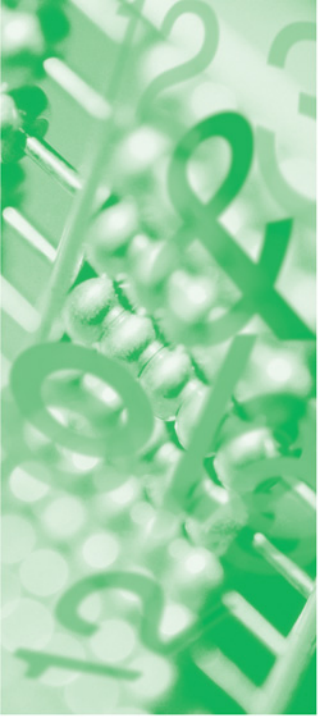


SAS® Macro Programming *Made Easy*

Second Edition

Michele M. Burlew

SAS Press Series

sas

Praise from the Experts

"Using SAS macros can provide functionality as well as flexibility within code. Reading Michele Burlew's book, *SAS Macro Programming Made Easy*, *Second Edition*, helps eliminate the 'fear factor' often associated with using macros, while offering valuable insight to programmers with a broad range of experience.

"This book appeals to the less experienced SAS programmer by explaining how a macro works in an easy-to-understand way and offers insight on various programming techniques. Michele compares the use of the macro procedure to using an office assistant to perform repetitive tasks in a way that most can relate to. Experienced programmers will also find this second edition of *SAS Macro Programming Made Easy* a useful tool in better understanding the mechanics associated with macros. This book was easy to follow and provides an excellent reference for macro programmers."

Suson vonLehmden
Supervisor, System Analysis & Programming
Research Computing Division
RTI International

"This second edition updates the classic macro book with SAS®9 features and new sections, making this excellent reference to the SAS macro language even better. Michele's friendly style is especially good for programmers who might be fearful of macro programming (like me!).

"This book is filled with examples showing how to store and reuse macro programs, build a library of routines, debug macro programs, and a stepwise method for writing macro code.

"The discussion of the autocall and compiled macro facilities is very well done. In this section, Michele gives examples of both of these facilities, explaining the advantages and disadvantages of each. As an added value, you may want to include the macros she presents in your own macro library.

"Michele Burlew has added new material and brought her already excellent first edition up to date. This is a book that anyone who uses the macro facility needs to have in their collection."

Dr. Ron Cody
Professor (retired)
Robert Wood Johnson Medical School

"We all want a 'SAS programming assistant' to help us complete our jobs more quickly. In her book *SAS Macro Programming Made Easy, Second Edition*, Michele Burlew encourages us to take advantage of the SAS Macro Facility as our 'SAS programming assistant.' She demonstrates how macros can handle many of the SAS programming tasks that you presently spend a lot of time on.

"The Macro 'newbie' will learn well from the logical progression of topics and the in-depth coverage of concepts. Both beginner and intermediate macro programmers will benefit from the behind-the-scenes explanations of how macro programs process, the debugging tools and tips (because unexpected results do happen), and the stepwise macro development method, which is a wonderful approach to maintain your sanity when writing macros.

"Whether you read this book sequentially or jump right to topics you need to know, you will find this book to be a valuable resource."

Marje Fecht
Senior Partner
Prowerk Consulting

"Whether you are new to macro programming or at an intermediate level, this second edition of a first-time favorite, with its abundance of examples and helpful explanations, will show you how to shorten code, minimize repetitive tasks, and give you the tools to potentially make your programs dynamic in scope."

Robert Francis, Ph.D.
Contractor, NOVA Research Company

SAS® Macro Programming *Made Easy*

Second Edition

Michele M. Burlew

THE POWER TO KNOW®

The correct bibliographic citation for this manual is as follows: Burlew, Michele M. 2006. *SAS® Macro Programming Made Easy, Second Edition.* Cary, NC: SAS Institute Inc.

SAS® Macro Programming Made Easy, Second Edition

Copyright © 2006, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-59047-882-0

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a Web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the U.S. government is subject to the Agreement with SAS Institute and the restrictions set forth in FAR 52.227-19, Commercial Computer Software-Restricted Rights (June 1987).

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

1st printing, December 2006

SAS® Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/pubs** or call 1-800-727-3228.

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

Other brand and product names are registered trademarks or trademarks of their respective companies.

Contents

Preface ix
Acknowledgments xi

Part 1 Understanding the Concepts and Features of Macro Programming 1

Chapter 1 Introduction 3

What Is the SAS Macro Facility? 4
What Are the Advantages of the SAS Macro Facility? 6
Where Can the SAS Macro Facility Be Used? 12
Examples of the SAS Macro Facility 13

Chapter 2 Mechanics of Macro Processing 23

Introduction 23
The Vocabulary of SAS Processing 24
SAS Processing without Macro Activity 25
Understanding Tokens 26
Tokenizing a SAS Program 28
Comparing Macro Language Processing and SAS Language Processing 29
Processing a SAS Program That Contains Macro Language 30

Chapter 3 Macro Variables 39

Introduction 40
Basic Concepts of Macro Variables 40
Referencing Macro Variables 42
Understanding Macro Variable Resolution and the Use of Single and Double Quotation Marks 44
Displaying Macro Variable Values 46
Understanding Automatic Macro Variables 52

Understanding User-Defined Macro Variables 56
Combining Macro Variables with Text 59
Referencing Macro Variables Indirectly 65

Chapter 4 Macro Programs 73

Introduction 74
Creating Macro Programs 74
Executing a Macro Program 78
Displaying Notes about Macro Program Compilation in the SAS Log 80
Displaying Messages about Macro Program Processing in the SAS Log 82
Passing Values to a Macro Program through Macro Parameters 85

Chapter 5 Understanding Macro Symbol Tables and the Processing of Macro Programs 101

Introduction 102
Understanding Macro Symbol Tables 102
Processing of Macro Programs 122

Chapter 6 Macro Language Functions 133

Introduction 133
Macro Character Functions 134
Macro Evaluation Functions 138
Macro Quoting Functions 140
Macro Variable Attribute Functions 143
Other Macro Functions 147
SAS Supplied Autocall Macro Programs Used Like Functions 154

Chapter 7 Macro Expressions and Macro Programming Statements 159

Introduction 160
Macro Language Statements 160
Constructing Macro Expressions 163

Conditional Processing with the Macro Language 167
Iterative Processing with the Macro Language 177
Branching in Macro Processing 184

Chapter 8 Masking Special Characters and Mnemonic Operators 189

Introduction 190
Why Are Quoting Functions Called Quoting Functions? 191
Illustrating the Need for Macro Quoting Functions 191
Describing the Commonly Used Macro Quoting Functions 192
Understanding How Macro Quoting Functions Work 194
Applying Macro Quoting Functions 195
Specifying Macro Program Parameters That Contain Special Characters or Mnemonic Operators 203
Unmasking Text and the %UNQUOTE Function 213
Using Quoting Versions of Macro Character Functions and Autocall Macro Programs 214

Chapter 9 Interfaces to the Macro Facility 217

Introduction 218
Understanding DATA Step Interfaces to the Macro Facility 218
Using Macro Facility Features in PROC SQL 251
Using Macro Facility Features in SAS Component Language 262

Part 2 Applying Your Knowledge of Macro Programming 267

Chapter 10 Storing and Reusing Macro Programs 269

Introduction 270
Saving Macro Programs with the Autocall Facility 270
Saving Macro Programs with the Stored Compiled Macro Facility 278
Resolving Macro Program References When Using the Autocall Facility and the Stored Compiled Macro Facility 283

Chapter 11 Building a Library of Utilities 285

Introduction 285
Writing a Macro Program to Behave Like a Function 286
Programming Routine Tasks 290

Chapter 12 Debugging Macro Programming and Adding Error Checking to Macro Programs 297

Introduction 298
Understanding the Types of Errors That Can Occur in Macro Programming 298
Minimizing Errors in Developing SAS Programs That Contain Macro Language 299
Categorizing and Checking for Common Problems in Macro Programming 299
Understanding the Tools That Can Debug Macro Programming 303
Examples of Solving Errors in Macro Programming 307
Improving Your Macro Programming by Including Error Checking 326

Chapter 13 A Stepwise Method for Writing Macro Programs 335

Introduction 336
Building a Macro Program in Four Steps 336
Applying the Four Steps to an Example 337

Part 3 Appendixes 369

Appendix A Abridged Macro Language Reference 371

Selected SAS Options Used with the Macro Facility 372
Automatic Macro Variables 373
Macro Functions 377
Macro Language Statements 381
PROC SQL Interface to the Macro Facility 386
SAS Functions and Routines That Interface with the Macro Facility 387

Appendix B Reserved Words in the Macro Facility 391

Appendix C Sample Data Set 393

Appendix D Reference to Programs in This Book 399

Index 407

Preface

How Can This Book Help You Understand the SAS Macro Facility?

This book is for beginning through experienced users of SAS who want to learn about SAS macro programming. It assumes that you have beginning to intermediate experience writing SAS language programs, and it does not review SAS language and SAS programming concepts.

The focus of this book is to make the macro facility a tool you can use in your programming. It is less inclusive and spends less time on reference details than *SAS Macro Language: Reference*.

The technical aspects of macro processing are described in this book. While understanding the technical aspects is not necessary to begin to reap the benefits of the SAS macro facility, this knowledge might help you more wisely apply macro programming techniques.

Don't worry if the technical aspects are difficult to grasp at first. Instead, jump in and start using the simpler features of the macro facility. Try macro variables first. You're bound to make some errors, but those errors help you understand macro processing. Eventually, as your macro programming skills improve, a more thorough understanding of macro processing can reduce the number of macro programming errors you make and make it easier to debug your programs.

This book is grouped into three parts. The first part explains the elements and mechanics of the macro programming language. The second part shows ways of applying your knowledge of macro programming that you gained in the first part. The third part contains four appendixes that provide a quick reference to the macro language, the data set used for most of the examples, and a short description of the example programs.

This book starts with the easier features of the SAS macro facility. These features are building blocks for the later topics. The features of the macro facility are interrelated, and so occasionally you might see some features used before they are formally discussed.

Because macro facility features are interrelated, this book does not have to be read in a linear fashion. Work through sections as appropriate for your needs. Return to earlier sections when that information becomes pertinent. However, it is best to start with the technical information in Chapter 2 and move on to the macro variable chapter, Chapter 3. You might then work with macro variables extensively and try some of the features like macro functions and macro expressions that are described in Chapters 6 and 7. After

gaining an understanding of how macro variables work, you might try writing macro programs. You can learn how to do this in Chapter 4 and then try using the macro programming statements in Chapter 7. Read Chapter 13 to see how a macro program can be designed and constructed.

You might find it useful to learn about macro facility interfaces before you cover macro programs. Chapter 9 includes information useful for DATA step programmers, PROC SQL programmers, and SCL programmers.

About the Data and Programs in This Book

The examples in this book are illustrated with sales data from a fictitious bookstore. The DATA step to create this data set is in Appendix C. A PROC CONTENTS listing for this data set is also in Appendix C.

The programs in this book are written to expect the sales data set to be a permanent data set with a libref of BOOKS and a data set name of YTDSALES.

The examples and screens in this book were produced using SAS®9 under Windows XP.

The programs in each chapter are numbered. Appendix D lists all the numbered programs in this book along with a brief description.

The Typographical Styles in This Book

The typographical styles in this book follow that of SAS documention.

- Values in *italics* identify arguments and values that you supply.
- Arguments enclosed in angle brackets (< >) are optional.
- Arguments separated with a vertical bar (|) indicate mutually exclusive choices.

For example, the syntax of the %SYSEVALF function is written as follows:

```
%SYSEVALF(arithmetic expression|logical expression
    <,conversion-type>)
```

When specifying the %SYSEVALF function, you must specify either an arithmetic expression or a logical expression. Specifying a conversion type is optional.

Acknowledgments

Many thanks to all involved in producing the second edition of this book.

Thanks to my editors at SAS, John West and Julie Platt, for their guidance in completing this project. I appreciate the opportunity to write a second edition on a subject that I really like.

Thanks to the technical reviewers, Kay Alden, Patrick Garrett, Amy Gumm, Cynthia Johnson, Russ Tyndall, and Ed Vlazny, for their careful review of the material and useful suggestions. Thanks especially to Russ for his expertise and generous assistance in answering my multiple questions.

Thanks to the SAS Publishing copyedit and production team for their hard work in the layout, figure redesign, copyediting, and marketing of this book. The team members contributing to this book include Mike Boyd for copyediting, Patrice Cherry for cover design, Jennifer Dilley for graphic design and redesign of the figures for the second edition, Candy Farrell for layout, Shelly Goodin for preproduction marketing, Mary Beth Steinbach as SAS Press Managing Editor, and Liz Vallani for postproduction marketing.

As in the first edition, I want to acknowledge my friends and former coworkers from St. Paul Computer Center at the University of Minnesota, especially Jim Colten, Janice Jannett, Mel Sauve, Dave Schempp, Karen Schempp, and Terri Schultz. Not only were they a great group to work with, but the programming skills I learned from them have helped me over and over again.

Thanks to Mike Davern, Lynn Blewett, Pamela Johnson, and Michelle Casey with the Division of Health Policy and Management at the University of Minnesota, and Brian Gray with the U.S. Geological Survey Upper Midwest Environmental Sciences Center for the very interesting work that keeps me writing SAS macros on a nearly daily basis.

Part 1

Understanding the Concepts and Features of Macro Programming

Chapter 1 **Introduction 3**

Chapter 2 **Mechanics of Macro Processing 23**

Chapter 3 **Macro Variables 39**

Chapter 4 **Macro Programs 73**

Chapter 5 **Understanding Macro Symbol Tables and the Processing of Macro Programs 101**

Chapter 6 **Macro Language Functions 133**

Chapter 7 **Macro Expressions and Macro Programming Statements 159**

Chapter 8 **Masking Special Characters and Mnemonic Operators 189**

Chapter 9 **Interfaces to the Macro Facility 217**

Chapter 1

Introduction

What Is the SAS Macro Facility? 4
What Are the Advantages of the SAS Macro Facility? 6
Where Can the SAS Macro Facility Be Used? 12
Examples of the SAS Macro Facility 13

Imagine you have an assistant to help you write your SAS programs. Your assistant willingly and unfailingly follows your instructions allowing you to move on to other tasks. Repetitive programming assignments like multiple PROC TABULATE tables, where the only difference between one table and the next is the classification variable, are delegated to your assistant. Jobs that require you to run a few steps, review the output, and then run additional steps based on the output are not difficult; they are, however, time-consuming. With instructions on selection of subsequent steps, your assistant easily handles the work. Even having your assistant do simple tasks like editing information in TITLE statements makes your job easier.

Actually, you already have a SAS programming assistant: the SAS macro facility. The SAS macro facility can do all the tasks above and more. To have the macro facility work for you, you first need to know how to communicate with the macro facility. That's the purpose of this book: to show you how to communicate with the SAS macro facility so that your SAS programming can become more effective and efficient.

An infinite variety of applications of the SAS macro facility exist. An understanding of the SAS macro facility gives you confidence to appropriately use it to help you build your SAS programs. The more you use the macro facility, the more adept you become at using it. As your skills increase, you discover more situations where the macro facility can be applied. The macro programming skills you learn from this book can be applied throughout SAS.

You do not have to use any of the macro facility features to write good SAS programs, but, if you do, you might find it easier to complete your SAS programming assignments. The SAS programming language can get you from one floor to the next, one step after another. Using the macro facility wisely is like taking an elevator to get to a higher floor: you follow the same path, but you'll likely arrive at your destination sooner.

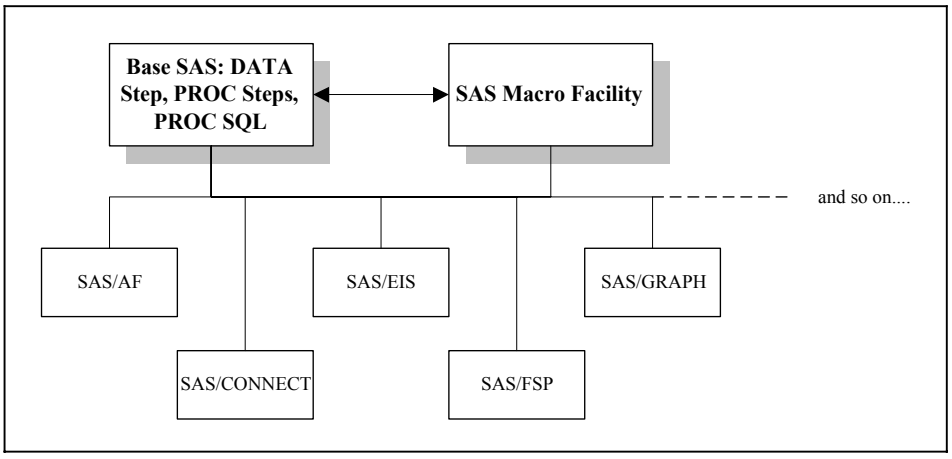
What Is the SAS Macro Facility?

Fundamentally, the SAS macro facility is a tool for text substitution. You associate a macro reference with text. When the macro processor encounters that reference, it replaces the reference with the associated text. This text can be as simple as text strings or as complex as SAS language statements. The macro processor becomes your SAS programming assistant in helping you construct your SAS programs.

The SAS macro facility is a component of Base SAS. The Base SAS product is integral to SAS and must be installed at your computing location if you want to write SAS programs or run SAS procedures in any of the SAS products. Therefore, if you have access to SAS, you have access to the macro facility, and you can include macro facility features in your programs. Indeed, many of the SAS products that you license contain programs that use the macro facility.

As shown in Figure 1.1, the SAS macro facility works side-by-side with Base SAS to build and execute your programs. The macro facility has its own language distinct from the SAS language, but the language and conventions of the macro facility are similar to the style and syntax of the SAS language. If you already write DATA steps, you have a head start on understanding the language and conventions of the macro facility.

Figure 1.1 How the SAS macro facility fits into SAS



The two main components of the SAS macro facility are ***SAS macro variables*** and ***SAS macro programs***. With SAS macro variables, you create references to larger pieces of text. A typical use of a macro variable is to repeatedly insert a piece of text throughout a SAS program. SAS macro programs use macro variables and macro programming statements to build SAS programs. Macro programs can direct conditional execution of DATA steps and PROC steps. Macro programs can do repetitive tasks such as creating or analyzing a series of data sets.

Example 1.1 shows how a macro variable can be used, and Example 1.2 shows how a macro program can be used.

Example 1.1: Using a Macro Variable to Select Observations to Process

The macro variable MONTH_SOLD defined in Program 1.1 is used to select a subset of a data set and place information in the report title. Macro language and macro variable references are in bold.

Program 1.1

```
%let month_sold=4;
proc print data=books.ytdsales
                    (where=(month(datesold)=&month_sold));
  title "Books Sold for Month &month_sold";
  var booktitle saleprice;
  sum saleprice;
run;
```

Example 1.2: Using a Macro Program to Execute the Same PROC Step on Multiple Data Sets

When Program 1.2 executes, it submits a PROC MEANS step three times: once for each of the years 2007, 2008, and 2009. Each time, it processes a different data set. The macro language and references that generate the three steps are in bold.

Program 1.2

```
%macro sales;
  %do year=2007 %to 2009;
    proc means data=books.sold&year;
      title "Sales Information for &year";
      class section;
      var listprice saleprice;
    run;
  %end;
%mend sales;
%sales
```

The macro facility was first released in SAS 82.0 in 1982. There are relatively few statements in the macro language, and these statements are very powerful.

In a world of rapidly changing software tools and techniques, the macro facility remains one of the most widely used components of SAS. What you learn now about the macro facility will serve you for many years of SAS programming.

What Are the Advantages of the SAS Macro Facility?

Your SAS programming productivity can improve when you know how and when to use the SAS macro facility. The programs you write can become reusable, shorter, and easier to follow.

In addition, by incorporating macro facility features in your programs you can

- accomplish repetitive tasks quickly and efficiently. A macro program can be reused many times. Parameters passed to the macro program customize the results without having to change the code within the macro program.
- provide a more modular structure to your programs. SAS language that is repetitive can be generated by macro language statements in a macro program,

and that macro program can be referenced in your SAS program. The reference to the macro program is similar to calling a subroutine. The main program becomes easier to read—especially if you give the macro program a meaningful name for the function that it performs.

Think about automated bill paying as a real-world example of the concepts of macro programming. When you enroll in an automated bill paying plan, you no longer initiate payments each month to pay recurring bills like the mortgage and the utilities. Without automated bill paying, it takes a certain amount of time each month for you to initiate payments to pay those recurring bills. The time that it takes to initiate the automated bill paying plan is likely longer in the month that you set it up than if you just submitted a payment for each monthly bill. But, once you have the automated bill paying plan established (and perhaps allowing the bank a little debugging time!), the amount of time you spend each month dealing with those recurring bills is reduced. You instruct your bank how to handle those recurring bills. In turn, they initiate those monthly payments for you.

That's what macro programming can do for you. Instead of editing the program each time parameters change (for example, same analysis program, different data set), you write a SAS program that contains macro language statements. These macro language statements instruct the macro processor how to make those code changes for you. Then, when you run the program again, the only changes you make are to the values that the macro language uses to edit your program–like directing the bank to add the water department to your automatic payment plan.

Example 1.3: Defining and Using Macro Variables

Consider another illustration of macro programming, this time including a sample program. The data set that is analyzed here is used throughout this book. The data represent computer book sales at a fictitious bookstore.

Program 1.3 produces two reports for the computer section of the bookstore. The first is a monthly sales report. The second is a pie chart of sales from the beginning of the year through the month of interest.

If you were not using macro facility features, you would have to change the program every time you wanted the report for a different month and/or year. These changes would have to be made at every location where the month value and/or year value were referenced.

Rather than doing these multiple edits, you can create macro variables at the beginning of the program that are set to the month and the year of interest, and place references to these macro variables throughout the program where they are needed. When you get ready to submit the program, the only changes you make are to the values of the macro variables. After you submit the program, the macro processor looks up the values of

month and year that you set and substitutes those values as specified by your macro variable references.

You don't edit the DATA step and the PROC steps; you only change the values of the macro variables at the beginning of the program. The report layout stays the same, but the results are based on a different subset of the data set.

Don't worry about understanding the macro language coding at this point. Just be aware that you can reuse the same program to analyze a different subset of the data set by changing the values of the macro variables.

Note that macro language statements start with a percent sign (%) and macro variable references start with an ampersand (&). Both features are in bold in the following code.

Program 1.3

```
%let repmonth=4;
%let repyear=2007;
%let repmword=%sysfunc(mdy(&repmonth,1,&repyear),monname9.);

data temp;
  set books.ytdsales;
  mosale=month(datesold);
  label mosale='Month of Sale';
run;

proc tabulate data=temp;
  title "Sales During &repmword &repyear";
  where mosale=&repmonth and year(datesold)=&repyear;
  class section;
  var saleprice listprice cost;
  tables section all='**TOTAL**',
         (saleprice listprice cost)*(n*f=4. sum*f=dollar10.2);
run;

proc gchart data=temp
           (where=(mosale <= &repmonth and
                   year(datesold)=&repyear));
  title "Sales Through &repmword &repyear";
  pie section / coutline=black percent=outside
                sumvar=saleprice noheading ;
run;
quit;
```

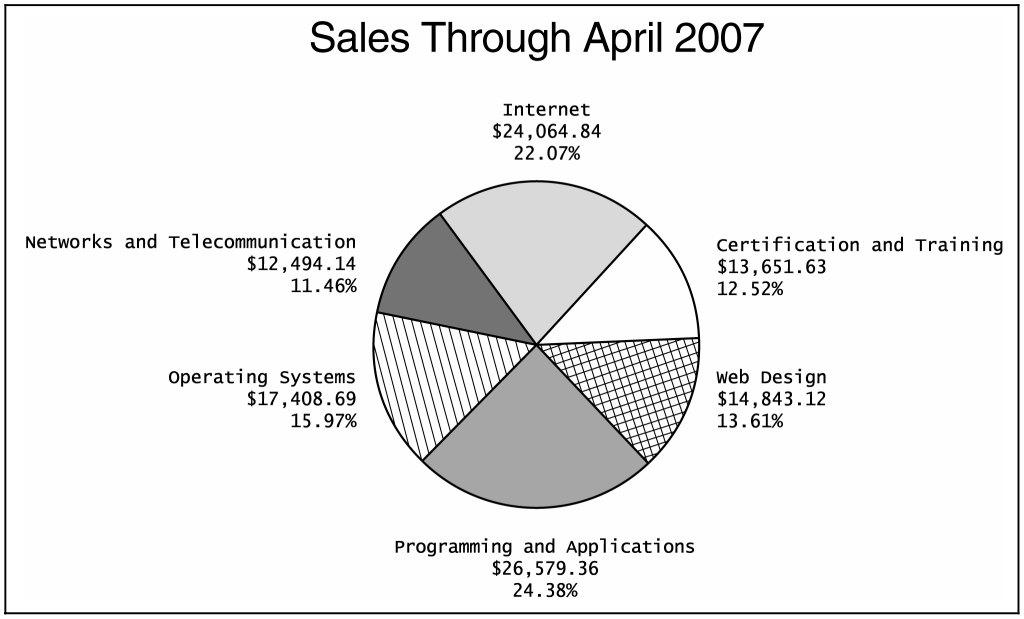
Output 1.3a presents the output from Program 1.3.

Output 1.3a Output from Program 1.3

Sales During April 2007

	Sale Price		List Price		Wholesale Cost	
	N	Sum	N	Sum	N	Sum
Section						
Certification and Training	62	$2,709.54	62	$2,745.90	62	$1,398.42
Internet	89	$3,896.43	89	$3,965.55	89	$2,018.03
Networks and Telecommunication	60	$2,627.57	60	$2,694.00	60	$1,376.47
Operating Systems	79	$3,467.53	79	$3,539.05	79	$1,780.11
Programming and Applications	130	$5,689.23	130	$5,806.50	130	$2,943.80
Web Design	57	$2,500.71	57	$2,559.15	57	$1,293.26
TOTAL	477	$20,891.01	477	$21,310.15	477	$10,810.10

(*continued*)



Changing just the first line of the program from

```
%let repmonth=4;
```

to

```
%let repmonth=5;
```

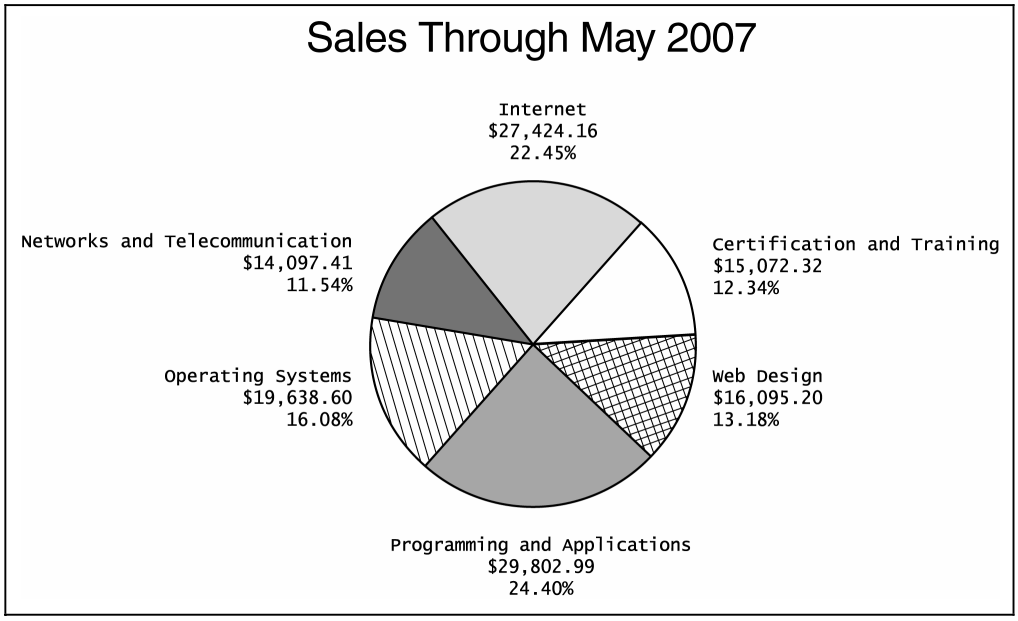
runs the same program, but now processes the data collected for May. No other editing of the program is required to process this subset. Output 1.3b presents the output for May.

Output 1.3b Output from revised Program 1.3

Sales During May 2007

	Sale Price		List Price		Wholesale Cost	
	N	Sum	N	Sum	N	Sum
Section						
Certification and Training	31	$1,420.69	31	$1,449.45	31	$741.81
Internet	79	$3,359.32	79	$3,425.05	79	$1,751.58
Networks and Telecommunication	38	$1,603.27	38	$1,660.10	38	$834.65
Operating Systems	51	$2,229.91	51	$2,284.45	51	$1,154.91
Programming and Applications	72	$3,223.63	72	$3,254.40	72	$1,639.38
Web Design	29	$1,252.09	29	$1,284.55	29	$650.97
TOTAL	300	$13,088.90	300	$13,358.00	300	$6,773.29

(*continued*)



Where Can the SAS Macro Facility Be Used?

The macro facility can be used with all SAS products. You've seen in the monthly sales report an example of macro programming in Base SAS.

Table 1.1 lists some SAS products and possible macro facility applications that you can create. It also lists existing macro applications that come with SAS.

Table 1.1 SAS macro facility applications

SAS Product	Typical Applications of the Macro Facility
Base SAS	Customizes data set processing Customizes PROC steps Customizes reports Passes data between steps in a program Conditionally executes DATA steps and PROC steps Iteratively processes DATA steps and PROC steps Contains libraries of macro program routines
SAS Component Language	Communicates between SAS program steps and SCL programs Communicates between SCL programs
SAS/CONNECT	Passes information between local and remote SAS sessions
SAS/GRAPH	Contains libraries of macro routines for annotating SAS/GRAPH output
SAS/TOOLKIT	Creates functions that can be used with the macro facility

Examples of the SAS Macro Facility

The following examples of the SAS macro facility illustrate some of the tasks that the macro processor can perform for you. There's no need to understand the coding of these programs at this point (although the code is included and might be useful to you later). What you should gain from this section is an idea of the kinds of SAS programming tasks that can be delegated to the macro processor.

In addition to the examples that follow, Program 1.3 demonstrates reuse of the same program by simply changing the values of the macro variables at the beginning of the program. A new subset of data is analyzed each time the values of the macro variables are changed.

It is relatively easy to create and reference these macro variables. Besides being able to reuse your program code, the advantages in using macro variables include reducing coding time and reducing programming errors by not having to edit so many lines of code.

Example 1.4: Displaying System Information

SAS comes with a set of automatic macro variables that you can reference in your SAS programs. Most of these macro variables deal with system-related items like date, time, operating system, and version of SAS. Using these automatically defined macro variables is one of the simplest applications of the macro facility.

Program 1.4 incorporates some of these automatic macro variables, and these macro variables are in bold in the code. Note that the automatic macro variable names are preceded by ampersands. Assume the report was run on February 20, 2008.

Program 1.4

```
title "Sales Report";
title2 "As of &systime &sysday &sysdate";
title3 "Using SAS Version: &sysver";
proc means data=books.ytdsales n sum;
  var saleprice;
run;
```

Output 1.4 presents the output from Program 1.4.

Output 1.4 Output for program using automatically defined SAS macro variables

```
                        Sales Report
                  As of 15:46 Wednesday 20FEB08
                    Using SAS Version: 9.1

                     The MEANS Procedure

            Analysis Variable : saleprice Sale Price

                        N             Sum
                    ---------------------
                    6096       263678.15
                    ---------------------
```

Example 1.5: Conditional Processing of SAS Steps

Macro programs can use macro variables and macro programming statements to select the steps and the SAS language statements to execute in a SAS program. These conditional processing macro language statements are similar in syntax and structure to SAS language statements.

Macro program DAILY in Program 1.5 contains two PROC steps. The first PROC MEANS step runs daily. The second PROC MEANS step runs only on Fridays. The conditional macro language statements direct the macro processor to run the second PROC step only on Fridays. Assume the program was run on Friday, August 17, 2007.

Macro language statements start with percent signs, and macro variable references start with ampersands.

Program 1.5

```
%macro daily;
  proc means data=books.ytdsales(where=(datesold=today()))
                  maxdec=2 sum;
    title "Daily Sales Report for &sysdate";
    class section;
    var saleprice;
  run;
  %if &sysday=Friday %then %do;
     proc means data=books.ytdsales
          (where=(today()-6 le datesold le today()))
          sum maxdec=2;
       title "Weekly Sales Report Week Ending &sysdate";
       class section;
       var saleprice;
     run;
  %end;
%mend daily;

%daily
```

Output 1.5 presents the output from Program 1.5.

Output 1.5 Output from Program 1.5 that uses conditional macro language statements

```
                Daily Sales Report for 17AUG07                    1

                     The MEANS Procedure

          Analysis Variable : saleprice Sale Price
                                        N
    Section                            Obs             Sum
    -----------------------------------------------------
    Certification and Training           5          229.16
    Internet                             7          312.96
    Networks and Telecommunication       3          122.76
    Operating Systems                    3          132.85
    Programming and Applications         2          100.41
    Web Design                           4          173.80
    -----------------------------------------------------

          Weekly Sales Report Week Ending 17AUG07                 2

                     The MEANS Procedure

          Analysis Variable : saleprice Sale Price

                                        N
    Section                            Obs             Sum
    -----------------------------------------------------
    Certification and Training          13          562.78
    Internet                            28         1210.22
    Networks and Telecommunication      15          645.16
    Operating Systems                   27         1206.67
    Programming and Applications        23         1007.57
    Web Design                          14          610.61
    -----------------------------------------------------
```

Example 1.6: Iterative Processing of SAS Steps

Coding each iteration of a programming process that contains multiple iterations is a lengthy task. The %DO loops in the macro language can take over some of that iterative coding for you. A macro program can build the code for each iteration of a repetitive programming process based on the specifications of the %DO loop.

Program 1.6 illustrates iterative processing. It creates 12 data sets, one for each month of the year. Without macro programming, you would have to enter the 12 data set names in the DATA statement and enter all the ELSE statements that direct observations to the right data set. A macro language %DO loop can build those statements for you.

Program 1.6

```
%macro makesets;
  data
    %do i=1 %to 12;
      month&i
    %end;
    ;
    set books.ytdsales;
    mosale=month(datesold);
    if mosale=1 then output month1;
    %do i=2 %to 12;
      else if mosale=&i then output month&i;
    %end;
  run;
%mend makesets;

%makesets
```

After interpretation by the macro processor, the program becomes:

```
 data month1 month2 month3 month4 month5 month6
      month7 month8 month9 month10 month11 month12
      ;
   set books.ytdsales;
   mosale=month(datesold);
   if mosale=1 then output month1;
   else if mosale=2 then output month2;
   else if mosale=3 then output month3;
   else if mosale=4 then output month4;
   else if mosale=5 then output month5;
   else if mosale=6 then output month6;
   else if mosale=7 then output month7;
   else if mosale=8 then output month8;
   else if mosale=9 then output month9;
```

```
    else if mosale=10 then output month10;
    else if mosale=11 then output month11;
    else if mosale=12 then output month12;
  run;
```

Macro language statements built the SAS language DATA statement and all of the ELSE statements in the DATA step for you.

A few macro programming statements direct the macro processor to build the complete DATA step for you. By doing this, you avoid the tedious task of entering all the data set names and all the ELSE statements. Repetitive coding tasks are a breeding ground for bugs in your programs. Thus, turning these tasks over to the macro processor can reduce the number of errors in your SAS programs.

Example 1.7: Passing Information between Program Steps

The macro facility can act as a bridge between steps in your SAS programs. The SAS language functions that interact with the macro facility can transfer information between steps in your SAS programs.

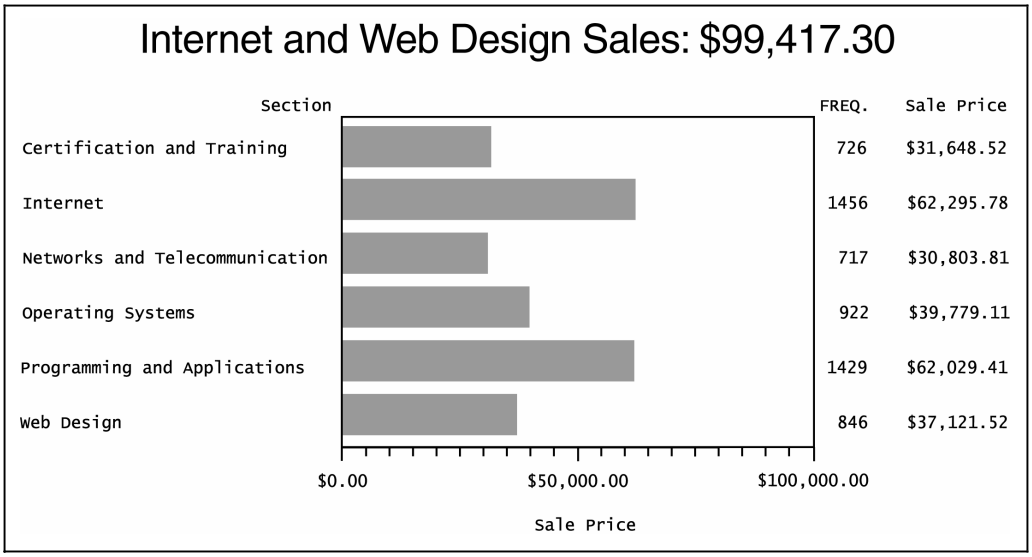
Program 1.7 calculates total sales for two sections in the computer department of the bookstore. That value is then inserted in the TITLE statement of the PROC GCHART output. The SYMPUTX SAS language routine instructs the macro processor to retain the total sales value in macro variable INTWEBSL after the DATA step finishes. The total sales value is then available to subsequent steps in the program.

Program 1.7

```
data temp;
  set books.ytdsales end=lastobs;
  retain sumintwb 0;
  if section in ('Internet','Web Design') then
    sumintwb=sumintwb + saleprice;
  if lastobs then
    call symputx('intwebsl',put(sumintwb,dollar10.2));
run;
proc gchart data=temp;
  title "Internet and Web Design Sales: &intwebsl";
  hbar section / sumvar=saleprice;
  format saleprice dollar10.2;
run;
quit;
```

Output 1.7 presents the output from Program 1.7.

Output 1.7 Output from Program 1.7 that passes data from a DATA step to a TITLE statement



Without the SYMPUTX routine, you would have to submit two programs. The first SAS program would calculate the total sales for the two sections. After the first program ends, you find the total sales value in the output. Then, before submitting the second program, you would have to edit the second program and update the TITLE statement with the total sales value that you found in the output from the first program.

Example 1.8: Interfacing Macro Language and SAS Language Functions

The SAS language has libraries of functions that also can be used in your macro language programs. Some uses of these functions include incorporating information about a data set in a title, checking the existence of a data set, and finding the number of observations in a data set.

Macro program DSREPORT in Program 1.8 produces a PROC MEANS report for a data set whose name is passed as a parameter to DSREPORT. The number of observations and creation date of the data set are obtained with the ATTRN SAS language function, and these attributes are inserted in the report title. The date value is formatted by the PUTN SAS language function. Macro program DSREPORT opens and closes the data set with the OPEN and CLOSE SAS language functions. The SAS language functions are underlined in Program 1.8.

Program 1.8

```
%macro dsreport(dsname);
  %*----Open data set dsname;
  %let dsid=%sysfunc(open(&dsname));

  %*----How many obs are in the data set?;
  %let nobs=%sysfunc(attrn(&dsid,nobs));

  %*----When was the data set created?;
  %let when = %sysfunc(putn(
                %sysfunc(attrn(&dsid,crdte)),datetime9.));

  %*----Close data set dsname identified by dsid;
  %let rc=%sysfunc(close(&dsid));

  title "Report on Data Set &dsname";
  title2 "Num Obs: &nobs   Date Created: &when";

  proc means data=&dsname sum maxdec=2;
    class section;
    var saleprice;
  run;
%mend dsreport;

%dsreport(books.ytdsales)
```

Output 1.8 presents the output from Program 1.8.

Output 1.8 Output from Program 1.8 that uses SAS language functions

Report on Data Set books.ytdsales
Num Obs: 6096 Date Created: 01JAN2007

The MEANS Procedure

Analysis Variable : saleprice Sale Price

Section	N Obs	Sum
Certification and Training	726	31648.52
Internet	1456	62295.78
Networks and Telecommunication	717	30803.81
Operating Systems	922	39779.11
Programming and Applications	1429	62029.41
Web Design	846	37121.52

Example 1.9: Building and Saving a Library of Utility Routines

In your work, you might frequently need to program the same process in different applications. Rather than rewriting the code every time, you might be able to write the code once and save it in a macro program. Later when you want to execute that code again, you just reference the macro program. With the macro facility, there are ways to save the code in special libraries and to even save the compiled code in permanent locations. For example, perhaps certain reports all require the same SAS options, titles, and footnotes. You can save these standardizations in a macro program and call the macro program rather than write the statements every time.

You can store the macro program in a special location called an autocall library. Then when you want to submit the macro program for compilation and execution during a later SAS session, you just need to tell SAS to look for macro programs in the autocall library, and you do not have to explicitly submit the code in the SAS session.

The macro program STANDARDOPTS in Program 1.9 submits an OPTIONS statement to ensure that three SAS options are in effect: NODATE, NUMBER, and BYLINE. It also specifies TITLE1 and FOOTNOTE1 statements. Assume STANDARDOPTS is stored in a file named STANDARDOPTS.SAS in Windows directory c:\mymacroprograms. (Note that if you were using UNIX, the filename must be in lowercase.)

Program 1.9

```
%macro standardopts;
  options nodate number byline;
  title "Bookstore Report";
  footnote1 "Prepared &sysday &sysdate9 at &systime using SAS
&sysver";
%mend standardopts;
```

In a later SAS session, you do not need to submit the previous code. Instead, you can submit the following OPTIONS statement and call the macro program STANDARDOPTS. The SASAUTOS option specifies that SAS have access to the autocall library shipped with SAS ("sasautos") and to the autocall library in the mymacroprograms folder.

```
options mautosource sasautos=(sasautos,'c:\mymacroprograms');
%standardopts
```

After submitting the macro program STANDARDOPTS, the title text on subsequent reports is

```
Bookstore Report
```

If STANDARDOPTS was submitted on February 22, 2008, from a SAS session that started at 8:36 using SAS 9.1, the footnote text on subsequent reports would be:

```
Prepared Friday 22FEB2008 08:36 using SAS 9.1
```

Chapter 2

Mechanics of Macro Processing

Introduction 23

The Vocabulary of SAS Processing 24

SAS Processing without Macro Activity 25

Understanding Tokens 26

Tokenizing a SAS Program 28

Comparing Macro Language Processing and SAS Language Processing 29

Processing a SAS Program That Contains Macro Language 30

Introduction

Understanding the steps that SAS takes to process a program will help you determine where macro facility features can be incorporated in your SAS programs. You do not need a detailed knowledge of the mechanics of macro processing to write SAS programs that include macro features. However, an understanding of the timing of macro language

processing, as it relates to SAS language processing, can help you write more powerful programs and make it easier for you to debug programs that contain macro features.

The examples in Chapter 1 showed how you can enhance your SAS programming with the macro facility. The examples in this chapter illustrate when and how the macro processor does its work.

There are just a few basic concepts in macro processing to add to your knowledge of SAS processing. If you already know how SAS programs are compiled and executed, you are well on your way to understanding the mechanics of macro processing.

As you read through this chapter, keep in mind that the macro processor is your SAS programming assistant, helping you code your SAS programs.

The Vocabulary of SAS Processing

In this chapter, several terms are used to describe SAS processing. Table 2.1 reviews these terms.

Table 2.1 Terms commonly used to describe SAS processing

Term	Description
input stack	Holds a SAS program after it is submitted and before it is processed by the word scanner.
word scanner	Scans the text it takes from the input stack and breaks the text into tokens. Determines the destination of the token: DATA step compiler, macro processor, etc.
token	Fundamental unit in the SAS language. SAS statements must be broken down into tokens, or tokenized, before the statements can be compiled. Tokens are the actual words in the SAS statements as well as the literal strings, numbers, and symbols.
compiler	Checks the syntax of tokens received from the word scanner. After it completes checking the syntax, the compiler translates the tokens into a form for execution.
macro processor	Processes macro language references and statements.
macro trigger	The symbols & and %, when followed by a letter or underscore, that signal the word scanner to transfer what follows to the macro processor.
macro symbol table	The area in memory where macro variables and their associated values are stored.

SAS Processing without Macro Activity

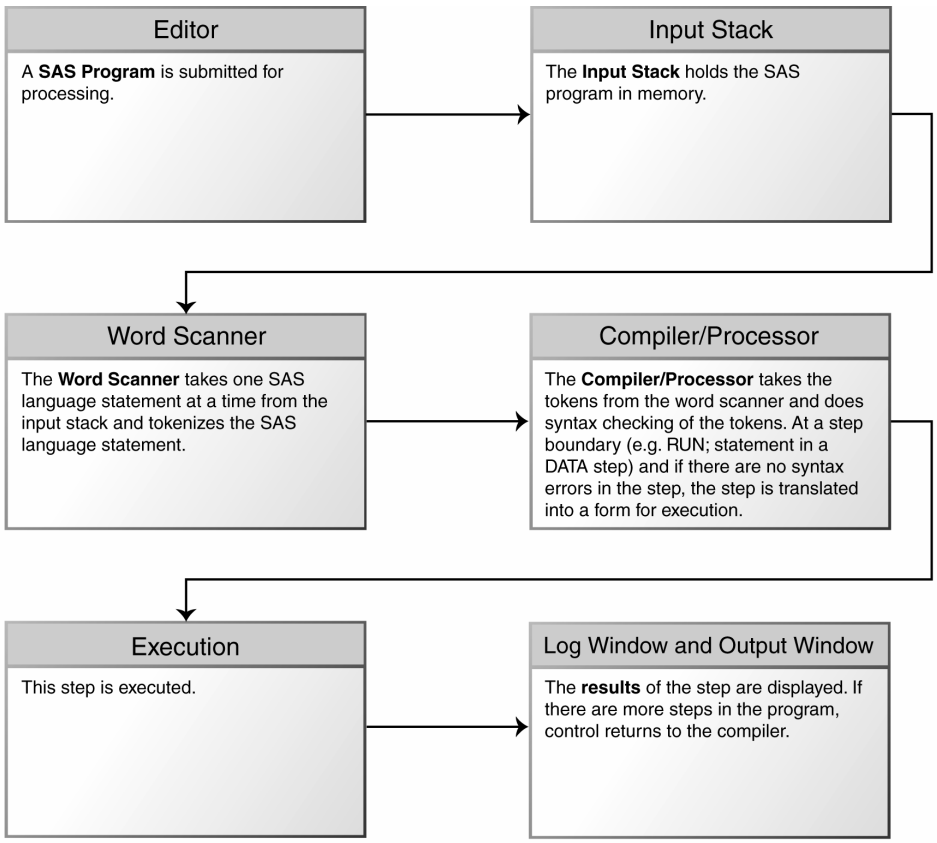
SAS programs can be submitted for processing from several locations including:

- an interactive SAS session from the Editor
- a batch program
- a noninteractive program
- from the command line in the SAS windowing environment
- an SCL SUBMIT block
- the SCL Compile command

In all cases, submitted SAS programs start in the input stack. The word scanner takes statements from the input stack and tokenizes the statements into the fundamental units of words and symbols. The word scanner's job is then to direct the tokens to the right location. The word scanner might direct tokens to the DATA step compiler, the macro processor, the command processor, or the SCL compiler. The compiler or processor that receives the tokens checks for syntax errors. If none are found, the step executes.

Figure 2.1 illustrates the processing of a SAS program that contains no macro facility features.

Figure 2.1 Basic processing of a SAS program



Understanding Tokens

The fundamental building blocks of a SAS program are the tokens that the word scanner creates from your SAS language statements. Each word, literal string, number, and special symbol in the statements in your program is a token.

The word scanner determines that a token ends when either a blank is found following a token or when another token begins. The maximum length of a token under SAS®9 is 32,767 characters.

Two special symbol tokens, when followed by either a letter or underscore, signal the word scanner to turn processing over to the macro processor. These two characters, the ampersand (&) and the percent sign (%), are called macro triggers.

Table 2.2 describes the four types of tokens that SAS recognizes.

Table 2.2 The types of tokens that SAS recognizes

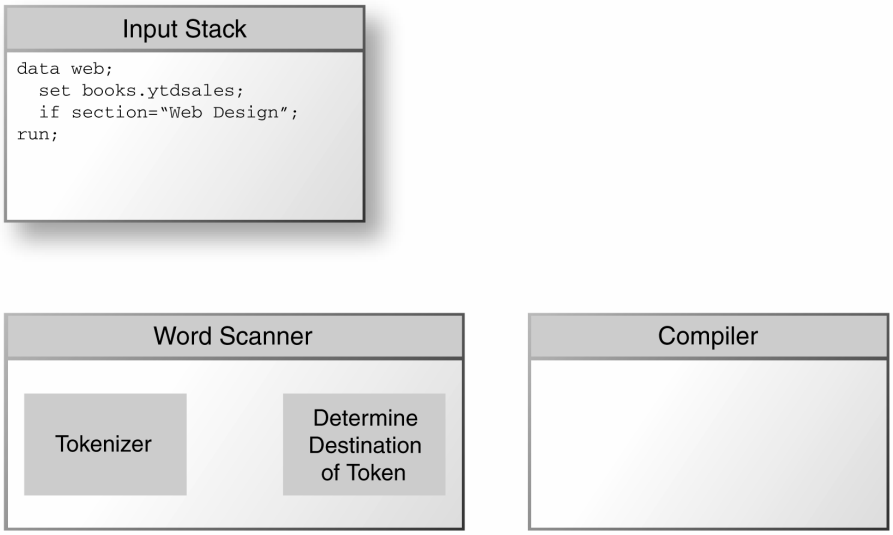
Type of Token	Description	Examples
literal	A string of characters enclosed in single or double quotation marks.	'My program text' "My program text"
numbers	A string of digits including integers, decimal values, and exponential notation. Dates, times, and hexadecimal constants are also number tokens.	123456 '30APR1982'D 98.7654 3. '01'x 6.023E23
names	The "words" in your programs. Name tokens are strings of characters beginning with a letter or underscore and continuing with letters, underscores, or digits. Periods can be part of a name token when referring to a format or informat.	proc _n_ ssn. if mmddyy10. descending var1 var2 var3 var4 var5
special	Characters other than a letter, number, or underscore that have a special meaning to SAS.	; + - * / ** ( ) { } & %

The importance of understanding tokenization is evident if you have ever dealt with unmatched quotation marks in your SAS language statements. Matched quotation marks delimit a literal token. When you omit a closing quotation mark, SAS continues to add text to your literal token beyond what you intended. The SAS programming statements added to the literal token never get tokenized by the word scanner. Eventually, the literal token terminates when either another quotation mark is encountered or the literal token reaches its maximum length (32K characters under SAS®9). At that point, your program cannot compile correctly and processing stops.

