Choosing the Best Way to Store and Manipulate Lists in SAS®

Dmitry Rozhetskin, Clinovo Inc., Sunnyvale, CA

ABSTRACT

A list is defined as a sequence of elements. One of the most interesting examples of a list is the sequence of character strings. In SAS®, there are different methods to store lists. We will demonstrate in this paper some of the most popular methods of storing lists, including records in a dataset, multiple macro variables, or a single macro variable.

We will compare the advantages and disadvantages of each method and provide the guidance on how to choose the most efficient method. Techniques for conversion among different storage methods will also be presented, allowing easy transition and thus proper selection of the list storage.

INTRODUCTION

At the core of any computer program are data and processes. When it comes to data, several abstract data models are recognized in computer science (Aho, 1995). They include graphs, trees, relations, lists, sets, etc. Once the abstract data model is selected, different programming languages offer various possibilities of its implementation. For example, in a popular C language, abstract lists can be implemented as linked lists, double-linked lists, or arrays (not to be confused with SAS arrays). For example, SAS dataset is an appropriate implementation for the relation data model underlying databases

SAS is not a traditional programming language. Yet, we can borrow from computer science certain definitions related to abstract lists. Those definitions describe some basic features of lists as well as operations on them. Once this is done, it is a natural step to consider how abstract lists and operations on them can be implemented in SAS. Finally, we will compare the implementations.

DEFINITION OF LISTS

A *list* is defined as a finite sequence of elements of a certain type. Here we drop the requirement of the predefined type and only consider lists of character strings:

$(a_1, a_2, a_3, \ldots, a_n)$

- where n is the *length* of the list, which is defined as a number of elements in the list

- where a_i is a character string, such as 'June', which may include alphanumeric characters.

Associated with each element is the concept of *position*, which is a number that uniquely identifies the element. In principle, position can be thought of a memory address, discreet coordinate, row number in a table, etc. It can be explicitly assigned to the corresponding element, or it can be implicit. System variable _n_ is an example of implicit specification, whereas the macro variable MVAR5 is an explicit specification of the element position 5.

For a list of strings, it is important to introduce *delimiters*, which we will define as a string which is not any part of the list. There are no universal delimiters, and they depend on the elements of the list. For example, a blank space is appropriate for a list consisting of single word names such as week days (Monday Tuesday Wednesday Thursday) but the lists of city names – such as (San Diego, San Francisco, Berkeley) – requires a different delimiter, which is composed by a comma and a blank space.

Different elements with different positions can have identical values.

REPRESENTATION OF LISTS IN SAS

Up to this point we have considered abstract lists. In everyday situations however, SAS programmers deal with specific objects that need to be described by the lists. How do we store them in SAS? While it is possible to imagine numerous ways of doing so, let us concentrate on the three most obvious and intuitive cases. Some of them were introduced earlier (Fehd, Carpenter, 2007). We will refer to any particular storage method as a *representation*. For example, if we store lists in a dataset, we will refer to this as a representation of lists by a dataset. We will describe below the three most obvious list representations in SAS.

The nature of the task may favor a particular representation. For example, when combining lists or applying procedures, lists stored as datasets might be favored, whereas when outputting lists to a report, the preferred method is likely to be a single macro variable.

A. USING DATASET

The first obvious choice is to use a single dataset with one variable.

```
data list;
length var $20;
infile datalines;
input var;
datalines;
January
February
March;
run;
```

Elements are represented as values of the variable. Position is represented implicitly as a row number in the dataset or as an automatic _n_ variable in the DATA step.

B. USING SINGLE MACRO VARIABLE

An alternative way is to put the list into a single macro variable &MLIST (ampersand & is not part of the macro variable name). For example,

```
%let mlist = January February March;
```

C. USING MULTIPLE (MACRO) VARIABLES

Finally, we can put the list into a sequence of macro variables. Below is an example of this storage method.

```
%let month1 = January;
%let month2 = February;
%let month3 = March;
```

It is worth mentioning that there are other ways to store this list. Here are some additional list representations in SAS:

- Delimited string in a dataset with a single variable and one observation. This is very similar to a single macro variable above (B).
- Dataset with a single observation and multiple variables
- Multiple datasets

DEFINITION OF BASIC OPERATIONS ON LISTS

While the decision of how to store lists can be dictated by how it will finally be used, it is also useful to look at other possible manipulations that the program will perform on the lists. Again, borrowing from the theory of abstract lists, let us introduce some basic operations. Table 1 below contains their definitions.

