Paper 105-2008

Using the SAS® DATA Step and PROC SQL to Create *Macro Arrays*

Stuart Long, Westat, Durham, NC
Ed Heaton, Westat, Rockville, MD

ABSTRACT

Successful analysis in SAS® often requires lengthy repetitive steps; the same code is executed on numerous data points. Iteration of SAS code is often easily achieved with the use of *Macro Arrays*. The programmer can employ numerous methods for creating these arrays, as well as choosing among different structural designs. PROC SQL and the SAS DATA step can each provide clear advantages to the creation of these arrays. PROC SQL and the DATA step can take advantage of descriptive information that is available to the programmer in the data set header line. This paper will discuss the benefits of the assignment of values to macro-variable array elements using CALL SYMPUTX() in the DATA step and the SELECT INTO expression in PROC SQL. This paper is intended for SAS programmers who have a working knowledge of Base SAS® and the SAS Macro Facility.

INTRODUCTION

The elements of macro variables are text strings. These strings can consist of variable names, data set names, character strings, procedure or DATA step options, or any other list of items or code that can be processed by the SAS System. This paper will concentrate on the aspect of iteration and will be further narrowed to deal with the iteration of variable names, or variable descriptive information, as used by SAS procedures and DATA steps.

Often an analysis will require iteration of SAS procedures for numerous data points. Executing a PROC MEANS or PROC FREQ for multiple variables can be as simple as listing the variables in the VAR or TABLES statement. However, if a programmer wishes to reduce the amount of output which must be reviewed or if the iteration of procedures is quite complex, it may be more prudent to process the analysis one data point(e.g. name of variable) at a time. One common method of achieving this type of iteration is with the use of *Macro Arrays*. If the list of data points is numbered in the tens, hundreds, or possibly even thousands, *Macro Arrays* can offer the programmer a method of reducing the amount of code that must be created.

Macro Arrays are normally used in one of two constructs.

1) *Array of Macro Variables*: An ordinal list of macro variables where each contains one element to be processed.

2) *Macro Variable Array*: One macro variable containing the entire array of elements to be processed.

The simplest way to create an *Array of Macro Variables* is by entering an ordinal sequence of macro variable assignments using %LET statements:

```
%LET disease1=diabetes;
%LET disease2=hbp;
%LET disease3=cvd;
%LET disease4=asthma;
```

The *Macro Variable Array* can be assigned using one %LET statement:

```
%LET disease=diabetes hbp cvd asthma;
```

The SAS DATA step and PROC SQL can serve as excellent tools for creating these macro variable constructs. The method which a programmer chooses to use may vary, depending on programming preferences.

ARRAYS

Have you ever wished you could run code like the following?

```
ARRAY symptom {*} symptom1-symptom40;
ARRAY disease {*} diabetes--asthma;
DO i = 1 TO DIM(symptom);
    DO j = 1 TO DIM(disease);
        PROC FREQ DATA=mydata;
            TABLES symptom{i}*disease{j} / CHISQ RELRISK;
        RUN;
    END;
END;
```

Since arrays of this nature must be defined and executed within a DATA step, we see that the above code is syntactically incorrect. However, by defining two sets of macro variables, we can achieve the above logic within the syntax requirements of the SAS system.

First, let's examine the structure of the DATA step array.

```
DATA myset;
    SET mydata;
    ARRAY disease {*} diabetes hbp cvd asthma;
    total_diseases=0;
    DO i = 1 TO DIM(disease);
        IF disease{i}=1 THEN total_diseases+1;
    END;
RUN;
```

- The name of the array is DISEASE.
- The index is I.
- The array element is identified by DISEASE{I}.
- Where I=1, DISEASE{I} = DIABETES.

ARRAY OF MACRO VARIABLES

In the example below we use an *Array of Macro Variables*, which has a nearly isomorphic relationship to the structure of a DATA step array.

```
%LET disease1=diabetes;
%LET disease2=hbp;
%LET disease3=cvd;
%LET disease4=asthma;
%MACRO construct1;
    %DO i = 1 %TO 4;
        PROC FREQ DATA=mydata;
            TABLES symptom1*&&disease&i;
        RUN;
    %END;
%MEND construct1;
```

- The root name of the array is DISEASE.
- The index is I.
- The array element is identified by &&DISEASE&I.
- Where I = 1, &&DISEASE&I = DIABETES.

