

# RECURSIVE SUBROUTINE MACROS

Chang Y. Chung  
Princeton University

## ABSTRACT

I present a set of macros to make it easier to write recursive, subroutine-like programming constructs within a data step. Several problems posted on SAS-L are solved demonstrating the use of these macros.

## INTRODUCTION

There are several ways to solve a difficult programming problem. Sometimes a problem must be divided into smaller, manageable pieces and sometimes a recursive approach must be taken. Since SAS® data step does not allow user-written subroutines or functions, this language does not fully support either method.

Instead, `link` and `return` statements must be used. These statements help users code a common block of statements once and “call” them from different places. Data step also permits a recursive `link` which makes it possible to `link` to itself from within a `link` block, but recursive links are limited to only ten-level deep. Parameterizing a `link` block is also quite awkward, further limiting the usefulness of `link` and `return` statements.

To combat this problem some clever users have developed and used subroutine-like programming constructs using `goto` statements and temporary arrays. With this approach, it is possible to have “parameterized” blocks of statements that could be recursively called more than ten times. It is almost like having recursive subroutines inside a data step.

Written mostly to solve specific problems, this approach has not been well-documented out of particular contexts. By encapsulating the approach in a set of macros, this paper aims to document and facilitate application.

In the next section, the technique is explained in plain data step code. Then, a set of macros, called %sub macros, are presented. It is followed by several examples using the macros to solve some problems, many of which have their origins in the postings on the on-line discussion group, SAS-L.

## GOTO WITH TEMPORARY ARRAY

There are two innovative ideas that gave rise to the approach. One is the re-discovery of the versatility of `goto` statement. Suppose that we are to recursively calculate the factorial of a positive integer. If it is less than 11, it is possible to code it with `link` and `return` as below (on the left). The compatible `goto` approach is on the right.

Notice that we get an error message when we attempt to calculate 11! using the `link` and `return`. This is because we are `linking` deeper than 10-levels. The data step on the right column shows that we can get around the limitation by using `goto`'s instead. The `link` and `return` are smarter than `goto` and `label`. When we `link` from multiple places in the code -- `Return` knows which `link` to go back to; `goto` does not. In fact, the `goto` approach above would not have worked, if the statement were not the last statement in the `fact` block. Fortunately, we can work around this problem in a straight-forward manner.

<pre>data _null_;  /* main *****/ f = 1; level = 11; link fact; put "11!=" f;</pre>	<pre>data _null_;  /* main *****/ f = 1; level = 11; goto fact; returnPoint;</pre>
---	--

```

stop;                                put "11!=" f;
                                        stop;

/* subroutine *****/
fact:;
  if level > 1 then do;
    f = level * f;
    level = level - 1;
    link fact;
  end;
return;

run;
/* on log
ERROR: More than 10 LINK statements have
been executed [...] Check your LINK/RETURN
logic.
f=39916800 level=1 _ERROR_=1 _N_=1
*/

/* subroutine *****/
fact:;
  if level > 1 then do;
    f = level * f;
    level = level - 1;
    goto fact;
  end;
goto returnPoint;

run;
/* on log
11!=39916800
*/

```

The second innovation is using (temporary) arrays in order to emulate the first-in-last-out (FILO) call stacks, which store the current values of "local" variables before the body of the "subroutine" is executed. It then restores those values when the call to the subroutine returns. Emulating push and pop operations is nothing more than keeping track of the cell index of the cell on the "top of the stack." Below (on the left) is the implementation of the factorial algorithm with goto's and temporary arrays. In order to show that the recursive call indeed returns, I added a put statement after the recursive call.

Each "call" of a subroutine now becomes a three step exercise. The first is setting the variable, rp, which tells which return point the control should come back to. The second is a goto statement sending the control to the "subroutine." The third is setting up the "call" specific return point. At the conclusion of an execution of the subroutine, a select-end block chooses an appropriate goto statement. This is the whole picture of how goto statements are used to mimic what link and return do.

```

data _null_;
/* initialization *****/
/* for a "local" variable level */
array level_stack[1:12] _temporary_;
top_of_level_stack = 0;

/* for return points */
array rp_stack[1:12] _temporary_;
top_of_rp_stack = 0;

/* main *****/
f = 1;
level = 11;
/* calling a subroutine */
rp = 1;
goto fact;
returnPoint1;

/* report when all done */
put "11!=" f;
stop;

/* subroutine *****/
fact:
/* push current values into stacks */
top_of_rp_stack + 1;
rp_stack[top_of_rp_stack] = rp;
top_of_level_stack + 1;
level_stack[top_of_level_stack] =
  level;

/* the content of subroutine */
if level > 1 then do;
  f = level * f;
  level = level - 1;

data _null_;
/* initialization *****/
%sub_init(fact, level)

/* main *****/
f = 1;
%sub_call(fact, 11)
put "11!=" f;
stop;

/* subroutine *****/
%sub_do(fact)
  if level > 1 then do;
    f = level * f;
    level = level - 1;
    %sub_call(fact, level)
    put level=;
  end;
%sub_end(fact)

run;
/* on log
level=1

```

```

        rp = 2;
        goto fact;
returnPoint2:;
        put level=; /* this is new */
        end;

/* pop current values from stacks */
rp = rp_stack[top_of_rp_stack];
top_of_rp_stack + (-1);
level =
        level_stack[top_of_level_stack];
top_of_level_stack + (-1);

/* goto back to appropriate return
point */
select(rp);
when(1) goto returnPoint1;
when(2) goto returnPoint2;
end;

run;
/* on log
level=1
level=2
...
level=10
11!=39916800
*/

```

```

level=2
...
level=10
11!=39916800
*/

```

On the right is the corresponding data step written with the %sub macros. There are four macros: %sub\_init, %sub\_call, %sub\_do, and %sub\_end. The first macro parameter is always the name of the subroutine, in this case "fact." The generated codes are very similar to what are shown on the left: %sub\_init sets up stacks; %sub\_call does the three step exercise of "calling" subroutines as mentioned above; %sub\_do pushes values onto the stacks; and %sub\_end pops the top of the stacks.

