SESUG 2020 - Paper 152

A Game Plan for Beating the IF-THEN-ELSE Overhead in DATA Steps

Jason J. Su, PPD, Inc., Research Triangle Park, North Carolina

ABSTRACT

The IF-THEN-ELSE logic enables programs to conditionally execute transformation codes and may be the most ubiquitous element in SAS® DATA steps. However, it is known that albeit its intuitive structure and simple syntax, the construct comes with discernable computational overhead for SAS programs, which exacerbates dramatically for the modern data containing millions or billions of records. Fortunately, there are multiple powerful technique advances and recent SAS upgrades at the programmer's disposal to bypass the logic. However, it has become baffling for programmers to promptly pick the right technique for the different tasks. Therefore, a practical game plan is devised here which is composed of five (5) short evaluation questions and their straightforward technical solutions. If followed, the game plan can help programmers avoid the overuse of the IF-THEN statement and land them lickety-split the specific tool for the job at hand with much reduced or no overhead.

INTRODUCTION

IF-THEN-ELSE logic basically controls execution of some codes upon certain conditions are satisfied, is a truly ubiquitous component in the programming world. In SAS, besides its usage in macro program as well in PROC SQL in the appearance of CASE-WHEN-THEN structure, the logic is primarily employed in DATA steps. Due to the nature of the DATA steps, every executable statement is processed at default for each observation from the input data unless some logic statements such as IF-THEN-ELSE were put in place, therefore it is a primary duty for a programmer to ensure all executable statements are required and necessary for the data derivation. However, even these IF-THEN-ELSE logics themselves are coming with discernable computational overhead, especially when the data to be processed contain millions or billions of records as in the national healthcare or survey data.

Fortunately, over the years, SAS and its ardent user community have come up with an array of efficient tools which may serve well as alternatives to the traditional IF-THEN construct. They are composed of multiple delicate functions and powerful programming structures, where the conditional processing is either already explicitly incorporated or is totally tolerable with the situations where the IF-THEN logic tries to handle. The former includes SAS functions such as IFC, IFN, etc. and the later, SAS functions such as CATS, CATX, COALESCE, COALESCEC, etc. and the Do-loop of Whitlock (DoW) structure. Additionally, there are new functions like WHICHC, WHICHN, CHOOSEC & CHOOSEN, and other elements such as SELECT statements & comparison operators like ( ) at SAS users' disposal for circumventing the issue and improving program efficiency. With all these powerful tools around, how does a programmer, when facing a programming task, quickly analyze the situation and choose the right tool for the task at hand?

Here for my day-by-day SAS programming, I draw up a simple game plan as in Figure 1, which contains five (5) short checking items, and instantly land me the right techniques to handle the issues at hand. The plan itself and the involved tools are not intended to be all-

inclusive but instead to provide one exemplary and straightforward guideline to assist fellow SAS programmers in finding an alternative tool with much reduced or no overhead. With the principle, SAS users are encouraged to develop their own idiosyncratic plans to tackle their daily tasks. Real-world examples are used to demonstrate the straight-forward application of these guidelines in daily work.

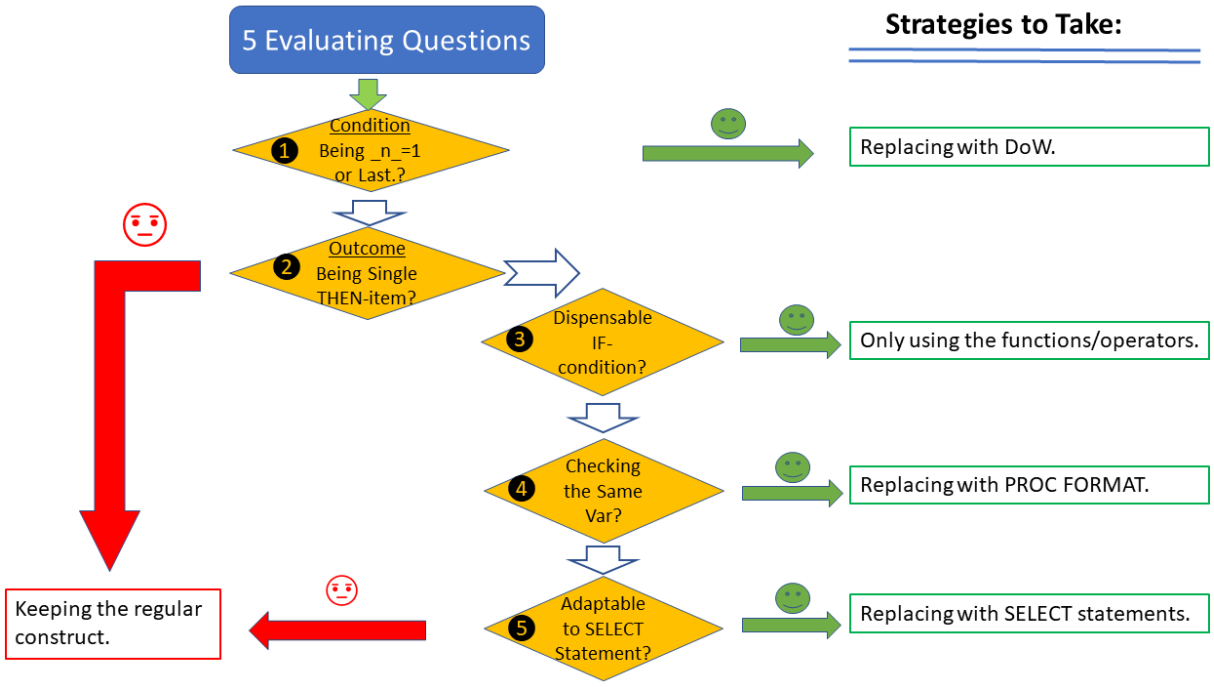

Figure 1. Game Plan with 5-Step for Assessing the Task and Taking the Replacement Strategies

THE GAME PLAN IN A NUTSHELL

As a cheat sheet for programmers, the game plan is composed of five (5) short evaluation questions to check lickety-split when the case is up for employing an IF-THEN logic. Out of them, the first two are aiming at the IF-part and the THEN-part respectively, and the rest three are only carried out only if the evaluation on the second question is affirmative. These questions and their corresponding strategies are elaborated as follows:

1. Condition being _n_=1, or last.?

   Long version of this evaluation is: Do the conditions contain system automatic variables such as _n_, end statement option-based (end=), or LAST., and are only run once in the beginning or end of the by group or the whole data set? If so, DoW is considered.

