ABSTRACT
A SAS procedure is a program usually written in a high-level language that performs a specific task. In this paper, we will aid a programmer in writing a SAS procedure for Version 5 by examining the parts of a procedure, explaining the steps involved in writing a procedure, and discussing areas in writing a procedure that are likely to give the programmer problems.

INTRODUCTION
A SAS procedure is a program usually written in a high-level language that interfaces with the SAS System to perform a specific task. A program used for a procedure in Version 5 can be written in PL/I, C, or FORTRAN. This paper is directed at the PL/I programmer. However, the principles discussed have general application.

By writing a SAS procedure, a programmer can often benefit from the built-in features of the SAS System such as statement processing, data management, and memory allocation, as opposed to supporting a stand-alone program. However, it is not always necessary or advantageous to take the approach of writing a procedure. You may need only to write a SAS program or a SAS macro. Many of the benefits of a procedure are available through the SAS macro language. However, the SAS macro language has limitations in

- interface calls to data base management systems' in other languages
- secrecy of algorithms
- implementation beyond the scope of the SAS language.

If your work site has a wide application for your program and is staffed with individuals familiar with the general SAS syntax, you may find that writing a SAS procedure is the most advantageous approach. Documentation is simplified since only the function of the procedure, special derivations, the output, and any new language statements need to be explained. The data used in your procedure or output from your procedure can be sorted, printed, manipulated in SAS data steps, and analyzed using other SAS procedures. This approach extends the capability of the SAS System to the special needs of your users.

Once you have decided to write a SAS procedure, where do you begin? The steps in writing a SAS procedure are:

- defining and writing the SAS grammar
- writing the application program that performs the function of the procedure
- linking the SAS grammar and application program together so that the SAS System can use the procedure.

Step 1: DEFINING AND WRITING THE SAS GRAMMAR
A grammar in the SAS System is the set of rules that define the format of the statements used to invoke your procedure. A parser is the part of the SAS System that analyzes statements entered by the user according to the grammar. The
The following SAS statements invoke PROC COUNT:

```sas
PROC COUNT DATA=TEMP SUM OUT=TEST;
VAR X Y Z;
BY T;
```

The grammar specifies that the DATA= option names a SAS data set to be used as input. The default input SAS data set is _LAST_, usually the last SAS data set created. The SUM option adds the values of all nonmissing numeric variables listed in the VAR statement. The OUT= option names a SAS data set to be created by the procedure. There is no default output SAS data set. The VAR statement gives numeric variables from the input SAS data set to be used by the procedure. The default VAR statement includes all numeric variables from the input SAS data set. The BY statement specifies the subsets by which the user wants the data processed. The BY statement requires that the input SAS data set be sorted in ascending order unless one of the keywords DESCENDING or NOTSORTED is given. The variables for the BY statement can be either numeric or character. A grammar for the modified PROC COUNT that is described is given below. Comments are delineated by the pound (#) symbol.

```
#include stubgrm.

# program production required
program = anystmt endjbs,

anystmt = countstmt | varstmt | bystmt

# * STATEMENT DEFINITIONS *

countstmt =@procinit # initialize
@stmitinit(II) # allocate statement structure
"count" countopt*
@dsdef(11,1) # default SAS data set name
@stmitend, # end structure

varstmt = @stmitproc # store in procedure
@stmlist(1,1) # numeric list
("var" "variables") varlist,

bystmt = @stmitproc # store in procedure
@stmlist(0.3) # mixed list
"by" byvarlist,

# * PROC OPTIONS *
# countopt* allows for multiple options.

countopt = ("data" "=>") dsfield # both strings required
| ("out" "=>" dsfield @ds(10,2)) # output SAS data set
| ("sum" @soft(0)) # sum of all numeric variables

# standard productions

#define * PROC OPTIONS *
"count" countopt*
#define * STATEMENT DEFINITIONS *
anystmt = countstmt | varstmt | bystmt

#include stubgrm.
```

The parser allows only the statements defined in the grammar or global statements (statements allowed anywhere in a SAS program). The parser ensures that the options, parameters, and data set names are entered as described in your grammar. The parser checks each variable's type (character or numeric) in statements that permit lists of variables. The parser also creates data structures that we call "statement structures" and stores in them information provided by the user. These structures are accessed by your procedure during its execution to determine which options, parameters, data set references, and statements were invoked. If the procedure reads a SAS data set, the statement structure may also contain one or more lists of variable names.

