

AN INTRODUCTION TO THE SAS[®] MACRO FACILITY

Erika M. Stone
Carolina Population Center
The University of North Carolina at Chapel Hill

INTRODUCTION AND GOALS

The SAS macro facility is important for two reasons. First, it can significantly expand your capabilities as a programmer; second, because it is part of base SAS, it is available to you on all platforms. My aim in this tutorial is to give you a sense of how the SAS macro facility works and how it can help you in your programming.

You will not get a step-by-step lesson in how to construct macro language statements, use the automatic macro variables, or understand the macro functions. Rather, by a sequence of examples and by the examination of concepts, including the principles by which the macro facility works, I want to show you that you have a potentially valuable tool at your disposal. You will see that you don't have to aspire to be a systems builder or macro guru to justify learning the SAS macro facility. Knowing the potential of the macro facility and learning how to take advantage of it can have far-reaching benefits for your programming, since the macro facility will let you do things you cannot do as easily in any other way. This will be true whether you work in a production or a research environment, and regardless of your area of application.

THE BENEFITS OF THE SAS MACRO FACILITY

So what are the rewards of understanding the SAS macro facility? The papers presented at SUGI meetings in the last five years either state explicitly or imply over two dozen benefits of the macro facility to SAS programmers. Of so many possible motivations, I will focus on those three which I see as being directly related to writing better programs and, consequently, of potential interest to all programmers. The macro facility makes possible (1) modularity in program design, (2) reduction of keyed program text, and (3) generality in program function. All three, alone or in combination, can result in better programs. Let's take them one by one.

Modularity in program design means that you think of your program in terms of distinct subtasks and translate each into a block of code which can be individually and systematically tested and debugged. This approach, making use of program building blocks, can result in programs that are more quickly debugged, programs whose logic is more clearly laid out, and programs that are more easily modified and maintained. The potential is for programs that run correctly in less programming time and are modifiable with less programming effort; in other words, better programs.

Reducing the amount of keyed program text, or source code, results in shorter programs that are less error-prone to key; that is, better programs. Finally, generality in program function means that your programming effort results in more flexible, reusable, or sharable code, useful to other programs or other programmers--again, better programs.

Let us first examine the principles by which the macro facility works and then turn to some real examples to illustrate these ideas.

HOW TO THINK OF THE SAS MACRO PROCESSOR AS A TEXT GENERATOR

Programs with and without macro activity

A program that does not make use of the SAS macro facility is said to be "without macro activity." That is, such a program does not request work from the macro processor, one of the two components of the macro facility along with the macro language. When you submit a program without macro activity to the SAS system, you think primarily of communicating with the SAS compiler--that part of the system that checks the syntax of your SAS language statements and translates your program, one DATA or PROC step at a time, into a form the computer can execute. While you have no choice in the matter of dealing with the SAS compiler, whether or

not you deal with the SAS macro processor is up to you to decide.

In a program with macro activity, recognizable by the use of the symbols % and &, you communicate not only with the compiler, but also with the SAS macro processor. You can think of the macro processor as a part of the SAS system separate from the compiler and between you and the compiler. With macro language statements, you can request the macro processor to help you build the program code that the compiler will see. The principles by which the SAS macro processor functions as a text, or code, generator can be illustrated by very simple examples.

Letting the macro processor generate code

Suppose you want a 2-way frequency table for the variables A, B in the SAS data set ALPHA stored in a SAS data library referenced by IN. Let's look at three different ways you could write your program and achieve the same result--the first (Version 1.) without macro activity, and the second and third (Version 2. and Version 3.) with macro activity:

Version 1. Program without macro activity

```
PROC FREQ DATA=IN.ALPHA;
TABLES A*B;
RUN;
```

Version 2. Program with macro activity

```
%MACRO TBL;
  PROC FREQ DATA=IN.ALPHA;
  TABLES A*B;
  RUN;
%MEND TBL;

%TBL
```

Version 3. Program with macro activity

```
%MACRO TBL(MEMBER);
  PROC FREQ DATA=IN.&MEMBER;
  TABLES A*B;
  RUN;
%MEND TBL;

%TBL(ALPHA)
```

In Version 1., the compiler compiles the two statements of the PROC step and the program is executed. In Version 2. and Version 3., we achieve the same result by requesting the macro processor to generate the code that the compiler will see.

