Paper 177-2008

Using Data-Set Values and Variable Names Outside of the DATA Step

Bruce Gilsen, Federal Reserve Board

INTRODUCTION

SAS ® system users sometimes need to use data set values or variable names outside of a DATA step, for tasks such as the following.

- Call a macro once per observation of a data set, using variable values from the data set as macro parameter values.
- Call a macro for selected observations of a data set, based on the value of a variable (e.g., when GNP > 100) or once for each unique value of a variable.
- Build a list of variable values for subsequent use, such as with a DATA step IN operator or the macro %SCAN function, or to export to another application.
- Build a list of data set variable names for subsequent use, such as in an ARRAY definition statement.

Using a series of simple but detailed examples, this paper shows some alternate ways to build lists of data set values and variable names, and provides a brief introduction to DICTIONARY tables.

Example 1. Call a macro once for each observation of a data set. Use variable values from the data set as macro parameters.

Data set ONE has the following values. All variables except COMPANY are numeric.

```
Obs   income  expenses  company

 1      22       33     ibm
 2      -4       55     aol
 3      66       77     gmc
 4      88       99     att
```

Macro MAC1 has two parameters, INC and FIRM. Call MAC1 once for each observation of data set ONE, using the values of INCOME and COMPANY as parameter values, as follows.

- The first time MAC1 is called,
  - INC = the value of INCOME from the first observation of data set ONE.
  - FIRM = the value of COMPANY from the first observation of data set ONE.
  - MAC1 is called as follows: %MAC1(INC=22, FIRM=ibm)
- The second time MAC1 is called,
  - INC = the value of INCOME from the second observation of data set ONE.
  - FIRM = the value of COMPANY from the second observation of data set ONE.
  - MAC1 is called as follows: %MAC1(INC=-4, FIRM=aol)
- Subsequent calls to MAC1 work the same way.

This task can be thought of as having two steps.

Step 1. Copy the values of INCOME and COMPANY to macro variables. Two ways to do this are in a DATA step (Example 1.a) or with PROC SQL (Example 1.b).

Step 2. In a macro, use a loop to call MAC1 once for each set of values of INCOME and COMPANY.

To make this example as simple as possible, let's assume the following.

- All values of INCOME, a numeric variable, have the same number of digits.

- All values of COMPANY, a character variable, have the same length.
- Since ONE has 4 observations, and values from all observations are used, MAC1 will be invoked 4 times.

Example 1.a. Create the macro variables (step 1) in a DATA step.

Code and Results.

```
%macro mac1(inc=,firm=);      /* Small, useless macro for illustrative purposes */
  %put in macro mac1 inc= &inc  firm= &firm;
%mend mac1;

  /* Step 1: copy values of INCOME and COMPANY to macro variables */
data _null_;
  set one;
  call symput('income'|| put(_n_,1.), put(income,2.));
  call symput('company'|| put(_n_,1.), company);
run;

  /* Step 2: call macro MAC1 once for each set of values of INCOME and COMPANY */
%macro macloop1;
  %local j;
  %do j = 1 %to 4;
    %mac1(inc=&&income&j,firm=&&company&j);
  %end;
%mend macloop1;
%macloop1;  /* call the macro */

%put _all_;      /* show all macro variables, including the ones we created */
```

Macro MAC1 writes the following text to the SAS log.

```
in macro mac1 inc= 22  firm= ibm
in macro mac1 inc= -4  firm= aol
in macro mac1 inc= 66  firm= gmc
in macro mac1 inc= 88  firm= att
```

The text written to the SAS log by the %PUT _ALL_ statement includes the following.

```
GLOBAL COMPANY1 ibm
GLOBAL COMPANY2 aol
GLOBAL COMPANY3 gmc
GLOBAL COMPANY4 att
GLOBAL INCOME1 22
GLOBAL INCOME2 -4
GLOBAL INCOME3 66
GLOBAL INCOME4 88
```

Details.

1. CALL SYMPUT.

```
call symput('income'|| put(_n_,1.), put(income,2.));
call symput('company'|| put(_n_,1.), company);
```

CALL SYMPUT copies a value from a DATA step to a macro variable. It takes two arguments.

The first argument is the name of the macro variable.