The SAS System passes control to your procedure as soon as it reads the procedure name in the PROC statement. The grammar that you have specified gives instructions for interpreting the rest of the procedure statements. Control is passed back to the SAS System when another PROC statement, DATA statement, RUN statement, or the end of the job is encountered.

The grammar specifies that the DATA= option names a SAS data set to be used as input. The default input SAS data set is _LAST_, usually the last SAS data set created. The SUM option adds the values of all nonmissing numeric variables listed in the VAR statement. The OUT= option names a SAS data set to be created by the procedure. There is no default output SAS data set. The VAR statement gives numeric variables from the input SAS data set to be used by the procedure. The default VAR statement includes all numeric variables from the input SAS data set. The BY statement specifies the subsets by which the user wants the data processed. The BY statement requires that the input SAS data set be sorted in ascending order unless one of the keywords DESCENDING or NOTSORTED is given. The variables for the BY statement can be either numeric or character. A grammar for the modified PROC COUNT that is described is given below. Comments are delineated by the pound (#) symbol.
Simply speaking, a grammar consists of a series of single rules that are applied during the parsing process. These rules are called productions and consist of the production name and the rule:

```
production name=rule, production name=rule, ..., production name=rule.
```

Productions are separated by commas. The end of the entire grammar is denoted by a period. We include a standard stub grammar (%INCLUDE STUBGRM.) to which we add productions for our procedure. In the PROC COUNT grammar given above, there are six productions specified (PROGRAM, ANYSTMT, COUNSTMT, VARSTMT, BYSTMT, and COUNTOPT). The rule in a production consists of terminals, lexicals, semantic actions, and other production names.

**TERMINALS**

are the keywords allowed by the procedure, including punctuation. Terminals are defined by enclosing the term to be specified by the user in double quotes. In the example above there are eight terminals that the grammar allows the user to specify ("COUNT", "VAR", "VARIABLES", "BY", "DATA", ",", "OUT", and "SUM").

**LEXICALS**

define the various categories of input for the procedure statements such as valid SAS names (NAME), numbers (INT, NBR), and quoted strings (QS). In the grammar above, the lexical ENDJB is a special case that indicates the end of the PROGRAM production that defines all of your statements. Other lexicals can be found in the stub grammar.

**SEMANTIC ACTIONS**

signal the parser to perform some action on the terminal or lexical associated with the semantic action and always begin with @. The semantic actions given in the grammar above are @PROCINIT, @STMTINIT, @STMTLIST, @DSDFLT, @STMTEND, @STMTPROC, @STMTPROXY, @STMTPROC, @STMTINIT, @STMTLIST, @DS, and @OPT.

**OTHER PRODUCTION NAMES**

indicate nesting of rules. These can be names that are defined in either your own grammar or the stub grammar. Standard productions that are part of the stub grammar and that can be used within productions are given in the SAS Programmer's Guide for PL/I, Version 5 Edition. Standard productions that we have used in the grammar above are VARLIST, BYVARLIST, and DSFIELD.

The statement %INCLUDE STUBGRM brings in a standardized stub grammar to which we add statements specific to the COUNT procedure. This stub contains several productions.

The first statement after %INCLUDE STUBGRM. is the program production that is required in a SAS grammar (PROGRAM=ANYSTMT ENDJB). ANYSTMT is a production name that expands into COUNSTMT | VARSTMT | BYSTMT and gives all possible statements that the procedure can have; that is, PROC COUNT has a PROC COUNT statement, a VAR statement, and a BY statement.

COUNSTMT defines the procedure statement. Within COUNSTMT we call @PROCINIT, which initializes a data structure that contains information about the parsing. @STMTINIT initializes and allocates a statement structure for the procedure where the parser stores options, parameters, data set names, and variable lists. The argument to @STMTINIT sets the size of the statement structure. "COUNT" defines the name of the procedure. COUNTOPT will be expanded, and the asterisk allows for multiple options. @DSDFLT sets up a default input (second argument) SAS data set whose location will be stored in the statement structure (first argument). @STMTEND ends the statement structure.