Note that there are no "local" variables, but rather we have call-stack managed variables. These variables may be assigned new values while the subroutine is executed; they get the old values back when a call returns.

The actual macro implementation has some additional complications like handling expressions passed as subroutine parameters (they have to be evaluated right after pushing), and prefixing variable names so that they are unlikely to collide with existing names.

## %SUB MACROS

The macro source is appended at the end of this paper. It consists of eighteen macros. In addition to four %sub macros proper, there are six general utility macros; four that deal with what I call NameTypeLen(gth) notation; three for implementing LIFO stacks; and one macro to demonstrate the %sub macro usage.

The utility macros are not an absolute necessity (it is possible to write macros without them), but they help make macro codes easier to read.

%iif macro is named after a worksheet function of a popular spreadsheet application program and works in the same way. That is, it returns the true part when the condition is evaluated true and the false part when false.

%forEach macro is a %do loop with bells and whistles. Depending on the unit, the macro makes it possible to adjust starting, ending, and increment (by) values. It also makes it possible to run the loop for each item in a delimited list given in the macro parameter (in). The body of the %do loop is given by invoke, where you are guaranteed to have a macro variable, &unit., available. For example, if you call %forEach with item then you will have a macro variable called item available for you during each invoke:

```
%macro putMe(what);
```

```

    %put ***&what.***;
%mend putMe;
%forEach(item, in=a b c, invoke=%nrstr(%putMe(&item.)))
/* on log
***a***
***b***
***c***
*/

```

The NameTypeLen(gth) (NTL) notation is used to efficiently express variable name, type, and length. In NTL notation, `f00` means a numeric type (named `f00`) whose length is 8. On the other hand, `bar$1`, indicates a character type, `bar`, with a length of 1. The NTL macros make it easy to deal with the NTL notation. For example:

```

%put ***%ntl_type(bar$1)***;
/* on log
***$***
*/

```

The stack macros rely on NTL. The `%stack_init` sets up a temporary array of given name, type, and length. It also creates a retained variable for pointing the top of the stack. The default value of `max` is 1000, which results in an array with 1001 cells, indexed from 0 to 1000. The zero cell is initialized to missing, indicating that the top of the stack is empty. The `%stack_pop` pops the top. It is written such a way that it can be used like a function in the right side of the equal sign in an assignment statement.

The following sets up a stack and pushes three letters on it. It then pops the top three times while putting the values on log. The name of the temporary array is: `bar_stack`. The pointer variable is called `bar_stack_top`. Notice that the first macro parameter of the `%stack_init` is an NTL, while it is name only for `%stack_push` and `%stack_pop`.

```

data _null_;
  %stack_init(bar$1)
  do bar = "A", "B", "C";
    %stack_push(bar, bar)
  end;
  bar = %stack_pop(bar);
  do while (not missing(bar));
    put bar=;
    bar = %stack_pop(bar); /* semi-colon required */
  end;
run;
/* on log
bar=C
bar=B
bar=A
WARNING: stack bar is empty already.
*/

```

Given plenty of helpers, the `%sub` macros alone are relatively short. `%sub_init` has two positional parameters that are required. The first is the name of the subroutine, the other is a delimited (by default – with a space) list of `args` or a list of variables to be call-stacked, written in NTL notation. In practice, the list will include both the subroutine parameters and the “local” variables. From the macro implementation point of view, they are the same – both are the call-stack managed variables. The `heap` parameter gives the size of the temporary arrays (minus one); that is, the maximum possible depth of recursive calls.

Once the stacks are set up, the macro also creates global macro variables in order to keep track of the argument names and return points. It also creates the return point data step variable to be used at the end of the subroutine. Note that all the names are prefixed by the subroutine name. This creates a sort of “name space” so that similar names do not collide across different subroutines. `%sub_init` is to be issued before any other `%sub` macros.

The `%sub_call` “calls” the subroutine. In addition, it stores the given subroutine call parameters (`values`) in a series of global macro variables. The items in `values` will end up being assigned to the variables that are

mentioned in `args` and created by `%sub_init`. Thus, the `value`'s should match in order with the `arg`'s in terms of type and length. A `value` can be an expression.

`%sub_call` can be used in many places including within the subroutine itself (recursive calls). It cannot, however, appear in the code before `%sub_init`. It also cannot be placed after `%sub_end`. Usually, this is not a problem, but if you have multiple subroutines that call each other, then the place of the `%sub_call` in the code matters. A useful hack in such instances is shown in one of the examples below.

`%sub_do` marks the beginning of the subroutine of the given `name`. The first thing it does is to write out a label with the same `name`. Then, it pushes the current value of each call-stack managed variables. After that, it passes control to the end of the subroutine, where the subroutine call parameters (`values`) are evaluated and assigned. The control then comes back to the body of the subroutine.

`%sub_end` marks the end of the subroutine body. It does pop the stacks, but it also writes out two `select-end` blocks. One is to select the return points. The other is the parameter evaluation block that `%sub_do` passes the control to and pro.

That is it for the explanation of `%sub` macros. In the following several sections, I present some examples of using `%sub` macros. The source code for the examples can be found in the appendix, in the macro called, `%sub_test`.