In Version 2., the complete PROC step is packaged into a unit of code called a "macro" to which the name TBL is given. The macro language statement %MACRO TBL; begins the definition of the macro named TBL, and the statement %MEND TBL; ends it. The body of the macro in this example consists of pure SAS language statements.

When the program in Version 2. is submitted to the SAS system, the percent sign, followed immediately by the word MACRO, activates the macro processor, which compiles the macro and stores it for possible later use. The %TBL statement triggers the macro processor again, this time requesting it to execute the macro TBL. The execution of the macro TBL causes the SAS language statements which form the body of the macro to be inserted at the point where the macro is called, or invoked, by the statement %TBL and thus placed in the stream of text directed to the compiler. The compiler never sees the statements addressed to the macro processor; instead, it sees the text generated by the macro processor, namely the code:

```
PROC FREQ DATA=IN.ALPHA;
TABLES A*B;
RUN;
```

In Version 3., the macro TBL is written with one parameter named MEMBER. A parameter is a special case of a macro variable. Such variables belong to the macro language and their values are strings. In the body of the macro TBL the symbol &MEMBER refers to the current value of the macro variable MEMBER. The current value of MEMBER is determined from the statement that invokes, or calls, the macro. That is, the statement %TBL(ALPHA) asks the macro processor to execute the macro TBL using the string ALPHA as the value of the macro variable MEMBER. The result is that the code the compiler sees is again exactly the same as in the first two versions of the program.

Macros written with parameters, as in Version 3. above, are flexible. They can be invoked many times in one program with different values for the parameters. In this case macro TBL could be invoked more than once in the same program to produce a sequence of 2-way frequency

tables for the variables A, B in different SAS data sets in the same SAS data library.

The three programs above, which result in identical executable code, differ in how much work the macro processor is requested to do. In the first example, the macro processor is ignored and hence does no work; in the second and third examples the macro processor is engaged to generate the code--in this case all the code--that will be compiled and executed. It is this text-generating capability of the macro processor that gives us the potential to write not only more interesting, but also better, programs, as we will see in the sections which follow.

THE POTENTIAL OF THE SAS MACRO FACILITY ILLUSTRATED

Background

Programmers at the Carolina Population Center work with survey data. The Center receives data from all over the world, either as the result of original data collection efforts on projects or from large national and international surveys that have been released to the public for research purposes. Data sets range in size from several thousand observations to millions of observations. Recognizing the superior ability we have with SAS to manipulate, transform, combine, and analyze data, the data on all major projects at the Center is subject to SAS processing.

All the examples which follow come from work with survey data. We will introduce them in a sequence of scenarios in which there will be three characters: you and your program, the SAS macro processor (MP), and, in the background, the SAS compiler. In each situation you will have a choice: to use the macro facility or not. The SAS macro processor, endowed with a power of speech we all wish it had in real life, will tell you which way to go.

First example: breaking up a tedious task

In this example you will see macros used in their simplest form--i.e., to package portions of pure SAS code--in order to systematically construct, test, and debug a DATA step which is conceptually simple but extremely tedious.

You have obtained raw data from abroad, data at the community level for a country you are studying. You want to convert this data as quickly as possible to a SAS data set. Conceptually, the task is as simple as can be, yet on closer examination it has the following characteristics which make it more tedious than you like.

There are seventeen records per case (per community) and close to one thousand variables. The variable names have been selected to facilitate easy referencing of the variables in the survey documentation. The problem is that you have about one thousand names to key and doing so will be very tedious since they look like this:

```
SA1 SA2 .. SA7_9 SA10_13... SA87_89 ... SA94_96
... SB10_12 SB94_96 ... SG31_32 SG33 SG34_35 ...
SP15_16 SP17_18_ .... SP94_95 ... SQ25 SQ26 SQ27
.....
```

Your task to create a SAS data set from this data boils down to writing a tedious and error-prone INPUT statement.

You: How can I proceed in a systematically debuggable way?

MP: Break it up! Break it up!

You: How?

MP: Focus on the variables coming from one record of data at a time.