   Typically, these conditions are needed in the beginning for initiating the objects such as hash or variables before starting accumulation and in the end for outputting the last record or summary results for the by group. Please note that either single or multiple outcome then-items can be contained in the logic.

2. Outcome being single THEN-item?

   Long version: Is the THEN-part a single item? If not, a regular IF-THEN-ELSE structure

is used.

For example, in this statement "if VAR1=5 then var2=1", there is only one THEN-item which is "var2=1". If the evaluation is negative, the process is terminated, and a regular IF-THEN-ELSE structure is used. Otherwise, the process will go on for the next three evaluations before settling with the regular IF-THEN blocks. However, in daily programming, it can be applied flexibly for us to creatively use other powerful techniques, as shown in following scenarios.

3. Dispensable IF-condition?

   Long version: Is the IF-condition dispensable? If so, it should be simply deleted.

   Although the evaluation seems so obvious, you might be really surprised by the number of the superfluous IF-THEN logics based on this evaluation. In this case, the THEN-item is a relatively simple conversion typically where only 1-2 functions or a comparison operator is involved.

4. Checking the same variable?

   Long version: Are the same variables used in comparison repetitively? If so, it can be converted to PROC FORMAT.

   This evaluation is for this scenario: One variable is to be checked with a sequence of values and the other variable is derived accordingly. If this evaluation is affirmative, the FORMAT procedure is a proper fit due to the advantage of readability and computational efficiency. Otherwise, the process moves to the next evaluation. Please note that this evaluation and the previous (dispensable IF-condition?) can both turn out to be true, and multiple solutions may be obtained.

5. Adaptable to SELECT statement?

   Long version: Can it be converted to SELECT statement?

   Like IF-THEN-ELSE block, SELECT-WHEN statement checks for a series of mutually exclusive conditions. However, there is a big advantage of SELECT statement over the regular IF-THEN-ELSE: It is easy to read and debug in the program.

The evaluation order is based on the recognition of an order of the combined power of efficiency and readability in these techniques, which is

function/operator/DoW (from Evaluation questions #1, #2 & #3)> PROC FORMAT (from #4)> SELECT statements (from #5)> IF-THEN logic.

The highest rank item contains SAS functions such as CATS, CATX, COALESCE, COALESCEC, IFN, IFC, etc. comparison operators, and DoW structure. The application of the game plan is stepwise demonstrated in the following scenarios.

SCENARIO #1 STRATEGY TAKEN: REPLACING WITH DOW

The DoW structure was developed and perfected primarily by Whitlock, Henderson, Dorfman and others. Due to the inherent nature of isolation of a certain break-event from actions performed before and after the loop, it elegantly renders the conditional statements superfluous, where the break-event is the start or end of the file, the start or end of a BY group, etc. One popular example for the former is to set up a hash object in the beginning of the program.

Suppose we want to compare a list of medications packed in one variable (TERM1) with another list (TERM2) on each record and output the new medications only in the 2nd list

(NEW_MED) as well as a flag (NEW_MED_FLAG). With Hash, a regular IF-THEN logic might go this way:

```
data new_medication;
    length new_med_flg $1 new_med $200;

    if _n_=1 then
          do;
             declare hash h();
             h.definekey('drug');
             h.definedone();
          end;

      set work.test;

      **STORING THE DRUGS INTO THE HASH;
      if ^missing(term1) then
            do i=1 to countw(term1,';');
                drug=scan(term1,i,';');
                rc  =h.ref();
            end;

      **COMPARING AND FLAG;
      if ^missing(term2) then
            do i=1 to countw(term2,';');
                drug=scan(term2,i,';');
                if h.check() then
                    do;
                        **SETTING FLAG AND NO NEED TO CHECK FURTHER;
                        new_med_flg='Y';
                        new_med=catx(';',new_med,drug);
                    end;
            end;

      h.clear();
    drop i;
run;
```

According to the evaluation question #1 of the game plan (Condition being _n_=1 or last.?), it contains _n_=1 testing and is perfectly eligible for the DoW structure. The new replacement is to simply list these THEN-items before the DoW loop as follows:

```
data new_medication;
    length new_med_flg $1 new_med $200;

    declare hash h();
    h.definekey('drug');
    h.definedone();

    do until (eof);
        set here.test end=eof;

        **STORING THE DRUGS INTO THE HASH;
        if ^missing(term1) then
            do i=1 to countw(term1,';');
                drug=scan(term1,i,';');
                rc  =h.ref();
```

```
            end;

        **COMPARING AND FLAG;
        if ^missing(term2) then
            do i=1 to countw(term2,';');
                drug=scan(term2,i,';');
                if h.check() then
                    do;
                        **SETTING FLAG AND NO NEED TO CHECK FURTHER;
                        new_med_flg='Y';
                        new_med=catx(';',new_med,drug);
                    end;
            end;

         output;
         h.clear();
    end;

    stop;
    drop i;
run;
```

The new program automatically sets up the hash object once for all and bypassing repetitive checking of the same condition (if _n_=1 then) for each observation in the data set, therefore increasing the program efficiency.

For the case of the break-event being the start or end of a BY group, a common example is to calculate the summary. If a regular IF-THEN logic is used, it would be necessary to conditionally set up and initiate a variable in the beginning of every BY group and therefore the condition is executed in every observation as follows.

