Chapter 15

Using SAS/IML Software to Generate IML Statements

Chapter Table of Contents

Chapter 15

Using SAS/IML Software to Generate IML Statements

Overview

This chapter describes ways of using SAS/IML software to generate and execute statements from within the Interactive Matrix Language. You can execute statements generated at run time, execute global SAS commands under program control, or create statements dynamically to get more flexibility.

Generating and Executing Statements

You can push generated statements into the input command stream (queue) with the PUSH, QUEUE, and EXECUTE subroutines. This can be very useful in situations that require added flexibility, such as menu-driven applications or interrupt handling.

The PUSH command inserts program statements at the front of the input command stream, whereas the QUEUE command inserts program statements at the back. In either case, if they are not input to an interactive application, the statements remain in the queue until IML enters a pause state, at which point they are executed. The pause state is usually induced by a program error or an interrupt control sequence. Any subsequent RESUME statement resumes execution of the module from the point where the PAUSE command was issued. For this reason, the last statement put into the command stream for PUSH or QUEUE is usually a RESUME command.

The EXECUTE statement also pushes program statements like PUSH and QUEUE, but it executes them immediately and returns. It is not necessary to push a RESUME statement when you use the CALL EXECUTE command.

Executing a String Immediately

The PUSH, QUEUE, and EXECUTE commands are especially useful when used in conjunction with the pause and resume features because they enable you to generate a pause-interrupt command to execute the code you push and return from it via a pushed RESUME statement. In fact, this is precisely how the EXECUTE subroutine is implemented generally.

CAUTION: Note that the push and resume features work this way only in the context of being inside modules. You cannot resume an interrupted sequence of statements in immediate mode, that is, not inside a module.

For example, suppose that you collect program statements in a matrix called CODE. You push the code to the command input stream along with a RESUME statement

and then execute a PAUSE statement. The PAUSE statement interrupts the execution, parses and executes the pushed code, and returns to the original execution via the RESUME statement.

```
proc iml;
start testpush;
   print '*** ENTERING MODULE TESTPUSH ***';
   print '*** I should be 1,2,3: ';
   /* constructed code * /
   code = ' do i = 1 to 3; print i; end;   ';
   /* push code+resume */
   call push (code, 'resume;');
   /* pause interrupt  */
   pause;
   print '*** EXITING MODULE TESTPUSH ***';
finish;
```

When the PAUSE statement interrupts the program, the IML procedure then parses and executes the line:

```
do i=1 to 3; print i; end; resume;
```

The RESUME command then causes the IML procedure to resume the module that issued the PAUSE.

Note: The EXECUTE routine is equivalent to a PUSH command, but it also adds the push of a RESUME command, then issues a pause automatically.

A CALL EXECUTE command should be used only from inside a module because pause and resume features do not support returning to a sequence of statements in immediate mode.

Feeding an Interactive Program

Suppose that an interactive program gets responses from the statement INFILE CARDS. If you want to feed it under program control, you can push lines to the command stream that is read.

For example, suppose that a subroutine prompts a user to respond **YES** before performing some action. If you want to run the subroutine and feed the **YES** response without the user being bothered, you push the response as follows:

```
   /* the function that prompts the user */
start delall;
   file log;
   put 'Do you really want to delete all records? (yes/no)';
   infile cards;
   input answer $;
   if upcase(answer)='YES' then
      do;
         delete all;
         purge;
```

```
         print "*** FROM DELALL:
         should see End of File (no records to list)";
         list all;
      end;
finish;
```

The latter DO group is necessary so that the pushed **YES** is not read before the RUN statement. The following example illustrates the use of the module DELALL given above:

```
   /* Create a dummy data set for delall to delete records  */
xnum = {1 2 3, 4 5 6, 7 8 0};
create dsnum1 from xnum;
append from xnum;
   do;
      call push ('yes');
      run delall;
   end;
```

Calling the Operating System

Suppose that you want to construct and execute an operating system command. Just push it to the token stream in the form of an X statement and have it executed under a pause interrupt.

The following module executes any system command given as an argument:

```
start system(command);
   call push(" x '",command,"'; resume;");
   pause;
finish;
run system('listc');
```

The call generates and executes a LISTC command under MVS:

```
x 'listc'; resume;
```

Calling Display Manager

The same strategy used for calling the operating system works for SAS global statements as well, including calling display manager by generating DM statements.

The following subroutine executes a display manager command:

```
start dm(command);
   call push(" dm '",command,"'; resume;");
   pause;
finish;
run dm('log; color source red');
```

The call generates and executes the statements

```
dm 'log; color source red'; resume;
```

which take you to the LOG window, where all source code is written in red.

