USING MACRO VARIABLE LISTS

Presented at the Pittsburgh SAS User's Group, November 19, 1987
Ken Whitaker
Duquesne Systems Inc.

ABSTRACT

A macro variable list is a SAS macro variable that contains a list of words. Although these lists are only a special use of SAS macro variables, they can be very powerful. Users can parameterize a system with them, and the system can drive itself by extracting the words from the list, invoking macros using each word as a parameter, and by generating other lists for itself. This paper describes how macro variable lists can be used, and supplies several useful SAS macros that build and use lists.

INTRODUCTION

Duquesne Systems Inc. has developed a job accounting and chargeback system in SAS called the Billing Database Facility (BDBF). This is a very flexible and dynamic system that changes to accommodate the needs of the user.

For example, the starter set only processes SMF data (for jobs, TSO sessions, and started tasks). As the product is installed, the user adds other components as needed. If the user wants to bill for IMS, he copies three supplied members of a PDS into the base system, and BDBF starts processing IMS data, too. New data steps appear that process IMS data, new datasets are created, and all reporting and summarization processing automatically expand to include this new data source.

In fact, BDBF offers several different methods of processing IMS data:

* BDBF can read the IMS log tape with its own processor.
* BDBF can extract IMS data from the Performance Database Facility (another product of Duquesne Systems).
* BDBF can extract data from MXG (Merrill’s Expanded Guide to CPE, a product of Merrill Consultants).
* Users can write their own BDBF components. Users have written components that use IMF (IMS Measurement Facility, by Boole and Babbage) their own IMS log tape processors, and can extract IMS data from MICS (a product of Morino Associates).

It makes no difference to BDBF, even though each approach produces different datasets in different formats with different dataset names and variable names. BDBF modifies itself automatically.

All reporting and summarization code is driven with macro variable lists. As the word IMS is added to the file list, new reports appear, new files are saved to tape, more files are summarized, etc. The system is fundamentally changed.

Many BDBF parameters are specified via macro variable lists. Some of these are used simply as variable keep lists. Others have a more profound effect. For example, users enter a list of summary variables in a macro variable. If a user adds a new variable (for instance, SHIFT), the summarization process is changed and new reports begin to appear showing how charges are distributed over the new summary variable.

This paper describes some of the inner workings of BDBF so these techniques may be applied to other systems. It will discuss how macro variable lists can be used to control the expansion of macros, how lists can be built dynamically from data encountered in SAS datasets, and how some fairly impressive special effects can be produced with surprisingly little effort. Code for some useful list manipulation macros and some examples of coding techniques are included.

COUNTING AND EXTRACTING WORDS

A macro variable list is a SAS macro variable that contains a list of words. If the list is only a list of variable names that are to be inserted into a keep list, then the entire macro variable can be routinely substituted into a keep statement. However, as they are used here, it is necessary to be able to extract individual words and count the number of words.

For example, if FILELIST is a list of data sources, such as:

```
%LET FILELIST=JOB TSO IMS;
```
and we must invoke a macro to calculate charges on each data source, then we want to generate calls such as these:

%BILL(JOB);
%BILL(TSO);
%BILL(IMS);

To accomplish this, we must be able to extract each word from the list.

As another example, suppose SUMLIST is a list of summary variables, such as:

%LET SUMLIST=SYSTEM SHIFT;

and we want to summarize a file by SYSTEM and SHIFT, then we may want to generate code like this:

BY SYSTEM SHIFT;
IF FIRST.SHIFT THEN DO;
...
IF LAST.SHIFT THEN DO;

To accomplish this, we must be able to count the words in the list, and extract the last word in the list.

The SAS macro function SCAN is perfect for this job. It can return the first word, the second word, and so on, and it can be used repeatedly to count the number of words in the list.

Here is the macro function that BDBF uses to count words in a list:

%MACRO WORDCNT(LIST);
%LOCAL COUNT;
%LET COUNT = 0;
%IF &LIST HE THEN %DO;
%LET WORD = %SCAN(&LIST,1);
%DO %WHILE (&WORD NE);
%LET COUNT = %EVAL(&COUNT+1);
%LET WORD = %SCAN(&LIST,&COUNT+1);
%END;
%END;
&COUNT
%MEND;

As you can see, this macro extracts each successive word in the list until it has extracted the last word, and counts the number of words it found.

A purist might call this macro inefficient. We are making SAS scan the list for the first word, then scan the list from the beginning to the second word, then scan the list again up to the third word, etc. However, remember that this work is done only during compilation, not during execution for each record read.

Note that this is a macro function. SAS offers many macro functions such as %SUBSTR and %SCAN. The above macro can also be used as a function. For example, it can be used like this:

%DO I=1 %TO %WORDCNT(&LIST);
...
%END;

It is easy to write a macro that behaves like a macro function. Most macros generate lines of SAS code. This macro generates only a number, and there is no semicolon generated. As SAS parses this %DO statement, it will expand the macro %WORDCNT with a parameter of &LIST and will replace %WORDCNT(&LIST) with the generated number.