## GOD JUL OR MERRY CHRISTMAS

This example (test case number 2) is from Wahlgren (2003)'s question about generating a certain permutation involving six different letters. If the permutations are generated in a certain order, the 194<sup>th</sup> permutation forms "GODJUL," which is "Merry Christmas!" in Swedish (once we put a space between the letters D and J). McLerran (2003) posted an elegant recursive macro solution (the URL of the archived posting is given in the reference).

```

%*-- test2 translated from macro permute                --*;
%*-- by Dale McLerran (sas-l 2003-12-15)              --*;

data _null_;

/* initialization *****/
%sub_init(permute, c$6 chars$6 word$6 len i)

/* main *****/
counter = 0;
%sub_call(permute, "" "DGJLOU" "" 6 1)
put counter=;
stop;

/* subroutine *****/
%sub_do(permute)

    if missing(chars) then len = 0;
    else len = length(chars);

    if len > 0 then do;
        i = 1;
        do while (i <= len); /* you cannot call a sub within a */
            /* do i=## to ## since to-expression cannot be altered.*/
            %sub_call(permute, %str(
                substr(chars,i,1)          # /* c */
                trimn(compress(chars, c)) # /* chars */
                trimn(trimn(word) || c)    # /* word */
                len                         # /* len */
                i                           # /* i */
            ), dlm=#)
            i = i + 1;
        end;
    end; else do;
        counter ++ 1;
        if counter = 194 then put "The 194th word is " word;
    end;

```

```

        %sub_end(permute)

run;
/* on log
The 194th word is GODJUL
counter=720
NOTE: DATA statement used (Total process time):
      real time           0.09 seconds
      user cpu time       0.06 seconds
      system cpu time     0.03 seconds
      Memory              275k
*/

```

I translated the macro solution to %sub macro implementation. The code demonstrates how and what to include in the args for %sub\_init and values for %sub\_call. The %sub\_init shows that the first three arguments are of character type with the same length of six. Both the "local" variables len and i are also to be call-stack managed, since their values should be restored when the recursive call returns. The second %sub\_call demonstrates that you can specify expressions instead of variables.

It has been well-known that we can implement recursive algorithms in macro. The main drawbacks, however, have been the slow execution (or macro compilation) speed and the limited size of the macro buffer. The %sub macro implementation, in contrast, should run faster since it is a data step. Note that it should not be affected by the size of the macro buffer (except in an extreme case).

As a side note, the do while loop inside the subroutine cannot be replaced with an ordinary do loop (i.e., do i = 1 to len;). If we do, then the len is evaluated only once before looping. This is due to the optimization that the data step compiler does.

## TWO STEP FORWARD AND ONE STEP BACK

The next example (test case number 3) is trivial, except that it shows how to use two subroutines together. Notice that %sub\_call cannot appear after the corresponding %sub\_end. This example also reminds us that there are no real "local" variables. That is the reason why we cannot have the call-stack managed variables that have the same name, even though they will be "local"ized in different subroutines. Otherwise, using two or more subroutines is straightforward.

```

%*-- test3 a silly one, involving two sub-routines          --*;

data _null_;

/* initialization *****/
%sub_init(addTwo,      a)
%sub_init(subtractOne, b) /* you cannot use a again */

/* main *****/
x = 0;
do while (x <= 10);
  put x=;
  %sub_call(addTwo, x)
  %sub_call(subtractOne, x);
end;
stop;

/* subroutines *****/

%sub_do(addTwo)
  %sub_call(subtractOne, %str(x+3))
%sub_end(addTwo)

%sub_do(subtractOne)
  x = b - 1;
  /* you cannot call addTwo here, since
  %sub_end(addTwo) would not know
  anything about the call. See test 7 in the
  below for an workaround when two sub-routines

```

```

        call each other
        */
        %sub_end(subtractOne)

run;
/* on log
x=0
x=1
x=2
x=3
x=4
x=5
x=6
x=7
x=8
x=9
x=10
*/

```

## “IT WOULD BE DIFFICULT TO DO THIS WITH IF-THEN”

In response to Flom's (2002) SAS-L posting saying that `goto` and `link` are redundant given `if-then`, Hamilton(2002) and Crawford(2002) pointed out that `link` is the only way to set the next observation from more than one point in the code.

Using `%sub` macros is obviously another way (as shown in the test case 4). In the same thread, Whitlock(2002) emphasized that `link` facilitated better organization of large and complex data step. This virtue applies to `%sub` macros as well.

```

%*-- test4. "it would be difficult to do this with IF-THEN"      --*;
%*-- from Jack Hamilton (sas-l 2002-07-24)                      --*;

/* create a test dataset */
data one;
  do i = 1 to 10;
    output;
  end;
run;

data two;

/* initialization *****/
%sub_init(getIt)

/* main *****/
if not end then do;
  %sub_call(getIt)
  put "just got one obs. " i;
end;

  put / "...doing something else..." /;

if not end then do;
  %sub_call(getIt)
  put "got another obs. " i;
end;

if end then stop; else return;

/* subroutine *****/
%sub_do(getIt)
  set one end=end;
%sub_end(getIt)

run;
/* on log
just got one obs. i=1

```

```

...doing something else...

got another obs.  i=2
just got one obs. i=3

...doing something else...

got another obs.  i=4
just got one obs. i=5

...doing something else...

got another obs.  i=6
just got one obs. i=7

...doing something else...

got another obs.  i=8
just got one obs. i=9

...doing something else...

got another obs.  i=10
*/

```

## PSEUDO-RECURSIVE SAS MACRO

Benjamin Jr.(1999)'s article in *Observations* is the oldest reference I found about doing the recursive subroutines in a data step. In fact, the article was mentioned multiple times in SAS-L, whenever the needs arose to defeat the false claim that it is impossible to do recursion in a data step. The article does, however, stop a bit short of crystallizing the general programming technique separated from the specific problem.