Two macro variables are created In each observation. Their names are INCOME or COMPANY concatenated with the automatic variable _N_, e.g.,

- In the first observation, macro variables INCOME1 and COMPANY1 are created.
- In the second observation, macro variables INCOME2 and COMPANY2 are created.

_N_ is the number of times the DATA step has executed (and is also the observation number in straightforward DATA steps). _N_ is converted from numeric to character with the PUT function before it is concatenated. There are less than 10 observations, so _N_ always has 1 digit, and the PUT function uses the 1. format.

The second argument is the value of the macro variable, which is set to the value of INCOME or COMPANY in the current observation. For example, the following macro variables are created from the data values in the first two observations of ONE:

```
Observation   Macro variable created   Value   Origin of macro variable value
     1              INCOME1             22    INCOME in 1st observation
     1              COMPANY1           IBM    COMPANY in 1st observation
     2              INCOME2             -4    INCOME in 2nd observation
     2              COMPANY3           AOL    COMPANY in 2nd observation
```

Since macro variable values are always character, not numeric, the value of INCOME is converted to character with the PUT function before it is assigned to the macro variable. All values of INCOME have the same length, 2, so the PUT function uses the 2. format. COMPANY is already character; no conversion is needed.

2. The %DO loop.

```
%do j = 1 %to 4;
  %mac1(inc=&&income&j,firm=&&company&j);
%end;
```

The double ampersand (&&) in the %DO loop is called an "indirect reference" to a macro variable. The macro processor resolves two ampersands to one ampersand (&& to &) rather than applying the & to the text that follows.

The first time the loop executes, macro variable J (&J) is 1, and the macro processor processes &&INCOME&J left to right in two passes as follows:

```
Pass 1:
1. && resolves to &
2. INCOME is text and is unchanged
3. &J is resolved to 1
Result: &&INCOME&J is resolved to &INCOME1

Pass 2:
1. &INCOME1 is resolved to 22, so the value of the macro parameter INC is 22.
```

In the same fashion, &FIRM resolves in two passes to IBM, resulting in the following call to the macro MAC1:

```
%mac1(inc=22,firm=ibm);
```

The second time the loop executes, macro variable J (&J) is 2, and the macro processor processes &&INCOME&J left to right in two passes as follows:

```
Pass 1:
1. && resolves to &
2. INCOME is text and is unchanged
3. &J is resolved to 2
Result: &&INCOME&J is resolved to &INCOME2

Pass 2:
1. &INCOME2 is resolved to -4, so the value of the macro parameter INC is -4.
```

In the same fashion, &FIRM resolves in two passes to AOL, resulting in the following call to the macro MAC1:

```
%mac1(inc=-4,firm=aol);
```

The && is needed to make the macro processor scan twice. If you code &INCOME&J instead of &&INCOME&J, the macro processor tries to resolve &INCOME and &J and concatenate the values. This generates an error unless &INCOME was previously defined (in that case, there is no error message but an unexpected result).

3. A series of macro variables like &INCOME1 &INCOME2 .... &INCOME*n* that each contain one element of information and differ by an ascending integer suffix is sometimes called an *array of macro variables* in conference papers and the SAS-L Internet newsgroup.

Example 1.b. Create the macro variables (step 1) with PROC SQL.

Code and Results.

```
%macro mac1(inc=,firm=);      /* Small, useless macro for illustrative purposes */
  %put in macro mac1 inc= &inc  firm= &firm;
%mend mac1;

  /* Step 1: copy values of INCOME and COMPANY to macro variables */
proc sql noprint;
  select income, company
    into
      :income1-:income4,
      :company1-:company4
    from one;
quit;

  /* Step 2: call macro MAC1 once for each set of values of INCOME and COMPANY */
%macro macloop1;
  %local j;
  %do j = 1 %to 4;
    %mac1(inc=&&income&j,firm=&&company&j);
  %end;
%mend macloop1;
%macloop1;  /* call the macro */

%put _all_; /* show all macro variables, including the ones we created */
```

Details.

The SELECT statement copies values of INCOME and COMPANY from data set ONE to macro variables as follows.

