Using SAS/SCL® to Create Flexible Programs... A Super-Sized Macro
Ellen Michaliszyn, College of American Pathologists, Northfield, IL

ABSTRACT

SAS is a powerful programming language. When you find yourself writing repetitive code, you should use a macro. You probably already know that, and might even do it sometimes. There are limitations to what a macro can do, and that can get frustrating. It may even be a deterrent to trying. The big problems come when you need a parameter that is unknown at compile time... or when you spend almost as much time writing the macro calls as you would just writing the whole thing "longhand."

With SCL you are able to write code that is both efficient and stable. Once the code has been tested, it will not need to be edited if the parameters change. Hard coding of macro calls means there is potential for errors, which means that testing would be necessary after every macro call change. When there are lots of macro calls with multiple parameters, these updates can be time consuming, and it is easy to make mistakes. When the parameters that are entered manually could be found programmatically based on the data, there has to be a better way.

INTRODUCTION

Using SCL, you can dynamically create Base SAS code based on parameters either calculated or read in from a dataset. You can create Base SAS code from SCL by using submit statements to put the code in a preview buffer. The code that goes in the preview buffer can change depending on the values of the SCL variables and other data.

Anything that you can write in Base SAS can be run in SCL. Base SAS code enclosed in a submit block will stay in a preview buffer until you are ready to run it, similar to a macro. This means that you can build your code, line by line if necessary, based on whatever SCL variables, loops, and if-statements you want to use. SCL variables can be read from datasets, kept in a list, read from a SAS/AF interface or assigned conditionally.

Unlike a macro, the values of the parameters do not have to be known at compile time. I repeat... The values of the parameters DO NOT have to be known at compile time. This is a really big deal. It means values of parameters can change based on data, loops, conditions, other datasets, data read in from external sources, calculated randomly, pretty much anything you want.

An SCL program is amazingly flexible. For example, you can use SCL to write the first part of your base SAS code, leave it in the preview buffer for later, use SCL code to read in data, use that data to create more code for the preview buffer, perform some calculations in SCL, use the calculated value in a loop to add code to the preview buffer, then use SCL to put the closing statements of the base SAS code in the preview buffer and run it. When that is done, you can continue the SCL code, even reading in your newly created dataset(s) to create more base SAS code if you choose.

With this method of programming, the SCL code can be compiled and saved for later use. (NOTE: an SCL license is NOT required to run the compiled code.)
IMPLEMENTATION

SCL code can contain submit blocks. These start with submit and end with endsubmit. For the final submit block, you will use submit continue, and the matching endsubmit will prompt the code in the preview buffer to be run.

SCL variables can be used in a very similar way to macro variables. Inside submit blocks, the SCL variables are referenced the same way (&varname). One small difference is that SCL vars will be resolved inside single quotes, not just double quotes. Below are some examples of how to take advantage of the power of SCL.

EXAMPLE 1

This example includes three versions of the same program. Section 1 uses a simple macro with parameters passed to it, Section 2 uses SCL code with parameters hard coded, and Section 3 uses SCL code to read a dataset that provides the parameters.

The program itself is simple. It sorts the data and creates a report of the specified summary variables and by variables. In this example, there are two reports created. Note: This code can be run as it is written below, because it uses the sample datasets in the sashelp library.

Section 1: Macro Version

```sas
%macro SortAndSummary(dset,byvars,summaryVar);
    /* Sort the data by specified variable(s) */
    proc sort data=&dset out=tempDset;
        by &byvars;
    run;

    /* Run a procedure with parameters specified in macro */
    proc means data=tempDset;
        class &byvars;
        var &summaryVar;
    run;
%mend;

/* Report 1 */
%SortAndSummary(dset=sashelp.shoes,byvars=region subsidiary,
                summaryVar=sales);

/* Report 2 */
%SortAndSummary(dset=sashelp.shoes,byvars=product,summaryVar=sales);

/* Repeat for more reports, if necessary */
```
Section 2: SCL Code Version

In this section, the macro code is modified to use SCL exactly like the macro above. It still needs to be modified whenever the parameters change. Notice the section SORTANDSUMMARY that contains the code that was inside the macro SortAndSummary. The section GETPARMS assigns each of the parameters to an SCL variable, which are used in the SORTANDSUMMARY section.