All items in the *Array of Macro Variables* can be mapped to the DATA step array, except the "&&" syntax which controls the final resolution of the macro variables in the array. The advantage of the *Array of Macro Variables* is that you can create iterative code, which executes outside of the data step on a list of variables in a manner similar to iterative code that is executed on a list of variables within a DATA step.

MACRO VARIABLE ARRAY

In the *Macro Variable Array*, all of the items exist in one macro variable.

```
%LET disease= diabetes hbp cvd asthma;
%MACRO construct2;
    %LET i =1 ;
    %DO %UNTIL(NOT %LENGTH(%SCAN(&disease,&i))) ;
        PROC FREQ DATA=mydata;
            TABLES symptom1*%SCAN(&disease,&i);
        RUN;
        %LET i=%EVAL(&i+1);
    %END;
%MEND construct2;
```

- The root name of the array is DISEASE.
- The index is I.
- The array element is identified by "%SCAN(&DISEASE,&I)".
- Where I = 1, %SCAN(&DISEASE,&I) = DIABETES.

CALL SYMPUTX:
Elements are assigned to macro variables in the DATA step by using the CALL SYMPUT routine:

```
CALL SYMPUT(MacroVariable, Value);
```

The contents of *MacroVariable* can be a SAS variable that contains a valid SAS macro variable name as its value or it can be a character string enclosed within quotes where the character string is a valid macro variable name. *Value* contains the element assigned to *MacroVariable.*

A basic example of the CALL SYMPUT routine is in the following code:

```
DATA _NULL_;
    INPUT varname $ 1-10;
    CALL SYMPUT("disease",varname);
CARDS;
diabetes
;
```

This is equivalent to the following %LET assignment made outside of the Data Step:

```
%LET disease=diabetes;
```

In our examples, we will be concatenating numeric values with character strings to create the names of our ordinal macro variables. The CATS function, as seen in the example below, trims leading and trailing blanks created by the numeric to character conversion, as well as avoiding any warnings to the SAS LOG. By using the CALL SYMPUTX when assigning the numeric value of _N_ to the macro variable, we are, once again, able to avoid leading and trailing blanks in the numeric to character conversion. SYMPUTX also removes leading and trailing blanks from character values before loading them into the macro variable. These two features are almost always desired. The execution of CALL SYMPUTX is identical to the execution of CALL SYMPUT with the exception of the removal of these extraneous blanks. For this reason, we will use CALL SYMPUTX exclusively for the remainder of this paper.

When creating an *Array of Macro Variables* using this method, ordinal macro variables can be generated from within the DATA step:

```
DATA _NULL_;
    INFILE CARDS EOF=lastrecord;
    INPUT varname $ 1-10 ;
    CALL SYMPUTX(CATS("disease_",_N_),varname);
    RETURN ;
    lastRecord: CALL SYMPUTX("dimvars",_N_);
CARDS;
diabetes
hbp
cvd
asthma
;
```

In the above example, CALL SYMPUTX is used to make assignments of values to the macro variables "disease_1 – disease_4". In the statement

```
CALL SYMPUTX(CATS("disease_",_N_),varname);
```

processing of the first observation will result in the following resolution:

```
CATS("disease_" , _N_)  resolves to "disease_1"
varname resolves to "diabetes"
```

This statement produces the below assignment:

```
CALL SYMPUT("disease_1","diabetes");
```

This DATA step creates an *Array of Macro Variables* equivalent to the four %LET statements seen earlier in this paper:

```
%LET disease1=diabetes;
%LET disease2=hbp;
%LET disease3=cvd;
%LET disease4=asthma;
```

One additional macro variable is created, “dimvars”, which identifies the dimension of this *Array of Macro Variables* for future macro processing.

Similarly, we can use the DATA step to create a *Macro Variable Array*:

```
DATA _NULL_;
    INFILE CARDS EOF=lastRecord ;
    LENGTH var_array $32767;
    INPUT varname $;
    RETAIN var_array;
    var_array=CATX(' ' , var_array , varname );
    RETURN ;
    lastRecord: CALL SYMPUTX("disease",var_array);
CARDS;
diabetes
hbp
cvd
asthma
;
```

The CATX function allows us to concatenate the array elements while a space is inserted between each one. This DATA _NULL_ produces a macro variable assignment which is identical to the previously shown %LET assignment:

```
%LET disease=diabetes hbp cvd asthma;
```

Due to the nature of processing *Macro Variable Arrays*, a separate macro variable containing the dimension of this array is not needed.