VARSTMT allows the user to enter "VAR" or "VARIABLES" as the term in the variables statement. The information for the variable list is found in the procedure's statement structure (@STMTPROC), which stores the location of the list in the statement structure (first argument of @STMTPROXY). We allow only numeric variables (second argument of @STMTPROXY) and have a list of attributes of variables (VARLIST).

BYSTMT follows the same course as the VARSTMT except it allows for numeric and character variables and has attributes of BY variables. COUNTOPT expands the options.

The user can specify an input SAS data set. "DATA" and "," are both required. DSFIELD is a standard production that sets up a SAS data set. $DS permits the user to specify a SAS data set. The first argument to $DS gives the location of the SAS data set in the statement structure. The second argument indicates that the SAS data set is for input; the third argument indicates that variables specified by users should be looked up in this SAS data set.

The output SAS data set follows the same course as the input SAS data set except "OUT" and "=" are required.

The user has an on/off option ("SUM"), which is stored in the statement structure.

The grammar is input to a grammar processor. The processor expands the grammar into a function (source code in PL/I). You compile the source code and call the function in your procedure. The way to invoke the grammar processor varies under different operating systems. You can refer to the SAS Programmer's Guide for PL/I, Version 5 Edition for the specific instructions for your environment.
Grammars become easier to write after you have studied several examples and are familiar with the coding, operators, lexicals, standard productions, and semantic actions. One approach is to write a simple grammar for your procedure and test this grammar with your application program before adding extra options, parameters, output SAS data sets, and so on. You will want to place the most effort on coding and testing your application program.

Step 2: WRITING THE APPLICATION PROGRAM

You have chosen to write your application program within the boundaries of the SAS System, which allows you to take advantage of SAS Software for such actions as checking and executing the user's syntax, processing data, dynamically allocating memory, printing output, and so on. You have written your grammar, which is the interface between you, the SAS System, and the SAS user. Programming your application program can be divided into (A) interacting with the SAS System and (B) coding your algorithm within the system boundaries.

(A) Interacting with the SAS System

In writing your application program, you need to use X-routines and sublib routines that are provided by the SAS System. X-routines are a fundamental part of writing SAS procedures. They allow you to perform I/O on SAS data sets, dynamically allocate memory, print output, and give information on observations and variables in SAS data sets. X-routines are grouped by name so that those with a common prefix perform related actions. The general categories of actions are:

- BY-group processing (XBY routines)
- communications area (XCOMGET routine)
- exit and errors (XE routines)
- formats and functions (XF routines)
- options (XGET routines)
- environment initialization (XINIT routine)
- writing to SAS log (XLOG routines)
- memory management (XM routines)
- SAS data set I/O (XO routines)
- printing output (XP routines)
- utility files (XR routines)
- parsing (XS and XT routines)
- variable lists and names (XV routines).

Sublib routines are additional routines that allow you to perform special tasks related to your program application that require special programming techniques to interface correctly with the SAS System.

The general requirements for interacting with the SAS System are:

- initializing the procedure environment
- calling the grammar function
- setting up to process the data
- processing data
- exiting.

Initializing the procedure environment

In writing your program, you need to declare data structures used by the SAS System and establish communications with the SAS System by calling the XINIT routine.

Calling the grammar function

You interact with the user by invoking the grammar function in the XSPARSE routine that checks the SAS statements the user entered and stores information in data structures about SAS data sets, variable lists, options, parameters, parsing information, and so on. You access the information in these data structures throughout your application program.

Setting up to process the data

In general, you have data from an input SAS data set that you want to process through your algorithm. You may have an output SAS data set that you want to create for further analyses using other procedures and data steps. You cannot read directly from an input SAS data set or write directly to an output SAS data set. You can indirectly access a SAS data set using XV routines. You set up the input to your procedure by (1) determining which variables (defined by the user) you want to access from the input SAS data set and (2) cross referencing these SAS variables to declarations in your application program. This process maps the variables defined in the algorithm to the variables defined by the user. Memory management routines (XM routines) are necessary to allow the user flexibility in the number of variables used in the procedure. Similarly, you set up the output SAS data set by linking variables that you declared in your program to variables in an output SAS data set.

