A Macro to Generate All Combinations of *n* Items Taken *k* at a Time

(how it was designed and written, and how it might be adapted to other uses: a programming example)

by Francis J. Kelley
jkelley@arches.uga.edu

Abstract.
Developing a program, especially one that may be used several times or in ways not originally considered, tends to be a step-by-step process. One begins with the specific problem to be solved and proceeds to the more general. The stopping point often may not be known in advance; indeed, the successive refinements usually arise from a new or unanticipated need. The program, as developed in this paper, closely follows an actual problem involving combinatorics the macro was used to resolve. It cannot be said to be a programming model (although it did have two characteristics desired by programmers: it did the job(s) desired and it was fun to write). This poster describes how it was developed, written and expanded; as such, it is a "programming example."

Introduction.
It is not uncommon for a researcher to need to examine some or all possible combinations of some (seemingly) arbitrary string of numbers or letters or words. These individual elements might be variables, subjects, test instruments or the like. When the order of the elements in the string is important (for example, if the string *a b c* is **not** the same as *c b a*), such a "combination" is more properly called a "permutation" and is not the subject of this paper. When the order is not important (that is, when the strings *a b c*, *b c a*, *c a b*, etc. are regarded as the same), the term "combination" is correctly applied (in the mathematical sense of the term), and producing these combinations is the topic.

The number of ways any subset of the elements may be combined is referred to as the "number of *n* elements taken *k* at a time" and is described as:

$$^{n}C_{k} = n! / [(n-k)!k!]$$

So, if selecting 3 out of 5 elements (n=5 and k=3), there are: 5! / 2!3! = 10 possible combinations.

Solving the equation is easy using SAS® or other software. It is somewhat more difficult to list the individual combinations, though with thought, this too may be done. Two problems remain: generalizing the code so it will generate combinations of 3 from 9, 4 from 20, 5 from 10 and so on; and applying the (admittedly) stilted example to a somewhat more realistic problem.

Designing the code.
Before writing the first line of code, before concocting test data, before opening our copy of SAS Language: Reference we should consider what is to be produced. For example, if we wish all combinations of 2 out of 3, we will get 1-2, 1-3, and 2-3. If we wish all combinations of 3 out of 4 we will have:

1-2-3
1-2-4
1-3-4
2-3-4

The listing above starts sequentially (1-2-3) and ends sequentially (2-3-4) and uses a kind of lexicographic indexing (1-2-4 follows 1-2-3 and so on) to increment. A looping structure of some sort seems best for this, we just need to be sure the loops start and stop in the right places. Here is a code fragment that might do it:

```
do i = 1 to 2 ;
   do j = 2 to 3 ;
      do k = 3 to 4 ;
         put i= j= k= ;
```

Actually, this won't work, but it is a start. What will be printed will be

```
i=1 j=2 k=3  *
i=1 j=2 k=4  *
i=1 j=3 k=3
i=1 j=3 k=4  *
i=2 j=2 k=3
i=2 j=2 k=4
i=2 j=3 k=3
i=2 j=3 k=4  *
```

The (*) indicates the desired values; the rest are incorrect. The problem is that the loop indices overlap: for example, note that it is possible for both j and k to equal 3 at the same time. A solution is to use variables instead of "hard-coded" values as the starting points for the loops:

```
do i = 1 to 2 ;
   do j = (i + 1) to 3 ;
      do k = (j + 1) to 4 ;
         put i= j= k= ;
```

This time the results are:

```
i=1 j=2 k=3
i=1 j=2 k=4
i=1 j=3 k=4
i=2 j=3 k=4
```

Which, of course, is exactly what is wanted.

This solves the '3 out of 4" case, but suppose the next problem is "2 out of 4", or "3 out of 5", or "5 out of 9", or (basically) something similar but not identical to the problem just solved? Each problem may be solved using ordinary DATA step programming techniques; in the case of "5 out of 9" this will require 5 loops with the maximum value of the last loop 9, the next-to-last loop 8, the "one before that" 7 and so on. This is not really difficult, but it is tedious and error-prone; besides, enough is known about the problem to solve it in a more general way. The "more general way" is to write a macro to handle the problem.

