Using Metadata for Data Driven Programming
Brian Varney, Trilogy Consulting, Kalamazoo, Michigan

ABSTRACT
Creating dynamic, data driven programs is a very powerful tool. Often we can use the metadata or the project data itself to help write our programs. This paper will introduce some of the concepts related to writing SAS® programs which will generate pieces of SAS code in a data driven manner.

Parts of the SAS language used in this discussion include the following:
1. macro call routines (call execute, call symput)
2. PROC SQL
3. SAS Dictionary Tables

Programming techniques used in this discussion include the following:
1. using sequential macro variables with macro do loops
2. generating SAS code, writing it to an external file, and %including it at a later time
3. generating and executing SAS code using call execute
4. checking if a data set exists
5. checking if a variable exists in a data set
6. checking if a data set is empty
7. using subqueries for multiple data set subsetting

INTRODUCTION
A major problem that can occur in project code is hard coding project specific facts into a program. A good example is a program used across multiple protocols in a clinical trial. For instance, we may know that all protocols except for 3 and 5 already have the variable age calculated in the demographics module. This may tempt us to have something like the following in our program.

%if &prot.=3 or &prot=5 %then
%do;
%* calculate the age variable.;
%end;

This works fine but what happens if more protocols come up at a later time which also lack the age variable. The code will have to be changed when this happens. It may be beneficial to programatically check to see if the variable age exists in the demographics module. Based on that data driven check, the age variable can be calculated if necessary.

Using the parts of the SAS language and programming techniques discussed in this paper can result in low maintenance, reusable programs. They can also sometimes save a lot of typing!

This paper is intended for those with previous exposure to macro and SQL. Please be advised that some of these SAS tools are not available prior to SAS version 6.12.

PARTS OF THE SAS LANGUAGE
First we will review the parts of the SAS language utilized in this paper.

1. CALL SYMPUT
Call symput is a data step statement used to create a macro variable from a value or data set variable. Following is an example...

data _null_;   
  call symput('macvar','macro variable value');
run;

The first argument is the name of the macro variable. The second argument is the macro variable value. This can also be a data set variable.

CALL EXECUTE
Call execute is also a data step statement which is used to execute whatever is passed into it. An example follows...

data _null_;   
  call execute('proc options; run;');
run;

There is only one argument in a call execute.

Caution: There is a version 7 bug which compresses out spaces when combining multiple call executes. For example, the following code runs in version 6.12 but not in version 7.

data _null_;   
  call execute('proc ');
  call execute('options; run;');
run;

Version 7 ends up trying to execute the code procoptions; run; instead of proc options; run;

This is fixed in Version 8 Developers Release.

2. PROC SQL
PROC SQL will be used in this paper to access the dictionary tables as well as create macro variables similar to how we can use call symput.

An example follows which places the data set label from SASHELP.RETAIL into a macro variable called dslabel;

proc sql noprint;
  select memlabel into :dslabel
  from dictionary.tables
  where upcase(libname)='SASHELP' and
  upcase(memname)='RETAIL';
quit;

Following is an example creating sequential macro variables from multiple records. This loads variable names from SASHELP.RETAIL into macro variables var1, var2, ..., varn. Even though room is left for 9,999 sequential macro variables only the necessary ones are created.

proc sql noprint;
  select name into :var1 - :var9999
  from dictionary.columns
  where upcase(libname)='SASHELP' and
  upcase(memname)='RETAIL';
quit;

Following is an example creating one concatenated macro variable from multiple records. This loads variable names from SASHELP.RETAIL into macro variable varlist separated by whatever we specify. This provides us with a
handy list of variables to include in a keep= or drop= data set option later in the program.

```sql
proc sql noprint;
  select name into :varlist separated by ' ' 
  from dictionary.columns
  where upcase(libname)='SASHELP' and 
  upcase(memname)='RETAIL';
quit;
```

If you are interested in building up a macro variable of quoted values try the following

separated by ' ** ** '

3. DICTIONARY TABLES

SAS dictionary tables contain the data behind the scenes in SAS a.k.a. the metadata. The dictionary tables are only directly accessible from PROC SQL. There is also a corresponding set of data step views but it is much less efficient to use these due to where statements not getting pushed back to the dictionary tables. Following is a list of dictionary tables and their contents.