PROC SQL

We can create an *Array of Macro Variables* with PROC SQL which is identical to that created by the DATA step:

```
PROC SQL;
    CREATE TABLE mydata (varname CHAR(10));
    INSERT INTO mydata
        VALUES("diabetes")
        VALUES("hbp")
        VALUES("cvd")
        VALUES("asthma");
    SELECT varname INTO :disease1 - :disease&SysMaxLong FROM mydata;
    %LET dimvars = &sqlObs;
QUIT;
```

Note that in the SELECT ... INTO … clause, we allowed for more macro variables than we need. PROC SQL doesn't care, as long as we have enough. In this case, it will only create the first four macro variables.
Whenever we run a PROC SQL query, SAS creates an automatic macro variable named &SQLOBS which holds the number of rows output. Since this is an automatic macro variable and thus subject to change by the system, we should write the value to another macro variable that we control.

Likewise, we can create a *Macro Variable Array* using the following PROC SQL code:

```
PROC SQL;
    CREATE TABLE mydata (varname CHAR(10));
    INSERT INTO mydata
        VALUES("diabetes")
        VALUES("hbp")
        VALUES("cvd")
        VALUES("asthma");
    SELECT varname INTO :disease SEPARATED BY " " FROM mydata;
QUIT;
```

We tell SAS to separate each value that it puts into the array with a space. We don't have to use a space; we can use a comma or a dash or even more than one character like a comma followed by a space. Without the SEPARATED BY clause, each value put into the macro variable would overwrite the previous value and we would end up with an array with the single value "asthma".

CREATING MACRO ARRAYS USING THE DATA STEP ARRAY

We have seen how *Macro Arrays* are close in structure to the DATA step array. This similarity between the two allows us to use the DATA step array to create a *Macro Array.* First, we'll look at how to generate an *Array of Macro Variables*.

```
DATA _NULL_;
    ARRAY diseases {*} diabetes  hbp  cvd asthma;
    CALL SYMPUTX("dim_row",DIM(diseases));
    DO i = 1 TO DIM(diseases);
        CALL SYMPUT(CATS("disease_",i),VNAME(diseases{i}));
    END;
RUN;
```

Although this example creates the same *Array of Macro Variables* as can be created using the previously presented four %LET statements, there is no real coding advantage for the programmer. However, if we read an existing SAS data set using the SET statement, new information is now available to assist the creation of *Arrays of Macro Variables*.

```
DATA _NULL_;
    IF 0 THEN SET mydata;
    ARRAY disease {*} diabetes -- asthma;
    CALL SYMPUTX("dim_row",DIM(disease));
    DO i = 1 TO DIM(disease);
        CALL SYMPUT(CATS("row_",i),VNAME(disease{i}));
    END;
RUN;
```

When you submit a DATA step for execution, it is first compiled and then executed. The DATA step reads the SET statement at compile time and adds the variable names and attributes to the descriptor information. Observations are read at execution time. Since the SET statement is in a conditional expression that is always false (IF 0), the DATA step will read only the data set header, containing the variable names and attributes. The descriptor information is then available for the creation of macro variables without processing any actual observations -- saving execution time.

In this example, diabetes, hbp, cvd and asthma occur positionally in order within the data set. This fact is stored in the data set header. We take advantage of this information by using the "- -" construct to assign all four of these variables to the DATA step array, "disease". This edge does not seem great when dealing with four variables, but if we had 100 medical conditions, this method presents a clear advantage over coding the assignment of elements to 100 variables using 100 %LET statements.