- INCOME and COMPANY from the first observation are copied to macro variables INCOME1 and COMPANY1.
- INCOME and COMPANY from the second observation are copied to macro variables INCOME2 and COMPANY2.
- INCOME and COMPANY from the third observation are copied to macro variables INCOME3 and COMPANY3.
- INCOME and COMPANY from the fourth observation are copied to macro variables INCOME4 and COMPANY4.

Macro variables INCOME1-INCOME4 and COMPANY1-COMPANY4 have the same values as in Example 1.a. Macro MAC1, Step 2, and the results are the same as in Example 1.a.

Example 1.c. Call the macro with CALL EXECUTE.

A third way to approach this problem is to use CALL EXECUTE, which allows you to generate SAS code within a DATA step. The code is executed at the following times.

- Macros and macro language elements execute immediately.
- SAS language statements (whether generated by CALL EXECUTE or by macro language elements generated by CALL EXECUTE) execute at the next step boundary (after the current DATA step finishes executing).

In this example, CALL EXECUTE is used to call macro MAC1. MAC1 writes the same results to the SAS log as in Examples 1.a and 1.b.

Code and Results.

```
%macro mac1(inc=,firm=);      /* Small, useless macro for illustrative purposes */
  %put in macro mac1 inc= &inc  firm= &firm;
```

```
%mend mac1;

data _null_;
  set one;
  call execute ('%mac1(inc='||income||',firm='||company||')');
run;
```

Details.

Because macro language elements execute immediately and SAS language statements execute after a step boundary, CALL EXECUTE must be used with caution. Timing errors, described in the *SAS 9.1.3 Macro Language: Reference* (*Interfaces with the Macro Facility* chapter, *CALL EXECUTE Routine Timing Details* section) occur if CALL EXECUTE generates a macro that has references to macro variables created by CALL SYMPUT in that macro. For that reason, it might be safer to use the methods in Examples 1.a and 1.b for tasks similar to those in this paper.

Example 2. Call a macro for selected observations of a data set, based on the values of a variable in the data set. Use variable values from the data set as macro parameter values.

Data set TWO has the following values. All variables except COMPANY are numeric. We want to call macro MAC1 for values of INCOME greater than 0. As in the first example, the values of INCOME and COMPANY are used as parameter values when MAC1 is called.

```
Obs   income  expenses  company

 1      22       33     ibm
 2      -4       55     aol
 3      66       77     gmc
 4     888       99     microsoft
```

This example is similar to Example 1, with a few more realistic assumptions that make the code a bit more detailed but much more generalized and useful. Now, let's assume the following.

- Call macro MAC1 for values of INCOME greater than 0.
- Values of INCOME, a numeric variable, can have up to 10 digits, including 2 after the decimal point.
- Values of COMPANY, a character variable, can have up to 20 characters.
- The number of observations is between 1 and 999.

Example 2.a. Create the macro variables (step 1) in a DATA step.

Code and Results.

```
%macro mac1(inc=,firm=);         /* Small, useless macro for illustrative purposes */
  %put in macro mac1 inc= &inc  firm= &firm;
%mend mac1;

  /* Step 1: copy values of INCOME and COMPANY to macro variables for all
     observations in which INCOME is greater than 0 */
data _null_;
  set two end=last;
  where income gt 0;
  call symput('income'|| compress(put(_n_,3.)), compress(put(income,11.2)));
  call symput('company'|| compress(put(_n_,3.)), trim(left(company)));
  if last then call symput('num_values', compress(put(_n_,3.)));
run;

  /* Step 2: call macro MAC1 once for each set of values of INCOME and COMPANY
     that were written to macro variables */
%macro macloop1;
  %local j;
  %do j = 1 %to &num_values;
```

```
    %mac1(inc=&&income&j,firm=&&company&j);
  %end;
%mend macloop1;
%macloop1;  /* call the macro */

%put _all_;  /* show all macro variables, including the ones we created */
```

Macro MAC1 writes the following text to the SAS log.

```
in macro mac1 inc= 22.00  firm= ibm
in macro mac1 inc= 66.00  firm= gmc
in macro mac1 inc= 888.00  firm= microsoft
```

The text written to the SAS log by the %PUT _ALL_ statement includes the following.

```
GLOBAL INCOME3 888.00
GLOBAL COMPANY1 ibm
GLOBAL INCOME2 66.00
GLOBAL INCOME1 22.00
GLOBAL COMPANY2 gmc
GLOBAL COMPANY3 microsoft
GLOBAL NUM_VALUES 3
```

Details.