**DICTIONARY.CATALOGS**
- **LIBNAME** char(8) label='Library Name',
- **MEMNAME** char(8) label='Member Name',
- **MEMTYPE** char(8) label='Member Type',
- **OBJNAME** char(8) label='Object Name',
- **OBJTYPE** char(8) label='Object Type',
- **OBJDESC** char(40) label='Object Description',
- **MODIFIED** char(8) label='Date Modified',
- **ALIAS** char(8) label='Object Alias'

**DICTIONARY.COLUMNS**
- **LIBNAME** char(8) label='Library Name',
- **MEMNAME** char(8) label='Member Name',
- **MEMTYPE** char(8) label='Member Type',
- **NAME** char(8) label='Column Name',
- **TYPE** char(4) label='Column Type',
- **LENGTH** num label='Column Length',
- **NPOS** num label='Column Position',
- **VARNUM** num label='Column Number in Table',
- **LABEL** char(40) label='Column Label',
- **FORMAT** char(16) label='Column Format',
- **INFORMAT** char(16) label='Column Informat',
- **IDXUSAGE** char(9) label='Column Index Type'

**DICTIONARY.EXFILES**
- **FILEREF** char(8) label='Fileref',
- **XPATH** char(80) label='Path Name',
- **XENGINE** char(8) label='Engine Name'

**DICTIONARY.INDEXES**
- **LIBNAME** char(8) label='Library Name',
- **MEMNAME** char(8) label='Member Name',
- **MEMTYPE** char(8) label='Member Type',
- **NAME** char(8) label='Column Name',
- **IDXUSAGE** char(9) label='Column Index Type',
- **INDXNAME** char(8) label='Index Name',
- **INDXPOS** num label='Pos. of Column in Concatenated Key',
- **NOMISS** char(3) label='Nomiss Option',
- **UNIQUE** char(3) label='Unique Option'

**DICTIONARY.MEMBERS**
- **LIBNAME** char(8) label='Library Name',
- **MEMNAME** char(8) label='Member Name',
- **MEMTYPE** char(8) label='Member Type',
- **ENGINE** char(8) label='Engine Name',
- **INDEX** char(8) label='Indexes',
- **PATH** char(80) label='Path Name'

**DICTIONARY.OPTIONS**
- **OPTNAME** char(16) label='Session Option Name',
- **SETTING** char(200) label='Session Option Setting',
- **OPTDESC** char(80) label='Option Description'

**DICTIONARY.TABLES**
- **LIBNAME** char(8) label='Library Name',
- **MEMNAME** char(8) label='Member Name',
- **MEMTYPE** char(8) label='Member Type',
- **MEMLABEL** char(40) label='Dataset Label',
- **TYPEMEM** char(8) label='Dataset Type',
- **CRDATE** num format=DATETIME label='Date Created',
- **MODATE** num format=DATETIME label='Date Modified',
- **NOBS** num label='Number of Observations',
- **OBSLEN** num label='Observation Length',
- **NVAR** num label='Number of Variables',
- **PROTECT** char(3) label='Type of Password Protection',
- **COMPRESS** char(8) label='Compression Routine',
- **REUSE** char(3) label='Reuse Space',
- **BUFSIZE** num label='Bufsize',
- **DELOBS** num label='Number of Deleted Observations',
- **INDXTYPE** char(9) label='Type of Indexes'

**DICTIONARY.VIEWS**
- **LIBNAME** char(8) label='Library Name',
- **MEMNAME** char(8) label='Member Name',
- **MEMTYPE** char(8) label='Member Type',
- **ENGINE** char(8) label='Engine Name'

**DICTIONARY.TITLES**
- **TYPE** char(1) label='Title Location',
- **NUMBER** num label='Title Number',
- **TEXT** char(200) label='Title Text'

**DICTIONARY.MACROS**
- **SCOPE** char(9) label='Macro Scope',
- **NAME** char(8) label='Macro Variable Name',
- **OFFSET** num label='Offset into Macro Variable',
- **VALUE** char(200) label='Macro Variable Value'

Now that we have discussed the major SAS tools of interest in this paper, let's talk about using these components to write code that writes code.

In version 8 there are changes to the dictionary tables. Some of the values in the tables are in mixed case instead of all upper case. For instance, the values for the variable NAME in dictionary.columns appears in mixed case instead of strictly upper case now. It is important to remember to go back to old programs and make sure case sensitivity is addressed.

**PROGRAMMING TECHNIQUES**

Following is one problem solved using three different programming techniques.

