Observations®
THE TECHNICAL JOURNAL
for SAS Software Users
www.sas.com/obs

A Pseudo-Recursive SAS® Macro

William E. Benjamin , Jr.

William E. Benjamin, Jr. is a lead programmer working for American Express Co. His areas of expertise include base SAS® software, SAS/AF® software, and SAS macros. William has a BS degree in computer science from Arizona State University. He has been a user of SAS software since 1983 and working full time as a computer programmer since 1973. His programming experience spans from vacuum tube computers, large mainframe computers, and all the way to current PC computers, and using assembly language through fourth-generation computer languages.

Abstract

The purpose of this article is to explain a method of how to use macros to simulate recursion, a programming technique available in other programming languages. This method combines the use of arrays, links, and a macro, with parameters that modify the execution of the macro so that the macro repeatedly processes data as though it were on a run-time stack. The article also uses an indexed file to perform a depth-first search of selected nodes from a list of nodes with dependencies. The use of recursive programming routines is typically for solving problems with a "divide and conquer" type of approach. In programs of this nature, the object is to divide the problem into smaller and smaller pieces until an individual piece has a trivial solution. Then all of the pieces are re-assembled to create the completed solution. Recursive routines generally use large amounts of memory because each new call to the recursive routine requires a new set of local variables. Languages that directly support recursion usually dynamically allocate the memory needed for the "run-time stack" and therefore do not need to know how many times the routine is going to be called recursively (that is, the depth of the recursive routines). SAS software requires pre-allocation of the arrays required to simulate the recursion and therefore limits the depth of the recursive problem that can be solved using this method. The amount of memory available to the user will affect the size of the arrays that can be defined. This routine was developed using SAS software because some of the pre-existing input files were SAS files. This macro was developed using Release 6.08 on an IBM® mainframe computer.

Contents

Introduction

In the Pascal programming language, recursion is when the programmer defines a subroutine and from within the subroutine calls the same subroutine. Then the run-time system takes care of all of the issues related to the concepts of "local" and "global" variable relationships. Eventually the subroutine will have a final condition in which the subroutine no longer calls itself, and then all of the other calls are resolved one at a time until the first call is satisfied and the program continues. In this model, all of the "local" variable values are stored in a "run-time stack" that is defined by the compiler and maintained by the run-time system. SAS software does not provide all of these services to the user in the same way that Pascal does; in fact, the user has to know how to build all of these structures, and only then can they be used. This article shows which structures are needed and how to build them from the data structures that SAS software provides.

The Program

The task of finding or automating the search for all of the required pieces of source code for a program can be a real challenge if the software system has more than a few pieces of code. If the user has a list of all of the code modules, and for each module a list of all modules required to create that module (that is, a list of dependencies), then a depth first search of the dependencies will yield the required list of code modules. An easy way to code a depth first search is to use a recursive routine and examine each node until all of the dependent nodes are located. SAS software, however, does not allow direct recursion.

By way of example, if a small universe of code modules were created, the dependency list could look like the sample below:

```
/*************************************************************
/* SEARCH FILE (COL 3  = ROOT NODE,
/*              COL 10 = COUNT,
/*             LEAVES  = COLS 15, 20,25, Etc
/*************************************************************
/* note - use only the data lines below when coding the example
/*        these comments are not part of the data lines.
/*************************************************************
/*ROOT, COUNT-LEAVES
/*************************************************************/
  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
```

And a sample list of required modules could look like the following:

```
BB
EE
FF
```

From this list we can derive the following:

1. Module BB requires modules GG and HH, and while module GG is complete, module HH needs module II, which needs module JJ, which needs module KK, which needs module LL, which needs module MM.

2. Module EE is complete.

3. Module FF needs module MM.

4. Therefore the complete list of modules needed is BB, GG, HH, II, JJ, KK, LL, MM, EE, FF, and MM. Of course for an application where the code will be compiled only one copy of MM is needed because a subroutine can be called from many places.

From this small example we can see that if the dependency list has over 10,000 entries with up to 50 dependent code items per entry, a manual search for the code items is not feasible. Furthermore, each of the entries on the input list of variables required (BB, EE, and FF) could be viewed as the root node of a tree.