1. Determining the number of times to call MAC1.

Since the number of observations is between 1 and 999, and MAC1 is only called when INCOME is greater than 0,

- the variable used to construct the suffixes can have 1-3 digits.
- the number of times to invoke macro MAC1 varies because it is data-dependent, and must be determined.

Macro variable NUM_VALUES, the number of values of INCOME and COMPANY copied to macro variables, is used in the %DO loop in MACLOOP1 to call MAC1 the appropriate number of times.

END=LAST on the SET statement creates a temporary variable called LAST that is 0 until the SET statement reads the last observation of the data set, when it is set to 1.

The conditional expression IF LAST is only true (has a non-zero value) after the last observation has been read, so that is the only time the CALL SYMPUT statement that creates NUM_VALUES is executed. NUM_VALUES is created after all observations in data set TWO have been read and processed, when we know how many macro variables were created.

2. The first CALL SYMPUT statement.

```
call symput('income'|| compress(put(_n_,3.)), compress(put(income,11.2)));
```

COMPRESS(PUT(_N_,3.)) ensures that a macro variable name does not have extraneous blanks, e.g., is "INCOME1", not "INCOME 1" or "INCOME 1". It works as follows.

- PUT(_N_,3.) converts the value of _N_ from numeric to character, with a length of 3, right-justified. That is, 2 is converted to " 2" (2 blanks, then a 2), 25 is converted to " 25" (1 blank, then 25), etc.

- COMPRESS removes blanks, e.g., changing " 2" to "2" or " 25" to "25".

Similarly, COMPRESS(PUT(INCOME,11.2)) ensures that a macro variable value does not have extraneous blanks, as follows.

- PUT(INCOME,11.2)) converts the value of INCOME to character, allowing 11 characters for the largest possible value. One character is the decimal point, and two characters are digits to the right of the decimal point. The result is right-justified, so that 22 becomes "   22.00" (6 blanks, then 22.00) and 888 becomes "   888.00" (5 blanks, then 888.00).

- COMPRESS removes blanks, e.g., changing "   22.00" to "22.00" or "   888.00" to "888.00".

3. The second CALL SYMPUT statement.

```
call symput('company'|| compress(put(_n_,3.)), trim(left(company)));
```

COMPRESS(PUT(_N_,3.)) is the same as in note 2 above.

TRIM(LEFT(COMPANY)) removes trailing blanks from the value of COMPANY when it is copied to a macro variable. TRIM(LEFT( )) is used instead of COMPRESS( ) to account for values of COMPANY that have blanks, e.g., "general motors".

4. If we must determine when to write values to macro variables inside the DATA step (for example, with a subsetting IF statement instead of the WHERE statement used above), _N_ cannot be used as the suffix of the macro variable names, because unselected observations lead to non-contiguous macro variable names (e.g., INCOME1, INCOME3, INCOME4,....). In that case, add a variable that keeps count of the number of values written to macro variables, use it as the suffix, and increment it by one every time an observation meets the selection criteria.

Example 2.b. Create the macro variables (step 1) with PROC SQL.

In this example, create the macro variables with PROC SQL instead of a DATA step. Otherwise, the code and results are the same as in Example 2.a.

Code and Results.

```
%macro mac1(inc=,firm=);      /* Small, useless macro for illustrative purposes */
  %put in macro mac1 inc= &inc  firm= &firm;
%mend mac1;

  /* Step 1: copy values of INCOME and COMPANY to macro variables for all
     observations in which INCOME is greater than 0 */
proc sql noprint;
  select count(*)
    into :num_values
    from two
    where income gt 0;

  %let num_values=&num_values;      /* remove leading and trailing blanks */

  select income, company
  into
    :income1-:income&num_values,
    :company1-:company&num_values
    from two
    where income gt 0;
quit;

 /* Step 2: call macro MAC1 once for each set of values of INCOME and COMPANY
    that were written to macro variables */
%macro macloop1;
  %local j;
  %do j = 1 %to &num_values;
    %mac1(inc=&&income&j,firm=&&company&j);
  %end;
%mend macloop1;
%macloop1;  /* call the macro */

%put _all_; /* for illustrative purposes, display all macro variables, including
               those we created */
```

Details.