Processing data

You have set up the links between the application program, the information provided by the user, and the SAS data set. The next task is to make observations available to your program for processing and to make observations.
produced by your program available for writing to a SAS data set. You get the data for processing by calling the XO routines, which read an observation, and by calling one of the XVGET routines to move the data from the observation into variables in your application program. To write to a SAS data set, you call one of the XVPUT routines, which gathers the values from your program, creates an output observation, and writes the observation to the SAS data set. You either read and process all of the observations in the SAS data set or read and process subsets of the SAS data set by calling the XBY routines. You usually read observations from the SAS data set in sequential order, but you can read a specified observation by calling the XOPNT routine. You can move in a SAS data set to the beginning of the current BY group or the beginning of the SAS data set by calling the XORWND routine.

Exiting

You exit from a procedure at any time by calling the XEXIT routine. The XEXIT routine closes all of the files and prints the current SAS log and listing pages (output listing from a SAS procedure). In a normal exit, you call the XEXIT routine at the end of the application program. In an error exit, you exit from the procedure when you detect the error. In either case, you can print additional messages on the SAS log by calling the XLOG routines.

The following source code is a modified version of the application program for PROC COUNT found in the SAS Programmer's Guide for PL/I, Version 5 Edition. Comments are delineated by /* */. We will go over the code in this application program in some detail so that you can see how the basic requirements for interacting with the SAS System are met. We will go over the code specific to interacting with the SAS System. By looking at this example and others in the SAS Programmer's Guide for PL/I, Version 5 Edition, you can see how to write your application program to interact with the SAS System.

