ABSTRACT

When producing a statistics table where one or a set of descriptive statistics repeat across rows and columns on a page, much time and code is spent organizing results from a PROC step into the desired presentation order and format. A macro is typically not considered because differences such as the number of columns, one or many class variables, and subtotal columns and rows can make these tables look very different. Nonetheless, an easy to use and capable macro that produces such layouts saves valuable time and effort. This paper summarizes the construction and use of a macro that requires minimum user input yet produces full feature tables. Examples include summary statistics (N, Mean, SD, etc.) and frequency tables such as lab, change from baseline, shift, adverse event, and tables that need dummy rows.

INTRODUCTION

Tables that summarize numeric or character data from multiple populations, parameters, and time points offer an easy way to spot differences that may be of interest to an investigator. The descriptive statistics those tables request tend to be either summary (N, Mean, Standard Deviation, etc.) or frequency based, but the layout for each table depend on the number of populations and parameters. For programmers, producing the results can be done with a PROC MEANS/FREQ accompanied by a DATA STEP or two, but organizing the results into the finished product can prove time consuming. For clinical programmers that produce such tables on a daily basis, constructing a macro that produces not only the results, but in the layout desired could be worth pursuing.

INPUT DATASET FORMAT

The first decision to make before starting is the type of input dataset as the macro must be designed around datasets available to the user. The structure of the datasets, specifically long (normalized) or wide (denormalized) format, impacts the design of input parameters. From a clinical programming point of view, a wide dataset has one observation per subject, with analysis results stored in numerous variables, whereas a long dataset stores only one result per observation with other variables classifying the origin of this result. There are also datasets that contain both long and wide elements. For the author, choosing the long format was an easy choice due to the adoption of CDISC standards in the pharmaceutical industry leading to non-demographic SDTM and ADaM datasets taking the long format. It also doesn't hurt that classification variables in long datasets are useful for specifying the order of presentation in a table.

CONTROLLING OUTPUT

Rather than trying to abstract the design of inputs, examples show the outputs the author needed to accommodate and the parameter input design that resulted. Short excerpts of input datasets in addition to tables are provided.
In this first example, the macro call reads:

From the dataset ADLB, perform analysis on the result variable AVAL. Create columns by taking observations with TRT01A="A" for the first column, "B" for the second, and both "A" and "B" for the third. The presentation hierarchy is LBCAT at the highest, PARAM second, and AVISIT last. Finally, print the results using the provided headers.

Note that no particular presentation order for the hierarchy class variables was requested in this call so the macro executed its default behavior. Other defaults visible are indent, lineskip and auto decimal detection.

data ADLB1;
set ADLB;
output;
if AVISITN>0 then do;
   AVISIT=strip(AVISIT)||' Change from Baseline';
   AVISITN=AVISITN+0.5;
   AVAL=ACHG;
output;
end;
run;
%mtable(
   InDSN=ADLB1,
   ResVar=AVAL,
   ColVar=TRT01A, ColList=A:B:"A" "B",
   Var1=LBCAT,
   Var2=PARAM,
   Var3=AVISIT,
   Print=Descriptor:Arm A:Arm B:Total
);

In the second example, we take greater control over the output.

First, we generate change from baseline observations by systematically outputting new values of AVAL, AVISIT, and AVISITN. Second, we specified that for the class variable LBCAT, we want the value "Hematology" to come first, then "Chemistry". Lastly, we present the values of AVISIT using the order provided by the numeric variable AVISITN, which has an one-to-one mapping with AVISIT.

Note that no order was chosen for PARAM, hence the macro performed its default action: present the unique values of this variable by order of first appearance in the input dataset. The first value of PARAM to occur where LBCAT="Hematology" happens to be "Hemoglobin (g/dL)".
In example three a frequency table is requested, so the options keyword Freq and macro variable prefix “n” were specified. The “n” tells the macro to use macro variables n1, n2, n3 for calculating the percentages in each column. These macro variables should be created before the call and the suffix matched to each column.