The next SAS code excerpt reads the previous dependency list and creates an indexed SAS work data set called MASTER. The SAS data set has 12 variables VARIABLE, COUNT, and DEP_01 through DEP_1 0). VARIABLE is the name of the root variable, COUNT is the count of the number of dependents, and DEP_01 through DEP_10 are the names of dependent variables. If there are fewer than 10 dependent variables, they are left blank.

```
DATA MASTER (INDEX = (VARIABLE=(VARIABLE) ) );
INPUT
@03 VARIABLE $CHAR2.
@10 D_COUNT 1.
@15 DEP_01 $CHAR5.
@20 DEP_02 $CHAR5.
@25 DEP_03 $CHAR5.
@30 DEP_04 $CHAR5.
@35 DEP_05 $CHAR5.
@40 DEP_06 $CHAR5.
@45 DEP_07 $CHAR5.
@50 DEP_08 $CHAR5.
@55 DEP_09 $CHAR5.
@60 DEP_10 $CHAR5.
;
cards;
... note - insert dependency list here ...
;;;;
```

The next routine is the heart of the recursion illusion. The SAS System builds macros from tokens supplied to the macro compiler, and the resulting code becomes a part of the program, which can link to and then use these routines. Also, if the macro is preceded by a label and followed by a return, then the code can be treated as a subroutine. Furthermore, there is no restriction on how many times a macro can be called in the same DATA step, nor is there a restriction on how many times it can be defined in a DATA step. The amount of memory available is the only limit to the number of macro definitions in a program. The only real problem is in the program logic; DO loops that use an index variable or LABELS that are defined in a macro to be branch locations are frozen in the macro code when the macro is defined. These program structures must be unique. Therefore, care must be taken to make them unique with *each and every* call to the macro.

This code segment is a SAS MACRO, which will get entered only if a variable on the main list has a dependent variable. The reference numbers have been added to all of the following code segments to enable a line-by-line discussion of the code.

```
(01) /*****************************************************************
(02) /* MACRO DEPEND - A PSEUDO-RECURSIVE MACRO
(03) /*****************************************************************/
(04) %MACRO DEPEND(UP_LVL,IDX_VAR,NEXT_LVL);
(05)
(06)  LEVEL = LEVEL + &UP_LVL;
(07)  LINK    PUSH_VAR;
(08)
(09)  DO &IDX_VAR = 1 TO 10;
(10)   IF T_STACK{LEVEL,&IDX_VAR} NOT = '  '
(11)    THEN DO;
(12)      VARIABLE = T_STACK{LEVEL,&IDX_VAR};
(13)      LINK GET_MSTR;
```

```
(14)
(15)       IF (READ_OK =  'OK') THEN OUTPUT;
(16)
(17)       IF (T_COUNT{LEVEL} > 0 AND READ_OK = 'OK')
(18)         THEN DO;
(19)           T_COUNT{LEVEL} = T_COUNT{LEVEL} - 1;
(20)           LINK DEPEND&NEXT_LVL;
(21)         END;
(22)     END;
(23)  END;
(24)
(25)  LINK    POP_VARS;
(26)  LEVEL = LEVEL - &UP_LVL;
(27) %MEND DEPEND;
```

- Line 4: This is the macro definition, giving it the name DEPEND and three parameters with no initial values. The parameters have the following uses:

    1. UP_LVL: A numeric value added to the variable LEVEL indicating how much to change the value of LEVEL. This value is either zero or one.

    2. IDX_VAR: A character value used as the name of an index variable for a DO loop. A new variable is needed each time the macro is invoked to act as a LOCAL variable.

    3. NEXT_LVL: A character value used as the last character of a label name defining the object of a LINK instruction. The macro will be preceded by a label and followed by a return statement; therefore, each macro invocation will be part of a subroutine. The NEXT_LVL variable will define which subroutine to call next.

Note that for a routine to be recursive, it needs a place to store data variables from the previous execution of the routine so that when the current routine finishes, the previous routine can have its values restored and it can finish too. The Pascal language uses a "run-time stack" for this task. SAS software does not provide one of those for the user, but the user can define a multi-dimension array. The LEVEL variable points to the area of the arrays that hold the values that are LOCAL for this invocation of the macro DEPEND.

- Lines 6 and 7: Change the execution (pseudo-recursion) level and call subroutine, PUSH_VAR, to save the current variables on the multi-dimensional array that serves as the "run-time stack."

- Line 9: Opens a DO statement to process the array of dependent variables at the current execution level. The name of the index variable is passed as a parameter to the macro. The DO loop is closed in line 23.

- Line 10: Tests the temporary stack, the "run-time stack," at the current execution level for the currently indexed dependent variable, looking for a non-blank value. Lines 9 and 10 together search the list of dependent variables looking for the next variable to be processed.