The COMB Macro.
A macro is needed in this case because the variable number of loops imply writing a variable amount of SAS source code, and the Macro language does exactly this. The macro code must also set the starting and stopping values for each loop, but it is known the final loop must have the maximum value. In the general case of *n* items taken *k* at a time, there must be *k* loops and the *k* th loop (the innermost) must have a maximum value of *n*. The *k*-1 loop must have a maximum value of *n*-1, and so on. We see the *k*-j loop must have a maximum value of *n*-j. It is also known that first loop (the outermost) begins its index at 1 and all subsequent loop indices will have a starting point that is 1 greater than the current index of the preceding loop (in the example above, when the index value I is 1, the index value J will be I+1 = 2) . Because the loops must be "built" from the outermost to innermost, the maximum index value of the outermost loop must be assigned first. This is easy; it is (*n* - *k*) + 1 . We now have the number of loops to be used and the starting and stopping points for each loop. With this, there is sufficient information to write the macro. A brief word first: macro programming styles vary from programmer to programmer. The author attempts to distinguish SAS Macro Language statements from SAS statements by indenting the latter considerably. Even so, the similarity of the two languages and the fact that macro variables often are used in SAS programming statements, can make reading this code difficult especially for those unfamiliar with the Macro language. Explanations are provided where possible. Note that this and other macros will make use of LOCAL variables. This is done to avoid the possibility of one macro overwriting a variable used by another. Though not really necessary in this example, this practice can make later modifications much easier.

```
COMB.MAC

%macro comb(dsout=_NULL_,n=,k=);

  %local  dsout  n  k  start  stop  i ;

  %let start = 1 ;
    %let stop  = %eval( &n - &k) ;
```

Initialize the Macro variables START to 1 and STOP to n - k (a 1 will be added to STOP below).

```
               data &dsout ;
               retain j 1 ;
```

The DATA step variable J will be a counter of how many combinations are produced. Please note that in the program statements below, only the "`do var&i =` " is DATA step code; the other statements (all beginning with "%") are macro code. As a side note, the "%(" and "%)" are necessary to retain the parenthesis in the macro variable "&START".

```
    %do i = 1 %to &k;
       %let stop  = %eval(&stop  + 1) ;
               do var&i = &start to &stop ;
       %let start = %str(%(var&i + 1 %)) ;
    %end ;
```

The algorithm requires that *k* loops be executed (or, for the macro, generated). A macro variable, I, will be the index for the macro loop, but note that it writes the code for the data step loops (whose indices, starting and stopping points are - or include - macro variables). If an output dataset were desired, it should be specified via the DSOUT parameter in the macro call (the current default is _NULL_). Next, replace `put _ALL_` with `output` to write the results to the dataset named in DSOUT.

```
               put _ALL_ ;
               j + 1 ;

  %do i = 1 %to &k ;
           end ;
  %end ;
```

Since *k* loops were generated above, k "END; " statements must be generated now.

```
           run ;

%mend  comb ;
```

Using the MPRINT Option, the code generated can be displayed (regrettably, without indentation).
Here are the results using n=9 and k=4:

```
35         %comb(n=9,k=4)
MPRINT(COMB):   DATA _NULL_ ;
MPRINT(COMB):   RETAIN J 1 ;
MPRINT(COMB):   DO VAR1 = 1 TO 6 ;
MPRINT(COMB):   DO VAR2 = (VAR1 + 1 ) TO 7 ;
MPRINT(COMB):   DO VAR3 = (VAR2 + 1 ) TO 8 ;
MPRINT(COMB):   DO VAR4 = (VAR3 + 1 ) TO 9 ;
MPRINT(COMB):   PUT _ALL_ ;
MPRINT(COMB):   J + 1 ;
MPRINT(COMB):   END ;
MPRINT(COMB):   END ;
MPRINT(COMB):   END ;
MPRINT(COMB):   END ;
MPRINT(COMB):   RUN;
```

A selected portion of the output is reproduced below:

```
J=1 VAR1=1 VAR2=2 VAR3=3 VAR4=4
J=2 VAR1=1 VAR2=2 VAR3=3 VAR4=5
J=3 VAR1=1 VAR2=2 VAR3=3 VAR4=6
J=4 VAR1=1 VAR2=2 VAR3=3 VAR4=7
J=5 VAR1=1 VAR2=2 VAR3=3 VAR4=8
J=6 VAR1=1 VAR2=2 VAR3=3 VAR4=9
J=7 VAR1=1 VAR2=2 VAR3=4 VAR4=5
J=8 VAR1=1 VAR2=2 VAR3=4 VAR4=6
J=9 VAR1=1 VAR2=2 VAR3=4 VAR4=7
J=10 VAR1=1 VAR2=2 VAR3=4 VAR4=8
J=11 VAR1=1 VAR2=2 VAR3=4 VAR4=9
J=12 VAR1=1 VAR2=2 VAR3=5 VAR4=6
    ......
J=124 VAR1=5 VAR2=6 VAR3=8 VAR4=9
J=125 VAR1=5 VAR2=7 VAR3=8 VAR4=9
J=126 VAR1=6 VAR2=7 VAR3=8 VAR4=9
```