Tokenizing a SAS Program

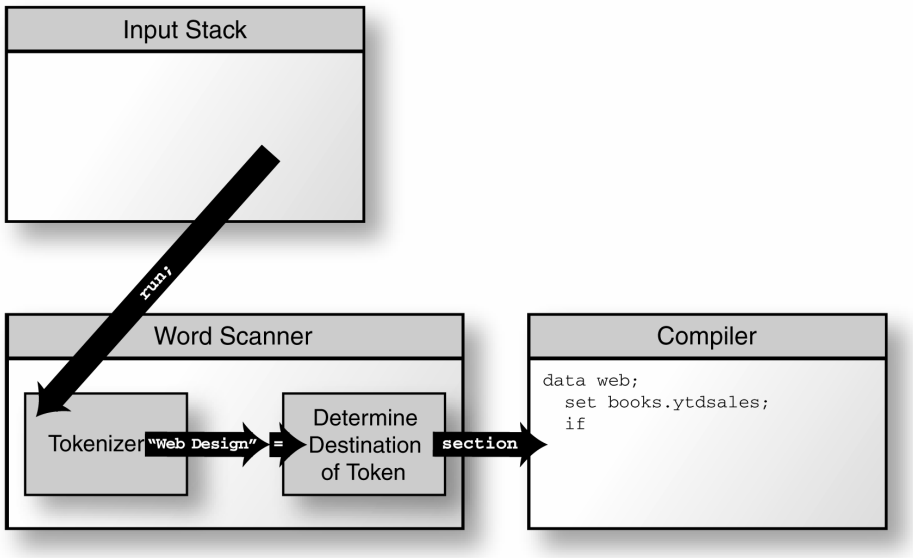
The next two figures illustrate the tokenization of a DATA step. In Figure 2.2, the program has been submitted and is waiting in the input stack for tokenization by the word scanner.

Figure 2.2 A SAS program has been submitted and waits in the input stack for tokenization



In Figure 2.3, the word scanner tokenizes the program and determines the destination of the tokens. In this example, the word scanner sends the tokens to the DATA step compiler.

Figure 2.3 The word scanner tokenizes the SAS language statements in the program



When the compiler receives the semicolon following the RUN statement, it stops taking tokens from the word scanner. The compiler looks for syntax errors. If it finds no errors, it compiles and executes the step.

Comparing Macro Language Processing and SAS Language Processing

It is important to realize that there are differences between the SAS language and the SAS macro language. You probably are familiar with terms like variables and statements in the SAS language. The macro language also has variables and statements, but these variables and statements are different from those in the SAS language and serve different purposes.

The main function of the macro facility is to help you build *SAS language statements* that can be tokenized, compiled, and executed. By taking over some of your coding tasks, the macro processor decreases the amount of coding you do.

Recall that all SAS programs are compiled and executed the same way. After a SAS program is submitted, the statements wait in the input stack for processing. The word scanner then takes each SAS language statement from the input stack and tokenizes it. The compiler requests the tokens, does syntax checking, and (at a step boundary) passes the compiled statements on for execution.

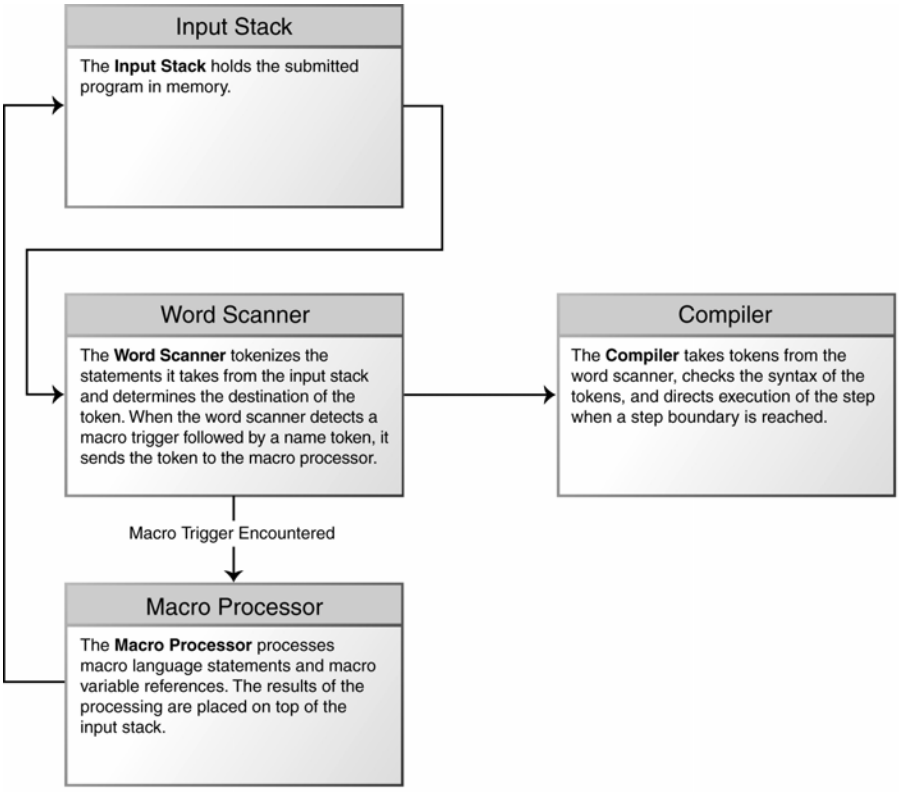
When the word scanner detects a macro trigger followed by a name token, it sends what follows to the macro processor and temporarily turns processing over to the macro processor. The word scanner suspends tokenization while the macro processor completes its job. Therefore, processing of a macro language reference occurs *after* tokenization and *before* compilation.

As your SAS programming assistant, the macro processor codes SAS language statements for you based on the guidelines you give it. The way you communicate your requests to the macro processor is through the macro language. The macro processor takes the macro language statements you write and turns them into SAS language statements. The macro processor puts the SAS language statements that it builds back on top of the input stack. The word scanner then resumes its work by tokenizing the newly built SAS language statements that have come from the macro processor.

Processing a SAS Program That Contains Macro Language

This section describes how SAS processes a program that includes macro language statements. The macro processor starts working when the word scanner encounters a macro trigger followed by a letter or underscore. The word scanner then directs the results of its tokenization to the macro processor. The word scanner sends tokens to the macro processor until the macro reference is terminated. The macro processor resolves macro language references and returns the results to the top of the input stack. The word scanner then resumes tokenization. Figure 2.4 illustrates this process.

Figure 2.4 SAS processing when macro facility features are included in the program



The next several figures illustrate the process described in Figure 2.4 with the program from Figure 2.2. This program now contains one macro language statement and one macro variable reference.

The %LET macro language statement assigns a value to a macro variable. The %LET statement tells the macro processor to store the macro variable name and its associated text in the macro symbol table.

The macro variable is placed in the DATA step where the text associated with the macro variable should be substituted. The ampersand token followed by a name token is the instruction to the macro processor to look in the macro symbol table for the text associated with the macro variable whose name follows the ampersand. At the location of the macro variable reference in the DATA step, the macro processor replaces the reference with the macro variable's value.

The value of the macro variable REPGRP in the program in the next several figures is used to define a subset of the data set. In this example, only observations from the section "Web Design" are written to the output data set.

The value of the macro variable is stored in the macro symbol table for the duration of the SAS session. When a SAS session starts, SAS automatically defines several macro variables. These automatic macro variables are also stored in the macro symbol table. A few of the automatic macro variables are included in the figures.

In Figure 2.5, the program has been submitted.

Figure 2.5 The program with macro facility features has been submitted and the word scanner is ready to tokenize

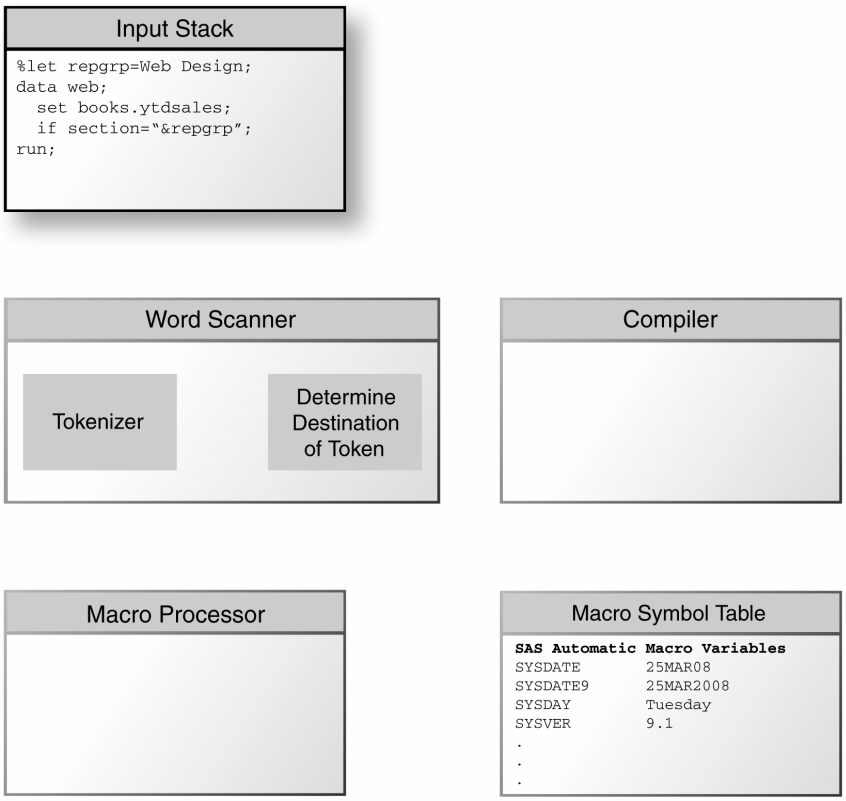
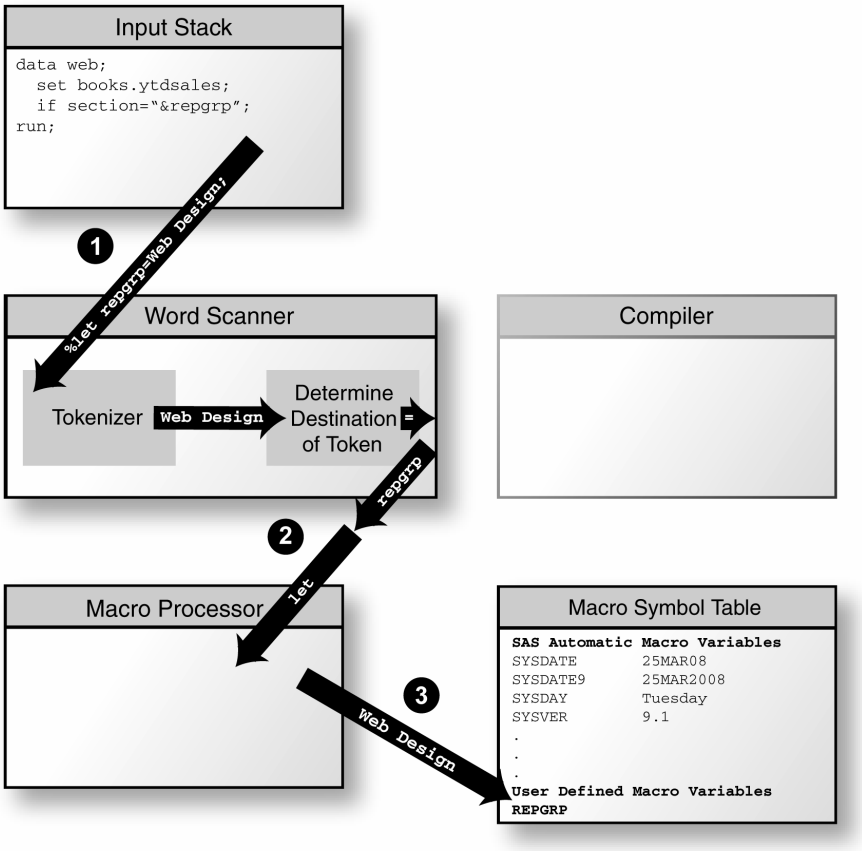


Figure 2.6 shows how a macro language statement is taken from the input stack, tokenized by the word scanner, and passed to the macro processor.

Figure 2.6 A macro language statement is processed



The three steps in Figure 2.6 are:

❶ Statements are taken from the input stack one at a time. The %LET statement is the first one to be processed by the word scanner.

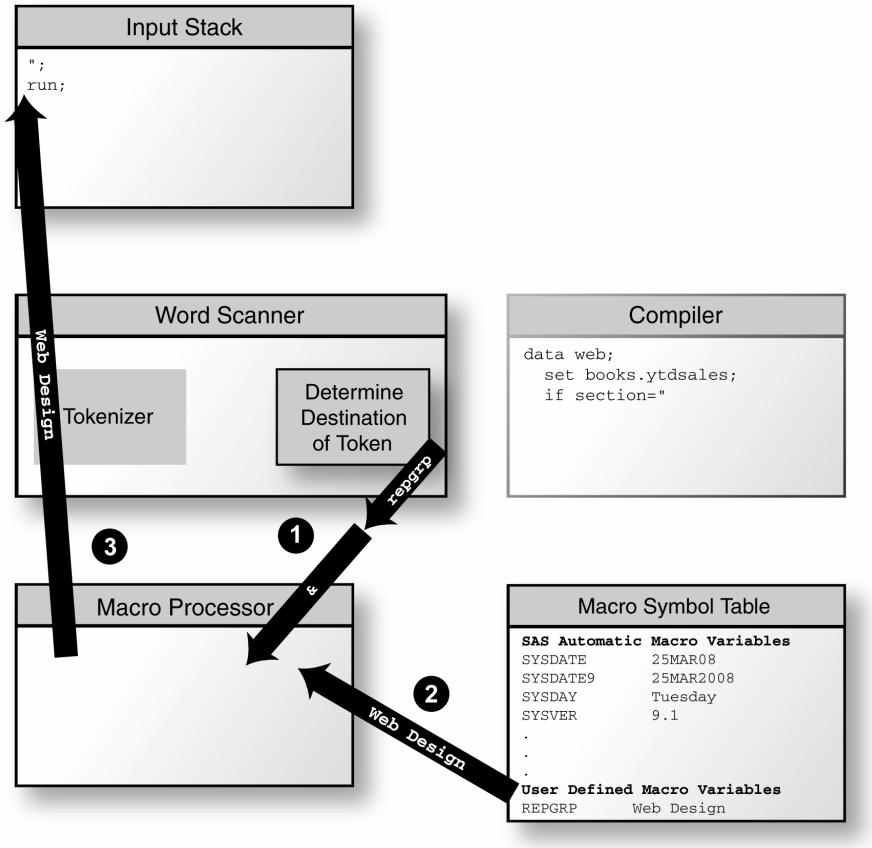
❷ The word scanner detects a macro trigger when it encounters a percent sign (%) followed by the word LET. This causes the word scanner to direct the tokens that follow to the macro processor. The word scanner stops sending tokens to the macro processor when it encounters the semicolon (;) that terminates the %LET statement.

❸ Last, the macro processor places the macro variable REPGRP and its associated text, *Web Design*, in the macro symbol table.

No quotation marks enclose the literal token *Web Design*. This is one way in which the macro language is different from the SAS language. Macro variable values are always text; quotation marks are not needed to indicate text constants in the macro language.

The word scanner continues to tokenize the program. It now encounters the macro variable reference to REPGRP and directs resolution of this reference to the macro processor. This is shown in Figure 2.7.

Figure 2.7 The macro processor resolves the macro variable reference &REPGRP



The three steps in Figure 2.7 are:

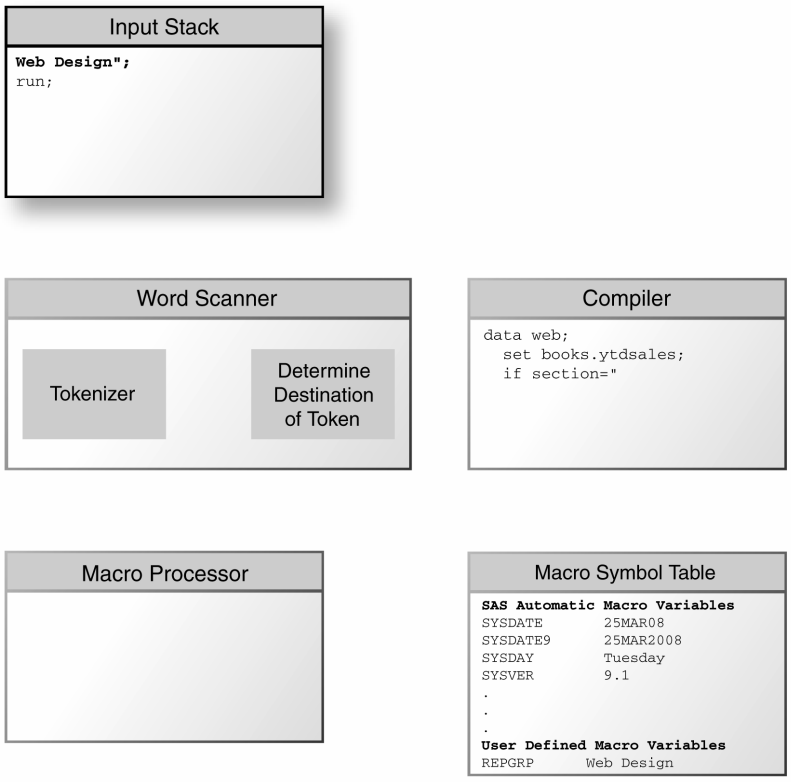
❶ When the word scanner encounters the ampersand (&) followed by REPGRP, it directs processing to the macro processor.

❷ The macro processor looks up the macro variable REPGRP and takes its value from the macro symbol table.

❸ The macro processor places the value of the macro variable REPGRP on top of the input stack.

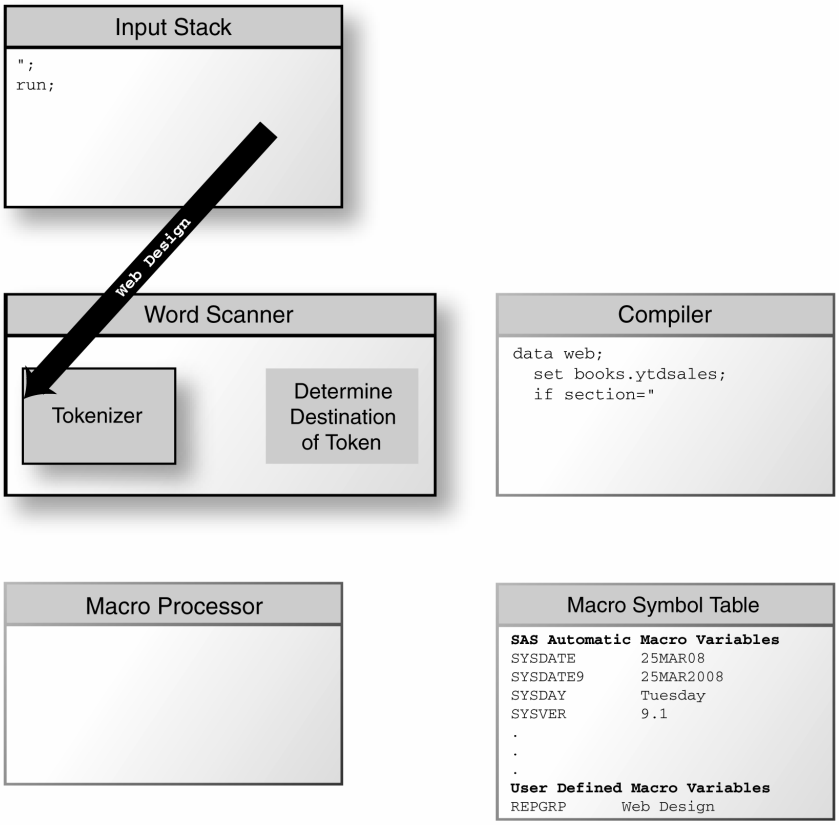
The value of the macro variable REPGRP is now on top of the input stack as shown in Figure 2.8. Remember that the macro variable reference in the SAS language IF statement was enclosed in double quotation marks. Therefore, the value of the macro variable is treated as one literal token.

Figure 2.8 The value of macro variable REPGRP is on top of the input stack



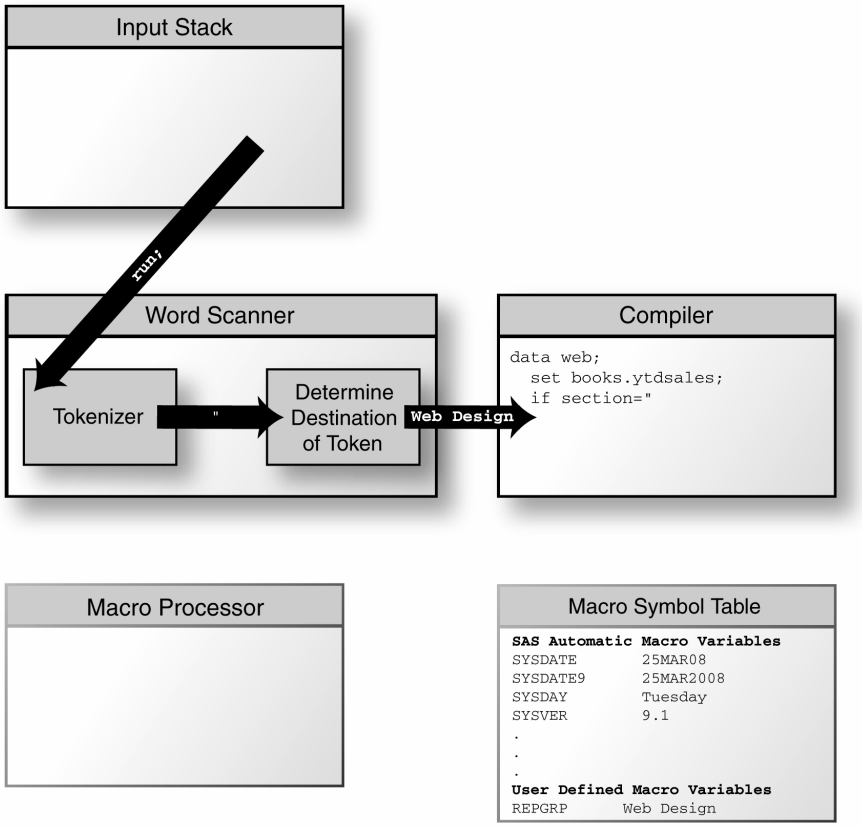
Next, the value of the macro variable is transferred from the input stack to the word scanner. Figure 2.9 shows this action.

Figure 2.9 The value of the macro variable REPGRP is transferred to the word scanner



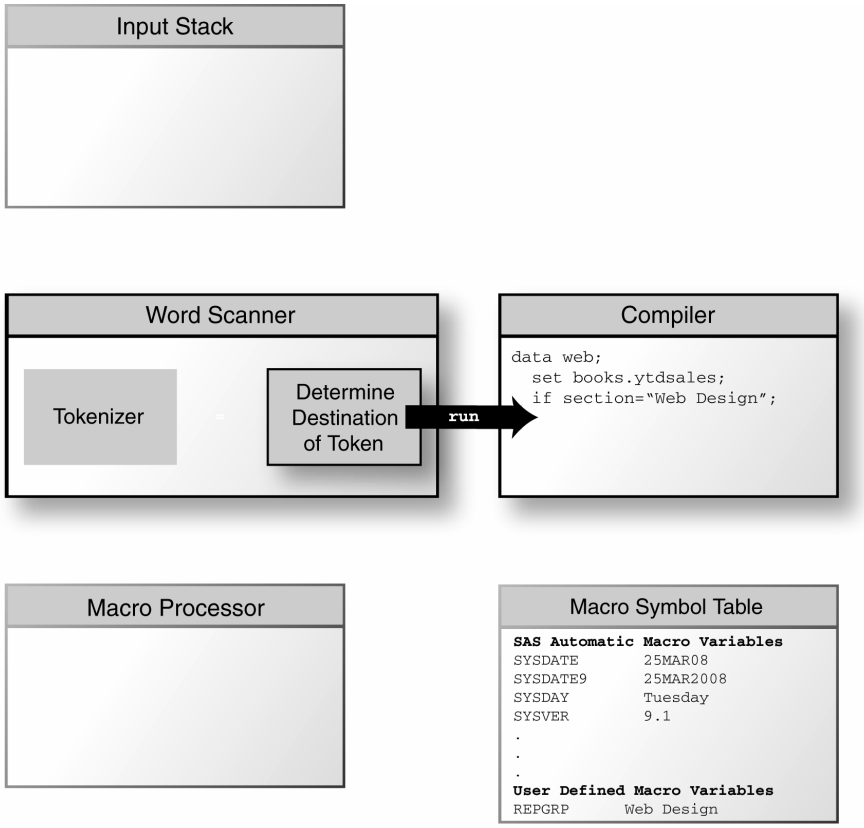
The value of the macro variable REPGRP passes through the word scanner as a literal token and is transferred to the compiler. Now, the last statement in the DATA step, RUN, is sent to the word scanner as shown in Figure 2.10.

Figure 2.10 The compiler receives the value of the macro variable REPGRP and the last statement in the DATA step is sent to the word scanner



Finally, the RUN statement terminating the DATA step is sent to the compiler and the compiled step executes, which is shown in Figure 2.11.

Figure 2.11 The RUN statement is transferred to the compiler and the compiled step executes



In conclusion, when you are writing SAS programs that include macro facility features, remember the distinction between when SAS language is processed and when macro language is processed. Macro language builds SAS language. Macro language is resolved before SAS language is compiled and executed. The discussion on how SAS processes macro language continues in Chapter 5.

Chapter 3

Macro Variables

Introduction 40

Basic Concepts of Macro Variables 40

Referencing Macro Variables 42

Understanding Macro Variable Resolution and the Use of Single and Double Quotation Marks 44

Displaying Macro Variable Values 46

- Using the %PUT Statement 46
- Displaying Macro Variable Values As They Resolve by Enabling the SYMBOLGEN Option 50

Understanding Automatic Macro Variables 52

Understanding User-Defined Macro Variables 56

- Creating Macro Variables with the %LET Statement 56

Combining Macro Variables with Text 59

- Placing Text before a Macro Variable Reference 60
- Placing Text after a Macro Variable Reference 61

Concatenating Permanent SAS Data Set Names and Catalog Names with Macro Variables 63

Referencing Macro Variables Indirectly 65

Resolving Two Ampersands That Precede a Macro Variable Reference 65

Resolving Multiple Ampersands before a Macro Variable Reference 69

Introduction

Macro variables are the most fundamental part of the SAS macro facility. They are the tools to use when you want to begin writing reusable programs. There is relatively little to learn and yet there is great potential in the application of macro variables.

This chapter describes how to define and use SAS macro variables as symbols for text substitution. Before you finish this chapter, you will be able to write programs that contain macro variables. Programming with macro variables will give you an appreciation of the timing of macro processing and provide you with a foundation for understanding the features described later in this book.

Basic Concepts of Macro Variables

Some of the many features of macro variables are summarized in the following list.

- **A macro variable can be referenced anywhere in a SAS program other than in data lines.**
  When the macro processor encounters the reference, the value assigned to the macro variable is substituted in the reference's place. Macro variables can modify SAS language statements in DATA steps, PROC steps, and SCL programs.
- **Macro variables can be used in open code as well as in macro programs.**
  When macro variables are used outside of macro programs, they are in open code. Macro programs are described in the next chapter. The examples in this chapter deal only with macro variables in open code.

- **Macro variables can be created by SAS and by your programs.**
  There are two types of macro variables: automatic macro variables and user-defined macro variables. Automatic macro variables are defined by SAS each time a SAS session is started, and they remain available for you to use throughout your SAS session. Most automatic macro variables contain information about your SAS session such as time of day that the session was invoked, version of SAS, and the operating system you're using. Most automatic macro variable values remain constant throughout your SAS session and most cannot be modified by you.

  User-defined macro variables are defined by you for your own applications. They can be defined anywhere in your SAS program other than in data lines.

- **Macro variables can be stored in either the global symbol table or in a local symbol table.**
  When a macro variable is created, the macro processor adds the macro variable to a macro symbol table. There are two types of macro symbol tables: global and local.

  Macro variables created in open code reside in the global symbol table. A macro variable created in a macro program can reside either in a macro symbol table local to that macro program or in the global symbol table. Since the examples in this chapter deal with macro variables created in open code, these macro variables reside in the global symbol table.

  The value that you assign to a macro variable stored in the global symbol table stays the same throughout the SAS session unless you change it. SAS stores automatic macro variables in the global symbol table.

  The value assigned to a macro variable created in a macro program and defined as local to a macro program stays the same throughout execution of the macro program unless you change it. A local macro variable is deleted when the macro program that created it ends.

- **Macro variable values are text values**.
  All values assigned to macro variables are considered text values. This includes numbers. When you want to do calculations with macro variables, you must tell the macro processor to treat the values as numbers. The result of a calculation is considered a text value.

  The case of the character value assigned to a macro variable is preserved. That is, a value in lowercase remains lowercase, uppercase remains uppercase, and mixed case remains mixed case.

Leading and trailing blanks are removed from the macro variable value before the value is placed in the macro symbol table unless masked by special macro functions. (These functions are described in Chapters 6 and 8.)

The maximum length of text that can be assigned to a macro variable value in SAS®9 is 65,534 (64K); the minimum length is zero characters. You do not have to declare the length of a macro variable; its length is determined each time a value is assigned.

- **The name assigned to a macro variable must be a valid SAS name.**
  A macro variable name can be up to 32 characters in SAS®9. The macro variable name must start with a letter or underscore and continue with letters, numbers, or underscores. For a list of reserved words that should not be used to name macro variables, see Appendix B.
- **Macro variables are not data set variables, and their purpose is different from data set variables.**
  Macro variables help you build your SAS programs and do not directly relate to observations in a data set. A macro variable has only one value while a data set variable can have multiple values, one for each observation in a data set.

Referencing Macro Variables

You tell the macro processor to resolve a macro variable value by preceding the macro variable name with an ampersand (&). When referencing macro variables in a SAS statement, double quotation marks enclosing a string allow resolution of macro variable references while single quotation marks do not. The next section describes the use of quotation marks.

Example 3.1: Defining and Referencing Macro Variables

The two %LET statements in Program 3.1a create two macro variables, REPTITLE and REPVAR, in open code and assign them values. (The %LET statement is described in more detail later in this chapter.) The program then references the macro variables in the TITLE statement and in the TABLES statement associated with PROC FREQ. Program 3.1 also references an automatic macro variable, SYSDAY, that was defined by SAS at the start of the SAS session.

Note the double quotation marks around the text on the TITLE statement.

Program 3.1a

```
%let reptitle=Book Section;
%let repvar=section;

title "Frequencies by &reptitle as of &sysday";
proc freq data=books.ytdsales;
  tables &repvar;
run;
```

After the macro processor resolves the macro variable references, and assuming Program 3.1a was submitted on a Friday, the program that executes follows. The items derived from the macro variable values are in bold.

```
title "Frequencies by Book Section as of Friday";
proc freq data=books.ytdsales;
  tables section;
run;
```

Since these macro variables were created in open code, you can reference them anywhere in your program any number of times. The values of these macro variables remain the same until you tell the macro processor to change them. Program 3.1b adds a PROC MEANS step to Program 3.1a.

Program 3.1b

```
%let reptitle=Book Section;
%let repvar=section;

title "Frequencies by &reptitle as of &sysday";
proc freq data=books.ytdsales;
 tables &repvar;
run;

title "Means by &reptitle as of &sysday";
proc means data=books.ytdsales;
  class &repvar;
  var saleprice;
run;
```

The two user-defined macro variables REPTITLE and REPVAR were defined only once, but they were referenced more than once. After the macro processor resolves the macro variable references, and assuming Program 3.1b was submitted on a Friday, the program that executes follows. The results from resolving the macro variable references are in bold.

```
title "Frequencies by Book Section as of Friday";
proc freq data=books.ytdsales;
  tables section;
run;

title "Means by Book Section as of Friday";
proc means data=books.ytdsales;
  class section;
  var saleprice;
run;
```

Understanding Macro Variable Resolution and the Use of Single and Double Quotation Marks

When you want a macro variable's value to be included as part of a literal string in the SAS language, you must enclose the string with double quotation marks. A macro variable reference enclosed within single quotation marks is not resolved. The word scanner does not look for macro triggers in the characters between single quotation marks.

Example 3.2: Resolving Macro Variables Enclosed in Quotation Marks

The first TITLE statement in Program 3.2 encloses text in double quotation marks. The macro variable references on that statement are resolved. The second TITLE statement has text enclosed in single quotation marks. The macro variable references on that statement are not resolved.

Program 3.2

```
%let reptitle=Section;
%let repvar=section;

title "Frequencies by &reptitle as of &sysday";
proc freq data=books.ytdsales;
  tables &repvar;
run;

title 'Means by &reptitle as of &sysday';
proc means data=books.ytdsales sum maxdec=2;
  class &repvar;
  var saleprice;
run;
```

After the macro processor resolves the macro variable references, and assuming Program 3.2 was submitted on a Friday, the program that executes follows. The items derived from the macro variable values are in bold. The unresolved macro variable references are underlined and in bold.

```
title "Frequencies by Section as of Friday";
proc freq data=books.ytdsales;
  tables section;
run;

title 'Means by &reptitle as of &sysday';
proc means data=books.ytdsales sum maxdec=2;
  class section;
  var saleprice;
run;
```

The macro variable references on the second TITLE statement are not sent to the macro processor for resolution. The references instead are treated as part of the text in the TITLE statement, and thus no warnings or error messages are displayed. Output 3.1 presents the output from the PROC MEANS step in Program 3.2.

Output 3.1 Output for PROC MEANS step in Program 3.2 with title enclosed in single quotation marks

```
              Means by &reptitle as of &sysday

                    The MEANS Procedure

          Analysis Variable : saleprice Sale Price

                                        N
Section                               Obs             Sum
-----------------------------------------------------------
Certification and Training            726        31648.52
Internet                             1456        62295.78
Networks and Telecommunication        717        30803.81
Operating Systems                     922        39779.11
Programming and Applications         1429        62029.41
Web Design                            846        37121.52
-----------------------------------------------------------
```

Displaying Macro Variable Values

Displaying macro variable values as the macro processor resolves them is very useful in showing you how and when the macro processor does its work. Furthermore, this information can help you debug your programs. See Chapter 12 for information on debugging macro language.

Two ways to display macro variable values are with the macro language statement %PUT and with the SAS system option SYMBOLGEN. Both of these features write the values of macro variables to the SAS log.

Using the %PUT Statement

The %PUT statement instructs the macro processor to write information to the SAS log. Text and macro variable values can be displayed with %PUT. The %PUT statement can be submitted by itself from the windowing environment Editor or from within a SAS program. Since %PUT is a macro language statement, it does not need to be part of a DATA step or PROC step, nor *can* it be part of a DATA step or PROC step. A %PUT statement displays only text and information about macro variables.

The syntax of the %PUT statement follows.

```
%put <text |
      _ALL_ | _AUTOMATIC_ | _GLOBAL_ | _LOCAL_ | _USER_ |
      ERROR: | WARNING: | NOTE: >;
```

The text option includes both literal text and references to macro variables. The first five options are keywords that list different classifications of macro variables. The last three options are keywords that can simulate SAS generated messages. Table 3.1 describes the features of these eight %PUT statement options.

Table 3.1 %PUT statement options

Option	Usage
ALL	Lists the values of all user-defined and automatic macro variables.
AUTOMATIC	Lists the values of automatic macro variables. The automatic variables listed depend on the SAS products installed at your site and on your operating system.
GLOBAL	Lists user-defined global macro variables.
LOCAL	Lists user-defined local macro variables. Local macro variables are those defined in the currently executing macro program that have not been designated as global macro variables.
USER	Lists user-defined global and local macro variables.
ERROR:	Simulates a SAS error message by displaying the text ERROR: and remaining specifications on the %PUT statement in red.
WARNING:	Simulates a SAS warning message by displaying the text WARNING: and remaining specifications on the %PUT statement in green.
NOTE:	Simulates a SAS note message by displaying the text NOTE: and remaining specifications on the %PUT statement in blue.

When you debug your macro programming or when you write complex macro programs that others can use, you might find the ERROR:, WARNING:, and NOTE: features of the %PUT statement useful. These features display text in different colors just like SAS generated ERROR, NOTE, and WARNING messages. Note that the keywords must be capitalized and must terminate with a colon (:). Some examples in Chapter 12 make use of these features when debugging macro programs.

Example 3.3: Submitting %PUT _AUTOMATIC_

The _AUTOMATIC_ option on the %PUT statement in Program 3.3 lists the values of all the automatic macro variables.

Program 3.3

```
%put _automatic_;
```

An excerpt of the SAS log after submitting Program 3.3 from the Editor of a SAS®9 session under Windows XP follows.

```
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
AUTOMATIC AFSTR1
AUTOMATIC AFSTR2
AUTOMATIC FSPBDV
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCC 3000
AUTOMATIC SYSCHARWIDTH 1
AUTOMATIC SYSCMD
AUTOMATIC SYSDATE 12JAN08
AUTOMATIC SYSDATE9 12JAN2008
AUTOMATIC SYSDAY Saturday
AUTOMATIC SYSDEVIC
AUTOMATIC SYSDMG 0
AUTOMATIC SYSDSN       _NULL_
AUTOMATIC SYSENDIAN LITTLE
AUTOMATIC SYSENV FORE
AUTOMATIC SYSERR 0
AUTOMATIC SYSFILRC 0
AUTOMATIC SYSINDEX 8
AUTOMATIC SYSINFO 0
AUTOMATIC SYSJOBID 2736
AUTOMATIC SYSLAST BOOKS.YTDSALES
AUTOMATIC SYSLCKRC 0
AUTOMATIC SYSLIBRC 0
AUTOMATIC SYSMACRONAME
AUTOMATIC SYSMAXLONG 2147483647
AUTOMATIC SYSMENV S
AUTOMATIC SYSMSG
AUTOMATIC SYSNCPU 2
AUTOMATIC SYSPARM
AUTOMATIC SYSPBUFF
AUTOMATIC SYSPROCESSID 41D5AD21T288F5C34020000000000000
AUTOMATIC SYSPROCESSNAME DMS Process
AUTOMATIC SYSPROCNAME
AUTOMATIC SYSRC 0
AUTOMATIC SYSSCP WIN
AUTOMATIC SYSSCPL XP_PRO
AUTOMATIC SYSSITE 0099999999
AUTOMATIC SYSSIZEOFLONG 4
AUTOMATIC SYSSIZEOFUNICODE 2
AUTOMATIC SYSSTARTID
AUTOMATIC SYSSTARTNAME
AUTOMATIC SYSTIME 13:58
AUTOMATIC SYSUSERID My Name
AUTOMATIC SYSVER 9.1
```

```
AUTOMATIC SYSVLONG 9.01.01M3P061705
AUTOMATIC SYSVLONG4 9.01.01M3P06172005
```

Example 3.4: Submitting %PUT _GLOBAL_

Once you start writing macro programs, you might find the _GLOBAL_ and _LOCAL_ references on the %PUT statement useful in differentiating the domains of your macro variables. The _GLOBAL_ reference lists in the SAS log all user-defined macro variables stored in the global symbol table, while the _LOCAL_ reference lists in the SAS log all user-defined macro variables stored in the local symbol table defined within a macro program.

The _GLOBAL_ option on the %PUT statement in Program 3.4 lists in the SAS log the two macro variables the program defines in open code. It identifies them as global macro variables, and the output includes the text “GLOBAL”.

Program 3.4

```
%let reptitle=Book Section;
%let reptvar=section;
%put _global_;
```

The SAS log from these statements follows.

```
87   %symdel x;
88   %let reptitle=Book Section;
89   %let reptvar=section;
90   %put _global_;
GLOBAL REPTITLE Book Section
GLOBAL REPTVAR section
```

Example 3.5: Submitting %PUT Statements to Display Text and Macro Variable Values

Text added to %PUT statements can make the results more informative and easier to read. Program 3.1a is modified below in Program 3.5 to include three %PUT statements that list text and macro variable values.

Program 3.5

```
%let reptitle=Book Section;
%let reptvar=section;
%put My macro variable REPTITLE has the value &reptitle;
%put My macro variable REPTVAR has the value &reptvar;
%put Automatic macro variable SYSDAY has the value &sysday;
```

```
title "Frequencies by &reptitle as of &sysday";
proc freq data=books.ytdsales;
  tables &reptvar;
run;
```

The SAS log from Program 3.5 follows:

```
31   %let reptitle=Book Section;
32   %let reptvar=section;
33
34   %put My macro variable REPTITLE has the value &reptitle;
My macro variable REPTITLE has the value Book Section
35   %put My macro variable REPTVAR has the value &reptvar;
My macro variable REPTVAR has the value section
36   %put The automatic macro variable SYSDAY has the value
&sysday;
Automatic macro variable SYSDAY has the value Friday
37
38   title "Frequencies by &reptitle as of &sysday";
39   proc freq data=books.ytdsales;
40     tables &reptvar;
41   run;

NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.04 seconds
      cpu time            0.03 seconds
```

Note that the values of the macro variables on the %PUT statements are displayed in the SAS log. The values of the same macro variables in the PROC FREQ step are not displayed as they are resolved. The next section describes how you can display, in the SAS log, the values of macro variables that are included in SAS language statements.

Displaying Macro Variable Values As They Resolve by Enabling the SYMBOLGEN Option

SAS option SYMBOLGEN is the most useful SAS option to use as you start writing programs that contain macro variables. With SYMBOLGEN enabled, SAS presents the results of the resolution of macro variables in the SAS log. SYMBOLGEN displays the value of a macro variable in the SAS log near the statement with the macro variable reference.

SYMBOLGEN shows the values of both automatic and user-defined macro variables. The SYMBOLGEN option helps you debug your programs. If you are getting unexpected results when using macro variables, enable this option and read the SAS log.

It is easier to enable SYMBOLGEN than to write %PUT statements. However, SYMBOLGEN displays the values of **all** macro variables you reference in your program, while %PUT lets you selectively display macro variable values. The %PUT statement gives you control of where and when a macro variable value is displayed. Option SYMBOLGEN displays macro variable values only when they are referenced; macro variable values are not displayed at the time they are created with %LET.

Example 3.6: Displaying Macro Variable Values with the SYMBOLGEN Option

Program 3.6 references three macro variables: two user-defined and one automatic. The OPTIONS statement enables SYMBOLGEN. (To turn off SYMBOLGEN, enter: `options nosymbolgen;`)

Note that Program 3.6 is the same as Program 3.1b with the addition of the OPTIONS statement.

Program 3.6

```
options symbolgen;

%let reptitle=Book Section;
%let repvar=section;

title "Frequencies by &reptitle as of &sysday";
proc freq data=books.ytdsales;
  tables &repvar;
run;

title "Means by &reptitle as of &sysday";
proc means data=books.ytdsales;
  class &repvar;
  var saleprice;
run;
```

The SAS log for Program 3.6 follows. The SYMBOLGEN messages are in bold.

```
26   options symbolgen;
27
28   %let reptitle=Book Section;
29   %let repvar=section;
30
SYMBOLGEN:  Macro variable REPTITLE resolves to Book Section
```

```
SYMBOLGEN:  Macro variable SYSDAY resolves to Tuesday
31   title "Frequencies by &reptitle as of &sysday";
32   proc freq data=books.ytdsales;
SYMBOLGEN:  Macro variable REPVAR resolves to section
33     tables &repvar;
34   run;
NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.15 seconds
      cpu time            0.01 seconds

SYMBOLGEN:  Macro variable REPTITLE resolves to Book Section
SYMBOLGEN:  Macro variable SYSDAY resolves to Tuesday
35
36   title "Means by &reptitle as of &sysday";
37   proc means data=books.ytdsales;
SYMBOLGEN:  Macro variable REPVAR resolves to section
38     class &repvar;
39     var saleprice;
40   run;
NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.03 seconds
```

Understanding Automatic Macro Variables

SAS automatically defines a set of macro variables when a SAS session starts. The macro processor maintains these variables and their values in the macro symbol table. Other than in data lines, you can use these macro variables anywhere in your SAS programs. Typically, automatic macro variables are used to store information about your SAS session such as time of day the SAS session was invoked, version of SAS, and site number.

Table 3.2 lists a few automatic macro variables, and Appendix A presents a more complete list. There are three types of automatic macro variables. The macro variables in Table 3.2 are grouped by type:

- The first type of automatic macro variable has values fixed at the start of the SAS session; these values never change during the SAS session.
- The values of the second type are also set at the start of the SAS session, but these values can be changed by SAS.
- The values of the third type are initialized at the start of the SAS session and these values can be modified by you or by SAS.

Table 3.2 A sample of automatic macro variables

Type of Automatic Macro Variable	Automatic Macro Variable	Description
Values that remain fixed	SYSDATE	the character value that is equal to the date the SAS session started in DATE7. format
	SYSDATE9	the character value that is equal to the date the SAS session started in DATE9. format
	SYSDAY	text of the day of the week the SAS session started
	SYSVER	the character value representing the release number of SAS that is executing
	SYSTIME	the character value representing the time the SAS session started
Values that can be changed by SAS	SYSERR	return code set at end of each DATA step and most PROC steps
	SYSERRORTEXT	text of the last error message generated within the SAS log
	SYSFILRC	return code from most recent FILENAME statement
	SYSLIBRC	return code from most recent LIBNAME statement

(*continued*)

Table 3.2 (*continued*)

Type of Automatic Macro Variable	Automatic Macro Variable	Description
Values that can be changed by SAS (continued)	SYSMACRONAME	returns the name of the currently executing macro program
	SYSRC	returns a value corresponding to an error condition
	SYSWARNINGTEXT	text of the last warning message generated within the SAS log
Values that can be changed by you or by SAS	SYSDSN	name of the most recently created data set in two fields: WORK TEMP
	SYSLAST	name of the most recently created data set in one field: WORK.TEMP

The values of all automatic macro variables are text values— even the date and time values are treated as text.

The SYS prefix is reserved for automatic macro variables. Avoid using this prefix when creating your own macro variables. Also, don't define any of your own macro variables with the name of an automatic macro variable. If you do, you will probably get an error message since most automatic macro variables are read-only and fixed in value at the time of their definition. The few automatic macro variables that can be modified are defined for items that can change throughout your SAS session, such as last data set accessed.

Example 3.7: Using Automatic Variables

Program 3.7 references five automatic macro variables. The automatic macro variable names are in bold. Assume that the program was run on June 22, 2007. The program extracts a subset of observations from BOOK.YTDSALES for the Web Design section for books sold from June 16–June 22, 2007, and PROC PRINT lists the selected observations.

Program 3.7

```
data web;
  set books.ytdsales;
  if section='Web Design' and datesold > "&sysdate"d-6;
run;

proc print data=web;
  title "Web Design Titles Sold in the Past Week";
  title2 "Report Date: &sysday &sysdate &systime";
  footnote1 "Data Set Used: &syslast SAS Version: &sysver";

  var booktitle datesold saleprice;
run;
```

Output 3.2 presents partial output from Program 3.7.

Output 3.2 Partial output from Program 3.7 that contains macro variables

```
          Web Design Titles Sold in the Past Week
             Report Date: Tuesday 22JUN07 13:50

 Obs         booktitle            datesold     saleprice

   1    Web Design Title 250    06/22/2007        $40.95
   2    Web Design Title 454    06/22/2007        $36.95
   3    Web Design Title 92     06/23/2007        $50.95
   4    Web Design Title 84     06/17/2007        $40.95
   5    Web Design Title 16     06/28/2007        $36.86
   6    Web Design Title 302    06/26/2007        $40.95
   7    Web Design Title 368    06/25/2007        $36.86
        .
        .
        .

Data Set Used: WORK.WEB                          SAS Version: 9.1
```

Understanding User-Defined Macro Variables

The applications of user-defined macro variables are limitless. Other than in data lines, user-defined macro variables can be created and referenced anywhere in your programs. The macro processor maintains the values of user-defined macro variables in the macro symbol table. Tasks that you can accomplish with user-defined macro variables include the following:

- annotating reports
- selecting subsets of data sets
- passing information between PROC steps and DATA steps
- using variables in macro programs

Creating Macro Variables with the %LET Statement

One way to create and update a macro variable is with a %LET statement. The %LET statement tells the macro processor to add the macro variable to the macro symbol table if the macro variable does not exist. It also tells the macro processor to associate a text value with that macro variable name.

The %LET statement is written as follows and is terminated with a semicolon. No quotation marks are required to enclose the macro variable value.

```
%let macro-variable-name=macro-variable-value;
```

The %LET statement can be submitted from the Editor window or from a SAS program. It is a SAS macro language statement, not a SAS language statement. It creates macro variables, not SAS data set variables.

The %LET statement is executed as soon as the macro processor receives it from the word scanner. For example, if you place a %LET statement within a DATA step, the %LET statement is processed before the DATA step executes. This happens in the midst of the tokenization of the SAS language statements in the DATA step, before compilation and execution of the DATA step. The DATA step eventually executes after all the SAS language statements in the DATA step are collected by the compiler.

Example 3.8: Using the %LET Statement

Examples of %LET statements follow in Program 3.8. These statements assign values to macro variables. They demonstrate how the macro language treats macro values as text. Unless otherwise instructed, the macro language does not do arithmetic calculations as demonstrated by the first few statements. An annotated SAS log showing the resolution of the statements follows Program 3.8.

Program 3.8

```
%let nocalc=53*21 + 100.1;

%let value1=982;
%let value2=813;
%let result=&value1 + &value2;

%let reptext=This report is for ***  Department XYZ  ***;

%let region=Region 3;
%let text=Sales Report;
%let moretext="Sales Report";
%let reptitle=&text &region;
%let reptitl2=&moretext &region;

%let sentence=      This one started with leading blanks.;

%let chars=Symbols: !@#$%^&*;

%let novalue=;

%let holdvars=varnames;
%let &holdvars=title author datesold;
```

The following SAS log results from submission of the preceding statements. A %PUT statement was added after each %LET statement to display the value of the macro variable created with the %LET statement. Text was added to the %PUT statement for the macro variable SENTENCE to more clearly show that the leading blanks were removed.

Specific concepts are identified by number and described after the SAS log.

```
1    %let nocalc=53*21 + 100.1;    ❶
2    %put &nocalc;
53*21 + 100.1

3    %let value1=982;
4    %put &value1;
```

```
982
5    %let value2=813;
6    %put &value2;
813
7    %let result=&value1 + &value2;  ❶
8    %put &result;
982 + 813

9    %let reptext=This report is for ***  Department XYZ  ***;
10   %put &reptext;
This report is for ***  Department XYZ  ***

11   %let region=Region 3;
12   %put &region;
Region 3
13   %let text=Sales Report;  ❷
14   %put &text;
Sales Report
15   %let moretext="Sales Report";
16   %put &moretext;
"Sales Report"
17   %let reptitle=&text &region;
18   %put &reptitle;
Sales Report Region 3
19   %let reptitl2=&moretext &region;
20   %put &reptitl2;
"Sales Report" Region 3

21   %let sentence=      This one started with leading blanks.;
22   %put Now no leading blanks:&sentence;          ❸
Now no leading blanks:This one started with leading blanks.

23   %let chars=Symbols: !@#$%^&*;
24   %put &chars;               ❹
Symbols: !@#$%^&*
25   %let novalue=;
26   %put &novalue;                ❺

27   %let holdvars=varnames;
28   %put &holdvars;
varnames                        ❻
29   %let &holdvars=title author datesold;
30   %put &holdvars;
varnames
31   %put &varnames;
title author datesold
```

General observations to make from the %LET assignments include:

- The macro processor uses the semicolon to detect the end of the assignment of a value to a macro variable.
- All the values that were assigned are acceptable macro variable values.