You read in the "SAS Guide to Macro Processing" (which you discover stands beside the "SAS Language Reference" on your book shelf) how you can package code into a macro. You decide to package up the variable names and column locations from one record at a time into separate macros, one per record, and test your INPUT statement in stages. Your macro for the variables from the first record, which you name LINE1, looks like this:

%MACRO LINE1;

```
SA1 1 SA2 2 VNUM $ 3-4 VNAME $ 5-6 SA7_9 7-9
SA10_13 10-13 SA14_16 14-16 SA17_19 17-19
SA20_20 SA21_22 21-22 SA23_24 23-24
SA25 25 SA26_27 26-27 SA28_28 SA29_30 29-30
SA31_32 31-32 SA33_34 33-34 SA35_36 35-36
SA37_38 37-38 SA39_40 39-40 SA41_42 41-42
SA43_44 43-44 SA45_45 SA46_46 SA47_47
SA48_49 48-49 SA50_51 50-51 SA52_53 52-53
SA54_55 54-55 SA56_57 56-57 SA58_59 58-59
```

```
SA60 60 SA61 61 SA62 62 SA63 63 SA64 64 SA65 65
SA66 66 SA67 67 SA68 68 SA69 69 SA70 70
SA71 72 71-72 SA73 75 73-75 SA76 76 SA77 77
SA78 79 78-79 SA80 82 80-82 SA83 83 SA84 84
SA85 86 85-86 SA87 89 87-89 SA90 90 SA91 91
SA92 93 92-93 SA94 96 94-96 SA97 97
```

```
%MEND LINE1;
```

You test a partially constructed INPUT statement, reading only the variables from the first record (of seventeen records) for each case, by invoking the macro LINE1:

```
OPTIONS MPRINT;
```

```
DATA VILL88;
  INFILE IN;
  LENGTH DEFAULT=4;
```

```
INPUT
  #1 %LINE1
  #17
;
RUN;
```

```
PROC CONTENTS;
PROC FREQ;
```

As a result of the MPRINT system option, you can see in the SAS log exactly what the macro processor does for you. The macro processor generates code which is inserted into your DATA step at the point of invocation of your macro and before the compiler compiles your DATA step. The code in this case consists of the body of the macro LINE1 and appears in the SAS log following the call to macro LINE1:

```
DATA VILL88;
  INFILE IN;
  LENGTH DEFAULT=4;
```

```
INPUT
  #1 %LINE1
```

```
MPRINT(LINE1): SA1 1 SA2 2 VNUM $ 3-4 VNAME
$ 5-6 SA7_9 7-9 SA10_13 10-13 SA14_16 14-16
SA17_19 17-19 SA20 20 SA21 22 21-22 SA23_24 23-24
SA25 25 SA26_27 26-27 SA28 28 SA29_30 29-30
SA31_32 31-32 SA33_34 33-34 SA35_36 35-36 SA37_38
37-38 SA39_40 39-40 SA41 42 41-42 SA43_44 43-44
SA45 45 SA46 46 SA47 47 SA48_49 48-49 SA50_51
50-51 SA52_53 52-53 SA54_55 54-55 SA56_57 56-57
SA58_59 58-59 SA60 60 SA61 61 SA62 62 SA63 63
```

```
SA64 64 SA65 65 SA66 66 SA67 67 SA68 68 SA69 69
SA70 70 SA71 72 71-72 SA73 75 73-75 SA76 76 SA77
77 SA78 79 78-79 SA80 82 80-82 SA83 83 SA84 84
SA85 86 85-86 SA87 89 87-89 SA90 90 SA91 91
SA92 93 92-93 SA94 96 94-96 SA97 97
```

```
#17
```

```
;
RUN;
```

Using the Proc Contents listing and the frequency tables produced by your run, you correct any bugs in this part of the INPUT statement. After debugging your program to this point, you add a second macro named LINE2 which packages up the names and column locations for the variables from the second record. To test the next part of the INPUT statement, you comment out the first macro invocation and run the DATA step as follows:

```
DATA VILL88;
  INFILE IN;
  LENGTH DEFAULT=4;
```

```
INPUT
  /* #1 %LINE1 */
  #2 %LINE2
  #17
;
RUN;
```

```
PROC CONTENTS;
PROC FREQ;
```