So, to generate the three %BILL statements given in the above example, code the following:

%DO I=1 %TO %WORDCNT(&FILELIST);
%BILL(%SCAN(&FILELIST, &I));
%END;

And, to generate FIRST.SHIFT and LAST.SHIFT as in the other example above, code:

%LET WORD=%SCAN(&SUMLIST, %WORDCNT(&SUMLIST));
IF FIRST.&WORD THEN DO;
...
IF LAST.&WORD THEN DO;
ELIMINATING DUPLICATE WORDS

Duplicates frequently arise from a variety of causes. Users may inadvertently enter the same word twice in a parameter list. Also, as the system is generating a list automatically, a list with duplicates can be the result.

For example, BDBF offers a different list of summary variables for each data source. A user may want to summarize job data by shift and project, but may only want to summarize DASD space usage by project (not by shift). When BDBF makes a list of all summary variables by concatenating lists from all data sources, project will appear twice.

Duplicates can be harmful for several reasons. If the same variable is listed twice in a keep statement, SAS flags one of them as never having been initialized. It is also inefficient to calculate charges for the same file twice, and incorrect to show job charges twice on each invoice.

The next example defines macro UNIQUE, which eliminates duplicate words from a list. This macro communicates with the rest of the system via global variables. Its input is passed in global macrovariable LIST. Macro UNIQUE builds a temporary list of unique words, then replaces the global variable LIST with this result. As a by-product, it also sets a global variable COUNT to the number of words in the new list.

```sas
%MACRO UNIQUE;
%GLOBAL LIST COUNT;
%LOCAL I,
%LET UNIQUE=,
%LET UCOUNT=0;
%DO I=1 %TO %WORDCNT(&LIST);
%LET VAR=%SCAN(&LIST,&I):
%IF &UCOUNT=0 %THEN %DO;
%LET UNIQUE=&VAR;
%LET UCOUNT=1;
%END;
%ELSE %DO;
%DO J=1 %TO &UCOUNT;
%IF &VAR NE %SCAN(&UNIQUE,&J) %THEN %LET VAR=;
%END;
%IF &VAR NE %THEN %DO;
%LET UNIQUE=UNIQUE &VAR;
%LET UCOUNT=%EVAL(&UCOUNT+1);
%END;
%END;
%LET LIST=&UNIQUE;
%LET COUNT=&UCOUNT;
%MEND;
```

A fairly elegant special effect was produced with this macro. BDBF produces an invoice that shows charges by data source, that is, it shows job charges in one section, followed by TSO charges in another, etc. These sections are usually printed in alpha order, that is, if a user had only jobs, TSO, and CICS, CICS would appear first. Some users requested the ability to define the order of the sections.

We did not want to make the user have to supply a complete list of sections in the desired order. Some users do not care about this feature, and we did not want to make them enter a list. Also, if a user installed another component, the invoice program would not automatically pick it up. The user would have to remember to add new data sources to this invoice list.

The solution was to allow the interested user to specify the order of sections. BDBF appends to this list a list of all installed components (that appears in alpha order), then it eliminates duplicates. The results are:

* Uninterested users do not need to enter a list. They see all components in alpha order.
* Users can name one or two important data sources, and they appear first. Remaining sections appear next, in alpha order.
* Users can name a complete list of data sources in any order, but if they add a new component and forget to modify this list, the new data source will appear on the invoice, although at the end.
BUILDING LISTS

Many lists are automatically built. BDBF builds lists of installed components, lists of SAS datasets, lists of variables in a dataset, etc. These lists are built by scanning data files and calling SYMPUT from a data step. This means that lists vary according to the data encountered, which in turn means the system behaves differently from day to day and from user to user.

For example, BDBF stores the billing data for jobs from day 1 in a summary dataset called JOB001. Jobs from day 2 go into a dataset called JOB002. If a user adds the IMS component on day 3, then processing on day 3 will produce datasets JOB003 and IMS003. There will be no datasets called IMS001 and IMS002. BDBF keeps its own directory dataset containing the dataset names it builds, and can build a macro variable list containing each of these dataset names.

The next example is macro SETLIST. It does a SET against a BDBF directory dataset and builds a macro variable list. INPUT is the name of the directory dataset. VAR is the variable in the dataset to add to the list. The list might contain dataset names like JOB001, or only file names like JOB. TEST can restrict the contents by selecting certain files, such as only JOB files, or only files from day 3. The results are returned in global macro variables LIST and COUNT.