INIT:
    link getParms;
    return;

GETPARMS:
    /* Report 1 */
    /* Assign SCL variables (same as macro variables in Section 1) */
    dset='sashelp.shoes';
    byvars='region subsidiary';
    summaryVar='sales';
    link SortAndSummary;

    /* Report 2 */
    dset='sashelp.shoes';
    byvars='product';
    summaryVar='sales';
    link SortAndSummary;

    /* Repeat for more reports, if necessary */
    return;

SORTANDSUMMARY:
    /* Here is the code that parallels the macro */
    /* It is called using a link statement */
    submit continue;
    /* Sort the data by specified variable(s) */
    proc sort data=&dset out=tempDset;
    by &byvars;
    run;

    /* Run a procedure with parameters specified in macro */
    proc means data=tempDset;
    class &byvars;
    var &summaryVar;
    run;
    endsubmit;
    return;
Section 3: SCL Code Using Parameters from a Dataset

This section reads the input parameters from a dataset and runs the reports. With this code, all edits would be done to the control dataset, and the code can be tested, compiled, and would not need to be modified if the report was needed on an additional dataset and/or variables. A preliminary step could be added to read external data and use that data to determine the parameters.

See Appendix: Example 1 for the code to create the controlParms dataset. It will look like the following:

<table>
<thead>
<tr>
<th>dset</th>
<th>byvars</th>
<th>summaryVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>sashelp.shoes</td>
<td>region subsidiary</td>
<td>sales</td>
</tr>
<tr>
<td>sashelp.shoes</td>
<td>product</td>
<td>sales</td>
</tr>
</tbody>
</table>

Below is the SCL code modified to read the parameters from the dataset. The SORTANDSUMMARY section has not been changed, and the GETPARMS section now reads in the data.

```
INIT:
   link getParms;
   return;

GETPARMS:
   /* Read parameters from control dataset */
   /* Open the control dataset */
   dsID=open('controlParms','i');
   /* Read the first row */
   rc=fetch(dsID);

   /* Repeat for every row read from the dataset */
   do until(rc=0);
      dset=getVarC(dsID,1);
      byvar=getVarC(dsID,2);
      summaryVar=getVarC(dsID,3);
      link SortAndSummary;

      /* Read the next row of control dataset */
      rc=fetch(dsID);
   end;

   /* Close the control dataset */
   if dsID then rc=close(dsID);
   return;

SORTANDSUMMARY:
   submit continue;
   proc sort data=&dset out=tempDset;
      by &byvar;
   run;

   proc means data=tempDset;
      class &byvar;
      var &summaryVar;
   run;
endsubmit;
return;
```
The benefits of doing it this way is that it can be controlled with a dataset and the code does not need to be changed if the same report with different data is requested. This method can be used for anything that a macro can do, and many things that a macro cannot. The SAS/AF interface could also be used to give the user control, either by typing the parameters or selecting from a list at run time.

EXAMPLE 2:

Here we start with data that contains a count of items for an ID in one long row. The output dataset should contain one row per item. Many times data comes from other databases or spreadsheets (especially from non-programmers) that are not organized efficiently. If you look at the data, there is often a repeating pattern. One example would be buyer, price, ID, date, price, ID, date, price, ID, date...

In this example, we have a row for each buyer. That row includes the buyer name, and for each item it includes how many of each item were purchased at what price. Below is the input data. All variables from V010 to V150 and P010 to P150 exist (counting by 10s). For those of you following along, the code to create this sample input dataset is in the appendix.

<table>
<thead>
<tr>
<th>Buyer</th>
<th>N010</th>
<th>P010</th>
<th>N020</th>
<th>P020</th>
<th>N030</th>
<th>...</th>
<th>P150</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name 1</td>
<td>16</td>
<td>5.99</td>
<td>20</td>
<td>15.49</td>
<td>13</td>
<td>...</td>
<td>23.99</td>
<td>4135551212</td>
</tr>
<tr>
<td>Name 2</td>
<td>17</td>
<td>6.99</td>
<td>21</td>
<td>16.49</td>
<td>14</td>
<td>...</td>
<td>24.99</td>
<td>7085551212</td>
</tr>
</tbody>
</table>

The final data should include one row per ID for every buyer. Each row will have the buyer name, item ID, count, price, and total cost for that item. Below is the requested output data. The value of ID matches the numeric portion of the variable name.