The following list describes specific observations to make from the preceding code.

❶ The SAS log shows that SAS treats all macro variable values as text. No arithmetic calculations are done. (SAS log line 1 and SAS log line 7)

❷ When assigning values to macro variables, quotation marks are not used to enclose the value (SAS log lines 11 and 13). When quotation marks are used, the quotation marks become part of the text associated with the macro variables. (SAS log line 15) (The use of quotation marks in macro programming is a special topic in the macro facility and is discussed in Chapter 8.)

❸ Leading blanks are removed from the value of a macro variable. (SAS log line 21)

❹ Blanks and special characters are valid macro variable values. The macro processor does not interpret the ampersand (&) and percent (%) symbols as macro triggers when they are not followed by a letter or underscore. (SAS log line 23)

❺ A macro variable can have a null value. (SAS log line 25)

❻ You can assign macro variable values to other macro variables and combine macro variables in a %LET statement to create a new macro variable. The macro processor recognizes the ampersand (&) and percent (%) symbols as macro triggers when they are followed by a letter or underscore. (SAS log lines 27 and 29)

Combining Macro Variables with Text

This section demonstrates some of the interesting ways that you can program with macro variables. When you combine macro variable references with text or with other macro variable references, you can create new macro variable references. These new macro variable references are resolved before the SAS language statements in which they are placed are tokenized.

A concatenation operator is not needed to combine macro variables with text. However, periods (.) act as delimiters of macro variable references and might be needed to delimit a macro variable reference that precedes text.

Placing Text before a Macro Variable Reference

When placing text before a macro variable reference or when combining macro variable references, you do not have to separate the references and text with a delimiter.

Example 3.9: Placing Text before a Macro Variable Reference

Program 3.9 illustrates how you can create a new macro variable reference by placing text or other macro variable references before a macro variable reference. The underlined text indicates where macro variable references are combined with other macro variable references and text. Note that no concatenation operator was used to combine the macro variable references with text.

Both programs start out by defining two macro variables in open code. Statements in the DATA step and PROC step reference these macro variables.

Program 3.9

```
%let mosold=4;
%let level=25;

data book&mosold&level;
  set books.ytdsales(where=(month(datesold)=&mosold));

  attrib over&level length=$3 label="Cost > $&level";

  if cost > &level then over&level='YES';
  else over&level='NO';
run;

proc freq data=book&mosold&level;
title "Frequency Count of Books Sold During Month &mosold";
title2 "Grouped by Cost Over $&level";
  tables over&level;
run;
```

After the macro processor creates the two macro variables and resolves the macro variable references, the program becomes:

```
data book425;
  set books.ytdsales(where=(month(datesold)=4));

  attrib over25 length=$3 label="Cost > $25";

  if cost > 25 then over25='YES';
  else over25='NO';
run;
```

```
proc freq data=book425;
title "Frequency Count of Books Sold During Month 4";
title2 "Grouped by Cost Over $25";
  tables over25;
run;
```

With this technique, you can write a program once and reuse it for a different subset by changing the values of the macro variables. For example, changing the values of the two macro variables in the preceding program to the values in the following two %LET statements produce the same style of report, but on different subsets of the data set.

```
%let mosold=12;
%let level=50;
```

After the macro processor creates the two macro variables and resolves the macro variable references, the program becomes:

```
data book1250;
  set books.ytdsales(where=(month(datesold)=12));

  attrib over50 length=$3 label="Cost > $50";

  if cost > 50 then over50='YES';
  else over50='NO';
run;

proc freq data=book1250;
title "Frequency Count of Books Sold During Month 12";
title2 "Grouped by Cost Over $50";
  tables over50;
run;
```

Placing Text after a Macro Variable Reference

When you follow a macro variable reference with text, you must place a period at the end of the macro variable reference to terminate the reference. The macro processor recognizes that a period signals the end of a macro variable name and determines that the name of the macro variable is the text between the ampersand and the period. All macro variable references can be terminated with periods.

The code in Example 3.9 does not require terminating periods for proper resolution of the macro variable references. A space or semicolon after the macro variable reference delimits the macro variable reference. The macro processor knows that blanks and semicolons cannot be part of a macro variable name. Similarly, if you place a macro variable reference after another macro variable reference, the ampersand of the second macro variable reference delimits the previous macro variable reference.

Example 3.10: Placing Text after a Macro Variable Reference

Text follows macro variable references in Program 3.10a. No periods follow the macro variable references so the program does not execute as needed.

The goal of this example is to compute frequency counts for the responses to the first five questions of a customer survey: QUESTION1, QUESTION2, QUESTION3, QUESTION4, and QUESTION5. These five variables are in data set BOOK.SURVEY. Program 3.10a defines one macro variable, PREFIX, that is set to the text of the first part of the five variables' names. The macro variable references on the TABLES statement should resolve to the five variables' names. With the omission of the period delimiter, however, this does not happen. Program 3.10a does not execute.

Program 3.10a

```
*----WARNING: This program does not execute;
%let prefix=QUESTION;

proc freq data=books.survey;
  tables &prefix1 &prefix2 &prefix3 &prefix4 &prefix5;
run;
```

After resolving the macro variable references, the program becomes:

```
proc freq data=books.survey;
  tables &prefix1 &prefix2 &prefix3 &prefix4 &prefix5;
run;
```

The following messages are listed in the SAS log.

```
WARNING: Apparent symbolic reference PREFIX1 not resolved.
WARNING: Apparent symbolic reference PREFIX2 not resolved.
WARNING: Apparent symbolic reference PREFIX3 not resolved.
WARNING: Apparent symbolic reference PREFIX4 not resolved.
WARNING: Apparent symbolic reference PREFIX5 not resolved.
```

Since the five macro variables PREFIX1, PREFIX2, PREFIX3, PREFIX4, and PREFIX5 have not been defined in Program 3.10a, the macro processor cannot resolve the five macro variable references. The macro processor sends the macro variable references back to the input stack as they were received. The word scanner cannot tokenize the TABLES statement. The PROC FREQ step does not execute.

Program 3.10b contains the necessary delimiters that tell the macro processor when the macro variable references end, which are missing in Program 3.10a. Now the macro variable references resolve as desired, and the text that follows the references is concatenated to the results of the resolution.

Program 3.10b

```
*----This program executes correctly;
%let prefix=QUESTION;

proc freq data=books.survey;
  tables &prefix.1 &prefix.2 &prefix.3 &prefix.4 &prefix.5;
run;
```

The macro processor substitutes QUESTION for the &PREFIX macro variable reference. After macro variable resolution, the program becomes:

```
proc freq data=books.survey;
   tables QUESTION1 QUESTION2 QUESTION3 QUESTION4 QUESTION5;
run;
```

Concatenating Permanent SAS Data Set Names and Catalog Names with Macro Variables

The macro processor understands that periods delimit macro variable references. Periods are also used in the SAS language when referring to permanent data sets and catalogs. Permanent data sets and catalogs have multi-part names, each part delimited with a period.

When macro variable references are concatenated with permanent data set names or catalog names, your coding must distinguish the role of the period in your statement. The question to ask yourself when coding these kinds of macro variable references is whether the period terminates the macro variable reference or whether it is part of the name of a data set or catalog.

When a macro variable reference precedes the period in a data set or catalog name, add one extra period after the macro variable reference. The macro processor looks up the macro variable reference delimited by the first period and determines that the macro variable name is complete because of the terminating period. The macro variable value is put on the input stack and the word scanner tokenizes it. The word scanner recognizes the second period as text. That second period is then part of the data set name or catalog name.

Example 3.11: Referencing Permanent SAS Data Set Names and Macro Variables

Program 3.11a illustrates the necessity of using two periods. The intention is to analyze data in data set BOOKSURV.SURVEY1. Macro variable SURVLIB contains the libref.

Program 3.11a

```
*----WARNING: This program does not execute;
%let survlib=BOOKSURV;

proc freq data=&survlib.survey1;
  tables age;
run;
```

After macro variable resolution, the program becomes:

```
*----WARNING: This program does not execute;
proc freq data=BOOKSURVsurvey1;
  tables age;
run;
```

The macro processor does its work before the PROC FREQ statement is completely tokenized. In Program 3.11a, the word scanner suspends processing when it encounters the macro variable reference. The macro processor looks for the value of the SURVLIB macro variable in the global symbol table. The reference to SURVLIB is terminated with a period. The macro processor interprets that period as terminating the macro variable reference. The macro processor finds the value for SURVLIB, which is BOOKSURV and puts BOOKSURV on top of the input stack. The rest of the data set name, SURVEY1, now ends up being concatenated to the text BOOKSURV. The period could be used only once, and the macro processor used it first. The program cannot execute because the data set BOOKSURVSURVEY1 does not exist.

For the program to resolve the macro variable reference and to construct a permanent data set name, add another period as shown in Program 3.11b. The macro processor is done with its work after seeing the first period. The remaining text, `.SURVEY1`, which is the rest of the data set name, is now concatenated to BOOKSURV.

Program 3.11b

```
*----This program executes;
%let survlib=BOOKSURV;

proc freq data=&survlib..survey1;
  tables age;
run;
```

After macro variable resolution, the program becomes:

```
*----This program executes;
proc freq data=BOOKSURV.survey1;
  tables age;
run;
```

Referencing Macro Variables Indirectly

This section discusses the techniques of indirect referencing of macro variables. When working with a series of macro variables, these techniques add more flexibility to your macro programming. In an indirect macro variable reference, the resolution of a macro variable reference leads to the resolution of another macro variable reference.

The macro variable references that have been described so far are written with one ampersand preceding the macro variable name. This is a direct reference to a macro variable. For some applications, it is necessary to add a period to delimit the macro variable reference.

In indirect referencing, more than one ampersand precedes a macro variable reference. The macro processor follows specific rules in resolving references with multiple ampersands. You can take advantage of these rules to create new macro variable references.

The rules that the macro processor uses to resolve macro variable references that contain multiple ampersands follow.

- Macro variable references are resolved from left to right.
- Two ampersands (&&) resolve to one ampersand (&).
- Multiple leading ampersands cause the macro processor to rescan the reference until no more ampersands can be resolved.

Resolving Two Ampersands That Precede a Macro Variable Reference

The first example of referencing macro variables indirectly follows in Program 3.12. Six macro variables define six sections in the computer department of the bookstore. A report program analyzes sales information for a section. The macro variable N represents the section number. Program 3.12 produces the sales information for Section 4, Operating Systems.

The indirect macro variable reference in Program 3.12 is &&SECTION&N. Note that there are two ampersands preceding SECTION. The macro processor scans the macro variable reference twice, once for each of the preceding ampersands.

Program 3.12

```
%let section1=Certification and Training;
%let section2=Internet;
%let section3=Networking and Communication;
%let section4=Operating Systems;
%let section5=Programming and Applications;
%let section6=Web Design;

*----Look for section number defined by macro var n;
%let n=4;
proc means data=books.ytdsales;
  title "Sales for Section: &&section&n";
  where section="&&section&n";
  var saleprice;
run;
```

After macro variable resolution, the preceding program becomes:

```
proc means data=books.ytdsales;
  title "Sales for Section: Operating Systems";
  where section="Operating Systems";
  var saleprice;
run;
```

Now consider what happens if only one ampersand precedes SECTION, and you write the reference as &SECTION&N as shown in Program 3.13a.

The macro processor resolves &SECTION&N in two parts: &SECTION and &N. &N can be resolved, and in this example, &N equals 4. The macro variable &SECTION is not defined and cannot be resolved, thus causing a warning message to be written to the SAS log. The following statements demonstrate how &SECTION&N does not resolve as desired.

Program 3.13a

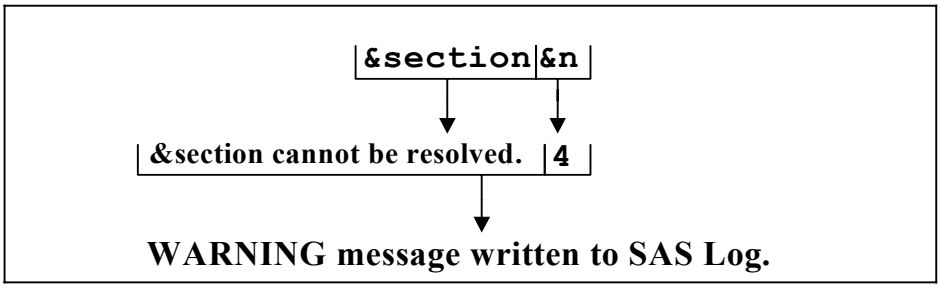
```
options symbolgen;
%let section4=Operating Systems;
%let n=4;

%put &section&n;
```

The SAS log for Program 3.13a follows.

```
1    options symbolgen;
2    %let section4=Operating Systems;
3    %let n=4;
4
5    %put &section&n;
WARNING: Apparent symbolic reference SECTION not resolved.
```

Figure 3.1 shows the process of resolving the macro variable references in Program 3.13a.

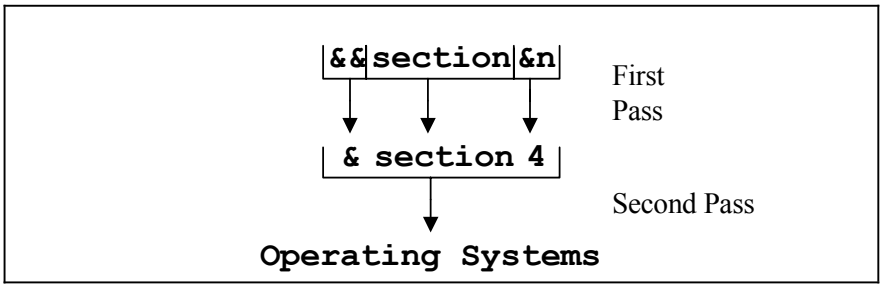
Figure 3.1 How the macro processor resolves the two concatenated macro variable references in Program 3.13a



To resolve the macro variable reference as desired, add another ampersand before SECTION: `&&SECTION&N`. This forces the macro processor to scan the reference twice.

On the first pass, the two ampersands are resolved to one and the reference to `&N` is resolved to 4, yielding `&SECTION4`. On the second pass, the macro variable reference `&SECTION4` is resolved to Operating Systems, as shown in Figure 3.2.

Figure 3.2 Using two ampersands to force the macro processor to scan a macro variable reference twice



Program 3.13a is modified below in Program 3.13b to include another ampersand before SECTION.

Program 3.13b

```
options symbolgen;
%let section4=Operating Systems;
%let n=4;

%put &&section&n;
```

The SAS log for Program 3.13b follows.

```
6    options symbolgen;
7    %let section4=Operating Systems;
8    %let n=4;
9
10   %put &&section&n;
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable N resolves to 4
SYMBOLGEN:  Macro variable SECTION4 resolves to Operating
Systems
Operating Systems
```

Recall that the SYMBOLGEN option traces the resolution of indirect macro variable references. The SAS log for Program 3.12 with that option enabled follows.

```
131  options symbolgen;
132  %let section1=Certification and Training;
133  %let section2=Internet;
134  %let section3=Networking and Communication;
```

```
135  %let section4=Operating Systems;
136  %let section5=Programming and Applications;
137  %let section6=Web Design;
138  *----Look for section number defined by macro var n;
139  %let n=4;
140  proc means data=books.ytdsales;
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable N resolves to 4
SYMBOLGEN:  Macro variable SECTION4 resolves to Operating
Systems
141    title "Sales for Section: &&section&n";
142    where section="&&section&n";
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable N resolves to 4
SYMBOLGEN:  Macro variable SECTION4 resolves to Operating
Systems
143    var saleprice;
144  run;

NOTE: There were 922 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Operating Systems';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.07 seconds
      cpu time            0.01 seconds
```

Resolving Multiple Ampersands before a Macro Variable Reference

Program 3.14 illustrates how the macro processor resolves multiple ampersands preceding a macro variable reference. Three ampersands precede the macro variable reference.

Program 3.14 provides flexibility in specifying the WHERE statement for a PROC MEANS step. The macro variable WHEREVAR is assigned the name of the data set variable that defines the WHERE selection. In the example, the goal is to compute PROC MEANS for Section 4, Operating Systems.

Program 3.14

```
options symbolgen;
%let section1=Certification and Training;
%let section2=Internet;
%let section3=Networking and Communication;
%let section4=Operating Systems;
%let section5=Programming and Applications;
```

```
%let section6=Web Design;
%let dept1=Computer;
%let dept2=Reference;
%let dept3=Science;

%let n=4;
%let wherevar=section;

proc means data=books.ytdsales;
  title "Sales for &wherevar: &&&wherevar&n";
  where &wherevar="&&&wherevar&n";
  var saleprice;
run;
```

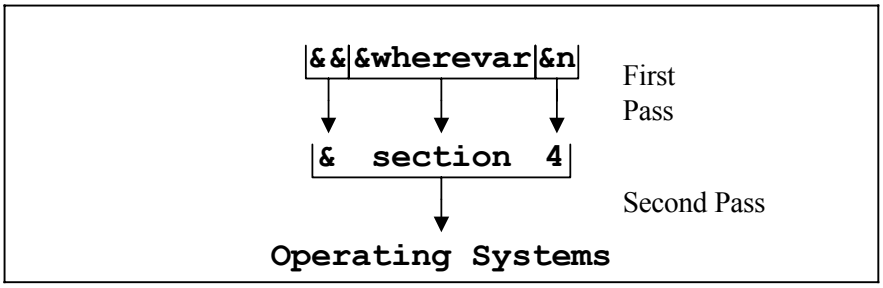
The SAS log for Program 3.14 follows. Note how SYMBOLGEN traces each scanning step in the resolution of the macro variable reference.

```
29   options symbolgen;
30   %let section1=Certification and Training;
31   %let section2=Internet;
32   %let section3=Networking and Communication;
33   %let section4=Operating Systems;
34   %let section5=Programming and Applications;
35   %let section6=Web Design;
36   %let dept1=Computer;
37   %let dept2=Reference;
38   %let dept3=Science;
39
40   %let n=4;
41   %let wherevar=Section;
42
43   proc means data=books.ytdsales;
SYMBOLGEN:  Macro variable WHEREVAR resolves to Section
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable WHEREVAR resolves to Section
SYMBOLGEN:  Macro variable N resolves to 4
SYMBOLGEN:  Macro variable SECTION4 resolves to Operating
Systems
44     title "Sales for &wherevar: &&&wherevar&n";
SYMBOLGEN:  Macro variable WHEREVAR resolves to Section
45     where &wherevar="&&&wherevar&n";
SYMBOLGEN:  && resolves to &.
SYMBOLGEN:  Macro variable WHEREVAR resolves to Section
SYMBOLGEN:  Macro variable N resolves to 4
SYMBOLGEN:  Macro variable SECTION4 resolves to Operating
Systems
```

```
46     var saleprice;
47   run;

NOTE: There were 922 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Operating Systems';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

The macro processor scans the reference &&&WHEREVAR&N twice. Figure 3.3 shows how the macro processor breaks down this reference.

Figure 3.3 How the macro processor resolves multiple ampersands preceding a macro variable reference



Chapter 4

Macro Programs

Introduction 74

Creating Macro Programs 74

Executing a Macro Program 78

Displaying Notes about Macro Program Compilation in the SAS Log 80

Displaying Messages about Macro Program Processing in the SAS Log 82

- Using MPRINT to Display the SAS Statements Submitted by a Macro Program 83
- Using the MLOGIC Option to Trace Execution of a Macro Program 84

Passing Values to a Macro Program through Macro Parameters 85

- Specifying Positional Parameters in Macro Programs 85
- Specifying Keyword Parameters in Macro Programs 88
- Specifying Mixed Parameter Lists in Macro Programs 92
- Defining a Macro Program That Can Accept a Varying Number of Parameter Values 95

Introduction

A macro program is another tool for text substitution. Macro programs are like macro variables: text is associated with a name. The difference is that the text that is substituted by a macro program can be created using more powerful macro language programming statements than have been described so far. Combining these macro language statements with macro variables and macro functions allows you to write more complicated instructions for the macro processor than what you can write with macro variables alone.

Each macro program is assigned a name. When you reference a macro program, the statements inside the macro program execute. The text that results from the execution is substituted into your SAS program at the location of the macro program reference.

Macro programs use macro variables and macro language statements to generate the text that builds your SAS programs. The SAS macro programming language has the same type of statements as other programming languages. Many macro language statements resemble their SAS language counterparts.

Several macro language statements can be used only inside macro programs. The macro language statements that we have seen so far, %LET and %PUT, can be used inside or outside macro programs. Macro language statements and macro variable references placed outside a macro program, like we've seen in the last chapter, are referred to as being in open code.

This chapter describes how to create and use macro programs.

Creating Macro Programs

A macro program is defined with the following statements:

```
%MACRO program <(parameter-list)></ option(s)>;;
    <text>
%MEND <program>;
```

It starts with the %MACRO statement and terminates with the %MEND statement. Table 4.1 lists the elements of a macro program definition. This table briefly describes the %MACRO statement options. Many are illustrated with examples later in this book. For additional information and for those beyond the scope of this book, refer to *SAS Macro Language: Reference*.

Table 4.1 Elements of a macro program definition

Macro Program Element	Description	
%MACRO	Marks the beginning of the macro program definition.	
program	Name assigned to the macro program. The macro program name must be a valid SAS name (no more than 32 characters in SAS®9) and must start with a letter or underscore with the remaining characters any combination of letters, numbers, and underscores. The macro program name must not be a reserved word in the macro facility (see Appendix B). Note that the macro program name is not preceded with a percent sign on the %MACRO statement.	
<parameter-list>	Names one or more local macro variables whose values you specify when you invoke the macro program. Defining parameters is optional. The two types of parameters, positional and keyword, are described in this chapter.	
</option(s)>	The optional arguments include:	
	CMD	Specifies that the macro program can accept either a name-style invocation or a command-style invocation. Macro programs defined with the CMD option are sometimes called *command-style macros*. For additional information on this option, refer to *SAS Macro Language: Reference.*
	DES='*text*'	Specifies a description (up to 40 characters) for the macro program entry in the macro catalog.
	PARMBUFF **PBUFF**	Assigns the entire list of parameter values in a macro program call as the value of the automatic macro variable SYSPBUFF. Using the PARMBUFF option, you can define a macro program that accepts a varying number of parameter values.
	STMT	Specifies that the macro can accept either a name-style invocation or a statement-style invocation. Macros defined with the STMT option are sometimes called *statement-style macros*. For additional information on this option, refer to *SAS Macro Language: Reference.*

(*continued*)

Table 4.1 Elements of a macro program definition (*continued*)

</option(s)>	**SOURCE** **SRC**	Combines and stores the source of the compiled macro program with the compiled macro program code as an entry in a SAS catalog in a permanent SAS data library. The SOURCE option requires that the STORE option and the MSTORED option be set. See Chapter 10 and *SAS Macro Language: Reference* for more information.
	STORE	Stores the compiled macro program as an entry in a SAS catalog in a permanent SAS data library. See Chapter 10 and *SAS Macro Language: Reference* for more information
<text>	is any combination of • text strings • macro variables, macro functions, or macro language statements • SAS programming statements SAS programming statements are treated as text within the macro program definition.	
%MEND	Marks the end of the macro program. Including the macro program name on the %MEND statement is optional.	

Program 4.1 shows an example of a macro program definition. Macro program SALESCHART produces a horizontal bar chart analyzing profit for the current week by section of the bookstore.

Program 4.1

```
%macro saleschart;
  goptions reset=all;
  pattern1 c=graybb;
  goptions ftext=swiss rotate=landscape;

  title "Sales Report for Week Ending &sysdate9";
  proc gchart data=temp;
    where today()-6 <= datesold <= today();
    hbar section / sumvar=profit type=sum;
  run;
  quit;
%mend saleschart;
```

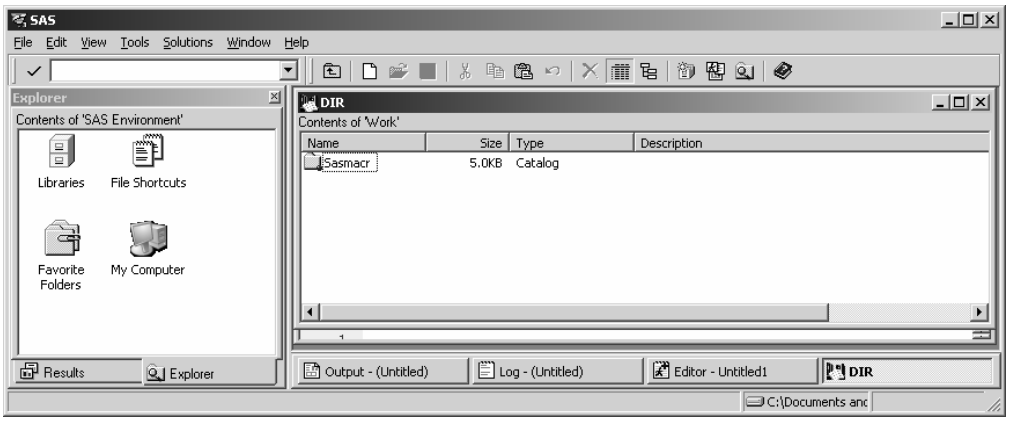
To compile this macro program for use later in your SAS session, submit the macro program definition from the Editor or from within the SAS program that calls it. The word scanner tokenizes the macro program and sends the tokens to the macro processor for compilation.

When the macro processor compiles the macro language statements in the macro program, it saves the results in a SAS catalog. By default, SAS stores macro programs in a catalog in the WORK library called SASMACR. Macro programs can also be saved in permanent catalogs and structures called autocall libraries. Chapter 10 discusses how to do this.

A compiled macro program can be reused within the same SAS session. A macro program has to be submitted only once in your SAS session. The compiled macro program remains in the SASMACR catalog throughout the SAS session. When the SAS session ends, SAS deletes the SASMACR catalog that contains the compiled macro program. Chapter 10 describes ways to store compiled macro program code.

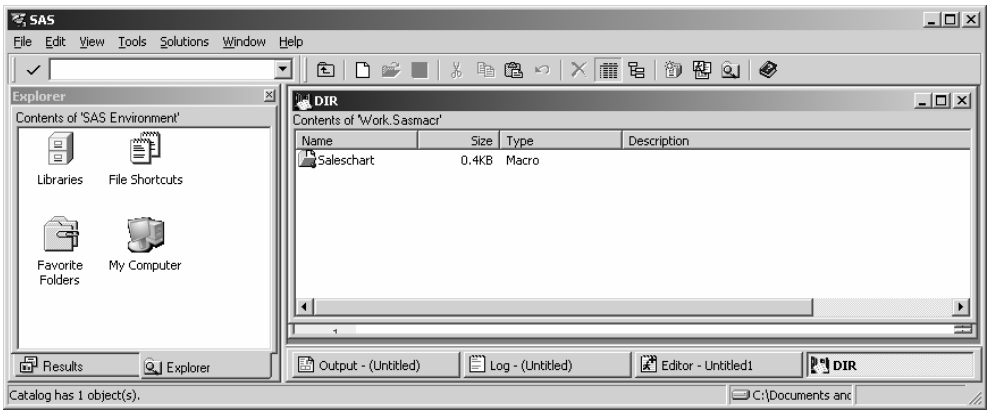
After submitting the preceding program from the Editor, the WORK directory looks like Display 4.1. Currently, one catalog, the SASMACR catalog, exists.

Display 4.1 The results of the DIR command



Opening the SASMACR catalog displays the window presented in Display 4.2. Note that the SALESCHART macro program is in this catalog. The entry in the catalog for SALESCHART is the compiled version of SALESCHART. The code for the compiled version cannot be accessed and viewed. (See Chapter 10 for ways to retrieve the code of stored compiled macro programs.)

Display 4.2 Contents of the WORK.SASMACR catalog



You can also list the entries in WORK.SASMACR catalog by submitting the following PROC CATALOG step in Program 4.2.

Program 4.2

```
proc catalog c=work.sasmacr;
  contents;
run;
quit;
```

Executing a Macro Program

A macro program is executed by submitting a reference to the macro program. To execute a macro program, submit the following statement from the Editor or from within your SAS program.

```
%program
```

where *program* is the name assigned to the macro program.

A reference to a macro program that has been successfully compiled can be placed anywhere in your SAS program except in data lines. This call to the macro program is preceded by a percent sign (%). The percent sign tells the word scanner to direct processing to the macro processor. The macro processor takes over and looks for the compiled program in the WORK.SASMACR catalog of session compiled macro programs. If found, the macro processor directs execution of the compiled macro program. If not found, an error message is written to the SAS log. Chapter 10 describes ways to tell SAS to look in other locations for compiled macro program code.

No semicolon follows the call to the macro program. The call to a macro program is not a SAS statement. Indeed, using a semicolon to terminate the call to the macro program might cause errors in the execution of your macro program.

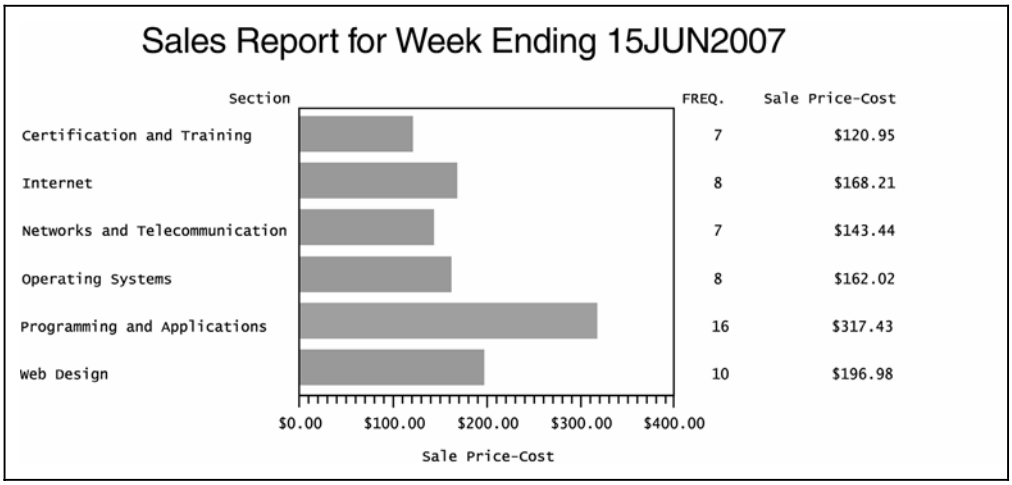
Program 4.3 calls the macro program defined in Program 4.1. Assume that the macro program in Program 4.1 was already submitted and its compiled code is in the SASMACR catalog. Assume the program was submitted on Friday, June 15, 2007. When Program 4.3 is submitted, the DATA step and the PROC GCHART step in macro program SALESCHART execute.

Program 4.3

```
data temp;
  set books.ytdsales;
  attrib profit label='Sale Price-Cost' format=dollar8.2;
  profit=saleprice-cost;
run;

%saleschart
```

Output 4.1 presents the output from Program 4.3.

Output 4.1 Output from Program 4.3 that contains a call to macro program SALESCHART defined in Program 4.1



Displaying Notes about Macro Program Compilation in the SAS Log

The SAS option MCOMPILENOTE writes notes to the SAS log about whether a macro program compiles successfully. The option can be set to one of three values: NONE, NOAUTOCALL, or ALL. The NONE value suppresses display of all macro program compilation notes. The NOAUTOCALL value suppresses compilation notes for autocall macro programs and displays all other types of macro program compilation notes. The ALL value causes display of all macro program compilation notes. Autocall macro programs are those stored in external files or SAS catalogs. Chapter 10 describes how to work with autocall macro programs.

The default setting for MCOMPILENOTE= is NONE. Assuming that is the case when executing Program 4.1, SAS writes no compilation notes to the SAS log. With Program 4.1 modified in Program 4.4a to include the MCOMPILENOTE= option set to ALL, SAS now writes compilation notes to the SAS log.

Program 4.4a

```
options mcompilenote=all;
%macro saleschart;
  goptions reset=all;
  pattern1 c=graybb;
  goptions ftext=swiss rotate=landscape;

  title "Sales Report for Week Ending &sysdate9";
  proc gchart data=temp;
    where today()-6 <= datesold <= today();
    hbar section / sumvar=profit type=sum;
  run;
  quit;
%mend saleschart;
```

The SAS log produced by Program 4.4a follows, and it indicates that SALESCHART was compiled with four instructions.

```
NOTE: The macro SALESCHART completed compilation without
      errors.
      4 instructions 292 bytes.
```

Program 4.4b shows the type of note that SAS writes when there are compilation errors in a macro program. Program 4.4b adds an %IF statement to Program 4.4a, but omits the required percent sign on the %END statement.

Program 4.4b

```
options mcompilenote=all;
%macro saleschart;
  goptions reset=all;
  pattern1 c=graybb;
  goptions ftext=swiss rotate=landscape;

  %if &sysday=Friday %then %do;
    title "Sales Report for Week Ending &sysdate9";
    proc gchart data=temp;
      where today()-6 <= datesold <= today();
      hbar section / sumvar=profit type=sum;
    run;
    quit;
  end;
%mend saleschart;
```

The SAS log produced by Program 4.4b follows. The ERROR message indicates that SALESCHART did not compile. The last part of the NOTE indicates that no instructions were saved.

```
ERROR: There were 1 unclosed %DO statements.  The macro
       SALESCHART will not be compiled.
NOTE: The macro SALESCHART completed compilation without
      errors.
      0 instructions 0 bytes.
```

Displaying Messages about Macro Program Processing in the SAS Log

SAS options MPRINT and MLOGIC write to the SAS log information about the processing of macro programs. These options assist you in debugging and reviewing your macro programs. The SYMBOLGEN option described earlier displays information about macro variables. SYMBOLGEN displays information about macro variables that are created in open code or inside macro programs.

The SAS log for Program 4.3 follows. The SYMBOLGEN, MPRINT, and MLOGIC options are turned off. The two notes following the call to SALESCHART are all the processing information we have about the SAS language statements submitted by the macro program SALESCHART.

```
517  options nosymbolgen nomprint nomlogic;
518  data temp;
519    set books.ytdsales;
520    attrib profit label='Sale Price-Cost' format=dollar8.2;
521    profit=saleprice-cost;
522  run;

NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
NOTE: The data set WORK.TEMP has 6096 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

523
524  %saleschart

NOTE: There were 56 observations read from the data set
```

```
      WORK.TEMP.
      WHERE (datesold>=(TODAY()-6)) and
      (datesold<=TODAY());
NOTE: PROCEDURE GCHART used (Total process time):
      real time           0.12 seconds
      cpu time            0.12 seconds
```

Using MPRINT to Display the SAS Statements Submitted by a Macro Program

When you submit a SAS program, SAS normally writes SAS code and processing messages about the compilation and execution of the SAS language statements to the SAS log. By default, SAS language statements submitted from within a macro program are not written to the SAS log. If you want to see the SAS code that the macro processor constructs and submits, enable the MPRINT option.

The SAS log that follows is for Program 4.3 with the MPRINT option enabled. Now you can verify the SAS language statements that have been constructed by the macro program.

```
525  options nosymbolgen mprint nomlogic;
526  data temp;
527    set books.ytdsales;
528    attrib profit label='Sale Price-Cost' format=dollar8.2;
529    profit=saleprice-cost;
530  run;

NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
NOTE: The data set WORK.TEMP has 6096 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

531
532 %saleschart
MPRINT(SALESCHART):   goptions reset=all;
MPRINT(SALESCHART):   pattern1 c=graybb;
MPRINT(SALESCHART):   goptions ftext=swiss rotate=landscape;
MPRINT(SALESCHART):   title "Sales Report for Week Ending
15JUN2007";
MPRINT(SALESCHART):   proc gchart data=temp;
MPRINT(SALESCHART):   where today()-6 <= datesold <=
today();
MPRINT(SALESCHART):   hbar section / sumvar=profit type=sum;
MPRINT(SALESCHART):   run;
```

```
MPRINT(SALESCHART):   quit;
NOTE: There were 56 observations read from the data set
      WORK.TEMP.
      WHERE (datesold>=(TODAY()-6)) and
      (datesold<=TODAY());
NOTE: PROCEDURE GCHART used (Total process time):
      real time           0.11 seconds
      cpu time            0.10 seconds
```

Using the MLOGIC Option to Trace Execution of a Macro Program

The MLOGIC option traces the execution of macro programs. The information written to the SAS log when MLOGIC is enabled includes the beginning and ending of the macro program and the results of arithmetic and logical macro language operations. The MLOGIC option is useful for debugging macro language statements in macro programs. The SAS log of Program 4.3 with MLOGIC enabled follows. Examples in Chapter 7 that deal with macro programming statements further illustrate the usefulness of this option.

```
573  options nosymbolgen nomprint mlogic;
574  data temp;
575    set books.ytdsales;
576    attrib profit label='Sale Price-Cost' format=dollar8.2;
577    profit=saleprice-cost;
578  run;

NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
NOTE: The data set WORK.TEMP has 6096 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

MLOGIC(SALES):  Beginning execution.
579
580  %saleschart

NOTE: There were 56 observations read from the data set
      WORK.TEMP.
      WHERE (datesold>=(TODAY()-6)) and (datesold<=TODAY());
```

```
NOTE: PROCEDURE GCHART used (Total process time):
      real time           0.13 seconds
      cpu time            0.10 seconds

MLOGIC(SALES):  Ending execution.
```

Passing Values to a Macro Program through Macro Parameters

Macro program parameters expand the reusability and flexibility of your macro programs by allowing you to initialize macro variables inside your macro programs. When you use parameters, macro program code does not have to be modified each time you want the macro variables to start out with different values. Think of macro programs with parameters as similar to subroutines in other programming languages.

Macro parameter names are specified on the %MACRO statement. The names assigned to the parameters must be the same as the names of the macro variables that you want to reference inside the macro program.

The initial values of the parameters are specified on the call to the macro program. When the macro program starts, the corresponding macro variables are initialized with the values of the parameters.

The two types of macro program parameters, positional and keyword, are described in the next two sections. The third section shows how to combine positional and keyword parameters in one macro program definition. The last section defines a macro program with the PARMBUFF option. This option provides the capability to define a macro program that accepts a varying number of parameters at each invocation.

Specifying Positional Parameters in Macro Programs

Positional macro program parameters define a one-to-one correspondence between the list of parameters on the %MACRO statement and the values of the parameters on the macro program call. The following is the general format of a macro program definition containing positional parameters.

```
%macro program(positional-1, positional-2, ...,positional-n);

   macro program referencing the macro variables in the
   positional parameter list

%mend <program>;
```

Positional parameters are enclosed in parentheses and are separated with commas. There is no limit to the number of positional parameters that can be defined. However, too many positional parameters can make it unwieldy to write the call to the macro program.

When you call a macro program that uses positional parameters, you must specify the same number of values in the macro program call as the number of parameters listed on the %MACRO statement. Valid values include null values and text. If you want to assign a positional parameter a null value and you want to assign values to subsequent positional parameters, use a comma as a placeholder.

The general format of a call to a macro program that uses positional parameters is

```
%program(value-1, value-2, ..., value-n)
```

Example 4.1: Defining a Macro Program with Positional Parameters

Program 4.5 defines the macro program LISTPARM. This macro program defines three positional parameters, OPTS, START, and STOP, and calls LISTPARM twice. It computes specific statistics with PROC MEANS on SALEPRICE by SECTION for a specific time period in the BOOKS.YTDSALES data set. The parameter value for OPTS specifies statistics that PROC MEANS computes and other options valid on the PROC MEANS statement. The parameter values specified for START and STOP define the reporting time period.

Program 4.5

```
options mprint mlogic;

%macro listparm(opts,start,stop);
  title "Books Sold by Section Between &start and &stop";
  proc means data=books.ytdsales &opts;
    where "&start"d le datesold le "&stop"d;
    class section;
    var saleprice;
  run;
%mend listparm;
```

```
*----First call to LISTPARM, all 3 parameters specified;
%listparm(n sum,01JUN2007,15JUN2007)
*----Second call to LISTPARM, first parameter is null,;
*----second and third parameters specified;
%listparm(,01SEP2007,15SEP2007)
```

The first call to LISTPARM specifies values for each of the three parameters, and commas separate the parameters. The first parameter specifies two options, N and SUM, for the PROC MEANS step.

Note that the two options comprising the first parameter are separated by a space. If you separated them with a comma, the %LISTPARM macro program call would not execute, and you would receive an error message that more positional parameters (4) were found than defined (3). See Chapter 8 for ways to specify parameters that contain commas and other special characters.

The first parameter in the second call to LISTPARM is null. Commas separate the parameters. Nothing precedes the first comma, so OPTS is null, and no options are added to the PROC MEANS statement. Therefore, PROC MEANS computes default statistics and uses default options.

Note that the MLOGIC option displays the parameter values at the start of macro program execution.

The SAS log for the first call to LISTPARM follows.

```
30   *----First call to LISTPARM, all 3 parameters specified;
31   %listparm(n sum,01JUN2007,15JUN2007)
MLOGIC(LISTPARM):  Beginning execution.
MLOGIC(LISTPARM):  Parameter OPTS has value n sum
MLOGIC(LISTPARM):  Parameter START has value 01JUN2007
MLOGIC(LISTPARM):  Parameter STOP has value 15JUN2007
MPRINT(LISTPARM):   title "Books Sold by Section Between
01JUN2007 and 15JUN2007";
MPRINT(LISTPARM):   proc means data=books.ytdsales n sum;
MPRINT(LISTPARM):   where "01JUN2007"d le datesold le
"15JUN2007"d;
MPRINT(LISTPARM):   class section;
MPRINT(LISTPARM):   var saleprice;
MPRINT(LISTPARM):   run;

NOTE: There were 105 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01JUN2007'D and datesold<='15JUN2007'D);
```

```
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MLOGIC(LISTPARM):  Ending execution.
```

The SAS log for the second call to LISTPARM follows.

```
32   *----Second call to LISTPARM, first parameter is null,;
33   *----second and third parameters specified;
34   %listparm(,01SEP2007,15SEP2007)
MLOGIC(LISTPARM):  Beginning execution.
MLOGIC(LISTPARM):  Parameter OPTS has value
MLOGIC(LISTPARM):  Parameter START has value 01SEP2007
MLOGIC(LISTPARM):  Parameter STOP has value 15SEP2007
MPRINT(LISTPARM):   title "Books Sold by Section Between
01SEP2007 and 15SEP2007";
MPRINT(LISTPARM):   proc means data=books.ytdsales ;
MPRINT(LISTPARM):   where "01SEP2007"d le datesold le
"15SEP2007"d;
MPRINT(LISTPARM):   class section;
MPRINT(LISTPARM):   var saleprice;
MPRINT(LISTPARM):   run;

NOTE: There were 318 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01SEP2007'D and datesold<='15SEP2007'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MLOGIC(LISTPARM):  Ending execution.
```

Specifying Keyword Parameters in Macro Programs

A call to a macro program that has been defined with keyword parameters contains both the parameter names and the initial parameter values. On the %MACRO statement defining the macro program, an equal sign (=) follows each parameter name.

A macro program with many parameters is easier to define and use with keyword parameters rather than positional parameters. With keyword parameters, you do not have to keep track of the positions of the parameters when writing the call to the macro program.

Another advantage of using keyword parameters is that you can specify default values for the keyword parameters when you define the macro program. Then, when you want

to use a default value, you can omit the keyword parameter completely from the macro program call.

Keyword parameters can be specified in any order. A one-to-one correspondence between the parameters on the %MACRO statement and the values on the call to the macro program is not required as it is with positional parameters. The general format of a macro program definition containing keyword macro program parameters is

```
%macro program(keyword1=value, keyword2=value, ...,
keywordn=value);

   macro program referencing the macro variables in the keyword
   parameter list

%mend <program>;
```

Keyword parameter lists are enclosed in parentheses, and keyword references are separated with commas. In the preceding format, *keyword1, keyword2*, and so on represent the names of the parameters and the corresponding macro variables in the macro program.

The values following the equal signs are the default values passed to the macro program. It is not necessary to specify default values for keyword parameters. However, if a default value has been defined and you want to call the macro program and use that default value, you do not have to specify the corresponding keyword parameter in the macro program call. There is no limit to the number of keyword parameters that can be defined.

The general format of a call to a macro program that uses keyword parameters is:

```
%program(keyword1=value1,keyword2=value2,...,keywordn=valuen)
```

Valid keyword parameter values include null values and text. In a macro program call, no value after the equal sign of a parameter initializes the macro variable with a null value.

Example 4.2: Defining a Macro Program with Keyword Parameters

Program 4.6 defines the macro program KEYPARM and calls it three times. Program 4.6 does the same task as Program 4.5. The difference is that the macro program in Program 4.6 is defined with keyword parameters while macro program LISTPARM in Program 4.5 is defined with positional parameters. Macro program KEYPARM defines three keyword parameters: OPTS, START, and STOP.

The first call specifies all three keyword parameters. The second call specifies a null value for a keyword parameter value. The third call demonstrates how a macro program defined with keyword parameters that are assigned default values processes.

Program 4.6

```
options mprint mlogic;

%macro keyparm(opts=N SUM MIN MAX,
               start=01JAN2007,stop=31DEC2007);
  title "Books Sold by Section Between &start and &stop";
  proc means data=books.ytdsales &opts;
    where "&start"d le datesold le "&stop"d;
    class section;
    var saleprice;
  run;
%mend keyparm;

*----First call to KEYPARM: specify all keyword parameters;
%keyparm(opts=n sum,start=01JUN2007,stop=15JUN2007)

*----Second call to KEYPARM: specify start and stop,;
*----opts is null: should see default stats for PROC MEANS;
%keyparm(opts=,start=01SEP2007,stop=15SEP2007)

*----Third call to KEYPARM: use defaults for start and stop,;
*----specify opts;
%keyparm(opts=n sum)
```

The first call to the KEYPARM macro program specifies values for all three keyword parameters. The SAS log for the first call to KEYPARM follows.

```
45   *----First call to KEYPARM: specify all keyword parameters;
46   %keyparm(opts=n sum,start=01JUN2007,stop=15JUN2007)
MLOGIC(KEYPARM):  Beginning execution.
MLOGIC(KEYPARM):  Parameter OPTS has value n sum
MLOGIC(KEYPARM):  Parameter START has value 01JUN2007
MLOGIC(KEYPARM):  Parameter STOP has value 15JUN2007
MPRINT(KEYPARM):   title "Books Sold by Section Between
01JUN2007 and 15JUN2007";
MPRINT(KEYPARM):   proc means data=books.ytdsales n sum;
MPRINT(KEYPARM):   where "01JUN2007"d le datesold le
"15JUN2007"d;
MPRINT(KEYPARM):   class section;
MPRINT(KEYPARM):   var saleprice;
MPRINT(KEYPARM):   run;
```

```
NOTE: There were 105 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01JUN2007'D and datesold<='15JUN2007'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MLOGIC(KEYPARM):  Ending execution.
```

The second call to KEYPARM specifies a null value for the OPTS parameter. That means that the null value replaces the default value. The result is that the statistics keywords—N, SUM, MIN, and MAX—are not on the PROC MEANS statement. Instead, PROC MEANS computes its set of default statistics and uses its set of default options. The SAS log for the second call to KEYPARM follows.

```
48   *----Second call to KEYPARM: specify start and stop,;
49   *----opts is null: should see default stats for PROC MEANS;
50   %keyparm(opts=,start=01SEP2007,stop=15SEP2007)
MLOGIC(KEYPARM):  Beginning execution.
MLOGIC(KEYPARM):  Parameter OPTS has value
MLOGIC(KEYPARM):  Parameter START has value 01SEP2007
MLOGIC(KEYPARM):  Parameter STOP has value 15SEP2007
MPRINT(KEYPARM):   title "Books Sold by Section Between
01SEP2007 and 15SEP2007";
MPRINT(KEYPARM):   proc means data=books.ytdsales ;
MPRINT(KEYPARM):   where "01SEP2007"d le datesold le
"15SEP2007"d;
MPRINT(KEYPARM):   class section;
MPRINT(KEYPARM):   var saleprice;
MPRINT(KEYPARM):   run;

NOTE: There were 318 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01SEP2007'D and datesold<='15SEP2007'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MLOGIC(KEYPARM):  Ending execution.
```

The third call to KEYPARM specifies only one parameter, OPTS. The parameter value for OPTS requests two statistics, N and SUM. The call omits values for the START and STOP keyword parameters. Therefore, PROC MEANS computes the N and SUM statistics on observations that fall between the default dates specified for START and STOP, which are January 1, 2007, and December 31, 2007. The SAS log for the third call to KEYPARM follows.

```
52   *----Third call to KEYPARM: use defaults for start and
stop,;
53   *----specify opts;
54   %keyparm(opts=n sum)
MLOGIC(KEYPARM):  Beginning execution.
MLOGIC(KEYPARM):  Parameter OPTS has value n sum
MLOGIC(KEYPARM):  Parameter START has value 01JAN2007
MLOGIC(KEYPARM):  Parameter STOP has value 31DEC2007
MPRINT(KEYPARM):   title "Books Sold by Section Between
01JAN2007 and 31DEC2007";
MPRINT(KEYPARM):   proc means data=books.ytdsales n sum;
MPRINT(KEYPARM):   where "01JAN2007"d le datesold le
"31DEC2007"d;
MPRINT(KEYPARM):   class section;
MPRINT(KEYPARM):   var saleprice;
MPRINT(KEYPARM):   run;

NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2007'D and datesold<='31DEC2007'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MLOGIC(KEYPARM):  Ending execution.
```

Specifying Mixed Parameter Lists in Macro Programs

Positional parameters and keyword parameters can both be defined for the same macro program. Positional parameters must be placed ahead of keyword parameters in the definition and in the call to the macro program. Otherwise, the same rules for defining and using each type of parameter apply.

The general format of a macro program definition containing positional parameters and keyword parameters is

```
%macro program(positional-1, positional-2, ...,positional-n,
    keyword1=value,keyword2=value, ...,  keywordm=value);

   macro program referencing both kinds of parameters

%mend <program>;
```

The general format of a call to a macro program that contains positional parameters and keyword parameters is

```
%program(positionalvalue-1, positionalvalue-2, ...,
         positionalvalue-n,
         keyword1=value, keyword2=value, ...,  keywordm=value)
```

Example 4.3: Defining a Macro Program with Positional Parameters and Keyword Parameters

Program 4.7 defines the macro program, MIXDPARM, with two positional parameters and two keyword parameters and calls that macro program twice. It does the same task as Programs 4.5 and 4.6.

Macro program MIXDPARM specifies a PROC MEANS step and its two positional parameters, STATS and OTHROPTS, specify PROC MEANS statistics and options. The two keyword parameters, START and STOP, specify a date range for the calculations.

The first call to MIXDPARM specifies a null value for the STATS positional parameter, which causes calculation of the default PROC MEANS statistics. The call assigns the value MISSING to parameter OTHROPTS. The MISSING option on the PROC MEANS statement requests that PROC MEANS include missing values as valid values in creating combinations of the class variables. The call sets the START keyword parameter to December 1, 2007, and it does not assign a value to the STOP keyword parameter. Without a value specified for STOP, the default value of December 31, 2007, specified in the macro program definitions, is used as the stop date. The first call to MIXDPARM analyzes information for December 2007.

The second call to MIXDPARM specifies no values for either the positional or the keyword parameters. With parameters STAT and OTHROPTS both null, PROC MEANS calculates default statistics. Since the second call specifies no values for START and STOP, the PROC MEANS step calculates statistics for all of 2007.

In the second call, note that a comma does not separate the null values for positional parameters STATS and OTHROPTS. If a value were specified for OTHROPTS, a comma placeholder preceding the OTHROPTS parameter value would be needed.

Program 4.7

```
options mprint mlogic;

%macro mixdparm(stats,othropts,
                start=01JAN2007,stop=31DEC2007);
  title "Books Sold by Section Between &start and &stop";
  proc means data=books.ytdsales &stats &othropts;
    where "&start"d le datesold le "&stop"d;
    class section;
    var saleprice;
  run;
%mend mixdparm;

%*----Compute default stats for December 2007 and allow
      missing values to be valid in creating combinations of
      the CLASS variables;
%mixdparm(,missing,start=01DEC2007)

%*----Compute default stats for all of 2007;
%mixdparm()
```

The SAS log for the first call to MIXDPARM follows.