/*Some steps are omitted here. One keeps the highest severity for each body system and subject, another creates a sorting variable for AEBOYS: COUNT*/

Sometimes every section of a frequency table needs the same rows regardless of whether the values appear in the dataset. In example four, those values are specified with the Shell keyword, repeating the result variable as the last hierarchy variable, then listing the values needed. This results in a dummy row of 0’s where a PROC FREQ yields nothing.

Subtotals were added to the table as n’s using the Subtotal keyword, percents were calculated from the subtotals using the Subpercent keyword.
/*Two steps are omitted: one DATA STEP outputs a "Marginal" observation for every "Normal" or "Abnormal" observation and concatenates treatment number and baseline test result (N or A) into new column variable: TRTBASE, another step stores a DATA STEP inside the macro */

\texttt{\%mtable}

\texttt{Options=freq subtotal shell,}
\texttt{InDSN=ADLB1, Where=LBCAT="Hematology", OutDSN=a6,}
\texttt{ResVar=ADRIND,}
\texttt{ColVar=TRTBASE, Collist=1N:1A:"1N":2N:2A:"2N":2A",}
\texttt{Var2=PARAM,}
\texttt{Var3=AVISIT, Order3=AVISITN,}
\texttt{Var4=ANRIND, List4=Normal:Abnormal:Marginal,}
\texttt{Advanced=\texttt{\%str,}}
\texttt{Print=Descriptor:Normal:Abnormal:Marginal:Normal:Abnormal:Marginal,}
\texttt{Advprt=\texttt{\_page \_c1 desc ("Placebo" \_1-_3) ("Treatment" \_4-_6));}

In the fifth controlling output example, the macro is not designed to calculate percents using a number in the lower right of a 3 by 3 matrix. However, with enough familiarity with the output dataset, one can insert custom code into the macro via the Advanced= parameter modifying the final dataset before the PROC REPORT step is executed.

Similarly, while the print parameter does not allow grouping of columns under a common header, knowledge of the internal variable name scheme allows using the Advprt= parameter to replace the macro’s default COLUMN statement in PROC REPORT.

With this last example, it is clear that the more nuances there are to a table, a general macro would need more and more inputs to achieve the output. While it would be great to have one macro that does everything, a few macros that are easy to use may be better than having one complicated macro or many inflexible macros. Balancing functionality with ease of use should be the goal.

Once a selection of outputs are decided upon, and some sample datasets are selected, it is time to start coding.

The input examples in this section along with the layout planning and tips offered in the next section may not be relevant to designing a macro that operates on a wide dataset. The main differences are: a large number of variables need to be invokes for a wide dataset and without classification variables, there is a need to specify far more inputs to generate a table hierarchy. However, a comparable macro could be created if it is limited to one level of hierarchy, perhaps provided by the variables’ labels.

**MACRO LAYOUT & SAMPLE CODE**

**Error Detection**

Many thing can go wrong with many inputs, and once the macro is complete most will come from human error. Error detection might not be written first but it will be in front of every step to ensure assumptions are met. This could be making sure the required parameters are populated, variables submitted are present in the source dataset, and that options selected make sense and don’t clash.

**Take What You Need**

Some datasets, especially lab, are very large. Early on, process the Where= parameter if present and keep only variables mentioned in the macro call.
Create Helper Macros

Some code will be repeated many times, consider turning them into small internal macros. This one for example splits a string delimited by colons into a macro variable array with specified prefix, and the prefix itself will hold the number of items in the string.

```macro
%macro split (string=, prefix=);
%global &prefix;
%local w;
%let w=1;
%do %while (%scan(&string,&w,':') ne );
%global &prefix&w;
%let &prefix&w=%scan(&string,&w,':');
%if %index(&&prefix&w, %str(”))=0 %then %let &prefix&w="&&&prefix&w";
%let w=%eval(&w+.1);
%end;
%let &prefix=%eval(&w-1);
%mend split;
```