Executing Any Command in an EXECUTE Call

The EXECUTE command executes the statements contained in the arguments using the same facilities as a sequence of CALL PUSH, PAUSE, and RESUME statements. The statements use the same symbol environment as that of the subroutine that calls them. For example, consider the following subroutine:

```
proc iml;
start exectest;
/*    IML STATEMENTS  */
   call execute ("xnum = {1 2 3, 4 5 6, 7 8 0};");
   call execute ("create dsnum1 from xnum;");
   call execute ("append from xnum;");
   call execute ("print 'DSNUM should have 3 obs and 3 var:';");
   call execute ("list all;");
/*    global (options) statement  */
   call execute ("options linesize=68;");
   call execute ("print 'Linesize should be 68';");
finish;
run exectest;
```

The output generated from EXECTEST is exactly the same as if you had entered the statements one at a time:

```
 DSNUM should have 3 obs and 3 var:

 OBS      COL1      COL2      COL3
------ --------- --------- ---------
     1    1.0000    2.0000    3.0000
     2    4.0000    5.0000    6.0000
     3    7.0000    8.0000         0

        Linesize should be 68
```

CALL EXECUTE could almost be programmed in IML as shown here; the difference between this and the built-in command is that the following subroutine would not necessarily have access to the same symbols as the calling environment:

```
start execute(command1,...);
   call push(command1,...," resume;");
   pause;
finish;
```

Making Operands More Flexible

Suppose that you want to write a program that prompts a user for the name of a data set. Unfortunately the USE, EDIT, and CREATE commands expect the data set name as a hardcoded operand rather than an indirect one. However, you can construct and execute a function that prompts the user for the data set name for a USE statement.

```
/* prompt the user to give dsname for use statement */
start flexible;
   file log;
   put 'What data set shall I use?';
   infile cards;
   input dsname $;
  call execute('use', dsname, ';');
finish;
run flexible;
```

If you enter USER.A, the program generates and executes the line

```
use user.a;
```

Interrupt Control

Whenever a program error or interrupt occurs, IML automatically issues a pause, which places the module in a paused state. At this time, any statements pushed to the input command queue get executed. Any subsequent RESUME statement (including pushed RESUME statements) resume executing the module from the point where the error or interrupt occurred.

If you have a long application such as reading a large data set and you want to be able to find out where the data processing is just by entering a break-interrupt (sometimes called an attention signal), you push the interrupt text. The pushed text can, in turn, push its own text on each interrupt, followed by a RESUME statement to continue execution.

For example, suppose you have a data set called TESTDATA that has 4096 observations. You want to print the current observation number if an attention signal is given. The following code does this:

```
start obsnum;
   use testdata;
   brkcode={"print 'now on observation number',i;"
            "if (i<4096) then do;"
            "call push(brkcode);"
            "resume;"
            "end;"
            };
   call push(brkcode);
   do i=1 to 4096;
      read point i;
   end;
```

```
finish;
run obsnum;
```

After the module has been run, enter the interrupt control sequence for your operating system. Type S to suspend execution. The IML procedure prints a message telling which observation is being processed. Because the pushed code is executed at the completion of the module, the message is also printed when OBSNUM ends.

Each time the attention signal is given, OBSNUM executes the code contained in the variable BRKCODE. This code prints the current itteration number and pushes commands for the next interrupt. Note that the PUSH and RESUME commands are inside a DO group, making them conditional and ensuring that they are parsed before the effect of the PUSH command is realized.

Specific Error Control

A PAUSE command is automatically issued whenever an execution error occurs, putting the module in a holding state. If you have some way of checking for specific errors, you can write an interrupt routine to correct them during the pause state.

In this example, if a singular matrix is passed to the INV function, the IML procedure pauses and executes the pushed code to make the result for the inverse be set to missing values. The code uses the variable SINGULAR to detect if the interrupt occurred during the INV operation. This is particularly necessary because the pushed code is executed on completion of the routine, as well as on interrupts.

```
proc iml;
a = {3 3, 3 3};                          /* singular matrix */
/* If a singular matrix is sent to the INV function,        */
/* IML normally sets the resulting matrix to be empty       */
/* and prints an error message.                             */
b = inv(a);
print "*** A should be non-singular", a;
start singtest;
   msg="     Matrix is singular - result set to missing ";
   onerror=
      "if singular then do; b=a#.; print msg; print b;
       call push(onerror); resume; end;";
   call push(onerror);
   singular = 1;
   b = inv(a);
   singular = 0;
finish ;
call singtest;
```