<table>
<thead>
<tr>
<th>Buyer</th>
<th>ID</th>
<th>Count</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name 1</td>
<td>10</td>
<td>16</td>
<td>5.99</td>
<td>95.84</td>
</tr>
<tr>
<td>Name 1</td>
<td>20</td>
<td>20</td>
<td>15.49</td>
<td>309.8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Name 2</td>
<td>150</td>
<td>21</td>
<td>24.99</td>
<td>524.79</td>
</tr>
</tbody>
</table>
/* This program builds the Base SAS code to rearrange the dataset */

/* Put the beginning of the data step in the preview buffer */
submit;
  data outputDS(keep=buyer ID count price total);
  set inputDS(keep=buyer
    /* Don't close the parenthesis so more variables can be added below */
  endsubmit;

/* Add each variable to the keep statement */
/* list vars that are multiples of 10 */
numVars=0;
do i=10 to 150 by 10;
  varnameN='N'||put(i,z3.);
  /* format z3. includes leading zeros */
  varnameP='P'||put(i,z3.);
  numVars+1;
  /* for each i, put the varnames in the preview buffer */
  submit;
    &varnameN &varnameP
  endsubmit;
end;

/* Add close parenthesis for keep statement to the preview buffer */
submit;
  );
endsubmit;

/* Continue the data step and run the code in the preview buffer */
submit continue;
/* Assign the arrays so they contain all of the N* or P* variables */
array allVarsN (*) N: ;
array allVarsP (*) P: ;

/* For each, assign ID, count and price, calculate total, and output */
do j=1 to dim(allVarsN);
  ID=j*10;
  count=allVarsN[j];
  price=allVarsP[j];
  /* Calculate Total */
  total=count*price;
  if count ne '' then output;
end;
run;
endsubmit;
return;
When the SCL code (Section 3, above) is executed, the SAS Log will contain the code that was in the preview buffer. The first part of the log is included below. Notice the keep statement. The variable names were created using SCL and each pair was added inside the do-loop.

```sas
data outputDS(keep=buyer ID count price total);
  set inputDS(keep=buyer
  /* Don’t close the parenthesis so more variables can be added below */
  N010 P010
  N020 P020
  N030 P030
  ...
  N140 P140
  N150 P150
  );
  /* Assign the arrays so they contain all of the N* or P* variables */
  array allVarsN {*} N:;
  ...
```

**EXAMPLE 3:**

The dataset mergeMe contains a list of datasets that need to be merged by byVars. This example reads the parameters from mergeMe and adds the value from each observation to the SAS preview buffer.