There are many options for taking advantage of the descriptor information in the data set header. Below are just a few when using the ARRAY statement for creating *Macro Arrays*.

```
ARRAY symptom  {*} symptom:;                *all variables with a common prefix;
ARRAY numerics {*} _numeric_;               *all numeric variables in a data set;
ARRAY disease  {*} diabetes-numeric-asthma; *all numeric variables in a range;
ARRAY charvar  {*} $ county-character-state; *all character variables in a range;
```

CREATING MACRO ARRAYS USING PROC SQL

There is no one-step PROC SQL equivalent to the DATA step array procedure outlined above. At best, we can use PROC CONTENTS to generate a table of variable names and then use PROC SQL to create the *Array of Macro Variables*.

```
PROC CONTENTS
    DATA=mydata( KEEP = diabetes -- asthma )
    OUT=vars( KEEP = name varNum )
    NOPRINT;
RUN ;

PROC SQL NOPRINT ;
    SELECT name INTO :row_1 - :row_&SysMaxLong
        FROM vars
        ORDER BY varnum ;
QUIT ;
```

For each *Macro Array* we wish to generate with PROC SQL, we will need to output a dataset using PROC CONTENTS in the above fashion. CPU time required for the PROC SQL and DATA step solutions is nearly equivalent. Since the PROC SQL solution requires the creation of a data set that is not needed for each *Marco Array,* the authors believe that the DATA step approach presents a method easier for instruction purposes. For this reason, we will forgo the PROC SQL solution during the remainder of this paper.

USING ARRAYS OF MACRO VARIABLES TO EXECUTE PROCEDURES ITERATIVELY

To demonstrate the use of *Macro Arrays*, we will code a simple example using the PROC FREQ seen on the first page of this paper. In the data set, "mydata", we have 4 disease variables positionally ordered within the data set and 40 symptom variables, ordinally named from symptom1 to symptom40. The DATA _NULL_ in the following example will use information from the data set header, while avoiding the processing of any observations.

ARRAY OF MACRO VARIABLES

We first use the DATA _NULL_ to create the two *Arrays of Macro Variables* with corresponding dimension variables:

```
DATA _NULL_;
    IF 0 THEN SET mydata;
    ARRAY disease {*} diabetes -- asthma;
    ARRAY symptom {*} symptom:;
    CALL SYMPUTX("dim_disease",DIM(disease));
    CALL SYMPUTX("dim_symptom",DIM(symptom));
    DO i = 1 TO DIM(disease);
        CALL SYMPUTX(CATS("disease_",i),VNAME(disease{i}));
    END;
    DO i = 1 TO DIM(symptom);
        CALL SYMPUTX(CATS("symptom_",i),VNAME(symptom{i}));
    END;
RUN;
```

The macro executes the PROC FREQ within 2 nested %DO loops:

```
%MACRO runfreq(_row    = ,
               _col    = ,
               _dimrow = ,
               _dimcol = ,
               _dset   = );
    %LOCAL i j;
    %DO i = 1 %TO &_dimrow;
        %DO j = 1 %TO &_dimcol;
            PROC FREQ DATA=&_dset;
                TABLE &&&_row.&i * &&&_col.&j / CHISQ RELRISK;  RUN;
        %END;
    %END;
%MEND runfreq;
```

The macro call passes parameters for the root name of each *Array of Marco Variables*, their corresponding dimensions and the name of the data set to be analyzed:

```
%runfreq(_row    = disease_,
         _col    = symptom_,
         _dimrow = &dim_disease,
         _dimcol = &dim_symptom,
         _dset   = mydata)
```

The above macro, **RUNFREQ()**, will generate 4 x 40, or 160, 2x2 tables using PROC FREQ. The first three iterations of the %DO loop will generate the following SAS code:

```
PROC FREQ DATA=mydata;
    TABLE diabetes * symptom1 / CHISQ RELRISK;
RUN;
PROC FREQ DATA=mydata;
    TABLE hbp * symptom1      / CHISQ RELRISK;
RUN;
PROC FREQ DATA=mydata;
    TABLE cvd * symptom1      / CHISQ RELRISK;
RUN;
```

MACRO VARIABLE ARRAY

Macro Variable Arrays can be created using DATA step arrays, as well:

```
DATA _NULL_;
    IF 0 THEN SET mydata;
    LENGTH row_array col_array $32767;
    ARRAY disease {*} diabetes -- asthma ;
    ARRAY symptom {*} symptom: ;
    DO i = 1 TO DIM(disease);
        row_array=CATX( " " , row_array , VNAME(disease{i}) );
    END;
    DO i = 1 TO DIM(symptom);
        col_array=CATX( " " , col_array , VNAME(symptom{i}) );
    END;
    CALL SYMPUTX("disease",row_array);
    CALL SYMPUTX("symptom",col_array);
    STOP;
RUN;
```