1*----------------------------------------------------------------------*/
1* APPLICATION PROGRAM FOR PROC COUNT
1* MODIFIED VERSION FOR THIS PAPER
1* INCLUDE MACPORT;
COUNT: PROC PORTMAIN;
DCL {J,NBY,NVAR,NMISS,SUM,FLAG) FLOAT BIN(FL_DBL);
DCL DATADRBY CHAR(9) ,
DCL X(l) FLOAT BIN(FL DBL) BASED(XPTR), XPTR PTR;
DCL COUNTG ENTRY RETURNS(PTR):
1* PARSING INFORMATION
%INCLUDE XCOM;
1* COMMUNICATIONS AREA
%INCLUDE XLINK:
1* ALL X-ROUTINES EXCEPT XINIT, XCOMGET
%INCLUDE STMTSTR;
1* STORAGE FOR PARSED INFORMATION
%INCLUDE PROCPRS;
1* GENERAL PARSING INFORMATION
%INCLUDE NAMESTR;
1* SAS VARIABLES
%INCLUDE $MISSN:
%INCLUDE $MISS:
1* CHECKS FOR MISSING VALUES
%INCLUDE $MISSV; SET VALUE TO MISSING VALUE
DCL ADDR BUILTIN;
CALL XINIT:
XCOMPTR=XCOMGET();
CALL XSPARSE(COUNTG());
CALL XTOKE;
IF PROCERRORh=O THEN CALL XEXIT(PROCERROR,O);
STMTPTR=PROCPTR;
FIELDPTR=ADDR(STMTLD(11));
DATAPTR=FIELDFILE;
/* GET FILE ID FOR DATA= DATA SET */
IF DATAPTR=NULL() THEN DO;
CALL XLOG1('ERROR: NO INPUT DATA SET.);
/* WRITE TO LOG */
CALL XEXIT(XEXITSEMANTIC,O);
/* EXIT - SEMANTIC ERROR */
END;
FIELDPTR=ADDR(STMTLD(10));
/* OUT= DATA SET OPTION */
IF STMTLDTYPE(10)=O THEN OUTFILE=NULL();
ELSE OUTFILE=FIELDFILE;
FIELDPTR=ADDR(STMTLD(8));
/* SET UP BY STMT LIST */
CALL XVLST(DATAPTR,8,FIELADDR,(FIELDS));
FIELDPTR=ADDR(STMTLD(1));
/* SET UP VAR STMT LIST */
CALL XVLIST(DATAPFILE,1,FIELDADDR,FIELDN);  
CALL XVDFLST(DATAPFILE,1);  */ SET UP DEFAULT VAR LIST */  
NVAR=XVCOUNT(DATAPFILE,1);  */ COUNT VARIABLES IN VAR */  
NBY=XVCOUNT(DATAPFILE,8);  */ COUNT VARIABLES IN BY */  
IF NBY=0 THEN DATAORBY='DATA SET.';  
ELSE DATAORBY='BY GROUP.';  
XPTR=XMEMEX(NVAR*SIZE~FL~DBL);  /* ALLOCATE MEMORY. EXIT IF FAILS */  
CALL XVGETI(DATAPFILE,NVAR,XVGETPTR);  /* INITIALIZE INPUT */  
DO J=1 TO NVAR;  */ TELL WHERE TO PUT THE VALUE OF EACH VARIABLE */  
   CALL XVGETD(XVGETPTR,1,J,ADDR(X{J}),SIZE~FL_DBL, 'F');  
END;  
IF STMTFLDTYPE(10)=0 THEN DO;  /* SET UP OUTPUT DATA SET */  
   WORKPTR=XMEMEX(XVNAMEL());  
   CALL XVNAMEI(WORKPTR),  
   NAMESTRPTR=WORKPTR;  
   /* SET UP OUTPUT DATA SET */  
   CALL XVPUTI(OUTFILE,3,XVOUTPTR);  /* INITIALIZE OUTPUT */  
   NNAME='COUNT'; NTYPE=1; NLNG=SIZE_FL_DBL;  
   CALL XVPUTD(XVOUTPTR,NAMESTRPTR,ADDR(NREAD),NLNG);  
   NNAME='MISS'; NTYPE=1; NLNG=SIZE_FL_DBL;  
   CALL XVPUTD(XVOUTPTR,NAMESTRPTR,ADDR(NMISS),NLNG);  
   NNAME='SUM'; NTYPE=1; NLNG=SIZE_FL_DBL;  
   CALL XVPUTD(XVOUTPTR,NAMESTRPTR,ADDR(SUM),NLNG);  
   CALL XVPUTE(XVOUTPTR);  
END;  
DO WHILE(XBYNEXT(DATAPFILE)>0);  /* BY-GROUP LOOP */  
   NREAD=0; NMISS=0; SUM=0; FLAG=0;  
   DO WHILE(XOGET(DATAPFILE)=0);  /* OBSERVATION LOOP */  
      NREAD=NREAD+1;  
      CALL XVGET(XVGETPTR);  
      NMISS=NMISS+$MISSN(XPTR,NVAR);  
      IF STMTOPTS(0) THEN DO;  /* SUM OPTIONS */  
         DO J=1 TO NVAR;  
            IF $MISS(X(J))=0 THEN DO; FLAG=1; SUM=SUM+X{J}; END;  
            END;  
         END;  
      END;  
   END;  
   CALL XLOG3('NOTE: # OBSERVATIONS IN #',XPI12(NREAD),DATAORBY);  
   IF NMISS>0 THEN  
      CALL XLOG1('NOTE: # OBSERVATIONS HAD MISSING VALUES.',  
                  XPI12(NMISS));  
   IF STMTOPTS(0) THEN DO;  
      IF NVAR=0 THEN CALL XLOG1('NOTE: NO NUMERIC VARIABLES');  
      ELSE IF FLAG=0 THEN CALL XLOG1('NOTE: ALL NUMERIC VARIABLES MISSING FOR #',  
                                           DATAORBY);  
      ELSE CALL XLOG1('NOTE: # IS THE SUM OF NUMERIC VARIABLES IN #',  
                           XPI12(SUM),DATAORBY);  
      END;  
   IF OUTFILE=$NULL() THEN DO;  /* OUTPUT */  
      WORKPTR=XORPTR(DATAPFILE);  
      IF FLAG=0 THEN SUM=$MISSV(' ');  
      CALL XVPUT2(XVOUTPTR,WORKPTR);  
      END;  
END;  
CALL XEXIT(XEXITNORMAL,0);  /* NORMAL EXIT */  
END COUNT;  
/* ------------------------------------------ */

We begin by declaring data structures and entry points that the SAS System and the COUNT procedure require. Most data structures used within the SAS System are BASED. This is an important concept to understand because it permits dynamic allocation of memory each time a procedure is invoked using the SAS System for direction. Dynamic allocation avoids allocating quantities of memory that may be either excessive or insufficient. To use a based data structure, you must do the following:
• declare the structure as based on the value of a pointer variable

• declare the pointer on which the data structure is BASED

• issue commands to allocate space for the structure in memory and store the address in the pointer variable, or store the address in the pointer variable of a block of memory that has already been allocated.