```
343  %mixdparm(,missing,start=01DEC2007)
MLOGIC(MIXDPARM):  Beginning execution.
MLOGIC(MIXDPARM):  Parameter STATS has value
MLOGIC(MIXDPARM):  Parameter OTHROPTS has value missing
MLOGIC(MIXDPARM):  Parameter START has value 01DEC2007
MLOGIC(MIXDPARM):  Parameter STOP has value 31DEC2007
MPRINT(MIXDPARM):   title "Books Sold by Section Between
01DEC2007 and 31DEC2007";
MPRINT(MIXDPARM):   proc means data=books.ytdsales missing;
MPRINT(MIXDPARM):   where "01DEC2007"d le datesold le
"31DEC2007"d;
MPRINT(MIXDPARM):   class section;
MPRINT(MIXDPARM):   var saleprice;
MPRINT(MIXDPARM):   run;

NOTE: There were 356 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01DEC2007'D and datesold<='31DEC2007'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
MLOGIC(MIXDPARM):  Ending execution.
```

The SAS log for the second call to MIXDPARM follows.

```
344  %mixdparm()
MLOGIC(MIXDPARM):  Beginning execution.
MLOGIC(MIXDPARM):  Parameter STATS has value
MLOGIC(MIXDPARM):  Parameter OTHROPTS has value
MLOGIC(MIXDPARM):  Parameter START has value 01JAN2007
MLOGIC(MIXDPARM):  Parameter STOP has value 31DEC2007
MPRINT(MIXDPARM):   title "Books Sold by Section Between
01JAN2007 and 31DEC2007";
MPRINT(MIXDPARM):   proc means data=books.ytdsales ;
MPRINT(MIXDPARM):   where "01JAN2007"d le datesold le
"31DEC2007"d;
MPRINT(MIXDPARM):   class section;
MPRINT(MIXDPARM):   var saleprice;
MPRINT(MIXDPARM):   run;

NOTE: There were 6096 observations read from the data set
BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2007'D and datesold<='31DEC2007'D);
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
MLOGIC(MIXDPARM):  Ending execution.
```

Defining a Macro Program That Can Accept a Varying Number of Parameter Values

When your application requires that you execute a macro program multiple times by simply changing one parameter value each time, you might find it useful instead to define your macro program with the PARMBUFF option. With PARMBUFF, you could submit your macro program only once by specifying the complete list of values that you want processed on that single call to your macro program. Additionally with PARMBUFF, you can specify a varying number of parameter values each time you execute your macro program.

The PARMBUFF option assigns the entire list of parameter values in your macro program call to the automatic macro variable SYSPBUFF. Your macro program can parse the list extracting each value and directing that specific steps be executed for each value in the list.

The general format of a macro program definition containing the PARMBUFF option is:

```
%macro program(<positional-1,positional-2,...,positional-n)> /
           PARMBUFF;

   macro program referencing a varying number of parameter values

%mend <program>;
```

A typical simple application of the PARMBUFF option is to define the macro program with no parameters. However, you can define positional and keyword parameters as well with the PARMBUFF option. These defined parameters receive values, and the values are also assigned to automatic variable SYSPBUFF. Typically in this situation, because you could be dealing with a varying number of parameter values, you would place your defined parameters at the beginning of the macro program call while the list of varying values would be at the end. When parsing values in SYSPBUFF, your code would have to account for the positional parameter values before processing your list of varying values. As with the definition of macro programs that have both positional and keyword parameters, make sure you place positional parameters ahead of the keyword parameters.

The general format of a call to a macro program defined with the PARMBUFF option is

```
%program(parmvaluevalue-1, parmvalue-2, ...,
         parmvalue-n)
```

Example 4.4: Defining a Macro Program with the PARMBUFF Option

Program 4.8 defines macro program PBUFFPARMS with the PARMBUFF option. PBUFFPARMS produces a bar chart of total sales by bookstore section. The program is written to process as its parameter values a list of months. For each month specified in the value list, the macro program submits a PROC GCHART step that displays monthly sales by section. When no parameter values are specified, PBUFFPARMS submits a summary PROC GCHART step that computes total sales by section with subgroups in the bars defined by the quarter of the sale.

This example uses macro functions and programming statements that are more fully described in later chapters. Chapter 6 describes macro functions, and Chapter 7 describes macro programming statements.

Briefly, the %SCAN function extracts words from its argument where the words in the argument are separated by delimiters. The %IF-%THEN-%ELSE and %DO-%UNTIL statements process blocks of code. The combination of %SCAN and %DO-%UNTIL provides you the tools to process each of the values in the list of parameter values passed to your macro program.

The macro processor stores the list of parameter values in automatic macro variable SYSPBUFF. Macro program PBUFFPARMS processes the contents of SYSPBUFF. References to SYSPBUFF are in bold in Program 4.8.

Program 4.8 submits two calls to PBUFFPARMS. The first call specifies two parameter values, 8 and 11, which produces one monthly sales chart for August and one for November. The second call to PBUFFPARMS specifies no parameter values. This causes PBUFFPARMS to submit the summary PROC GCHART step.

The value assigned to SYSPBUFF by the first call is:

```
(8,11)
```

The value assigned to SYSPBUFF by the second call is:

```
()
```

Note that parentheses are included in the value assigned to SYSPBUFF.

Example 7.7 presents another example of a macro program defined with the PARMBUFF option.

Program 4.8

```
%macro pbuffparms / parmbuff;
  goptions reset=all;
  pattern1 c=graybb;
  pattern2 c=graydd;
  pattern3 c=white;
  pattern4 c=grayaa;
  goptions ftext=swiss rotate=landscape;

  %*----Process this section when parameter values specified;
  %if &syspbuff ne %then %do;
    %let i=1;
    %let month=%scan(&syspbuff,&i);
    %do %while(&month ne);
      proc gchart data=books.ytdsales
                    (where=(month(datesold)=&month));
        title "Sales Report for Month &month";
        hbar section / sumvar=saleprice type=sum;
      run;
      quit;
      %let i=%eval(&i+1);
      %let month=%scan(&syspbuff,&i);
   %end;
  %end;
```

```
  %*----Process this section when no parameter values
        specified;
  %else %do;
    proc gchart data=books.ytdsales;
      title "Annual Sales by Quarter";
      hbar section / sumvar=saleprice type=sum
                     subgroup=datesold coutline=black;
      format datesold qtr.;
    run;
    quit;
  %end;
%mend pbuffparms;

*----Analyze sales for August and November;
%pbuffparms(8,11)

*----Analyze sales for entire year;
%pbuffparms()
```

The SAS log for Program 4.8 with the MPRINT option enabled follows. This shows that three PROC GCHART steps were submitted, two from the first call to PBUFFPARMS and one from the second call to PBUFFPARMS.

```
110  *----Analyze sales for August and November;
111  %pbuffparms(8,11)
MPRINT(PBUFFPARMS):   goptions reset=all;
MPRINT(PBUFFPARMS):   pattern1 c=graybb;
MPRINT(PBUFFPARMS):   pattern2 c=graydd;
MPRINT(PBUFFPARMS):   pattern3 c=white;
MPRINT(PBUFFPARMS):   pattern4 c=grayaa;
MPRINT(PBUFFPARMS):   goptions ftext=swiss rotate=landscape;

MPRINT(PBUFFPARMS):   proc gchart
data=books.ytdsales(where=(month(datesold)=8));
MPRINT(PBUFFPARMS):   title "Sales Report for Month 8";
MPRINT(PBUFFPARMS):   hbar section / sumvar=saleprice type=sum;
MPRINT(PBUFFPARMS):   run;
MPRINT(PBUFFPARMS):   quit;
NOTE: There were 500 observations read from the data set
      BOOKS.YTDSALES.
      WHERE MONTH(datesold)=8;
NOTE: PROCEDURE GCHART used (Total process time):
      real time           0.15 seconds
      cpu time            0.17 seconds

MPRINT(PBUFFPARMS):   proc gchart
data=books.ytdsales(where=(month(datesold)=11));
```

```
MPRINT(PBUFFPARMS):   title "Sales Report for Month 11";
MPRINT(PBUFFPARMS):   hbar section / sumvar=saleprice type=sum;
MPRINT(PBUFFPARMS):   run;
MPRINT(PBUFFPARMS):   quit;

NOTE: There were 649 observations read from the data set
      BOOKS.YTDSALES.
      WHERE MONTH(datesold)=11;
NOTE: PROCEDURE GCHART used (Total process time):
      real time           0.03 seconds
      cpu time            0.04 seconds
112
113  *----Analyze sales for entire year;
114  %pbuffparms()
MPRINT(PBUFFPARMS):   goptions reset=all;
MPRINT(PBUFFPARMS):   pattern1 c=graybb;
MPRINT(PBUFFPARMS):   pattern2 c=graydd;
MPRINT(PBUFFPARMS):   pattern3 c=white;
MPRINT(PBUFFPARMS):   pattern4 c=grayaa;
MPRINT(PBUFFPARMS):   goptions ftext=swiss rotate=landscape;

MPRINT(PBUFFPARMS):   proc gchart data=books.ytdsales;
MPRINT(PBUFFPARMS):   title "Annual Sales by Quarter";
MPRINT(PBUFFPARMS):   hbar section / sumvar=saleprice type=sum
subgroup=datesold coutline=black;
MPRINT(PBUFFPARMS):   format datesold qtr.;
MPRINT(PBUFFPARMS):   run;
MPRINT(PBUFFPARMS):   quit;
NOTE: There were 6096 observations read from the data set
BOOKS.YTDSALES.
NOTE: PROCEDURE GCHART used (Total process time):
      real time           0.06 seconds
      cpu time            0.06 seconds
```

Chapter 5

Understanding Macro Symbol Tables and the Processing of Macro Programs

Introduction 102

Understanding Macro Symbol Tables 102

- Understanding the Global Macro Symbol Table 104
- Understanding Local Macro Symbol Tables 108
- Working with Global Macro Variables and Local Macro Variables 116
- Defining the Domain of a Macro Variable by Using the %GLOBAL or %LOCAL Macro Language Statements 118

Processing of Macro Programs 122

- How a Macro Program Is Compiled 122
- How a Macro Program Is Executed 127

Introduction

As your macro programming applications become more complex, an understanding of the technical aspects of macro processing becomes more important. This knowledge will likely speed up development and debugging of your programs.

The discussion of the technical aspects of macro processing that began in Chapter 2 is continued in this chapter. This chapter describes symbol tables, both global and those created for macro programs. It illustrates how macro programs are processed and how macro programs access symbol tables.

Understanding Macro Symbol Tables

There are two types of macro symbol tables: global and local. The *global macro symbol table*, as the name implies, stores macro variables that can be referenced throughout your SAS session, both from open code and from within macro programs. A *local macro symbol table* stores macro variables defined within a macro program. References to local macro variables can be resolved only from within the macro program that defined them.

At the start of a SAS session, the macro processor creates the global macro symbol table to store the values of automatic macro variables. The global macro symbol table also stores the values of the macro variables that you create in open code or that you explicitly define as global in your macro program.

A local macro symbol table is created by executing a macro program that contains macro variables. By default, macro variables defined in the macro program are stored in the local macro symbol table associated with the macro program. (There are special cases where this is not true, however, such as creating a macro variable using CALL SYMPUT or CALL SYMPUTX, which is described in Chapter 9.) These local macro variables by default can be referenced only from within the macro program. A local macro symbol table and all its macro variables are deleted when the macro program that is associated with it ends.

As your SAS programming assistant, the macro processor keeps track of the domain of each macro variable that is defined in your SAS program. The context in which you reference a macro variable directs the macro processor's search for the macro variable value when called upon to resolve a macro variable.

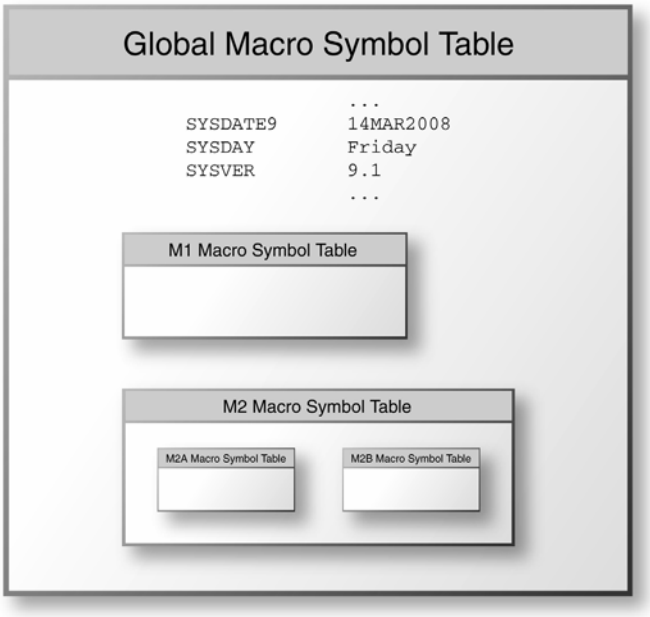
Figure 5.1 presents an example of how the domains of the macro symbol tables are defined by default. Two macro programs, M1 and M2, are invoked. In addition, two more macro programs, M2A and M2B, are invoked from within macro program M2 when macro program M2 is executing.

Observations to make from Figure 5.1 include:

- Macro variables in the global macro symbol table can be referenced in open code and from within M1, M2, M2A, and M2B.
- The macro variables that are created by M1 are available only to M1.
- The macro variables that are created by M2 can be referenced by M2A and M2B.
- The macro variables that are created by M2A are available only to M2A.
- The macro variables that are created by M2B are available only to M2B.

As your macro programs become more complex and call one another, you might find it useful to diagram their relationships as in Figure 5.1.

Figure 5.1 Example of macro symbol tables in a SAS session



Understanding the Global Macro Symbol Table

The macro processor creates the global macro symbol table at the start of a SAS session. The first macro variables that are placed in the global macro symbol table are the automatic macro variables that SAS defines. For example, the automatic macro variables SYSDATE9, SYSDAY, and SYSVER, as shown in Figure 5.1, are stored in the global macro symbol table.

User-defined macro variables can also be added to the global macro symbol table. The user-defined macro variables created in Chapter 3 were all stored in the global macro symbol table. There are three ways that you can add macro variables to the global macro symbol table:

- Create the macro variable in open code.
- List the macro variable on a %GLOBAL statement in the macro program in which it is defined.
- Create the macro variable in a DATA step with either of the SAS language routines, CALL SYMPUT or CALL SYMPUTX. (These routines provide an interface between the SAS DATA step and the macro processor, and their usages are described in Chapter 9.)

Global macro variables can be referenced throughout the SAS session in which they are created. They can be referenced from open code and from inside macro programs. You can modify the values of user-defined global macro variables throughout your SAS session. As described in Chapter 3, you can modify a few of the automatic macro variables as well.

Program 5.1 creates the macro variable SUBSET in open code and references it both from open code and from within macro program MAKEDS. The domain of the macro variable SUBSET is the global macro symbol table because it was created in open code. Therefore, the macro variable can be successfully referenced and resolved both from open code and from within a macro program.

Assume Program 5.1 was submitted on March 14, 2008.

Program 5.1

```
options symbolgen mprint;

%let subset=Internet;

%macro makeds;
  data temp;
    set books.ytdsales(where=(section="&subset"));
    attrib qtrsold label='Quarter of Sale';
    qtrsold=qtr(datesold);
  run;
%mend makeds;

%makeds

proc tabulate data=temp;
  title "Book Sales Report Produced &sysdate9";
  class qtrsold;
  var saleprice listprice;
  tables qtrsold all,
         (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
         box="Section: &subset";
  keylabel all='** Total **';
run;
```

Program 5.1 generates the following SAS log.

```
46   options symbolgen mprint;
47
48   %let subset=Internet;
49
50   %macro makeds;
51     data temp;
52       set books.ytdsales(where=(section="&subset"));
53       attrib qtrsold label='Quarter of Sale';
54       qtrsold=qtr(datesold);
55     run;
56   %mend makeds;
57
58   %makeds
MPRINT(MAKEDS):   data temp;
SYMBOLGEN:  Macro variable SUBSET resolves to Internet
MPRINT(MAKEDS):   set
books.ytdsales(where=(section="Internet"));
MPRINT(MAKEDS):   attrib qtrsold label='Quarter of Sale';
```

```
MPRINT(MAKEDS):   qtrsold=qtr(datesold);
MPRINT(MAKEDS):   run;

NOTE: There were 1456 observations read from the data set
BOOKS.YTDSALES.
      WHERE section='Internet';
NOTE: The data set WORK.TEMP has 1456 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           1.21 seconds
      cpu time            0.06 seconds

59
60   proc tabulate data=temp;
SYMBOLGEN:  Macro variable SYSDATE9 resolves to 14MAR2008
61     title "Book Sales Report Produced &sysdate9";
62     class qtrsold;
63     var saleprice listprice;
64     tables qtrsold all,
65            (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
66            box="Section: &subset";
SYMBOLGEN:  Macro variable SUBSET resolves to Internet
67     keylabel all='** Total **';
68   run;

NOTE: There were 1456 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.56 seconds
      cpu time            0.04 seconds
```

Output 5.1 presents the output for Program 5.1.

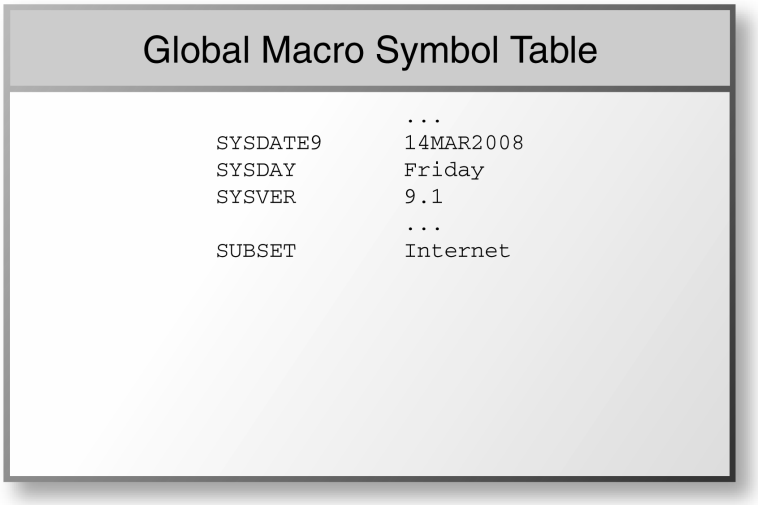
Output 5.1 Output for macro program MAKEDS in Program 5.1 using global macro variable SUBSET

Book Sales Report Produced 14MAR2008

Section: Internet	Sale Price		List Price	
	N	Sum	N	Sum
Quarter of Sale				
1	477	$20,168.41	477	$20,658.15
2	216	$9,397.20	216	$9,581.20
3	341	$14,711.59	341	$14,995.95
4	422	$18,018.59	422	$18,428.90
** Total **	1456	$62,295.78	1456	$63,664.20

Figure 5.2 shows a representation of the global macro symbol table that results when macro program MAKEDS in Program 5.1 executes. Since MAKEDS does not create any macro variables, the macro processor does not create a local macro symbol table for MAKEDS.

Figure 5.2 The global macro symbol table when MAKEDS in Program 5.1 executes



Understanding Local Macro Symbol Tables

If your macro program creates macro variables and does not specify them as global macro variables, the macro processor creates a local macro symbol table whenever that macro program executes. A local macro symbol table stores the values of the macro variables that the macro program creates. When the macro program finishes, the macro processor deletes the associated local macro symbol table.

It is important to understand the boundaries of the macro symbol tables. By default, the domain of macro variables that are created in a macro program is local to the macro program that defines them. For a given local macro variable, if no identically named macro variable is defined as global, a reference made in open code to that macro variable cannot be resolved.

A macro program can also be called from within another macro program. The macro program that is invoked from within another macro program can reference the macro variables created by the macro program that invoked it. Looking back at Figure 5.1, macro program M2 calls macro programs M2A and M2B. The local macro variables that M2 creates are available to M2A and M2B. Conversely, macro program M2 cannot resolve references to local macro variables created by M2A or M2B.

Therefore, you can have multiple macro variables, each with the same name, in one SAS session. Obviously, this can become confusing and is something to avoid, at least while you are learning how to write macro programs.

There are macro language functions that can assist in you determining domains of macro variables. These functions are described in the "Macro Variable Attribute Functions" section in Chapter 6.

Macro parameters are always local to the macro program that defines them. These macro variables are stored in the local macro symbol table associated with the macro program. You cannot make these macro variables global, but you can assign their values to global macro variables.

Program 5.1 in the previous section is revised below so that a parameter passes the value that defines the subset to MAKEDS. Now, macro variable SUBSET is stored in the local macro symbol table that is associated with the MAKEDS macro program. SUBSET is stored in the MAKEDS macro symbol table because it is defined as a parameter to MAKEDS. Assume that the macro variable SUBSET is not available globally. (A global macro variable can be deleted by macro language statement %SYMDEL, which is described in Chapter 7.)

Program 5.2

```
options symbolgen mprint;

%macro makeds(subset);
  data temp;
    set books.ytdsales(where=(section="&subset"));
    attrib qtrsold label='Quarter of Sale';
    qtrsold=qtr(datesold);
  run;
%mend makeds;

%makeds(Internet)

proc tabulate data=temp;
  title "Book Sales Report Produced &sysdate9";
  class qtrsold;
  var saleprice listprice;
  tables qtrsold all,
         (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
         box="Section: &subset";
  keylabel all='** Total **';
run;
```

The SAS log for the revised program follows.

```
124  options symbolgen mprint;
125
126  %macro makeds(subset);
127    data temp;
128      set books.ytdsales(where=(section="&subset"));
129      attrib qtrsold label='Quarter of Sale';
130      qtrsold=qtr(datesold);
131    run;
132  %mend makeds;
133
134  %makeds(Internet)
MPRINT(MAKEDS):   data temp;
SYMBOLGEN:  Macro variable SUBSET resolves to Internet
MPRINT(MAKEDS):   set
books.ytdsales(where=(section="Internet"));
MPRINT(MAKEDS):   attrib qtrsold label='Quarter of Sale';
MPRINT(MAKEDS):   qtrsold=qtr(datesold);
MPRINT(MAKEDS):   run;

NOTE: There were 1456 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Internet';
NOTE: The data set WORK.TEMP has 1456 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

135  proc tabulate data=temp;
SYMBOLGEN:  Macro variable SYSDATE9 resolves to 14MAR2008
136    title "Book Sales Report Produced &sysdate9";
137    class qtrsold;
138    var saleprice listprice;
139    tables qtrsold all,
140           (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
141            box="Section: &subset";
WARNING: Apparent symbolic reference SUBSET not resolved.
142    keylabel all='** Total **';
143  run;

NOTE: There were 1456 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds
```

Note the warning in the SAS log for the PROC TABULATE step. The macro processor cannot resolve the reference to macro variable SUBSET. This reference is made in open code, outside of the MAKEDS macro program. By the time PROC TABULATE executes, the MAKEDS macro program has ended and the MAKEDS symbol table has already been deleted. At that point, the macro processor searches only the global macro symbol table to resolve the SUBSET macro variable reference.

Output 5.2 presents the output from Program 5.2. This output shows that the macro processor did not resolve the macro variable reference in the box text.

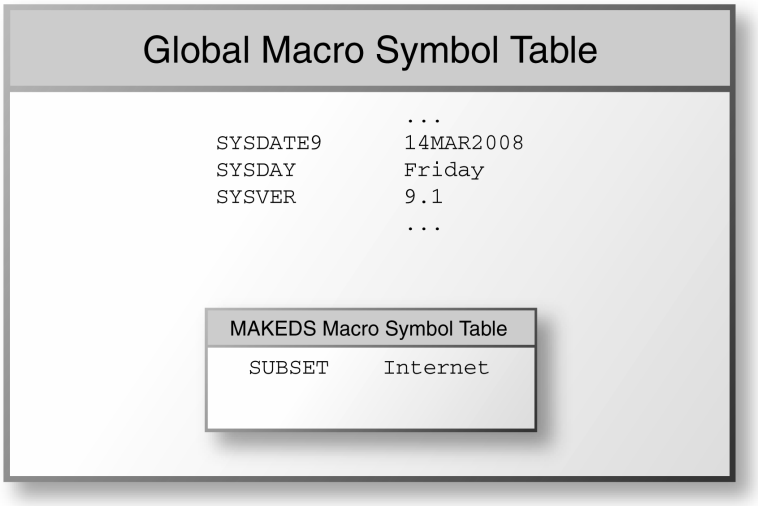
Output 5.2 Output for macro program MAKEDS in Program 5.2 using local macro variable SUBSET

Book Sales Report Produced 14MAR2008

Section: &subset	Sale Price		List Price	
	N	Sum	N	Sum
Quarter of Sale				
1	477	$20,168.41	477	$20,658.15
2	216	$9,397.20	216	$9,581.20
3	341	$14,711.59	341	$14,995.95
4	422	$18,018.59	422	$18,428.90
** Total **	1456	$62,295.78	1456	$63,664.20

Figure 5.3 shows a representation of the global macro symbol table and the local macro symbol table when the macro program executes.

Figure 5.3 The macro symbol tables during execution of the MAKEDS macro program in Program 5.2 when SUBSET is a local macro variable



Following are two ways to revise the program so that the reference to macro variable SUBSET in the PROC TABULATE step can be resolved. Program 5.3 presents the easier of the two. It places the PROC TABULATE step within the MAKEDS macro program. Macro variable SUBSET is local to macro program MAKEDS.

Program 5.3

```
options symbolgen mprint;

%macro makeds(subset);
  data temp;
    set books.ytdsales(where=(section="&subset"));
    attrib qtrsold label='Quarter of Sale';
    qtrsold=qtr(datesold);
  run;

  proc tabulate data=temp;
    title "Book Sales Report Produced &sysdate9";
    class qtrsold;
    var saleprice listprice;
```

```
    tables qtrsold all,
           (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
           box="Section: &subset";
    keylabel all='** Total **';
  run;
%mend makeds;

%makeds(Internet)
```

In the second version, Program 5.4, the %GLOBAL statement instructs the macro processor to create a macro variable, GLBSUBSET, and store it in the global macro symbol table. A %LET statement in the macro program assigns the value of the SUBSET macro variable to GLBSUBSET within the MAKEDS macro program. Now, the value of a local macro variable can be transferred to a global macro variable. The macro variable reference in PROC TABULATE is changed to the name of the global macro variable, GLBSUBSET.

Program 5.4

```
options symbolgen mprint;

%macro makeds(subset);
  %global glbsubset;
  %let glbsubset=&subset;

  data temp;
    set books.ytdsales(where=(section="&subset"));
    attrib qtrsold label='Quarter of Sale';
    qtrsold=qtr(datesold);
  run;
%mend makeds;

%makeds(Internet)

proc tabulate data=temp;
  title "Book Sales Report Produced &sysdate9";
  class qtrsold;
  var saleprice listprice;
  tables qtrsold all,
         (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
         box="Section: &glbsubset";
  keylabel all='** Total **';
run;
```

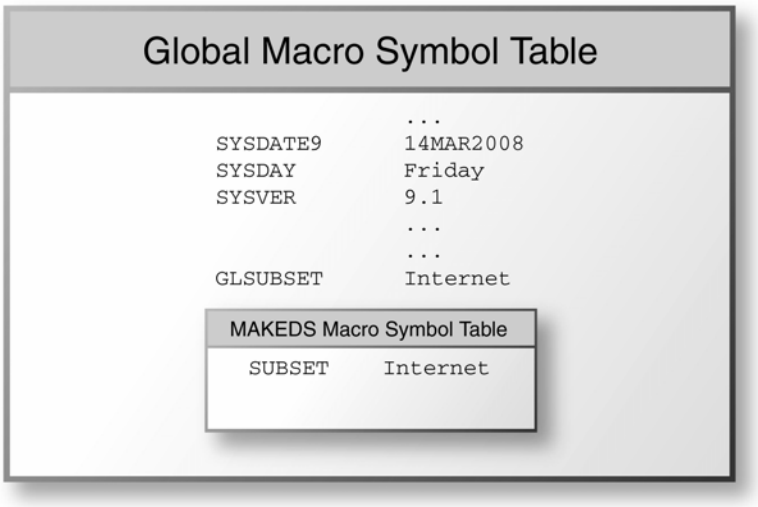
The SAS log for this second version follows.

```
194  options symbolgen mprint;
195
196  %macro makeds(subset);
197    %global glbsubset;
198    %let glbsubset=&subset;
199
200    data temp;
201      set books.ytdsales(where=(section="&subset"));
202      attrib qtrsold label='Quarter of Sale';
203      qtrsold=qtr(datesold);
204    run;
205  %mend makeds;
206
207  %makeds(Internet)
SYMBOLGEN:  Macro variable SUBSET resolves to Internet
MPRINT(MAKEDS):   data temp;
SYMBOLGEN:  Macro variable SUBSET resolves to Internet
MPRINT(MAKEDS):   set
books.ytdsales(where=(section="Internet"));
MPRINT(MAKEDS):   attrib qtrsold label='Quarter of Sale';
MPRINT(MAKEDS):   qtrsold=qtr(datesold);
MPRINT(MAKEDS):   run;

NOTE: There were 1456 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Internet';
NOTE: The data set WORK.TEMP has 1456 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

208
209  proc tabulate data=temp;
SYMBOLGEN:  Macro variable SYSDATE9 resolves to 14MAR2008
210    title "Book Sales Report Produced &sysdate9";
211    class qtrsold;
212    var saleprice listprice;
213    tables qtrsold all,
214           (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
215            box="Section: &glbsubset";
SYMBOLGEN:  Macro variable GLBSUBSET resolves to Internet
216    keylabel all='** Total **';
217  run;

NOTE: There were 1456 observations read from the data set
```

```
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

Figure 5.4 shows a representation of the macro symbol tables after the %LET statement inside the MAKEDS macro program executes.

Figure 5.4 The macro symbol tables after the %LET statement inside MAKEDS in Program 5.4 has executed



Working with Global Macro Variables and Local Macro Variables

The macro processor follows a specific set of rules to resolve macro variable references. The context in which you place a macro variable reference tells the macro processor which symbol table to begin with in its attempt to resolve a macro variable reference.

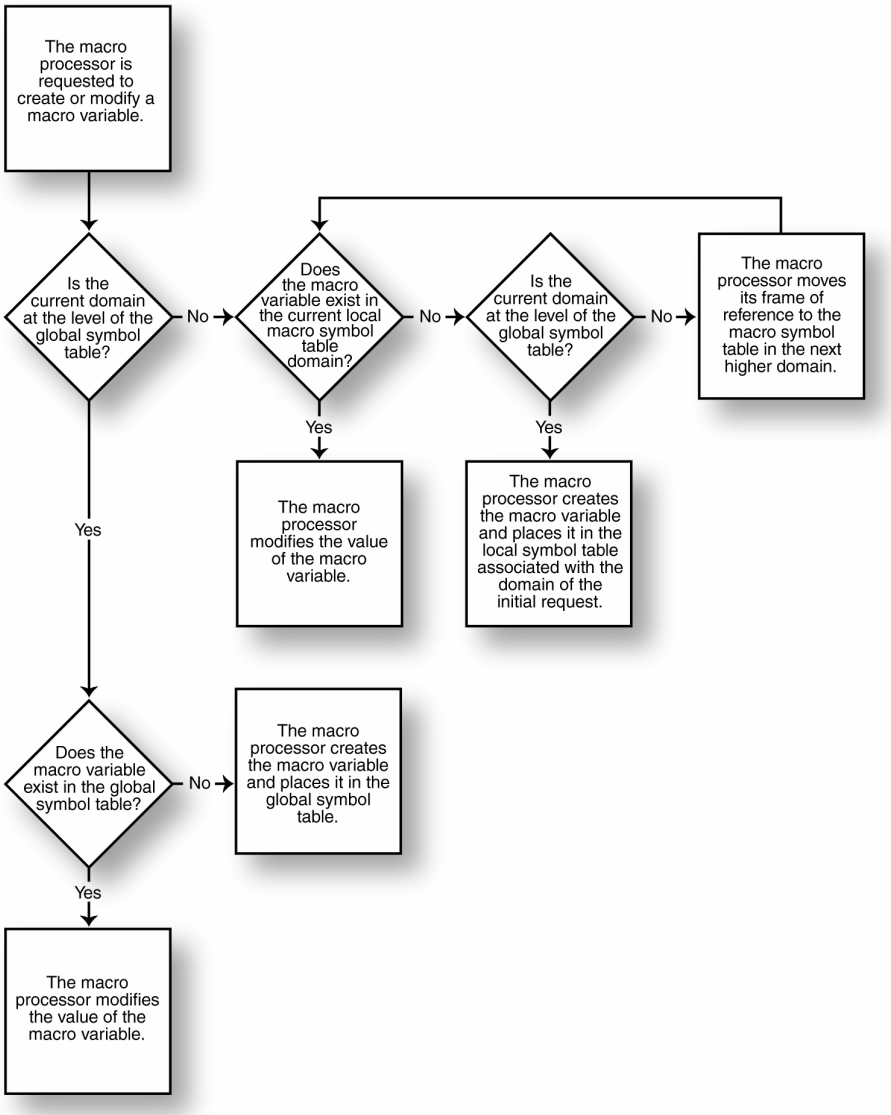
When you ask the macro processor to resolve a macro variable reference, the macro processor first looks in the most local domain of that macro variable. If the macro variable is called from within a macro program, the most local domain of the macro variable is the local macro symbol table associated with the macro program. If the macro variable is not in the most local domain, the macro processor moves to the next higher domain. The search stops when the macro processor reaches the domain of the global macro symbol table. If the macro variable cannot be found in the global macro symbol table, the macro processor issues a warning.

When your macro program references a global macro variable, the macro processor does not by default create a local macro variable with that name. Instead, the macro processor uses the global macro variable.

When you define more than one macro program in your SAS program, the order of the macro program definitions does not matter. Furthermore, if one macro program calls another, you do not have to nest the definition of the called macro program within the definition of the first. Each macro program definition is its own entity, like a subroutine. Indeed, it will probably be easier to read your code if you do not nest macro program definitions.

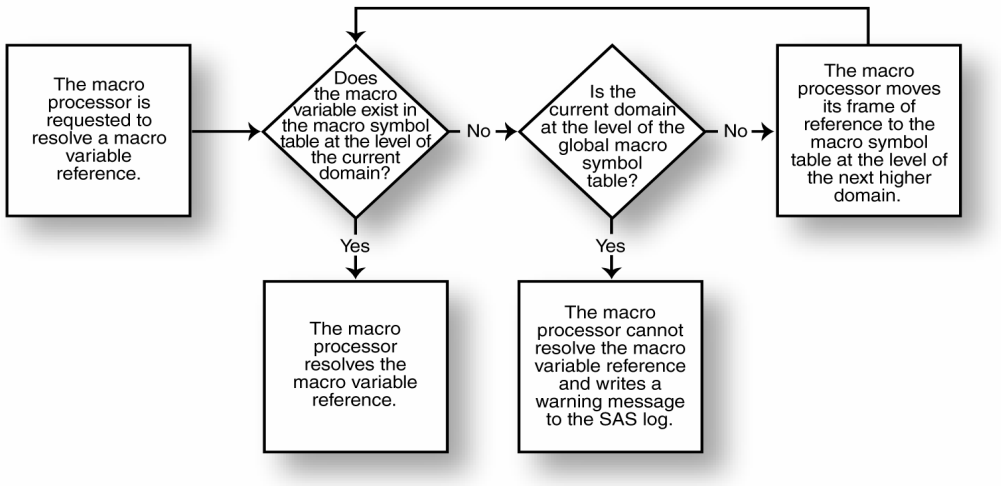
The macro processor follows the path in Figure 5.5 when it creates or modifies macro variables.

Figure 5.5 How the macro processor accesses symbol tables when it creates or modifies macro variables



The process that the macro processor follows when it resolves a macro variable reference is shown in Figure 5.6.

Figure 5.6 How the macro processor accesses symbol tables when it resolves a macro variable reference



Defining the Domain of a Macro Variable by Using the %GLOBAL or %LOCAL Macro Language Statements

The %GLOBAL and %LOCAL macro language statements explicitly direct where the macro processor stores macro variables. These statements can override the rules described in the previous section as well as help document your programs.

The syntax of both statements is

```
%GLOBAL macro-variable(s) ;
%LOCAL macro-variable(s) ;
```

The %GLOBAL statement tells the macro processor to create the macro variables listed on the statement and store them in the global macro symbol table. The macro processor initially sets these macro variables to a null value. These macro variables can be used throughout the SAS session. The %GLOBAL statement can be used in open code or inside a macro program. A %GLOBAL statement was used in the preceding section in Program 5.4.

The macro processor places the macro variables listed on the %LOCAL statement in the domain of the macro program in which the %LOCAL statement was issued. The

%LOCAL statement can only be used within a macro program. The macro processor will look only in the local domain to resolve references to macro variables on the %LOCAL statement.

Program 5.5 uses the same name, SUBSET, for two macro variables in different domains. The first reference to macro variable SUBSET is made in open code where it is specified on a %LET statement. The %LET statement assigns the global version of SUBSET the value `Internet`.

Macro program LOCLMVAR first references SUBSET on the %LOCAL statement. This causes the macro processor to look only locally to resolve a reference to SUBSET during execution of macro program LOCLMVAR. The %LET statement that follows the %LOCAL statement assigns the local version of SUBSET the value `Web Design`.

Program 5.5

```
options symbolgen mprint;

%let subset=Internet;

%macro loclmvar;
  %local subset;
  %let subset=Web Design;

  proc means data=books.ytdsales n sum maxdec=2;
    title "Book Sales Report Produced &sysdate9";
    title2 "Uses LOCAL SUBSET macro variable: &subset";
    where section="&subset";
    var saleprice;
  run;
%mend loclmvar;

%loclmvar

proc means data=books.ytdsales n sum maxdec=2;
  title "Book Sales Report Produced &sysdate9";
  title2 "Uses GLOBAL SUBSET macro variable: &subset";
  where section="&subset";
  var saleprice;
run;
```

The SAS log for the preceding program follows.

```
269  options symbolgen mprint;
270
271  %let subset=Internet;
272
273  %macro loclmvar;
274    %local subset;
275    %let subset=Web Design;
276
277    proc means data=books.ytdsales n sum maxdec=2;
278      title "Book Sales Report Produced &sysdate9";
279      title2 "Uses LOCAL SUBSET macro variable: &subset";
280      where section="&subset";
281      var saleprice;
282    run;
283  %mend loclmvar;
284
285  %loclmvar
MPRINT(LOCLMVAR):   proc means data=books.ytdsales n sum
maxdec=2;
SYMBOLGEN:  Macro variable SYSDATE9 resolves to 14MAR2008
MPRINT(LOCLMVAR):   title "Book Sales Report Produced
14MAR2008";
SYMBOLGEN:  Macro variable SUBSET resolves to Web Design
MPRINT(LOCLMVAR):   title2 "Uses LOCAL SUBSET macro variable:
Web Design";
SYMBOLGEN:  Macro variable SUBSET resolves to Web Design
MPRINT(LOCLMVAR):   where section="Web Design";
MPRINT(LOCLMVAR):   var saleprice;
MPRINT(LOCLMVAR):   run;

NOTE: There were 846 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Web Design';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

286
287  proc means data=books.ytdsales n sum maxdec=2;
SYMBOLGEN:  Macro variable SYSDATE9 resolves to 14MAR2008
288    title "Book Sales Report Produced &sysdate9";
SYMBOLGEN:  Macro variable SUBSET resolves to Internet
289    title2 "Uses GLOBAL SUBSET macro variable: &subset";
290    where section="&subset";
```

```
SYMBOLGEN:  Macro variable SUBSET resolves to Internet
291    var saleprice;
292  run;

NOTE: There were 1456 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Internet';
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

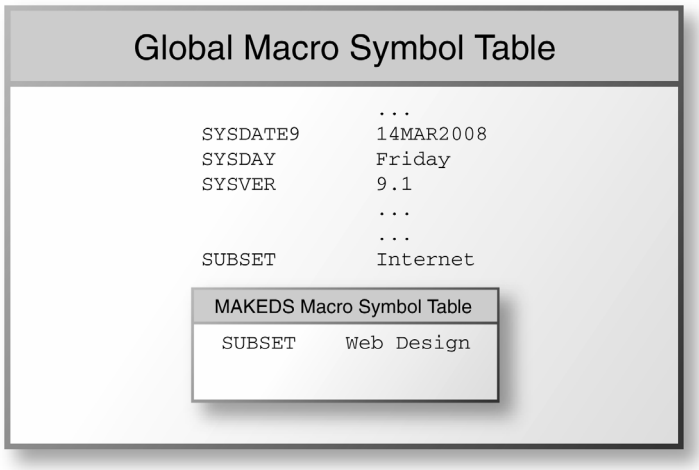
Output 5.3 presents the output from Program 5.5.

Output 5.3 Output from Program 5.5, which uses the same name for a local macro variable and for a global macro variable

```
          Book Sales Report Produced 14MAR2008                    1
       Uses LOCAL SUBSET macro variable: Web Design

                  The MEANS Procedure

        Analysis Variable : saleprice Sale Price

                     N             Sum
                ---------------------
                 846        37121.52
                ---------------------


          Book Sales Report Produced 14MAR2008                    2
       Uses GLOBAL SUBSET macro variable: Internet