- Line 11: The DO statement, opened here and closed on line 22, encloses the work done when the dependent variable is found.

- Line 12: Sets up the key for an indexed read of the master file in the subroutine GET_MSTR.

- Line 13: Calls the subroutine GET_MSTR to read an indexed SAS file, using the variable named VARIABLE as the key to the file containing a list of valid variables and their dependents.

- Line 15: Verifies that the key value exists on the SAS file of variables and their dependents, and when the read is successful, a SAS observation is written for this dependent variable.

- Line 17: This line tests the variable T_COUNT for a value greater than zero, for the current level of recursion, if the last SAS read from GET_MSTR was OK. In other words, if the variable currently being tested has more dependent variables then the program will keep looking for more variables by executing lines 18 through 21.

- Line 18: Opens a DO loop closed at line 21. This DO loop contains the LINK to the next subroutine in the "pseudo-recursive" chain of subroutines, and is only executed if more dependent variables exist for the current variable (that is, as long as T_COUNT has a value greater than zero).

- Line 19: This line reduces by one the number of dependent variables to be searched for in the next level of recursion. This array of counters is indexed by the recursion level of the current test.

- Line 20: This line of code contains the pseudo-recursive macro call and must be different every time that the macro is invoked because it links to a pre-defined label in the SAS DATA step. The SAS LINK instruction is used to build the address; the address in this example is prefixed with the letters DEPEND and completed by adding the contents of the macro variable NEXT_LVL to the label. Macro names and code definitions are fixed before the DATA step executes, which allows the code on line 20 to work.

- Lines 25 and 26: Call a subroutine to restore the previous variables from the multi-dimensional array that serves as the "run-time stack," and change the execution level (pseudo-recursion level). This link to the subroutine "POP_VARS" is executed after all ten of the current dependent variable array slots are tested for more dependent variables.

- Line 27: This terminates the definition of the SAS macro DEPEND.

The next portion of the code is the main control loop of the SAS DATA step that invokes the macro DEPEND. This DATA step calls several subroutines using the LINK / RETURN statement combination, and each of these subroutines will be described separately:

```
(28) /*************************************************************
(29) /* SEARCH - A ROUTINE TO FIND ALL DEPENDENT VARIABLES, GIVEN A LIST
(30) /*************************************************************/
(31)
(32) DATA DEP_LIST(KEEP=VARIABLE LEVEL ) /STACK=40;
(33)
(34) ARRAY DEP                 {10}           $5   DEP_01-DEP_10;
(35)
(36) ARRAY VAR_NAME   {0:10}        $5    _TEMPORARY_;
(37) ARRAY T_COUNT       {0:10}          5   _TEMPORARY_;
(38) ARRAY T_STACK       {0:10,10}    $5   _TEMPORARY_;
(39)
(40) RETAIN          LEVEL        0;
(41)
(42) INFILE VARLIST; /* VARLIST is a user-supplied filename reference */
(43) INPUT @** VARIABLE $CHAR5.; /* adjust ** to data location on line */
(44)
(45) LEVEL = 0;
(46)
(47) LINK GET_MSTR;
(48)
(49) IF READ_OK = 'OK' THEN DO;
(50)  OUTPUT;
(51)  IF D_COUNT > 0 THEN LINK DEPEND;
(52) END;
(53)
(54) RETURN;
```

- Line 32: Begins the SAS DATA step. This defines the output SAS work data file called DEP_LIST that keeps the two SAS variables named VARIABLE and LEVEL. Furthermore, this line also adds another key element of the program. The characters /STACK= is an option for the DATA command that changes the number of LINK commands that can be invoked in a SAS DATA step before SAS assumes that the DATA step is in an infinite loop. In the normal execution mode, SAS allows ten LINK commands without a RETURN before stopping the program because of a loop. This DATA step set the number of LINK commands to 40.

- Line 34: The SAS Array DEP is defined, containing ten variables. The variables are DEP_01 to DEP_10, which are character variables of length 5. These variables are read into the DATA step from the SAS file called MASTER, and are only meaningful when the results of the GET_MSTR routine are OK. This DATA step compares the user's input list of variables against the SAS data set called MASTER. The user's list may have variables that are not on the MASTER file. This routine does not output these variables, but each is reset to missing at the start of processing for each observation.