You move systematically through the variable names and their column positions building the INPUT statement and testing and debugging as you go along. Finally you are ready to create the permanent SAS data set with the following program:

```
/* ** MACRO DEFINITIONS ** */
```

```
%MACRO LINE1;
```

```
%MEND LINE1;
```

```
%MACRO LINE2;
```

```
%MEND LINE2;
```

```

%MACRO LINE3;
.
.
%MEND LINE3;
.
.
%MACRO LINE17;
.
.
%MEND LINE17;

/* END MACRO DEFINITIONS */

DATA OUT.VILL88;
  INFILE IN;
  LENGTH DEFAULT=4;

  INPUT
    #1  %LINE1
    #2  %LINE2
    #3  %LINE3
    #4  %LINE4
    #5  %LINE5
    #6  %LINE6
    #7  %LINE7
    #8  %LINE8
    #9  %LINE9
    #10 %LINE10
    #11 %LINE11
    #12 %LINE12
    #13 %LINE13
    #14 %LINE14
    #15 %LINE15
    #16 %LINE16
    #17 %LINE17
  ;
RUN;
PROC CONTENTS;

```

Note that the program code is in two sections--first the macro definitions and then a main part in which the macros are invoked. The INPUT statement was constructed in stages using macros as building blocks. The main motivation for engaging the macro processor in this task was to break up a tedious INPUT statement and to test and debug systematically, focusing on one part of the problem at a time.

Second example: taking advantage of a pattern in names

In this example the macro processor is used to generate variable names and thus reduce the amount of keyed program text.

Your task is to make a large selection of variables from an existing SAS data set in which the variables have awkward names (through no fault of your own). This is a large national survey for which the data has been released as a SAS data set with variable names selected to relate to a survey instrument. Again, your task is conceptually simple but it is tedious and error-prone in the following sense: you want to insert 193 variables names into the KEEP= option in the SET statement and the variable names look like this:

```

RTIID FI7E1BM FI7E1BY FI7E1EM FI7E1EY
FI7E2BM FI7E2BY FI7E2EM FI7E2EY FI7E3BM
FI7E3BY FI7E3EM FI7E3EY FI7E4BM FI7E4BY
FI7E4EM FI7E4EY ..... FI17A1BM FI17A1BY
FI17A1EM FI17A1EY ..... FI19B4EM FI19B4EY .....
FI20B1BM FI20B1BY .....

```

The problem is you don't really want to key all those names. You hear the macro processor:

MP: Try to find a pattern! Try to find a pattern!
 You: What kind of pattern?
 MP: That's up to you.

So you stare at the variable names and patterns begin to emerge. Logically, besides the variable RTIID, the variables happen to fall into three groups. They are the respondent's job related dates (128 variables), unemployment related dates (32 variables), and school attendance dates (32 variables). In the first group of 128 variables the following pattern repeats itself three more times with 8, 9, and 10 in place of 7 in the third character position of the variable names:

```

FI7E1BM FI7E3BM FI7E5BM FI7E7BM . . . .
FI7E1BY FI7E3BY FI7E5BY FI7E7BY . . . .
FI7E1EM FI7E3EM FI7E5EM FI7E7EM . . . .
FI7E1EY FI7E3EY FI7E5EY FI7E7EY . . . .
FI7E2BM FI7E4BM FI7E6BM FI7E8BM . . . .
FI7E2BY FI7E4BY FI7E6BY FI7E8BY . . . .
FI7E2EM FI7E4EM FI7E6EM FI7E8EM . . . .
FI7E2EY FI7E4EY FI7E6EY FI7E8EY . . . .

```

You: OK, now that I see patterns, what next?
 MP: You have to tell me how to generate the patterns.
 You: How do I do that?
 MP: With the SAS macro language.

So you search through the "SAS Guide to Macro Processing" again. . . . After some time you come up with a sequence of %DO loops and you want to see if the macro processor understands your macro language instructions to generate the variable list. You begin to key the first nested %DO loops inside your DATA step like this:

```
DATA ACTDATA;
  SET INSAS.FU5NLS;

  %DO J=7 %TO 10;
    %DO I=1 %TO 8;
```

MP: Stop! Stop! You can't put %DO in open code!!