                  The MEANS Procedure

        Analysis Variable : saleprice Sale Price

                     N             Sum
                ---------------------
                1456        62295.78
                ---------------------
```

A representation of the macro symbol tables when the LOCLMVAR macro program executes is shown in Figure 5.7.

Figure 5.7 The macro symbol tables when LOCLMVAR executes



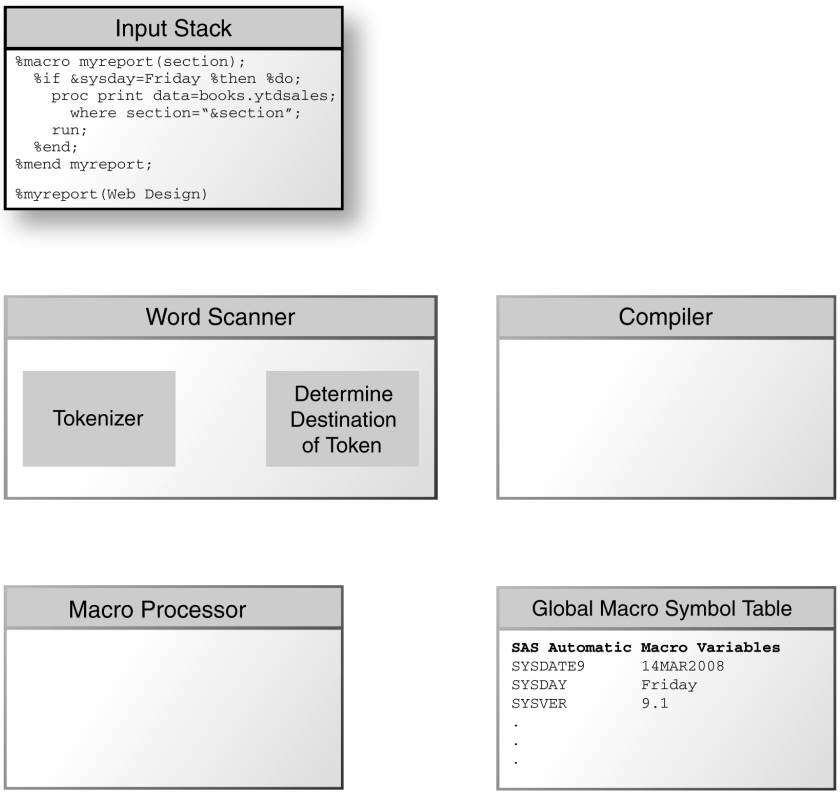
Processing of Macro Programs

Understanding how the macro processor compiles and executes macro programs will help you more quickly write and debug SAS programs that contain macro language. This section describes how SAS and the macro processor process macro programs.

How a Macro Program Is Compiled

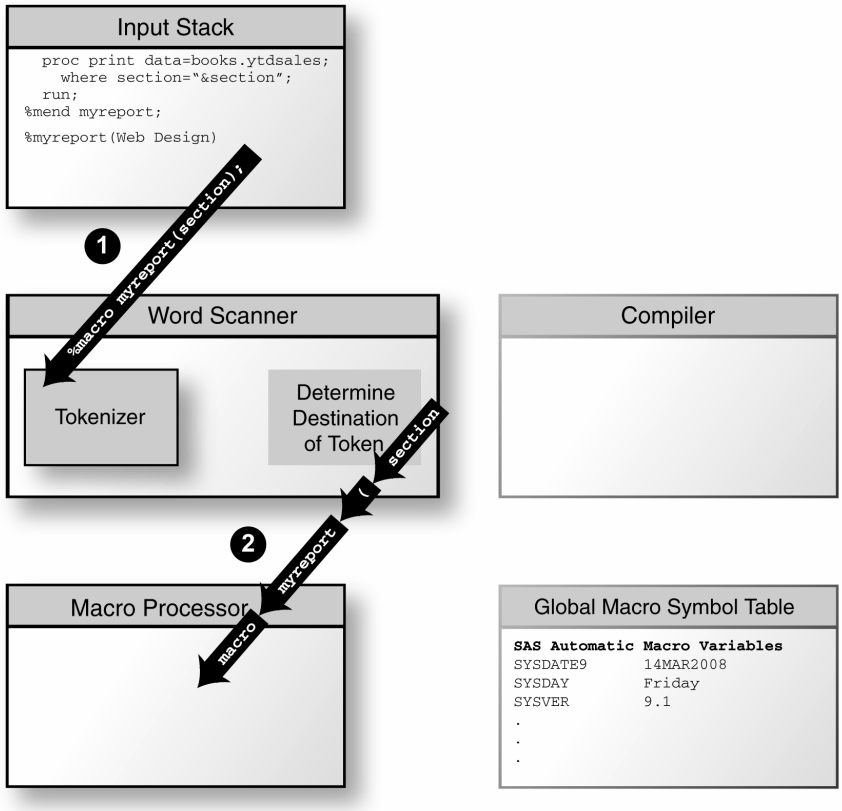
The SAS program in Figure 5.8 contains a macro program definition and a call to the macro program. The remaining figures in this section show the path that SAS and the macro processor follow in compiling the macro program definition.

Figure 5.8 A SAS program containing a macro program definition and a call to the macro program has been submitted for processing



Tokenization of the program begins in Figure 5.9.

Figure 5.9 Statements are transferred from the input stack to the word scanner



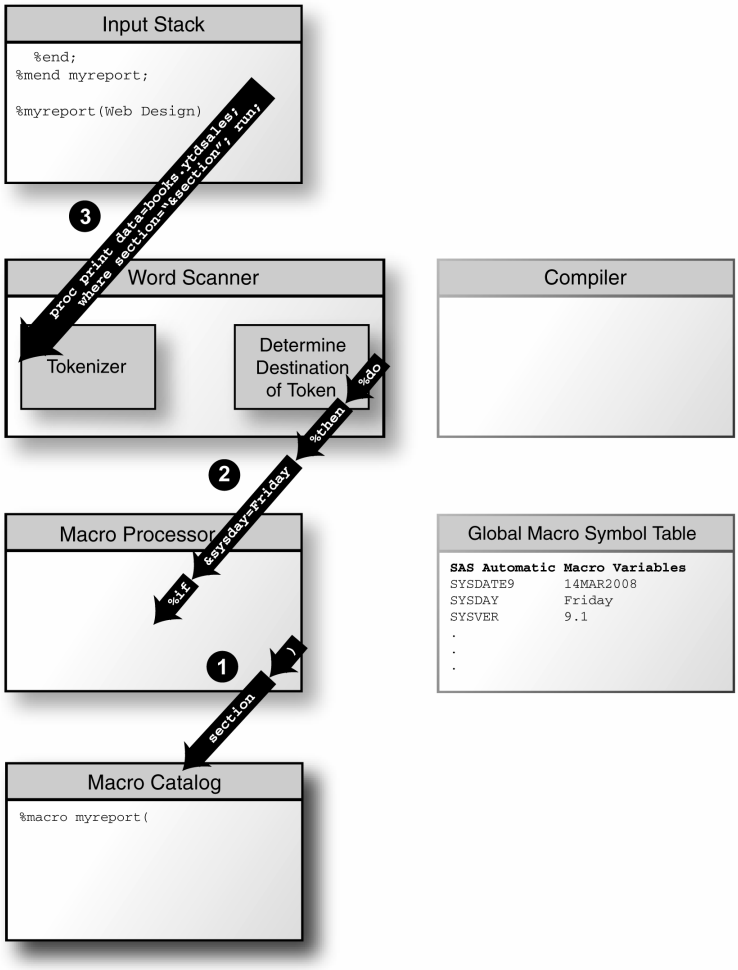
The two steps in Figure 5.9 are as follows:

❶ The MACRO statement is passed to the word scanner for tokenization.

❷ The word scanner detects a percent sign followed by a nonblank character and sends subsequent tokens to the macro processor.

Figure 5.10 shows that the word scanner continues to send tokens to the macro processor.

Figure 5.10 Tokens continue to be transferred to the macro processor



The three steps in Figure 5.10 are:

❶ An entry in the macro catalog for macro program MYREPORT is created.

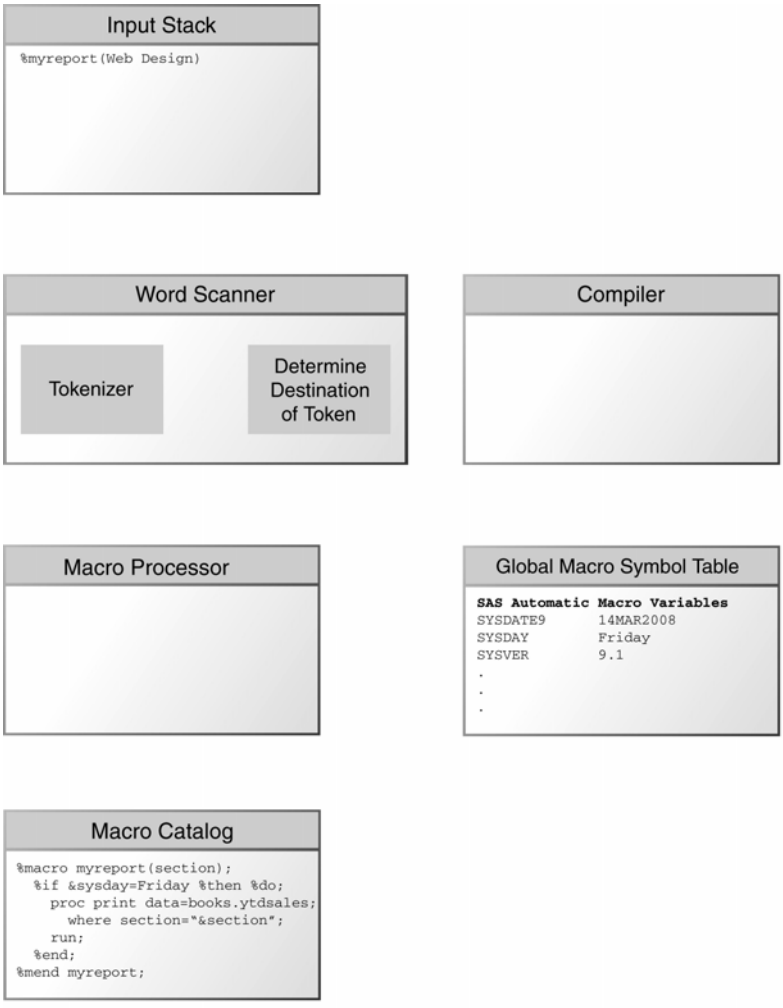
❷ The word scanner passes the tokens from the %IF statement to the macro processor. The expression `&SYSDAY=Friday` is temporarily considered one token and will not be completely tokenized and resolved until MYREPORT executes.

❸ The entire PROC PRINT step is considered one text token and is passed to the macro processor for storage with the MYREPORT macro program.

Figure 5.11 shows that the compilation of the macro statements in MYREPORT is complete. The expression `&SYSDAY=Friday` is stored as text and will not be resolved until MYREPORT is executed. The PROC PRINT step is stored as text and will not be tokenized and compiled until MYREPORT executes.

The call to macro program MYREPORT is in the input stack and ready for processing in Figure 5.11.

Figure 5.11 Compilation of macro program MYREPORT is complete

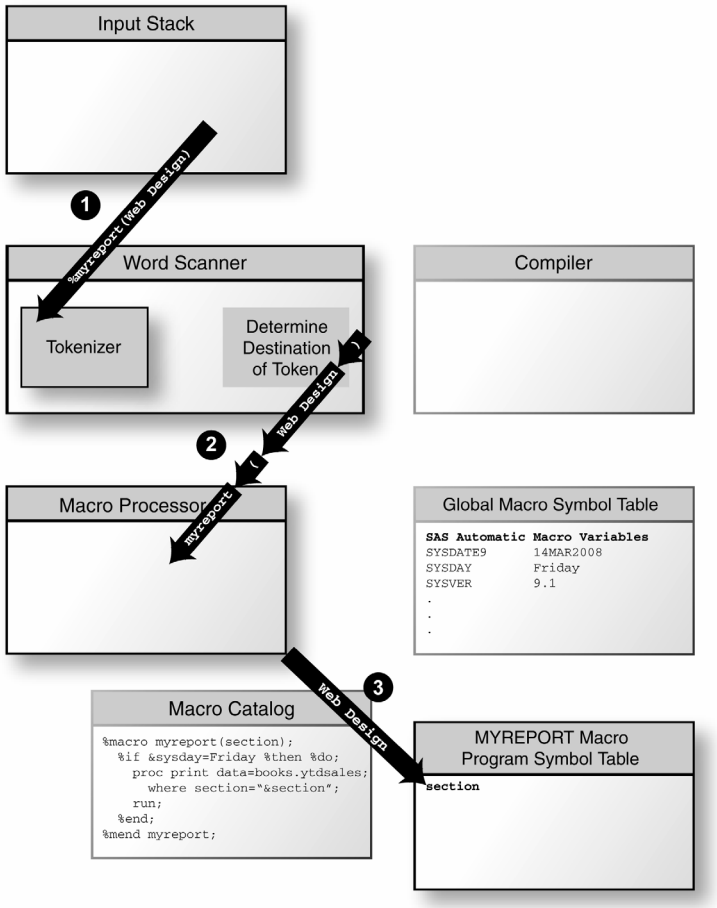


How a Macro Program Is Executed

This section continues with the example from the previous section by executing a call to the macro program MYREPORT. The figures in this section describe the process.

Figure 5.12 shows that a call to the macro program MYREPORT has been made and that the value assigned to the parameter SECTION is `Web Design`.

Figure 5.12 The macro program MYREPORT has been called and begins executing



The three steps in Figure 5.12 are:

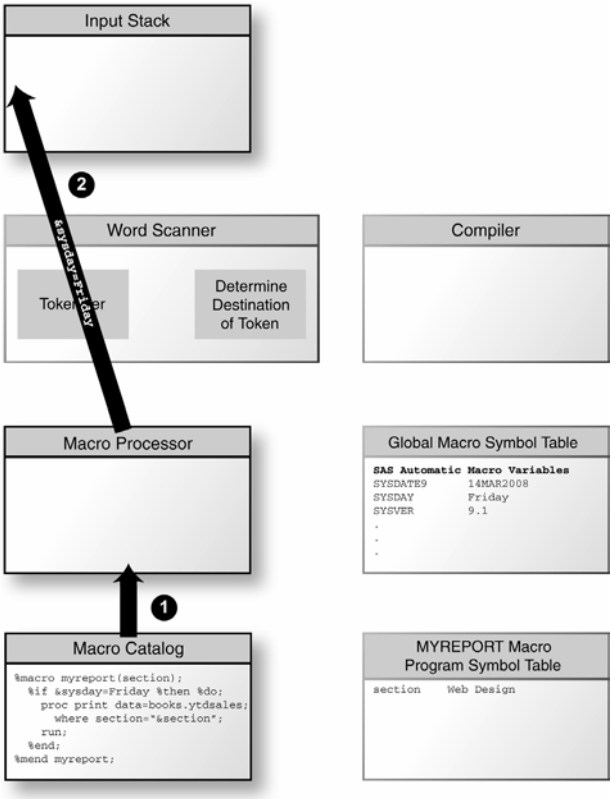
❶ Macro program MYREPORT has been called.

❷ The word scanner detects the percent sign macro trigger followed by text and transfers tokens to the macro processor.

❸ The macro processor begins executing macro program MYREPORT. The macro processor creates a macro symbol table for MYREPORT. It adds macro variable SECTION to the MYREPORT symbol table. The initial value for SECTION that is passed as a parameter to the macro program MYREPORT is placed in the symbol table.

In Figure 5.13, the macro processor starts executing MYREPORT.

Figure 5.13 The macro program MYREPORT continues executing



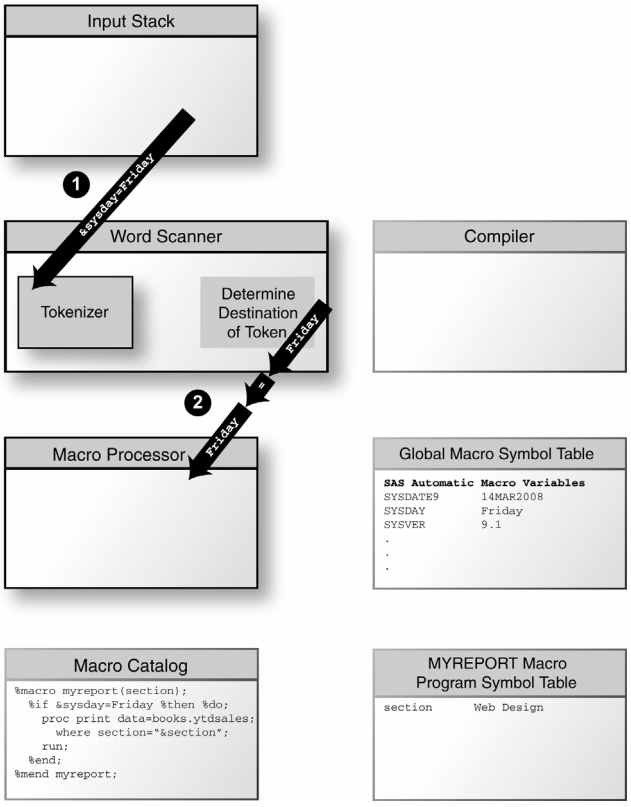
The two steps in Figure 5.13 are:

❶ The macro processor executes the compiled %IF statement.

❷ The macro processor puts the text `&SYSDAY=FRIDAY` on the input stack so that it can be tokenized by the word scanner.

Next, the word scanner tokenizes the `&SYSDAY=Friday` expression and directs resolution of the macro variable reference to the macro processor.

Figure 5.14 The word scanner receives the &SYSDAY=Friday expression for tokenization and evaluation of the expression is passed to the macro processor



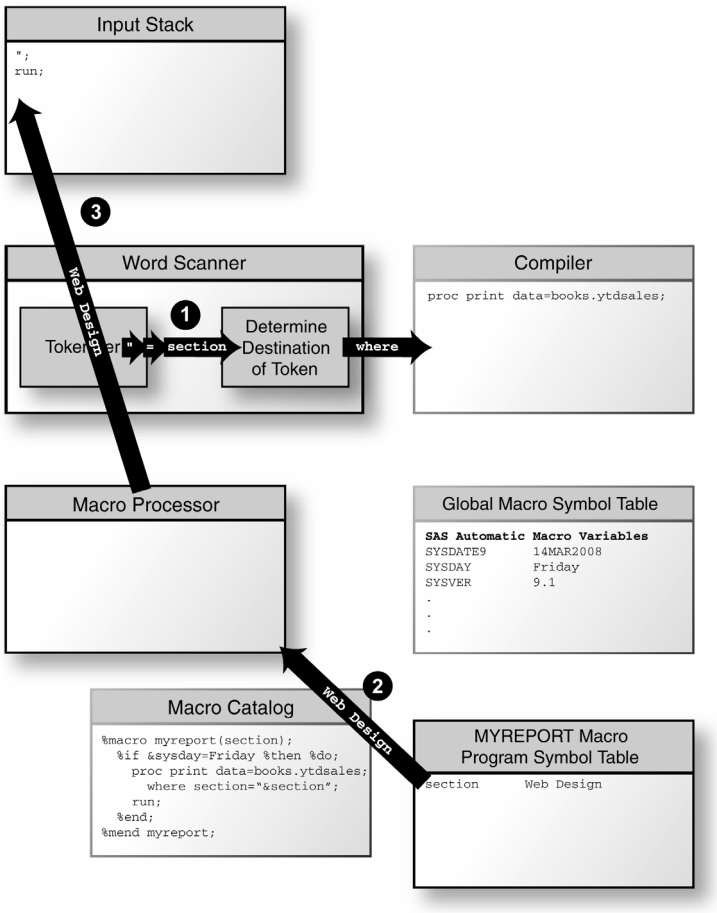
The two steps in Figure 5.14 are:

❶ The word scanner receives the `&SYSDAY=Friday` expression.

❷ After receiving the resolved value of SYSDAY from the macro processor, the word scanner sends the tokens to the macro processor for evaluation.

Assume that MYREPORT was run on a Friday. Therefore, the %IF condition is true. The statements in the PROC PRINT step are placed in the input stack by the macro processor. Execution of MYREPORT continues in Figure 5.15.

Figure 5.15 The PROC PRINT step is tokenized and the macro variable reference to &SECTION is resolved



The three steps in Figure 5.15 are:

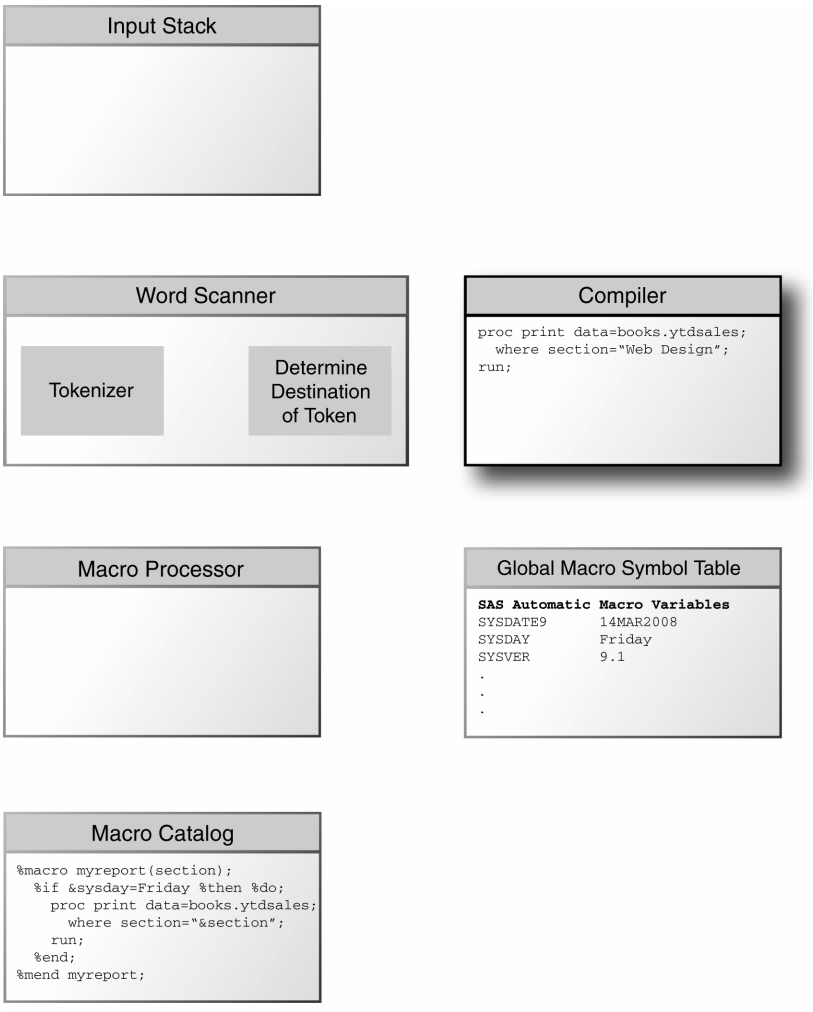
❶ The word scanner tokenizes the SAS language statements and passes them to the compiler.

❷ The macro processor resolves the reference to &SECTION, which is made within the MYREPORT macro program. The macro symbol table for MYREPORT is the first place the macro processor looks to resolve &SECTION.

❸ The macro processor sends the value of macro variable SECTION to the input stack. This value is treated as one token.

Figure 5.16 shows that all statements have been tokenized and macro variable references have been resolved. The macro processor is put "on hold" while the PROC PRINT step executes. After the step executes, control returns to the macro processor. Since there are no additional steps or statements in the macro program, the %MEND statement executes and the macro processor deletes the macro symbol table associated with macro program MYREPORT.

Figure 5.16 The SAS program is ready for compilation



Chapter 6

Macro Language Functions

Introduction 133
Macro Character Functions 134
Macro Evaluation Functions 138
Macro Quoting Functions 140
Macro Variable Attribute Functions 143
Other Macro Functions 147
SAS Supplied Autocall Macro Programs Used Like Functions 154

Introduction

The preceding chapters describe the basic structures of the macro programming language and the mechanics involved in processing macro language. This chapter describes the functions that are available in the macro programming language.

Macro functions greatly extend the use of macro variables and macro programming. Macro functions can be used in open code and in macro programs. The arguments of a macro function can be text strings, macro variables, macro functions, and macro program calls. The *result* of a macro function is always text. This result can be assigned to a macro variable. A macro function can also be inserted directly into your SAS statements to build SAS statements.

Most macro functions have SAS language counterparts. If you know how to write DATA step programs, you already have a familiarity with the style and structure of many of the macro functions.

Some of the tasks you can do with macro functions include:

- extracting substrings of macro variables
- searching for a string of characters in a macro variable
- temporarily converting macro values to numeric so that you can use the macro variables in calculations
- using SAS language functions and functions created with SAS/TOOLKIT in your macro language statements
- allowing semicolons to be treated as a text value rather than as a symbol to terminate a statement

This chapter classifies the macro functions into five categories: character, evaluation, quoting, macro variable attribute, and other. These categories and some of the functions in each category are briefly described below.

Additionally, SAS ships a library of autocall macro programs with its software, which may or may not be installed at your site. Autocall macro programs are uncompiled source code and text stored as entries in SAS libraries. This set of autocall macro programs can be used like macro language functions. This SAS supplied autocall macro program library is described at the end of this chapter. Also, see Chapter 10 for more discussion on the application of autocall macro programs.

Macro Character Functions

Macro character functions operate on strings of text or on macro variables. These functions modify their arguments or provide information about their arguments. Several

of the character functions you might be familiar with in the SAS language have macro language counterparts. Table 6.1 lists the macro character functions.

Macro functions %SCAN, %SUBSTR, and %UPCASE each have a version that should be used if the result of the macro function could contain a special character or mnemonic operator. The names of these macro functions are %QSCAN, %QSUBSTR, and %QUPCASE.

Table 6.1 Macro character functions

Function	Action
%INDEX(*source, string*)	returns the position in *source* of the first character of *string.*
%LENGTH(*string*|*text expression*)	returns the length of *string* or the length of the results of the resolution of *text expression.*
%SCAN(*argument, n <,delimiters>*)	returns the nth word in *argument* where the words in *argument* are separated by *delimiters.* Use %QSCAN when you need to mask special characters or mnemonic operators in the result.
%SUBSTR(*argument,position<,length>)*	extracts a substring of *length* characters from *argument* starting at *position.* Use %QSUBSTR when you need to mask special characters or mnemonic operators in the result.
%UPCASE(*string*|*text expression)*	converts *character string* or *text expression* to uppercase. Use %QUPCASE when you need to mask special characters or mnemonic operators in the result.

Example 6.1: Using %SUBSTR to Extract Text from a Macro Variable Value

Program 6.1 shows how the %SUBSTR function extracts text from strings of characters. The WHERE statement selects observations from the first day of the current month through the day the program was run.

Program 6.1

```
proc means data=books.ytdsales;
  title "Sales for 01%substr(&sysdate,3,3) through &sysdate9";
  where "01%substr(&sysdate,3)"d le datesold le "&sysdate"d;
  class section;
  var saleprice;
run;
```

After resolution of the macro variable references, the PROC MEANS step looks as follows when submitted on September 15, 2007.

```
proc means data=books.ytdsales;
  title "Sales for 01SEP through 15SEP2007";
  where "01SEP07"d le datesold le "15SEP07"d;
  class section;
  var saleprice;
run;
```

Example 6.2: Using %SCAN to Extract the Nth Item from a Macro Variable Value

The %SCAN macro character function in Program 6.2 extracts a specific word from a string of words that are separated with blanks. The code specifies that %SCAN, through the value of REPMONTH, extract the third word from macro variable MONTHS.

One of the default delimiters for %SCAN is a blank. Therefore, the optional third argument to %SCAN is not specified in Program 6.2.

Under ASCII systems, the other default delimiters for %SCAN are:

. < (+ & ! $ *) ; ^ - / , % |

Under EBCDIC systems, the other default delimiters for %SCAN are:

. < (+ | & ! $ *) ; ¬ - / , % ¦ ¢

Program 6.2

```
%let months=January February March April May June;
%let repmonth=3;

proc print data=books.ytdsales;
  title "Sales Report for %scan(&months,&repmonth)";
  where month(datesold)=&repmonth;
  var booktitle author saleprice;
run;
```

After resolution of the macro variable references, the PROC PRINT step becomes:

```
proc print data=books.ytdsales;
  title "Sales Report for March";
  where month(datesold)=3;
  var booktitle author saleprice;
run;
```

Example 6.3: Using %UPCASE to Convert a Macro Variable Value to Uppercase

Macro program LISTTEXT in Program 6.3 lists all the titles sold that contain a specific text string. The text string is passed to the macro program through the parameter KEYTEXT. This text string might be in different forms in the title: lowercase, uppercase, or mixed case. Because of this, both the macro variable's value and the value of the data set variable TITLE are converted to uppercase. This increases the likelihood of matches when the two are compared.

Program 6.3

```
%macro listtext(keytext);
  %let keytext=%upcase(&keytext);
  proc print data=books.ytdsales;
  title "Book Titles Sold Containing Text String &keytext";
    where upcase(booktitle) contains "&keytext";
    var booktitle author saleprice;
  run;
%mend;

%listtext(web)
```

When the macro program executes, the TITLE statement resolves to

```
Book Titles Sold Containing Text String WEB
```

The WHERE statement at execution resolves to

```
where upcase(booktitle) contains "WEB";
```

Macro Evaluation Functions

The two macro evaluation functions, %EVAL and %SYSEVALF, evaluate arithmetic expressions and logical expressions. These expressions are comprised of operators and operands that the macro processor evaluates to produce a result. The arguments to one of these macro evaluation functions are temporarily converted to numbers so that a calculation (arithmetic or logical) can be completed. The macro evaluation function converts the result that it returns to text.

Arithmetic expressions use arithmetic operators such as plus signs and minus signs. Logical expressions use logical operators such as greater than signs and equal signs.

The %EVAL function evaluates expressions using integer arithmetic. The %SYSEVALF function evaluates expressions using floating point arithmetic. Macro expressions are constructed with the same arithmetic and comparison operators found in the SAS language. A section in Chapter 7 discusses in more detail how to construct macro expressions.

The syntax of the %EVAL function is

```
%EVAL(arithmetic expression|logical expression)
```

The syntax of the %SYSEVALF function is

```
%SYSEVALF(arithmetic expression|logical expression
    <,conversion-type>)
```

By default, the result of the %SYSEVALF function is left as a number which is converted back to text. Otherwise, you can request %SYSEVALF to convert the result to a different format, as shown in Table 6.2. When requesting one of these four conversion types, specify the conversion type as the second argument to %SYSEVALF.

Table 6.2 Conversion types that can be specified on the %SYSEVALF function

Conversion Type	Result that is returned by %SYSEVALF
BOOLEAN	0 if the result of the expression is 0 or null 1 if the result is any other value (The 0 and 1 are treated as text.)
CEIL	text that represents the smallest integer that is greater than or equal to the result of the expression
FLOOR	text that represents the largest integer that is less than or equal to the result of the expression
INTEGER	text that represents the integer portion of the expression's result

For more discussion on %SYSEVALF and %EVAL, see Example 6.8 later in this chapter.

The %EVAL function does *integer arithmetic*. Therefore, this function treats numbers with decimal points as text. The %EVAL function generates an error when there are characters in the arguments that are supplied to %EVAL, as demonstrated by the second example in Table 6.3.

The statements in Table 6.3 show examples of the %EVAL and %SYSEVALF functions. The %PUT statements were submitted, and the results were written to the SAS log.

Table 6.3 Examples of %EVAL and %SYSEVALF evaluation functions

%PUT Statement	Results in SAS log
`%put %eval(33 + 44);`	`77`
`%put %eval(33.2 + 44.1);`	`ERROR: A character operand was found in the %EVAL function or %IF condition where a numeric operand is required. The condition was: 33.2 + 44.1`
`%put %sysevalf(33.2 + 44.1);`	`77.3`
`%put %sysevalf(33.2 + 44.1,integer);`	`77`
`%let a=3;` `%let b=10;`	
`%put %eval(&b/&a);`	`3`
`%put %sysevalf(&b/&a);`	`3.3333333333`
`%put %sysevalf(&b/&a,ceil);`	`4`
`%put %sysevalf(&b/&a,boolean);`	`1`
`%let missvalu=.;` `%put %sysevalf(&b-&missvalu,boolean);`	`NOTE: Missing values were generated as a result of performing an operation on missing values during %SYSEVALF expression evaluation.` `0`

Macro Quoting Functions

Macro quoting functions mask special characters and mnemonic operators in your macro language statements so that the macro processor does not interpret them. The macro processor instead treats these items simply as text.

For example, you might want to assign a value to a macro variable that contains a character that the macro processor interprets as a macro trigger. The macro processor considers ampersands and percent signs followed by text as macro triggers. You must use a macro quoting function to tell the macro processor to ignore the special meaning of the ampersands and percent signs and instead treat them as text.

Consider what happens if you assign the name of the publisher, Doe&Lee Ltd., to a macro variable:

```
%let publisher=Doe&Lee Ltd.;
```

If you have not already defined a macro variable named LEE in your SAS session, you will see the following message displayed in your SAS log:

```
WARNING: Apparent symbolic reference LEE not resolved.
```

(If you had already defined a macro variable named LEE in your SAS session, you would not see the warning. Instead, the macro processor would resolve the reference to &LEE with the value assigned to the macro variable LEE.)

To prevent the macro processor from interpreting the ampersand as a macro trigger in the value being assigned to PUBLISHER, you must mask the value that you assign to the macro variable PUBLISHER. The macro quoting function %NRSTR correctly masks &LEE from view by the macro processor when the instruction is compiled. Therefore, when you apply %NRSTR to the text string, the macro processor ignores the ampersand as a macro trigger, does not attempt to resolve the value of the macro variable &LEE, and considers this use of the ampersand simply as text.

```
%let publisher=%nrstr(Doe&Lee Ltd.);
```

As it steps through its tasks in compiling and executing the %LET statement, the macro processor defines a global macro variable, PUBLISHER, and assigns the text `Doe&Lee Ltd.` to PUBLISHER.

The macro quoting functions can be grouped into three types based upon when they act: compilation, execution, and preventing resolution during execution.

Table 6.4 lists the macro quoting functions. Chapter 8 discusses the topic of masking characters in macro programming more thoroughly, and it includes several examples that illustrate the concepts on how and when to apply the macro quoting functions.

Table 6.4 Macro quoting functions

Function	Action
%BQUOTE(*character-string | text expression*)	Mask special characters and mnemonic operators in a character string or the value of resolved text expression at macro execution. Compared to %QUOTE, %BQUOTE does not require that unmatched quotation marks or unmatched parentheses be marked with a preceding percent sign (%).
%NRBQUOTE(*character-string | text expression*)	Does the same as %BQUOTE and additionally masks ampersands (&) and percent signs (%).
%QUOTE(*character-string | text expression*)	Mask special characters and mnemonic operators in a character string or the value of resolved text expression at macro execution. Compared to %BQUOTE, %QUOTE requires that unmatched quotation marks and unmatched parentheses be marked with a preceding percent sign (%).
%NRQUOTE(*character-string | text expression*)	Does the same as %QUOTE and additionally masks ampersands (&) and percent signs (%).
%STR(*character-string*)	Mask special characters and mnemonic operators in constant text at macro compilation.
%NRSTR(*character-string*)	Does the same as %STR and additionally masks ampersands (&) and percent signs (%).
%SUPERQ(*macro-variable-name*)	Masks all special characters including ampersands (&) and percent signs (%) and mnemonic operators at macro execution and prevents further resolution of the value. Returns the value of a macro variable and does not resolve any macro references contained in that macro variable's value.
%UNQUOTE(*character-string | text expression*)	Unmasks all special characters and mnemonic operators in a value at macro execution.

Macro Variable Attribute Functions

The three macro variable attribute functions supply information about the existence and the domain (global vs. local) of macro variables. These functions can be especially useful when debugging problems with macro variable resolution. Table 6.5 lists the macro variable attribute functions.

Table 6.5 Macro variable attribute functions

Function	Action
`%SYMEXIST(`*`macro-variable-name`*`)`	returns a 0 or 1 depending on whether the named macro variable exists. The search starts with the most local symbol table, and the search proceeds up the hierarchy through other local symbol tables, ending the search at the global symbol table. If the macro variable exists, %SYMEXIST return a value of 1; otherwise, it returns a 0.
`%SYMGLOBL(`*`macro-variable-name`*`)`	returns a 0 or 1 depending on whether the named macro variable is found in the global symbol table. If the macro variable exists in the global symbol table, %SYMGLOBL returns a value of 1; otherwise, it returns a 0.
`%SYMLOCAL(`*`macro-variable-name`*`)`	returns a 0 or 1 depending on whether the named macro variable is found in a local symbol table. The search starts with the most local symbol table, and the search proceeds up the hierarchy through other local symbol tables, ending the search at the local symbol table highest up in the hierarchy. If the macro variable exists in a local symbol table, %SYMLOCAL returns a value of 1; otherwise, it returns a 0.

Example 6.4: Using Macro Variable Attribute Functions to Determine Domain and Existence of Macro Variables

Chapter 5 discusses domains of macro variables. A macro variable can exist in either the global or local macro symbol table. You can successfully reference a macro variable stored in the global symbol table throughout your SAS session including within macro programs. There is only one global symbol table.

A local macro symbol table is created by executing a macro program that contains macro variables. If the macro variables do not already exist in the global table, macro variables defined in the macro program are stored in the local macro symbol table associated with the macro program. These local macro variables can be referenced only from within the macro program. The macro processor deletes a local macro symbol table when the macro program associated with the table ends. You can have more than one local macro symbol table at a time if one macro program calls another.

Program 5.4 in Chapter 5, which demonstrates domains of macro variables, is modified below in Program 6.4 to include the three functions described in Table 6.5 and illustrates their use. This program introduces a new statement, %SYMDEL, which deletes macro variables from the global symbol table.

Program 6.4

```
%* For example purposes only, ensure these two macro
   variables do not exist in the global symbol table;
%symdel glbsubset subset;

%macro makeds(subset);
  %global glbsubset;
  %let glbsubset=&subset;

   %* What is domain of SUBSET and GLBSUBSET inside MAKEDS?;
   %put ******** Inside macro program;
   %put Is SUBSET a local macro variable(0=No/1=Yes):
%symlocal(subset);
   %put Is SUBSET a global macro variable(0=No/1=Yes):
%symglobl(subset);
   %put Is GLBSUBSET a local macro variable(0=No/1=Yes):
%symlocal(glbsubset);
   %put Is GLBSUBSET a global macro variable(0=No/1=Yes):
%symglobl(glbsubset);
   %put ********;
```

```
  data temp;
    set books.ytdsales(where=(section="&subset"));
    attrib qtrsold label='Quarter of Sale';
    qtrsold=qtr(datesold);
  run;
%mend makeds;

%makeds(Internet)

%* Are SUBSET and GLBSUBSET in global symbol table?;
%put Does SUBSET exist (0=No/1=Yes): %symexist(subset);
%put Is SUBSET a global macro variable(0=No/1=Yes):
%symglobl(subset);
%put Is GLBSUBSET a global macro variable(0=No/1=Yes):
%symglobl(glbsubset);

proc tabulate data=temp;
  title "Book Sales Report Produced &sysdate9";
  class qtrsold;
  var saleprice listprice;
  tables qtrsold all,
         (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
         box="Section: &glbsubset";
  keylabel all='** Total **';
run;
```

The following SAS log for the program shows how the functions %SYMLOCAL, %SYMGLOBL, and %SYMEXIST resolve in this example.

```
288  %* For example purposes only, ensure these two macro
289     variables do not exist in the global symbol table;
290
291  %symdel glbsubset subset;
WARNING: Attempt to delete macro variable GLBSUBSET failed.
         Variable not found.
WARNING: Attempt to delete macro variable SUBSET failed.
         Variable not found.
292
293  %macro makeds(subset);
294    %global glbsubset;
295    %let glbsubset=&subset;
296
297     %* What is domain of SUBSET and GLBSUBSET inside
MAKEDS?;
298     %put ******** Inside macro program;
299     %put Is SUBSET a local macro variable(0=No/1=Yes):
299! %symlocal(subset);
```

```
300     %put Is SUBSET a global macro variable(0=No/1=Yes):
300! %symglobl(subset);
301     %put Is GLBSUBSET a local macro variable(0=No/1=Yes):
301! %symlocal(glbsubset);
302     %put Is GLBSUBSET a global macro variable(0=No/1=Yes):
302! %symglobl(glbsubset);
303     %put ********;
304
305    data temp;
306      set books.ytdsales(where=(section="&subset"));
307      attrib qtrsold label='Quarter of Sale';
308      qtrsold=qtr(datesold);
309    run;
310  %mend makeds;
311
312  %makeds(Internet)
******** Inside macro program
Is SUBSET a local macro variable(0=No/1=Yes): 1
Is SUBSET a global macro variable(0=No/1=Yes): 0
Is GLBSUBSET a local macro variable(0=No/1=Yes): 0
Is GLBSUBSET a global macro variable(0=No/1=Yes): 1
********

NOTE: There were 1456 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Internet';
NOTE: The data set WORK.TEMP has 1456 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


313
314  %* Are SUBSET and GLBSUBSET in global symbol table?;
315  %put Does SUBSET exist (0=No/1=Yes):%symexist(subset);
Does SUBSET exist (0=No/1=Yes):0
316  %put Is SUBSET a global macro variable(0=No/1=Yes):
316! %symglobl(subset);
Is SUBSET a global macro variable(0=No/1=Yes): 0
317  %put Is GLBSUBSET a global macro variable(0=No/1=Yes):
317! %symglobl(glbsubset);
Is GLBSUBSET a global macro variable(0=No/1=Yes): 1
318
319  proc tabulate data=temp;
320    title "Book Sales Report Produced &sysdate9";
321    class qtrsold;
```

```
322    var saleprice listprice;
323    tables qtrsold all,
324           (saleprice listprice)*(n*f=6. sum*f=dollar12.2) /
325           box="Section: &glbsubset";
326    keylabel all='** Total **';
327  run;

NOTE: There were 1456 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

Other Macro Functions

The four macro functions described in this section (Table 6.6) do not fit into any of the four categories of macro functions described so far. These four functions do one of the following:

- apply SAS language functions to macro variables or text
- obtain information from the rest of SAS or the operating system in which the SAS session is running

The %SYSFUNC and %QSYSFUNC functions are especially useful in extending the use of the macro facility. These functions allow you to apply SAS language and user-written functions to your macro programming applications. Several examples of %SYSFUNC follow. Chapter 8, which presents the topic of masking special characters and mnemonic operators, also includes an example that applies %QSYSFUNC. Macro function %QSYSFUNC does the same as %SYSFUNC and it masks special characters and mnemonic operators in the result.

Table 6.6 Other macro functions

Function	Action
%SYSFUNC(*function(argument(s))* <,*format*>)	executes SAS language *function* or user-written *function* and returns the results to the macro facility (see also macro statement %SYSCALL)
%QSYSFUNC(*function(argument(s))* <,*format*>)	does the same as %SYSFUNC with the addition of masking special characters and mnemonic operators in the result
%SYSGET(*host-environment-variable*)	returns the value of *host-environment-variable* to the macro facility
%SYSPROD(*SAS-product*)	returns a code to indicate whether *SAS-product* is licensed at the site where the SAS System is currently running

Using the %SYSFUNC and %QSYSFUNC Macro Functions

The functions %SYSFUNC and %QSYSFUNC apply SAS programming language functions to text and macro variables in your macro programming. With access to the many SAS language functions in your macro programming applications, %SYSFUNC and %QSYSFUNC greatly extend the power of your macro programming.

Since these two functions are macro language functions and the macro facility is a text-handling language, the arguments to the SAS programming language function are not enclosed in quotation marks; it is understood that all arguments are text. Also, the values returned through the use of these two functions are considered text.

Functions cannot be nested within one call to %SYSFUNC and %QSYSFUNC. Each function must have its own %SYSFUNC or %QSYSFUNC call, and these %SYSFUNC and %QSYSFUNC calls can be nested.

Example 6.5: Using %SYSFUNC to Format a Date in the TITLE Statement

The TITLE statement in Program 6.5 shows how the elements of a date can be formatted using %SYSFUNC and the DATE SAS language function.

Program 6.5

```
title
  "Sales for %sysfunc(date(),monname.) %sysfunc(date(),year.)";
```

On January 30, 2007, the title statement would resolve to

```
Sales for January 2007
```

Example 6.6: Using %SYSFUNC to Execute a SAS Language Function and Assign the Result to a Macro Variable

Program 6.6 uses %SYSFUNC to access the SAS language function GETOPTION. The GETOPTION function displays the values of SAS options. The %SYSFUNC function invokes the GETOPTION function and returns the result to macro variable OPTVALUE. The %PUT statement lists the value assigned to OPTVALUE. The single parameter to GETOPTION is the name of the SAS option that should be checked.

Program 6.6

```
%macro getopt(whatopt);
  %let optvalue=%sysfunc(getoption(&whatopt));
  %put Option &whatopt = &optvalue;
%mend getopt;

%getopt(ps)
%getopt(ls)
%getopt(date)
%getopt(symbolgen)
%getopt(compress)
```

The SAS log for Program 6.6 follows.

```
58   %getopt(ps)
Option ps = 54
59   %getopt(ls)
Option ls = 90
60   %getopt(date)
Option date = DATE
```

```
61   %getopt(symbolgen)
Option symbolgen = NOSYMBOLGEN
62   %getopt(compress)
Option compress = NO
```

Example 6.7: Using %SYSFUNC and the NOTNAME and NVALID SAS Language Functions to Determine If a Value Is a Valid SAS Variable Name

Since the macro language generates SAS code for you, a common task that macro programmers have is to construct SAS items such as variable names and format names. In doing so, it might be important to check the value that will be used to name the item to make sure that it does not contain any invalid characters or is too long.

Macro program CHECKVARNAME in Program 6.7 checks a SAS macro variable value to see if it can be used as a variable name. It uses both the NOTNAME and the NVALID SAS functions. The NOTNAME function finds the first position in the value that is an invalid character in naming a variable. The NVALID function determines if the value can be used as a variable name.

The second argument to NVALID in this example is V7. This argument requires three conditions to be true if the value is to be determined valid:

- The value must start with a letter or underscore.
- All subsequent characters must be letters, underscores, or digits.
- Its length must be no greater than 32 characters.

The parameter passed to CHECKVARNAME is the prospective variable name that should be examined. The %SYSFUNC macro function is used in conjunction with the NOTNAME SAS language function and again with the NVALID SAS language function. The macro program writes messages to the SAS log about whether the value can be used as a variable name.

Program 6.7 calls CHECKVARNAME four times. The parameter values specified for the first two calls to CHECKVARNAME are valid SAS names. The parameter values specified in the third and fourth calls to CHECKVARNAME are not valid. The space in the parameter in the third call to CHECKVARNAME is invalid. The length of the parameter in the fourth call, as well as the exclamation point in the last position, makes the value invalid as a SAS name.

Program 6.7

```
%macro checkvarname(value);
  %let position=%sysfunc(notname(&value));
  %put **** Invalid character in position: &position (0 means
&value is okay);
  %let valid=%sysfunc(nvalid(&value,v7));
  %put
     **** Can &value be a variable name(0=No, 1=Yes)? &valid;
  %put;
  %put;
%mend checkvarname;

%checkvarname(valid_name)
%checkvarname( valid_name)
%checkvarname(invalid name)
%checkvarname(book_sales_results_for_past_five_years!)
```

The four calls to macro program CHECKVARNAME produce the following SAS log.

```
235  %checkvarname(valid_name)
**** Invalid character in position: 0 (0 means valid_name is
okay)
**** Can valid_name be a variable name(0=No, 1=Yes)? 1


236  %checkvarname( valid_name)
**** Invalid character in position: 0 (0 means valid_name is
okay)
**** Can valid_name be a variable name(0=No, 1=Yes)? 1


237  %checkvarname(invalid name)
**** Invalid character in position: 8 (0 means invalid name is
okay)
**** Can invalid name be a variable name(0=No, 1=Yes)? 0


238  %checkvarname(book_sales_results_for_past_five_years!)
**** Invalid character in position: 39 (0 means
book_sales_results_for_past_five_years! is okay)
**** Can book_sales_results_for_past_five_years! be a variable
name(0=No, 1=Yes)? 0
```

Example 6.8: Using %SYSFUNC to Apply a SAS Statistical Function to Macro Variable Values

This example uses %SYSFUNC to apply the SAS statistical function MEAN to four macro variable values and compute the mean of the four values. In addition to using a SAS language function in the macro programming environment, this example also illustrates several concepts of macro programming.

The values assigned to the four macro variables A, B, C, and D are treated as *text* values in the macro programming environment. However, note that the MEAN function interprets them as *numbers*. The %SYSEVALF function is not needed to temporarily convert the values to numbers in order to compute the mean. Note also that two periods follow &MEANSTAT in the %PUT statement. The first period terminates the macro variable reference. The second period appears in the text written to the SAS log.

Program 6.8a

```
%let a=1.5;
%let b=-2.0;
%let c=1.978;
%let d=-3.5;
%let meanstat=%sysfunc(mean(&a,&b,&c,&d));
%put ****** The mean of &a, &b, &c, and &d is &meanstat..;
```

After the above code is submitted, the following is written to the SAS log:

```
****** The mean of 1.5, -2.0, 1.978, and -3.5 is -0.5055.
```

A section earlier in this chapter describes the necessity of using the %EVAL and %SYSEVALF functions when you need to temporarily convert macro variable values to numbers to perform calculations was mentioned. In this example, if you wanted to compute the mean using only macro language statements, you would need to use the %SYSEVALF function. You could not use the %EVAL function because the values are specified with decimal places. Also, %EVAL would return an integer result, which would be inaccurate. The code that includes %SYSEVALF could be written as follows.

Program 6.8b

```
%let a=1.5;
%let b=-2.0;
%let c=1.978;
%let d=-3.5;
%let meanstat=%sysevalf( (&a+&b+&c+&d)/4);
%put ****** The mean of &a, &b, &c, and &d is &meanstat..;
```

After submitting this code, the macro processor writes the same text statement to the SAS log as the one generated by the code that uses %SYSFUNC and MEAN in Program 6.8a:

```
****** The mean of 1.5, -2.0, 1.978, and -3.5 is -0.5055.
```

Example 6.9: Using the %SYSFUNC Function to Apply Several SAS Language Functions That Obtain and Display Information about a Data Set

Program 6.9 uses the %SYSFUNC macro function and several SAS language file functions to obtain the last date and time that a data set was updated and to insert that descriptive information in the title of a report. It also uses %SYSFUNC to format the date/time value.

The name of the data set is assigned to macro variable DSNAME. The data set specified by the value of DSNAME is opened with the SAS language OPEN function. Then, the SAS language ATTRN function obtains the last update information by specifying the argument MODTE. The results of the ATTRN function are stored in the macro variable LASTUPDATE. Finally, the SAS language CLOSE function closes the data set.

The value returned by ATTRN is the SAS internal date/time value and is formatted for display in the title with the DATETIME format. The format is applied to the macro variable value stored in LASTUPDATE with %SYSFUNC and the PUTN SAS language function. The second argument to PUTN is the format name DATETIME.

Note that none of the arguments to the SAS file functions are enclosed in quotation marks. This is because the macro facility is a text-handling language and it treats all values as text. The SAS language functions are underlined in this example.

Program 6.9

```
%let dsname=books.ytdsales;
%let dsid=%sysfunc(open(&dsname));
%let lastupdate=%sysfunc(attrn(&dsid,modte));
%let rc=%sysfunc(close(&dsid));

proc report data=books.ytdsales nowd headline;
  title "Publisher List Report &sysdate9";
  title2 "Last Update of &dsname:
%sysfunc(putn(&lastupdate,datetime.))";

  column publisher saleprice;
  define publisher / group width=30;
  define saleprice / format=dollar11.2;
  rbreak after / dol dul summarize;
run;
```

Output 6.1 presents the output from Program 6.9.

Output 6.1 Output from Program 6.9 that includes several applications of the %SYSFUNC macro function

```
          Publisher List Report 18JAN2008
       Last Update of books.ytdsales: 04JAN08:15:27:36

Publisher                          Sale Price
-----------------------------------------
AMZ Publishers                     $22,485.31
Bookstore Brand Titles             $20,120.64
Doe&Lee Ltd.                       $22,688.46
Eversons Books                     $24,091.55
IT Training Texts                  $22,039.65
Mainst Media                       $20,764.50
Nifty New Books                    $21,390.29
Northern Associates Titles         $21,213.95
Popular Names Publishers           $21,058.70
Professional House Titles          $23,555.11
Technology Smith                   $21,766.46
Wide-World Titles                  $22,503.55
                                   ===========
                                   $263,678.15
                                   ===========
```

SAS Supplied Autocall Macro Programs Used Like Functions

Autocall macro programs are uncompiled source code and text stored as entries in SAS libraries. SAS has written several of these useful macro programs and ships them with SAS software. Not all SAS sites, however, install this autocall macro library, and some autocall macro programs can be site-specific as well.

These macro programs can be used like macro functions in your macro programming. Many of the functions perform actions comparable to their similarly named SAS language counterpart. For example, one autocall macro program is %LOWCASE. This

autocall macro program converts alphabetic characters in its argument to lowercase. Similarly, you could use %SYSFUNC and the LOWCASE SAS language function to do the same action. Chapter 10 discusses how you can save your own macro programs in your own autocall libraries.

Table 6.7 lists several of the autocall macro programs. Autocall macro programs %CMPRES, %LEFT, and %LOWCASE each have a version that you should use if the result could contain a special character or mnemonic operator. The names of those autocall macro programs are: %QCMPRES, %QLEFT, and %QLOWCASE.

Table 6.7 Selected SAS supplied autocall macro programs

Function	Action
%CMPRES(*text | text expression*)	Remove multiple, leading, and trailing blanks from the argument. Use %QCMPRES if the result might contain a special character or mnemonic operator.
%DATATYP(*text | text expression*)	Returns the data type (CHAR or NUMERIC) of a value.
%LEFT(*text | text expression*)	Aligns an argument to the left by removing leading blanks. Use %QLEFT if the result might contain a special character or mnemonic operator.
%LOWCASE(*text | text expression*)	Changes a value from uppercase characters to lowercase. Use %QLOWCASE if the result might contain a special character or mnemonic operator.
%VERIFY(*source | excerpt*)	Returns the position of the first character unique to an expression.

Example 6.10: Determining with %VERIFY and %UPCASE If a Value Is in a Defined Set of Characters

This example examines the value of a macro variable to see if its value is a valid response to a survey question. Macro program CHECKSURVEY in Program 6.10a has one parameter, RESPONSE. The value of RESPONSE is examined, and the result of the examination is printed in the SAS log with the %PUT statement. A response to a survey question in this example must be a digit from 1 to 5 or 9, or a letter from A to E or Z.

The example converts the survey response value to uppercase with the %UPCASE macro function. It then examines the value with the %VERIFY autocall macro program. The %VERIFY autocall macro program returns the position in a text value of the first character that is not in the list supplied as the second argument. In this example, assume that survey responses are single characters, and only single characters will be specified as parameters to CHECKSURVEY. Therefore, the %VERIFY function in this example can only return one of two values: zero (0) for a valid response and one (1) for an invalid response. The returned value of 1 corresponds to the first and only character specified as the parameter to CHECKSURVEY.

Since the value of macro variable RESPONSE is converted to uppercase, only the uppercase letters of the alphabet are specified in the list of valid responses assigned to the macro variable VALIDRESPONSES. Note that the string of valid values is not enclosed in quotation marks. Since %VERIFY is a macro program, it treats all values as text and therefore enclosing quotation marks are not specified as they would be when processing text in SAS language statements.

Program 6.10a

```
%macro checksurvey(response);
  %let validresponses=123459ABCDEZ;
  %let result=%verify(%upcase(&response),&validresponses);
  %put ******* Response &response is valid/invalid (0=valid
1=invalid): &result;
%mend checksurvey;

%checksurvey(f)
%checksurvey(a)
%checksurvey(6)
```

After submitting the preceding three calls to macro program CHECKSURVEY, the following is written to the SAS log:

```
175  %checksurvey(f)
******* Response f is valid/invalid (0=valid 1=invalid): 1
```

```
176  %checksurvey(a)
******* Response a is valid/invalid (0=valid 1=invalid): 0
177  %checksurvey(6)
******* Response 6 is valid/invalid (0=valid 1=invalid): 1
```

The same information can be obtained by using the %SYSFUNC macro function in conjunction with the VERIFY and UPCASE SAS language functions. Program 6.10a is revised in Program 6.10b to use %SYSFUNC, VERIFY, and UPCASE.

Note that two calls to %SYSFUNC are made, once for each of the two SAS language functions. As mentioned at the beginning of this section, you cannot nest multiple calls to SAS language functions within one call to %SYSFUNC, but you can nest multiple %SYSFUNC calls.

This example nests only one %SYSFUNC call. When you need to have multiple %SYSFUNC calls, it might be easier to step through the processing by specifying multiple %LET statements rather than trying to nest several calls on one %LET statement. Doing so can prevent frustrating debugging tasks as you figure out the proper positioning of all the parentheses, commas, and arguments.

The SAS language functions are underlined in Program 6.10b.

Program 6.10b

```
%macro checksurvey(response);
  %let validresponses=123459ABCDEZ;
  %let result=
%sysfunc(verify(%sysfunc(upcase(&response)),&validresponses));
  %put ******* Response &response is valid/invalid (0=valid
1=invalid): &result;
%mend checksurvey;

%checksurvey(f)
%checksurvey(a)
%checksurvey(6)
```

After submitting the above three macro language statements, the following is written to the SAS log:

```
193  %checksurvey(f)
******* Response f is valid/invalid (0=valid 1=invalid): 1
194  %checksurvey(a)
******* Response a is valid/invalid (0=valid 1=invalid): 0
195  %checksurvey(6)
******* Response 6 is valid/invalid (0=valid 1=invalid): 1
```

Chapter 7

Macro Expressions and Macro Programming Statements

Introduction 160

Macro Language Statements 160

Constructing Macro Expressions 163

- Understanding Arithmetic Expressions 164
- Understanding Logical Expressions 165
- Understanding the IN Operator As Used in Macro Language Statements 166

Conditional Processing with the Macro Language 167

Iterative Processing with the Macro Language 177

- Writing Iterative %DO Loops in the Macro Language 177
- Conditional Iteration with %DO %UNTIL 180
- Conditional Iteration with %DO %WHILE 182

Branching in Macro Processing 184

Introduction

This chapter presents information about macro expressions and macro language statements, and it shows you how to accomplish several programming techniques with them. This chapter also shows you how to construct expressions and use macro language statements to conditionally process SAS steps, branch to different sections of a macro program, and perform iterative processing. The examples in this chapter use several of the functions described in Chapter 6. Chapter 8 continues the discussion of macro programming techniques with the topic of applying quoting functions to mask special characters and mnemonic operators.