Two more helper macros created for the macro used in the examples are:

1. Detect when specified dataset has 0 observations, output a warning message to log, exits macro.
2. Similar to `%split` except by inputting a dataset, variable, and order variable rather than a string. The macro variable array stores unique values from the specified variable in the order of values from the provided order variable. (%auto below)

**Internally Order EVERYTHING**

There can be up to eight levels of class variables for the macro `%mtable`, and that is because the highest the author has run into is four. In the macro code however, dragging around &var1, &var2, etc. can be tedious. The required column variable aside, we need to know how many class variables are present and their vertical presentation order. After running the code below, we know both. The macro calls and what they do are in the section above, the variable n here is equal to the observation number in the input dataset created during the “take what you need” step.

```macro
%let n=0;
%do i=1 %to 8;
%if &&var&i ne %then %do;
%let n=%eval(&n+1);
%local v&n l&n;
%let v&n=&&var&i;
%let l&n=&&list&i;
%if &&list&i ne %then %split(string=&&l&n, prefix=_mtl&n);
%else %do;
%if &&order&i= %then %let order&i=n;
%auto(data=_mtemp, var=&&var&i, order=&&order&i, prefix=_mtl&n);
%end;
%end;
%end;
```

Now that the macro call parameters are mostly internalized, there is one more step: creating a new dataset where all variables needed from this point onward is transformed into a straightforward naming scheme. The author used TRT for the column variable, RES for the result variable, and V1, V2,... V&n for the other class variables. Best of all, BY statements become much easier, such as `by v1-v&n`.

**Create a KEY Dataset**

The order of presentation is already stored in macro variable arrays at this point, all there is to do is create a key dataset to merge onto the alphabetically sorted PROC step results. The key will attach sorting variables that will with one SORT step make the results take the desired presentation order.

```macro
proc sort data=_mtemp out=_mtkey nodupkey;
 by v1-v&n;
run;
```
data _mtkey;
  set _mtkey(drop=trt res);
  %do i=1 %to &n;
    %do j=1 %to &&_mtl&i;
      if v&i=&&_mtl&i&j then c&i=&j;
    %end;
  %end;
run;

Omitted from this code is a step to process the Shell option in the examples above.

Process Data

Finally, we are ready to perform the PROC FREQ or MEANS step. Programmers interested in this macro should be all too familiar with these steps. To summarize, the results are formatted, variables are concatenated if necessary, transposed, and we add in one new step: merging in the key dataset and sorting. Finally, indents, lineskip, pagebreak, and other misc tasks are performed. If desired, the macro can include a print procedure built in.

Cleaning Up

Datasets and global macro variables can be cleaned up at the end of the macro to reduce clutter.

%macro clean (prefix=);
  data _mtk;
    set sashelp.vmacro;
    if scope='GLOBAL' and substr(name, 1, %length(&prefix))=upcase(&prefix);
  run;
  data _null_; 
    call symdel(name);
  run;
  proc datasets library=work;
    delete %sysfunc(cat(&prefix,:));
  quit;
%mend clean;

This macro deletes global macro variables and work datasets with specified prefix, this is just one of many ways to clean up after a macro. Local macro variables should be used wherever possible.

Function, Efficiency, & Feasibility

The author’s experience with this general reporting macro is that it helps for more than half of all tables encountered. Both the time to write those tables and the length of the program code was reduced by 75%. If care is taken in coding the macro, run speed can be faster than regular programs by avoiding macro do loops involving DATA or PROC steps or multiple utility macro calls.

CONTACT INFORMATION:

Your comments and questions are valued and encouraged. Contact the author at:
Wayne Zhong
Octagon Research Solutions Inc.
585 East Swedesford Road, Wayne, PA
Work Phone: (610) 535-6500 x5535
Email: wzhong@octagonresearch.com

ACKNOWLEDGMENTS:
SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.