Finally, you write this program which does the job:

```
/* ** M A C R O   D E F I N I T I O N S   * * */
%MACRO JOBS; /*generate 128 variable names*/
  %DO J=7 %TO 10;
    %DO I=1 %TO 8;
      FI&J.E&I.BM FI&J.E&I.BY
      FI&J.E&I.EM FI&J.E&I.EY
    %END;
  %END;
%MEND JOBS;

%MACRO NOJOBS; /*generate 32 variable names*/
  %DO I=1 %TO 8;
    FI17A&I.BM FI17A&I.BY
    FI17A&I.EM FI17A&I.EY
  %END;
%MEND NOJOBS;

%MACRO SCHOOLS; /*generate 32 variable names*/
  %DO J=19 %TO 20;
    %DO I=1 %TO 4;
      FI&J.B&I.BM FI&J.B&I.BY
      FI&J.B&I.EM FI&J.B&I.EY
    %END;
  %END;
%MEND SCHOOLS;
/* E N D   M A C R O   D E F I N I T I O N S   */
```

```
DATA ACTDATA;
  SET INSAS.FU5NLS(KEEP=RTIID %JOBS
                  %NOJOBS %SCHOOLS);
```

The main motivation for engaging the macro processor in this task was to **reduce the amount of keyed text**. In fact, except for the variable RTIID, you didn't have to key a single one of the other 192 variable names that you did not want to key. The macro processor generates lists of variable names from your macro language instructions and inserts the names into the KEEP= option at the point of invocation of your macros and before the compiler compiles the DATA step.

Third example: a useful reusable macro

In this example a macro with parameters is used as a building block for a DATA step and results in reusable code.

Your SAS data set contains the birth dates of children born to the respondents in a survey of women and you want to examine child spacing, that is, the lengths in months of the intervals between successive births. Each observation in your data set has the month (1 to 12) and year (2 digits) of birth of up to five children in variables MO1 YR1 MO2 YR2 MO3 YR3 MO4 YR4 MO5 YR5. You want to do what some sociologists like to do, i.e., define new variables DOB1-DOB5 which are the children's birth dates in century months.

To get century months, we let January 1900 be 1, and then apply the rule: century month = 12 x year + month. Easy enough; for the first birth, the assignment statement DOB1=12*YR1 + MO1 defines the century month DOB1. But then there are the 2nd, 3rd, 4th, and 5th births; and what about dates of marriages, separations, and divorces which you may also want to express in century months? Should you repeat the assignment statement as many times as you need to (that's one solution) or. . . . You hear the macro processor:

MP: Use a macro with parameters! Use a macro with parameters!

After consulting the "SAS Guide to Macro Processing" once more, you write the macro CM with three

parameters and a DATA step which invokes it five times, once for each birth date:

```
OPTIONS MPRINT;

%MACRO CM(MONTH,YEAR,CMDATE);
  %*convert MONTH, YEAR to century;
  %*month date CMDATE;
  &CMDATE=12*&YEAR + &MONTH;
%MEND CM;
```

```
DATA CMBIRTHS;
  SET IN.BIRTHS;
```

```
  %CM(MO1,YR1,DOB1)
  %CM(MO2,YR2,DOB2)
  %CM(MO3,YR3,DOB3)
  %CM(MO4,YR4,DOB4)
  %CM(MO5,YR5,DOB5)
```

```
RUN;
```

The MPRINT system option shows you the work of the macro processor. With the first call to the macro CM, the variable name MO1 becomes the value of the parameter MONTH, the variable name YR1 becomes the value of the parameter YEAR, and the new variable name DOB1 becomes the value of the parameter CMDATE. Analogous substitutions of variable names for parameters occur with each succeeding invocation of the macro CM. In the SAS log you see the code generated by the macro processor:

```
DATA CMBIRTHS;
  SET IN.BIRTHS;
```

```
  %CM(MO1,YR1,DOB1)
MPRINT(CM): DOB1=12*YR1 + MO1;
  %CM(MO2,YR2,DOB2)
MPRINT(CM): DOB2=12*YR2 + MO2;
  %CM(MO3,YR3,DOB3)
MPRINT(CM): DOB3=12*YR3 + MO3;
  %CM(MO4,YR4,DOB4)
MPRINT(CM): DOB4=12*YR4 + MO4;
  %CM(MO5,YR5,DOB5)
MPRINT(CM): DOB5=12*YR5 + MO5;
```

```
RUN;
```