Suppose we need to calculate the average of heart rate (HR) for each SUBJID from a clinical vital sign dataset basing on all the records for the SUBJID. Normally this task may be implemented with a FIRST. variable checking to initiating the value to zero for each SUBJID and a LAST. variable checking to calculate the final values and output the final record as follows:

```
data summary;
    retain sum count;
    set vitals;
    by subjid;

    if first.subjid then call missing(sum,count);
    sum = sum(sum,hr);
    count=sum(count,1);
    if last.subjid then
        do;
            mean = sum/count;
            output;
        end;
run;
```

According to the evaluation question of the game plan #1 (Condition being _n_=1 or last.?), IF-THEN logic here contains LAST. and FIRST. system variables, and it is applicable for DoW structure. The new logic is implemented simply with the inherent checking for the first and the last records as well as the automatic retaining and re-initiation capacities for the new

variables such as SUM and COUNT. Once the break-event happens that is the end of the BY group, the MEAN is naturally calculated. The new program may be implemented as follows:

```
data summary;
    do until (last.subjid);
        set vitals;
        by subjid;

        sum = sum(sum,hr);
        count=sum(count,1);
    end;
    mean = sum/count;
run;
```

Although only two executable IF-THEN and one declarative RETAIN statements are eliminated in this structure, the benefit on the saved system source or computational efficiency might be significant especially for data containing millions or billions of records.

For us to quickly test this, a dataset was created having the two variables (SUBJID & HR) and 105 million dummy records. In one testing environment, SAS Grid is used but the disturbance of the features such as load sharing was minimized through the above two snippets being submitted in one SAS program and later processed by the same server CPU. The regular DATA step spent 8.42 seconds CPU or real time processing the data whereas the DoW 7.40 seconds. Additionally, a non-grid X64_SRV12 WIN 6.2.9200 SAS Server tested the same program, and the result was: The regular DATA step 10.11 seconds CPU time or 10.94 real time vs. the DoW 8.90 seconds CPU time or 9.52 real time. Therefore, DoW construct seems to be 10% more efficient for the dataset across the two platforms. Indeed, replacing the IF-THEN conditions containing _n_=1 or last. with the innovative DoW construct does expediate the processing, especially for datasets with millions or billions of observations.

It is noted that DoW loop was included here primarily for the purpose of IF-THEN logic replacement but the loop itself is an extremely flexible structure and far greater power has been explored. Over the years, multiple variations are developed such as double DoW or nested DoW to elegantly handle double transpose, hash of hash, and other rather complex issues. No further examples are given here due to space limitation, but readers are encouraged to check out the intricate delicacy of this structure.

Additionally, the evaluation question #1 of the game plan (Condition being _n_=1 or last.?) is applicable for the end-of-file checking commonly used for macro variable defining with SYMPUT or SYMPUTX routine at the end of the DATA step as follows:

```
set data end=eof;

**REGULAR PROCESSING CODES HERE;
Total=sum(total,somevalue);

if eof then call symputx('total_value',total);
```

This condition to check whether the end of the file is reached can be just skipped and the new version simply be as:

```
call symputx('total_value',total);
```

Although SYMPUT or SYMPUTX may carry some overhead itself, the difference between the two overheads is arbitrarily deemed negligible and I personally would not use the IF-THEN

logic unless the data set contains millions of records. Additionally, in the absence of this kind of logic, programs become easier to read and debug.

SCENARIO #2 STRATEGY TAKEN: ONLY USING THE FUNCTIONS/OPERATORS

FUNCTIONS ALONE

The evaluation questions #2 (Outcome being Single THEN-item?) and next #3 (Dispensable IF-condition?) should be always raised whenever a single function is used. For many functions, the IF-THEN logic is already built inside and therefore the additional checking is superfluous. Some examples are listed as follows:

```
if site ne '' then siteid=strip(site);
if svdtn ne . or dthdtn ne . then rfpendtn= max(svdtn, dthdtn);
```

Since these functions STRIP and MAX as well as other SUM functions such as MIN, MEAN, ROUND & SUM can perfectly handle the argument-being-null conditions, IF-THEN logics are redundant. Its elimination does not affect the final data. For MAX function, for us to avoid the possible warning (i.e. Missing values were generated) in the log, an additional argument of 0 can be appended if allowed. The new statements would be as follows:

```
siteid=strip(site);
rfpendtn= max(svdtn, dthdtn, 0);
```

Sometime, a little tweaking is needed as in the following example:

```
if str='' then str=memname;
   else str=strip(str)||' '||memname;
```

Since the traditional concatenation operator (||) does not properly handle the operand-being-null situation, these two statements can be combined and replaced with a CATX function as follows:

```
str=catx(' ',str,memname);
```

If STR is null, CATX function will automatically skip it for concatenation, as in the original statements dictate.

Additionally, LENGTH function is used for determining the length of a character value, but if the character is null, it still returns a value of 1 which may generate erroneous results. Traditionally IF-THEN logic is used to prevent this miscalculation as follows,

```
if ^missing(str) then var_length=length(str);
    else var_length=0;
```

This can be easily replaced with a sister function LENGTHN, which returns 0 for null arguments and skips the IF-THEN-ELSE structure totally as follows,

```
var_length=lengthn(str);
```

IFC/IFN FUNCTIONS

Functions such as IFC and IFN might be the natural replacement if these primary functions or comparison operators cannot handle the conditions traditionally set in the IF-THEN logic. The two functions are relatively new Base SAS functions since version 9 whose result depends on whether a user-supplied logical expression is true, false, or missing. IFC returns

a character result and IFN a numeric result. Although the same logic expression may be evaluated the function counterparts compared with generic IF-THEN logic may run faster. Additionally, this construct makes the program less cluttered. For instance, we have two data sources first stacked together, and then derive visit starting time and end time from visit time or lab time, depending on the data source. If we resort to the IF-THEN construct, it might be as follows:

```
data sv2;
   set sv1(in=a)
       lb1(in=b);

   if a then do;
      SVSTDTC=VISDTC;
      SVENDTC=VISDTC;
   end;

   if b then do;
      SVSTDTC=LBDTC;
      SVENDTC=LBDTC;
   end;
run;
```

