value 3 is in the array A. If it is, a 1 is returned, otherwise a 0.
❺ The WHICHN function returns an item number if the value in the first argument is found in the succeeding argument(s) (here a list of variables). If the value is not found a 0 is returned. Since we are only interested in building a binary flag, the returned value is converted to a 0 or a 1 by the use of a double negation. For the first observation, the WHICHN function returns a 2 (a 3 is found in the second variable – X2), this is negated to a 0 (^^2 becomes ^0), which is in turn negated to a 1(^0 becomes 1).

As long as we do not think of a null value as equivalent to missing, a similar approach can be taken with the macro language. The double negation of macro variables that contain integers will also produce 0, 1 results.

```
%let ageflg = %eval(^^&age);
```

Here the double negation converts all non-zero values of &AGE to 1. For non-integer values of &AGE the %SYSEVALF function could be used instead of %EVAL. This gives us a similar result to the %SYSEVALF example using the BOOLEAN option which was shown above.

Replace Missing with 0

For reporting purposes missing values can be replaced by a 0 using a simple assignment statement. The COALESCE function returns the first non-missing value. In this example if DOB is missing a 0 is returned. Prior to the inclusion of the COALESCE function, this same operation was sometimes accomplished using the SUM function. Be careful when working with dates as was done here. Remember that, although both are false, a date of missing and a date of 0 have different meanings.

```
z = coalesce(dob,0);
```

```
y = sum(dob,0);
```

The previous two expressions do not result in a Boolean value. If you want to convert all missing values to 0 and all other values to 1 (including 0) you can use the negation of the MISSING function. In this expression MVAL will be 1 for all numbers except missing.

```
mval=^missing(val);
```

Determine Positive or Negative Values

Numeric values fall into one of four groups; ., <0, 0, >0. Relative to 'positive' and 'negative' and with the potential of missing values and the 0 value, we have four distinct ways of grouping these values into binary flags. A given value can be positive, not positive, negative, and not negative. Because of the presence of both the 0 and missing, the groups of values that are positive and those that are non-negative are not necessarily the same. Fortunately we can build a Boolean flag for each of these four possibilities, with the use of the SIGN function, which returns -1 for values < 0, 0 for values=0, 1 for values > 0, and missing for missing values.

```
data posneg;
   do v=.,-2 to 2;
      *if positive;
      pos = sign(v)=1;
      * Not positive;
      notpos = (sign(v) in(-1,0));
      * Negative;
      neg = sign(v)=-1;
      * Not negative;
      notneg = sign(v) in (0,1);
      output posneg;
   end;
   run;
```

```
Boolean Conversions
Positive or Negative?
Obs   v   pos   notpos   neg   notneg
 1    .    0      0       0      0
 2   -2    0      1       1      0
 3   -1    0      1       1      0
 4    0    0      1       0      1
 5    1    1      0       0      1
 6    2    1      0       0      1
```

The SIGN function can also be used in the macro language to achieve the same flags for macro variables. The %EVAL function forces the numeric comparison with the result of the SIGN function. Notice the logic is slightly different as we do not have to worry about missing values.

```
%let pos    = %eval(%sysfunc(sign(&v))=1);
%let notpos = %eval(%sysfunc(sign(&v))<1);
%let neg    = %eval(%sysfunc(sign(&v))=-1);
%let notneg = %eval(%sysfunc(sign(&v))>-1);
```

DO LOOP SPECIFICATIONS

The iterative DO loop is usually defined using a single loop specification. The most common form of the specification is taken directly from the documentation; however there are a number of variations that are less commonly applied. Looping can be controlled using combinations of specifications that include constants and compound specifications and these can be combined with DO WHILE and DO UNTIL specifications.

The index variable for the DO loop can be a declared to be a series of either numeric or character constants that are comma separated.

```
do count=1, 26, 33;
do month = 'Jan', 'Feb', 'Mar';
```

In addition to constants, for numeric index variables the compound loop definitions can contain a combination of iterative and constant specifications. Here the variable COUNT will take on the values of: `1, 2, 3, 5, 10, 15, 20, 26, 33.`

```
do count=1 to 3, 5 to 20 by 5, 26, 33;
```

The iterative DO loop is evaluated at the bottom of the loop. This means that the index variable is incremented and then evaluated. For the DO loop shown here, the variable COUNT exits the loop with a value of COUNT=4. By adding an UNTIL specification we can prevent the index variable from being incremented beyond the maximum value. When this iterative DO loop terminates the variable COUNT exits the loop with a value of COUNT=3.

```
do count=1 to 3;
```

```
do count=1 to 3 until(count=3);
```

The DO UNTIL and DO WHILE loop forms of the DO loop will be executed indefinitely until some exit criteria is met. For these loops any index variable must be incremented manually by the programmer, often with a SUM statement as is shown here. It is also possible to set up an infinite loop with conditional exit and increment the index variable automatically. Notice that this iterative DO loop does not have a TO specified, therefore it does not automatically have an exit criteria based on K. It does however automatically increment the value of K for each iteration.

```
do until(x=5);
  k+1;
```

```
do k=1 by 1 until(x=5);
```

SUMMARY

Are the statement forms shown in this paper really that unusual or non-standard? If they are not commonly used then it is most likely due to the fact that programmers lack an understanding of how expressions are used and evaluated in statements such as SUM and assignment statements.

There is so much that can be easily done with these statement structures. Programming alternatives are often convoluted and less efficiently executed. It is definitely worth the programmer's effort to understand these statement forms.

ABOUT THE AUTHOR

Art Carpenter's publications list includes five books, and numerous papers and posters presented at SUGI, SAS Global Forum, and other user group conferences. Art has been using SAS® since 1977 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Advanced Professional programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

ACKNOWLEDGEMENTS

The instigation for this paper was in part suggested by Dr. Chris Ake of Kaiser Permanente.

REFERENCES

Many of the examples in this paper have been borrowed (with the author's permission) from the book Carpenter's Guide to Innovative SAS® Techniques by Art Carpenter (SAS Press, 2012).

Several of the code examples on Boolean transformations have been suggested by Howard Schreier in the sasCommunity.org article
http://www.sascommunity.org/wiki/Numeric_transformations

The sasCommunity.org tip
http://www.sascommunity.org/wiki/Tips:Creating_a_flag_avoiding_the_If_..._Then_Structure discusses the use of this type of expression in an assignment statement. The discussion tab includes alternative forms that can be used in an SQL step.

The example of flags used to indicate presence of a value in a list was suggested by Chang Chung and Mike Rhoads. Their examples can be found at:
http://www.sascommunity.org/wiki/Tips:Double_negatives_to_normalize_a_boolean_value and
http://www.listserv.uga.edu/cgi-bin/wa?A2=ind1101c&L=sas-l&D=1&O=D&P=9693, respectively.

There are a number of related topics that touch on our understanding of the various logical and comparison operators. More information can be found at:
http://www.sascommunity.org/wiki/Tips:DATA_Step_Comparison_Operators_Are_Non-Associative

TRADEMARK INFORMATION