Macro Language Statements

Macro language statements communicate your instructions to the macro processor. With macro language statements, you can write macro programs that conditionally or repetitively execute sections of code. Many macro language statements have a SAS language counterpart. The syntax and function of similarly named statements are usually the same or very similar.

Remember, however, that macro language statements *build* SAS programs and are processed *before* the SAS programs they build. Macro language statements are *not* part of the DATA step programming language. They operate in a different context. They *write* SAS programs.

Macro language statements can be grouped into two types:

- statements that can be used either in open code or inside a macro program
- statements that can be used only inside a macro program

Tables 7.1 and 7.2 list most of the macro language statements. Some are shown by example in this chapter. Detailed reference information on these statements is in *SAS Macro Language: Reference*.

Table 7.3 lists the macro language statements that can be used to display windows and supply values to macro variables during macro execution, including prompting users for values. Discussion of this material is beyond the scope of this book. For detailed information on writing macro code that includes these statements, see *SAS Macro Language: Reference.*

As an aid in remembering the type of a macro language statement, observe that the statements in Table 7.1 work on macro variables or act as definition type statements. These statements can be used either in open code or inside a macro program.

On the other hand, most of the macro language statements in Table 7.2 are active programming statements that control processing and work in conjunction with other statements. These statements can be used only inside a macro program.

Table 7.1 Macro language statements that can be used either in open code or inside a macro program

Statement	Action
`%* comment;`	Add descriptive text to your macro code.
%COPY	Copy specified items from a SAS library.
%GLOBAL	Create macro variables that are stored in the global symbol table and that will be available throughout the SAS session.
%LET	Create a macro variable and/or assign it a value.
%MACRO	Begin the definition of a macro program.
%PUT	Write text or macro variable values to the SAS log.
%SYMDEL	Delete the specified macro variable(s) from the global symbol table.
%SYSCALL	Invoke a SAS or user-written call routine using macro variables as the arguments and generating text as the result (see also the %SYSFUNC macro function).
%SYSEXEC	Execute operating system commands immediately and return the result to automatic macro variable SYSRC.
%SYSLPUT	Create a new macro variable or modify the value of an existing macro variable on a remote host or server. Used with SAS/CONNECT.
%SYSRPUT	Assign the value of a macro variable on a remote host to a macro variable on the local host. Used with SAS/CONNECT.

Table 7.2 lists the macro language statements that can be used only inside a macro program.

Table 7.2 Macro language statements that can be used only inside a macro program

Statement	Action
%ABORT	Stop the macro program that is executing along with the current DATA step, SAS job, or SAS session.
%DO	Signal the beginning of a %DO group; the statements that follow form a block of code that is terminated with a %END statement.
%DO, iterative	Repetitively execute a section of macro code by using an index variable and the keywords %TO and %BY; the section of macro code is terminated with a %END statement.
%DO %UNTIL	Repetitively execute a section of macro code *until* the macro expression that follows the %UNTIL is true; the section of macro code is terminated with a %END statement.
%DO %WHILE	Repetitively execute a section of macro code *while* the macro expression that follows the %WHILE is true; the section of macro code is terminated with a %END statement.
%END	Terminate a %DO group.
%GOTO	Branch macro processing to the specified macro label within the macro program.
%IF-%THEN %ELSE	Conditionally process the section of macro code that follows the %THEN when the result of the macro expression that follows %IF is true. When the macro expression that follows %IF is false, the section of macro code that follows the %ELSE is executed.
%label:	Identify a section of macro code; typically used as the destination of a %GOTO statement.
%LOCAL	Create macro variables that are available only to the macro program in which the %LOCAL statement was issued.
%MEND	End a macro program that was created with a %MACRO statement.
%RETURN	Cause normal termination of the currently executing macro program.

Table 7.3 lists the macro language statements that can be used to display windows and supply values to macro variables during macro execution. These statements can be included in open code and inside macro programs.

Table 7.3 Macro language statements that can be used to display windows and prompt users for input

Statement	Action
%DISPLAY	Display a macro window.
%INPUT	Supply values to macro variables during macro execution.
%WINDOW	Define a customized window.

Constructing Macro Expressions

Previous chapters have shown examples of text expressions. A text expression is any combination of text, macro variables, macro functions, or macro program calls. This section describes two more types of macro expressions: arithmetic and logical.

Arithmetic and logical macro expressions are comprised of operators and operands that the macro processor evaluates to produce a result. Arithmetic expressions use arithmetic operators such as plus signs and minus signs. Logical expressions use logical operators such as greater than signs and equal signs.

Arithmetic and logical expressions in the macro language are constructed similarly to expressions in the SAS programming language. Both macro expressions and SAS language expressions use the same arithmetic and logical operators. The exceptions to this are that the MIN and MAX operators are not available in the macro language. The same precedence rules also apply. Parentheses act to group expressions and control the order of evaluation of expressions.

The section on macro evaluation functions in Chapter 6 presents several examples of arithmetic expressions when using %EVAL and %SYSEVALF. Subsequent sections in this chapter show examples that require arithmetic and logical expressions.

Table 7.4 lists the operators in order of precedence of evaluation of arithmetic and logical operators. The symbols for the NOT and NE operators depend on your computer system. Do not place percent signs in front of the mnemonic operators.

Table 7.4 Arithmetic and logical operators and their precedence in the macro language

Operator	Mnemonic	Action	Precedence Rating
**		exponentiation	1
+		positive prefix	2
-		negative prefix	2
^	NOT	logical not	3
*		multiplication	4
/		division	4
+		addition	5
-		subtraction	5
<	LT	less than	6
<=	LE	less than or equal to	6
=	EQ	equal	6
#	IN	equal to one of a list*	6
^=	NE	not equal	6
>	GT	greater than	6
>=	GE	greater than or equal to	6
&	AND	logical and	7
|	OR	logical or	8

*Note that IN is available starting in SAS 9.2.

Understanding Arithmetic Expressions

Since the macro language is a text-based language, working with numbers is the exception. Therefore, special considerations are needed when you write expressions that use numbers where you want to perform calculations with them.

The macro evaluation functions described in Chapter 6, %EVAL and %SYSEVALF, temporarily convert their arguments to numbers in order to resolve arithmetic expressions.

Several macro language statements and functions require numeric or logical expressions. These elements automatically invoke the %EVAL function to convert the expressions from text.

The macro functions that automatically invoke %EVAL around the expressions supplied to them are

- %SUBSTR and %QSUBSTR
- %SCAN and %QSCAN

The macro language statements that automatically invoke %EVAL around the expressions supplied to them are

- %DO
- %DO %UNTIL
- %DO %WHILE
- %IF

Therefore, when you use these functions and statements, explicitly coding the %EVAL function around the macro arithmetic expression is redundant.

Refer to the section in Chapter 6 on macro evaluation functions for examples of arithmetic macro expressions. Many examples in this chapter include arithmetic expressions as well.

Understanding Logical Expressions

A logical expression in the macro language compares two macro expressions. These macro expressions consist of text, macro variables, macro functions, arithmetic expressions, and other logical expressions. If the comparison is true, the result is a value of one (1). If the comparison is false, the result is zero (0). Expressions that resolve to integers other than zero (0) are also considered true. Expressions that resolve to zero (0) are false. The comparison operators in Table 7.4 construct logical expressions in the macro language.

As the macro processor resolves a macro expression, it places a %EVAL around each of the operands in the expression to temporarily convert the operands to integers. If an operand cannot be an integer, the macro facility then treats all operands in the expression as text. Comparisons are then based on the sort sequence of characters in the host operating system.

When you want numbers with decimal points to be compared as numbers and not compared as text, place the %SYSEVALF function around the logical expression. The %SYSEVALF function with the BOOLEAN conversion type acts like a logical expression because it yields a true-false result of one (1) or zero (0). Logical expressions are used in conditional processing. Examples of logical expressions and conditional processing are provided in the next section.

Understanding the IN Operator As Used in Macro Language Statements

As you write your macro programs, you will have situations where you want to execute a section of code when a macro variable can have any one of the values in a set of values. Testing for that involves writing the multiple conditions and connecting them with the OR operator:

```
%if &month=JANUARY or &MONTH=APRIL or &MONTH=AUGUST or
   &MONTH=DECEMBER %then %do;
… statements to execute when one of the conditions is true…
%end;
```

Beginning with SAS 9.2, you can simplify the %IF statement by using the IN operator and following the IN operator with the list of acceptable values:

```
%if &month in JANUARY APRIL AUGUST DECEMBER %then %do;
… statements to execute when one of the conditions is true…
%end;
```

The way you specify your list of values depends on the value of the SAS option MINDELIMITER. This can be set either with the OPTIONS statement or as an option in the %MACRO statement when you define your macro program. A value specified for MINDELIMITER on the %MACRO statement overrides the value of the MINDELIMITER= SAS option for the duration of execution of the macro program. The default value for MINDELIMITER is a blank, and this default value is used in the %IF statement above.

An example of specifying a %MACRO statement with the MINDELIMITER= option follows where the delimiter is a comma (,). The delimiter must be a single character enclosed in single quotation marks.

```
%macro lists(author) / mindelimiter=',';
```

Also, the SAS system option MINOPERATOR | NOMINOPERATOR becomes available with SAS 9.2. This option controls whether the word "IN" (case insensitive) or special symbol # is recognized by the SAS Macro Facility as an infix operator when evaluating logical or integer expressions.

Conditional Processing with the Macro Language

A basic feature of any programming language is conditional execution of code. SAS macro language uses %IF-%THEN/%ELSE statements to control execution of sections of code. The sections of code that can be selected include macro language statements or text.

Remember that text in the macro facility can be SAS language statements like DATA steps and PROC steps. Thus, within your macro programs, based on evaluation of conditions you set, you can direct the macro processor to submit specific SAS statements for execution. With this capability, one macro program can contain many SAS language statements and steps and can be used repeatedly to manage various processing tasks.

The syntax of the %IF statement is

```
%IF expression %THEN action;
<%ELSE elseaction;>
```

Multiple %ELSE statements can be specified to test for multiple conditions. The expression that you write is usually a logical expression. The macro processor invokes the %EVAL function around the expression and resolves the expression to true or false. When the evaluation of the expression is true, *action* is executed. When the evaluation of the expression is false and a %ELSE statement is specified, *elseaction* is executed.

Example 7.1: Using Logical Expressions

This example illustrates evaluation of logical expressions. Macro program COMP2VARS in Program 7.1 has two parameters. Four different types of logical expression evaluations that compare the two parameters are made with each call to COMP2VARS. Program 7.1 calls macro program COMP2VARS three times.

Note that the sort sequence of your operating system determines the outcome. These examples were run under Windows where in ASCII a lowercase letter comes before an uppercase letter; in EBCDIC, uppercase letters sort before lowercase letters.

Program 7.1

```
%macro comp2vars(value1,value2);
  %put COMPARISON 1:;
  %if &value1 ne &value2 %then
    %put &value1 is not equal to &value2..;
  %else %put &value1 equals &value2..;

  %put COMPARISON 2:;
  %if &value1 > &value2 %then
    %put &value1 is greater than &value2..;
  %else %if &value1 < &value2 %then
    %put &value1 is less than &value2..;
  %else %put &value1 equals &value2..;

  %put COMPARISON 3:;
  %let result=%eval(&value1 > &value2);
  %if &result=1 %then
    %put EVAL result of &value1 > &value2 is TRUE.;
  %else %put EVAL result of &value1 > &value2 is FALSE.;

  %put COMPARISON 4:;
  %let result=%sysevalf(&value1 > &value2);
  %if &result=1 %then
    %put SYSEVALF result of &value1 > &value2 is TRUE.;
  %else %put SYSEVALF result of &value1 > &value2 is FALSE.;

%mend comp2vars;

*----First call to COMP2VARS;
%comp2vars(3,4)

*----Second call to COMP2VARS;
%comp2vars(3.0,3)

*----Third call to COMP2VARS;
%comp2vars(X,x)
```

The SAS log for % COMP2VARS (3,4) follows.

```
63   %comp2vars(3,4)
COMPARISON 1:
3 is not equal to 4
COMPARISON 2:
3 is less than 4.
```

```
COMPARISON 3:
EVAL result of 3 > 4 is FALSE.
COMPARISON 4:
SYSEVALF result of 3 > 4 is FALSE.
```

The SAS log for % COMP2VARS (3.0,3) follows.

```
65   %comp2vars(3.0,3)
COMPARISON 1:
3.0 is not equal to 3
COMPARISON 2:
3.0 is greater than 3.
COMPARISON 3:
EVAL result of 3.0 > 3 is TRUE.
COMPARISON 4:
SYSEVALF result of 3.0 > 3 is FALSE.
```

The SAS log for % COMP2VARS (X,x) follows.

```
67  %comp2vars(X,x)
COMPARISON 1:
X is not equal to x.
COMPARISON 2:
X is less than x.
COMPARISON 3:
EVAL result of X > x is FALSE.
COMPARISON 4:
SYSEVALF result of X > x is FALSE.
```

Example 7.2: Using Macro Language to Select SAS Steps for Processing

Program 7.2 shows how you can instruct the macro processor to select certain SAS steps. Macro program REPORTS contains code for two types of reports: a summary report and a detail report. The first parameter, REPTYPE, determines which of the two types of reports should be produced.

The expected values for REPTYPE are either SUMMARY or DETAIL. The second parameter, REPMONTH, is to be specified as the numeric value of the month for which to produce the report.

When REPTYPE is specified as SUMMARY, the first PROC TABULATE step executes. If REPMONTH is equal to the last month of a quarter (March, June, September, or December), then the second PROC TABULATE step executes.

When REPTYPE is specified as DETAIL, the PROC TABULATE steps are skipped and only the PROC PRINT step in the %ELSE section executes.

This example calls macro program REPORTS twice. The first call to REPORTS requests a summary report for September. Both PROC TABULATE steps execute since September is the last month in the third quarter.

The second call to REPORTS requests a detail report for October. Macro program REPORTS executes a PROC PRINT step that lists the detailed information for October.

Program 7.2

```
%macro reports(reptype,repmonth);
  %let lblmonth=
    %sysfunc(mdy(&repmonth,1,%substr(&sysdate,6,2)),monname.);

  %*----Begin summary report section;
  %if %upcase(&reptype)=SUMMARY %then %do;
    %*----Do summary report for report month;
    proc tabulate data=books.ytdsales;
      title "Sales for &lblmonth";
      where month(datesold)=&repmonth;
      class section;
      var listprice saleprice;
      tables section,
        (listprice saleprice)*(n*f=6. sum*f=dollar12.2);
    run;
    %*----If end of quarter, also do summary report for qtr;
    %if &repmonth=3 or &repmonth=6 or &repmonth=9
        or &repmonth=12 %then %do;
      %let qtrstart=%eval(&repmonth-2);

      %let strtmo=
     %sysfunc(mdy(&qtrstart,1,%substr(&sysdate,6,2)),monname.);

      proc tabulate data=books.ytdsales;
        title "Sales for Quarter from &strtmo to &lblmonth";
        where &qtrstart le month(datesold) le &repmonth;
        class section;
        var listprice saleprice;
        tables section,
          (listprice saleprice)*(n*f=6. sum*f=dollar12.2);
      run;
     %end;
  %end;
  %*----End summary report section;
  %*----Begin detail report section;
  %else %if %upcase(&reptype)=DETAIL %then %do;
    %*----Do detail report for month;
    proc print data=books.ytdsales;
      where month(datesold)=&repmonth;
```

```
      var booktitle cost listprice saleprice;
      sum cost listprice saleprice;
    run;
  %end;
  %*----End detail report section;
%mend reports;

*----First call to REPORTS does a Summary report for September;
%reports(Summary,9)

*----Second call to REPORTS does a Detail report for October;
%reports(Detail,10)
```

The first call to REPORTS specifies summary reports for September. Macro program REPORTS submits the following code, which is shown after resolution of the macro variables.

```
proc tabulate data=books.ytdsales;
  title "Sales for September";
  where month(datesold)=9;
  class section;
  var listprice saleprice;
  tables section,(listpric saleprice)*(n*f=6.
sum*f=dollar12.2);
run;
proc tabulate data=books.ytdsales;
  title "Sales for Quarter from July to September";
  where 7 le month(datesold) le 9;
  class section;
  var listprice salepric;
  tables section,(listprice saleprice)*(n*f=6.
sum*f=dollar12.2);
run;
```

The second call to REPORTS specifies a detail report for October. Macro program REPORTS submits the following code, which is shown after resolution of the macro variables.

```
proc print data=books.ytdsales;
  where month(datesold)=10;
  var title cost listprice saleprice;
  sum cost listprice saleprice;
run;
```

Example 7.3: Using %IF-%THEN/%ELSE Statements to Modify and Select Statements within a Step

Example 7.3 shows how %IF-%THEN/%ELSE statements can select the statements *within* a step to submit for processing. The previous example (Example 7.2) selected different steps, but did not select different statements within the step.

Macro program PUBLISHERREPORT in Program 7.3 constructs a PROC REPORT step that summarizes information about publishers. It has one parameter REPTYPE that can take one of three values: BASIC, DETAIL, and QUARTER. These values each specify a different report by requesting display and computation of different columns in the PROC REPORT step.

All three reports list the values of data set variables PUBLISHER and SALEPRICE. Following is a description of the actions that the macro program takes for each of the three possible parameter values.

REPTYPE=BASIC: Compute and display PROFIT for each value of PUBLISHER and overall. Do not display COST, but use it in the COMPUTE block to compute the value of PROFIT. Specify option NOPRINT on the DEFINE statement for COST.

REPTYPE=DETAIL: Compute and display PROFIT for each value of PUBLISHER and overall. Compute the N statistic and label this column "Number of Titles Sold." Display COST and use it in the COMPUTE block to calculate the value of PROFIT.

REPTYPE=QUARTER: Compute and display PROFIT for each value of PUBLISHER and overall. Display COST and use it in the COMPUTE block to compute the value of PROFIT. Define DATESOLD as an ACROSS variable and format the values of DATESOLD as calendar quarters. Define SALEPRICE2 as an alias for SALEPRICE and nest SALEPRICE2 underneath DATESOLD. Underneath each of the four values displayed for DATESOLD, display the sum of SALEPRICE2. These columns are the totals of SALEPRICE for each quarter.

A FOOTNOTE statement displays information about the processing. It prints the name of the macro program using automatic macro variable &SYSMACRONAME, and it lists the value of parameter REPTYPE.

Program 7.3 calls macro program PUBLISHERREPORT three times, once for each of the three valid values of REPTYPE. The first %LET statement in the macro program converts the value of REPTYPE to uppercase, making coding of the %IF statement easier so that only one possible value has to be examined.

The macro language statements that select SAS language code are in bold.

Program 7.3

```
%macro publisherreport(reptype);
  %let reptype=%upcase(&reptype);

  title "Publisher Report";
  footnote
    "Macro Program: &sysmacroname  Report Type: &reptype";

  proc report data=books.ytdsales nowd headline;
    column publisher saleprice cost profit
      %if &reptype=DETAIL %then %do;
        n
      %end;
      %else %if &reptype=QUARTER %then %do;
          datesold,(saleprice=saleprice2)
      %end;
     ;

    define publisher / group width=25;
    define saleprice / analysis sum format=dollar11.2;

    define cost / analysis sum format=dollar11.2
                  %if &reptype=BASIC %then %do;
                    noprint
                  %end;
                  ;
    define profit / computed format=dollar11.2 'Profit';

    %if &reptype=DETAIL %then %do;
      define n / 'Number of Titles Sold' width=6;
    %end;
    %else %if &reptype=QUARTER %then %do;
      define saleprice2 / 'Quarter Sale Price Total';
      define datesold / across ' ' format=qtr.;
    %end;

    compute profit;
      profit=saleprice.sum-cost.sum;
    endcomp;

    rbreak after / summarize dol;
    compute after;
      publisher='Total for All Publishers';
    endcomp;

  run;
%mend publisherreport;
```

```
%* First call to PUBLISHERREPORT, do BASIC report;
%publisherreport(basic)

%* Second call to PUBLISHERREPORT, do DETAIL report;
%publisherreport(detail)

%* Third call to PUBLISHERREPORT, do QUARTER report;
%publisherreport(quarter)
```

First call to PUBLISHERREPORT: The PROC REPORT step that PUBLISHERREPORT submits when REPTYPE=BASIC follows. The features unique to the version specified by REPTYPE=BASIC are in bold.

```
title "Publisher Report";
footnote "Macro Program: PUBLISHERREPORT  Report Type: BASIC";
proc report data=books.ytdsales nowd headline;
  column publisher saleprice cost profit;

  define publisher / group width=25;
  define saleprice / analysis sum format=dollar11.2;
  define cost / analysis sum format=dollar11.2 noprint;
  define profit / computed format=dollar11.2 'Profit';

  compute profit;
    profit=saleprice.sum-cost.sum;
  endcomp;

  rbreak after / summarize dol;
  compute after;
    publisher='Total for All Publishers';
  endcomp;
run;
```

Second call to PUBLISHERREPORT: The PROC REPORT step that PUBLISHERREPORT submits when REPTYPE=DETAIL follows. The features unique to the version specified by REPTYPE=DETAIL are in bold.

```
title "Publisher Report";
footnote
  "Macro Program: PUBLISHERREPORT  Report Type: DETAIL";
proc report data=books.ytdsales nowd headline;
  column publisher saleprice cost profit n;

  define publisher / group width=25;
  define saleprice / analysis sum format=dollar11.2;
  define cost / analysis sum format=dollar11.2;
```

```
  define profit / computed format=dollar11.2 'Profit';
  define n / 'Number of Titles Sold' width=6;

  compute profit;
    profit=saleprice.sum-cost.sum;
  endcomp;

  rbreak after / summarize dol;
  compute after;
    publisher='Total for All Publishers';
  endcomp;
run;
```

Third call to PUBLISHERREPORT: The PROC REPORT step that PUBLISHERREPORT submits when REPTYPE=QUARTER follows. The features unique to the version specified by REPTYPE=QUARTER are in bold.

```
title "Publisher Report";
footnote "Macro Program: PUBLISHERREPORT Report Type: QUARTER";
proc report data=books.ytdsales nowd headline;
  column publisher saleprice cost profit
         datesold,(saleprice=saleprice2);

  define publisher / group width=25;
  define saleprice / analysis sum format=dollar11.2;
  define cost / analysis sum format=dollar11.2 ;
  define profit / computed format=dollar11.2 'Profit';
  define saleprice2 / 'Quarter Sale Price Total';
  define datesold / across ' ' format=qtr.;

  compute profit;
    profit=saleprice.sum-cost.sum;
  endcomp;

  rbreak after / summarize dol;
  compute after;
    publisher='Total for All Publishers';
  endcomp;
run;
```

Example 7.4: Writing %IF-%THEN/%ELSE Statements That Use the IN Operator

Macro program VENDORTITLES in Program 7.4 defines a TITLE2 statement based on the value of the parameter PUBLISHER. Assume multiple publishers use the same vendor. Rather than writing multiple logical expressions on the %IF statement and connecting them with the OR operator, this example uses the IN operator, which is available starting in SAS 9.2. The multiple publishers mapping to one vendor are listed after the IN operator, and names of the publishers are separated by exclamation points.

The exclamation point delimiter is specified on the %MACRO statement for VENDORTITLES with the MINDELIMITER= option. This specification overrides the current setting of the MINDELIMITER= SAS option during execution of the macro program.

If the MINDELIMITER= option was omitted in this example, the macro program would not execute correctly unless the exclamation point delimiter had been previously specified with the SAS OPTIONS statement.

Note that the MINOPERATOR SAS option must be in effect as well when Program 7.4 is submitted. This option available with SAS 9.2 controls whether the word "IN" (case insensitive) or special symbol # is recognized by the SAS Macro Facility as an infix operator when evaluating logical or integer expressions.

Program 7.4

```
%macro vendortitles(publisher) / mindelimiter='!';
  title "Vendor-Publisher Report";
  %if &publisher in
           AMZ Publishers!Eversons Books!IT Training Texts
        %then %do;
    title2 "Vendor for &publisher is Baker";
  %end;
  %else %if &publisher in
          Northern Associates Titles!Professional House Titles
              %then %do;
    title2 "Vendor for &publisher is Mediasuppliers";
  %end;
  %else %do;
    title2 "Vendor for &publisher is Basic Distributor";
  %end;
%mend vendortitles;

%vendortitles(AMZ Publishers)

%vendortitles(Mainst Media)
```

The first call to VENDORTITLES defines the following TITLE2 statement:

```
title2 "Vendor for AMZ Publishers is Baker";
```

The second call to VENDORTITLES defines the following TITLE2 statement:

```
title2 "Vendor for Mainst Media Publishers is Basic
        Distributor";
```

Iterative Processing with the Macro Language

The iterative processing statements in the macro language instruct the macro processor to repetitively process sections of code. The macro language includes %DO loops, %DO %UNTIL loops, and %DO %WHILE loops. With iterative processing, you can instruct the macro processor to write many SAS language statements, DATA steps, and PROC steps. The three types of iterative processing statements are described below. These statements can be used only from within a macro program.

Writing Iterative %DO Loops in the Macro Language

The iterative %DO macro language statement instructs the macro processor to execute a section of code repeatedly. The number of times the section executes is based on the value of an index variable. The index variable is a macro variable. You define the start value and stop value of this index variable. You can also control the increment of the steps between the start value and the stop value; by default, the increment value is one.

The syntax of an iterative %DO loop is as follows.

```
%DO macro-variable=start %TO stop <%BY increment>;
  macro language statements and/or text
%END;
```

Do not put an ampersand in front of the index variable name in the %DO statement even though the index variable is a macro variable. Any reference to it later within the loop, however, requires an ampersand in front of the reference.

The start and stop values are integers or macro expressions that can be resolved to integers. If you want to increment the index macro variable by something other than one, follow the stop value with the %BY keyword and the increment value. The increment value is either an integer or a macro expression that can be resolved to an integer.

Example 7.5: Building PROC Steps with Iterative %DO Loops

Program 7.5 uses the iterative %DO to generate several PROC MEANS and PROC GCHART steps. Macro program MULTREP generates statistics and a bar chart for each year between the bounds on the %DO statement. In this example, PROC MEANS and PROC GCHART are each executed three times: once for 2005, once for 2006, and once for 2007.

Program 7.5

```
%macro multrep(startyear,stopyear);
  %do yrvalue=&startyear %to &stopyear;
    title "Sales Report for &yrvalue";
    proc means data=sales.year&yrvalue;
      class section;
      var cost listprice saleprice;
    run;

    proc gchart data=sales.year&yrvalue;
      hbar section / sumvar=saleprice type=sum;
    run;
    quit;
  %end;
%mend multrep;

*----Produce 3 sets of reports: one for 2005, one for 2006,
*----and one for 2007;
%multrep(2005,2007)
```

After the macro processor processes the macro language statements and resolves the macro variables references, the following SAS program is submitted.

```
title "Sales Report for 2005";
proc means data=sales.year2005;
  class section;
  var cost listprice saleprice;
run;

proc gchart data=sales.year2005;
  hbar section / sumvar=saleprice type=sum;
run;

title "Sales Report for 2006";
proc means data=sales.year2006;
  class section;
  var cost listprice saleprice;
run;
```

```
proc gchart data=sales.year2006;
  hbar section / sumvar=saleprice type=sum;
run;

title "Sales Report for 2007";
proc means data=sales.year2007;
  class section;
  var cost listprice saleprice;
run;

proc gchart data=sales.year2007;
  hbar section / sumvar=saleprice type=sum;
run;
```

Example 7.6: Building SAS Statements within a Step with Iterative %DO Loops

Iterative %DO statements can build SAS statements within a SAS DATA step or SAS PROC step. Macro program SUMYEARS in Program 7.6 concatenates several data sets in a DATA step. The first %DO loop constructs the names of the data sets that the DATA step concatenates.

Note that a semicolon is not placed after the reference to the data set within the first %DO loop. If a semicolon was placed after the data set reference, the semicolon would terminate the SET statement on the first iteration. On each subsequent iteration, a semicolon after the data set reference would make the data set reference look like a SAS statement, which results in errors.

The second %DO loop creates the macro variable YEARSTRING that contains the values of all the processing years. Each iteration of the second %DO loop concatenates the current iteration's value for YEARVALUE to the current value of YEARSTRING.

Program 7.6

```
%macro sumyears(startyear,stopyear);
  data manyyears;
    set
    %do yearvalue=&startyear %to &stopyear;
      sales.year&yearvalue
    %end;
    ;
  run;

  %let yearstring=;
  %do yearvalue=&startyear %to &stopyear;
    %let yearstring=&yearstring &yearvalue;
  %end;
```

```
  proc gchart data=manyyears;
    title "Charts Analyze Data for: &yearstring";
    hbar section / sumvar=saleprice type=sum;
  run;
  quit;
%mend sumyears;

*----Concatenate three data sets: one from 2005, one from;
*----2006, and one from 2007;
%sumyears(2005,2007)
```

The macro processor resolves the call to YEARLYCHARTS as follows.

```
data manyyears;
  set sales.year2005 sales.year2006 sales.year2007;
run;

proc gchart data=manyyears;
  title "Charts Analyze Data for: 2005 2006 2007";
  hbar section / sumvar=saleprice type=sum;
run;
quit;
```

Conditional Iteration with %DO %UNTIL

With %DO %UNTIL, a section of code is executed *until* the condition on the %DO %UNTIL statement is true. The syntax of %DO %UNTIL is

```
%DO %UNTIL (expression);
  macro language statements and/or text
%END;
```

The expression on the %DO %UNTIL statement is a macro expression that resolves to a true-false value. The macro processor evaluates the expression at the bottom of each iteration. Therefore, a %DO %UNTIL loop always executes at least once.

Example 7.7: Building SAS Steps with %DO %UNTIL Loops

This example demonstrates the use of %DO %UNTIL. Macro program MOSALES defined in Program 7.7 computes statistics for each month in the list of values passed to the program. When a list of month values is not specified, MOSALES computes statistics for all observations in the analysis data set.

Program 7.7 defines macro program MOSALES with the PARMBUFF option. This %MACRO statement option is described at the end of Chapter 4. The PARMBUFF option allows you to specify a varying number of parameter values. The macro processor assigns the list of values to the automatic macro variable SYSPBUFF. Macro program

MOSALES parses SYSPBUFF and submits a PROC MEANS step for each month value specified in the list of parameter values.

The %SCAN function selects each month value from SYSPBUFF. The macro variable LISTINDEX determines which item in the list of months the %SCAN function should select, and the program increments it by one at the bottom of the %DO %UNTIL loop. Observations are selected for processing with a WHERE statement.

When a list of parameter values is not specified, as in the second call to MOSALES, the macro program does an overall PROC MEANS step and does not apply a WHERE statement to the step. This overall PROC MEANS is accomplished by taking advantage of the features of %DO %UNTIL: a %DO %UNTIL loop executes at least once. When parameter values are not specified, the following occurs:

- The %SCAN function is not able to extract any text from SYSPBUFF so the result of the evaluation of the %DO %UNTIL condition is true.
- The value of REPMONTH is assigned a null value.
- The code within the %DO %UNTIL loop executes once.
- The first TITLE statement and the WHERE statement do not execute because REPMONTH is null.

Program 7.7 calls MOSALES twice. The first call to MOSALES submits three PROC MEANS steps: one for March, one for May, and one for October. The second call to MOSALES submits one PROC MEANS step, a summarization of all the observations in the data set.

Program 7.7

```
%macro mosales / parmbuff;
  %let listindex=1;
  %do %until (%scan(&syspbuff,&listindex) eq );
    %let repmonth=%scan(&syspbuff,&listindex);
    proc means data=books.ytdsales n sum;
      %if &repmonth ne %then %do;
        title "Sales during month &repmonth";
        where month(datesold)=&repmonth;
      %end;
      %else %do;
        title "Overall Sales";
      %end;
      class section;
      var saleprice;
    run;
    %let listindex=%eval(&listindex+1);
  %end;
%mend;
```

```
*----First call to MOSALES: produce stats for March, May, and
*----October;
%mosales(3 5 10)

*----Second call to MOSALES: produce overall stats;
%mosales()
```

The first call to MOSALES requests statistics for March, May, and October. The macro processor generates the following SAS program.

```
proc means data=books.ytdsales n sum;
  title "Sales during month 3";
  where month(datesold)=3;
  class section;
  var saleprice;
run;
proc means data=books.ytdsales n sum;
  title "Sales during month 5";
  where month(datesold)=5;
  class section;
  var saleprice;
run;
proc means data=books.ytdsales n sum;
  title "Sales during month 10";
  where month(datesold)=10;
  class section;
  var saleprice;
run;
```

The second call to MOSALES does not specify any months. Therefore, the %DO %UNTIL loop executes once, generates overall statistics, and selects the second TITLE statement. The SAS program that the macro processor creates from this call follows:

```
proc means data=books.ytdsales n sum;
  title "Overall Sales";
  class section;
  var saleprice;
run;
```

Conditional Iteration with %DO %WHILE

With %DO %WHILE, a section of code is executed *while* the condition on the %DO %WHILE statement is true. The syntax of %DO %WHILE is:

```
%DO %WHILE (expression);
  macro language statements and/or text
%END;
```

The expression on the %DO %WHILE statement is a macro expression that resolves to a true-false value. The macro processor evaluates the expression at the top of the loop. Therefore, it is possible that a %DO %WHILE loop does not execute. This occurs when the condition starts out as false.

Example 7.8: Building SAS Steps with %DO %WHILE Loops

This example shows an application of %DO %WHILE. Macro program STAFFSALES defined in Program 7.8 computes sales statistics for specific sales associates during a specific month. It has two parameters: SALESREPS and REPMONTH.

The parameter SALESREPS is defined to be a list of the initials of the sales associates for whom to compute sales statistics. The second parameter, REPMONTH, is the month for which to compute the statistics. The program is written to expect only one value for REPMONTH, and it is assumed that it will be a number between one and twelve.

This example's call to STAFFSALES requests statistics for three sales associates for May. The %DO %WHILE loop executes three times, once for each associate. The %SCAN function selects each sales associate's initials from SALESREPS. The macro variable PERSONNUMBER determines which set of initials the %SCAN function should select, and the program increments PERSONNUMBER by one at the bottom of the %DO %WHILE loop.

The %DO %WHILE loop does not execute a fourth time. On the fourth iteration, the %SCAN function does not find initials for a fourth sales associate. Therefore, the macro expression on the %DO %WHILE statement resolves to false, and this causes the loop to stop executing. The one call to STAFFSALES in Program 7.8 generates sales reports for three sales associates during May.

Program 7.8

```
%macro staffsales(salesreps,repmonth);
  %let personnumber=1;
  %do %while (%scan(&salesreps,&personnumber) ne );
    %let salesinits=%scan(&salesreps,&personnumber);
    proc means data=books.ytdsales n sum;
      title "Sales for &salesinits during month &repmonth";
      where saleinit="&salesinits" and
                 month(datesold)=&repmonth;
      class section;
      var saleprice;
    run;
    %let personnumber=%eval(&personnumber+1);
  %end;
%mend staffsales;

%staffsales(MJM BLT JMB,5)
```

After resolution by the macro processor, the SAS code submitted for compilation and execution is as follows. Three PROC MEANS steps are created: one for each of the three sales associates.

```
proc means data=books.ytdsales n sum;
  title "Sales for MJM during month 5";
  where saleinit="MJM" and month(datesold)=5;
  class section;
  var saleprice;
run;

proc means data=books.ytdsales n sum;
  title "Sales for BLT during month 5";
  where saleinit="BLT" and month(datesold)=5;
  class section;
  var saleprice;
run;

proc means data=books.ytdsales n sum;
  title "Sales for JMB during month 5";
  where saleinit="JMB" and month(datesold)=5;
  class section;
  var saleprice;
run;
```

Since the %DO %WHILE loop executes only while the condition on the statement is true, consider what happens if no sales initials are specified on the call to %STAFFSALES as follows:

```
%staffsales(,5)
```

The %DO %WHILE loop in this situation does not execute because the condition on the %DO %WHILE statement is never true. No processing is done and no messages are written to the SAS log.

Branching in Macro Processing

When you want to branch to a different section of a macro program, label the text and use a %GOTO statement. The %GOTO statement directs processing to that labeled text. Labeled text and the %GOTO statement are allowed only in macro programs. Macro language statements, macro expressions, and constant text can be labeled. Macro text is labeled as follows:

```
%label: macro-text
```

Place the label before the macro text that you want to identify. The label is any valid SAS name. Precede the label with a percent sign (%) and follow the label name with a colon (:). The colon tells SAS to treat *%label* as a statement label and not the invocation of a macro program named *%label*.

The syntax of the %GOTO statement is

```
%GOTO label;
```

On the %GOTO statement, you can specify the label as text or as a macro expression that resolves to the label name. Do not put a percent sign in front of the label on the %GOTO statement. If you do specify a percent sign, the macro processor interprets that as a request to execute a macro program that has the name of your label.

Example 7.9: Using %GOTO to Branch in a Macro Program

The following example shows how labels and %GOTO statements can be used. Macro program DETAIL defined in Program 7.9 starts out by determining if the data set named by its first parameter, DSNAME, exists. If it does, it executes a PROC PRINT step listing the variables specified by the second parameter, VARLIST. When the step ends, the program branches to the label %FINISHED.

If the data set specified by DSNAME does not exist, the program skips over the PROC PRINT step and branches to the label %NODATASET. The program then writes a message to the SAS log, determines the libref of the data set specified by DSNAME, and executes a PROC DATASETS step that lists the data sets in the library specified by the data set's libref. The output from PROC DATASETS might help in figuring out the problem in specifying a value for DSNAME.

This example calls DETAIL three times. The code that executes is described below.

Program 7.9

```
%macro detail(dsname,varlist);
  %* Does DSNAME exist?;
  %let foundit=%sysfunc(exist(&dsname));
  %if &foundit le 0 %then %goto nodataset;

  title "PROC PRINT of &dsname";
  proc print data=&dsname;
    var &varlist;
  run;
  %goto finished;
```

```
  %nodataset:
    %put ERROR: **** Data set &dsname not found. ****;
    %put;
    %* Find the data set libref. If it is not;
    %* specified, assume a temporary data set;
    %* and assign WORK to DSLIBREF;
    %let period=%index(&dsname,.);
    %if &period gt 0 %then
       %let dslibref=%scan(&dsname,1,.);
    %else %let dslibref=work;
    proc datasets library=&dslibref details;
    run;
    quit;

  %finished:
%mend detail;

*----First call to DETAIL, data set exists;
%detail(books.ytdsales,datesold booktitle saleprice)

*----Second call to DETAIL, data set does not exist;
%detail(books.ytdsaless,datesold booktitle saleprice)

%*----Third call to DETAIL, look for data set in WORK library;
%detail(ytdsales,datesold booktitle saleprice)
```

First call to DETAIL: The first call to the macro program DETAIL executes a PROC PRINT of the data set since the data set exists. The PROC PRINT step lists the variables specified in VARLIST. After completion of the step, the program skips over the section labeled as %NODATASET and branches to the section labeled %FINISHED. The macro processor generates the following code:

```
title "PROC PRINT of books.ytdsales";
proc print data=books.ytdsales;
  var datesold booktitle saleprice;
run;
```

Second call to DETAIL: The data set name is misspelled in the second call to DETAIL. Assume a data set with this misspelled name does not exist in the library specified by BOOKS. The program skips the PROC PRINT section and executes the section labeled with %NODATASET. The macro processor writes an error message in red to the SAS log that data set BOOKS.YTDSALESS does not exist. The program determines that a permanent data set was specified for DSNAME so it executes a PROC DATASETS step on the library specified in DSNAME. The following PROC DATASETS code is submitted.

```
proc datasets library=books details;
run;
quit;
```

Third call to DETAIL: The value of DSNAME in the third call to DETAIL is YTDSALES. A libref for this data set is not specified, which implies that the data set to be processed is in the WORK directory. If YTDSALES exists in the WORK directory, then the PROC PRINT step executes. If YTDSALES does not exist in the WORK directory, the program skips over the PROC PRINT step and branches to the section labeled as %NODATASET. The statements that immediately follow the %NODATASET label examine the value of DSNAME and determine if it contains a libref. If it does not, the program assigns a libref of WORK to the value of DSLIBREF. It then executes the PROC DATASETS step and lists the SAS data files in the WORK directory.

For the third call, if YTDSALES exists in the WORK directory, the macro program submits the following code:

```
title "PROC PRINT of ytdsales";
proc print data=ytdsales;
  var datesold booktitle saleprice;
run;
```

If YTDSALES does not exist in the WORK directory, the macro program submits the following code:

```
proc datasets library=work details;
run;
quit;
```

Chapter 8

Masking Special Characters and Mnemonic Operators

Introduction 190

Why Are Quoting Functions Called Quoting Functions? 191

Illustrating the Need for Macro Quoting Functions 191

Describing the Commonly Used Macro Quoting Functions 192

Understanding How Macro Quoting Functions Work 194

Applying Macro Quoting Functions 195

Specifying Macro Program Parameters That Contain Special Characters or Mnemonic Operators 203

Unmasking Text and the %UNQUOTE Function 213

Using Quoting Versions of Macro Character Functions and Autocall Macro Programs 214

Introduction

The SAS macro language is a text-handling language that relies on specific syntax structures to perform its tasks in constructing SAS code for you. It relies on triggers such as ampersands and percent signs to understand when you're requesting it to resolve a macro variable and invoke a macro program. It relies on symbols such as parentheses and plus signs, and on mnemonic operators like GT and EQ, to construct expressions and determine how to evaluate them.

Occasionally, however, your applications might require that the macro processor interpret special characters and operators simply as text and not as triggers or symbols. This chapter addresses how to write your macro programming instructions so that the macro processor interprets special characters and mnemonic operators as text.

The macro language contains several functions that you can apply to mask these special characters and mnemonic operators from interpretation by the macro processor. This chapter describes how to apply five commonly used quoting functions:

- %STR and %NRSTR
- %BQUOTE and %NRBQUOTE
- %SUPERQ

This chapter also describes a sixth macro quoting function, %UNQUOTE, which removes the mask from a value so that the special characters and mnemonic operators in the value are interpreted as directions.

Additionally, several functions and autocall macro programs listed in Chapter 6 have a quoting version, and a few examples of them are presented at the end of this chapter. This set of quoting functions and autocall macro programs perform the same actions as their nonquoting counterparts, and they also mask special characters and mnemonic operators. These functions include:

- %QSCAN
- %QSUBSTR
- %QSYSFUNC
- %QUPCASE

The autocall macro programs include:

- %QCMPRES
- %QLEFT
- %QLOWCASE
- %QTRIM

Why Are Quoting Functions Called Quoting Functions?

Macro functions that mask special characters and mnemonic operators are called quoting functions because they behave like single quotation marks in the SAS language. Just as characters that are enclosed in single quotation marks in a SAS language statement are ignored, so too are the special characters and mnemonic operators that are in the arguments to, or results of, a macro quoting function. The difference is that the macro quoting functions offer much more flexibility in what characters to ignore and when to ignore them.

Illustrating the Need for Macro Quoting Functions

Consider how SAS processes the following code where the intention is to assign the three statements in a PROC PRINT step as the value of a macro variable.

```
%let wontwork=proc print data=books.ytdsales;var saleprice;run;
```

After you submit the %LET statement, the macro processor assigns the underlined text to the macro variable WONTWORK. The macro processor treats the first semicolon it encounters as termination of the macro variable assignment. This semicolon terminating the PROC PRINT statement is not stored in the macro variable WONTWORK. After the macro processor assigns the underlined text to the macro variable WONTWORK, processing returns to the input stack and the word scanner. The word scanner tokenizes the next two statements and sends the tokens to the compiler. SAS cannot compile the VAR statement since it is not submitted as part of a PROC step. An error condition is generated as shown in the following SAS log.

```
1122  %let wontwork=proc print data=books.ytdsales;var
saleprice;
                                                      ---
                                                      180
1122! run;
ERROR 180-322: Statement is not valid or it is used out of
               proper order.
```

This %LET statement demonstrates that SAS macro programmers need a way to mask semicolons, other special characters, and mnemonic operators from the macro processor's interpretation of them. Sometimes, the task requires you specify that certain special characters and mnemonic operators be treated simply as text.

The next %LET statement solves the problems with the above %LET statement. It applies the macro quoting function %STR to the entire PROC step. This function blocks the macro processor from interpreting the semicolons within the step as %LET statement terminators when it compiles the %LET statement. Now all three PROC step statements, including the semicolons, are assigned to WILLWORK.

```
%let willwork=%str(proc print data=books.ytdsales;var
saleprice;run;);
```

If you submit a %PUT statement to display the value of WILLWORK, the macro processor writes the following to the SAS log:

```
1124  %put &willwork;
proc print data=books.ytdsales;var saleprice;run;
```

This %PUT statement does not cause the PROC PRINT step to execute. Instead, it just displays the value of the macro variable WILLWORK. If you submit the following, the PROC PRINT step does execute:

```
&willwork
```

Describing the Commonly Used Macro Quoting Functions

This section presents a brief description of the five most commonly used macro quoting functions: %STR, %NRSTR, %BQUOTE, %NRBQUOTE, and %SUPERQ.

Two lesser-used functions, %QUOTE and %NRQUOTE, are mentioned at the end of this section. Use the %STR and %NRSTR functions to mask items during compilation. Use

the %BQUOTE and %NRBQUOTE functions to mask text or resolved values of text expressions during execution.

The two "NR" functions, %NRSTR and %NRBQUOTE, do the same as their non-"NR" counterparts, %STR and %BQUOTE, and these functions also mask the ampersand (&) and percent sign (%) macro triggers.

The %SUPERQ function is also an execution function, but operates differently from %BQUOTE and %NRBQUOTE. Use it to mask the value of a macro variable so that its value is treated as text and any further macro references in the value are not resolved.

The special characters and mnemonic operators that macro quoting functions mask include:

blank	;	¬	^	~
,	'	"	)	(
+	-	*	/	<
>	=	|		
AND	OR	NOT	EQ	NE
LE	LT	GE	GT	IN
%	&	#		

%STR and %NRSTR

These two functions mask special characters at the time of *compilation*. These functions cause their arguments to be tokenized as text. For example, use these functions when you want to assign special characters to a macro variable as was done in the preceding example, or when a macro parameter contains special characters. %STR masks all special characters and mnemonic operators except for ampersands and percent signs. %NRSTR masks the same items as %STR and also masks ampersands and percent signs.When you have an unmatched single quotation mark, an unmatched double quotation mark, or an unmatched parenthesis, precede the unmatched character with a percent sign.

%BQUOTE and %NRBQUOTE

These two functions mask special characters and mnemonic operators contained in the results from resolving macro expressions. The macro processor resolves macro expressions during *execution*. Use these functions when the operands in your expressions might contain special characters or mnemonic operators at resolution and you want those resolved results to be treated as text. In contrast to %STR and %NRSTR, which mask constant text, the functions %BQUOTE and %NRBQUOTE mask resolved values, and resolution occurs at execution. %BQUOTE masks all special characters and mnemonic operators except for ampersands and percent signs. %NRBQUOTE masks the same items as %BQUOTE and additionally masks ampersands and percent signs. When you have an

unmatched single quotation mark, an unmatched double quotation mark, or an unmatched parenthesis, do *not* precede the unmatched character with a percent sign.

%SUPERQ

This function masks the value of a macro variable so that the value is treated solely as text. Percent signs and ampersands in the value of a macro variable are not resolved. The argument to %SUPERQ is the name of a macro variable without the ampersand in front of the macro variable name. The %SUPERQ function operates similarly to the %NRBQUOTE function, but is more complete in its masking. With %NRBQUOTE, the macro processor masks the argument *after* it resolves macro variable references and values. With %SUPERQ, the macro processor masks the argument *before* it resolves any macro variable references or values.

%QUOTE and %NRQUOTE

Two other macro quoting functions, %QUOTE and %NRQUOTE, operate during execution and are equivalent to %BQUOTE and %NRBQUOTE with one exception. The exception is in how the two sets of functions process unmatched parentheses. Both %BQUOTE and %NRBQUOTE do not require that quotation marks or parentheses without a match be marked with a preceding percent sign, while %QUOTE and %NRQUOTE do require a preceding percent sign.

Understanding How Macro Quoting Functions Work

When the macro processor masks a value, it prefixes and suffixes the value with a hexadecimal character called a delta character. The macro processor selects the delta character at the time it processes the function instruction. To use macro quoting functions productively, you do not need to know what this character is. It might be helpful though to realize that the macro processor places this delta character at the beginning and end of your text string. The macro processor selects the character based on the type of quoting you specify, and it uses this character to preserve leading and trailing blanks in your value. These characters are not included as part of the expression when the macro processor evaluates comparisons. Think of them in these situations as acting like single quotation marks in a SAS language statement.

When you have the SYMBOLGEN option in effect, the macro processor writes a message in the SAS log informing you that it has unquoted the value before displaying it. This message relates to the handling of the delta characters. The following statements cause this SYMBOLGEN message to be displayed, and this message is in bold in the SAS log:

```
options symbolgen;
%let monthstring=%str(Jan,Feb,Mar);
%let month=%substr(&monthstring,5,3);
%put **** Characters 5-7 in &monthstring = &month;
```

The SAS log for the preceding code follows.

```
8    options symbolgen;
9    %let monthstring=%str(Jan,Feb,Mar);
10   %let month=%substr(&monthstring,5,3);
SYMBOLGEN:  Macro variable MONTHSTRING resolves to Jan,Feb,Mar
SYMBOLGEN:  Some characters in the above value which were
            subject to macro quoting have been unquoted for
            printing.
11   %put **** Characters 5-7 in &monthstring = &month;
SYMBOLGEN:  Macro variable MONTHSTRING resolves to Jan,Feb,Mar
SYMBOLGEN:  Some characters in the above value which were
            subject to macro quoting have been unquoted for
            printing.
SYMBOLGEN:  Macro variable MONTH resolves to Feb
**** Characters 5-7 in Jan,Feb,Mar = Feb
```

Applying Macro Quoting Functions

This section applies macro quoting functions to commonly encountered situations that require masking of special characters or mnemonic operators. The open code examples show results with and without a macro quoting function, and they use %PUT statements to display the results in the SAS log.

Example 8.1: Using %STR to Prevent Interpretation of the Semicolon As a SAS Statement Terminator

This example demonstrates masking semicolons at compilation. The goal is to assign all the code for a PROC SQL step to one macro variable, MYSQLSTEP. The underlined portion in each %LET statement shows what does get assigned to the macro variable MYSQLSTEP.

The first %LET and %PUT statements show the results when you do not apply a quoting function to the value assigned to MYSQLSTEP. The second %LET and %PUT statements show the results of applying the %STR function to the value that is assigned to MYSQLSTEP.