- Line 36: The SAS array VAR_NAME is defined as an array of 11 SAS temporary character variables, each of which is five characters long. This array holds the name of the variable being processed for each of the 11 levels of the pseudo-recursive routine. These variables are not reset to missing at the start of processing for each observation.

- Line 37: The SAS array T_COUNT is defined as an array of 11 SAS temporary numeric variables, each of which is five bytes long. This array holds the number of dependent variables for the variable being processed at the current level of the pseudo-recursive routine. These variables are not reset to missing at the start of processing for each observation.

- Line 38: The SAS array T_STACK is defined as an array of 11 groups of ten SAS temporary variables, each of which is five characters long. This array holds the list of ten possible dependent variables for each of the 11 levels of the pseudo-recursive routine. These variables are not reset to missing at the start of processing for each observation.

- Line 40: By issuing a RETAIN command for variable LEVEL, the value is not reset to missing at the start of each observation, and this variable will point to the current group of dependent variables being processed.

- Line 42: This INFILE command is pointing to the external file VARLIST that contains an input list of variables that is being searched for unnamed dependent variables (BB, EE, and FF from our example). The exact form of this line will vary depending on the computer system in use. This macro was developed on an IBM® mainframe and the file name VARLIST was defined in the execution job control information. Other forms of the INFILE statement are documented in the SAS language reference manuals for the system in use.

- Line 43: The INPUT statement reads the input variable value from column 3 of the input record. This variable is not automatically placed into the output file because the input list may contain variable names not in the master list of variables. This is the first variable of a new "tree" of variables to be searched. The variable VARIABLE is the "key" for the indexed SAS file of variables and their dependencies.

- Line 45: LEVEL is reset to zero at the start of each processing cycle. This variable could be the first of hundreds of values output by these routines.

- Line 47: The LINK to GET_MSTR invokes the subroutine to read the indexed SAS file of valid variables and their dependencies.

- Line 49: This line tests the results of the GET_MSTR routine. An "OK" result means that the current variable being tested was found on the master list of variables. This DO group ends in line 52.

- Line 50: The output command writes the current variable from the user-input list to the output SAS data set.

- Line 51: Links to the label DEPEND if there is one or more dependent modules for the current variable. This is the start of the pseudo-recursion for a new tree with the user-input variable as the root of the tree.

- Line 54: This ends the main control loop for the routine.

The following code is broken into the subroutines for clarity:

```
(55) /*****************************************************************
(56) /* CLEAR VARIABLE STACK.
(57) /*****************************************************************/
(58) EMPTY_ST:
(59)
(60) DO Y = 0 TO 10;
(61)
(62)   VAR_NAME{Y} = '';
(63)   T_COUNT {Y} = 0;
(64)
(65)   DO Z = 1 TO 10;
(66)     T_STACK  {Y,Z} = '';
(67)   END;
(68) END;
(69)
(70) RETURN;
```

- Line 58: This line contains a label that starts the EMPTY_ST subroutine. This routine clears the three arrays VAR_NAME, T_COUNT, and T_STACK. This is done at the beginning of processing for a new tree.

- Line 60: Opens a new DO loop to cycle through all 11 elements of the three arrays. This DO loop ends at line 68.

- Line 62: Clears the VAR_NAME array.

- Line 63: Clears the T_COUNT array.

- Line 65: Opens a DO loop to clear the T_STACK array elements in the second dimension of the array, which contains 110 total elements. This DO loop is ended in line 67.

- Line 70: Ends the subroutine and returns to the instruction following the link that invoked the subroutine.

```
(71) /*****************************************************************
(72) /* POP  VARIABLE AND DEPENDENCIES FROM A STACK.
(73) /*****************************************************************/
(74) POP_VARS:
(75)
(76)  VARIABLE = VAR_NAME{LEVEL};
(77)  D_COUNT  = T_COUNT {LEVEL};
(78)
```

```
(79)  VAR_NAME{LEVEL} = '';
(80)  T_COUNT {LEVEL} = 0;
(81)
(82)  DO Z = 1 TO 10;
(83)   DEP{Z}   = T_STACK  {LEVEL,Z};
(84)   T_STACK  {LEVEL,Z} = '';
(85)  END;
(86)
(87)  RETURN;
```

- Line 74: This label begins the subroutine POP_VARS. This routine moves data from the temporary (local variable) holding arrays back to the current (global) variable names. This routine is executed when leaving the pseudo-recursive routine and simulates restoring values from a “run-time stack” to the previous level of recursion.