According to the game plan, evaluation question #3 (Dispensable IF-condition?) can apply to both logic blocks and they be replaced with this:

```
SVSTDTC= ifc(a,VISDTC,LBDTC);
SVENDTC= SVSTDTC;
```

Or others such as:

```
SVSTDTC= ifc(b,LBDTC,VISDTC);
SVENDTC= SVSTDTC;
```

It must be noted that the pair have a subtle caveat that each of their arguments are executed regardless the first-argument logit test. Two common examples are like here:

```
log_X  = ifn(x>0, log(x), .);
AEPTCD = ifc(^missing(AETERM_PT_CODE), substr(AETERM_PT_CODE,2),'');
```

In the case of x being -1 or AETERM_PT_CODE null, although correct results are derived in the final data there will be annoying warnings in the log like these:

```
NOTE: Mathematical operations could not be performed at the following
places. The results of the operations have been set to missing values.

NOTE: Invalid second argument to function SUBSTR at line 27 column 40.
AETERM_PT_CODE=  AEPTCD=  _ERROR_=1 _N_=1
```

In these situations, traditional IF-THEN logic may be used instead for the avoidance of the nuisance in the log.

Besides their usage in DATA steps, IFN/IFC have other utilization beyond the IF-THEN logic as follows:

- They can be used in PROC SQL;
- They can be embedded in each other in the second or third arguments;
- IFC can be conveniently used in macro variable definition.

For the last point, suppose that a macro variable (&mvar_1) is contingent on another macro variable (&mvar) being empty, the whole logic can be nicely compacted into a much shorter version as:

```
%let mvar_1=%sysfunc(ifc(&mvar=,something, something2));
```

WHICHC/WHICHN FUNCTIONS

The newish function pair came from version 9.2. Both functions search the second and subsequent arguments for a value equal to the first argument. Although WHICHC & WHICHN are used for character and numerical value targets, respectively, they always return a numerical value which is the index of the first matching value. Due to this nature, they are especially useful for converting a series of character/numeric values to corresponding numeric values.

Suppose VISITNUM is converted from VISIT, then a regular IF-THEN-ELSE construct may be as follows:

```
if visit='Screening' then visitnum= 1;
else if visit='Run-In Period' then visitnum= 2;
else if visit='Day 1' then visitnum= 3;
else if visit='Day 8' then visitnum= 4;
else if visit='Day 15' then visitnum= 5;
else if visit='Day 22' then visitnum= 6;
```

After the evaluation questions #2 (Outcome being single THEN-item?) and #3 (Dispensable IF-condition?) being passed, the function WHICHC can be used to replace the IF-THEN-ELSE logic with this:

```
visitnum=whichc(visit,'Screening','Run-In Period', 'Day 1', 'Day 8', 'Day
15','Day 22') ;
```

There come the advantages including the code being compact and computation overhead reduced, as well as the disadvantages including low readability and rigidly consecutive numerical order. For example, if the conversion includes that when VISIT is equal to string "Unscheduled" VISITNUM is to set at 14, the function alone cannot implement the conversion as above since there isn't a consecutive sequence anymore.

In sum, I personally would use WHICHC/WHICHN only for simple sequential conversion less than three items. If there are more than three items in the conversion sequence, no matter if it is consecutive or not, I would resume the evaluation questions in the game plan and settle down with a PROC FORMAT with the evaluation question #4 (Checking the Same Var?).

WHICH-CHOOSE FUNCTION COMBO

CHOOSEC/CHOOSEN function pair came from version 9. Exactly opposite to WHICHC/WHICHN, this function pair picked up one value from the second and subsequent arguments based on the numeric index value from the first argument. CHOOSEC & CHOOSEN are picking character & numeric targets respectively.

Due to their complimentary nature, in some cases, the WHICHC/WHICHN & CHOOSEC/CHOOSEN function pairs can be mixed and combined to implement regular IF-THEN logics. One example is to assign RACE character values based on other numeric variables:

```
if RACEWT=1 then RACE='WHITE';
else if RACEAIA=1 then RACE='AMERICAN INDIAN OR ALASKA NATIVE';
else if RACEA=1 then RACE='ASIAN';
else if RACEAA=1 then RACE='BLACK OR AFRICAN AMERICAN';
else if RACENH=1 then RACE='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER';
```

Here is the analysis: The evaluation question #2 (Outcome being single THEN-item?) in the game plan is a simple "Yes" and the evaluation question #3 (Dispensable IF-condition?) is applicable to this logic, which therefore can be replaced with WHICHN-CHOOSEC construct as follows:

```
    race = choosec(whichn(1,RACEWT,RACEAIA,RACEA,RACEAA,RACENH)
,'WHITE','AMERICAN,INDIAN,OR,ALASKA,NATIVE','ASIAN','BLACK,OR,AFRICAN,AMERI
CAN','NATIVE,HAWAIIAN,OR,OTHER,PACIFIC,ISLANDER');
```

WHICHN searches among the five variables and returns the index number of the first hit to CHOOSEC, and then CHOOSEC returns the corresponding character value on the indexed position, which is exactly what the IF-THEN- ELSE block does. However, the disadvantage is that with all these data values crammed into this statement that might span several code lines, the orders in both sequences have to be correctly aligned. If messed up, it might get tough to debug the statement. I would consider other options before adopting this replacement and one of them is PROC FORMAT in scenario #3.