Many times we have access to files which contain variable labels. Instead of typing all of these in or doing some awkward cutting and pasting. Consider the following data set called work.vardata containing the variables var and varlabel. A data set called demo contains the variables var1, var2, and var3 which we want to apply these variable labels to.
There are various ways that we can generate the label statement programmatically. Even though there are only three variables here, there are times that we are presented with hundreds or thousands of variables which need labels generated and we do not want to type them in! What we want to end up with is the following label statement inside of a SAS data step or procedure.

Label var1='Patient #' var2='Date of Visit' var3='Date of Birth';

1. USING SEQUENTIAL MACRO VARIABLES WITH MACRO DO LOOPS
proc sql noprint;
  select var into :VAR1-:VAR9999 from vardata;
  select varlabel into :VL1-:VL9999 from vardata;
  select count(*) into :NUMVL from vardata;
quit;
%macro runit;
  proc datasets lib=work;
    modify demo;
    label
      %do a=1 %to &NUMVL.;
      &&VAR&a="&&VL&a.”
      %end;
  quit;
%mend runit;
Note: macro do loops must be inside of a macro definition.
The double ampersand reference causes the macro reference to be scanned twice. For instance, when a=1 &&var&a is resolved in the following way.
First scan: when two ampersands are encountered, they are resolved into one ampersand. The &a resolves as it normally would.
Second scan: &var1 resolves as it normally would.

2. GENERATING SAS CODE, WRITING IT TO AN EXTERNAL FILE, AND %INCLUDING IT AT A LATER TIME
data _null_;
  file 'tempprog.sas';
  put @1 'proc datasets lib=work;';
  put @4 'modify demo;';
  put @4 'label
    %do a=1 %to &NUMVL.;
      &&VAR&a="&&VL&a.”
    %end;
  quit;
%mend runit;
%include 'tempprog.sas';
Another handy thing to do programmatically is to check the status of a data set.

4. CHECKING TO SEE IF A DATA SET EXISTS
proc sql noprint;
  select left(put(count(*),8.)) into :exist
    from dictionary.members
    where upcase(libname)=’WORK’ and
      upcase(memname)=’JUNK’;
quit;
If the data set work.junk exists, the macro variable exist will be equal to 1; otherwise it will be equal to 0.

5. CHECKING TO SEE IF A VARIABLE EXISTS IN A DATA SET
proc sql noprint;
  select left(put(count(*),8.)) into :exist
    from dictionary.columns
    where upcase(libname)=’WORK’ and
      upcase(memname)=’JUNK’ and
      upcase(name)=’DOG’;
quit;
If the variable dog exists in the data set work.junk, the macro variable exist will be equal to 1; otherwise it will be equal to 0.

6. CHECKING TO SEE IF A DATA SET IS EMPTY
proc sql noprint;
  select left(put(nobs,8.)) into :nobs
    from dictionary.tables
    where upcase(libname)=’WORK’ and
      upcase(memname)=’JUNK’ ;
quit;
If the data set work.junk is empty, the macro variable exists will be equal to 1; otherwise it will be equal to 0.

7. USING SUBQUERIES FOR CROSS DATA SET SUBSETTING
proc sql;
  create table final as
    select *
      from prot008.demo
    where patient not in (select distinct patient
      from prot008.vitals);
quit;
The data set final only has records from demo for patients who do not appear in the vitals data set.
CONCLUSION
As one can imagine, there are many opportunities to apply these techniques in our day to day programming. Whenever developing SAS code to be used over again or as part of a larger system of SAS programs, we should try to avoid hard coding information into our programs. This paper only scratches the surface in providing examples of data driven programming. Try to apply these ideas to your programming problems.

For even more ideas in data driven programming please see the paper “%SYSFUNC – The Brave New Macro World” by Chris Yindra. This paper begins on page 259 of the SUGI 23 Proceedings.

REFERENCES
SAS Institute Inc, Getting Started with the SQL Procedure, Version 6, First Edition
“%SYSFUNC – The Brave New Macro World”, Chris Yindra

ACKNOWLEDGMENTS
I would like to take this time to thank everyone I have worked with and learned from in my exciting SAS adventures.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. The author may be contacted at:

Brian Varney
Trilogy Consulting
5148 Lovers Lane
Kalamazoo, Michigan 49002
Work Phone: (800) 831-6314
Fax: (616) 344-1887
Email: BKVarney@TrilogyUSA.com
Web: www.TrilogyUSA.com