The resulting output is shown below:

```
ERROR: (execution) Matrix should be non-singular.

       Error occured in module SINGTEST at line    67 column   9
       operation : INV                  at line    67 column  16
```

```
 operands  : A

A            2 rows      2 cols    (numeric)

         3         3
         3         3

 stmt: ASSIGN                     at line    67 column   9

Paused in module SINGTEST.


            MSG
                 Matrix is singular - result set to missing


                                        B
                                        .         .
                                        .         .


Resuming execution in module SINGTEST.
```

General Error Control

Sometimes, you may want to process or step over errors. To do this, put all the code into modules and push a code to abort if the error count goes above some maximum. Often, you may submit a batch job and get a trivial mistake that causes an error, but you do not want to cause the whole run to fail because of it. On the other hand, if you have many errors, you do not want to let the routine run.

In the following example, up to three errors are tolerated. A singular matrix **A** is passed to the INV function, which would, by itself, generate an error message and issue a pause in the module. This module pushes three RESUME statements, so that the first three errors are tolerated. Messages are printed and execution is resumed. The DO loop in the module OOPS is executed four times, and on the fourth iteration, an ABORT statement is issued and you exit IML.

```
proc iml;
a={3 3, 3 3};                          /* singular matrix */
   /*                                                     */
   /* GENERAL ERROR CONTROL -- exit iml for 3 or more errors */
   /*                                                     */
 start;                          /* module will be named MAIN */
    errcode = {" if errors >= 0 then do;",
               "    errors = errors + 1;",
               "    if errors > 2 then abort;",
               "    else do; call push(errcode); resume; end;",
               " end;" } ;
    call push (errcode);
    errors = 0;
    start oops;                        /* start module OOPS */
       do i = 1 to 4;
           b = inv(a);
       end;
```

```
   finish;                                    /* finish OOPS */
   run oops;
finish;                                       /* finish MAIN */
errors=-1;                                    /* disable     */
run;
```

The output generated from this example is shown below:

```
 ERROR: (execution) Matrix should be non-singular.

 Error occured in module OOPS     at line    41 column  17
 called   from    module MAIN     at line    44 column  10
 operation : INV                  at line    41 column  24
 operands  : A

A             2 rows      2 cols    (numeric)

         3         3
         3         3

 stmt: ASSIGN                     at line    41 column  17

Paused in module OOPS.

Resuming execution in module OOPS.
ERROR: (execution) Matrix should be non-singular.

 Error occured in module OOPS     at line    41 column  17
 called   from    module MAIN     at line    44 column  10
 operation : INV                  at line    41 column  24
 operands  : A

A             2 rows      2 cols    (numeric)

         3         3
         3         3

 stmt: ASSIGN                     at line    41 column  17

Paused in module OOPS.

Resuming execution in module OOPS.
ERROR: (execution) Matrix should be non-singular.

 Error occured in module OOPS     at line    41 column  17
 called   from    module MAIN     at line    44 column  10
 operation : INV                  at line    41 column  24
 operands  : A

A             2 rows      2 cols    (numeric)

         3         3
         3         3
```

```
 stmt: ASSIGN                    at line    41 column  17

Paused in module OOPS.

Exiting IML.
```

Actually, in this particular case it would probably be simpler to put three RESUME statements after the RUN statement to resume execution after each of the first three errors.

Macro Interface

The pushed text is scanned by the macro processor; therefore, the text can contain macro instructions. For example, here is an all-purpose routine that shows what the expansion of any macro is, assuming that it does not have embedded double quotes:

```
    /* function: y = macxpand(x);                          */
    /* will macro-process the text in x,                   */
    /* and return the expanded text in the result.         */
    /* Do not use double quotes in the argument            */
    /*                                                     */
start macxpand(x);
   call execute('Y="',x,'";');
   return(y);
finish;
```

Consider the following statements:

```
%macro verify(index);
    data _null_;
       infile junk&index;
       file print;
       input;
       put _infile_;
    run;
%mend;
y = macxpand('%verify(1)');
print y;
```

The output produced is shown below:

```
    Y

    DATA _NULL_;      INFILE JUNK1;      FILE PRINT;      INPUT;
    PUT _INFILE_;      RUN;
```