%MACRO SETLIST(INPUT,VAR,TEST=);
%GLOBAL LIST COUNT;
%LOCAL I;
DATA _NULL_
LENGTH STRING $200;
RETAIN STRING ' ';
RETAIN COUNT 0 I 1;
SET &INPUT END=EOF:
%IF %STR(&TEST) NE %STR() %THEN %00;
  IF &TEST THEN 00;
RUN;
END;
COUNT=COUNT+1;
IF LENGTH(STRING) + 1 + LENGTH( VAR) > 200 THEN DO;
  I=I+1;
  STRING=' ';
END;
IF STRING='' THEN STRING=&VAR;
ELSE STRING=TRIM(STRING) ** ' ' ** &VAR;
%IF %STR(&TEST) NE %STR() %THEN %00;
END;
%ENO;
IF EOF THEN DO;
  CALL SYMPUT('COUNT',PUT(COUNT,3.));
  CALL SYMPUT('I',LEFT(PUT(I,3.)));
  CALL SYMPUT('L' •• TRIM(LEFT(PUT(I,3.,)), TRIM(STRING));
END;
%LET LIST=;
%DO %WHILE(&I GT 0);
%LET LIST = &&L&I &LIST;
%LET I=%EVAL(&I-1);
%END;
%MEND;

A complication is that SAS character variables have a maximum length of 200, while the resulting macro variables might have a length much longer than that. The solution was to generate smaller macro variables called L1, L2, etc, and then to combine these temporary macro variables into the final macro variable outside the data step.

For example, in order to construct a list containing all installed components, one might code this:

%SETLIST(SUMMARY.DIR, FILE);
%UNIQUE;

The %SETLIST macro might produce a list containing many duplicates. The %UNIQUE macro reduces this list to a list of installed components.

This macro would help if a list had to be saved in a SAS dataset for subsequent runs. This could become complicated if the list can be longer than 200 characters. The individual words can be saved in a SAS dataset with one word per observation. Then the list can easily be reconstructed using macro %SETLIST.
This code could save a list:

```sas
%MACRO LISTSAVE;
  DATA DD.LIST;
  KEEP WORD;
  %DO I=1 %TO %WORDCNT(&LIST);
    WORD=%SCAN(&LIST, &I);
    OUTPUT;
  %END;
RUN;
%MEND;
%LISTSAVE;
```

This code would restore the list in a later SAS job:

```sas
%SETLIST (DD.LIST, WORD);
```

**BUILDING LISTS WITH %INCLUDE**

Another method of building potentially large lists uses the %INCLUDE feature. This feature allows a SAS program to generate SAS code, then include and run that generated code. This feature has been available in SAS since its early releases. In fact, it was the only way to accomplish some of these tasks before the %MACRO feature was added. The %INCLUDE feature is not as widely used now, because the %MACRO feature is so much more powerful. However, it still has its place.

The next example is macro VARLIST. This macro generates several macro variable lists simultaneously. It is more efficient to use the %INCLUDE feature than to call macro SETLIST many times. The %INCLUDE feature also makes some tasks that could be done with the macro facility a little easier to accomplish.

The input is a BDBF parameter dataset that contains all the resources that the user has selected for each file. (Different users charge for different things.) Each observation has a FILE variable that contains a data source name, like CICS or JOB, and a four digit resource code that identifies a billable resource like CPU time.

For example, if the user is only billing for jobs and CICS, and is only charging three resources, the input dataset might look like this:

<table>
<thead>
<tr>
<th>FILE</th>
<th>RESOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB</td>
<td>1300</td>
</tr>
<tr>
<td>JOB</td>
<td>1600</td>
</tr>
<tr>
<td>CICS</td>
<td>2010</td>
</tr>
</tbody>
</table>

Then this macro creates these macro variables:

```sas
&NJOB=2
&QJOB=Q1300 Q1600
&CJOB=C1300 C1600
&NCICS=1
&QCICS=Q2010
&CCICS=C2010
```

&NJOB is the number of billable resources for jobs. This is used as a count variable with the next two macro variable lists. &QJOB is a list of BDBF quantity variables. For example, Q1300 is a variable containing the number of CPU seconds that were consumed. &CJOB is a list of BDBF cost variables. For example, C1300 contains the cost of the CPU time represented in Q1300. Similarly, these three macro variables are produced for each installed component.

```sas
%MACRO VARLIST (DD);
DATA NULL;
ARRAY RCSAVE $4 RCSAV1-RCSAV99;
RETAIN CTR RCSAV1-RCSAV99;
LENGTH WORD $5;
SET &DD..PARMS;
BY FILE;
IF FIRST.FILE THEN CTR=1;
ELSE CTR+1;
```

1535
The macro reads the dataset and retains the resource codes for a file in an array. At the end of a file, it generates some statements to the INCLUDE file. First, it generates `%GLOBAL declarations. Then it generates `%LET statements for each macro variable. There is no 200 byte limit here, because words are put to the INCLUDE dataset one by one, and statements can flow from one line to the next.

This could be accomplished without the `%INCLUDE feature, but would be more difficult to code, and would require many more data steps and multiple passes of the input file.

A complication is that the macro variables are required to be globals, and there is no data step function like SYMPUT that can do this. It could be done without the `%INCLUDE feature, however. If a macro variable called FILE had the value JOB (which could be done via SYMPUT), then this macro statement would define the global:

```
%GLOBAL FILE;
```

But the variable must be defined as a global before it is filled in with a SYMPUT call. This means that two data steps are required. There must be a data step that can determine that the first file is JOB and call SYMPUT to define the variable FILE. Then the `%GLOBAL statement can be executed. Only then can the other data step run that fills &JOBFILE with another SYMPUT. Allowing a single data step to write the `%GLOBAL and `%LET statements is much simpler.