- Line 76: Restores the value of VARIABLE from the previous level.

- Line 77: Restores the value of D_COUNT from the previous level.

- Line 79: Resets VAR_NAME to null to clean up this level of the "run-time stack" for the next execution of the SAS macro DEPEND.

- Line 80: Resets T_COUNT to 0 to clean up this level of the "run-time stack" for the next execution of the SAS macro DEPEND.

- Lines 82 to 85: Execute a loop to restore the values of DEP from the previous level and clean up the temporary array of T_STACK at this level.

- Line 87: Ends this subroutine.

```
(88) /*************************************************************
(89) /* PUSH VARIABLE AND DEPENDENCIES ONTO A STACK.
(90) *************************************************************/
(91)  PUSH_VAR:
(92)
(93)   VAR_NAME{LEVEL} = VARIABLE;
(94)   T_COUNT {LEVEL} = D_COUNT;
(95)
(96)   DO Z = 1 TO 10;
(97)    T_STACK  {LEVEL,Z} = DEP  {Z};
(98)   END;
(99)
(100)  RETURN;
```

- Line 91: This label begins the subroutine PUSH_VAR. This routine moves data to the temporary (local variable) holding arrays from the current (global) variable names. This routine is executed when entering the pseudo-recursive routine and simulates saving values in a “run-time stack” for this level of recursion.

- Line 93: Saves the value of VARIABLE for the current level.

- Line 94: Saves the value of D_COUNT for the current level.

- Lines 96 to 98: Executes a loop to save the values of DEP for the current level into the temporary array T_STACK at this level.

- Line 100: Ends this subroutine.

```
(101) /*****************************************************************
(102) /*  SELECT OUT OF GROUP DEPENDENCIES.
(103) /*****************************************************************/
(104) DEPEND : LINK EMPTY_ST ;
(105)         %DEPEND(0,A,0); RETURN;
(106) DEPEND0: %DEPEND(1,B,1); RETURN;
(107) DEPEND1: %DEPEND(1,C,2); RETURN;
(108) DEPEND2: %DEPEND(1,D,3); RETURN;
(109) DEPEND3: %DEPEND(1,E,4); RETURN;
(110) DEPEND4: %DEPEND(1,F,5); RETURN;
(111) DEPEND5: %DEPEND(1,G,6); RETURN;
(112) DEPEND6: %DEPEND(1,H,7); RETURN;
(113) DEPEND7: %DEPEND(1,I,8); RETURN;
(114) DEPEND8: %DEPEND(1,J,9); RETURN;
(115) DEPEND9: %DEPEND(1,K,A); RETURN;
(116) DEPENDA: ABORT RETURN 1111 ; RETURN;
```

This segment of SAS code may look strange, but the compiler has no problem knowing that these 12 lines of code contain the backbone of the pseudo-recursiveness of this routine. Each line (or pair of lines) begins with a label (DEPEND*:), is followed by a SAS MACRO invocation (%DEPEND(#,$,*);), and closes with RETURN. The last line (116) differs in that it forces an ABORT if the depth exceeds the limits defined by %DEPEND. This structure can be viewed as a stack of macros that allows for a tree depth of 11 levels of dependent variables in a tree. Each of these macros will call the next macro in line and start the process over again. Each of these macros will save the current values upon entry and restore them on the way out. The first call clears all temporary variables for a clean start; this is also the first non-root node to be processed, with the user-input list being a list of root nodes.

The structure of the macro call is defined as follows, for the calls %DEPEND(#,$,*); where

%DEPEND is the name of the macro defined above.

is the value of the variable UP_LVL with a range of zero to one.

$ is the value of the variable IDX_VAR, which is the name of the variable used in the DO loop defined in line 9 of the macro definition.