Expanding the Macro.
The COMB macro is interesting and solves several problems, but at the end, one may ask how this solves any problem that is not simply a set of numbers. If, for example, the combinations desired are not numbers, but letters, names or words (or discontinuous numbers: 1, 2, 3, 100, 301, 302, and so on), how might this macro help then? There are probably several ways this might be solved, but two methods come quickly to mind, and both are based upon the assumption that the sequence numbers generated (the values in VAR1, VAR2, etc.) are correct: change the way the numbers are displayed (this could be done with a user-defined format) or use the numbers as subscripts for an array of values. The author has chosen the latter method as it seems the more flexible. Now consider what is (or may reasonably be) known at the start of the program as well as what is desired as output. For input, all that is known is that there is some number of “names” (and we will let SAS tell us how many there are - no need to count/miscount ourselves), and combinations of 4 (for this example) are wanted. The additional “features” could be incorporated into the COMB macro, but that will weigh it down somewhat and require a bit more information be passed in the macro call than the author cares to do. A new macro, SETUP, will be written to get the data used into a form that COMB can use, then call COMB using parmameters already specified or computed. Some modifications to COMB are needed, though: it must now read an already-existing SAS dataset (created with SETUP). A usually-subtle problem may show up when working with character strings: length. The default length for character variables in SAS is 8 and many procedures (in particular, older versions of Proc FREQ) will not display more than 16 (this is not the case in versions 6.09E and 6.12). SETUP will check for the largest length it can find and pass that to COMB as well.

SETUP will need the name of the variable being used and the number of names to be in the combination. Optionally, the dataset containing the names to be used and an output dataset may be passed as well, however defaults are used (output is to _NULL_ as before, and the “most-recently-created” - &SYSLAST - is the default input).

The SETUP Macro.
SETUP assumes one-name-per-observation on the input dataset (DSIN). The input dataset is passed to Proc FREQ which produces an output dataset containing all values of the variable being used (by default, the output dataset will

report the values of “name” in alphabetical order). A modest “error check” is made here for duplicate names: if found, the name and the number of occurrences is written to the Log. The value of *n* is also found here by simply counting the number of unique names in the output dataset - this is assigned to the macro variable TOTVAR and will become the N used in COMB. A more serious error check is made to verify that the number of elements requested in each combination (the *k*) is not greater than the total number of elements available (the *n*). The “warning” printed will appear in the same way other warnings (or errors) might in a SAS program Log, with appropriate colors, etc. The author regrets the `%GOTO` , but it is the fastest way out of such a situation. The branch will go to the end of the SETUP Macro.

```
SETUP.MAC

%macro setup(dsout=_NULL_,dsin=&syslast,var=,ks=) ;

  %local dsout  dsin  var totvar  ks ;

           proc freq   data=&dsin     noprint ;
             tables  &var   / out=__count ;

           data _NULL_   ;
             set __count   end=last ;
             retain len oldlen 0 ;
             nk + 1 ;
             len = length(&var) ;
             if ( len > oldlen ) then oldlen = len ;
             if count > 1 then do;
               put “WARNING: The variable %upcase(&var)”   /
                   “WARNING: has “ count “ occurrences of the value “
        var;
             end ;
           if last then do ;
              if &ks > nk then do;
                 put “ERROR:  K greater than the Number of values (N)” ;
                 put “ERROR:  Value of K: &ks    Value of N: “ nk ;
                 put “ERROR:  EXITING Macro!!!!!!!!!!!!!!!!!” ;
              end ;

              call symput(‘totvar’,trim(left(put(nk,5.))) ) ;
              call symput(‘length’,put(oldlen,3.) );
           end ;
           run ;

  %if &ks > &totvar %then %goto exit ;
```