1. The first SELECT statement.

```
select count(*)
  into :num_values
  from two
  where income gt 0;
```

The summary function COUNT determines the number of observations in data set TWO where INCOME is greater than 0, and

the result is stored in the macro variable NUM_VALUES.

Initially, &NUM_VALUES is right-justified because leading blanks are preserved when writing a numeric field to a macro variable. The macro variable is assigned to itself with a %LET statement to remove leading and trailing blanks.

The summary function COUNT used in a PROC SQL SELECT statement differs from the DATA step function COUNT, which counts occurrences of a substring of characters in a character string. FREQ and N are aliases for COUNT.

2. The second SELECT statement.

```
select income, company
into
  :income1-:income&num_values,
  :company1-:company&num_values
  from two
  where income gt 0;
```

The number of macro variables to create is determined by the data. In this example, the number of observations is 3, so the first SELECT statement sets &NUM_VALUES to 3, and the second SELECT statement resolves to the following.

```
select income, company
into
  :income1-:income3,
  :company1-:company3
  from two
  where income gt 0;
```

Macro variables INCOME1, INCOME2, INCOME3, COMPANY1, COMPANY2, and COMPANY3 are created.

Example 3. Call a macro once for each distinct value of a variable in a data set. Use the variable's values as macro parameter values.

Data set THREE has the following values. All variables except COMPANY are numeric. We want to call macro MAC2 once for each distinct value of COMPANY. The values of COMPANY are parameter values when MAC2 is called.

```
Obs   income  expenses  company

 1      22       33     ibm
 2      -4       55     aol
 3      99       66     ibm
 4     777       88     microsoft
 5     100      666     ibm
```

Example 3.a. Create the macro variables (step 1) with PROC SORT and a DATA step.

Code and Results.

```
%macro mac2(firm=);              /* Small, useless macro for illustrative purposes */
  %put in macro mac2 firm= &firm;
%mend mac2;

  /* Step 1: copy distinct values of COMPANY to macro variables */
proc sort data=three (keep=company) out=_three nodupkey;
  by company;
run;
data _null_;
  set _three end=last;
  call symput('company'|| compress(put(_n_,3.)), trim(left(company)));
  if last then call symput('num_values', compress(put(_n_,3.)));
run;

  /* Step 2: call macro MAC2 once for each values of COMPANY written to macro variables */
%macro macloop2;
```

```
  %local j;
  %do j = 1 %to &num_values;
    %mac2(firm=&&company&j);
  %end;
%mend macloop2;
%macloop2;  /* call the macro */

%put _all_;  /* show all macro variables, including the ones we created */
```

Macro MAC2 writes the following text to the SAS log.

```
in macro mac2 firm= aol
in macro mac2 firm= ibm
in macro mac2 firm= microsoft
```

The text written to the SAS log by the %PUT _ALL_ statement includes the following.

```
GLOBAL COMPANY1 aol
GLOBAL COMPANY2 ibm
GLOBAL COMPANY3 microsoft
```

Details.

Here are details about parts of the code that differ from Example 2.a.

1. COMPANY is the only variable from data set THREE needed in the PROC SORT step. The KEEP= data set option specifies that only COMPANY is read from data set THREE, which is more efficient than reading all variables.

2. The PROC SORT NODUPKEY option eliminates observations with duplicate BY values, so each distinct value of COMPANY is written to the output data set once.

Example 3.b. Create the macro variables (step 1) with PROC SQL.

In this example, create the macro variables with PROC SQL instead of PROC SORT and a DATA step. Otherwise, the code and results are the same as in Example 3.a.