Once translated into %sub macro implementation (test case number 5), the code clearly shows that at the heart of his solution lies the recursion which can be easily understood. The code also demonstrates that the %sub macros are not limited to the data \_null\_ steps.

```

%*-- test5. Benjamin Jr's Observations 18 article          --*;
%*-- "pseudo-recursive sas macro"                        --*;
%*-- http://www.sas.com/service/library/periodicals/obs/  --*;
%*--  obswww18/index.html                                --*;
%*-- a little bit modified so that input is a dataset, also --*;
%*-- uses different variable names.                      --*;

/* create master and varlist datasets and define a utility macro */
data master(index=(var));
  array cards[13] $30 (
    "AA 6 BB CC DD EE FF GG"
    "BB 2 GG HH"
    "CC 3 EE FF HH"
    "DD 3 FF GG HH"
    "EE 0"
    "FF 1 MM"
    "GG 0"
    "HH 1 II"
    "II 1 JJ"
    "JJ 1 KK"
    "KK 1 LL"
    "LL 1 MM"
    "MM 0"
  );
do _n_ = lbound(cards) to hbound(cards);
  var = scan(cards[_n_], 1);
  count = input(scan(cards[_n_], 2), best.);
  array d[1:6] $2 d1-d6;
  do i = 3 to 8;
    d[i-2] = scan(cards[_n_], i);
  end;

```

```

        output;
        keep var count d1-d6;
    end;
run;

data varlist; /* this is the input dataset */
    do v = "BB", "EE", "FF";
        output;
    end;
run;

%macro c2i(cc );index("ABCDEFGHIJKLM",substr(&cc.,1,1))%mend;

data dep_list;

/* initialization *****/
%sub_init(depend, var$2 count d1$6 d2$6 d3$6 d4$6 d5$6 d6$6 i);
%sub_init(writeModules);

/* main *****/
array modules[1:13] $2 _temporary_;
array d[1:6]          $2 d1-d6;

set varlist end=end;
%sub_call(depend, v 0 "" "" "" "" "" "" 0)
if end then do;
    put "Requires: " @;
    %sub_call(writeModules);
end;
return;

/* subroutines *****/
%sub_do(depend)
    modules[%c2i(v)] = v;
    %sub_call(writeModules)
    set master key=var/unique; /* aka get_mstr */
    i = 1;
    do while (i <= count);
        modules[%c2i(d[i])] = d[i];
        %sub_call(depend, d[i] count d1 d2 d3 d4 d5 d6 i)
        i + 1;
    end;
%sub_end(depend)

%sub_do(writeModules)
    do i = 1 to 13;
        if not missing(modules[i]) then put modules[i] $3. @;
    end;
    put;
%sub_end(writeModules)

run;
/* on log
BB
BB GG
BB GG HH
BB GG HH II
BB GG HH II JJ
BB GG HH II JJ KK
BB GG HH II JJ KK LL
BB GG HH II JJ KK LL MM
BB EE GG HH II JJ KK LL MM
BB EE FF GG HH II JJ KK LL MM
BB EE FF GG HH II JJ KK LL MM
Requires: BB EE FF GG HH II JJ KK LL MM
*/

```

On a side note, the data step, `master`, is a bit more complicated than necessary. This is due to the fact that you cannot embed data lines within a macro and I was trying to put all the examples together in a macro, `%sub_test`.

## “EFFICIENT, WELL STRUCTURED, CLEAR SAS CODE. CLASSIC.”

DeVenezia(2003) solved a geographical adjacency problem originally posted by Stratton(2003). He used plain data step code that utilized exactly the same technique discussed here. DeVenezia's code was so well written, Dorfman(2003) commented on it. In part, he wrote, “Efficient, well structured, clear SAS code. Classic.”

In the %sub macro implementation (test case number 7), I define a subroutine (claimColumn) within another (claimRow), since all the %sub\_call's should come before the corresponding %sub\_end. In a case like this, it is necessary to hide the subroutine definition between if 0 then do; and end; so that the subroutine is not executed without a proper call.

```
%*-- test7. Geographical Adjacency problem                --*;
%*-- Robert Stratton's problem (sas-l 2003-08-05)         --*;
%*-- Richard DeVenezia's solution (sas-l 2003-08-05)     --*;
%*-- NOTE: Dorfman fully endorses Richard's original solution: --*;
%*-- "Efficient, well structured, clear SAS code. Classic. --*;
%*-- very aesthetic, too. (Dorfman. sas-l 2003-08-05)"    --*;

/* Stratton's data */
data groups;
  cards = "1 A 1 B 1 C 2 D 3 A 3 F 4 H 4 I 4 J 4 K";
  keep set id;
  do i = 1 to 10;
    set = input(scan(cards, 2*(i-1) + 1),1.);
    id = scan(cards, 2*i);
    output;
  end;
run;
proc print data=groups(obs=10);
run;
/* on lst
Obs    set    id
  1     1     A
  2     1     B
  3     1     C
  4     2     D
  5     3     A
  6     3     F
  7     4     H
  8     4     I
  9     4     J
 10     4     K
*/

/* the solution translated with %sub_ macros */
proc sql;
  reset noprint;
  select count (distinct set) into :nSet from groups;
quit;

data _null_;

/* initialization *****/
%sub_init(putS)
%sub_init(claimRow , ri rj)
%sub_init(claimColumn, ci cj)

/* main *****/
array S[&nSet.,26] _temporary_;

do while (not eog);
  set groups end=eog;
  S[set,rank(id)-64] = -1; /* for ascii box only -- cyc */
end;

put "before: ";
%sub_call(putS)
ss = 0;
do i = 1 to &nSet.;
  do j = 1 to 26;
```



I provide a few more examples in the macro %sub\_test (See Appendix).

## SUMMARY

"Divide and conquer" remains a simple but effective way of solving complex problems. SAS data step as a language does not explicitly support user written subroutines or functions, making it difficult to organize complicated and lengthy code. `link` and `return` permit recursive calls, but with very limited depth.

In this paper, I have explained the clever programming technique of using `goto`'s and temporary arrays as subroutine calls and stacks to circumvent these limitations. I have also encapsulated the technique in a set of macros called %sub macros, so that it is more easily accessible.