The output dataset written by Proc FREQ will contain two variables: the name of the variable whose frequency is being counted and the COUNT of the occurrences of each value. The next step counts the number of unique values and reports if any value was found more than once (COUNT > 1). The macro variable TOTVAR, which has the number of unique values, is saved with all leading (and trailing) blanks removed. This is necessary because of the way it will be used in the next step. The macro variable LENGTH is found by successively comparing the lengths of each value of the “variable of interest” (VAR) and saving the maximum.

```
           data __d1 ;
             set __count end=last ;
             array _vals {&totvar} $ &length;
```

```
            retain _vals1-_vals&totvar ;
            keep _vals1-_vals&totvar ;
            _vals{_N_} = &var ;
            if last then output ;
```

This step will write the unique values found in FREQ to an array. The array values are preserved over successive iterations of the DATA step by using RETAIN. This is why TOTVAR had the leading blanks removed - had they been present, the specification “_vals&totvar” might have been resolved into something like “_vals   8” instead if “_vals8”. Only the final observation is Output; this is because only then have all the values been assigned. Note that “&length” is used in the ARRAY statement to prevent truncation of the character variable to 8.

```
            %comb(k=&ks,n=&totvar,len=&length)
            run ;

%exit: %mend  setup ;
```

As is already known, the COMB macro has been modified. Here is the new version:

```
%macro comb(dsout2=&dsout,dsin2=__d1,k=,n=,len=) ;

  %local dsin2 dsout2 numvar i start stop n k len ;
```

As before, there is an output dataset defined; this time it is passed from SETUP and is the dataset defined there (in COMB it is DSOUT2, while in SETUP it is DSOUT). An input dataset is also defined - it defaults to the dataset created by the last Data step in SETUP (__D1). As before, K and N are also passed, as is the length (LEN) of the longest character string. The input dataset has only a single observation, but contains the names of all the values to be used (as determined by Proc FREQ). Parts of the next steps are identical to the earlier version of COMB, but there have been some changes as well.

```
  %let start = 1 ;
  %let stop  = %eval( &n - &k) ;

          data &dsout2 ;
            set &dsin2 ;
            array _vals {&n} $ ;
            array combo {&k} $ &len ;
            keep j combo1 - combo&k ;
            retain j 1 ;
```

The _VALS array contains N elements, while the COMBO array will contain K elements. A counter value, J, is initialized to 1.

```
  %do i = 1 %to &k;
     %let stop  = %eval(&stop  + 1) ;
            do var&i = &start to &stop ;
     %let start = %str(%(var&i + 1 %)) ;
  %end ;
```

The code above will write the *k* loops needed to generate the values that are assigned to VAR&I. Since VAR&I is just a number, it can be used as a subscript in the code below.

```
  %do i = 1 %to &k ;
            combo{&i} = _vals{var&i} ;
  %end ;
```

Now the assignment of K values is done, the results Output, and the counter incremented.

```
             output ;
             j + 1 ;

  %do i = 1 %to &k ;
             end ;
  %end ;

           run;

%mend  comb;
```

Here is a program that will call the macros:

```
    data test ;
      infile cards ;
      input name $ 1-80 ;
    cards;
    John
    Paul
    George
    Ringo
    Arthur
    Jack
    A_long_name
    Hortense
    ;

    %setup(var=name,ks=4,dsout=combin)
    proc print  data=combin ;
```