Code and Results.

```
%macro mac2(firm=);              /* Small, useless macro for illustrative purposes */
  %put in macro mac2 firm= &firm;
%mend mac2;

  /* Step 1: copy distinct values of COMPANY to macro variables */
proc sql noprint;
  select count(distinct company)
    into :num_values
    from three;

  %let num_values=&num_values;      /* remove leading and trailing blanks */

  select distinct company
    into :company1-:company&num_values
    from three;
quit;

  /* Step 2: call macro MAC2 once for each values of COMPANY written to macro variables */
%macro macloop2;
  %local j;
  %do j = 1 %to &num_values;
    %mac2(firm=&&company&j);
  %end;
%mend macloop2;
%macloop2;  /* call the macro */

%put _all_;  /* show all macro variables, including the ones we created */
```

Details.

Here are details about parts of the PROC SQL code that differ from Example 2.b.

1. The first SELECT statement.

```
select count(distinct company)
  into :num_values
  from three;
```

The DISTINCT keyword specifies that the number of unique values of COMPANY is stored in the macro variable NUM_VALUES.

2. The second SELECT statement.

```
select distinct company
  into :company1-:company&num_values
  from three;
```

Macro variables COMPANY1, COMPANY2, ... COMPANY*num_values* are created, containing the distinct values of COMPANY.

Example 4. Copy the distinct values of a variable from a data set to a macro variable, separated by a delimiter such as a comma or blank.

In Example 3.a, each distinct value of the variable was copied to a different macro variable (COMPANY1, COMPANY2,...). Now, all distinct values of the variable are copied to a *single* macro variable. An example of why this technique could be useful is shown in this example, where a comma-separated list of the distinct values of COMPANY_ID is used in a subsequent DATA step with the DATA step IN operator.

Data sets FOUR and MASTER_SALES have the following values. All variables except STATE are numeric.

Data set FOUR:

Obs	price	company_id
1	22	1
2	5	20
3	99	3
4	777	10
5	100	3
6	200	3

Data set MASTER_SALES:

Obs	state	my_company
1	ny	1
2	dc	55
3	va	3
4	md	3
5	ca	4

Code and Results.

```
  /* Create macro variable ALL_COMPANY_ID containing the unique values of COMPANY_ID
     separated by commas. */
proc sql noprint;
  select distinct company_id
    into :all_company_id separated by ','
    from four;
quit;

%put &all_company_id; /* display global macro variable ALL_COMPANY_ID */

  /* In a DATA step, test if the values of variable MY_COMPANY match the values of
     COMPANY_ID from data set FOUR. */
data important;
  set master_sales;
  if my_company in (&all_company_id) then do;
    put 'Observation ' _n_ 'matched COMPANY_ID';
    /* other SAS code here */
  end;
```

```
run;
```

The PUT statement in the DATA step writes the following text to the SAS log.

```
Observation 1 matched COMPANY_ID
Observation 3 matched COMPANY_ID
Observation 4 matched COMPANY_ID
```

The %PUT statement writes the following text to the SAS log.

```
1,3,10,20
```

Details.

1. When specifying a character that separates the values, do not use a separator that is contained in the values. Two typical mistakes are inappropriately specifying a blank (e.g., data values like “General Motors”) or a comma (e.g., data values like “Washington, DC”).

2. Other possible reasons to copy all unique data values to a single macro variable include displaying them or writing them to an external file for use by another program or software package.

3. A single macro variable that contains multiple elements separated by a delimiter is sometimes called a *macro variable array* in conference papers and the SAS-L Internet newsgroup.

Example 5. A macro variable contains a series of values separated by blanks. Call a macro once for each value.

In the previous examples, the values were generated from a SAS data set, and the number of values could be easily determined in a DATA step or with the PROC SQL summary function COUNT. To make the code more generalized, let's assume that the macro variable values could come from another user or application, and the number of values is not known.

Code and Results.

```
%macro mac2(firm=);              /* Small, useless macro for illustrative purposes */
  %put in macro mac2 firm= &firm;
%mend mac2;

  /* ALLFIRMS is a macro variable with blank-separated values to be passed to a macro */
%let allfirms=ibm aol microsoft gmc att;

 /* Loop through macro variable ALLFIRMS.  Parse it into separate words (a.k.a.
    values or tokens), splitting it at the blanks.  Call macro MAC2 once per value. */
%macro macloop3;
  %local j currentfirm;
  %let currentfirm = %scan(&allfirms, 1, %str( ));          /* parse 1st word */
  %let j = 1;                          /* parse 2nd, 3rd, ... word in %DO loop */

  %do %while (&currentfirm ne ) ;             /* stop when %scan returns null */
    %mac2(firm=&currentfirm);                       /* call MAC2 for ith firm */
    %let j=%eval(&j+1);                     /* set counter to parse next word */
    %let currentfirm = %scan(&allfirms, &j, %str( ));       /* parse next word */
  %end;
%mend macloop3;
%macloop3;
```

Macro MAC2 writes the following text to the SAS log.

```
  in macro mac2 firm= ibm
  in macro mac2 firm= aol
  in macro mac2 firm= microsoft
  in macro mac2 firm= gmc
  in macro mac2 firm= att
```

Details.