<table>
<thead>
<tr>
<th>dsetName</th>
<th>byVars</th>
</tr>
</thead>
<tbody>
<tr>
<td>maps.chile</td>
<td>ID</td>
</tr>
<tr>
<td>maps.chile2</td>
<td>ID</td>
</tr>
</tbody>
</table>

```sas
/* Merge all datasets found in the dataset mergeMe */

INIT:
  link runMerge;
  return;

RUNMERGE:
  submit;
    /* Put beginning of data step into the preview buffer */
    data mergedDS;
      merge
dendsubmit;

    /* Open the dataset and fetch the first observation */
    dsid=open('mergeMe','i');
    if dsid ne 0 then do;
      rc=fetch(dsid);
    ```
/* Loop through each observation in the dataset */
do until(rc ne 0);
  dset=getvarc(dsid,varnum(dsid,'dsetName'));
  byVars=getvarc(dsid,varnum(dsid,'byVars'));

  submit;
  /* Add dataset name to preview buffer */
  &dset
  endsubmit;
  rc=fetch(dsid);
  end;

  submit continue;
  /* Close data step and run code in preview buffer */
  ;
  by &byVars;
  run;
  endsubmit;
  end;
  return;

TERM:
  /* Close the mergeMe dataset */
  if dsID then rc=close(dsID);
  return;

The code from the preview buffer from this example is below. The important part of this is that the list of datasets is external.

/* Add dataset name to preview buffer */
data mergedDS;
  merge
  /* Add dataset name to preview buffer */
  maps.chile
  /* Add dataset name to preview buffer */
  maps.chile2
  /* Close data step and run code in preview buffer */
  ;
  by ID;
  run;
EXAMPLE 4:

In this example, labels are assigned for a long row of repetitive data that has generic variable names, and a report is run for each grouping (Min, Max, Mean and SD). This is a fairly typical example of data that is created and updated by someone (not a programmer, of course) in Excel. If a car make or model is added or deleted, there could be a big problem with the code. Everything would need to be renumbered if there was a make added or deleted, and the number of models (car N) would need to be manually re-checked every time. Section 3, which is the SCL version, reads in the information from another dataset and creates the code using that data. After one check of that dataset, the code would be generated perfectly.

The input dataset transData contains the min, max, mean, and SD for each car make (Acura, Audi, BMW, ..., Volvo). COL1 is Acura Min, COL2 is Acura Max, COL3 is Acura Mean, COL4 is Acura SD, COL5 is Audi Min, COL6 is Audi Max, ..., COL152 is Volvo SD.

Dataset transData
This input dataset contains summary price data for each car make.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>COL1</th>
<th>COL2</th>
<th>COL3</th>
<th>COL4</th>
<th>...</th>
<th>COL151</th>
<th>COL152</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSRP</td>
<td>23820</td>
<td>89765</td>
<td>42938.5</td>
<td>22189.0</td>
<td>...</td>
<td>36314.2</td>
<td>6176.6</td>
</tr>
<tr>
<td>Invoice</td>
<td>21761</td>
<td>79978</td>
<td>38590.8</td>
<td>19550.8</td>
<td>...</td>
<td>34216.2</td>
<td>5804.4</td>
</tr>
</tbody>
</table>

Dataset carTypes
The input dataset carTypes contains a list of car makes and number of car models for each, in the same order as the transData dataset.

<table>
<thead>
<tr>
<th>Make</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acura</td>
<td>7</td>
</tr>
<tr>
<td>Audi</td>
<td>19</td>
</tr>
<tr>
<td>BMW</td>
<td>20</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Volvo</td>
<td>12</td>
</tr>
</tbody>
</table>

The output for each make should be printed as follows:

<table>
<thead>
<tr>
<th>N=7</th>
<th>Acura Min</th>
<th>Acura Max</th>
<th>Acura Mean</th>
<th>Acura SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSRP</td>
<td>$23,820</td>
<td>$89,765</td>
<td>$42,939</td>
<td>$22,189</td>
</tr>
<tr>
<td>Invoice</td>
<td>$21,761</td>
<td>$79,978</td>
<td>$38,591</td>
<td>$19,551</td>
</tr>
</tbody>
</table>

The code to create the input datasets for this example is in the Appendix.
Section 1 - Base SAS Version

/* Create Labels for Reporting */
data outputData2;
  set transData;
  label col1='Acura Min'
    col2='Acura Max'
    col3='Acura Mean'
    col4='Acura SD'
    col5='Audi Min'
    /* LOTS more code here - 125 lines total */
  ;
run;

proc print data=outputData2 noobs label;
  label type="N=7";
  var type col1-col4;
run;

proc print data=outputData2 noobs label;
  label type="N=19";
  var type col5-col8;
run;

/* Repeat for every make - 38 total */

Section 2 - Macro Version:

%macro labelSet(make,startNum);
  %let varNum2=%eval(&startNum+1);
  %let varNum3=%eval(&startNum+2);
  %let varNum4=%eval(&startNum+3);

  col&startNum="&make Min"
  col&varNum2="&make Max"
  col&varNum3="&make Mean"
  col&varNum4="&make SD"
%mend;

data outputData2;
  set transData;
  label
    %labelSet(make=Acura,startNum=1)
    %labelSet(make=Audi,startNum=5)
    %labelSet(make=BMW,startNum=9)
    /* Repeat for each make - 38 lines total */
    %labelSet(make=Volvo,startNum=149)
  ;
run;
%macro printIt(startNum, carN);
    %let lastNum=%eval(&startNum+3);

    proc print data=outputData2 noobs label;
        label type="N=&carN";
        var type col&startNum-col&lastNum;
    run;
%mend;

%printIt(startNum=1, carN=7);
%printIt(startNum=5, carN=19);
%printIt(startNum=9, carN=20);
/* Repeat for each make - 38 lines total */
%printIt(startNum=149, carN=12);

Section 3 - SCL Version

In this example, the code written to the preview buffer is based on the carTypes dataset, so the SCL code does not need to be updated even if data is added or deleted or the number of rows in carTypes and number of variables in transData change.

INIT:
    link createDS;
    link printReport;
    return;

CREATEDS:
    /* Open the carTypes dataset read only */
    carTypeDS=open('carTypes','i');

    submit;
        data outputData2;
        set transData;
        label type='Type'
    endsSubmit;

    /* Assign labels */
    rc=fetch(carTypeDS);
    rowNum=0;

    /* for each row in the carTypes dataset, assign the 4 labels */
    do until(rc ne 0);
        carMake=getvarc(carTypeDS,1);
        rowNum+1;
        varNum1=rowNum*4-3;
        varNum2=rowNum*4-2;
        varNum3=rowNum*4-1;
        varNum4=rowNum*4;
submit;
    col&varNum1="&carMake Min"
    col&varNum2="&carMake Max"
    col&varNum3="&carMake Mean"
    col&varNum4="&carMake SD"
endsubmit;
rc=fetch(carTypeDS);
end;

/* end data statement and submit code in preview buffer */
submit continue;
    ;
    if type='_FREQ_' then make='N';
    else if type='_TYPE_' then delete;
    run;
endsubmit;
return;

PRINTREPORT:
/* Print each report */
/* Read dataset starting at the beginning */
rc=fetchObs(carTypeDS,1);
rowNum=0;
/* for each row in the carTypes dataset, run the requested report */
do until(rc ne 0);
    carN=getvarc(carTypeDS,2);
    rowNum+1;
    startNum=rowNum*4-3;
    lastNum=rowNum*4;
    submit continue;
        proc print data=outputData2 noobs label;
            label type="N=&carN";
            format col&startNum-col&lastNum dollar10.0;
            var type col&startNum-col&lastNum;
            run;
        endsubmit;
rc=fetch(carTypeDS);
    end;
    if carTypeDS then rc=close(carTypeDS)
return;
Included below is the code that was put into the preview buffer and executed. This can be found in the SAS Log after the SCL program has been run. You can see that the SCL code created what looks quite a bit like the code in Section 1 above, where the whole code was typed out.