Program 8.1

```
%let mysqlstep=proc sql;title "SAS Files in Library
BOOKS";select memname, memtype from dictionary.members where
libname='BOOKS';quit;
%put WITHOUT Quoting Functions MYSQLSTEP=&mysqlstep;

%let mysqlstep=%str(proc sql;title "SAS Files in Library
BOOKS";select memname, memtype from dictionary.members where
libname='BOOKS';quit;);
%put WITH Quoting Functions MYSQLSTEP=&mysqlstep;
```

The SAS log for the preceding statements follows.

```
1173  %let mysqlstep=proc sql;title "SAS Files in Library
BOOKS"
1173! ;select memname, memtype from dictionary.members where
1173! libname='BOOKS';quit;
1173  %let mysqlstep=proc sql;title "SAS Files in Library
BOOKS"
1173! ;select memname, memtype from dictionary.members where
       ------
       180
1173! libname='BOOKS';quit;
ERROR 180-322: Statement is not valid or it is used out of
               proper order.

1174  %put WITHOUT Quoting Functions MYSQLSTEP=&mysqlstep;
WITHOUT Quoting Functions MYSQLSTEP=proc sql
1175
1176  %let mysqlstep=%str(proc sql;title "SAS Files in Library
1176! BOOKS";select memname, memtype from dictionary.members
1176! where libname='BOOKS';quit;);
1177  %put WITH Quoting Functions MYSQLSTEP=&mysqlstep;
WITH Quoting Functions MYSQLSTEP=proc sql;title "SAS Files in
Library BOOKS";select memname, memtype from dictionary.members
where libname='BOOKS';quit;
```

Example 8.2: Using %STR to Prevent Interpretation of the Comma As an Argument Delimiter

This example demonstrates masking commas from interpretation as delimiters between arguments to the macro function %SUBSTR. The goal is to extract text from a string that contains commas. Commas also serve as delimiters between the arguments to %SUBSTR.

The first argument to %SUBSTR is the string from which the text should be extracted. In Program 8.2, this string contains the first three letters of the names of three months separated by commas. The underlined portion in each %LET statement shows what the macro processor decides to interpret as the first argument to %SUBSTR.

Demonstrated with the first %LET and %PUT statements, when the commas in the string `Jan,Feb,Mar`, are not masked, the macro processor sees five arguments to %SUBSTR. The syntax of %SUBSTR requires two or three arguments, and the presence of five arguments generates errors. Additionally, %SUBSTR tries to convert the text `Feb` and the text `Mar` to numbers to determine from which position it should begin to extract text and how many characters it should extract.

The second %LET and %PUT statements show the results of applying the %STR function to the first argument that is passed to %SUBSTR.

Program 8.2

```
%let month=%substr(Jan,Feb,Mar,5,3);
%put WITHOUT Quoting Functions MONTH=&month;
%let month=%substr(%str(Jan,Feb,Mar),5,3);
%put WITH Quoting Functions MONTH=&month;
```

The SAS log after submitting the four statements follows.

```
1178  %let month=%substr(Jan,Feb,Mar,5,3);
ERROR: Macro function %SUBSTR has too many arguments.  The
       excess arguments will be ignored.
ERROR: A character operand was found in the %EVAL function or
       %IF condition where a numeric operand is required. The
       condition was: Feb
ERROR: Argument 2 to macro function %SUBSTR is not a number.
ERROR: A character operand was found in the %EVAL function or
       %IF condition where a numeric operand is required. The
       condition was: Mar
ERROR: Argument 3 to macro function %SUBSTR is not a number.
1179  %put WITHOUT Quoting Functions MONTH=&month;
WITHOUT Quoting Functions MONTH=
1180  %let month=%substr(%str(Jan,Feb,Mar),5,3);
1181  %put WITH Quoting Functions MONTH=&month;
WITH Quoting Functions MONTH=Feb
```

Example 8.3: Using %STR to Preserve Leading and Trailing Blanks

This example shows how to preserve leading and trailing blanks in text assigned to a macro variable at compilation. The two %LET statements assign text to a macro variable. By default, the macro processor removes leading and trailing blanks from a text string when assigning it to a macro variable. Applying the %STR function to the text string in the second %LET statement prevents this action.

Both %PUT statements print asterisks adjacent to the start and end of the resolved value assigned to TITLETEXT to make it easier to see that the %STR function preserves leading and trailing blanks.

Program 8.3

```
%let titletext=   B o o k  S a l e s ;
%put WITHOUT Quoting TITLETEXT=*&titletext*;

%let titletext=%str(   B o o k  S a l e s );
%put WITH Quoting TITLETEXT=*&titletext*;
```

The SAS log for the previous statements looks like this:

```
15   %let titletext=   B o o k  S a l e s ;
16   %put WITHOUT Quoting TITLETEXT=*&titletext*;
WITHOUT Quoting TITLETEXT=*B o o k  S a l e s*
17
18   %let titletext=%str(   B o o k  S a l e s );
19   %put WITH Quoting TITLETEXT=*&titletext*;
WITH Quoting TITLETEXT=*   B o o k  S a l e s *
```

Example 8.4: Using %NRSTR to Mask Macro Triggers

This example shows how to prevent the two macro triggers, ampersands and percent signs, from interpretation at compilation by masking the triggers with the %NRSTR function. The goal is to assign text that contains an ampersand and a percent sign to the macro variable, REPORTTITLE.

The previous examples in this section used the %STR function, which does not mask macro triggers. The %NRSTR function masks all that %STR does, and it also masks macro triggers.

Without masking the ampersand, the macro processor interprets the text following the ampersand as a macro variable that should be resolved. The text following the ampersand in this example is `Feb`. Assume when the statements execute in this example, the macro variable named `Feb` does not exist in the global symbol table.

Without masking the percent sign, the macro processor interprets the text following the percent sign as a macro program call that it should execute. The text following the percent sign in this example is `Sales.` Assume when the statements execute in this example, a macro program named SALES has not already been compiled.

Execution of the first %LET and first %PUT statements generate warnings, not errors. The macro processor does assign a value to REPORTTITLE. Every time it attempts to resolve REPORTTITLE, it also tries to resolve FEB as a macro variable and SALES as a macro program invocation.

Program 8.4

```
%let reporttitle=Jan&Feb %Sales Report;
%put WITHOUT Quoting Functions REPORTTITLE=&reporttitle;
%let reporttitle=%nrstr(Jan&Feb %Sales Report);
%put WITH Quoting Functions REPORTTITLE=&reporttitle;
```

The SAS log for the four statements in Program 8.4 follows:

```
1188  %let reporttitle=Jan&Feb %Sales Report;
WARNING: Apparent symbolic reference FEB not resolved.
WARNING: Apparent invocation of macro SALES not resolved.
1189  %put WITHOUT Quoting Functions REPORTTITLE=&reporttitle;
WARNING: Apparent symbolic reference FEB not resolved.
WARNING: Apparent invocation of macro SALES not resolved.
WITHOUT Quoting Functions REPORTTITLE=Jan&Feb %Sales Report
1190  %let reporttitle=%nrstr(Jan&Feb %Sales Report);
1191  %put WITH Quoting Functions REPORTTITLE=&reporttitle;
WITH Quoting Functions REPORTTITLE=Jan&Feb %Sales Report
```

Example 8.5: Using %STR and %BQUOTE to Mask Unbalanced Quotation Marks and Preceding Percent Signs

This example shows how to mask an unbalanced quotation mark. The goal is first to assign a string of three names to the macro variable NAMES and then to extract the third name from the string and assign this value to another macro variable, NAME3. Each name contains a single quotation mark.

A macro quoting function is needed in the first %LET statement to mask the quotation marks. If you use %STR, then you also need to precede each of the three quotation marks with a percent sign.

If you submit the first four statements in Program 8.5a without applying %STR and you do not include the preceding percent signs in the first %LET statement, the next three statements do not execute because of the unbalanced quotation marks. Because of this cascade of errors, the SAS log for the first four statements is not shown.

Note that the example selects the third name from NAMES with the %QSCAN macro function instead of the %SCAN function. The %QSCAN function quotes the result of the %SCAN function. The result contains an unmatched single quotation mark. If you used %SCAN, this unmatched single quotation mark generates errors in the statements that follow. Therefore, using %QSCAN masks the single quotation mark in NAME3, which prevents these errors.

Program 8.5a

```
%let names=O'DONOVAN,O'HARA,O'MALLEY;
%let name3=%qscan(&names,3);
%put WITHOUT STR and Percent Signs NAMES=&names;
%put WITHOUT STR Quoting Function NAME3=&name3;
%let names=%str(O%'DONOVAN,O%'HARA,O%'MALLEY);
%let name3=%qscan(&names,3);
%put WITH STR and Percent Signs NAMES=&names;
%put WITH STR Quoting Function NAME3=&name3;
```

Because of the errors generated with the first group of statements, the SAS log for only the second group is shown:

```
28   %let names=%str(O%'DONOVAN,O%'HARA,O%'MALLEY);
29   %let name3=%qscan(&names,3);
30   %put WITH STR and Percent Signs NAMES=&names;
WITH STR and Percent Signs NAMES=O'DONOVAN,O'HARA,O'MALLEY
31   %put WITH STR Quoting Function NAME3=&name3;
WITH STR Quoting Function NAME3=O'MALLEY
```

The value being assigned to NAMES could instead be masked with %BQUOTE. When you use %BQUOTE, as in Program 8.5b, you would not need to precede the unmatched quotation marks with percent signs. The first %LET statement below is modified from that in Program 8.5a to use %BQUOTE.

Program 8.5b

```
%let names=%bquote(O'DONOVAN,O'HARA,O'MALLEY);
%let name3=%qscan(&names,3);
%put WITH BQUOTE Quoting Function NAMES=&names;
%put WITH BQUOTE Quoting Function NAME3=&name3;
```

The results of submitting these four statements follow.

```
32   %let names=%bquote(O'DONOVAN,O'HARA,O'MALLEY);
33   %let name3=%qscan(&names,3);
34   %put WITH BQUOTE Quoting Function NAMES=&names;
WITH BQUOTE Quoting Function NAMES=O'DONOVAN,O'HARA,O'MALLEY
35   %put WITH BQUOTE Quoting Function NAME3=&name3;
WITH BQUOTE Quoting Function NAME3=O'MALLEY
```

Example 8.6: Masking Macro Triggers and Unbalanced Quotation Marks with %NRSTR and Preceding Percent Signs

This example modifies the code in Example 8.5 by replacing the comma delimiter in the string of names with an ampersand delimiter. Since the ampersand is a macro trigger, %STR does not mask this character. It is necessary to use %NRSTR instead to mask the two ampersands. This prevents attempted resolution of a macro variable named O. Since the string of names contains unmatched single quotation marks, percent signs are added preceding each quotation mark.

Program 8.6

```
%let names=%nrstr(O%'DONOVAN&O%'HARA&O%'MALLEY);
%let name3=%qscan(&names,3);
%put WITH NRSTR Quoting Function NAMES=&names;
%put WITH NRSTR Quoting Function NAME 3 is: &name3;
```

The SAS log from the preceding code follows.

```
36   %let names=%nrstr(O%'DONOVAN&O%'HARA&O%'MALLEY);
37   %let name3=%qscan(&names,3);
38   %put WITH NRSTR Quoting Function NAMES=&names;
WITH NRSTR Quoting Function NAMES=O'DONOVAN&O'HARA&O'MALLEY
39   %put WITH NRSTR Quoting Function NAME 3 is: &name3;
WITH NRSTR Quoting Function NAME 3 is: O'MALLEY
```

Example 8.7: Using %BQUOTE to Prevent Interpretation of Mnemonic Operators

The %SYSEVALF function in Program 8.7 does a Boolean evaluation of a logical expression. It demonstrates why it might be necessary to mask elements of an expression from the macro processor at the time of execution.

Program 8.7 starts by assigning the state abbreviation for Oregon, OR, to the macro variable STATE. Next, it tests whether the value of STATE equals OR. The result of the test is returned as a Boolean value: 0 means false and 1 means true.

You must tell the macro processor when you want a mnemonic operator treated as text. In Program 8.7, you would use a quoting function to mask OR so that it is treated as text and not as a mnemonic operator.

The third %LET statement masks the value of STATE with the %BQUOTE function. The %STR function masks the text string to which the value of STATE is compared. The macro processor is able to evaluate the condition and, in this situation, assigns a value of 1 to the macro variable VALUE because the condition it tested is true.

Program 8.7

```
%let state=OR;
%let value=%sysevalf(&state eq OR, boolean);
%put WITHOUT Quoting Functions VALUE=&value;
%let value=%sysevalf( %bquote(&state) eq %str(OR), boolean);
%put WITH Quoting Functions VALUE=&value;
```

The SAS log for the previous statements looks like this:

```
1200  %let state=OR;
1201  %let value=%sysevalf(&state eq OR, boolean);
ERROR: A character operand was found in the %EVAL function or
       %IF condition where a numeric operand is required. The
       condition was: OR eq OR
1202  %put WITHOUT Quoting Functions VALUE=&value;
WITHOUT Quoting Functions VALUE=
1203  %let value=%sysevalf( %bquote(&state) eq %str(OR),
1203! boolean);
1204  %put WITH Quoting Functions VALUE=&value;
WITH Quoting Functions VALUE=1
```

Example 8.8: Using %SUPERQ to Prevent Resolution of Special Characters in a Macro Variable Value

The %SUPERQ macro function in this example masks from interpretation text that looks like a macro variable reference. Program 8.8 starts with a PROC MEANS step that analyzes variable SALEPRICE for the publisher Doe&Lee Ltd. The publisher name is written such that the ampersand is adjacent to "Lee." The program includes the publisher name in a text string that is assigned to a macro variable. When the macro variable is referenced, the usage of %SUPERQ prevents "&Lee" in the text string from being interpreted as a macro variable reference.

The PROC MEANS step computes the total of SALEPRICE for this publisher and saves the sum in the output data set SALESDL. A DATA step follows that creates the macro variable TOTSALES_DL with CALL SYMPUTX. The text assigned to TOTSALES_DL is inserted in the FOOTNOTE statement. The %SUPERQ function is applied to TOTSALES_DL in the FOOTNOTE statement, and this prevents the macro processor from attempting to resolve the "&Lee" as a macro variable reference. CALL SYMPUTX is a SAS language function that assigns values to macro variables. Its features are described in Chapter 9.

Program 8.8

```
proc means data=books.ytdsales
            (where=(publisher='Doe&Lee Ltd.')) noprint;
  var saleprice;
  output out=salesdl sum=;
run;

data _null_;
  set salesdl;
  call symputx('totsales_dl',
          cat('The total sales for Doe&Lee Ltd is ',
              put(saleprice,dollar10.2),'.'));
run;
footnote "%superq(totsales_dl)";
```

Program 8.8 executes without any warnings or errors. The footnote becomes:

```
The total sales for Doe&Lee Ltd is $22,688.46.
```

Without the %SUPERQ function, the macro processor writes a WARNING to the SAS log. The FOOTNOTE statement without the %SUPERQ function is:

```
footnote "&totsales_dl";
```

Since macro variable LEE does not exist, the WARNING states that the macro processor was unable to resolve the reference to macro variable LEE.

```
WARNING: Apparent symbolic reference LEE not resolved.
```

Specifying Macro Program Parameters That Contain Special Characters or Mnemonic Operators

The preceding discussion and examples describe the use of five quoting functions: %STR, %NRSTR, %BQUOTE, %NRBQUOTE, and %SUPERQ. This section also applies these functions and does so in the context of passing parameters to macro programs where the parameter values could contain special characters or mnemonic operators.

When writing your macro programs and defining parameters for these programs, you need to consider the range of values your parameters could take. Sometimes the

parameters passed to your macro program can contain special characters and mnemonic operators that need to be masked to prevent the macro processor from interpreting them.

For example, consider what happens if the value of your parameter contains a comma. The macro processor interprets commas in a macro program call to be separator characters between parameters. When a parameter value contains a special character such as a comma, you must mask that special character so that the macro processor ignores it. Examples below demonstrate this application.

As another example, consider what happens if the value of a parameter contains a mnemonic operator and the parameter is part of a macro expression inside the macro program. As shown in the preceding examples, these elements of a macro expression might need to be masked to prevent the macro processor from interpreting them as operands of the macro expression. The following examples demonstrate this application.

Example 8.9: Masking Special Characters in Parameter Values

When your parameter values can contain special characters, you need to mask them so that the macro processor does not interpret them as anything but text. To do this, you would typically place either the %STR or %NRSTR function around the text that needs to be masked. If the value contains any special character other than an ampersand or percent sign adjacent to text, you can use %STR. If the value can contain an ampersand or percent sign adjacent to text, use %NRSTR to prevent the macro processor from interpreting either of those characters as macro triggers. When you mask a parameter value, it stays masked within the macro program, unless you unmask it with %UNQUOTE.

Macro program MOSECTDETAIL in Program 8.9 generates a PROC PRINT step that lists books sold during specific months from a specific section. It has two parameters. The first is the list of months with months specified numerically. The second is one specific section. The list of months will be inserted as the object of the IN operator on the WHERE statement. The program expects the list of months to be separated by commas.

Macro program MOSECTDETAIL is called twice. The first call does not surround the list of months with the %STR function while the second call does. The first call does not execute. The second call executes a PROC PRINT step that lists the books sold for March and June in the Internet section. The underlined part of each call to MOSECTDETAIL shows the value that the macro processor interprets as the first parameter, MONTHLIST.

Program 8.9

```
%macro mosectdetail(monthlist,section);
  proc print data=books.ytdsales;
    title "List of titles sold for months &monthlist";
    where month(datesold) in (&monthlist)
          and section="&section";
    var booktitle saleprice;
  run;
%mend mosectdetail;

%mosectdetail(3,6,Internet)

%mosectdetail(%str(3,6),Internet)
```

After submitting the first call to MOSECTDETAIL, the macro processor writes the following to the SAS log and does not execute the PROC PRINT step. It sees three positional parameters in the call to MOSECTDETAIL. The comma that separates 3 and 6 is interpreted as the separator between two parameters.

```
ERROR: More positional parameters found than defined.
```

The second call to MOSECTDETAIL masks the comma between 3 and 6 from interpretation as the separator between parameter values. After resolution by the macro processor, the following PROC PRINT step is executed by the second call to MOSECTDETAIL.

```
proc print data=books.ytdsales;
  title "List of titles sold for months 3,6";
  where month(datesold) in (3,6) and section="Internet";
  var booktitle saleprice;
run;
```

Note that if you wanted to run this report for only one month you would not need to mask that value. For example, to request a report for only December for the Certification and Training section, specify the call to MOSECTDETAIL as follows.

```
%mosectdetail(12,Certification and Training)
```

Example 8.10: Masking Equal Signs in Parameter Values to Prevent Misinterpretation of Positional Parameters as Keyword Parameters

This example defines a macro program called PUBLISHERSALES that constructs a PROC REPORT step and has three positional parameters. The first parameter, DESTINATION, specifies the ODS destination of the report. The second and third parameters, STYLEHEADER and STYLEREPORT, specify style attributes for two report locations: the header and the body of the report.

ODS style attributes for these PROC REPORT locations are written in the format:

```
{style-element-name=style-attribute-specification(s)}
```

The macro program expects the parameters to be in this format as well since it inserts these specifications as is in the PROC REPORT statement STYLE(HEADER) option and the STYLE(REPORT) option.

Values for the second and third parameters passed to PUBLISHERSALES must be masked to prevent interpretation of the equal sign as a signal to the macro processor to process a keyword parameter. This example uses the %STR quoting function to mask the values of the second and third parameters.

The call to PUBLISHER in Program 8.10 sends the report to the HTML destination and specifies that the program write the headers in italics, insert row rules in the report, and not insert spaces between the cells of the report.

Program 8.10

```
%macro publishersales(destination,styleheader,stylereport);
  ods listing close;
  ods &destination;

  title "Sales by Publisher";
  proc report data=books.ytdsales
              style(header)={&styleheader}
              style(report)={&stylereport}
              nowd;
    column publisher saleprice n;
    define publisher / group;
    define saleprice / format=dollar10.2;
  run;

  ods &destination close;
  ods listing;
```

```
%mend publishersales;

%publishersales(html,
           %str(font_style=italic),
           %str(rules=rows cellspacing=0))
```

After resolution by the macro processor, SAS submits the following code:

```
ods listing close;
ods html;

title "Sales by Publisher";
proc report data=books.ytdsales
            style(header)={font_style=italic}
            style(report)={rules=rows cellspacing=0}
            nowd;
  column publisher saleprice n;
  define publisher / group;
  define saleprice / format=dollar10.2;
run;

ods html close;
ods listing;
```

The following call to PUBLISHERSALES does not apply the %STR function to the second and third parameters.

```
%publisher(html,font_style=italic,rules=rows cellspacing=0)
```

In this call, the macro processor sees three parameters being passed to PUBLISHERSALES. The first is a positional parameter whose value is `html`. The second is a keyword parameter, FONT_STYLE, with a value of `italic`. The third is a keyword parameter, RULES with a value of `rows cellspacing=0`.

The PUBLISHER macro program in Program 8.10 defined three positional parameters and did not define any keyword parameters. Submitting the preceding call to PUBLISHERSALES causes the macro processor to write the following error messages to the SAS log. No ODS statements are executed, and no report is produced.

```
ERROR: The keyword parameter FONT_STYLE was not defined with
       the macro.
ERROR: The keyword parameter RULES was not defined with the
       macro.
```

Example 8.11: Masking Special Characters and Mnemonic Operators in Parameter Values

This example presents several variations in masking special characters and mnemonic operators in parameters passed to a macro program. It shows ways of masking special characters in the macro program call and masking mnemonic operators within the macro program that might be part of a macro expression.

The purpose of the macro program MYPAGES is to specify text and attributes for a TITLE and FOOTNOTE statement. The macro program has six keyword parameters, three for the title and three for the footnote.

The three parameters for the TITLE statement specify the title text, the justification or position (left, center, or right) of the title, and the color of the title. The parameters for the FOOTNOTE statement are similar: one parameter specifies the footnote text, one specifies the justification, and the third specifies the color of the footnote.

The justification and color parameters for both the TITLE and FOOTNOTE statements have initial values. For the title, the default value for justification is center, and the default color is black. For the footnote, the default value for justification is right, and the default color is black.

The macro program checks to see if a value is specified for TITLETEXT, the text for the title. If not, titles are cleared by submitting a TITLE1 statement with no text. Similarly, the program checks the contents of FOOTNOTETEXT, the text for the footnote. When no value is specified for FOOTNOTETEXT, footnotes are cleared by submitting a FOOTNOTE1 statement with no text.

Problems might arise with this macro program if the values you specify for TITLETEXT or FOOTNOTETEXT contain special characters or mnemonic operators. A macro program call when one of these values contains special characters might not execute or it might execute incorrectly if you do not mask the special characters in the macro program call. When one of these values contains mnemonic operators, the %IF statement can fail unless you mask the parameter value at execution time within the macro program.

Program 8.11 calls MYPAGES four times, each time demonstrating different applications of macro quoting functions. An explanation of each of the four calls follows the code.

Program 8.11

```
%macro mypages(titletext=,jtitle=center,ctitle=black,
               footnotetext=,jfootnote=right,cfootnote=black);

  %if %superq(titletext)= %then %do;
    title1;
  %end;
  %else %do;
    title justify=&jtitle color=&ctitle
          "&titletext";
  %end;

  %if %superq(footnotetext)= %then %do;
    footnote1;
  %end;
  %else %do;
    footnote justify=&jfootnote color=&cfootnote
             "&footnotetext";
  %end;
%mend mypages;

options macrogen;

*----First call of MYPAGES;
%mypages(titletext=Sales Report,ctitle=blue,
         footnotetext=Last Review Date: Feb 1%str(,) 2008)

*----Second call of MYPAGES;
%mypages(titletext=2007+ Sales,
         footnotetext=Prepared with SAS &sysver)

*----Third call of MYPAGES;
%mypages(titletext=Sales Report,
         footnotetext=Last Reviewed by %str(O%'Malley))

*----Fourth call of MYPAGES;
%mypages(titletext=%nrstr(Audited&Approved),
         footnotetext=
           %nrstr(%Increase in Sales for Year was 8%%),
         jfootnote=center)
```

First call to MYPAGES. The first call to MYPAGES specifies text for the title, the color of the title, and text for the footnote. No special characters or mnemonic operators are present in the text for the title so the value is not masked. The text for the footnote does contain a special character, a comma. To prevent the macro processor from interpreting the comma as anything but text, the comma must be masked. The macro

quoting function %STR successfully masks the comma. The first call to MYPAGES submits the following two SAS statements.

```
title justify=center color=blue "Sales Report";
footnote justify=right color=black
                "Last Review Date: Feb 1, 2008";
```

Without the %STR mask around the comma, the macro processor interprets the first call to MYPAGES to have a positional parameter following the comma whose value is 2008. SAS requires that positional parameters precede keyword parameters. The macro processor stops executing the macro program when it detects this problem. It writes the following message to the SAS log.

```
ERROR: All positional parameters must precede keyword
       parameters.
```

Note that the call to MYPAGES masks only the comma in the value for FOOTNOTETEXT. The same footnote is produced if you mask the entire value of FOOTNOTETEXT:

```
%mypages(titletext=Sales Report,ctitle=blue,
         footnotetext=%str(Last Review Date: Feb 1, 2008)
```

Select the text to mask that is easiest for you to specify. The best way to do this might be to mask the entire value. It might be easier to "count" parentheses if you mask the entire value rather than masking each of the individual special characters within the text value.

Second call to MYPAGES. The second call to MYPAGES specifies text for the title and text for the footnote. The text for the title contains an operator, the plus sign (+). The text for the footnote contains a reference to the automatic variable, SYSVER, whose value is equal to the currently executing version of SAS. The second call to MYPAGES submits the following two SAS statements.

```
title justify=center color=black "2007+ Sales";
footnote justify=right color=black "Prepared with SAS 9.1";
```

While it is not necessary to mask the plus sign in the macro program call, it is necessary to mask the parameter in the %IF statement where it is referenced. The program applies the %SUPERQ function to the TITLETEXT value. In case a similar situation arises with the FOOTNOTETEXT value, the program applies the %SUPERQ function to FOOTNOTETEXT on the %IF statement where it is referenced. The %BQUOTE function would also work for this application, but to completely prevent resolution of macro triggers that might occur in the value, the %SUPERQ function is used instead.

Note that the reference to SYSVER in the parameter specification for FOOTNOTETEXT is not masked. In this situation, the goal is to display the resolved value of SYSVER in the footnote. Consider what happens if you omit the %SUPERQ function from the program and you rewrite the %IF statement as follows.

```
%if &titletext= %then %do;
```

The second call to MYPAGES would not execute without masking the value of TITLETEXT at execution. The macro processor interprets the plus sign in the value for TITLETEXT as an operator. With %SUPERQ removed, the same second call to MYPAGES produces the following error messages in the SAS log.

```
ERROR: A character operand was found in the %EVAL function or
       %IF condition where a numeric operand is required. The
       condition was: &titletext=
ERROR: The macro MYPAGES will stop executing.
```

Third call to MYPAGES. The third call to MYPAGES specifies text for the title and text for the footnote. The text for the title does not contain any operators or special characters and it is not masked. The text for the footnote contains an unmatched quotation mark, which must be masked. The third call to MYPAGES submits the following two SAS statements.

```
title justify=center color=black "Sales Report";
footnote justify=right color=black
              "Last Reviewed by O'Malley";
```

Both the %STR function and the percent sign preceding the unmatched quotation mark are required to mask the unmatched quotation mark. Without one of the two masking items, SAS does not see a complete call to MYPAGES, and this results in processing errors involving unmatched quotation marks and parentheses.

Fourth call to MYPAGES. The fourth call to MYPAGES specifies text for the title, text for the footnote, and center justification of the footnote. The text for the title contains the ampersand macro trigger followed by text. The text for the footnote contains the percent sign macro trigger and concludes with a percent sign. The fourth call to MYPAGES submits the following two SAS statements.

```
title justify=center color=black "Audited&Approved";
footnote justify=center color=black
              "%Increase in Sales for Year was 8%";
```

Since the parameter values for TITLETEXT and FOOTNOTETEXT contain macro triggers, the %NRSTR function must be used instead of the %STR function as in the first call to MYPAGES. Without masking the value for TITLETEXT, the macro processor attempts to resolve a macro variable named APPROVED. Without masking the value for

FOOTNOTETEXT, the macro processor attempts to execute a macro program named INCREASE. The macro program MYPAGES does execute with the unmasked parameters and produces the correct TITLE and FOOTNOTE statements. However, it does write the following warnings to the SAS log.

```
WARNING: Apparent symbolic reference APPROVED not resolved.
WARNING: Apparent invocation of macro INCREASE not resolved.
```

There are several ways to specify the concluding percent sign in the value specified for FOOTNOTETEXT. If you remember to leave a space after the percent sign, you do not need to provide any additional instruction to prevent the macro processor from interpreting the percent sign as anything but text.

```
%mypages(titletext=%nrstr(Audited&Approved),
         footnotetext=
           %nrstr(%Increase in Sales for Year was 8% ),
         jfootnote=center)
```

Since a percent sign can serve as a mask for an unmatched parenthesis, and if you put the right parenthesis next to the percent sign, the macro processor interprets the right parenthesis as text.

```
%mypages(titletext=%nrstr(Audited&Approved),
         footnotetext=
           %nrstr(%Increase in Sales for Year was 8%),
         jfootnote=center)
```

Submitting the preceding call to MYPAGES does not cause MYPAGES to execute because it needs another right parenthesis to fully specify the call, as shown here:

```
%mypages(titletext=%nrstr(Audited&Approved),
         footnotetext=
           %nrstr(%Increase in Sales for Year was 8%)),
         jfootnote=center)
```

Specifying an additional parenthesis as in the immediately preceding call to MYPAGES still does not produce the desired footnote. The concluding percent sign is not treated as text and a right parenthesis becomes part of the footnote. Submitting the preceding call to MYPAGES defines this footnote:

```
%Increase in Sales for Year was 8)
```

Another way to code the specification for FOOTNOTETEXT is to insert two concluding percent signs.

```
%mypages(titletext=%nrstr(Audited&Approved),
         footnotetext=
           %nrstr(%Increase in Sales for Year was 8%%),
         jfootnote=center)
```

This executes as desired, producing the following footnote:

```
%Increase in Sales for Year was 8%
```

Unmasking Text and the %UNQUOTE Function

Occasionally you might need to restore the meaning of special characters and mnemonic operators that have been masked. Applying the %UNQUOTE function to the masked value tells the macro processor to remove the mask and resolve the special characters and mnemonic operator.

Example 8.12: Using %UNQUOTE to Cause Interpretation of a Masked Character

In Program 8.12, the call to the macro program MAR has been masked and assigned to the macro variable M. This text is placed in the first TITLE statement. To have the value of M interpreted, the %UNQUOTE function must be used. The second TITLE statement contains the results of applying %UNQUOTE to the value of M.

Program 8.12

```
%macro mar;
  This is March
%mend;

%let m=%nrstr(%mar);
title "Macro call &m generates the following text";
title2 "%unquote(&m)";
```

The TITLE statements after submission of the preceding code are as follows:

```
Macro call %mar generates the following text
This is March
```

Using Quoting Versions of Macro Character Functions and Autocall Macro Programs

The results of macro character functions and autocall macro programs are unmasked or unquoted. The macro processor resolves special characters and mnemonic operators in the results. If you want to mask these items in a result, use the quoting version of the function or autocall macro program.

In Chapter 6, Table 6.1 included descriptions of the quoting versions of macro character functions, and Table 6.7 included descriptions of the quoting versions of autocall macro programs. The %QSCAN function was used in Example 8.5 with unbalanced quotation marks. Two additional examples of using the quoting versions of macro functions follow in Examples 8.13 and 8.14.

Example 8.13: Using %QSYSFUNC to Mask the Result from Applying a SAS Language Function

Described in Chapter 6, the %SYSFUNC function applies SAS language functions to macro variables and text and returns results to the macro facility. When your result could include special characters or mnemonic operators, you should use %QSYSFUNC, which is the quoting version of %SYSFUNC. This function does the same tasks as %SYSFUNC, and it also masks special characters and mnemonic operators.

The macro language statements in Program 8.13 demonstrate an application of %QSYSFUNC. The first %LET statement assigns a value to macro variable PUBLISHER. The next statements convert the text `and` to an ampersand and remove the blanks. Two %PUT statements display the results.

The second %LET statement converts the text `and` to an ampersand using the SAS language function TRANWRD and %SYSFUNC, and it stores the result in macro variable PUBLISHER2.

The third %LET statement removes the blanks in PUBLISHER2 using the SAS language function COMPRESS and %SYSFUNC, and it assigns the result to PUBLISHER3. Execution of this %LET statement causes the macro processor to write warnings to the SAS log since the result of the two functions is not quoted, and the macro processor tries to resolve the macro variable reference `&LEE` in the result.

The fourth %LET statement uses COMPRESS and %QSYSFUNC, and it assigns the result to PUBLISHER3. This time, the value assigned to PUBLISHER3 is quoted through the use of %QSYSFUNC, and the macro processor does not interpret `&LEE` as a macro variable reference.

Note that the %NRSTR function masks the macro function names in the two %PUT statements. If you do not mask these items, the macro processor attempts to execute these functions. In the %PUT statements, these two function names are meant to be displayed as text and not to be interpreted as calls to the functions. Therefore, without the use of %NRSTR, syntax errors are generated.

Program 8.13

```
%let publisher=Doe and Lee;
%let publisher2=%sysfunc(tranwrd(&publisher,and,&));
%let publisher3=%sysfunc(compress(&publisher2)) Ltd.;
%put PUBLISHER3 defined with %nrstr(%SYSFUNC): &publisher3;

%let publisher3=%qsysfunc(compress(&publisher2)) Ltd.;
%put PUBLISHER3 defined with %nrstr(%QSYSFUNC): &publisher3;
```

The SAS log from the preceding open code statements follow. Note that execution of the last %LET statement and %PUT statement does not produce any warnings in the SAS log.

```
230  %let publisher=Doe and Lee;
231  %let publisher2=%sysfunc(tranwrd(&publisher,and,&));
232  %let publisher3=%sysfunc(compress(&publisher2)) Ltd.;
WARNING: Apparent symbolic reference LEE not resolved.
233  %put PUBLISHER3 defined with %nrstr(%SYSFUNC):
&publisher3;
WARNING: Apparent symbolic reference LEE not resolved.
PUBLISHER3 defined with %SYSFUNC: Doe&Lee Ltd.
234
235  %let publisher3=%qsysfunc(compress(&publisher2,%str( )))
Ltd.;
236  %put PUBLISHER3 defined with %nrstr(%QSYSFUNC):
&publisher3;
PUBLISHER3 defined with %QSYSFUNC: Doe&Lee Ltd.
```

Example 8.14: Using %QSUBSTR to Mask the Results of %SUBSTR

Program 8.14 uses the %QSUBSTR macro function to mask the results of the %SUBSTR macro function. The macro variable MONTH3 is defined by extracting text from the MONTHS macro variable using the %SUBSTR macro function. This action results in a warning because the macro processor attempts to resolve what looks likes a macro variable reference in the text extracted by %SUBSTR.

The macro variable QMONTH3 is defined by extracting text from the MONTHS macro variable using the %QSUBSTR macro function. The %QSUBSTR macro function masks the ampersand in the extraction. No warning messages are generated because the macro

processor ignores the ampersand in the text extracted by the %QSUBSTR macro function.

Program 8.14

```
%let months=%nrstr(Jan&Feb&Mar);
%let month3=%substr(&months,8);
%put Unquoted: &month3;

%let qmonth3=%qsubstr(&months,8);
%put Quoted: &qmonth3;
```

The SAS log from Program 8.14 follows. The warnings result from execution of the %LET statement that defines MONTH3 and from execution of the first %PUT statement. The value assigned to MONTH3 is not masked. Therefore, the macro processor interprets &MAR as a macro variable reference. Since macro variable MAR does not exist in this example, the macro processor issues the warnings.

```
40   %let months=%nrstr(Jan&Feb&Mar);
41   %let month3=%substr(&months,8);
WARNING: Apparent symbolic reference MAR not resolved.
42   %put Unquoted: &month3;
WARNING: Apparent symbolic reference MAR not resolved.
Unquoted: &Mar
43
44   %let qmonth3=%qsubstr(&months,8);
45   %put Quoted: &qmonth3;
Quoted: &Mar
```

Chapter 9

Interfaces to the Macro Facility

Introduction 218

Understanding DATA Step Interfaces to the Macro Facility 218

- Understanding the SYMGET and SYMGETN Functions 219
- Understanding the SYMPUT and SYMPUTX Call Routines 226
- Understanding the CALL EXECUTE Routine 234
- Understanding the RESOLVE Function 245

Using Macro Facility Features in PROC SQL 251

- Creating and Updating Macro Variables with PROC SQL 251
- Using the Macro Variables Created by PROC SQL 258
- Displaying Macro Option Settings with PROC SQL and Dictionary Tables 260

Using Macro Facility Features in SAS Component Language 262

- Using the Macro Facility to Pass Information between SCL Programs 263
- Referencing Macro Variables in SUBMIT Blocks 264

Introduction

The interfaces described in this chapter provide you with a dynamic communication link between the SAS language and the macro facility. Until now, the discussion of the macro facility has emphasized the distinction between when macro language statements are resolved and when SAS language statements are resolved, and how the macro language can build SAS code and control SAS processing. With the interfaces described in this chapter, your SAS language programs can direct the actions of the macro processor.

The interfaces described in this chapter include:

- SAS language functions and routines
- PROC SQL
- SAS Component Language functions and routines

Additionally, two macro functions provide an interface with SAS/CONNECT: %SYSLPUT and %SYSRPUT. Discussion of these functions is beyond the scope of this book. Refer to *SAS Macro Language: Reference* for more information on these functions.

Understanding DATA Step Interfaces to the Macro Facility

Three functions and three call routines in the SAS language can interact with the macro processor *during* execution of a DATA step. Table 9.1 lists these six tools.

Table 9.1 DATA step interface tools

Tool	Description
`SYMGET(argument)`	SAS language function that obtains the value of a macro variable specified as *argument* and returns this as a *character* value during DATA step execution.
`SYMGETN(argument)`	SAS language function that obtains the value of a macro variable specified as *argument* and returns this as a *numeric* value. This function is available in SCL and is pre-production in SAS®9.
`CALL SYMPUT(macro-variable, value);`	SAS language routine that assigns *value* produced in a DATA step to a *macro-variable*. This routine does not trim leading and trailing blanks.
`CALL SYMPUTX(macro-variable, value <,symbol-table>);`	SAS language routine that assigns *value* produced in a DATA step to a *macro-variable*. This routine removes both leading and trailing blanks. Optionally, this routine can direct the macro processor to store the macro variable in a specific symbol table.
`CALL EXECUTE(argument);`	SAS language routine that executes the resolved value of *argument*. Arguments that resolve to a macro facility reference execute immediately. Any SAS language statements resulting from the resolution are executed at the end of the step.
`RESOLVE(argument)`	SAS language function that resolves *argument* during DATA step execution where *argument* is a text expression. Text expressions include macro variables and macro program calls.

Understanding the SYMGET and SYMGETN Functions

The SYMGET and SYMGETN SAS language functions retrieve macro variable values from the macro symbol tables during execution of a DATA step. The SYMGET function returns a character value while SYMGETN returns a numeric variable. With these

functions, you can create and update data set variables with information that the macro processor retrieves from macro variables. Note that SYMGETN is pre-production in SAS®9.

The macro variables that you reference with SYMGET and SYMGETN must exist before you apply SYMGET or SYMGETN in a DATA step. If you create a macro variable in the same DATA step with CALL SYMPUT or CALL SYMPUTX, you can retrieve the macro variable value with SYMGET or SYMGETN if these functions follow the CALL SYMPUT or CALL SYMPUTX calls.

By default, SYMGET creates a character variable with a length of 200 bytes. You can specify a different length with either the LENGTH or ATTRIB statement. If the DATA step variable is defined as numeric, SAS attempts to convert the value that SYMGET retrieves to a number and writes a warning message to the SAS log. In those situations, you might want to use SYMGETN since it returns a numeric value that does not require conversion.

The SYMGET and SYMGETN functions accept three types of arguments:

- the *name of a macro variable* that is enclosed in single quotation marks and without the leading ampersand. In the following example, assume X is a macro variable that was defined earlier in the program.

```
y=symget('x');
```

- the *name of a DATA step character variable* whose value is the name of a macro variable. (See Example 9.1 for a discussion of this code.)

```
%let certific=CNT283817;
%let internet=INT3521P8;
%let networks=NET3UD697;
%let operatin=OPSI18375;
%let programm=PRG8361WQ;
%let webdesig=WBD188377;

data temp;
  set books.ytdsales;

  attrib compsect length=$8 label='Section'
         sectioncode length=$9 label='Section Code';

  *----Construct macro variable name by compressing
       section name and taking the first 8 characters.
       When section=Web Design, COMPSECT="WEBDESIG";
```

```
    compsect=substr(compress(section),1,8);
    sectionid=symget(compsect);
  run;
```

- a *DATA step character expression*. The resolution of the character expression is the name of a macro variable. (See Example 9.2 for a discussion of similar code.)

```
%let factor1=1.10;
%let factor2=1.23;
%let factor3=1.29;

data projections;
  set books.ytdsales;
  array factor{3} factor1-factor3;
  array newprice{3} newprice1-newprice3;

  format newprice1-newprice3 dollar10.2;

  drop i;

  do i=1 to 3;
    factor{i}=symgetn(cats('factor',put(i,1.)));
    newprice{i}=factor{i}*saleprice;
  end;
run;
```

Following are several examples of the three types of arguments that SYMGET and SYMGETN can receive.

Example 9.1: Using a Data Set Variable Name As the Argument to the SYMGET Function

Program 9.1 shows how the value of a data set variable can be used to specify the macro variable whose value SYMGET obtains. The open code %LET statements and the DATA step were presented earlier in this section.

Preceding the DATA step, six global macro variables are created, one for each of the six sections in the BOOK.YTDSALES data set. As the DATA step processes each observation in BOOK.YTDSALES, the SYMGET function extracts a value from one of the six macro variables based on the current observation's value of the data set variable SECTION and stores the extracted value in the variable SECTIONID. The value that the

SYMGET function returns is a character value. The ATTRIB statement assigns a length of 9 bytes to SECTIONID, which overrides the default length of 200 bytes.

The data set variable COMPSECT that the data set creates stores the name of the macro variable that contains the specific section's identification code. COMPSECT equals the first eight characters of the section name with blanks in those first eight characters removed.

Program 9.1

```
%let certific=CNT283817;
%let internet=INT3521P8;
%let networks=NET3UD697;
%let operatin=OPSI18375;
%let programm=PRG8361WQ;
%let webdesig=WBD188377;

data temp;
  set books.ytdsales;

  attrib compsect length=$8 label='Section'
         sectionid length=$9 label='Section ID';

  *----Construct macro variable name by compressing
       section name and taking the first 8 characters.
       When section=Web Design, COMPSECT="WebDesig";

  compsect=substr(compress(section),1,8);
  sectionid=symget(compsect);
run;
proc print data=temp;
  title "Defining the Section Identification Code";
  var section compsect sectionid;
run;
```

A partial listing of the PROC PRINT output presented in Output 9.1 shows the values assigned to SECTIONID by SYMGET.

Output 9.1 Partial output from Program 9.1, which uses a data set variable as an argument to SYMGET

Defining the Section Identification Code

Obs	section	compsect	sectionid
1	Web Design	WebDesig	WBD188377
2	Certification and Training	Certific	CNT283817
3	Web Design	WebDesig	WBD188377
4	Programming and Applications	Programm	PRG8361WQ
5	Internet	Internet	INT3521P8
6	Programming and Applications	Programm	PRG8361WQ
7	Internet	Internet	INT3521P8
8	Web Design	WebDesig	WBD188377
9	Internet	Internet	INT3521P8
10	Programming and Applications	Programm	PRG8361WQ
11	Networks and Telecommunication	Networks	NET3UD697
12	Certification and Training	Certific	CNT283817
13	Programming and Applications	Programm	PRG8361WQ
14	Certification and Training	Certific	CNT283817
15	Internet	Internet	INT3521P8
.	. .		

Example 9.2: Retrieving Macro Variable Values and Creating Numeric Data Set Variables with SYMGETN

Program 9.2 directly references two macro variables with the SYMGETN function. The two macro variables are defined in open code preceding the DATA step in which they are referenced. On each iteration of the DATA step, it selects which macro variable value to retrieve based on the current observation's value for data set variable, SECTION. The DATA step selects specific observations from the data set and then creates a new numeric variable whose value is the product of a variable in the data set and the value of a macro variable.

Program 9.2

```
%let webfctr=1.20;
%let intfctr=1.35;

data temp;
  set books.ytdsales(where=(
     section in ('Web Design', 'Internet')));
  if section='Web Design' then costfctr=symgetn('webfctr');
  else if section='Internet' then costfctr=symgetn('intfctr');
  newprice=costfctr*cost;
run;
proc print data=temp;
  title "Prices based on COSTFCTR";
  var section cost costfctr newprice;
  format newprice dollar8.2;
run;
```

A partial listing of the PROC PRINT output presented in Output 9.2 shows that the COSTFCTR variable was created for each observation in the data set. The value of COSTFCTR depends on the value of SECTION.

Output 9.2 Partial output from Program 9.2, which specifies a direct reference to a macro variable in a call to function SYMGET

Prices based on COSTFCTR

Obs	section	cost	costfctr	newprice
1	Web Design	$18.48	1.20	$22.17
2	Web Design	$17.48	1.20	$20.97
3	Internet	$18.48	1.35	$24.94
4	Internet	$22.48	1.35	$30.34
5	Web Design	$22.48	1.20	$26.97
6	Internet	$17.48	1.35	$23.59
7	Internet	$25.48	1.35	$34.39
8	Web Design	$22.48	1.20	$26.97
9	Web Design	$22.98	1.20	$27.57
10	Internet	$22.98	1.35	$31.02
11	Internet	$24.57	1.35	$33.17
12	Internet	$22.98	1.35	$31.02
13	Internet	$22.48	1.35	$30.34
14	Internet	$22.48	1.35	$30.34
15	Internet	$22.98	1.35	$31.02
...				

Example 9.3: Using the Resolution of a Character Expression As an Argument to SYMGET

The DATA step in Program 9.3 resolves SAS language character expressions to obtain the names and values of macro variables. The goal of the program is to obtain the manager's initials for the quarter in which a book was sold. Preceding the DATA step, four %LET statements create four macro variables, one for the manager's initials in each quarter.

As the DATA step processes each observation in BOOK.YTDSALES, the SYMGET function extracts a value from one of the four macro variables based on the current observation's value of the data set variable DATESOLD. This value is assigned to data set variable MANAGERINITS. The quarter of the sale date is determined and the value of quarter (1, 2, 3, or 4) determines from which macro variable the SYMGET function retrieves a value.

The DATA step assigns a length of 3 bytes to MANAGERINITS and overrides the default length of 200 bytes.

Program 9.3

```
%let managerquarter1=HCH;
%let managerquarter2=EMB;
%let managerquarter3=EMB;
%let managerquarter4=JBR;

data managers;
  set books.ytdsales;

  length managerinits $ 3;

  managerinits=
     symget(cats('managerquarter',put(qtr(datesold),1.)));
run;
proc print data=managers;
  title "Sale Dates and Managers";
  var datesold managerinits;
run;
```

A partial listing of the PROC PRINT output (Output 9.3) shows the values assigned to MANAGERINITS by SYMGET.

Output 9.3 Output from Program 9.3, which resolves character expressions as arguments to the SYMGET function

```
                    Sale Dates and Managers

                  Obs      datesold    managerinits

                    1    01/18/2007        HCH
                    2    01/07/2007        HCH
                    3    01/24/2007        HCH
                    4    01/20/2007        HCH
. . .
                 2353    04/28/2007        EMB
                 2354    04/15/2007        EMB
                 2355    04/23/2007        EMB
                 2356    04/06/2007        EMB
. . .
                 3549    08/23/2007        EMB
                 3550    08/29/2007        EMB
                 3551    08/12/2007        EMB
                 3552    08/15/2007        EMB
. . .
                 4621    10/06/2007        JBR
                 4622    10/08/2007        JBR
                 4623    10/30/2007        JBR
                 4624    10/27/2007        JBR
. . .
```

Understanding the SYMPUT and SYMPUTX Call Routines

The SYMPUT and SYMPUTX SAS language call routines create macro variables during execution of a DATA step. If the macro variable already exists, these routines update the value of the macro variable.

CALL SYMPUT. The syntax of the SYMPUT routine is:

```
CALL SYMPUT(macro-variable,text)
```

CALL SYMPUT does not trim leading and trailing blanks from the value assigned to the macro variable. The two arguments to CALL SYMPUT can each be specified in one of three ways:

- as *literal text*. The following SAS language statement creates or updates the macro variable BOOKSECT with the value `Internet`. Since CALL SYMPUT is a SAS language routine, you must enclose literal text arguments in quotation marks.

```
call symput('booksect','Internet');
```

- as the *name of a data set character variable* whose value is a SAS variable name. The current value of the data set variable NHIGH is assigned to the macro variable N45. The name of the macro variable, N45, is saved in DATA step character variable RESULTVAR.

```
resultvar='n45';
call symput(resultvar,nhigh);
```

- as a *character expression*. The first argument to CALL SYMPUT below defines a macro variable name where the first part of the name is equal to the text AUTHORNAME. The second part of the macro variable name is equal to the automatic variable _N_. The second argument resolves to a text string. The literal text in the first part of the string and the current observation's value for AUTHOR are concatenated. During the fifth iteration of the DATA step that contains this statement, CALL SYMPUT would define a macro variable named AUTHORNAME5.

```
call symput(cats('authorname',put(_n_,4.)),
            cat('Author Name: ',author));
```

CALL SYMPUTX. The syntax of the CALL SYMPUTX routine is:

```
CALL SYMPUTX(macro-variable,text<,symbol-table>)
```

CALL SYMPUTX trims leading and trailing blanks from the value assigned to the macro variable. The first two arguments to CALL SYMPUTX are specified the same way as for CALL SYMPUT. The third argument is optional and it tells the macro processor the symbol table where the macro variable should be stored. This argument can be specified as a character constant, data set variable, or expression. The first non-blank letter in this optional argument determines where the macro processor stores the macro variable. Valid values for this optional argument can be one of three values:

- `G`, which specifies that the macro processor store the macro variable in the global symbol table even if the local symbol table exists.
- `L`, which specifies that the macro processor store the macro variable in the most local symbol table. If a macro program is not executing when this option is specified, there will be no local symbol table. In such a situation, the most local

symbol table is actually the global symbol table, and that is where the macro processor stores the macro variable.

- `F`, which specifies that if the macro variable exists in any symbol table, CALL SYMPUTX should update the macro variable's value in the most local symbol table in which it exists. If it does not exist in any symbol table, CALL SYMPUTX stores the macro variable in the most local symbol table.

Each of these two call routines updates the value of an existing macro variable. A macro variable can only have one value. Even though your DATA step might cause the call routine to be executed with each pass of the DATA step, the macro variable that the routines reference can still have only one value. When the DATA step ends, the value of the macro variable being updated has the last value that was assigned by SYMPUT or SYMPUTX.

Example 9.4: Saving the Sum of a Variable in a Macro Variable by Executing CALL SYMPUT Once at the End of a DATA Step

In Program 9.4, CALL SYMPUT creates a macro variable N45 whose value is the total number of books that sold for at least $45.00. The program then places this tally in the title of a PROC MEANS report.