1. The %SCAN function.

The %SCAN function takes three arguments.

%SCAN(*argument*, *n* <,*delimiters*>);

- *argument* is the character string or text expression to parse (search).
- *n* is the position of the word to extract, e.g., 3 for the 3rd word. It must be an integer or resolve to a integer if evaluated by %EVAL.
- *delimiters* are an optional list of characters that separate (delimit) words. In the code above, the delimiter is %STR( ), a blank. If this argument is omitted, the default delimiters are all of the following:
  - Mainframe, Linux, and PC: blank.<(+&!$*);-/,%|
  - also on the Mainframe: cents sign, vertical bar with break in middle, negation sign
  - also on Linux and PC: ^

%SCAN returns a null string if *n* is greater than the number of words (e.g., %SCAN('a bb d',*n*, %str( )); when *n* is greater than 3).

2. The %DO %WHILE loop.

There are many other ways to code the %DO loop and the rest of MACLOOP3. The style used in this example requires that the %LET CURRENTFIRM statement be coded twice. But, by calling MAC2 inside the %DO loop, no error occurs if the macro variable ALLFIRMS is empty. In this case, ALLFIRMS is hard-coded, but in an application that generates ALLFIRMS, it is good programming practice to allow for it to be empty.

Example 6. Create a macro variable containing the names of all numeric variables in a SAS data set. Use the macro variable in an array definition.

Data set FIVE has the following values. All variables except COMPANY are numeric; DATE contains SAS date values.

```
Obs    date        a        bbb     c1234567890   company

 1     16801       1         2          3         aol
 2     16832       4         5          6         ibm
 3     16860       7         8          9         microsoft
```

In the previous examples, SAS data set *values* were made available outside of a DATA step. Now, we are doing something different; making data set *variable names* available outside of a DATA step.

One approach, which is analogous to Examples 1.a, 2.a, and 3.a, is as follows.

- Execute PROC CONTENTS (or PROC DATASETS with the CONTENTS statement), and use the OUT= option or the Output Delivery System (ODS) to create a data set that contains variable names.
- In a DATA step, read the data set created in the first step and create the macro variables.

Instead, we will use PROC SQL to read the variable names from a DICTIONARY table. Before proceeding, a brief introduction to DICTIONARY tables is necessary.

What are DICTIONARY tables?

DICTIONARY tables are a set of read-only SAS data views that contain information about the current SAS session: data libraries, SAS data sets, SAS macros, and external files in use or available, and SAS system options settings.

DICTIONARY table content is updated as your SAS session progresses. So, information retrieved from DICTIONARY tables is current as of the time you retrieve the information.

You can access a DICTIONARY table one of the following ways.

- Use PROC SQL, as shown in this paper.
- Refer to the PROC SQL view of the table in the SASHELP library in any SAS PROC or DATA step.

An introduction to DICTIONARY tables is in the *DICTIONARY Tables* chapter in *SAS 9.1.3 Language Reference: Concepts.* For complete information, including a table of all dictionary tables, see *The SQL Procedure* chapter in the *Base SAS 9.1.3 Procedures Guide*.

Code and Results.

```
  /* Create macro variable NUMERIC_VARS containing the names of all
     numeric variables separated by a blank */
proc sql noprint;
  select name
    into :numeric_vars separated by ' '
    from dictionary.columns
    where libname="WORK"
      and memname="FIVE"
      and type = "num";
quit;

%put &numeric_vars;  /* display global macro variable NUMERIC_VARS */

  /* In a DATA step, use the macro variable NUMERIC_VARS in an array
     definition and loop through all numeric variables */
data new;
  set five;
  array numvars (*) &numeric_vars ;
  do j = 1 to dim(numvars);
    /* do something with each array element (each numeric variable) here */
  end;
run;
```

The %PUT statement writes the following text to the SAS log.

```
date a bbb c1234567890
```

The following ARRAY statement is generated in the DATA step.

```
array numvars (*) date a bbb c1234567890 ;
```

Details.