True to the purpose, the %sub macros themselves are written in such a way that the whole solution consists of smaller and re-usable utility macros.

This paper also demonstrated the usage of the %sub macros by implementing solutions to interesting problems, including those posted on SAS-L.

## REFERENCES

- Benjamin Jr., William E. (1999). "A Pseudo-Recursive SAS Macro." *Observations – The Technical Journal for SAS Software Users* No. obswww18. Archived at <http://support.sas.com/documentation/periodicals/obs/obswww18/index.html>.
- Crawford, Peter (2002) "Re: GOTO and LINK vs IF THEN DO." A SAS-L posting on Jul. 24, 2002. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0207D&L=sas-l&D=0&H=0&O=T&T=1&m=106219&P=26935>.
- DeVenezia, Richard A. (2003). "Re: List manipulation." A SAS-L posting on Aug 5, 2003. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0308A&L=sas-l&D=0&H=0&O=T&T=1&m=106219&P=28208>. Also available at his web site (with a javascript animation showing the recursive algorithm <http://www.devenezia.com/downloads/sas/samples/index.html>) at <http://www.devenezia.com/downloads/sas/samples/superset.sas>.
- Dorfman, Paul (2003). "Re: List manipulation." A SAS-L posting on Aug. 5, 2003. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0308A&L=sas-l&D=0&H=0&O=T&T=1&m=106219&P=29016>.
- Flom, Peter L. (2002). "GOTO and LINK vs IF THEN DO." A SAS-L posting on Jul. 24, 2002. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0207D&L=sas-l&D=0&H=0&O=T&T=1&m=106219&P=24665>.
- Hamilton, Jack (2002). "Re: GOTO and LINK vs IF THEN DO." A SAS-L postin on Jul. 24, 2002. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0207D&L=sas-l&D=0&H=0&O=T&T=1&m=106219&P=27680>.
- McLerran, Dale (2003). "Re: Permutation problem." A SAS-L posting on Dec. 15, 2003. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0312C&L=sas-l&P=R1885&D=0&H=0&O=T&T=1>.
- Stratton, Robert (2003). "List manipulation." A SAS-L posting on Aug. 5, 2003. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0308A&L=sas-l&D=0&H=0&O=T&T=1&m=106219&P=28208>.
- Wahlgren, Lars (2003). "Permutation problem." A SAS-L posting on Dec. 14, 2003. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0312B&L=sas-l&P=R26216&D=0&H=0&O=T&T=1>.
- Whitlock, Ian (2002). "Re: GOTO and LINK vs IF THEN DO" A SAS-L posting on Jul. 24, 2002. Archived at <http://listserv.uga.edu/cgi-bin/wa?A2=ind0207D&L=sas-l&D=0&H=0&O=T&T=1&m=106219&P=28195>.

## NECESSARY REMARK

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.

## **ACKNOWLEDGEMENTS**

I would like to thank all the SAS-L discussion group participants, from whose company I benefit greatly.

## **AUTHOR CONTACT INFORMATION**

Chang Y. Chung  
Senior Statistical Programmer / Data Archivist  
Office of Population Research  
Princeton University  
#216 Wallace Hall  
Princeton, NJ 08540  
(609) 258 - 2360  
cchung@princeton.edu