MACPORT is a set of special statements that contain preprocessor symbols used by all data structures in the SAS System. MACPORT also provides portability of the code across operating systems. This set of statements is included at the beginning of your application program:

```
%INCLUDE MACPORT; /* data structure */
```

Examples of the portability allowed by MACPORT can be found in the main procedure statement (COUNT:PROC PORTMAIN), an assignment statement (NLANG= SIZE_FL DBL), and a declaration statement (DCL X(1) FLOAT BIN(FL DBL)). PROC PORTMAIN will expand to PROC OPTIONS(MAIN,REENTRANT) ORDER under the IBM systems; NLANG=SIZE_FL DBL will expand to NLNG=8; and DCL X(1) FLOAT BIN(FL DBL) will expand to OCL X(1) FLOAT BIN(53).

XCOM gives the main communications structure with the SAS System:

```
%INCLUDE XCOM; /* communications area */
```

It contains pointers to various structures and entries that your procedure needs (X-routines). Declarations for the X-routines are included:

```
%INCLUDE XINIT; /* init X-routines */
%INCLUDE XLINK; /* all X-routines */
```

XLINK brings in all of the X-routines except XINIT.

We include the declaration for the based statement structure:

```
%INCLUDE STMTSTR; /* statement struct */
DCL STMPTR PTR;
```

This structure contains the data filled in by parsing and is BASED on the STMPTR pointer.

Declarations for the based parsing structure are included:

```
%INCLUDE PROCPRS; /* parsing struct */
```

This structure contains information about the results of the parsing and is BASED on a pointer stored in the XCOM structure.

The variable name structure (based data structure defined by NAMESTRPTR) contains the name and attribute information for variables in a SAS data set:

```
%INCLUDE NAMESTR; /* variable name struct */
DCL NAMESTRPTR PTR;
```

X-routines use the data in this structure to read and create SAS data sets.

We declare the entry point for the grammar function:

```
DCL COUNTG ENTRY RETURNS(PTR);
```

The following code declares sublib routines that are needed specifically for the COUNT application program:

```
%INCLUDE $MISSN;
%INCLUDE $MISS;
%INCLUDE $MISSV;
%INCLUDE $NULL;
```

$MISSN determines if any of the entries in a numeric array of variables have missing values. $MISS determines if a numeric variable has a missing value. $MISSV returns a missing value. $NULL sets a pointer to null.

The first step after your declarations is to establish communications with the SAS System:

```
CALL XINIT;
XCOMPTR=XCOMGET();
```

The XINIT routine completes the linkage to the X-routines and is called only in the main procedure. XCOMPTR=XCOMGET() sets up the main communications structure (XCOM) and must be called in the first executable statement of any module that interacts with the SAS System.

You need to invoke the grammar so you can see and check what the user has entered and store the information in statement structures. If an error is found, you exit from the procedure and error messages are printed on the SAS log:

```
CALL XSPARSE(COUNTG());
CALL XTOKE;
IF PROCERROR=0 THEN
CALL XEXIT(PROCERROR,0);
STMTPTR=PROCPTR;
```