With new confidence from your successes in communicating with the macro processor, you remember that you can take advantage of all kinds of patterns in

code and request the macro processor to generate the code. You examine the pattern in the five statements which call the macro CM in the program above and write a second macro BIRTHDTS to generate these calls. The following program, now consisting of two macros followed by a much shorter DATA step, achieves the same result as the previous version:

```
OPTIONS MPRINT;
```

```
%MACRO CM(MONTH,YEAR,CMDATE);
  %*convert MONTH, YEAR to century;
  %*month date CMDATE;
  &CMDATE=12*&YEAR + &MONTH;
%MEND CM;
```

```
%MACRO BIRTHDTS;
  %*generate calls to macro CM to get;
  %*century months for birth dates;
  %DO I=1 %TO 5;
    %CM(MO&I,YR&I,DOB&I)
  %END;
%MEND BIRTHDTS;
```

```
DATA CMBIRTHS;
  SET IN.BIRTHS;
```

```
  %BIRTHDTS
```

```
RUN;
```

Now you remember that you also want to convert to century months the respondents' dates of marriages, separations, and divorces which are stored in a separate SAS data set. You again hear the macro processor:

MP: Store your macro CM outside of your program so that other programs can use it!

By lifting your macro CM out of your program and storing it in a separate file, you can make it available to any program. For example, if you store your macro CM as the member CM in a partitioned data set under MVS named USTONE.SESUG93.MACLIB, it can be invoked from any program by using the FILENAME and %INCLUDE statements like this:

```
FILENAME IN 'USTONE.SESUG93.MACLIB';
```

```
%INCLUDE IN(CM);
```

```
DATA CMBIRTHS;  
  SET IN.BIRTHS;
```

```
  %CM(MO1,YR1,DOB1)
```

```
RUN;
```

The macros CM and BIRTHDTS above are examples of building blocks that can be used in a program to reduce the amount of keyed text when you need to do the same processing more than once. The macro CM also illustrates generality through the use of parameters, making it useful reusable code. With appropriate macros, the part of the program in which the macros are invoked can be written with greater simplicity.

SUMMARY

The examples presented in this tutorial were selected to illustrate how the SAS macro facility can help you write better programs. In particular, the macro processor's capability as a text generator can help you write programs that are more modular in structure, shorter, and more general in function.

The examples were also selected to have some important features in common. First, there was always a choice--to engage the SAS macro processor or not. We opted for the macro processor in each case because of the benefits to our program of doing so. Second, each situation required us to look ahead. We first had to picture the code for the entire program and then we requested the macro processor to generate a significant portion of that code.

Finally, the examples represent appropriate uses of the macro facility in programs regardless of whether they are written to run once or more than once. I believe, contrary to some opinion, that use of the macro facility is justified even in programs that run only once. The decision to use it should be based on much more significant issues, such as those discussed in this tutorial.

The SAS macro facility provides us with a way--in fact, the best way--to simulate the roles of programmer-written functions and subroutines in other programming

languages. It gives us capabilities beyond using straight DATA steps and PROC steps to break up a complex task into subtasks. The macro facility increases our potential to write code in a modular way, resulting in programs that are systematically testable, more easily debugged and modified, shorter, and more clearly laid out in terms of logic. Since these attributes are determinants of the programming time required to get programs to run correctly, they are important goals whether our work is in a production or a research environment. The macro facility can help us achieve these goals.

ACKNOWLEDGMENTS

I appreciated the support of the Carolina Population Center in the preparation of this paper. Special thanks to Karin Gleiter, Roxane Perles and other CPC friends, and to my husband Alan, who either read an earlier version of the paper or listened to a presentation of it. I also want to thank Sara Curran and other CPC graduate students who, in asking questions and even providing one of the examples, unwittingly encouraged me to attempt to write a tutorial on this complex topic.

REFERENCES

Kretzman, Peter. 1988. "Teflon and Tang: Forging Generic Macro Tools." In *Proceedings of the Thirteenth Annual SAS Users Group International Conference Orlando, Florida March 27-30, 1988*, 1300-1304. Cary, NC: SAS Institute Inc.

Kretzman, Peter. 1992. "Ifs, Ands, and Buts: A Case Study in Advanced Macro Implementation." In *Proceedings of the Seventeenth Annual SAS Users Group International Conference Honolulu, Hawaii April 12-15, 1992*, 176-182. Cary, NC: SAS Institute Inc.

O'Connor, Susan M. 1991. "A Roadmap to Macro Facility Error Messages and Debugging." In *Proceedings of the Sixteenth Annual SAS Users Group International Conference New Orleans, Louisiana February 17-20, 1991*, 215-222. Cary, NC: SAS Institute Inc.

SAS Institute Inc. 1990. *SAS Guide to Macro Processing*, Version 6, Second Edition. Cary, NC: SAS Institute Inc. 319 pp.

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