## %SUB MACROS SOURCE CODE

```
%*-- sub.sas -----*;  
%*-- -----*;  
%*-- macros simplify doing (recursive) subroutines in a data step ---*;  
%*-- -----*;  
%*-- version 0.17 by chang y. chung on 2004-07-29 -----*;  
%*-- -----*;  
%*-- These macros are continuously improved. They are offered with---*;  
%*-- no guarantees what so ever. Use at your own risk. Comments ---*;  
%*-- are welcome. The latest version will be available for -----*;  
%*-- download at changchung.com. -----*;  
  
%*-- utilities ---*;  
%macro iif(cond, true, false);  
  %if %unquote(&cond.) %then %do;%unquote(&>true.)%end;  
  %else %do;%unquote(&>false.)%end;  
%mend;  
%macro increase(mvar, by=1);  
  %let &mvar. = %eval(&&mvar. + (&by.));  
%mend;  
%macro decrease(mvar, by=1);  
  %increase(&mvar., by=-&by.)  
%mend;  
%macro globalLet(mvar, value);  
  %global &mvar.;  
  %let &mvar.=&value.;  
%mend;  
%macro uniqueName(length=8, seed=0);  
  %*;%sysfunc(ceil(1e%eval(&length.-1)*%sysfunc(ranuni(&seed.))))  
%mend;  
  
%*-- NameTypeLen -----*;  
%*-- a numeric var should be name only: -----*;  
%*-- a char var should be followed by a dollar sign and length ---*;  
%macro ntl_name(nameTypeLen);  
  %*;%scan(&nameTypeLen., 1, $)  
%mend;  
%macro ntl_type(nameTypeLen, character=$, numeric=%str());  
  %*;%iif(%nrstr(%index(&nameTypeLen.,$)), &character., &numeric.)  
%mend;  
%macro ntl_len(nameTypeLen, numeric=%str());  
  %*;%iif(%nrstr(%ntl_type(&nameTypeLen., character=$)=$)  
  , %nrstr(%scan(&nameTypeLen., 2, $))  
  , &numeric.)  
%mend;  
%macro ntl_missing(nameTypeLen);  
  %*;%iif(%nrstr(%ntl_type(&nameTypeLen., character=$)=$), %str(" "),.)  
%mend;  
  
%*-- stack ---*;  
%macro stack_init(nameTypeLen, max=1000);  
  %local name dollar len;  
  %let name =%ntl_name(&nameTypeLen.);  
  %let dollar =%ntl_type(&nameTypeLen.);  
  %let len =%ntl_len (&nameTypeLen.);  
  array &name._stack [0:&max.] &dollar.&len. _temporary_;  
  &name._stack[0] = %ntl_missing(&nameTypeLen.);  
  retain &name._stack_top 0;  
%mend;  
%macro stack_push(name, item);  
  do;  
    &name._stack_top + 1;  
    if &name._stack_top > hbound(&name._stack) then do;  
      put "ERROR: stack &name. over-flow.";  
      stop;  
    end;  
    &name._stack[&name._stack_top]=&item.;  
  end;  
end;
```

```

%mend;
%macro stack_pop(name);
  %*&name._stack[&name._stack_top];
  do;
    &name._stack_top + (-1);
    if &name._stack_top < 0 then do;
      &name._stack_top = 0;
      put "WARNING: stack &name. is empty already.";
    end;
  end /* semicolon intentionally left out */
%mend;

%*-- forEach (limited version) by chang y chung on 2004-01-03   --*;
%*-- inspired by Peter Crawford's, %mLoopsX (sas-l 2002-11-20) --*;
%macro forEach(unit, in=, dlm=%str( ,), from=, to=, by=1,
  invoke=%nrstr(%put ***&&unit.***));

  %local &unit. units idx start finish before after inf;
  %let inf      = 32767;
  %let before  = 0;
  %let after   = 0;
  %let units   = item|number;
  %let unit    = %lowcase(&unit.);

%branch: /* Richard DeVenezia's "Dynamic Branching" */
  %if %index(|&units.|, |&unit.|) %then %goto &unit.;
  %else %do;
    %put ERROR: "&unit." is not supported;
    %goto out;
  %end;

%item:
  %if &in.= %then %do;
    %put WARNING: empty list in IN=; %goto out;
  %end;
  %let start   = 1;
  %let finish  = &inf.;
  %let before  = %nrstr(
    %let &unit. = %scan(&in., &idx., &dlm.);
    "%&&unit.." = "" /* true when %scan returns nothing */
  );
  %goto loop;

%number:
  %if &from.= or &to.= %then %do;
    %put ERROR: needs a number in both FROM= and TO=; %goto out;
  %end;
  %let start   = &from.;
  %let finish  = &to.;
  %let before  = %nrstr(
    %let &unit. = &idx.;
    0           /* always false */
  );
  %goto loop;

%loop:
  %do idx = %unquote(&start.) %to %unquote(&finish.);
    %if %unquote(&before.) %then %goto out;
    %*;%unquote(&invoke.)
    %if %unquote(&after.) %then %goto out;
    %let idx = %eval(&idx. - 1 + (&by.));
  %end;

%out:
%mend forEach;

%*-- subroutine macros by chang y chung on 2004-01-03   --*;
%macro sub_init(name, args, heap=1000, dlm=%str( ));
  %globalLet(&name._args_dlm, &dlm.)
  %forEach(item, in=&args., dlm=&dlm., invoke=%nrstr(
    %stack_init(&name._&item., max=&heap.)
    length %ntl_name(&item.)
    %ntl_type(&item.)%ntl_len(&item., numeric=8);
  ));

```

```

    %ntl_name(&item.) = %ntl_missing(&item.);
  ))
%globalLet(&name._args,
  %forEach(item, in=&args., dlm=&dlm., invoke=%nrstr(
    %*;%ntl_name(&item.)&dlm.
  )))
%stack_init(&name._rp, max=&heap.)
%globalLet(&name._rp, 0)
retain &name._rp 0;
%mend;
%macro sub_call(name, values, dlm=%str( ));
  %increase(&name._rp)
  %globalLet(&name._call&&&name._rp._values, %nrstr(&values.))
  %globalLet(&name._call&&&name._rp._values_dlm, &dlm.)
  do;
    &name._rp = &&&name._rp.;
    goto &name.;
    &name._rp&&&name._rp.;;
  end;
%mend;
%macro sub_do(name);
  &name.:
  %stack_push(&name._rp , &name._rp )
  %forEach(item, in=&&&name._args, dlm=&&&name._args_dlm., invoke=%nrstr(
    %stack_push(&name._&item., &item.)
  ))
  goto &name._values;
  &name._do:;
%mend;
%macro sub_end(name);
  &name._rp = %stack_pop(&name._rp);
  %forEach(item, in=&&&name._args., invoke=%nrstr(
    &item. = %stack_pop(&name._&item.);
  ))
  select(&name._rp);
  %forEach(number, from=1, to=&&&name._rp., invoke=%nrstr(
    when (&number.) goto &name._rp&number.;
  ))
  end;
  &name._values:;
  select (&name._rp);
  %forEach(number, from=1, to=&&&name._rp., invoke=%nrstr(
    when (&number.)
    do;
      %forEach(item, in=&&&name._call&number._values.,
        dlm=&&&name._call&number._values_dlm., invoke=%nrstr(
          %scan(&&&name._args., &idx., &&&name._args_dlm.) = &item;
        ))
      end;
    ))
  end;
  goto &name._do;
%mend;

%macro sub_test(what);

%let what=%upcase(&what.);
%if &what.= %then %let what=;

%*-- test1 -- calc factorial by multiplication --*;
%if &what.=_ALL_ or %index(&what.,1) %then %do;
data _null_;
  %sub_init(foo, m level)
  f = 1;
  %sub_call(foo, 12 1)
  put "12!=" f;
  stop;

%sub_do(foo)
  put level=;
  if m>1 then do;
    f = m*f;
    m + (-1);
    %sub_call(foo, m level+1)
  end;