Two groups of operations are defined. The first group includes five operations that act on the entire list. The second group includes three operations involving individual elements (insert, delete, find). Each of these comes in several flavors depending on the specifications.

This list is somewhat arbitrary and by no means exhaustive. It serves the main purpose of illustrating our approach and can be easily extended. Other possible operations include:

- Merge by (Given two sorted lists, create a list containing elements from both lists that preserves the original order)
- Substitute (update) an element
- Reverse order

Table 1. Definition of basic operations on lists

#	OPERATION	GIVEN	RETURNS	DEFINITION
Operations on entire lists				
1	*Sort*	list	list	Given a list, return a sorted list with the same elements
2	*Find length*	list	number	Given a list, return its length
3	*Create a copy*	list	2 lists	Given a list, create an identical list
4	*Concatenate*	2 lists	list	Given two lists, create a list containing all the elements from the first list followed by all the elements from the second list
5	*Create a sublist*	list	list	Create a list containing only element from the original list or fewer, usually requires a rule
Operations on list elements				
6	Insert			
6.1	*Insert in the beginning*	list and element	list	Given an element, create a list starting with this element followed by the original list
6.2	*Insert at the end*	list and element	list	Given an element, create a list containing the original list followed by this element
6.3	*Insert at an arbitrary position*	list, element, and position	list	Given an arbitrary position and an element, insert this element at the specified position followed by the rest of the list
7	Delete			
7.1	*Delete first (head)*	list	list	Delete the first element of the list
7.2	*Delete last (tail)*	list	list	Delete the last element of the list
7.3	*Delete first occurrence*	list and element	list	Given an element, delete its first occurrence in the list
7.4	*Delete last occurrence*	list and element	list	Given an element, delete its last occurrence in the list
7.5	*Delete all occurrences*	list and element	list	Given an element, delete all of its occurrences in the list
7.6	*Delete by position*	list and position	list	Given a position, drop this element from the list, preserving the element that proceeded it and followed it in the original list
8	Find			
8.1	*Find by position*	list and position	element	Given a position, return an element at this position
8.2	*Find first*	list and element	position	Given an element, return the position of its first occurrence
8.3	*Find last*	list and element	position	Given an element, return the position of its last occurrence
8.4	*Find all*	list and element	position(s)	Given an element, return the position of all it occurrences

OPERATIONS ON LISTS IN SAS

Now that we have defined list operations, it is time to look at how they are implemented in SAS for each of our list representations in question. The following section contains the code for the most of the operations defined in the previous section. Additional examples can be found on line.

1. *Sort*

For lists represented by datasets, sorting is a truly elementary operation from the programmer's point of view. There is no reasonable justification to attempt to implement a sorting operation for the other two methods.

```
proc sort data=list out=list1;
by var;
run;
```

2. *Find length*

For the list represented by datasets we have:

```
proc sql;
select nobs
from dictionary.tables
where memname='LIST';
quit;
```

For the list represented by a single macro variable we have length of the list given by variable LEN:

```
data _null_;
```

```
x="&list";
len=count(x,', ')+1;
put len=;
run;
```

We assume that the programmer keeps track of the last element when lists are represented by the multiple macro variable, and thus the length of the list is *known*.

3. *Create a copy*

Copying one list into another is much less appealing in the case of multiple macro variables.

Dataset:

```
data list1;
      set list;
run;
```

Single macro variable:

```
%let list1 = &list;
```

Multiple macro variables:

```
macro copylist;
      %do i=1 %to 12;
            %global new&i;
            %let new&i = &&mvar&i;
      %end;
%mend copylist;
%copylist;
```

4. *Concatenate*

Again, the need to loop through each macro variables makes the code in the last case more cumbersome.

Dataset:

```
data list;
      set list1 list2;
run;
```

Single macro variable:

```
%let list12a = &list1, &list2;
```

Multiple macro variables:

```
%let mone1 = January;
%let mone2 = February;
%let mone3 = March;
%let mone4 = April;
%let mtwo1 = May;
%let mtwo2 = June
%let mtwo3 = July;
%macro concat;
      %do i=1 %to 4;
            %global mv&i;
            %let mv&i=&&mone&i;
      %end;
      %do i=1 %to 3;
            %local k;
            %let k=%eval(&i+4);
            %global mv&k;
            %let mv&k=&&mtwo&i;
      %end;
%mend concat;
%concat;
```

5. *Create a sublist*

Here the best candidate is the dataset method. The single macro variable method would involve significant parsing and cannot be regarded as a basic operation any longer.

Dataset:

```
data sublist;
      set list;
      where substr(var,1,1)='J';
run;
```

Multiple macro variables:

```
%let mvar1 = January;
…
%let mvar12 = December;

%macro sublist;
      %global k;
      %let k=1;
      %do i=1 %to 12;
            %global sub&k;
            %if %eval(%substr(&&mvar&i,1,1)=J) %then %do;
                  %let sub&k = &&mvar&i;
                  %let k=%eval(&k+1);
            %end;
      %end;
%mend sublist;
%sublist;
```

6. *Insert*

Let 's consider the operation of inserting a new element in the beginning of the list.

Dataset:

```
data list;
      input var $20.;
      infile datalines;
      datalines;
      February
      March
      ;
run;
data list;
      if _n_=1 then do;
      var='January';
      output;
      end;
      set list;
      output;
run;
```

Single macro variable:

```
%let newelement = January;
%let list = February, March;
%let list = &newelement, &list;
```

Multiple macro variables:

```
%let mvar1 = February;
%let mvar2 = March;
%let mvar3 = April;
%let mvar4 = May;
%macro insertbegin;
      %do i=4 %to 1 %by -1;
            %let k=%eval(&i+1);
            %global mvar&k;
            %let mvar&k = &&mvar&i;
      %end;
```

```
      %let mvar1=January;
%mend insertbegin;
%insertbegin;
```

7. *Delete*

Similarly to the insert operation, below is the code to delete the first element of a list. In this case, the most elegant solution is through the use of datasets.

Dataset:

```
data list;
      set list;
      if _n_=1 then delete;
run;
```

Single macro variable:

```
%let list = January, February, March, April, May, June, July, August,
September, October, November, December;
data _null_;
      x="&list";
      y=substr(x,index(x,', ')+2);
      call symput('list1',y);
run;
%put &list1;
```

Multiple macro variables:

```
%let mvar1 = January;
%let mvar2 = February;
%let mvar3 = March;
%let mvar4 = April;
%macro deletefirst;
      %do i=1 %to 3;
            %let k=%eval(&i+1);
            %let mvar&i = &&mvar&k;
      %end;
      %symdel mvar4;
%mend deletefirst;
%deletefirst;
```

8. *Find*

Finding the last element in the list will be our last example of a basic list operation. Since we are looking for the last occurrence of an element, it is implied that there are more than two identical elements contained in the list. Accordingly, examples below present these non-trivial cases. Note that the second list has three occurrences of the element ‘Monday’ and the third list has two occurrences of the element ‘July.’

Dataset:

```
data findlast;
      set list end=last;
      retain lastposn;
      if var='Tuesday' then lastposn=_n_;
      if last then put lastposn=;
run;
```

Single macro variable:

```
%let list = Monday, Tuesday, Wednesday, Monday, Tuesday, Monday,
Wednesday;
data _null_;
      x="&list";
      lastpos=count(substr(x,1,find(x,'Monday',-length(x))),', ')+1;
      put lastpos=;
run;
```

Multiple macro variables:

```
%let mvar1 = January;
```

```
%let mvar2 = February;
%let mvar3 = July;
…
%let mvar7 = July;
…
%let mvar12 = December;
%macro findlast;
      %global posn;
      %let posn=0;
      %do i=1 %to 12;
            %if &&mvar&i=July %then %do;
            %let posn=&i;
            %end;
      %end;
%mend findlast;
%findlast;
%put posn=&posn;
```

CONVERSION OF ONE REPRESENTATION INTO ANOTHER

As can be seen from the previous section, it is not always equally easy or even possible to implement the basic list operations described above using one of the three representations.

The solution would be to convert one of the representations into another. As we are looking at three possible storage methods, the corresponding total number of conversions among these methods equals six as described below.

A->B. FROM DATASET TO SINGLE MACRO VARIABLE

PROC SQL provides a convenient solution in the form of INTO clause of the SELECT statement. The name following the semicolon after INTO keyword specifies the macro variable name that will be used to store the list. The SEPARATED BY argument of the INTO clause generates delimiters between the different elements of the list. The following example creates a list separated by a comma and a blank space and saves it in the MLIST macro variable.

```
data list;
      length var $20.;
      infile datalines;
      input var;
      datalines;
      January
      February
      March
      April
run;

proc sql noprint;
      select var into: mlist separated by ', '
      from list;
quit;
```

B->A. FROM SINGLE MACRO VARIABLE TO DATASET

The inverse conversion utilizes DATA STEP and interfaces with macro facility provided by SYMGET function. For instance for the list created in the previous example, we can use the code below to reconstruct an original dataset.

```
data list;
      length var $20;
      varlist=symget('mlist');
```

```
        do until (var=' ');
                i+1;
                var=scan(varlist,i);
                if var ne '' then output;
        end;
        drop i varlist;
run;
```

The programmer needs to specify the length of the variable in the dataset to avoid truncation. The use of SCAN function is not universal. In addition, the programmer will have to substitute the SCAN function with a more involved combination of SUBSTR and FIND functions if the delimiters used by the list do not match those of SCAN function.

A->C. FROM DATASET TO MULTIPLE MACRO VARIABLES

The following example demonstrates how to put each list element stored in the dataset into a separate macro variable. The numeric suffix of the macro variable (e.g. 5 in MVAR5) corresponds to the row number in the dataset. The variable VAR from the dataset LIST contain the list elements.

```
data _null_;
        set list;
        call symput("mvar"||strip(put(_n_,8.)),var);
run;
```

C->A. FROM MULTIPLE MACRO VARIABLES TO DATASET

By introducing a DO loop in and with the help of SYMGET function we can achieve the opposite effect of going from the list of macro variables to a dataset.

```
data list;
        length var $20;
        do i=1 to 4;
                var=symget("mvar"||strip(put(i,8.)));
                output;
        end;
        drop i;
run;
```

B->C. FROM SINGLE MACRO VARIABLE TO MULTIPLE MACRO VARIABLES

This conversion is easier to perform using DATA step and parsing function such as SCAN. Again, the delimiters of the list need to be the delimiters of SCAN function.

```
.data list;
        length var $20;
        varlist="&mlist";
        do until (var=' ');
                i+1;
                var=scan(varlist,i);
                put var=;
                output;
                call symput("mvar"||strip(put(i,8.)),var);
        end;
run;
```

C->B. FROM MULTIPLE MACRO VARIABLES TO SINGLE MACRO VARIABLE

Finally, by simply using the %LET statement inside %DO loop, we can produce a list in the form of a single macro variable based on the of multiple macro variables.

```
%%macro toSingleMVar;
	%symdel mlist;
	%global mlist;
	%let mlist = %sysfunc(strip(&mvar1));
	%do i=2 %to 4;
		%let mlist = &mlist.%str(; )%sysfunc(strip(&&mvar&i));
	%end;
%mend;
%toSingleMVar;
```

COMPARISON OF REPRESENTATIONS

Finally, let us briefly point out some of the advantages and disadvantages for our three list representations. For dataset representation, one advantage is that all the techniques of combining datasets are available. Also, a dataset can be an input into SAS procedures. Among the disadvantages, it is worth mentioning specification of length which leads to additional disk space.

For single macro variable lists, one advantage is that the entire list can be accessed with one variable, which can make code very compact. The second advantage is efficient storage of long lists. The two disadvantages that can be singled out are the maximum length of 65,534 characters and the need to deal with delimiters.

For lists represented by multiple macro variables, let us point out the following advantages – direct access to the element from its position and ability to execute macros conditionally for each element. Disadvantages include looping even for simple operations, the need to maintain the position numbering. In general, the code that includes multiple macro variables tend to be longer than in the other two cases.

CONCLUSION

Decision of how to store lists in a SAS program is determined not only by how it will be finally used, but also how it will be transformed and manipulated in the program. In choosing the best method to do so, programmers can employ two simple approaches. In the first approach, they can try to anticipate as many operations as possible and then assess the corresponding difficulty. In the second approach, the lists can be converted from one representation to another as the need becomes apparent during coding. Overall, either approach can be used systematically as a programming design tool.

REFERENCES

Alfred Aho, Jeffrey Ullman. *Foundations of computer science* .W.H. Freeman and Company (1995)

Ronald J. Fehd, Art Carpenter. *List processing basics: creating and using lists of macro variables* (NESUG2007)

Arthur Carpenter. *Storing and using a list of values in a macro variable* (SUGI30)

Jerry Tsai. *Efficiently Handle Lists of Items using the SAS Macro Language* (WUSS2008)

ACKNOWLEDGMENTS

I would like to express special thank to I-Ling Hsiue for the fruitful discussions of the conversion techniques

CONTACT INFORMATION

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

Name:	Dmitry Rozhetskin
Enterprise:	Clinovo Inc.
Address:	1208 Arques Ave
City, State ZIP:	Sunnyvale, CA, 94085
Work Phone:	(408) 773-6251
E-mail:	drozhe@yahoo.com