STMTPTR=PROCPTR tells the program the location of the procedure's statement structure. The COUNT procedure statement structure indicates if the user had an input/output SAS data set, SUM option, BY statement, and VAR statement. The information in the statement structure is either data values (numeric parameters) or the location of the item of interest (variables, data sets, character parameters, and so on). Alternate formats are used in getting the location of the item of interest from the statement structure. The following is a listing of the declarations (%INCLUDE STMTSTR) for the statement structure and some of the alternate formats. A description is given by each variable as to its use.
/* ----------------- DECLARE STATEMENT STRUCTURE ---------------------*/
DCL I STMSTR BASED(STMTPTR),
  2 STMTPTR PTR, /* location of next statement struct */
  2 STMTPNAME CHAR(8), /* name of statement structure */
  2 STMTPOPTS(BF:63) BIT(1), /* on/off options */
  2 STMTPFLD FIXED BIN(FX_DBL), /* # of STMTPFLDS (STMTINIT field) */
  2 STMTPFLDS(100),
  3 STMTPFLDTYPE FIXED BIN(FX_DBL), /* type of info (format, list, */
  /* parameter, data set */
  2 STMTPFLDMODE FIXED BIN(FX_DBL), /* more on above */
  3 STMTPFLD FLOAT BIN(FL_DBL) ; /* either actual value or */
  /* location of actual value */
/* ------------- ALTERNATE FORMATS OF STATEMENT FIELDS ---------------*/
/* If the location of the data is stored in the STMTPFLD, we will use */
/* the following structure. */
DCL FIELD PTR,
  FIELDCHAR CHAR(8) BASED(FIELDPTR),
  FIELDFLOAT FLOAT BIN(FL_DBL) BASED(FIELDPTR),
  FIELDFILE PTR BASED(FIELDPTR);
/* List of data or variables */
DCL 1 FIELDLIST BASED(FIELDPTR),
  2 FIELD FIXED BIN(FX_SNG),
  2 FIELDK FIXED BIN(FX_SNG),
  2 FIELDADDR PTR;
/* We do the same for the SAS output data set */
/* except since we do not have a default output SAS */
/* data set for PROC COUNT, we set a null location */
/* if the user did not indicate an output SAS data */
/* set. */
FIELDPTR=ADDR(STMTFLD(10));
IF STMTPFLDTYPE(10)=0 THEN
  OUTFILE=$NULL();
ELSE OUTFILE=FIELDFILE;
For PROC COUNT, we have stored the location of
/* the SAS output data set in the tenth element of */
/* the STMTPFLD array in the statement structure. */
We create variable lists for the BY statement
/* and VAR statement BASED on the variables entered */
/* by the user. If the user does not enter any */
/* variables for the VAR statement, we set up a */
/* default list of all of the numeric variables in */
/* the input SAS data set. */
FIELDPTR=ADDR(STMTFLD(8));
CALL XVLIST(FIELDPTR,FIELDN,.FIELDADDR,FIELDN);
FIELDPTR=ADDR(STMTFLD(1));
CALL XVDFLST(FIELDPTR,FIELDN,.FIELDADDR,FIELDN);
CALL XVDFLST(FIELDPTR,FIELDN,FIELDADDR,FIELDN);
These lists are used throughout your code by
other X-routines.
We will be reading and processing one
observation at a time from a SAS data set.
Since we are indirectly reading the data, we
need to store the values for the variables in an

The information stored in a statement structure,
the length of the statement structure, and the
number of statement structures are defined in
your grammar. You will always have one
statement structure (STMTPNAME= name of
procedure) that contains the information about
the PROC statement. You can have more than one
statement structure that contains information
about the other statements for your procedure,
or you can store all of your information in the
PROC statement structure. For PROC COUNT, we
stored all of our information in the PROC
statement structure. We use the information
stored in the statement structure throughout the
application program.

In the following code, we are setting up to read
from a SAS data set by finding the location of
the input SAS data set. In PROC COUNT, we have
stored the location of the SAS data set in the
eleventh element of the STMTPFLD array in the
statement structure (defined in the grammar),
and by using an alternate format, we store the
location of the SAS data set in DATAPFILE so that
we can work with the SAS data set using
X-routines.

FIELDPTR=ADDR(STMTFLD(11));
DATAPFILE=FIELDFILE;

If the user did not specify a SAS data set, we
have specified in the grammar that the location of
the last created SAS data set should be
found.
array (X) that we declared in our program. If we knew how many variables we had for each input SAS data set, we could declare the array and the number of elements using automatic storage. Since we do not know how many variables are in the input SAS data set, we create a base variable (X) and dynamically create the memory.

\[{\text{NVAR}}=\text{XV_COUNT}({\text{DATAFILE}},1);\]
\[{\text{XPTR}}=\text{XMEMEX}({\text{NVAR}}\times \text{SIZE_FL_DBL});\]

We complete the process of setting up the input data set by initiating the process and establishing that the data for each variable will be stored in the X array (NVAR elements).

\[{\text{CALL XVGETI}}({\text{DATAFILE}},\text{NVAR},\text{XVGETPTR});\]
\[{\text{DO J=1 TO NVAR;}}\]
\[{\text{CALL XVGETD(XVGETPTR,1,J,ADDR(X(J)),}}\]
\[{\text{SIZE_FL_DBL,}'F'^{;}\}];\]
\[{\text{END;}}\]