COALESCE/COALESCEC

The COALESCE/COALESCEC function pair is used to select the first non-missing numerical and character value, respectively, in the order of the argument list. Based on this mechanism, it may be useful for conditional selection of variables. Suppose we need to set QSTPTNUM to the first non-missing value from SFTMP_CV, FACPOINT_CV, and TQOLTPT_CV:

```
qstptnum = coalescec(sftmp_cv,facpoint_cv,tqoltpt_cv);
```

Without this function, it would take several stacked lines of IF-THEN-ELSE statements.

COMPARISON OPERATORS

It is known that Boolean expressions such as comparison operators () return numeric values 1 or 0, which can be used for next step calculation. In the cases of numeric variable derivation and the condition being a YES/NO situation, comparison operators can be extremely useful to perform conditional logic. A simple example is for the numeric sex variable (SEXN) conversion where the male is being 1 while the female 0. The traditional IF-THEN logic would be as follows:

```
if sex='M' then sexn=1;
    else sexn=0;
```

Since both the evaluation questions #2 (Outcome being single THEN-item?) and #3 (Dispensable IF-condition?) turn affirmative, it can be handily replaced as follows:

```
sexn=(sex='M');
```

For more advanced examples, such as the study day of adverse event (AESTDY) being derived as the difference between the start time of adverse event (AESTDT) and the start time of medication exposure (EXSTDT), the tradition way with IF-THEN logic would be like this:

```
if aestdt >= exstdt then aestdy=aestdt-exstdt +1;
    else aestdy = aestdt-exstdt;
```

To replace it, Hortman solved it with this succinct statement:

```
aestdy = aestdt-exstdt +(aestdt >= exstdt);
```

For further illustration of their usage, included is another example by Carpenter where SEASON is assigned numerical values according to birthday month, i.e. SEASON being 1 when month 1-3, 2 when month 4-6, 3 when month 7-9, etc. It could be usually implemented with IF-THEN-ELSE logic like this:

```
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 9 le month(dob) le 12 then season = 4;
```

With the comparison operator, it would be creatively implemented as follows:

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

Although the overhead may not be significantly reduced with this replacement, the new statement is much better readable and easier to maintain.

In all, comparison operators are up for selection when numeric variables are derived, and the conditions are a YES/NO situation. The new implement is indeed succinct and powerful.

OTHERS

Depending on the specific environments, there might be other methods to remove the IF-THEN statements. For example, for INPUT function not to produce an error in the log, this may be needed,

```
if length(date)>=9 then var=input(date,date9.);
```

However, per the evaluation questions #2 (Outcome being single THEN-item?) and then #3 (Dispensable IF-condition?), the IF-THEN logic can be removed as follows,

```
var=input(date,??date9.);
```

The format modifier (??) removes the possible error messages from the log along with resetting the internal error code, rendering the IF-THEN logic optional. As a good practice, however, the results have to be checked and this modifier had better to use with other logics for dealing with the extreme data cases.

Due to development in SAS new releases, more and more such useful options may be released, and readers are encouraged to build their own toolbox in this regard and shake off the stereotype programming style.

SCENARIO #3 STRATEGY TAKEN: REPLACING WITH PROC FORMAT

It is well known that a binary searching algorithm is used for record-locating with PROC FORMAT, which is much faster than the sequential checking in the regular IF-THEN-ELSE construct, especially when the search sequence is lengthy. Additionally, the data values in PROC FORMAT can be aligned in a highly editable format and the PROC FORMAT itself listed anywhere before the corresponding DATA step, which make the program extremely easy to debug.

PROC FORMAT can handily deal with the 1-to-1 value matching as shown above. Additionally, it can easily deal with the N-to-1 situation. Suppose that we are assigning EPOCH values based on VISITNUM and there are instances of both situations. If implemented with a regular IF-THEN-ELSE logic, the program is as follows:

```
if      visitnum=1    then EPOCH='SCREENING';
else if visitnum=2    then EPOCH='RUN-IN';
else if 2<visitnum<=7 then EPOCH='TREATMENT';
else if visitnum=8    then EPOCH='FOLLOW-UP';
```

After the positive evaluation from the questions #2 (Outcome being single THEN-item?) & #4 (Checking the Same Var?), the logic can be replaced with this:

```
proc format;
    value epochf    1       ='SCREENING'
                    2       ='RUN-IN'
                    2<-7    ='TREATMENT'
                    8       ='FOLLOW-UP'
                    other   =' ';
run;
```

And in the later DATA step, this statement can be used to apply the format:

```
epoch   = put(visitnum,epochf.);
```

Sometime, the conversion with PROC FORMAT may demand a little analysis as in this example:

```
if PARAMCD = 'FPG' then var_2 = var_1 * 18;
else if PARAMCD = 'FPI' then var_2 = var_1 * 6.945;
else if PARAMCD in ('LDL' 'HDL') then var_2 = var_1 * 38.67;
else if PARAMCD = 'TRIG' then var_2 = var_1 * 88.57;
else var_2 = var_1;
```

Per the evaluation questions #2 (Outcome being single THEN-item?) & #3 (Dispensable IF-condition?), the IF-condition here is not dispensable since it is the case of multiple conditions. For the next evaluation question #4 (Checking the Same Var?), it is a good fit except that there are three variables involved (PARAMCD, VAR_2 & VAR_1), instead of two variables commonly dealt with in PROC FORMAT. However, the third variable (VAR_1) is consistent in the THEN-item. We can take advantage of this and the PROC FORMAT can run this way:

```
proc format;
    invalue convf   'FPG'       = 18
                    'FPI'       = 6.945
                    'LDL','HDL' = 38.67
                    'TRIG'      = 88.57
                    other       = 1;
run;
```

In later DATA steps, the above IF-THEN-ELSE block is replaced by this statement:

```
var_2=var_1*input(paramcd,convf.);
```