IML Line Pushing Contrasted with Using the Macro Facility

The SAS macro language is a language embedded in and running on top of another language; it generates text to feed the other language. Sometimes it is more convenient to generate the text using the primary language directly rather than embedding

the text generation in macros. The preceding examples show that this can even be done at execution time, whereas pure macro processing is done only at parse time. The advantage of the macro language is its embedded, yet independent, nature: it needs little quoting, and it works for all parts of the SAS language, not just IML. The disadvantage is that it is a separate language that has its own learning burden, and it uses extra reserved characters to mark its programming constructs and variables. Consider the quoting of IML versus the embedding characters of the macro facility: IML makes you quote every text constant, whereas the macro facility makes you use the special characters percent sign (%) and ampersand (&) on every macro item. There are some languages, such as REXX, that give you the benefits of both (no macro characters and no required quotes), but the cost is that the language forces you to discipline your naming so that names are not expanded inadvertently.

Example 15.1. Full-Screen Editing

The ability to form and submit statements dynamically provides a very powerful mechanism for making systems flexible. For example, consider the building of a data entry system for a file. It is straightforward to write a system using WINDOW and DISPLAY statements for the data entry and data processing statements for the I/O, but once you get the system built, it is good only for that one file. With the ability to push statements dynamically, however, it is possible to make a system that dynamically generates the components that are customized for each file. For example, you can change your systems from static systems to dynamic systems.