1. The case of DICTIONARY table values matters. In DICTIONARY.COLUMNS, used in this example, LIBNAME and MEMNAME are stored in upper case and TYPE is stored in lower case, so libname="work", memname="one", or type="NUM" would not work. If you are not sure of the case, print some DICTIONARY table values or use the UPCASE function and test against upper case values, e.g., UPCASE(libname) = "WORK".

2. To see how a DICTIONARY table is defined, including the column names, use a DESCRIBE TABLE statement. For example, the following statement shows the definition of DICTIONARY.COLUMNS.

```
proc sql;
   describe table dictionary.columns;
```

3. An easier solution is available for this specific example. You can omit the SQL step, and define the array as follows.

```
array numvars (*) _numeric_ ;
```

The SQL step was demonstrated in this example for use in Example 7, where it is necessary.

Example 7. Modify Example 6 to create a macro variable or SAS data set containing the names of selected variables in a SAS data set.

In this example, we will modify Example 6 slightly to create the following.

- a macro variable containing a blank-separated list of the names of all variables in a data set
- a macro variable containing a blank-separated list of the names of all numeric variables in a data set except DATE
- a data set containing the names of all numeric variables in a data set except DATE, one per observation

To create a macro variable containing a blank-separated list of all variable names in data set ONE, change the PROC SQL step in Example 6 to the following. The macro variable created in this step could not be used in an array (as in Example 6) because arrays cannot contain both numeric and character variables, but could have other uses.

```
proc sql noprint;
  select name
    into :all_vars separated by ' '
    from dictionary.columns
    where libname="WORK"
      and memname="FIVE";
quit;
```

To create a macro variable containing a blank-separated list of the names of all numeric variables in data set ONE except DATE, change the PROC SQL step in Example 6 to the following. This would be useful if (for example) we wanted to process all the numeric variables in a data set except DATE in an array. Since the case of variable names can vary, NAME is converted to upper case and compared to “DATE”.

```
proc sql noprint;
  select name
    into :numeric_vars separated by ' '
    from dictionary.columns
    where libname="WORK"
      and memname="FIVE"
      and type = "num"
      and upcase(name) ne "DATE";
quit;
```

To create data set NUMNAMES containing the variable NAME with the names of all numeric variables except DATE, one per observation, change the PROC SQL step in Example 6 to the following.

```
proc sql noprint;
  create table numnames as
  select name
    from dictionary.columns
    where libname="WORK"
      and memname="FIVE"
      and type = "num"
      and upcase(name) ne "DATE";
quit;
```

Data set NUMNAMES has 3 observations, as follows.

```
Obs    name

 1     a
 2     bbb
 3     c1234567890
```

CONCLUSION

This paper showed some alternate ways to build lists of data set values and variable names for use outside of a DATA step. The paper contained very simple examples, but it is easy to imagine uses that are much more complex.

For more information, contact

Bruce Gilsen
Federal Reserve Board, Mail Stop 157
Washington, DC 20551
phone: 202-452-2494
e-mail: bruce.gilsen@frb.gov

REFERENCES

Long, Stuart, and Ed Heaton. "Using the SAS DATA Step and PROC SQL to Create Macro Arrays," *Proceedings of the Twentieth Annual NorthEast SAS Users Group Conference*, November 2007. <http://www.nesug.org/proceedings/nesug07/cc/cc27.pdf>.

SAS Institute Inc. (2004), "*Base SAS 9.1.3 Procedures Guide*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Macro Language: Reference*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Language Reference: Concepts*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Language Reference: Dictionary, Volumes 1, 2, and 3*," Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The following people contributed extensively to the development of this paper: Heidi Markovitz and Donna Hill at the Federal Reserve Board and Michelle Buchecker at SAS Institute. Their support is greatly appreciated.

TRADEMARK INFORMATION