This version of the macro, **RUNFREQ()**, executes the PROC FREQ within 2 nested %DO %UNTIL loops:

```
%MACRO runfreq(_row    = ,
               _col    = ,
               _dset   = );
    %LOCAL i j;
    %LET i=1; %LET j=1;
    %DO %UNTIL(NOT %LENGTH(%SCAN(&_row,&i)));
        %DO %UNTIL(NOT %LENGTH(%SCAN(&_col,&j)));
            PROC FREQ DATA=&_dset;
              TABLE %SCAN(&_row,&i) * %SCAN(&_col,&j) / CHISQ RELRISK;
            RUN;
            %LET j = %EVAL(&j+1);
        %END;
        %LET i = %EVAL(&i+1);
    %END;
%MEND runfreq;
```

This macro call passes parameters for the names of each *Marco Variable Array an*d the name of the data set to be analyzed:

```
%runfreq(_row    = &disease,
         _col    = &symptom,
         _dset   = mydata)
```

As was seen with the example for the *Array of Macro Variables*, this version of the macro, **RUNFREQ()**, will generate the following SAS code during the first 3 iterations of the %DO %UNTIL loops:

```
PROC FREQ DATA=mydata;
    TABLE diabetes * symptom1 / CHISQ RELRISK;
RUN;
PROC FREQ DATA=mydata;
    TABLE hbp * symptom1      / CHISQ RELRISK;
RUN;
PROC FREQ DATA=mydata;
    TABLE cvd * symptom1      / CHISQ RELRISK;
RUN;
```

The use of PROC FREQ serves as an excellent demonstration for how to use Macro Arrays, but is not very practical. Let's now look at some more practical uses of this structure.

ITERATION OF AN ANALYSIS USING ARRAYS OF MACRO VARIABLES

The authors cannot stress enough the importance of properly labeling and formatting variables within SAS data sets. In addition, foresight is often needed when labeling and formatting to make sure that the data set can be used in the manner intended.

In the following example, the format variables are assigned the prefix of "r-" to the referent level of all variables that are to be modeled in a logistical regression analysis. All numeric variables will be processed to identify those with a predefined referent level. If a referent level is found for a particular variable, it will be added to the *Array of Macro Variables* for further processing.

Although we will read the descriptor information using DATA _NULL_, processing of the observational data is needed to identify the referent levels. Therefore, we will read the entire data set into the DATA step, not just the data set header.
The following are format variables assigned to the data set variables “farmsize” and “eyecolor” respectively.

```
VALUE _fs     0 = "r-None"
              1 = "< 50 acres"
              2 = "51-500 acres"
              3 = "501-1000 acres"
              4 = "> 1000 acres";
VALUE _color  1 = "Bown"
              2 = "Green"
              3 = "r-Blue"
              4 = "Black"
              5 = "Other";
```

A logistical regression analysis will be run on all variables in a data set that have a referent level identified in their corresponding formatted values. The model will be adjusted by the following covariables: age, state, race. These categorical covariables will appear in the CLASS statement, but will default to the highest ordered value as the referent level since we are only interested in the referent level comparison of our final modeled exposure variable. The model outcome will be “asthma”, which is coded as a binary variable (0=No Asthma / 1=Asthma). The first macro variable in the *Array of Macro Variables* will contain the exposure variable, “farmsize”, as its element, followed by “eyecolor” as the element of the second macro variable.