To illustrate this point, consider an IML system to edit an arbitrary file, a system like the FSEDIT procedure in SAS/FSP software but programmed in IML. You cannot just write it with open code because the I/O statements hardcode the filenames and the WINDOW and DISPLAY statements must hardcode the fields. However, if you generate just these components dynamically, the problem is solved for any file, not just one.

```
proc iml;
/*                 FSEDIT                                        */
/* This program defines and stores the modules FSEINIT,          */
/* FSEDT, FSEDIT, and FSETERM in a storage catalog called        */
/* FSED.  To use it, load the modules and issue the command      */
/* RUN FSEDIT;  The system prompts or menus the files and        */
/* variables to edit, then runs a full screen editing            */
/* routine that behaves similar to PROC FSEDIT                   */
/*                                                               */
/* These commands are currently supported:                       */
/*                                                               */
/* END         gets out of the system.  The user is prompted     */
/*             as to whether or not to close the files and       */
/*             window.                                           */
/* SUBMIT      forces current values to be written out,          */
/*             either to append a new record or replace          */
/*             existing ones                                     */
/* ADD         displays a screen variable with blank values      */
/*             for appending to the end of a file                */
/* DUP         takes the current values and appends them to      */
/*             the end of the file                               */
```

```
/* number      goes to that line number                        */
/* DELETE      deletes the current record after confirmation */
/*             by a Y response                                 */
/* FORWARD1    moves to the next record, unless at eof         */
/* BACKWARD1   moves to the previous record, unless at eof     */
/* EXEC        executes any IML statement                      */
/* FIND        finds records and displays them                 */
/*                                                             */
/* Use: proc iml;                                              */
/*         reset storage='fsed';                               */
/*         load module=_all_;                                  */
/*         run fsedit;                                         */
/*                                                             */
/*---routine to set up display values for new problem---      */
start fseinit;
   window fsed0 rows=15 columns=60 icolumn=18 color='GRAY'
   cmndline=cmnd group=title +30 'Editing a data set' color='BLUE';
   /*---get file name---                                       */
   _file="                ";
   msg =
      'Please Enter Data Set Name or Nothing For Selection List';
   display  fsed0.title,
      fsed0 ( / @5  'Enter Data Set:'
                +1  _file
                +4  '(or nothing to get selection list)' );
   if _file=' ' then
      do;
         loop:
         _f=datasets(); _nf=nrow(_f); _sel=repeat("_",_nf,1);
         display fsed0.title,
            fsed0 (/ "Select? File Name"/) ,
            fsed0 (/ @5  _sel +1 _f protect=yes ) repeat ;
         _l = loc(_sel^='_');
         if nrow(_l)^=1 then
            do;
               msg='Enter one S somewhere';
               goto loop;
            end;
         _file = _f[_l];
      end;
   /*---open file, get number of records---                    */
   call queue(" edit ",_file,";
               setin ",_file," NOBS _nobs; resume;"); pause *;
   /*---get variables---                                       */
   _var = contents();
   _nv = nrow(_var);
   _sel = repeat("_",_nv,1);
   display fsed0.title,
           fsed0 (/ "File:" _file) noinput,
           fsed0 (/ @10 'Enter S to select each var, or select none
                  to get all.'
                  // @3  'select?  Variable  '  ),
                  fsed0 ( / @5  _sel  +5 _var protect=yes ) repeat;
   /*---reopen if subset of variables---                       */
   if any(_sel^='_') then
      do;
         _var = _var[loc(_sel^='_')];
         _nv = nrow(_var);
         call push('close ',_file,'; edit ',_file,' var
```

```
         _var;resume;');pause *;
      end;
   /*---close old window---                                    */
   window close=fsed0;
   /*---make the window---*/
   call queue('window fsed columns=55 icolumn=25 cmndline=cmnd
              msgline=msg ', 'group=var/@20 "Record " _obs
              protect=yes');
   call queue( concat('/"',_var,': " color="YELLOW" ',
              _var,' color="WHITE"'));
   call queue(';');
   /*---make a missing routine---*/
   call queue('start vmiss; ');
   do i=1 to _nv;
      val = value(_var[i]);
      if type(val)='N' then call queue(_var[i],'=.;');
      else call queue(_var[i],'="',
                     cshape(' ',1,1,nleng(val)),'";');
   end;
   call queue('finish; resume;');
   pause *;
   /*---initialize current observation---*/
   _obs = 1;
   msg = Concat('Now Editing File ',_file);
finish;
  /*                                                         */
  /*---The Editor Runtime Controller---                      */
start fsedt;
   _old = 0; go=1;
   do while(go);
   /*--get any needed data--*/
      if any(_obs^=_old) then do; read point _obs; _old = _obs;
   end;
   /*---display the record---*/
   display fsed.var repeat;
   cmnd = upcase(left(cmnd));
   msg=' ';
   if cmnd='END' then go=0;
   else if cmnd='SUBMIT' then
      do;
         if _obs<=_nobs then
            do;
               replace point _obs; msg='replaced';
            end;
         else do;
                append;
                _nobs=_nobs+nrow(_obs);
                msg='appended';
              end;
      end;
   else if cmnd="ADD" then
       do;
          run vmiss;
          _obs = _nobs+1;
          msg='New Record';
       end;
   else if cmnd='DUP' then
      do;
         append;
```

```
         _nobs=_nobs+1;
         _obs=_nobs;
         msg='As Duplicated';
      end;
   else if cmnd>'0' & cmnd<'999999' then
      do;
         _obs = num(cmnd);
         msg=concat('record number ',cmnd);
      end;
   else if cmnd='FORWARD1' then _obs=min(_obs+1,_nobs);
   else if cmnd='BACKWARD1' then _obs=max(_obs-1,1);
   else if cmnd='DELETE' then
      do;
         records=cshape(char(_obs,5),1,1);
         msg=concat('Enter command Y to Confirm delete of'
                    ,records);
         display fsed.var repeat;
         if (upcase(cmnd)='Y') then
            do;
               delete point _obs;
               _obs=1;
               msg=concat('Deleted Records',records);
            end;
         else msg='Not Confirmed, Not Deleted';
      end;
   else if substr(cmnd,1,4)='FIND' then
      do;
         call execute("find all where(",
                       substr(cmnd,5),
                       ") into _obs;" );
          _nfound=nrow(_obs);
          if _nfound=0 then
             do;
                _obs=1;
                msg='Not Found';
             end;
           else
              do;
                 msg=concat("Found ",char(_nfound,5)," records");
              end;
      end;
   else if substr(cmnd,1,4)='EXEC' then
      do;
         msg=substr(cmnd,5);
         call execute(msg);
      end;
   else msg='Unrecognized Command; Use END to exit.';
   end;
finish;
  /*---routine to close files and windows, clean up---*/
start fseterm;
     window close=fsed;
     call execute('close ',_file,';');
     free _q;
finish;
   /*---main routine for FSEDIT---*/
start fsedit;
  if (nrow(_q)=0) then
     do;
```

```
        run fseinit;
     end;
  else msg = concat('Returning to Edit File ',_file);
  run fsedt;
  _q='_';
  display fsed ( "Enter 'q' if you want to close files and windows"
               _q " (anything else if you want to return later"
               pause 'paused before termination';
  run fseterm;
finish;
reset storage='fsed';
store module=_all_;
```

Summary

In this chapter you learned how to use SAS/IML software to generate IML statements. You learned how to use the PUSH, QUEUE, EXECUTE, and RESUME commands to interact with the operating system or with the SAS System in display manager. You also saw how to add flexibility to programs by adding interrupt control features and by modifying error control. Finally, you learned how IML compares to the SAS macro language.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/IML User's Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. 846 pp.

SAS/IML User's Guide, Version 8

ISBN 1–58025–553–1

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999