Using the MPRINT Option, the SAS code generated is displayed (I omit some portions):

```
    107        %setup(var=name,ks=4,dsout=combin)
     MPRINT(SETUP):   PROC FREQ DATA=WORK.TEST NOPRINT ;
     MPRINT(SETUP):   TABLES NAME / OUT=__COUNT ;
     MPRINT(SETUP):   DATA _NULL_ ;
     MPRINT(SETUP):   SET __COUNT END=LAST ;
     MPRINT(SETUP):   RETAIN LEN OLDLEN 0 ;
     MPRINT(SETUP):   NK + 1 ;
     MPRINT(SETUP):   LEN = LENGTH(NAME) ;
     MPRINT(SETUP):   IF (LEN > OLDLEN) THEN OLDLEN = LEN ;
     MPRINT(SETUP):   IF COUNT > 1 THEN DO;
     MPRINT(SETUP):   PUT "WARNING:               [remainder omitted]
     MPRINT(SETUP):   END ;
     MPRINT(SETUP):   IF LAST THEN DO ;
     MPRINT(SETUP):   IF &KS > NK THEN DO;
     MPRINT(SETUP):   PUT "ERROR:                 [remainder omitted]
     MPRINT(SETUP):   END;
     MPRINT(SETUP):   CALL SYMPUT('totvar',TRIM(LEFT(PUT(NK,5.))) ) ;
     MPRINT(SETUP):   CALL SYMPUT('length',PUT(OLDLEN,3.) ) ;
     MPRINT(SETUP):   END ;
     MPRINT(SETUP):   RUN ;
```

```
MPRINT(SETUP):   DATA __D1 ;
MPRINT(SETUP):   SET __COUNT END=LAST ;
MPRINT(SETUP):   ARRAY _VALS {8} $ 11;
MPRINT(SETUP):   RETAIN _VALS1-_VALS8 ;
MPRINT(SETUP):   KEEP _VALS1-_VALS8 ;
MPRINT(SETUP):   _VALS{_N_} = NAME ;
MPRINT(SETUP):   IF LAST THEN OUTPUT ;
```

At this point, the COMB macro is called:

```
MPRINT(COMB):   DATA COMBIN ;
MPRINT(COMB):   SET __D1 ;
MPRINT(COMB):   ARRAY _VALS {8} $ ;
MPRINT(COMB):   ARRAY COMBO {4} $ 11 ;
MPRINT(COMB):   KEEP J COMBO1 - COMBO4 ;
MPRINT(COMB):   RETAIN J 1 ;
MPRINT(COMB):   DO VAR1 = 1 TO 5 ;
MPRINT(COMB):   DO VAR2 = (VAR1 + 1 ) TO 6 ;
MPRINT(COMB):   DO VAR3 = (VAR2 + 1 ) TO 7 ;
MPRINT(COMB):   DO VAR4 = (VAR3 + 1 ) TO 8 ;
MPRINT(COMB):   COMBO{1} = _VALS{VAR1} ;
MPRINT(COMB):   COMBO{2} = _VALS{VAR2} ;
MPRINT(COMB):   COMBO{3} = _VALS{VAR3} ;
MPRINT(COMB):   COMBO{4} = _VALS{VAR4} ;
MPRINT(COMB):   OUTPUT ;
MPRINT(COMB):   J + 1 ;
MPRINT(COMB):   END ;
MPRINT(COMB):   END ;
MPRINT(COMB):   END ;
MPRINT(COMB):   END ;
MPRINT(COMB):   RUN;
```

Here is some selected output:

```
COMBO1         COMBO2      COMBO3      COMBO4       J

A_long_name    Arthur      George      Hortense     1
A_long_name    Arthur      George      Jack         2
A_long_name    Arthur      George      John         3
A_long_name    Arthur      George      Paul         4
A_long_name    Arthur      George      Ringo        5
A_long_name    Arthur      Hortense    Jack         6
```

Conclusion.
Developing the original (DATA step) solution required some analysis of the problem, specifically focused upon how the results could be organized and how that organization might be developed. The author concedes that long-ago work in Fortran was very helpful. The next refinement, generalizing from “3 from 5” to “*k* from *n*”, was very macro-language specific. The important part of this step was writing code to generate a variable number of loops in the resulting SAS code. The final step, moving from sequential numbers to character strings as output, required use of “SAS programming” - that is, using DATA and PROC steps to set up the data in a form that could be used. Although this last step required writing a second macro, that macro was appropriately named: SETUP, as it organized the data and collected information from it. The revisions to COMB were more “Fortran-ish” modifications.

Writing a SAS program, then modifying and/or “macro-izing” it can be a complex task. In this instance, the problem was a simple one, though this will often not be the case. The SAS Macro language can be intimidating, but planning the development in stages, reviewing what will be used and what is to be produced as well as some very simple stylistic conventions can make it easier.

Acknowledgements.
Continuing thanks to Jeannie McElhannon for her wonderful help in making sense of my manuscripts, and to my collegues at UCNS who put up with me when I am “writing.”

References.

SAS Institute, SAS Language: Reference, Version 6, First Edition Cary, NC: SAS Institute, Inc, 1990.
SAS Institute, SAS Macro Language: Reference, First Edition Cary, NC: SAS Institute, Inc, 1997.

Author contact: email: jkelley@arches.uga.edu