We will define a hash table to keep track of variable names as they are added to our *Array of Macro Variables*. Each time we locate a variable with a predefined referent level, we will check to see if it is in our hash table. If not, it will be added to the hash table, after which, we will assign values to the *Macro Array* for variable names and a corresponding *Macro Array* containing referent levels for the variable names.

```
%MACRO logist(dataset = ,
              outcome = ,
              covars  = );
    %LOCAL i;
    DATA _NULL_;
        LENGTH var_name $32;
        IF ( _N_ = 1 ) THEN DO;
            DECLARE HASH h();
            h.DEFINEKEY(“var_name”);
            h.DEFINEDONE();
        END;
        SET &dataset END=eof;
        ARRAY numvar {*} _NUMERIC_;
        DO i = 1 TO DIM(numvar);
            IF ( VVALUE(numvar{i}) EQ: “r-“ ) THEN DO;
               var_name = VNAME( numvar{i} );
               var_ref  = VVALUE(numvar(i));
               IF NOT h.ADD() THEN DO;
                  CALL SYMPUTX(CATS("ordinal_",h.NUM_ITEMS),var_name);
                  CALL SYMPUTX(CATS("ref_",h.NUM_ITEMS),var_ref);
               END;
            END;
        END;
        IF eof THEN CALL SYMPUTX("dimvars",h.NUM_ITEMS);
    RUN;
/* Execute the PROC LOGISTIC for each variable identified in the data set as having   */
/* a referent level.                                                                  */
    %DO i = 1 %TO &dimvars;
        PROC LOGISTIC DATA=&dataset ORDER=INTERNAL DESCENDING;
          CLASS  &covars
                 &&ordinal_&i (PARAM=REF REF="&&ref_&i");
          MODEL &outcome = &covars &&ordinal_&i;
        RUN;
    %END;
%MEND logist;
```

```
/* invoke the macro */
%logist(dataset = mydata ,
        outcome = asthma ,
        covars  = age state race)
```

The above macro will produce the following PROC LOGISTICs during the first two iterations of the %DO loop:

```
PROC LOGISTIC DATA=mydata ORDER=INTERNAL DESCENDING;
    CLASS  age state race
           farmsize (PARAM=REF REF="r-None");
    MODEL asthma = age state race farmsize;
RUN;

PROC LOGISTIC DATA=mydata ORDER=INTERNAL DESCENDING;
    CLASS  age state race
           eyecolor (PARAM=REF REF="r-Blue");
    MODEL asthma = age state race eyecolor;
RUN;
```

ITERATION OF CUSTOMIZED MACROS USING MACRO ARRAYS

A common practice among seasoned SAS programmers is to create a macro which produces a desired result, and then to nest this macro within another macro that codes iteration using *Macro Arrays*. We will demonstrate this use of *Macro Arrays* using the **SUMMARY()** macro discussed in the NESUG 2007 paper BB10[1]. The **SUMMARY()** macro will produce a SAS data set of summary statistics condensed into one observation, from three SAS procedures that are executed within the macro: a PROC FREQ, a crude PROC LOGISTIC and an adjusted PROC LOGISTIC.

The following is an example macro invocation for the SUMMARY macro:

```
%summary(__dset   = mydata,         /* data set to be analyzed                        */
         __var1   = asthma,         /* outcome variable                               */
         __var2   = farmsize,       /* modeled exposure variable                      */
         __ref    = 0,              /* referent level for the exposure variable       */
         __basevs = age state race, /* additional model covariables                   */
         __out    = asthma_summary, /* output data set containing summary statistics */
         __report = YES);           /* request to print a report of summary results   */
```

In the above invocation of the **SUMMARY()** macro, the SAS data set "asthma_summmary" is produced which is used to generate a user requested report as seen in diagram 1.

Diagram 1

Model adjusted for age state race.

Variable	Ever Diagnosed with asthma: Yes N	Yes %	No N	No %	P-Value Crude	P-Value Adjusted	OR Adjusted	95% CI Adjusted
Label/Format								
Size of Farm								
No Farm	346	46.9	6658	39.8	.	.		
< 500 acres	278	37.7	6358	38.0	0.0361	0.7049	0.949	0.722 1.246
500+ acres	113	15.3	3697	22.1	<.0001	0.5378	1.143	0.747 1.747

Foresight was used in the coding of this macro to allow it to be used iteratively by building a composite summary data set using an imbedded PROC APPEND, based on repetitive invocations of the macro. Therefore, each time the macro is invoked, the results will be added to the results obtained from all previous invocation of the macro, given that the "__out =" parameter remains the same.