```sas
data outputData2;
  set transData;
  label type='Type'
  col1="Acura Min" col2="Acura Max" col3="Acura Mean" col4="Acura SD"
  ;
  if type in ('_FREQ_','_TYPE_') then delete;
run;

proc print data=outputData2 noobs label;
  label type="N=7"
  format col1-col4 dollar10.0;
  var type col1-col4;
run;

proc print data=outputData2 noobs label;
  label type="N=12"
  format col149-col152 dollar10.0;
  var type col149-col152;
run;
```
CONCLUSION

Using SCL to build your Base SAS code is much more versatile than using a macro. Because the values of the parameters do not have to be known at compile time, it means values of parameters can change based on data, loops, conditions, other datasets, data read in from external sources, SCL variables, user input... pretty much anything you want.

This method of programming is extremely flexible and allows the program to be written once, tested, and used often and in many ways. Using an SCL List, a dataset, user input, or just about anything else to control the program allows a cleaner and less error-prone option than code rewrites. It brings SAS programming to the next level.

REFERENCES

SAS/AF Online Documentation: http://support.sas.com/documentation/onlinedoc/af/

ACKNOWLEDGEMENTS

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

CONTACT INFORMATION

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

Ellen Michaliszyn
College of American Pathologists
847-832-7194
emichal@cap.org

A SAS catalog is available with all the SCL and Base SAS code in this paper. Please send me a request. It takes just a couple clicks to send it. Also, I would love to help answer any questions you have implementing this powerful way of programming.
APPENDIX

Example 1 Section 3 Setup

Below is the code to create the control dataset (run in base SAS)

```sas
data controlParms;
  length dset byvars summaryVar $40;

  /* Report 1 */
  dset='sashelp.shoes';
  byvars='region subsidiary';
  summaryVar='sales';
  output;

  /* Report 2 */
  dset='sashelp.shoes';
  byvars='product';
  summaryVar='sales';
  output;

  /* Repeat for more reports, if necessary */
run;
```

Example 2 Setup

```sas
data inputDS(drop=i);
  /* this creates a dataset with 2 rows of dummy data */
  do i=1 to 2;
    Buyer='Name '||put(i,best2.);
    /* N010 is Count of ID 10, P010 is price of ID 10 */
    if i=1 then Phone='4135551212';
    else Phone='7085551212';
    output;
  end;
run;
```
Example 4 Setup

/* Create Datasets for Example 5 (Base SAS) */

proc means data=sashelp.cars noprint;
  class make;
  var msrp invoice;
  output out=inputData(where=(_type_ ne 0 and _stat_ ne 'N'));
run;

proc transpose data=inputData out=transData(rename=(make=TYPE))
  name=make;
run;

data transData;
  set transData;
  if type in ('MSRP','Invoice');
run;

proc freq data=sashelp.cars;
  tables make / noprint out=carTypes(drop=percent);
run;