There is one example shown in Scenario #2 where the evaluation question #3 (Dispensable IF-condition?) is applied and the IF-THEN-ELSE logic is replaced with WHICHN-CHOOSEC functions pair:

```
if RACEWT=1 then RACE='WHITE';
else if RACEAIA=1 then RACE='AMERICAN INDIAN OR ALASKA NATIVE';
else if RACEA=1 then RACE='ASIAN';
else if RACEAA=1 then RACE='BLACK OR AFRICAN AMERICAN';
else if RACENH=1 then RACE='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER';
```

Although different variables are used in the IF-condition, the syntax is consistent and the same variable (RACE) is used in the THEN-item. Therefore, the evaluation question #4 (Checking the same var?) is also applicable to this logic and the format would be as follows:

```
proc format;
    value $racef    'RACEWT'  ='WHITE'
                    'RACEAIA' ='AMERICAN INDIAN OR ALASKA NATIVE'
                    'RACEA'   ='ASIAN'
                    'RACEAA'  ='BLACK OR AFRICAN AMERICAN'
                    'RACENH'  ='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER';
run;
```

Combined with the format, these statements in DATA step replaces the original IF-THEN-ELSE logic:

```
array _race  race:;
do over _race;
    if _race=1 then race = put(upcase(vname(_race)),$racef. -l);
end;
```

The major benefit for this case may be the improved readability instead and easy to debug since the data values are now in the PROC FORMAT which can be placed right after program headers, rather than buried deep in the code lines. Additionally, due to the fast processing of array and PROC FORMAT (binary searching algorithm), the overhead benefit over the original IF-THEN logic may be achieved. However, readers can execute their own judgement in choosing the best implementation they are comfortable with.

Please note that NOTSORTED option disables the binary searching algorithm used by the procedure. Therefore, to reduce the overhead of the IF-THEN logic, SAS users should refrain to resort to this option unless necessary.

SCENARIO #4 STRATEGY TAKEN: REPLACING WITH SELECT STATEMENT

Out of all these techniques, SELECT statement is the closest one to the traditional IF-THEN logic: Both use the sequential searching, the code lines are not standalone like PROC FORMAT, and they don't much differ on the overhead. However, better readability is one advantage of SELECT statement over the long sequence of IF-THEN-ELSE statements. You specify the name of the variable on the SELECT statement, followed by a sequence of nicely aligned WHEN statements. Each WHEN statement specifies a value for that variable. If the variable has that value, the program conditionally executes that statements.

Consider the following example with traditional IF-THEN-ELSE syntax. The code assigns AVISIT values based on AVISITN, VISIT & VISITNUM. All AVISIT values are directly based on AVISITN except the last two conditions:

```
if avisitn = -4 then avisit = 'Screening';
else if avisitn = 0 then avisit = 'Week 0';
else if avisitn = 4 then avisit = 'Week 4';
else if avisitn = 8 then avisit = 'Week 8';
```

```
    else if avisitn = 12 then avisit = 'Week 12';
    else if avisitn = 16 then avisit = 'Week 16';
    else if avisitn = 20 then avisit = 'Week 20';
    else if avisitn = 28 then avisit = 'Week 28';
    else if avisitn = 36 then avisit = 'Week 36';
    else if visitnum ne . & index(upcase(visit), 'UNSCHEDULED') > 0 then
avisit = "Unscheduled "||strip(put(visitnum, best.));
    else avisit=propcase(visit);
```

A quick analysis with the game plan discovered that

1. If there weren't the last two conditions, the evaluation question #4 (Checking the Same Var?) would apply well and the IF-THEN-ELSE block be easily replaced with PROC FORMAT. In this circumstance SELECT statement can be used instead, based on the positive evaluation from the question #5 (SELECT statement convertible?).
2. The last two conditions can be combined into one statement based on the evaluation question #3 (Dispensable IF-condition?) as above.

The new form will be as follows:

```
     select (avisitn);
       when -4  avisit = 'Screening';
           when 0   avisit = 'Week 0';
           when 4   avisit = 'Week 4';
           when 8   avisit = 'Week 8';
           when 12  avisit = 'Week 12';
           when 16  avisit = 'Week 16';
           when 20  avisit = 'Week 20';
           when 28  avisit = 'Week 28';
           when 36  avisit = 'Week 36';
           otherwise avisit= ifc(find(visit,'unscheduled','i'),catx('
',propcase(visit),visitnum), propcase(visit));
```

Clearly SELECT statement has dramatically improved readability while possessing the same power as the traditional IF-THEN-ELSE logic. Alternatively, based on the analysis above, another good solution is to combine PROC FORMAT with the traditional IF-THEN LOGIC, as follows:

```
proc format;
    value avisitf   -4 = 'Screening'
                     0 = 'Week 0'
                     4 = 'Week 4'
                     8 = 'Week 8'
                    12 = 'Week 12'
                    16 = 'Week 16'
                    20 = 'Week 20'
                    28 = 'Week 28'
                    36 = 'Week 36';
run;
```

In later DATA steps, these statements can replace the previous IF-THEN-ELSE block in this way:

```
if ^mod(avisitn,4) then avisit = put(avisitn,avisitf. -l);
    else avisit = ifc(find(visit,'unscheduled','i'),catx(' ',
propcase(visit), visitnum), propcase(visit));
```

If AVISITN has other numbers that cannot be evenly divided by 4, the IN keyword combined by the list of these numbers in the PROC FORMAT can be used to prevent the entry. Anyhow, this replacement has been taken care of both the overhead problem and the readability problem.

Lastly, please note for SELECT-WHEN statement:

- The most frequent condition should be put on the top of the WHEN statements for the purpose of efficiency.
- Multiple THEN-items can be enclosed with a DO-THEN loop appended to the WHEN statement. However, the do-loop-enclosed form may hurt the readability so badly that it probably eliminates the advantage of this statement. Certainly, the Evaluation question #2 (Outcome being single THEN-item?) has to be applied less strictly if this property is going to be used.
- Its counterpart in PROC SQL is CASE-WHEN clauses.

Of them, the first two properties are shared with the IF-THEN-ELSE logic.

SCENARIO #5 KEEPING THE IF-THEN LOGIC, CREATIVELY

Due to the complexity of the real-world problems, it is impossible or unnecessary to totally avoid the traditional IF-THEN logic. However, even in this situation, there are still something that can be improved to make the program creatively and less resource costly. Several suggestions are listed here.

EXAMPLE #1 KEEPING THE REGULAR CONSTRUCT: ADDING KEYWORD "ELSE"

If there are multiple IF-statements, keyword ELSE should not be lightly skipped in most circumstances. Otherwise, the following IF-statement will be executed, which might create different results or system resources are wasted, to say at least. Here is the example used for scenario #2 (ONLY USING THE FUNCTIONS/OPERATORS) but it is applicable here since the system resource can be saved:

```
data sv2;
   set sv1(in=a)
       lb1(in=b);

   if a then do;
      SVSTDTC=VISDTC;
      SVENDTC=VISDTC;
   end;

   if b then do;
      SVSTDTC=LBDTC;
      SVENDTC=LBDTC;
   end;
run;
```

It can be changed to this with the 2nd logic evaluation being removed:

```
data sv2;
   set sv1(in=a)
       lb1(in=b);

   if a then do;
      SVSTDTC=VISDTC;
```

```
      SVENDTC=VISDTC;
   end;
   else do;
      SVSTDTC=LBDTC;
      SVENDTC=LBDTC;
   end;
run;
```

Although only one evaluation is removed in this over-simplified example, it does show that keyword ELSE can be easily added to make the program more efficient. As previously mentioned, function IFC may be a better replacement due to its power and concise structure.

There is another example for me to reiterate the issue. Suppose that we derive AVISIT & AVISITN based on VISIT values. A program implementation is coded as follows:

```
   if visit='Visit 1 Screen'
      then do;
              AVISIT= 'Screening';
             AVISITN=-2;
           end;
   if visit='Visit 2 Run In'
      then do;
              AVISIT= 'Run-in';
             AVISITN=-1;
           end;
   if visit='Visit 3 Baseline'
      then do;
              AVISIT='Baseline';
              AVISITN=0;
        end;
   if visit='Visit 4'
      then do;
              AVISIT='Week 4';
              AVISITN=4;
        end;
```

There are the following observations:

1 The evaluation question #2 (Outcome Being Single THEN-item?) can be flexible for #4 (Checking the Same Var?) to apply. The strategy is to replace the IF-THEN series with two formats for AVISIT and AVISITN, respectively.

2 Even if the evaluation question #2 (Outcome Being Single THEN-item?) is rigidly applied and the regular IF-THEN logic is to be used, at least the keyword ELSE should be added to avoid repetitive checking on the same variable.

3 As in other cases, the most frequent condition should be listed as the first to check. A quick PROC FREQ combined with the correct check order can save you significant processing time.

EXAMPLE #2 KEEPING THE REGULAR CONSTRUCT: AVOIDING REPETITIVE EVALUATIONS

Repetitive evaluations are costly regarding the processing power and program maintenance. Fortunately, they can be avoided with little efforts. Suppose that we are assigning values to the flag variable ANL01FL in multiple situations. The original program was coded as follows:

```
    if (ABLFL='Y' or AVISIT in ('Screening' 'Run-in' 'Baseline' 'Week 4'
'Week 8' 'Week 12' 'Week 13' 'Week 16')) and
        PARAMCD not in ('CREAT' 'HCG' 'HCAB' 'HBSAG' 'HIV12AB')
                then ANL01FL='Y';

    else if last.AVISITN and
       (ABLFL='Y' or AVISIT in ('Screening' 'Run-in' 'Baseline' 'Week 4'
'Week 8' 'Week 12' 'Week 13' 'Week 16')) and
        PARAMCD in ('CREAT')
                then ANL01FL='Y';

    if ABLFL = 'Y' then ANL01FL = 'Y';
```

Here the evaluation flow will pass through questions #1-#5 and the regular IF-THEN logic be kept. However, there are two evaluations (ABLFL='Y', and AVISIT in ()) repeated twice or three times. During execution, these evaluations are tested on every record since they are operands with AND/OR operators. After combination, the program may be in this way:

```
    if ABLFL = 'Y' then ANL01FL = 'Y';
        else if AVISIT in ('Screening' 'Run-in' 'Baseline' 'Week 4' 'Week
8' 'Week 12' 'Week 13' 'Week 16') then
            do;
                if PARAMCD not in ('CREAT' 'HCG' 'HCAB' 'HBSAG' 'HIV12AB')
| (PARAMCD in ('CREAT') & last.AVISITN) then ANL01FL = 'Y';
            end;
```

Here the evaluations appear only once. Further, they are separated with the ELSE keyword therefore some of them are not executed for the inapplicable data, which saves valuable resources and expedites the process. Additionally, it is easier to maintain and debug.

Another example is listed to stress the issue, as follows:

```
    if aestdtc < rfstdtc and rfstdtc ne "" then
            AEENRF="BEFORE";
        else if (rfstdtc <=  aestdtc <= rfendtc) and rfstdtc ne "" then
            AEENRF="DURING/AFTER";
        else if aestdtc > rfendtc then
            AEENRF="AFTER";
```

Similarly, the evaluation flow will pass through questions #1-#5 and the regular IF-THEN logic be used. However, this evaluation (rfstdtc ne "") being repeated. The new form is solving the issue as follows:

```
if aestdtc > rfendtc then AEENRF="AFTER";

    if ^missing(rfstdtc) then
        do;
            if aestdtc < rfstdtc then
                        AEENRF="BEFORE";
                else if (rfstdtc <=  aestdtc <= rfendtc)  then
                        AEENRF="DURING/AFTER";
        end;
```

Please note the evaluation (AESTDTC > RFENDTC) was moved forward in order to get the same results, and the combined form is logically equivalent to the previous form, while the process is streamlined and easier to follow.

EXAMPLE #3 KEEPING THE REGULAR CONSTRUCT: STILL YOU CAN DO

SOMETHING

Suppose we have a SDTM specification where column SOURCE lists the conditions and other three columns QSCAT, QSTESTCD & QSTEST the outcomes, as in Table 1.

SOURCE	QSCAT	QSTESTCD	QSTEST
if SFPERF_CV='N'.	Set to "SF-36 V2.0 STANDARD"	"SF36ALL"	"SF-36 Questionnaire"
if SFPERF_CV="Y" and original QSTESTCD="SF3601".	Set to "SF-36 V2.0 STANDARD"	"SF3601"	"SF-36 Question 1"
if SFPERF_CV="Y" and original QSTESTCD="SF3602".	Set to "SF-36 V2.0 STANDARD"	"SF3602"	"SF-36 Question 2"
if SFPERF_CV="Y" and original QSTESTCD="SF3603".	Set to "SF-36 V2.0 STANDARD"	"SF3603A"	"SF-36 Question 3A"
if SFPERF_CV="Y" and original QSTESTCD="SF3604".	Set to "SF-36 V2.0 STANDARD"	"SF3603B"	"SF-36 Question 3B"
if SFPERF_CV="Y" and original QSTESTCD="SF3605".	Set to "SF-36 V2.0 STANDARD"	"SF3603C"	"SF-36 Question 3C"
if SFPERF_CV="Y" and original QSTESTCD="SF3606".	Set to "SF-36 V2.0 STANDARD"	"SF3603D"	"SF-36 Question 3D"
if SFPERF_CV="Y" and original QSTESTCD="SF3607".	Set to "SF-36 V2.0 STANDARD"	"SF3603E"	"SF-36 Question 3E"

Table 1. Partial Mapping Specifications for SDTM QS Domain.

According to the negative evaluation from the question #2 (Outcome Being Single THEN-item?), the regular IF-THEN construct can be used. Even though, in SOURCE columns, the syntax is consistent mostly and only the value varies after the phrase "original QSTESTCD=". A final strategy taken is the combination of PROC FORMAT and the IF-THEN-ELSE construct which is for the first sub-condition "SFPERF_CV=". For the purpose for efficient and accurate input, initial import was taken with a PROC IMPORT (not shown), followed by a DATA step for parsing the extracted values such as "SF3601" and PROC FORMAT with the CNTLIN option for creating a format. The partial program for QSTEST format $testf. is as follows:

```
**FOR PARSING THE VALUES EXTRACTED FROM THE SPEC;
data test(drop=qstestcd rename=(qstest=label));

    set spec_import;
```

```
    **REMOVING QUOTATION MARKS;
    qstest  = compress(qstest,"-" , "adkis");

    **EXTRACTING & CLEANING THE THIRD DATA VALUES IN SOURCE;
    source = compress(scan(source,3,'='),'adki');

    rename source=start;
run;

**FOR PREPARING DATA FOR PROC FORMAT;
data test;
    retain fmtname 'testf' type 'c';

    do until (eof);
        set test end=eof;
       output;
    end;

    call missing(start);
    hlo='o';
    label=' ';
    output;
    stop;
run;

proc format cntlin=test;
run;
```

Similar programs are not shown for creating format $testcdf. for QSTESTCD. In later DATA steps, the regular long sequence of IF-THEN-ELSE logic can be replaced with these statements as follows:

```
    if sfperf_cv='N' then
            do;
                qscat       = "SF-36 V2.0 STANDARD";
                qstestcd    = "SF36ALL";
                qstest      = "SF-36 Questionnaire";
            end;
        else if sfperf_cv='Y' then
            do;
                qscat       = "SF-36 V2.0 STANDARD";
                qstest      = put(qstestcd,$testf. -l);
                qstestcd    = put(qstestcd,$testcdf. -l);
            end;
```

Although the program seems a little longer than the traditional IF-THEN logic built from the fetched data values, it is undeniably more robust due to that it is completely driven by data in the SDTM specification and no further maintenance effort is required upon later modifications from the specification. It is one nicety of the art of programming after all, isn't it?

CONCLUSION

There is computational benefit in many circumstances to replace the tradition IF-THEN logic in SAS DATA steps and there are multiple ways to do it. The current article sets up a straightforward game plan which primarily has five (5) short evaluation questions. When

the game plan is simply put in place, SAS users can do an immediate triage on the task at hand and intuitively come up with creative solutions with much reduced overhead. Even in the situations where the regular IF-THEN logic has to be employed, several practical principles are given and in the end the SAS programs are more efficient and pleasant to debug.

REFERENCES

Carpenter, Art. 2012. *Carpenter's Guide to Innovative SAS Techniques*,1st ed. Cary, NC: SAS Institute.

Dorfman Paul M. & Vyverman Koen. 2009. **"***The DOW-Loop Unrolled***."** *Proceeding of SAS Global Forum*, 038-2009. Cary, NC: SAS Institute Inc.

Available at http://support.sas.com/resources/papers/proceedings09/038-2009.pdf

Horstman, Joshua M. 2017. "*Beyond IF THEN ELSE: Techniques for Conditional Execution of SAS® Code." Proceeding of the 25th Annual SouthEast SAS Users Group (SESUG) Conference*, 326-2017. Cary, NC: SAS Institute Inc.

Available at https://support.sas.com/resources/papers/proceedings17/0326-2017.pdf

ACKNOWLEDGMENTS

The author is indebted to Ken Borowiak & Chad Mcghee for insightful comments.

RECOMMENDED READING

Cody, Ron. 2010. *SAS Functions by Example*, 2nd ed. Cary, NC: SAS Institute.

CONTACT INFORMATION

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

Jason J. Su
PPD, Inc.
3900 Paramount Parkway
Morrisville, NC 27560-7200
(919) 456 5717
jason.su@ppdi.com