This macro can be nested inside another macro that achieves iteration through the use of *Macro Arrays,* Note that this example has been simplified by standardizing all of the **SUMMARY()** macro parameters except "__var2 =" (the modeled exposure variable).

```
%MACRO runsum(arrayvars = );
    %LOCAL i;
 /* create the Array of Macro Variables */
    DATA _NULL_;
        IF 0 THEN SET mydata;
        ARRAY ex_vars {*} &arrayvars ;
        CALL SYMPUTX("dim_exposure",DIM(ex_vars));
        DO i = 1 TO DIM(ex_vars);
            CALL SYMPUTX(CATS("exposure_",i),VNAME(ex_vars(i)));
        END;
    RUN;

 /* insert the invocation of the SUMMARY macro within a %DO loop */
    %DO i = 1 %TO &dim_exposure;
        %IF &i=&dim_exposure %THEN %LET report=Yes; %ELSE
                                   %LET report=No;
        %summary(__dset   = mydata,
                 __var1   = asthma,
                 __var2   = &&exposure_&i,
                 __ref    = 0,
                 __basevs = age state race,
                 __out    = asthma_summary,
                 __report = &report);
    %END;
%MEND runsum;

%runsum(arrayvars = farmsize -- grain_years)
```

The above macro produces the summary report seen in diagram 2.

Diagram 2

Model adjusted for age state race.

Variable	Ever Diagnosed with asthma: Yes N	Yes %	No N	No %	P-Value Crude	P-Value Adjusted	OR Adjusted	95% CI Adjusted
Label/Format								
Size of Farm								
No Farm	346	46.9	6658	39.8	.	.		
< 500 acres	278	37.7	6358	38.0	0.0361	0.7049	0.949	0.722 1.246
500+ acres	113	15.3	3697	22.1	<.0001	0.5378	1.143	0.747 1.747
Frequency of Milking Cows								
Never	215	6.8	1275	7.8	.	.	.	. .
Monthly	225	7.1	1104	6.8	0.0679	0.0445	1.238	1.005 1.525
Weekly	1396	43.9	7195	44.0	0.0768	0.0007	1.325	1.125 1.560
Daily	1341	42.2	6773	41.4	0.0434	<.0001	1.426	1.205 1.689
Frequency of Tractor Use								
Never	2250	71.1	12203	74.9	.	.	.	. .
Monthly	744	23.5	3465	21.3	0.0010	0.0042	1.146	1.044 1.259
Weekly	138	4.4	528	3.2	0.0004	0.0040	1.341	1.098 1.637
Daily	33	1.0	95	0.6	0.0015	0.0186	1.642	1.086 2.481
Years Grown Grain								
None	51	17.9	1262	21.7	.	.	.	. .
< 5 years	47	16.5	917	15.7	0.2507	0.2305	1.287	0.852 1.943
5-9 years	90	31.6	1539	26.5	0.0393	0.0477	1.435	1.004 2.052
10-19 years	50	17.6	1005	17.3	0.3071	0.2669	1.258	0.839 1.887
20+ years	46	16.1	1083	18.6	0.8107	0.6627	1.096	0.725 1.657

CONCLUSION

Iterative processing is an important need in many SAS environments. *Macro Arrays* offer the programmer a strong tool for accomplishing many if not most of the repetitive tasks that are faced in both the research and business environment. When the iteration required is based on repetitive processing of large lists of data set variables, the DATA step and PROC SQL offer what may be an optimal solution for creating *Macro Arrays*. The authors have presented choices for the programmer between the *Array of Macro Variables* and the *Macro Variable Array*. In addition, the programmer can opt for using either the DATA step or PROC SQL for creating these *Macro Arrays*. Given whichever preference the programmer exercises to create these constructs, the authors believe many tasks can be better optimized by using *Macro Arrays* as created by the methods presented in this paper.

DISCLAIMER: The contents of this paper are the work of the authors and do not necessarily represent the opinions, recommendations, or practices of Westat. The examples presented in this paper are derived from analyses of the Agricultural Health Study by the National Cancer Institute, and the National Institutes of Environmental Health Sciences. All statistics included in this paper have been altered for presentation purposes and do not represent actual statistical associations from the study.

REFERENCES

1. Long, S., Abolafia, J., Park, L. (2006). Using SAS® ODS to extract and merge statistics from multiple SAS procedures into a single summary report, a detailed methodology. *Proceedings of the SUGI 31 Conference.*

CONTACT INFORMATION

Stuart Long (long3@niehs.nih.gov)
Westat
1009 Slater Road, Suite 110
Durham, NC 27703

Ed Heaton (EdHeaton@Westat.com)
Westat
1650 Research Blvd
Rockville, MD 20850

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.