```

```

        end;
    %sub_end(foo)
run;
/* on log
   12!=479001600
*/
%end;

%*-- test2 translated from macro permute          --*;
%*-- by Dale McLerran (sas-l 2003-12-15)         --*;
%if &what.=_ALL_ or %index(&what.,2) %then %do;
data _null_;
    %sub_init(permute, c$6 chars$6 word$6 len i)

    counter = 0;
    %sub_call(permute, "" "DGJLOU" "" 6 1)
    put counter=;
    stop;

    %sub_do(permute)
        if missing(chars) then len = 0;
        else len = length(chars);
        if len > 0 then do;
            i = 1;
            do while (i <= len); /* you cannot call a sub within a */
                /* do i=## to ## since to-expression cannot be altered. */
                %sub_call(permute, %str(
                    substr(chars,i,1)      # /* c      */
                    trimn(compress(chars, c)) # /* chars */
                    trimn(trimn(word) || c) # /* word  */
                    len                      # /* len   */
                    i                        # /* i     */
                ), dlm=#)
                i = i + 1;
            end;
        end; else do;
            counter ++ 1;
            if counter = 194 then put "The 194th word is " word;
        end;
    %sub_end(permute)
run;
%end;

%*-- test3 a silly one, involving two sub-routines --*;
%if &what.=_ALL_ or %index(&what.,3) %then %do;
data _null_;
    %sub_init(addTwo,      a)
    %sub_init(subtractOne, b) /* you cannot use a again */

    x = 0;
    do while (x <= 10);
        put x=;
        %sub_call(addTwo, x)
        %sub_call(subtractOne, x);
    end;

    stop;

    %sub_do(addTwo)
        %sub_call(subtractOne, %str(x+3))
    %sub_end(addTwo)
    %sub_do(subtractOne)
        x = b - 1; /* you cannot call addTwo here, since
                    %sub_end(addTwo) would not know
                    anything about the call. See test 7 in the
                    below for an workaround when two sub-routines
                    call each other
                    */
    %sub_end(subtractOne)
run;
%end;

%*-- test4. "it would be difficult to do this with IF-THEN" --*;
%*-- Jack Hamilton (sas-l 2002-07-24)             --*;
%if &what.=_ALL_ or %index(&what.,2) %then %do;

```

```

data one;
  do i = 1 to 10; output; end;
run;
data two;
  %sub_init(getIt)

  if not end then do;
    %sub_call(getIt)
    put "just got one obs. " i=;
  end;

  put / "...doing something else..." /;

  if not end then do;
    %sub_call(getIt)
    put "got another obs. " i=;
  end;

  if end then stop; else return;

  %sub_do(getIt)
  set one end=end;
  %sub_end(getIt)
run;
%end;

%*-- test5. Benjamin Jr^s Observations 18 article      --*;
%*-- "pseudo-recursive sas macro"                    --*;
%*-- http://www.sas.com/service/library/periodicals/obs/ --*;
%*--   obswww18/index.html                            --*;
%*-- a little bit modified so that input is a dataset, also --*;
%*-- uses different variable names.                   --*;
%if &what.=_ALL_ or %index(&what.,5) %then %do;
data master(index=(var));
  array cards[13] $30 (
    "AA 6 BB CC DD EE FF GG"
    "BB 2 GG HH"
    "CC 3 EE FF HH"
    "DD 3 FF GG HH"
    "EE 0"
    "FF 1 MM"
    "GG 0"
    "HH 1 II"
    "II 1 JJ"
    "JJ 1 KK"
    "KK 1 LL"
    "LL 1 MM"
    "MM 0"
  );
  do _n_ = lbound(cards) to hbound(cards);
    var = scan(cards[_n_], 1);
    count = input(scan(cards[_n_], 2), best.);
    array d[1:6] $2 d1-d6;
    do i = 3 to 8;
      d[i-2] = scan(cards[_n_], i);
    end;
    output;
    keep var count d1-d6;
  end;
run;

data varlist; /* this is the input dataset */
  do v = "BB", "EE", "FF"; output; end;
run;

%macro c2i(cc );index("ABCDEFGHIJKLM",substr(&c.,1,1))%mend;

data dep_list;

  /* init ----- */
  %sub_init(depend, var$2 count d1$6 d2$6 d3$6 d4$6 d5$6 d6$6 i);
  %sub_init(writeModules);

  array modules[1:13] $2 _temporary_;
  array d[1:6]          $2 d1-d6;

```

```

/* main ----- */
set varlist end=end;
%sub_call(depend, v 0 "" "" "" "" "" "" 0)
if end then do;
  put "Requires: " @;
  %sub_call(writeModules);
end;
return;

/* subs ----- */
%sub_do(depend)
  modules[%c2i(v)] = v;
  %sub_call(writeModules)
  set master key=var/unique; /* aka get_mstr */
  i = 1;
  do while (i <= count);
    modules[%c2i(d[i])] = d[i];
    %sub_call(depend, d[i] count d1 d2 d3 d4 d5 d6 i)
    i + 1;
  end;
%sub_end(depend)

%sub_do(writeModules)
  do i = 1 to 13;
    if not missing(modules[i]) then put modules[i] $3. @;
  end;
  put;
%sub_end(writeModules)
run;

/* we are looking for an output:
   Requires: BB EE FF GG HH II JJ KK LL MM
*/
%end;

%*-- test6. Euclid Algorithm to calculated GCD ---;
%*-- translated from Paul Dorfman^ macro gcd (sas-l 1999-12-20) ---;
%if &what._=ALL_ or %index(&what.,6) %then %do;

data _null_;

  %sub_init(getGcd, gcd ref)

  x = 2**15;
  y = 2**12;
  z = 2**10;
  put x= y= z=;
  %sub_call(getGcd, x y)
  gcd_xy = r;
  put "GCD(x,y)=" gcd_xy;
  %sub_call(getGcd, gcd_xy z)
  gcd_xyz = r;
  put "GCD(x,y,z)=" gcd_xyz;
  stop;

  %sub_do(getGcd)
  do while (ref > 0);
    res = mod(gcd, ref);
    gcd = ref;
    ref = res;
  end;
  r = gcd;
  %sub_end(getGcd)
run;
/* on log
x=32768 y=4096 z=1024
GCD(x,y)=4096
GCD(x,y,z)=1024
*/
%end;

%*-- test7. Geographical Adjacency problem ---;
%*-- Robert Stratton^s problem (sas-l 2003-08-05) ---;
%*-- Richard DeVenezia^s solution (sas-l 2003-08-05) ---;