* is the value of the variable NEXT_LVL, which is the last character of the label referenced by the LINK in line 20 of the macro definition. The labels are defined in lines 104 to 116.

```
(117) /*****************************************************************
(118) /*  LOOK UP VARIABLE NAME ON THE MASTER LIST OF VARIABLES.
(119) /*****************************************************************/
(120) GET_MSTR:
(121)
(122) _IORC_  = 0;
(123) _ERROR_ = 0;
(124)
(125) SET MASTER(KEEP=VARIABLE D_COUNT DEP_01-DEP_10
(126)     ) KEY=VARIABLE/UNIQUE;
(127)
(128) IORC_ = RESOLVE(_IORC_);
(129) READ_OK = 'NO';
(130) SELECT (IORC_);
(131) WHEN(%SYSRC(_DSEMTR))     PUT  'MULT TRANS NOT ON MASTER  ';
(132) WHEN(%SYSRC(_DSENMR))     PUT  'TRANS NOT ON MASTER       ';
(133) WHEN(%SYSRC(_DSENOM))     PUT  'NOT ON MASTER             ';
```

```
(134) WHEN(%SYSRC(_SEDSKFL))  PUT  'TAPE/DISK IS FULL ';
(135) WHEN(%SYSRC(_SEINVPS))  PUT  'VALUE OF POSITION INVALID ';
(136) WHEN(%SYSRC(_SENOCHN))  PUT  'NO CHANGE/NO DUPS ALLOWED ';
(137) WHEN(%SYSRC(_SENOLCK))  PUT  'RECORD LOCK NOT AVAILABLE ';
(138) WHEN(%SYSRC(_SENODEL))  PUT  'RECORD MAY NOT BE DELETED ';
(139) WHEN(%SYSRC(_SERECRD))  PUT  'NO REC READ FROM INPUT ';
(140) WHEN(%SYSRC(_SWDLREC))  PUT  'REC DELETED FROM FILE  ';
(141) WHEN(%SYSRC(_SWNOUPD))  PUT  'RECORD CANNOT BE UPDATED ';
(142) WHEN(%SYSRC(_SWNOWHR))  PUT  'NO LONGER SATISFY "WHERE" ';
(143) OTHERWISE READ_OK = 'OK';
(144) END;
(145)
(146) _ERROR_ = 0;
(147)
(148) RETURN;
```

This routine allows the MASTER file described above to be of any size, without degrading the performance of the routine. By using a KEY search of an indexed file, the access is rapid for even the largest of files.

- Line 120: This label starts the subroutine.

- Line 122: Clears the SAS System variable _IORC_ to ensure proper testing later.

- Line 123: Clears the SAS System variable _ERROR_ to ensure proper testing later.

- Lines 125 and 126: Use a SAS indexed read via a SET statement. The KEEP statement allows the Master file to contain many other variables that are not important to the search. The phrase KEY=VARIABLE/UNIQUE ensures that each time the file MASTER is searched the search will begin at the top of the file.

- Line 128: Places the return code from the SET statement I/O into a user variable IORC_. The system variable _IORC_ may be reset too often to test if more than one instruction is involved in the testing.

- Line 129: The READ_OK variable is preset to show that the indexed read of MASTER failed, therefore it need only be reset if the read is successful. This enables the following SELECT statement to remain simple.

- Lines 130 through 144: This SELECT statement tests the value of the user variable IORC_ for error conditions and writes the conditions to the SAS LOG when the read fails, or sets the read indicator (user variable READ_OK).

- Line 146: By resetting the system variable _ERROR_ to 0 after the testing of the _IORC_ values and before the start of the next SAS observation, the SAS run-time system will not stop execution due to an error in the reading of the MASTER file.

- Line 148: ends the subroutine GET_MSTR and the program.

Conclusion

The SAS System supports programming techniques and data structures that allow users to be creative in the way they solve problems. The only output from this routine is a SAS data set with two variables, but there are 12 places in the program where it is output. The output occurs once at line 50, in each of 11 macro calls, and at line 15 in the macro. The 11 macro calls are defined at lines 105-115. This type of programming for recursive applications *require*s a pre-existing knowledge of the underlying data to be processed, because the maximum size of the program is pre-set by the programmer. The fact that all of the code for the program is written by the macro compiler as a single-use subroutine rather than a called re-entrant code module is what makes this routine "pseudo-recursive." The requirement that the called routines be pre-defined is a limiting factor to this type of programming technique. Admittedly, the 11 levels of recursion used here for a tree does not allow for the very large tree structures usually associated with recursive programming. But it's possible to use some of those old dusty PASCAL routines that your college professors showed you once upon a time, so long ago.

References

SAS Institute Inc. (1990), *SAS Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1990), *SAS Procedures Guide, Version 6, Third Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1997), *SAS Macro Language: Reference, First Edition*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1991), SAS Technical Report P-222, *Changes and Enhancements to Base SAS Software, Release 6.07*, Cary, NC: SAS Institute Inc.

Questions and comments should be directed to :

William E. Benjamin Jr.
1531 W. Michigan Ave.
Phoenix AZ 85023
E-mail: WEBenjaminJr@email.msn.com