We need to set up the environment if we have an output SAS data set. The NAMESTR name structure contains the name and attribute information for a variable in a SAS data set. When a data set is opened, a NAMESTR name structure is created for each variable. You must not change a NAMESTR name structure from an input data set. If you need to change a NAMESTR name structure for an output SAS data set, you must create a new one and change the values in the new structure. We are creating three new variables for our output SAS data set; therefore, we will create a new NAMESTR name structure.

\[{\text{DO WHILE}}({\text{XBYNEXT}}({\text{DATAFILE}})>0);\]
\[{\text{IF STMTOPTS(10)^{;}}=0 \text{ THEN DO:}}\]
\[{\text{WORKPTR}=\text{XMEMEX}({\text{XVNAMEI}}());}\]
\[{\text{CALL XVNAMEI(WORKPTR);}}\]
\[{\text{NAMESTRPTR}=\text{WORKPTR};}}\]
\[{\text{CALL XVPUTI(OUTFILE,3,XVOUTPTR);}}\]
\[{\text{NAME}'=\text{COUNT}; \text{ NTYP}=1; \text{ NLNG}=\text{SIZE_FL_DBL};}\]
\[{\text{CALL XVPUTD(XVOUTPTR,NAMESTRPTR,}}\]
\[{\text{ADDR(NREAD),NLNG)}};\]
\[{\text{NAME}'=\text{MISS}; \text{ NTYP}=1; \text{ NLNG}=\text{SIZE_FL_DBL};}\]
\[{\text{CALL XVPUTD(XVOUTPTR,NAMESTRPTR,}}\]
\[{\text{ADDR(NMISS),NLNG)}};\]
\[{\text{NAME}'=\text{SUM}; \text{ NTYP}=1; \text{ NLNG}=\text{SIZE_FL_DBL};}\]
\[{\text{CALL XVPUTD(XVOUTPTR,NAMESTRPTR,}}\]
\[{\text{ADDR(SUM),NLNG)}};\]
\[{\text{CALL XVPUTE(XVOUTPTR);}}\]
\[{\text{END;}}\]

XVNAMEI returns the length of the current NAMESTR name structure. We allocate the memory for another structure, initialize the structure, and set the pointer. XVPUTI initializes the output process. We establish the name, type, and length for each variable. We call XVPUTD to set where to find the values for variables when an observation is written. XVPUTE completes the process of establishing the output SAS data set.

We have set up the environment and we are now ready to read from a SAS data set, analyze, and write to our output SAS data set for PROC COUNT. We will process either all of the data or subsets of the data (BY statement).
XBYNEXT moves to the beginning of the next BY group. XOGET reads the observation from the file. XQGET flags the end of the file or the end of the BY group. XVGET moves the observation from the input buffer to the program's storage area (X array). STMTOPTS(O) is the bit that we have designated in the grammar to indicate whether the SUM option is in effect. If we create an output SAS data set, we call XORPTR, which returns the location of the last successfully read observation and call XVPUT2 which writes from the output buffer to the output SAS data set.

Your application program interacts with the SAS System. The next step is to code your algorithm within the boundaries of the SAS System.

(B) Coding your algorithm within the system boundaries

In writing your algorithm, it is most helpful to become familiar with the X-routines and sublib routines that are available in the SAS System. The SAS Programmer's Guide for PL/I, Version 5 Edition documents these routines. As we have gone through the application program for the COUNT procedure, we have reviewed some of the routines that are necessary and others that make writing the application program easier. In writing many procedures, memory management becomes an important issue and the XM routines that are available in the SAS System should be reviewed and understood. There are multiple ways of handling memory and the method you choose will affect the way you code your algorithm.

Step 3: LINKING TO THE SAS SYSTEM

The last step is to compile and link your source code with the SAS System so that the SAS System can access your procedure when a user invokes it. The details for linking to the SAS System are operating system dependent. Check the SAS Programmer's Guide for PL/I, Version 5 Edition for more information.

CONCLUSION

We have discussed writing SAS procedure for Version 5. You can write a SAS procedure and take advantage of the SAS System, but you need to understand how to fit your algorithm into the boundaries of the SAS System. This is not a simple task but is well worth your effort in that you are extending the capacity of the SAS System to meet your needs.

SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA.