SAS® GLOBAL FORUM 2016

IMAGINE. CREATE. INNOVATE.

# The SAS® Test from H*ll

How Well do you REALLY know SAS?

#SASGF

# The **SAS ®** Test from H*LL

Glen Becker

USAA

## ABSTRACT ICLICK TO EDIT)

- **This SAS ® Test is a fun way to expand your SAS programming skills. It has ## questions to challenge even the most experienced SAS programmers, but beginners will be able to tackle many of them.**
- **Most questions are about the DATA step, but there are a few macro and SAS/Access ® questions thrown in.**
- **This presents each question in black, then after you click, it presents the answers in White.**
- **If in doubt, fire up SAS and try these yourself.**
- **You WILL be challenged, and maybe earn valuable bragging rights!**
- **Disclaimer: the question numbers are a little different than the version in the Proceedings.**
- **Good Luck!**

## The **SAS ®** Test from H*LL

Question 1: Explain the differences between the behavior of these three DATA step statements:

```
A = A + B;
A = SUM(A, B);
A + B;
```

Answer: All three add the value of B to the current value of A. Differences:

A = A + B; leaves A missing if either A or B is missing.

A = SUM(A,B); leaves A missing only if both A and B are missing.

A + B; same as A=SUM(A,B) but implicitly retains A.

## Question 2

Question 2: Please rewrite the following DATA step code to reduce duplicate code and eliminate unnecessary processing:

```
IF LoanAmt>525000 and state in('AK','HI','VI')  then Type='Jumbo';
else if LoanAmt>472000 and state not in('AK','HI','VI')
then Type='Jumbo';
else Type='Conv';
```

There are obviously several solutions to this. All of these run faster than the code above:

Here is a straightforward one that avoids all unnecessary processing, but allows trivial duplicate code:

```
IF state in ('AK','HI','VI')
then if LoanAmt > 525000 then Type='Jumbo';
                         else Type='Conv';
else if LoanAmt > 472000 then Type='Jumbo';
                         else Type='Conv';
```

Here is a more elegant one that does better at avoiding duplicate code, but is a little slower and harder to read:

```
IF LoanAmt > Ifn(state in('AK','HI','VI'),525000,472000) then Type='Jumbo';
                                                         else Type='Conv';
```

This is in SAS V8 dialect where IFN() was not available, and is as fast as the first solution, because it avoids the overhead of a function call. It is a little harder to read, but illustrates using logical conditions in mathematical expressions.

```
IF LoanAmt > 472000 + (525000-427000)*(state in('AK','HI','VI')) then Type='Jumbo';
                                                                  else Type='Conv';
```

# The **SAS ®** Test from H*LL

Glen Becker

USAA

## Questions 3 & 4

3. Identify the likely error in the following statement. Describe the statement's actual behavior in detail. Assume this is the only INPUT statement, and it contains the first mention of each variable in the DATA step.

**INPUT ID $8. DATE DATE9. CODE $7 VALUE 14.2;**

- ID will be read as character from columns 1-8.
- Date will be read as DDMONYYYY from columns 9-17.
- Code will be read as character from column 7, only, but defined with a length of 8 bytes. This is the likely error; the author probably meant **$7.** formatted input, but forgot the "dot" that indicates a format. This subtle error causes the variable to be read as column input, not formatted input.
- Value will be read as numeric from columns 8-21, with 2 places implied behind the decimal. Because columns 11-13 are probably a three-letter month, all reads will probably fail.

4. What is meant by the LENGTH of a numeric variable, and how does it affect the precision of numeric calculations? What restriction does SAS place on the placement of such a LENGTH statement in a DATA step?

The length of a numeric variable is the number of bytes used to **store** the variable when it is written to disk. It has **no** influence on the precision of calculations, but instead can compromise the precision recovered when the data is re-read from disk.

There are **no** restrictions of where a LENGTH statement is placed in a DATA step. (If you answered that the LENGTH statement must precede the SET or MERGE statement, you were thinking of character variables. Numeric is different.)

## Questions 5 & 6

5. Please describe the results of running the following DATA step:

```
DATA NEW;
  SET OLD(KEEP=NUM);
  IF NUM = 4 THEN CODE = 'X';
  ELSE IF NUM = 8 THEN CODE = 'X2';
  ELSE CODE = 'XMOD';
  KEEP CODE;
RUN;
```

The output dataset will have one variable named CODE, whose length is **one** character, **all** obs will have the value **X**. The trailing portions of X2 and XMOD are lost in truncation. There is no chance that CODE is defined as a longer variable by the SET statement, because it specifies a KEEP= dataset option.

6. Please describe the contents of TEXT1 and TEXT2 in the code below:

```
DATA NEW;
  RETAIN PREFIX 'ABC:' SUFFIX 123;
  LENGTH STRING1 STRING2 $10;
  STRING1=CAT(PREFIX, SUFFIX);
  STRING1=RIGHT(STRING1);
  STRING2=RIGHT(CAT(PREFIX, SUFFIX));
RUN;
```

String1 contains "ABC:123" with 3 leading blanks. String2 contains 3 trailing blanks, because CAT() returned only 7 characters, which were already right-justified. Assigning them to a 10-byte variable adds 3 trailing blanks.

## Questions 7 & 8

7. The following lines of DATA-step SAS code were written by an expert SAS programmer. Why did he call the CAT function? What is the practical purpose for passing the SCAN result through the CAT function instead of just assigning it directly to VARIABLE?

```
LENGTH VARIABLE $10;
VARIABLE = CAT(SCAN(LINE,4));
```

The programmer wants to know whether the SCAN result fits in the defined length of VARIABLE. The CAT function issues a SAS warning and sets _ERROR_ if not. Just assigning the SCAN results to VARIABLE truncates without warning.

8. Identify the likely error in the following statement. Describe the statement's actual behavior in detail. Assume this is the only INPUT statement, and it contains the first mention of each variable in the DATA step.

```
INPUT ID $8. DATE DATE9. CODE $7 VALUE 14.2;
```

ID will be read as character from columns 1-8.

Date will be read as DDMONYYYY from columns 9-17.

Code will be read as character from column 7, only, leaving the input pointer at column 8. CODE will be defined as 8 bytes. This is the likely error; the author probably intended **$7.** for formatted input, but forgot the "dot" that indicates a format. This subtle error causes the variable to be read as column input, not formatted input, and causes the next variable to be read from the wrong place.

Value will be read as numeric from columns 8-21, with 2 places implied behind the decimal. Because columns 11-13 are probably a three-letter month, all reads will probably fail and result in errors.

# The **SAS ®** Test from H*LL

Glen Becker

USAA

## Questions 9 & 10

9. Please describe the results of running this DATA step.  NUM is numeric.

```
  DATA NEW;
    SET OLD(KEEP=NUM) END=LASTOBS;
    RETAIN MIN MAX;
    IF NUM < MIN THEN MIN = NUM;
    IF NUM > MAX THEN MAX = NUM;
    IF LASTOBS;
  RUN;
```

It would seem that NEW would contain one observation with NUM being the input value from the last observation, MIN having the smallest value of NUM, and MAX the largest.   But, the RETAIN statement implicitly initializes both MIN and MAX to missing.   Because missing values are smaller than all real values, NUM is never less than MIN, so MIN stays missing!

Give yourself extra credit for identifying the sole exception:   if NUM ever has the SAS special missing value ._ (dot-underscore) MIN will be ._ because ._ is less than the normal missing value .   Period.

10. Please describe the difference in behavior between the following statements:

```
  DO COUNTER=1 TO LIMIT;

  DO COUNTER=1 BY 1 WHILE(COUNTER <= LIMIT);
```

Both statements begin iterative loops that use COUNTER as the counting variable and a compare COUNTER with LIMIT to terminate the loop . The first statement obtains the value of LIMIT before the loop runs the first time, and is insensitive to changes that may occur to LIMIT while the loop runs.   The second statement compares COUNTER with the then-current value of LIMIT before each iteration, and therefore is affected if the loop changes LIMIT.

So: The second form allows a loop to extend itself as it runs, the first does not.

## Questions 11 & 12

11. Please describe the behavior of the following DATA step statement:

```
        A = B = 2;
```

This separates SAS programmers from C, C++, and Java programmers who have dabbled in SAS.   In those languages, this statement is a common idiom to assign the value 2 to variables A and B.   In SAS, it is an ordinary assignment statement that evaluates the expression B = 2, which tests whether B is the number 2.  If so, A is one, if not A is zero.

It is perfectly legal SAS, so if you said it is an error, that's an error.

12. Explain the practical difference between these two DATA steps, if any. As the name implies, VERY.BIGFILE is a very big file.

```
DATA EXRACT1;
  SET VERY.BIGFILE;
  KEEP VAR1-VAR4 CODE1-CODE6;
RUN;

DATA EXRACT2;
  SET VERY.BIGFILE(KEEP=VAR1-VAR4 CODE1-CODE6);
RUN;
```

This is a performance question.  The output of the two will be identical.  The only difference is that the first reads all variables into the DATA step, the second reads only the selected variables.   If VERY is a SAS disk library on the local machine, the performance difference may be small, but if it uses SAS/Access to read a database, or uses the REMOTE engine (via SAS/Connect), the performance difference can be HUGE.

## Question 13

13. A junior SAS programmer shows you the following program, and says, "It does not work right."  Please explain what corrections are needed.

```
/* Keep a BY group if its first obs has a higher VALUE than
VALUE from the last obs of the previous BY group */
DATA ABC;
  SET XYZ;  BY GROUP;
  IF FIRST.GROUP THEN DO;
     IF VALUE > LAG(VALUE) THEN HIGHER='Y';
  END;
  IF HIGHER='Y'; DROP HIGHER; /* keep group if HIGHER=Y */
RUN;
```

The junior programmer got one thing right:  he must save the result of the test from the first obs of each by-group, then use that result to keep or delete the entire following by-group.

Three things wrong:
1. The programmer forgot to RETAIN HIGHER;   It gets reset to missing each obs.
2. The programmer forgot to reset HIGHER to not be Y if the test failed.
3. LAG(VALUE) returns VALUE as it was the last time LAG(VALUE)  was called, not as it existed in the previous observation.

Here is the corrected code:

```
DATA ABC;
  SET XYZ;  BY GROUP;
  RETAIN HIGHER;
  LAG_VALUE=LAG(VALUE);  /* LAG() unconditional */
  IF FIRST.GROUP AND VALUE>LAG_VALUE THEN HIGHER='Y';
                                     ELSE HIGHER=' ';
  IF HIGHER='Y';  DROP HIGHER LAG_VALUE;
RUN;
```

# The **SAS ®** Test from H*LL

Glen Becker

USAA

## Question 14

In a DATA step, SAS automatically sets all variables to missing at the beginning of each iteration of its implicit loop unless you request that it not do so. Please list **ALL** of the ways to arrange that a variable not be reset to missing. (There are six of them).

1. A **RETAIN** statement that mentions the variable, or just **RETAIN;** to implicitly retain all variables
2. An accumulation statement: **VAR + 1;**
3. The variable exists on any input dataset read with SET, MERGE or UPDATE. SAS assumes that reading the variable over-writes it, so SAS does not set it to missing, even if the statement is conditional. Thus, the following statement retains VAR: **IF 0 THEN SET ABC(KEEP=VAR);**
4. Elements of temporary arrays are not set to missing: **ARRAY ABC (4) _TEMPORARY_;**
5. SAS temporary variables are usually implicitly retained, such those produced by SAS statement options like IN=, NOBS=, etc.
6. Make up your own "automatic" variables: FIRST.ANYTHING, even if the DATA step does not use ANYTHING as a BY-variable. Yes, this works!

## Question 15

One SAS programmer says "The SYMGET() function is completely unnecessary. To use the value of a SAS macro variable in a DATA step, just invoke the variable directly, like **SUM = SAS_VAR + &MACRO_VAR;**"

Another SAS programmer, says, "Oh, no! That doesn't always work. You need to write it this way: **SUM = SAS_VAR + SYMGET('MACRO_VAR');** "

Who is right? Do you ever need to use SYMGET() instead of just invoking a macro variable directly?

In general, it is OK to use &Macro_var directly. The only time that you would need to use SYMGET() is if the macro variable could change while the DATA step runs.

This can happen because the same data step did a CALL SYMPUT, or invoked a macro via CALL EXECUTE that may have changed the macro variable.

Or you could have started an asynchronous task with the SYSTASK statement: SYSTASK COMMAND "…" NOWAIT STATUS=STATVAR; Then, a DATA step could periodically check the variable with SYMGET("STATVAR") to see whether the task had finished. Same applies to asynchronous RSUBMIT if you use SAS/Connect.

## Question 16

What are the differences between the NODUP, NODUPKEY and NODUPRECS keywords of PROC SORT?

What conditions would cause PROC SORT with any of these keywords to create sorted output that contains duplicate rows?

NODUPKEY eliminates all obs with identical keys, no possibility of duplicate rows.

NODUPS is the same as NODUPRECS, and eliminates identical rows **provided** the sort places them physically together. Here is an example of how NODUPS can produce duplicates:

```
data x;
  input name $ num;
  cards;  /* I am showing my age */
Joe 34
Sam 48
Joe 38
Joe 34
run;
proc sort nodups data=x out=y;
  by name;
run;
```

Results:

```
Joe 34
Joe 38
Joe 34   ← Not eliminated because SORT did not move it next to 1st obs
Sam 48
```

# The **SAS ®** Test from H*LL

Glen Becker

USAA

## Question 17

The computer department periodically creates a very large SAS dataset in a protected library each month for analysts to read. Performance was fine, until they added an index to the dataset, figuring that an index may help read performance, but cannot hurt anything, since the dataset is never updated.

Now, they are deluged with complaints of horrible performance by analysts that read the dataset using programs that had previously run well. What went wrong, and how can they correct the problem without removing the index?

The most likely explanation is that the dataset is sorted by the newly-indexed variable(s), has no sort assertion, and analysts are reading the dataset with BY statements that name the same variable(s).

SAS has a 3-level hierarchy to process data with a BY statement:

1. If the BY variables match the known sort order, SAS reads sequentially.
2. If there is an index on those variables, SAS uses the index to get the data in the desired order.
3. If neither, SAS reads sequentially and hopes the data is in the correct order.

Many SAS programmers grew up assuming #3 is the only way, because older SAS releases had neither indices nor recorded sort orders.

Up to now, analyst were likely falling to #3 in the hierarchy and reading sequentially. But, the presence of the index caused SAS to use the index to obtain each observation, with a huge performance cost. The solution to the problem is to use the SORTEDBY dataset option to add a correct sort assertion, so that SAS will read sequentially per choice #1 above. This can be done after-the fact:

```
PROC DATASETS LIB=ANYLIB NOLIST;
   MODIFY ANYMEMBER(SORTEDBY=KEYVAR ...);
QUIT;
```

## Question 18

You need to count the number of observations of a dataset in a V9 SAS library for each unique combination of these four variables: numeric ZIPCODE and ACTIVITY_DATE, and character PURCHASE_TYPE and CUSTOMER_CODE both having lengths of 16 chars. Each of these four variables has between 1,000 and 10,000 possible values, the input dataset has 250 million observations, and you expect to find 4 to 8 million unique combinations. What is the best way to do this and why?

This is a test of performance smarts. PROC SUMMARY or PROC MEANS is the best way to count rows at this scale. Why? The internal "length" of the result is 64 bytes (4 numeric vars @ 8 bytes each + 2 char vars @ 16 each), and 64*8 million is 640 million bytes of answer, which is small enough to fit in memory, even on 32-bit machines. PROC SUMMARY can thus summarize this monster without resorting to spill files.

A do-it-yourself count with a DATA step using the hash object is OK; that is essentially what PROC SUMMARY is doing under the covers.

The most common wrong answer would be PROC FREQ, because it is the most obvious way to count rows. However, PROC FREQ begins by building a matrix of all possible values of the TABLE variables, which cannot fit in memory, because even at the low of the range of unique values: 4^1000 is 1 trillion. Failure is certain.

Any answer that depends on PROC SORT is wrong, because sorting such a huge dataset is impractical. Similarly SELECT COUNT(*), … GROUP BY … is not acceptable, because SQL also does a sort under the covers.

More elaborate methods of breaking the problem up into segments would work, but would consume far more resources and take lots longer an a simple PROC SUMMARY. Same applies to asynchronous RSUBMIT if you use SAS/Connect.

## Question 19

The following code is found in dozens of programs at XYZ company:

```
DATA _NULL_;
  RANDOM = RANUNI(0);
  CALL SYMPUT('RANDOM', PUT(RANDOM, 8.6));
RUN;
```

The boss hated having so much repeated code, he told his newly-hired SAS programmer to make it into a SAS macro. The programmer added this file to the company's standard macro library and the boss told everyone to use it:

```
%macro random(seed=0);
  DATA _NULL_;
    RANDOM = RANUNI(&seed);
    CALL SYMPUT('RANDOM', PUT(RANDOM, 8.6));
  RUN;
%mend;
```

So, everyone deleted the DATA step and replaced it with %RANDOM; Then many people complained that the new macro did not work, but others reported that it worked fine. Why does it work in some programs and not others? Assume all programs invoke this %RANDOM macro from the standard macro library.

The macro author forgot to **%global RANDOM;** If RANDOM already was a SAS global macro variable, the macro works fine. But it if is not, the variable is automatically localized into the local symbol table, and lost when the macro completes.

Give yourself extra credit for this: The macro has an entry in its symbol table because of the seed=0 keyword parameter. Otherwise it would have an empty symbol table, and CALL SYMPUT would have implicitly written RANDOM to the global symbol table.

# The **SAS** ® Test from H*LL

Glen Becker

USAA

## Question 20

A secretive SAS programmer did not know about the SECURE option of the %Macro statement, so he wrote a macro that tries to hide its behavior by having unconditionally issuing the following statement:

```
Options nonotes noMprint noMlogic noSymbolgen;
```

A curious SAS programmer cannot locate the macro's source code, but wants to see the macro's behavior by somehow disabling that OPTIONS statement. How could he do that?

```
Options Implmac;
%macro options / stmt pbuff;
   %put program executed: Options &syspbuff;
%mend;
```

Give yourself partial credit for the old-timers solution:

```
Options Macrogen;
```

Afterwards, how can he reinstate the OPTIONS statement to its normal behavior?

The easy way: `Option noImplmac;`

The elegant way:

```
PROC CATALOG CAT=WORK.SASMACR;  ← Work.SASMAC1 in SAS/EG
   DELETE OPTIONS.MACRO;
Quit;
```

## Question 21

You run the following SAS program:

```
PROC SQL;
  CREATE TABLE EXTR AS SELECT * FROM ABC.XYZ
  ORDER BY LAST_NAME;
QUIT;

DATA EXTRACT;
  SET EXTR;
  BY LAST_NAME;
  etc…
RUN;
```

You get the following error:

**ERROR: BY variables are not properly sorted on data set WORK.EXTR.**

Please explain why.

Library ABC must use a SAS/Access engine, either for SQL/Server or Oracle. PROC SQL passed the ORDER BY statement to the database, which sorted LAST_NAME using the database's native sort order, which is not the same as the strict EBCIDIC or ASCII order used by SAS. A conflict can easily happen with people's last names, because they often contain hyphens, which the two sort orders treat differently. DB2 does not commit such crimes.

## Question 22

You are writing a macro to complete a task that needs communication with another SAS server. Your company has a standard that all SAS/Connect sessions to that server are named OHIO, but not every SAS session establishes the connection. If no connection exists, your macro must both SIGNON and SIGNOFF to OHIO. Otherwise, you must use the existing connection and most importantly, not SIGNOFF.

How do you silently test whether the connection exists? The macro must not generate any error messages while trying to use a non-existent connection.

Inside the macro…

```
/* Try to create a local LIBREF to OHIO's WORK lib */
%local Pre_existing_session;
%if %sysfunc(libname(RXWORK,,REMOTE,
             SERVER=OHIO SLIBREF=WORK)) %then %do;
    %let Pre_existing_session=0;
    … statements to SIGNON …
%end;
%else %do;  /* Zero return meant LIBNAME worked */
    %if %sysfunc(libname(RXWORK)) %then; /*free RXWORK */
    %let Pre_existing_session=1;
%end;
```

… at the end of the macro …

```
%if not &Pre_existing_session %then SIGNOFF;;
```

# The **SAS** ® Test from H*LL

Glen Becker

USAA

## Question 23

A junior programmer was asked to write a program that reads text data into SAS. Each line of input should result in an output observation, with each word in a different variable: WORD1, WORD2, etc. depending on the number of words in the line. He wrote the following macro. You may assume that TEXTFILE is a valid SAS fileref that refers to the correct text file.

```
%macro readwords;
data words;
  infile textfile;
  %do w=1 %to 999;
      input word&w $ @;
      if word&w = ' ' then stop;
  %end;
run;
%mend;
```

This programmer came to you for help. He explained that he was trying to have the macro's %DO loop only run as many times as required to read all the words in the line, but it just wasn't working. Also, some of his variables are truncated. He would like to have each WORDn variable automatically be large enough for the contents of the nth word, but cannot figure out how to do this.

Thoroughly explain how this program behaves as it is currently written, and what results to expect. We already know it does not produce the desired result, but exactly what does it do? Be specific about the structure and contents of the output dataset.

## Question 23, answer

The INFILE statement defaulted to FLOWOVER. So each INPUT statement will automatically go onto the next input line if there are no more words in the current line. The test for word&w being blank will never be true as long as there are more lines in the input file.

So the first output observation contains the first 999 words of the file as variables WORD1-WORD999, without regard to the number of input lines it read. But, because the INPUT statement ends with @ and not @@, SAS stops reading the current line after the 999th word, even though that line may contain more words.

The next iteration of the DATA step will continue at the beginning of the next line of input, reading enough new lines to fill out 999 variables, and so on. All obs will have 999 non-blank words. No exceptions.

The last partial block of words will never be written to the output dataset, because the STOP statement ends the DATA step prematurely.

All variables, WORD1-WORD999 will have a length of 8 characters, because that is the default length of character variables defined by an INPUT statement that provides no informat or column range.

## Question 24

From the previous problem, explain how to accomplish the goal of having the output dataset contain only the correct number of WORDn variables, and each one correctly sized for the longest length of the actual words. You do not have to write the actual program, just explain how it could be done.

The only way to do this is to pass the data twice. Any solution that tries to accomplish the goal in a single DATA step is **WRONG!**

One approach: The first pass counts the maximum number of words in any input line, and the longest length of the Nth word of each line. This information is passed via macro variables, to become part of a LENGTH statement in the DATA step that actually reads the data.

Another approach: Read the data into a SAS dataset that defines a very large number of WORDn variables with very long lengths. Then derive the number of actual variables required, and longest lengths of each WORDn variable, and pass the information via macro vars to another DATA step, whose LENGTH statement truncates the data.

# The **SAS** ® Test from H*LL

Glen Becker

USAA

## Question 25

Given a SAS dataset with several million observations, and only SAS/Base (no PROC SURVEYSELECT),

How do you randomly select ¼ of the records?

```
DATA MY.SMALLFILE;
  SET REALLY.BIGFILE;
  IF UNIFORM(0) < 0.25;  /* or RANUNI(0) */
RUN;
```

How do you randomly select 100 records, with reasonable performance?

```
DATA MY.SMALLFILE;
  P=INT(NOBS*Uniform(0));  /* NOBS set by compiler */
  SET REALLY.BIGFILE POINT=P NOBS=NOBS;
  IF _N_ > 100 THEN STOP;
RUN;
```

Nit: no provision to avoid duplicates, nor to retrieve the data in sorted order. Duplicates are unlikely (and statistically desirable), and sorting 100 records later is trivial. The STOP statement is essential; otherwise, the program goes into an infinite loop.

The following answers or their equivalents are nice tries, but wrong:

```
SET REALLY.BIGFILE(OBS=100);
```
← Reads first 100 records, only

```
SET REALLY.BIGFILE NOBS=NOBS;
IF  0=MOD(_N_, NOBS/100);
```
← Reads 100 evenly-spaced records;

## How Did You Do?

Sorry, this is a PowerPoint® presentation, so one must not expect it to be able to tally the answer….

… or do anything useful for that matter.

But, in a future world, it could tell how well you did from the expression on your face.

I hope you did well, and learned a few things on the way!

## This column intentionally left blank

SAS® GLOBAL FORUM 2016

IMAGINE. CREATE. INNOVATE.

LAS VEGAS | APRIL 18-21

#SASGF