```

```

%*-- NOTE: Dorfman fully endorses Richard's original solution:  --*;
%*--   "Efficient, well structured, clear SAS code. Classic.  --*;
%*--   very aesthetic, too. (Dorfman. sas-l 2003-08-05)"  --*;

%if &what.=_ALL_ or %index(&what.,7) %then %do;
/* Stratton's data */
data groups;
  cards = "1 A 1 B 1 C 2 D 3 A 3 F 4 H 4 I 4 J 4 K";
  keep set id;
  do i = 1 to 10;
    set = input(scan(cards, 2*(i-1) + 1),1.);
    id = scan(cards, 2*i);
    output;
  end;
run;
proc print data=groups(obs=10);
run;

/* the solution translated with %sub_ macros */
proc sql;
  reset noprint;
  select count (distinct set) into :nSet from groups;
quit;

data _null_;

  /* init ----- */
  %sub_init(putS)
  %sub_init(claimRow , ri rj)
  %sub_init(claimColumn, ci cj)

  /* main ----- */
  array S[&nSet.,26] _temporary_;

  do while (not eog);
    set groups end=eog;
    S[set, rank(id)-64] = -1; /* for ascii box only -- cyc */
  end;

  put "before: ";
  %sub_call(putS)
  ss = 0;
  do i = 1 to &nSet.;
    do j = 1 to 26;
      if S[i,j] = -1 then do;
        ss + 1;
        %sub_call(claimRow, i 0)
      end;
    end;
  end;
  put "after: ";
  %sub_call(putS)

  stop;

  /* subs ----- */
  %sub_do(claimRow)
  rj = 1;
  do until (rj > 26);
    if S[ri,rj] = -1 then do;
      s[ri,rj] = ss;
      %sub_call(claimColumn, 0 rj);
    end;
    rj + 1;
  end;
  if 0 then do; /* a hack by Dorfman. it is required since
                %sub_end(claimRow) has to come after
                the last call to it, which happens to be
                the inside of the claimColumn
                */
  %sub_do(claimColumn)
  ci = 1;
  do until (ci > &nSet.);
    if S[ci,cj] = -1 then do;
      S[ci,cj] = ss;
    end;
  end;

```

```

        %sub_call(claimRow, ci cj)
    end;
    ci + 1;
end;
%sub_end(claimColumn)
end;
%sub_end(claimRow)

%sub_do(putS)
    put +5 @; /* column header -- cyc */
    do jj = 1 to 26;
        letter = byte(64 + jj); /* for ascii box only -- cyc */
        put letter $2. +1 @;
    end;
    put; /* end column header */
    do ii = 1 to &nSet.;
        put ii 2. ' ' @;
        do jj = 1 to 26;
            put S[ii,jj] 2. + 1 @;
        end;
    end;
    put;
end;
%sub_end(putS)

run;
/* on log
before:
  A B C D E F G H I J K L M N
1. -1 -1 -1 . . . . . . . . . .
2. . . . -1 . . . . . . . . . .
3. -1 . . . -1 . . . . . . . . .
4. . . . . . . -1 -1 -1 -1 . . .

after:
  A B C D E F G H I J K L M N
1. 1 1 1 . . . . . . . . . .
2. . . . 2 . . . . . . . . . .
3. 1 . . . 1 . . . . . . . . . .
4. . . . . . . 3 3 3 3 . . . .
*/
%end;

%*-- test8. "Summing combinations of n objects taken m by m,    --*;
%*--   where for each combination we consider the product of    --*;
%*--   elements"                                                --*;
%*--                                                            --*;
%*-- Translated from Ian Whitlock's macro %B (sas-l 1998-07-09), --*;
%*-- where Ian says:                                           --*;
%*--                                                            --*;
%*-- "Now Fabrizio actually wanted the call                    " --*;
%*-- "                                                           " --*;
%*-- "   %put %b(2,60)                                         " --*;
%*-- "                                                           " --*;
%*-- "When I tried this my system ran out of resources.        " --*;
%*-- "So yes you can make recursive macro calls, but one       " --*;
%*-- "should remember that even a rather innocent looking     " --*;
%*-- "recursion can demand an awful lot of resources from     " --*;
%*-- "the system.                                              " --*;
%*-- "[...]                                                    " --*;
%*-- "I once was almost thrown out of a COBOL training program" --*;
%*-- "on my first programming job when I innocently asked if  " --*;
%*-- "COBOL could make recursive calls to perform.            " --*;
%*--                                                            --*;
%*-- [a semicolon ending %put statement in the original posting --*;
%*-- is not shown in the above quote]                          --*;
%if &what.=_ALL_ or %index(&what.,8) %then %do;
data _null_;
    %sub_init(b, m n temp) /* m and n are parameters temp is local */

    x = .;
    %sub_call(b, 2 40 .)
    put "B(2,40)=" x;

    x = .;
    %sub_call(b, 2 60 .)

```

```

put "B(2,60)=" x;
stop;

%sub_do(b)
if m = 0 and n >= 0 then x = 1;
else if m > n then x = 0;
else do;
    %sub_call(b, m n-1 .)
    temp = x;
    %sub_call(b, m-1 n-1 .)
    temp = temp + n * x;
    x = temp;
end;
*put "returning b(" m "," n ") is " x;
%sub_end(b)
run;
/* on log
B(2,40)=325130
B(2,60)=1637545
NOTE: DATA statement used:
      real time          0.22 seconds
      cpu time           0.21 seconds
*/
%end;
%mend sub_test;
options mprint nosymbolgen;
%*sub_test(_ALL_); /* uncomment this line to run the tests */

%*-- end -----*;

```