The program directs that CALL SYMPUT execute once when the DATA step reaches the end of the data set and store the formatted value of data set variable NHIGH in macro variable N45.

The PUT function formats the value assigned to N45 with the COMMA format. In applying the format to NHIGH, the value is converted from numeric to character. With the width of the format set at five characters, no leading blanks in the value pass to N45.

If the DATA step were written so that CALL SYMPUT executed with each pass of the DATA step, the macro variable value would be updated with each observation. Since the goal is to obtain the total number of books sold for more than $45.00, it is necessary to execute CALL SYMPUT only once after the tally is complete. The second IF statement directs that the CALL SYMPUT routine execute only when the DATA step reaches the end of data set BOOKS.YTDSALES.

Program 9.4

```
data _null_;
  set books.ytdsales end=eof;
  if saleprice ge 45 then nhigh+1;
  if eof then call symput('n45',put(nhigh,comma5.));
run;
```

```
proc means data=temp n mean min max;
  title "All Books Sold";
  title2 "Number of Books Sold for More Than $45: &n45";
  var saleprice;
run;
```

Output 9.4 presents the PROC MEANS report. The title includes the total number of books sold for at least $45.00.

Output 9.4 Output from Program 9.4, which calls SYMPUT once at the end of a DATA step

```
                          All Books Sold
          Number of Books Sold for More Than $45: 2,141

                       The MEANS Procedure

             Analysis Variable : saleprice Sale Price

          N            Mean         Minimum         Maximum
       ------------------------------------------------------
       6096      43.2542897      24.7600000      86.9500000
       ------------------------------------------------------
```

When you assign a numeric variable value to a macro variable with CALL SYMPUT, the numeric value is converted to character by default, and then this character value is stored in the macro variable. The default width of the character field passed to the macro variable is 12 characters and a numeric value is right aligned. For example, consider what happens when the CALL SYMPUT routine is modified and the PUT function is removed.

```
if eof then call symput('n45',nhigh);
```

The title now looks like the following and eight leading blanks precede the four-digit numeric value.

```
Number of Books Sold for More Than $45:         2141
```

Execution of this version of the CALL SYMPUT function causes the following note to be written to the SAS log:

```
NOTE: Numeric values have been converted to character
      values at the places given by: (Line):(Column).
```

You could also use the CALL SYMPUTX function. The note is not written to the SAS log, and CALL SYMPUTX removes the leading blanks before the value is stored in macro variable N45.

```
if eof then call symputx('n45',nhigh);
```

The title now looks like the following.

```
Number of Books Sold for More Than $45: 2141
```

Example 9.5: Executing CALL SYMPUTX Multiple Times in a DATA Step

In Program 9.5, CALL SYMPUTX executes with each pass of the DATA step, which is once for each record in the data lines. The value of the macro variable at the end of the DATA step is the value from the last observation read from the raw data.

Program 9.5

```
data newbooks;
  input booktitle $ 1-40;
  call symputx('lasttitle',booktitle);
datalines;
Hello Java Programming
My Encyclopedia of Networks
Strategic Computer Programming
Everyday Email Etiquette
run;
%put The value of macro variable LASTTITLE is &lasttitle..;
```

The %PUT statement writes the following to the SAS log.

```
The value of macro variable LASTTITLE is Everyday Email
Etiquette.
```

Example 9.6: Creating Several Macro Variables with CALL SYMPUT and CALL SYMPUTX

This example creates multiple macro variables with CALL SYMPUT and CALL SYMPUTX. This program creates two macro variables for each section in the output data set produced by PROC FREQ. PROC FREQ saves six observations in the SECTNAME output data set, one for each section in the BOOKS.YTDSALES data set. Therefore, the DATA step creates 12 macro variables. Six macro variables hold the names of the six sections. The other six macro variables hold the frequency counts for each of the sections. A %PUT _USER_ following the DATA step lists the 12 macro variables created in the DATA step.

Program 9.6

```
proc freq data=books.ytdsales noprint;
   tables section / out=sectname;
run;
data _null_;
  set sectname;
  call symput('name' || put(_n_,1.),section);
  call symputx('n' || put(_n_,1.),count);
run;

%put _user_;
```

The following %PUT _USER_ output displays the values of the macro variables defined in the DATA step. Note that CALL SYMPUTX trims the leading blanks from the values assigned to macro variables N1 through N6.

```
GLOBAL N1 726
GLOBAL NAME1 Certification and Training
GLOBAL N2 1456
GLOBAL NAME2 Internet
GLOBAL N3 717
GLOBAL NAME3 Networks and Telecommunication
GLOBAL N4 922
GLOBAL NAME4 Operating Systems
GLOBAL N5 1429
GLOBAL NAME5 Programming and Applications
GLOBAL N6 846
GLOBAL NAME6 Web Design
```

Example 9.7: Creating a Macro Variable with CALL SYMPUTX and Specifying Its Symbol Table

This example computes statistics on a subset of a data set and assigns the values of the statistics to global macro variables. The goal is to make these macro variables global so that they are available for subsequent processing.

Macro program STATSECTION in Program 9.7 computes with PROC MEANS the mean, minimum, and maximum sale price for a specific section in BOOKS.YTDSALES; the program saves the statistics in output data set SECTIONRESULTS. The parameter SECTION passes to STATSECTION the name of the section for which to compute the statistics.

The PROC MEANS step does not print a report, but it does save the three statistics in an output data set. A DATA step processes the output data set. It uses CALL SYMPUTX to create three macro variables to hold the three statistics and to assign values to the macro

variables. Additionally, CALL SYMPUTX specifies that the macro variables be stored in the global symbol table.

The three global macro variables created by this program are AVERAGE, MIN, and MAX. Three CALL SYMPUTX statements store the formatted values of the statistics in these macro variables.

If you did not specify that the macro variables should be stored in the global symbol table, they would be stored in the local symbol table defined by macro program STATSECTION. Once STATSECTION completed processing, its local symbol table would be deleted, and the values of the three macro variables would be lost.

A %PUT _LOCAL_ statement is included in the macro program to show that the only macro variable stored in the STATSECTION local macro symbol table is SECTION. (You could also use the %SYMGLOBL and %SYMLOCAL macro variable attribute functions described in Chapter 6 to determine whether a macro variable was stored globally or locally.)

Three TITLE statements follow the call to STATSECTION. These TITLE statements include references to the three macro variables created in STATSECTION.

Program 9.7

```
%macro statsection(section);
  proc means data=books.ytdsales noprint;
    where section="&section";
    var saleprice;
    output out=sectionresults mean=avgsaleprice
             min=minsaleprice max=maxsaleprice;
  run;

  data _null_;
    set sectionresults;

    call symputx('average',put(avgsaleprice,dollar8.2),'G');
    call symputx('min',put(minsaleprice,dollar8.2),'G');
    call symputx('max',put(maxsaleprice,dollar8.2),'G');
  run;

  %* Submit this statement to see the variables stored in the
     STATSECTION local symbol table;
  %put _local_;
%mend;

%statsection(Internet)

title "Section Results for Average Sale Price: &average";
```

```
title2 "Minimum Sale Price: &min";
title3 "Maximum Sale Price: &max";
```

Execution of the %PUT _LOCAL_ statement writes the following to the SAS log. The text "STATSECTION" refers to the name of the local symbol table.

```
STATSECTION SECTION Internet
```

After executing %STATSECTION and the subsequent TITLE statements, the titles become:

```
Section Results for Average Sale Price: $42.79
Minimum Sale Price: $24.76
Maximum Sale Price: $86.95
```

If you dropped the third argument in the three CALL SYMPUTX calls, the three macro variables would be stored in the STATSECTION local symbol table. When STATSECTION ends, the macro processor deletes the STATSECTION local symbol table; the values of the three macro variables are not available for insertion into the titles.

The three DATA step statements would be rewritten as follows.

```
call symputx('average',put(avgsaleprice,dollar8.2));
call symputx('min',put(minsaleprice,dollar8.2));
call symputx('max',put(maxsaleprice,dollar8.2));
```

The %PUT _LOCAL_ statement would now produce the following output in the SAS log.

```
STATSECTION MIN $24.76
STATSECTION MAX $86.95
STATSECTION SECTION Internet
STATSECTION AVERAGE $42.79
```

The references to the three macro variables in the TITLE statements cannot be resolved because the macro variables do not exist in the global symbol table. With the three macro variables stored in the local macro symbol table, the three titles become:

```
Section Results for Average Sale Price: &average
Minimum Sale Price: &min
Maximum Sale Price: &max
```

NOTE: If you want to try out the code that does not specify that the macro variables should be saved in the global symbol table and want to see the unresolved macro variable results in the titles, make sure you delete the three macro variables from the global symbol table. You can delete the three macro variables with the %SYMDEL statement.

```
%symdel average min max;
```

Understanding the CALL EXECUTE Routine

The CALL EXECUTE SAS language routine is a tool that you can use to specify execution of macro programs from within the DATA step. It takes as its argument a character expression or constant text that it resolves to a macro program invocation or SAS statement. This section describes how to use it to invoke macro programs. For detailed information on generating SAS statements without referencing a macro program, refer to SAS documentation.

When the argument to CALL EXECUTE is a macro program reference, that macro program executes immediately during execution of the DATA step. If that macro program generates SAS statements, however, those statements execute *after* the DATA step finishes.

The syntax of the CALL EXECUTE routine is

```
call execute('argument')
```

The three types of arguments that can be supplied to the routine are:

- a *text string enclosed in quotation marks*. Single quotation marks and double quotation marks are handled differently. Single quotation marks cause the argument to be resolved when the DATA step executes. Double quotation marks cause the argument to be resolved by the macro processor during construction of the DATA step, *before* compilation and execution of the DATA step.
- the *name of a data set character variable*. This variable's value can be a text expression or a SAS language statement. Do not enclose the variable name in quotation marks.
- a *text expression* that the DATA step can resolve to a SAS language statement or to a macro variable, macro language statement, or macro program reference.

Example 9.8: Illustrating the Timing of CALL EXECUTE When It Invokes a Macro Program That Submits Macro Statements

This simple example in Program 9.8 demonstrates how a macro program invoked by CALL EXECUTE executes immediately within the DATA step. A macro program LISTAUTOMATIC issues a %PUT statement that lists the automatic variables. The DATA step that follows contains a CALL EXECUTE statement that explicitly invokes LISTAUTOMATIC.

Program 9.8

```
%macro listautomatic;
  %put **** Start list of automatic macro variables;
  %put _automatic_;
  %put **** End list of automatic macro variables;
%mend listautomatic;

data _null_;
  call execute("%listautomatic");
run;
```

The SAS log for this program shows that the macro program executes *during* execution of the DATA step. The concluding DATA step processing notes follow the list of automatic variables.

```
48   %macro listautomatic;
49     %put **** Start list of automatic macro variables;
50     %put _automatic_;
51     %put **** End list of automatic macro variables;
52   %mend listautomatic;
53
54   data _null_;
55     call execute("%listautomatic");
**** Start list of automatic macro variables
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
AUTOMATIC AFSTR1
AUTOMATIC AFSTR2
AUTOMATIC FSPBDV
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCC 0
AUTOMATIC SYSCHARWIDTH 1
AUTOMATIC SYSCMD
```

```
AUTOMATIC SYSDATE 15FEB06
AUTOMATIC SYSDATE9 15FEB2006
.
.
.
AUTOMATIC SYSTIME 08:20
AUTOMATIC SYSUSERID My Userid
AUTOMATIC SYSVER 9.1
AUTOMATIC SYSVLONG 9.01.01M3P061705
AUTOMATIC SYSVLONG4 9.01.01M3P06172005
**** End list of automatic macro variables
56   run;

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: CALL EXECUTE routine executed successfully, but no SAS
statements were generated.
```

Example 9.9: Illustrating the Timing of CALL EXECUTE When It Invokes a Macro Program That Submits Macro Statements and a PROC Step

Program 9.9 defines and invokes a macro program LISTLIBRARY that submits two %PUT statements and a PROC DATASETS step. It demonstrates how a macro program invoked by CALL EXECUTE executes immediately within the DATA step and that SAS statements generated by the macro program execute after the DATA step.

The macro program LISTLIBRARY issues a PROC DATASETS for the BOOKS library. Immediately preceding the PROC step, it submits a %PUT statement. Immediately after the PROC step, it submits a second %PUT statement. The DATA step contains a CALL EXECUTE statement that explicitly invokes LISTLIBRARY.

Program 9.9

```
%macro listlibrary;
  %put **** This statement precedes the PROC step;
  proc datasets library=books;
  run;
  quit;
  %put **** This statement follows the PROC step;
%mend listlibrary;
data _null_;
  call execute("%listlibrary");
run;
```

The SAS log for this program shows that the macro program executes during execution of the DATA step. The text written by the %PUT statements appears in the SAS log during execution of the DATA step. The PROC DATASETS output, however, does not appear until after the DATA step concludes.

During execution of the DATA step, the macro processor directs immediate execution of the two %PUT statements while it places the PROC step on the input stack for execution *after* the DATA step concludes.

```
91   %macro listlibrary;
92     %put **** This statement precedes the PROC step;
93     proc datasets library=books;
94     run;
95     quit;
96     %put **** This statement follows the PROC step;
97   %mend listlibrary;
98   data _null_;
99     call execute("%listlibrary");
**** This statement precedes the PROC step
**** This statement follows the PROC step
100  run;

NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


NOTE: CALL EXECUTE generated line.
1   + proc datasets library=books;
                            Directory

           Libref         BOOKS
           Engine         V9
           Physical Name  f:\books
           File Name      f:\books


                            Member      File
               #  Name      Type        Size  Last Modified

               1  YTDSALES  DATA     1393664  02Feb08:16:21:13
1   +                                 run;

1   +                                       quit;
```

```
NOTE: PROCEDURE DATASETS used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Example 9.10: Using CALL EXECUTE to Conditionally Call a Macro Program

Program 9.10 uses CALL EXECUTE to conditionally execute the macro program REP60K. First, PROC MEANS computes the sales for each section in BOOKS.YTDSALES and stores the results in an output data set. A DATA step processes the output data set and examines the total sales per section. When the sales exceed $60,000, the CALL EXECUTE statement in the DATA step calls macro program REP60K.

The argument to CALL EXECUTE is the macro program name, REP60K. This macro program has one parameter, SECTION. The argument to CALL EXECUTE is a text expression that resolves to the call to REP60K with the parameter specified as the current value of SECTION. The CATS function concatenates the parts of the macro program call.

The SAS language statements in the macro program execute when the DATA step finishes. In this example, the CALL EXECUTE routine executes as many times as there are observations that satisfy the IF statement condition. Each time CALL EXECUTE executes, it calls the macro program REP60K for the section value of the current observation. Therefore, when the DATA step finishes, there can be several PROC REPORT steps on the input stack ready to process.

In this example, there are two sections, “Internet” and “Programming and Applications” where the total sales exceeded $60,000. Thus, two PROC REPORT steps execute after the DATA step.

Program 9.10

```
%macro rep60k(section);
  proc report data=books.ytdsales headline center nowd;
    where section="&section";
    title "Sales > $60,000 Summary for &section";
    column  publisher n saleprice;
    define  publisher / group;
    define  n / "Number of Books Sold" ;
    define  saleprice / sum format=dollar10.2 "Sale Price" ;
    rbreak after / summarize dol;
  run;
%mend rep60k;

options mprint;
```

```
proc means data=books.ytdsales nway noprint;
  class section;
  var saleprice;
  output out=sectsale sum=totlsale;
run;

data _null_;
  set sectsale;

  if totlsale > 60000 then
    call execute(cats('%rep60k(',section,')'));
run;
```

The SAS log for this program shows that two PROC REPORT steps execute after completion of the DATA step. The option MPRINT is in effect and shows the code for the two PROC REPORT steps. Note that the processing of CALL EXECUTE also lists the code for the PROC REPORT steps.

```
28   %macro rep60k(section);
29     proc report data=books.ytdsales headline center nowd;
30       where section="&section";
31       title "Sales > $60,000 Summary for &section";
32       column  publisher n saleprice;
33       define  publisher / group;
34       define  n / "Number of Books Sold" ;
35       define  saleprice / sum format=dollar10.2
35 ! "Sale Price";
36       rbreak after / summarize dol;
37     run;
38   %mend rep60k;
39
40   proc means data=books.ytdsales nway noprint;
41     class section;
42     var saleprice;
43     output out=sectsale sum=totlsale;
44   run;

NOTE: There were 6096 observations read from the data set
      BOOKS.YTDSALES.
NOTE: The data set WORK.SECTSALE has 6 observations and 4
      variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.23 seconds
      cpu time            0.03 seconds

45
46   data _null_;
```

```
47     set sectsale;
48
49     if totlsale > 60000 then
50       call execute(cats('%rep60k(',section,')'));
51   run;

MPRINT(REP60K):   proc report data=books.ytdsales headline
center nowd;
MPRINT(REP60K):   where section="Internet";
MPRINT(REP60K):   title "Sales > $60,000 Summary for Internet";
MPRINT(REP60K):   column publisher n saleprice;
MPRINT(REP60K):   define publisher / group;
MPRINT(REP60K):   define n / "Number of Books Sold" ;
MPRINT(REP60K):   define saleprice / sum format=dollar10.2
"Sale Price" ;
MPRINT(REP60K):   rbreak after / summarize dol;
MPRINT(REP60K):   run;
MPRINT(REP60K):   proc report data=books.ytdsales headline
center nowd;
MPRINT(REP60K):   where section="Programming and Applications";
MPRINT(REP60K):   title "Sales > $60,000 Summary for
Programming and Applications";
MPRINT(REP60K):   column publisher n saleprice;
MPRINT(REP60K):   define publisher / group;
MPRINT(REP60K):   define n / "Number of Books Sold" ;
MPRINT(REP60K):   define saleprice / sum format=dollar10.2
"Sale Price" ;
MPRINT(REP60K):   rbreak after / summarize dol;
MPRINT(REP60K):   run;
NOTE: There were 6 observations read from the data set
      WORK.SECTSALE.
NOTE: DATA statement used (Total process time):
      real time           0.10 seconds
      cpu time            0.00 seconds

NOTE: CALL EXECUTE generated line.
1   + proc report data=books.ytdsales headline center nowd;
where section="Internet";     title "Sales > $60,000 Summary
for Internet";     column  publisher n saleprice;     define
publisher / group;     define  n / "Number of Books Sold" ;
define
2   + saleprice / sum format=dollar10.2 "Sale Price" ;
rbreak after / summarize dol;   run;

NOTE: There were 1456 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Internet';
```

```
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.57 seconds
      cpu time            0.00 seconds

3   + proc report data=books.ytdsales headline center nowd;
where section="Programming and Applications";     title "Sales
> $60,000 Summary for Programming and Applications";     column
 publisher n saleprice;     define  publisher / group;
define  n /
4   + "Number of Books Sold" ;     define  saleprice / sum
format=dollar10.2 "Sale Price" ;     rbreak after / summarize
dol;   run;

NOTE: There were 1429 observations read from the data set
      BOOKS.YTDSALES.
      WHERE section='Programming and Applications';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Example 9.11: Using CALL EXECUTE to Call a Specific Macro Program

This example shows how you can conditionally call different macro programs during the execution of a DATA step. The PROC steps constructed by the macro program execute after the DATA step ends.

Program 9.11 defines two macro programs: LOWREPORT and HIGHREPORT. Each macro program generates a different PROC REPORT step.

As in Example 9.10, PROC MEANS computes total sales by section and saves the results in an output data set. A DATA step examines the results and, depending on the value of the total sales, it determines whether one of the two macro programs should be executed. If sales exceed $60,000, then CALL EXECUTE specifies a call to macro program HIGHREPORT. If sales are less than $35,000, then CALL EXECUTE specifies a call to macro program LOWREPORT. Neither macro program is called if sales are between $35,000 and $60,000.

The argument to each of the two CALL EXECUTE references is a text expression that resolves either to a call to HIGHREPORT or to a call to LOWREPORT. Both macro programs have the same parameter, SECTION. The current observation's value of SECTION is specified as part of the text expression that resolves to the macro program call. The CATS function concatenates the parts of the macro program call.

In this example, two sections, "Internet" and "Programming and Applications," exceed total sales of $60,000. The DATA step calls macro program HIGHREPORT twice, once for each of these sections.

Two sections, "Certification and Training" and "Networks and Telecommunications," have total sales less than $35,000. The DATA step calls macro program LOWREPORT twice, once for each of these sections.

Program 9.11

```
%macro highreport(section);
   proc report data=books.ytdsales headline center nowd;
    where section="&section";
    title "Sales > $60,000 Report for Section &section";
    column  publisher n saleprice;
    define  publisher / group;
    define  n / "Number of Books Sold" ;
    define  saleprice / sum format=dollar10.2 "Sale Price" ;
    rbreak after / summarize dol;
  run;
%mend highreport;

%macro lowreport(section);
   proc report data=books.ytdsales nowd;
     where section="&section";
     title "Sales < $35,000 Report for Section &section";
     column datesold n saleprice;
     define datesold / group format=month. "Month Sold"
                       width=6;
     define n / "Number of Books Sold";
     define saleprice / sum format=dollar10.2 "Sales Total";
     rbreak after / summarize dol;
   run;
%mend lowreport;

proc means data=books.ytdsales nway noprint;
  class section;
  var saleprice;
  output out=sectsale sum=totlsect;
run;

data _null_;
  set sectsale;
  if totlsect < 35000 then
    call execute(cats('%lowreport(',section,')'));
```

```
  else if totlsect > 60000 then
    call execute(cats('%highreport(',section,')'));
run;
```

The SAS log for this program shows the four calls to the two macro programs. Compared to Example 9.10, MPRINT is not in effect when Program 9.11 executes. The results of the CALL EXECUTE call, however, are displayed.

```
847  %macro highreport(section);
848     proc report data=books.ytdsales headline center nowd;
849      where section="&section";
850      title "Sales > $60,000 Report for Section &section";
851      column  publisher n saleprice;
852      define  publisher / group;
853      define  n / "Number of Books Sold" ;
854      define  saleprice / sum format=dollar10.2 "Sale Price";
855      rbreak after / summarize dol;
856    run;
857  %mend highreport;
858
859  %macro lowreport(section);
860     proc report data=books.ytdsales nowd;
861       where section="&section";
862       title "Sales < $35,000 Report for Section &section";
863       column datesold n saleprice;
864       define datesold / group format=month. "Month Sold"
865                         width=6;
866       define n / "Number of Books Sold";
867       define saleprice / sum format=dollar10.2 "Sales Total";
868       rbreak after / summarize dol;
869     run;
870  %mend lowreport;
871
872  proc means data=books.ytdsales nway noprint;
873    class section;
874    var saleprice;
875    output out=sectsale sum=totlsect;
876  run;

NOTE: There were 6096 observations read from the data set
BOOKS.YTDSALES.
NOTE: The data set WORK.SECTSALE has 6 observations and 4
variables.
```

```
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds

877
878  data _null_;
879    set sectsale;
880    if totlsect < 35000 then
881      call execute(cats('%lowreport(',section,')'));
882    else if totlsect > 60000 then
883      call execute(cats('%highreport(',section,')'));
884  run;

NOTE: There were 6 observations read from the data set
WORK.SECTSALE.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

NOTE: CALL EXECUTE generated line.
1   + proc report data=books.ytdsales nowd;      where
section="Certification and Training";
   title "Sales < $35,000 Report for Section Certification and
Training";      column datesold
n saleprice;      define datesold / group format=month. "Month
Sold"
2   +                     width=6;      define n / "Number of
Books Sold";      define
saleprice / sum format=dollar10.2 "Sales Total";      rbreak
after / summarize dol;    run;

NOTE: There were 726 observations read from the data set
BOOKS.YTDSALES.
      WHERE section='Certification and Training';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds

3   + proc report data=books.ytdsales headline center nowd;
where section="Internet";
title "Sales > $60,000 Report for Section Internet";     column
publisher n saleprice;
define  publisher / group;     define  n / "Number of Books
Sold" ;
4   + define  saleprice / sum format=dollar10.2 "Sale Price" ;
rbreak after / summarize
dol;   run;
```

```
NOTE: There were 1456 observations read from the data set
BOOKS.YTDSALES.
      WHERE section='Internet';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

5   + proc report data=books.ytdsales nowd;      where
section="Networks and
Telecommunication";      title "Sales < $35,000 Report for
Section Networks and
Telecommunication";      column datesold n saleprice;
define datesold / group
format=month. "Month
6   + Sold"                         width=6;      define n /
"Number of Books Sold";
define saleprice / sum format=dollar10.2 "Sales Total";
rbreak after / summarize dol;
run;

NOTE: There were 717 observations read from the data set
BOOKS.YTDSALES.
      WHERE section='Networks and Telecommunication';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

7   + proc report data=books.ytdsales headline center nowd;
where section="Programming and
Applications";     title "Sales > $60,000 Report for Section
Programming and Applications";
 column  publisher n saleprice;     define  publisher / group;
8   + define  n / "Number of Books Sold" ;     define
saleprice / sum format=dollar10.2 "Sale
Price" ;     rbreak after / summarize dol;   run;

NOTE: There were 1429 observations read from the data set
BOOKS.YTDSALES.
      WHERE section='Programming and Applications';
NOTE: PROCEDURE REPORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Understanding the RESOLVE Function

The RESOLVE SAS language function can resolve a macro variable reference or macro program call during execution of a DATA step. The result returned by RESOLVE is a

character value whose length equals that of the data set character variable to which the result is being assigned. It takes as its argument a text expression, a data set character variable, or a character expression that produces a text expression that can be resolved by the macro processor.

The RESOLVE function acts during execution of a DATA step. Therefore, a RESOLVE function could be coded so that it executes with each pass of a DATA step, and at each execution it could return a different value.

The RESOLVE function is similar to the SYMGET function in that both functions can resolve macro variable references during execution of a DATA step. However, the SYMGET function is more limited in its functionality and in the arguments it can accept. The RESOLVE function can accept a variety of different types of arguments, including macro variables and macro program calls, while SYMGET's sole function is to resolve a macro variable reference.

The SYMGET function can resolve macro variables only if they exist before execution of the DATA step in which the function is included. An exception to this is if a statement containing CALL SYMPUT or CALL SYMPUTX that defines the macro variable executes before the SYMGET function. As stated above, the RESOLVE function acts during execution, and a macro variable defined in a DATA step with CALL SYMPUT or CALL SYMPUTX can be retrieved in the same DATA step with the RESOLVE function.

The syntax of the RESOLVE function is

```
resolve('argument')
```

The three types of arguments that RESOLVE accepts are:

- a *text expression that is enclosed in single quotation marks*. This text expression can be a macro variable reference, an open code macro language statement, or a macro program call. If you enclose the text expression in double quotation marks, the macro processor will attempt to resolve it *before* the DATA step is compiled, while the SAS program is being constructed. Enclosing the argument in single quotation marks *delays* resolution until the DATA step executes.

  ```
  dsvar=resolve('&macvar');
  ```

- the *name of a data set character variable*. The value of this variable is a text expression representing a macro variable reference, an open code macro language statement, or a macro program call.

  ```
  %let label1=All the Books Sold;
  data temp;
    length textlabel $ 40;
  ```

```
  macexp='&label1';
  textlabel=resolve(macexp);
run;
```

- a *character expression that can be resolved to a text expression*. The text expression represents a macro variable reference, an open code macro language statement, or a macro program call.

```
%let quartersale1=Holiday Clearance;
%let quartersale2=2 for the Price of 1;
%let quartersale3=Back to School;
%let quartersale4=New Releases;
data temp;
  set books.ytdsales;
  length quartersalename $ 30;

  quarter=qtr(datesold);
  quartersalename=resolve(
       cats('&quartersale',put(quarter,1.)) );
run;
```

By default, the length of the text value returned by RESOLVE is 200 bytes. If you want the character variable that holds the result to be different than 200 bytes, you must explicitly define the length for the variable. This can be done with the LENGTH statement or with the ATTRIB statement.

Example 9.12: Obtaining Macro Variable Values with RESOLVE by Resolving Character Expressions

The code in Program 9.12 was presented above in the discussion on the types of arguments that RESOLVE can accept. The goal of the program is to create a character variable that contains the name of the sale in the quarter in which an item was sold. Sale names are stored in macro variables. The RESOLVE function in the DATA step looks up the correct sale name based on the quarter the item was sold. The argument to the RESOLVE function is a character expression that resolves to a macro variable name.

Program 9.12 defines four macro variables, QUARTERSALE1, QUARTERSALE2, QUARTERSALE3, and QUARTERSALE4, which contain the name of each quarter's sale. The DATA step processes data set BOOKS.YTDSALES and determines the quarter in which each item sold. A macro variable name is constructed by concatenating the text part of the macro variable name, QUARTERSALE, to the quarter number. This expression is the argument to RESOLVE. RESOLVE returns the value of the specific macro variable and assigns this value to data set character variable QUARTERSALENAME.

Note that the text &QUARTERSALE is enclosed in single quotation marks. The single quotation marks prevent the macro processor from attempting to resolve a macro variable with that name during compilation of the DATA step. Instead, no action is taken during compilation of the DATA step to resolve the macro variable reference.

Output 9.5 shows the results of the PROC FREQ crosstabulation of QUARTER and QUARTERSALENAME.

Program 9.12

```
%let quartersale1=Holiday Clearance;
%let quartersale2=2 for the Price of 1;
%let quartersale3=Back to School;
%let quartersale4=New Releases;

data temp;
  set books.ytdsales;
  length quartersalename $ 30;

  quarter=qtr(datesold);
  quartersalename=resolve(
       cats('&quartersale',put(quarter,1.)) );
run;
proc freq data=temp;
  title 'Quarter by Quarter Sale Name';
  tables quarter*quartersalename / list nocum nopct;
run;
```

Output 9.5 presents the PROC FREQ results produced by Program 9.12.

Output 9.5 PROC FREQ output produced by Program 9.12, which updates variables with the RESOLVE function

Quarter by Quarter Sale Name

The FREQ Procedure

quarter	quartersalename	Frequency
1	Holiday Clearance	2042
2	2 for the Price of 1	975
3	Back to School	1355
4	New Releases	1724

Example 9.13: Using RESOLVE to Call a Macro Program within a DATA Step That Assigns Text to a Data Set Variable

This example shows how you can call a macro program from within a DATA step with the RESOLVE function. The macro program executes with each pass of the DATA step returning text to the DATA step.

As in Example 9.12, Program 9.13 looks up a sale name based on the quarter. In this example, PROC MEANS computes total sales by quarter for variable SALEPRICE and saves the results in an output data set. A DATA step processes the output data set created by PROC MEANS. The RESOLVE function executes the same macro program with each pass of the DATA step. The value of quarter is passed as a parameter to the macro program.

The definition for macro program GETSALENAME precedes the PROC MEANS step. When called by the RESOLVE function, macro program GETSALENAME simply looks up a text value based on the value of parameter QUARTER and returns this text to the DATA step. The assignment statement in the DATA step assigns this text value to data set variable QUARTERSALENAME.

As in Program 9.12, the argument to the RESOLVE function in Program 9.13 is constructed during execution of the DATA step, and the argument is enclosed in single quotation marks to prevent the macro processor from attempting to resolve the call to the macro program during compilation of the DATA step.

Program 9.13

```
%macro getsalename(quarter);
  %if &quarter=1 %then %do;
    Holiday Clearance
  %end;
  %else %if &quarter=2 %then %do;
    2 for the Price of 1
  %end;
  %else %if &quarter=3 %then %do;
    Back to School
  %end;
  %else %if &quarter=4 %then %do;
    New Releases
  %end;
%mend getsalename;

proc means data=books.ytdsales noprint nway;
  class datesold;
  var saleprice;
  output out=quarterly sum=;
  format datesold qtr.;
run;

data quarterly;
  set quarterly(keep=datesold saleprice);

  length quartersalename $ 30;

  quartersalename=resolve(
cats('%getsalename(',put(datesold,qtr.),')') );
run;
proc print data=quarterly label;
  title 'Quarter Sales with Quarter Sale Name';
  label datesold='Quarter'
        saleprice='Total Sales'
        quartersalename='Sale Name';
run;
```

Output 9.6 presents the PROC PRINT report produced by Program 9.13.

Output 9.6 Output produced by Program 9.13, which uses the RESOLVE function to assign text to a variable

Quarter Sales with Quarter Sale Name

Obs	Quarter	Total Sales	Sale Name
1	1	$88,150.75	Holiday Clearance
2	2	$42,446.50	2 for the Price of 1
3	3	$58,905.70	Back to School
4	4	$74,175.19	New Releases

Using Macro Facility Features in PROC SQL

Elements of PROC SQL can interface with the macro facility. During execution of a PROC SQL step, you can create and update macro variables. Additionally, with each execution of PROC SQL, the procedure creates and maintains three macro variables that hold information about the processing of PROC SQL. This section describes only the macro facility interface features of PROC SQL. For complete information on PROC SQL, refer to PROC SQL documentation.

Creating and Updating Macro Variables with PROC SQL

The INTO clause on the SELECT statement creates and updates macro variables. Calculations that are done with the SELECT statement, as well as entire data columns, can be saved in the macro variables that you name with the INTO clause.

The INTO clause is analogous to the CALL SYMPUT and CALL SYMPUTX routines in the DATA step. Like these routines, the INTO clause creates and updates macro variables during execution of a step. In the case of the INTO clause, the step is a PROC SQL statement.

The INTO clause provides a link to the macro variable symbol table during execution of PROC SQL. Values that are assigned to the macro variables are considered to be text.

The macro variables that you create with PROC SQL are added to the most local macro symbol table available when PROC SQL executes. If PROC SQL is not submitted from

within a macro program, the macro processor stores the macro variables in the global macro symbol table.

The basic syntax of the INTO clause on the PROC SQL SELECT statement follows:

```
SELECT col1,col2,...
       INTO :macvar1, :macvar2,...
       FROM table-expression
       WHERE where-expression
       other clauses;
```

Note the punctuation on the INTO clause: the macro variable names are preceded with colons (:), not ampersands (&). Macro variables are named explicitly on the INTO clause. Numbered lists of macro variables can also be specified on the INTO clause. Examples of both follow.

PROC SQL preserves leading or trailing blanks when you specify a single macro variable. Otherwise, when specifying a range of macro variables or when using the SEPARATED BY option, PROC SQL trims leading and trailing blanks from the values assigned to the macro variables, unless you follow the macro variable specifications with the NOTRIM option.

The INTO clause cannot be used during creation of a table or view. It can be used only on outer queries of the SELECT statement.

Example 9.14: Using the INTO Clause in PROC SQL to Save Summarizations in Macro Variables

A simple application of the INTO clause follows in Program 9.14. The PROC SQL SELECT statement computes the total sales and the total number of books sold for a specific publisher identified by macro variable FINDPUBLISHER. It stores the computations in two macro variables, TOTSALES and NSOLD. A %PUT statement following the step writes the values of these two global macro variables to the SAS log.

Program 9.14

```
%let findpublisher=Technology Smith;
proc sql noprint;
  select sum(saleprice) format=dollar10.2,
         count(saleprice)
    into :totsales, :nsold
    from books.ytdsales
    where publisher="&findpublisher";
quit;
%put &findpublisher Total Sales=&totsales, Total Number
Sold=&nsold;
```

The SAS log for the preceding program follows.

```
27   %let findpublisher=Technology Smith;
28   proc sql noprint;
29     select sum(saleprice) format=dollar10.2,
30            count(saleprice)
31       into :totsales, :nsold
32       from books.ytdsales
33       where publisher="&findpublisher";
34   quit;

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

35   %put &findpublisher Total Sales=&totsales, Total Number
35 ! Sold=&nsold;
Technology Smith Total Sales=$21,766.46, Total Number Sold=
505
```

Example 9.15: Demonstrating the Default Action of the INTO Clause in Saving the First Row of a Table

The default action of the PROC SQL INTO clause stores the first row of a table in the macro variables on the INTO clause. This example demonstrates that action.

Program 9.15 sorts the BOOKS.YTDSALES data set by DATESOLD and saves the sorted observations in data set DATESORTED. The PROC SQL step creates three macro variables DATE1, TITLE1, and PRICE1, and it sets their values to the values of data set variables DATESOLD, BOOKTITLE, and SALEPRICE for the first observation in DATESORTED. Three %PUT statements following the step write the values of these three global macro variables to the SAS log. A PROC PRINT of the first five observations of DATESORTED shows that the values assigned to the macro variables were from the first observation in DATESORTED.

Program 9.15

```
proc sort data=books.ytdsales out=datesorted;
  by datesold;
run;
proc sql noprint;
  select datesold,booktitle,saleprice
    into :date1,:title1,:price1
    from datesorted;
quit;

%put One of the first books sold was on &date1;
```

```
%put The title of this book is &title1;
%put The sale price was &price1;

proc print data=datesorted(obs=5);
  title
   'First Five Observations of Sorted by Date BOOKS.YTDSALES';
run;
```

The SAS log displays the values of the three macro variables. Note that the leading blanks were preserved in the display of macro variable PRICE1.

```
6611  proc sql noprint;
6612    select datesold,booktitle,saleprice
6613      into :date1,:title1,:price1
6614      from datesorted;
6615  quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

6616  %put One of the first books sold was on &date1;
One of the first books sold was on 01/01/2007
6617  %put The title of this book is &title1;
The title of this book is Operating Systems Title 301
6618  %put The sale price was &price1;
The sale price was     $45.95
6619  proc print data=datesorted(obs=5);
6620    title 'First Five Observations of Sorted by Date
BOOKS.YTDSALES';
6621  run;

NOTE: There were 5 observations read from the data set
WORK.DATESORTED.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The PROC PRINT output is displayed in Output 9.7.

Output 9.7 PROC PRINT output from Program 9.15, which shows the first five observations of data set DATESORTED

First Five Observations of Sorted by Date BOOKS.YTDSALES

Obs	section	saleid	saleinit	datesold
1	Operating Systems	10000152	BLT	**01/01/2007**
2	Internet	10000184	BLT	01/01/2007
3	Certification and Training	10000227	BLT	01/01/2007
4	Networks and Telecommunication	10000275	BLT	01/01/2007
5	Internet	10000280	BLT	01/01/2007

Obs	booktitle	author
1	**Operating Systems Title 301**	Torres, Emma
2	Internet Title 107	Marshall, Jose
3	Certification and Training Title 395	Martinez, Carol
4	Networks and Telecommunication Title 188	Ross, Shirley
5	Internet Title 247	Bryant, Jennifer

Obs	publisher	cost	listprice	saleprice
1	Wide-World Titles	$22.98	$45.95	**$45.95**
2	Eversons Books	$25.48	$50.95	$50.95
3	IT Training Texts	$18.48	$36.95	$36.95
4	AMZ Publishers	$25.48	$50.95	$50.95
5	Wide-World Titles	$25.48	$50.95	$50.95

Example 9.16: Using the INTO Clause in PROC SQL to Create a Macro Variable for Each Row in a Table

Numbered lists on the INTO clause can store rows of a table in macro variables. The PROC SQL step in Program 9.16 totals the sales for each of six sections in the bookstore, producing an extract of six rows. The SELECT statement and the INTO clause save the six section names and six formatted total sales values in twelve macro variables.

Program 9.16

```
proc sql noprint;
  select section, sum(saleprice) format=dollar10.2
  into :section1 - :section6,
       :sale1 - :sale6  from books.ytdsales
  group by section;
quit;
%put *** 1: &section1 &sale1;
%put *** 2: &section2 &sale2;
%put *** 3: &section3 &sale3;
%put *** 4: &section4 &sale4;
%put *** 5: &section5 &sale5;
%put *** 6: &section6 &sale6;
```

The SAS log showing the execution of the %PUT statements follows:

```
95   %put *** 1: &section1 &sale1;
*** 1: Certification and Training $31,648.52
96   %put *** 2: &section2 &sale2;
*** 2: Internet $62,295.78
97   %put *** 3: &section3 &sale3;
*** 3: Networks and Telecommunication $30,803.81
98   %put *** 4: &section4 &sale4;
*** 4: Operating Systems $39,779.11
99   %put *** 5: &section5 &sale5;
*** 5: Programming and Applications $62,029.41
100  %put *** 6: &section6 &sale6;
*** 6: Web Design $37,121.52
```

Example 9.17: Storing All Unique Values of a Table Column in One Macro Variable with PROC SQL

A feature of the INTO clause allows you to store all values of a column in one macro variable. These values are stored side by side. To do this, add the SEPARATED BY option to the INTO clause to define a character that delimits the string of values.

The PROC SQL SELECT statement in Program 9.17 stores all unique section names in the macro variable ALLSECT. (If you did not use the UNIQUE function, PROC SQL would attempt to concatenate the values of SECTION for all observations in the data set.)

Program 9.17

```
proc sql noprint;
  select unique(section)
  into :allsect separated by '/'
  from books.ytdsales
  order by section;
quit;
%put The value of macro variable ALLSECT is &allsect;
```

The SAS log showing the execution of the %PUT statement follows:

```
6681  %put The value of macro variable ALLSECT is &allsect;
The value of macro variable ALLSECT is Certification and
Training/Internet/Networks and Telecommunication/Operating
Systems/Programming and Applications/Web Design
```

Example 9.18: Storing All Values of a PROC SQL Dictionary Table Column in One Macro Variable

This example is similar to Example 9.17 in that it saves all values of a column in one macro variable, but this example does not apply the UNIQUE function. Program 9.18 makes use of the DICTIONARY tables feature of PROC SQL. It saves the names of all the SAS data sets in a library specified by the value of macro variable LISTLIB in one macro variable, DATASETNAMES. A blank separates the data set names assigned to DATASETNAMES. Assume there are three SAS data sets in library BOOKS: YTDSALES, SALES2006, and SALES2005.

Program 9.18

```
%let listlib=BOOKS;
proc sql noprint;
  select memname
  into :datasetnames separated by ' '
  from dictionary.tables
  where libname="&listlib";
quit;
%put The datasets in library &listlib is(are) &datasetnames;
```

The SAS log showing the execution of the %PUT statement follows:

```
6720  %put The datasets in library BOOKS is(are) &datasetnames;
The datasets in library BOOKS is(are)
SALES2005 SALES2006 YTDSALES
```

Using the Macro Variables Created by PROC SQL

PROC SQL creates and updates three macro variables after it executes each statement. You can use these macro variables in your programs to control execution of your SAS programs. These macro variables are stored in the global macro symbol table. Table 9.2 lists the three PROC SQL macro variables.

Table 9.2 Macro variables created by PROC SQL

Macro Variable	Description
SQLOBS	set to the number of rows produced with a SELECT statement
SQLRC	set to the return code from an SQL statement
SQLOOPS	set to the number of iterations of the inner loop of PROC SQL

The Pass-Through Facility of PROC SQL also creates two macro variables, SQLXRC and SQLXMSG. These macro variables contain information about error conditions that might have occurred in the processing of Pass-Through SQL statements. For complete information on these macro variables, refer to SAS/ACCESS documentation. Table 9.3 describes the two macro variables.

Table 9.3 PROC SQL macro variables used with the Pass-Through Facility

Macro Variable	Description
SQLXRC	set to the return code generated by a Pass-Through Facility statement
SQLXMSG	set to descriptive information about the error generated by a Pass-Through SQL statement.

Example 9.19: Using the PROC SQL SQLOBS Automatic Macro Variable

Macro program LISTSQLPUB in Program 9.19 uses the SQLOBS macro variable to define the number of macro variables needed in a SELECT statement. It then lists the value of each of the macro variables created in the PROC SQL step.

After the first PROC SQL SELECT statement executes, PROC SQL updates the SQLOBS macro variable. The second SELECT statement uses the value of SQLOBS set by the first SELECT statement to determine the total number of macro variables that the INTO clause should create.

Each publisher name is stored in its own macro variable. Macro program LISTSQLPUB sets up an iterative %DO loop with its upper index equal to the value assigned to SQLOBS. A %PUT statement executes with each iteration of the DO loop and displays the value of each variable.

The second SELECT statement in Program 9.19 produces the same value for SQLOBS. This is the value used as the upper index of the iterative %DO loop. Depending on the complexity of your programming, you might want to save the value of SQLOBS in another macro variable after that PROC SQL step ends. This would prevent loss of the SQLOBS value you need in case you submit other SELECT statements before executing code that references that SQLOBS value.

The SYMBOLGEN option is in effect during execution of the PROC SQL step. This option displays in the SAS log the resolved value of SQLOBS as the value is resolved in the second SELECT statement.

Program 9.19

```
options mprint;

%macro listsqlpub;
  options symbolgen;
  proc sql;
    select unique(publisher)
      from books.ytdsales
      order by publisher;

    select unique(publisher)
      into :pub1 - :pub&sqlobs
    from books.ytdsales
    order by publisher;
  quit;
  options nosymbolgen;

  %put Total number of publishers: &sqlobs..;
  %do i=1 %to &sqlobs;
    %put Publisher &i: &&pub&i;
  %end;
%mend listsqlpub;

%listsqlpub
```

The SAS log after LISTSQLPUB executes follows:

```
MPRINT(LISTSQLPUB):   proc sql;
MPRINT(LISTSQLPUB):   select unique(publisher) from
books.ytdsales order by publisher;
SYMBOLGEN:  Macro variable SQLOBS resolves to 12
MPRINT(LISTSQLPUB):   select unique(publisher) into :pub1 -
:pub12 from books.ytdsales order by publisher;
MPRINT(LISTSQLPUB):   quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.04 seconds
      cpu time            0.04 seconds

MPRINT(LISTSQLPUB):   options nosymbolgen;
Total number of publishers: 12.
Publisher 1: AMZ Publishers
Publisher 2: Bookstore Brand Titles
WARNING: Apparent symbolic reference LEE not resolved.
Publisher 3: Doe&Lee Ltd.
Publisher 4: Eversons Books
Publisher 5: IT Training Texts
Publisher 6: Mainst Media
Publisher 7: Nifty New Books
Publisher 8: Northern Associates Titles
Publisher 9: Popular Names Publishers
Publisher 10: Professional House Titles
Publisher 11: Technology Smith
Publisher 12: Wide-World Titles
```

Displaying Macro Option Settings with PROC SQL and Dictionary Tables

Using PROC SQL, you can obtain information about your SAS session by accessing dictionary tables. These read-only SAS data views contain such information as option settings, librefs, member names and attributes in a library, and column names and attributes in a table or data set. An earlier example in this section used a dictionary table to capture the names of all the data sets in a specific library and saved that information in one macro variable.

One dictionary table, OPTIONS, provides information about the current settings of SAS system options including macro facility related options. Another dictionary table, MACROS, provides information about macro variables including their scope and values. With these dictionary tables, you can access information about your current SAS session and programmatically use that in controlling execution of your SAS programs.

Example 9.20: Accessing Macro Option Settings with PROC SQL and Dictionary Tables

The OPTIONS dictionary table contains current settings and descriptions for SAS system options. A column in the table, GROUP, assigns a category to the setting. One group is MACRO. To display the current settings of the options in the MACRO group, submit the following code.

Program 9.20a

```
proc sql;
  select * from dictionary.options
  where group='MACRO';
quit;
```

The PROC SQL step in Program 9.20b saves the setting of the macro option MINDELIMITER in a macro variable MYSETTING.

Program 9.20b

```
options mindelimiter='#';
proc sql noprint;
  select setting
  into :mysetting
  from dictionary.options
  where optname='MINDELIMITER';
quit;
%put My current MINDELIMITER setting is &mysetting;
```

The SAS log after submitting the preceding program follows.

```
62   %put My current MINDELIMITER setting is &mysetting;
My current MINDELIMITER setting is #
```

Program 9.20b illustrates a simple example in working with the MACRO group of the dictionary tables. It might, however, be much easier to simply submit the following %LET statement to define macro variable MYSETTING.

```
%let mysetting=%sysfunc(getoption(mindelimiter));
```

Example 9.21: Accessing Macro Variable Characteristics with PROC SQL and Dictionary Tables

The dictionary table MACROS contains information about macro variables. Included in this table are the macro variables that you create as well as automatic variables created by SAS. To display the list of macro variables currently defined in your session, submit the following PROC SQL step.

Program 9.21

```
proc sql;
  select * from dictionary.macros;
quit;
```

As seen earlier in this book, submitting the following statement would display much of the same information.

```
%put _all_;
```

Using Macro Facility Features in SAS Component Language

Macro facility features can be incorporated in your SAS Component Language (SCL) programs. SCL programs are compiled and executed the same way as SAS language programs. The word scanner tokenizes the SCL statements and passes the tokens on to the SCL compiler for compilation. Macro variable references in SCL programs outside of SUBMIT blocks are resolved during tokenization and compilation. SCL programs execute when they are called.

Macro variable references and macro programs specified in SCL programs are processed in much the same way as they are in SAS language programs. However, many SCL features can accomplish the same tasks as the macro facility. Using SCL features instead of macro facility features may be preferable in order to make your SCL programs easier to follow and maintain.

This section briefly describes how you can use macro variables in your SCL SUBMIT blocks and how these macro variables relate to SCL variables.

Using the Macro Facility to Pass Information between SCL Programs

The macro facility can pass information between SCL programs. A global macro variable created in one SCL program can be referenced in another SCL program. The SYMGET function and the SYMPUT and SYMPUTX routines can pass macro variable values between SCL programs when the SCL programs execute. Otherwise, any other references to macro variables resolve at compilation of the SCL program.

The SYMGET function and the SYMPUT and SYMPUTX routines operate the same in SCL as they do in the SAS language DATA step. In SCL, these tools update and retrieve information from the global macro symbol table during execution of the SCL program. Refer to the previous sections for more information on the use of these tools.

In addition, SCL has the SYMPUTN routine and SYMGETN functions. The SYMPUTN routine assigns a numeric value to a global macro variable while the SYMGETN function returns the value of a global macro variable as a numeric value.

Even though the macro facility can pass information between SCL programs, it might be easier to follow and maintain your programs if you use the SCL CALL statement with parameters and the associated ENTRY statement in the called program.

Example 9.22: Creating a Macro Variable in an SCL Program

The following example shows part of an SCL program that processes the initials that the user enters and then creates a macro variable containing those initials.

Program 9.22a

```
array okinits{*} $ 3 ('MJM' 'JMB' 'BLT');

init:
  control label;
return;

term:
return;
```

```
inits:
  erroroff inits;
  if inits not in okinits then do;
    erroron inits;
    _msg_='NOTE: Please enter valid initials.';
  end;
  else do;
    call symput('USERINIT',inits);
  end;
return;
```

The program excerpt in Program 9.22b executes after the previous program. This program obtains the user's initials at the time of execution by using the SYMGET function.

Program 9.22b

```
length inits $ 3;

init:
  inits=symget('USERINIT');
return;

main:
.
.
.
```

Referencing Macro Variables in SUBMIT Blocks

Macro variable references in SCL programs are resolved at the time of tokenization and compilation. The exception to this is when SAS programs in SUBMIT blocks contain macro variable references.

Macro variable references in SUBMIT blocks do not resolve until the SAS program in the SUBMIT block is tokenized. During tokenization, SAS first checks to see if the macro variable reference corresponds to an SCL variable in the SCL program. If it does, the value of the SCL variable is substituted for the reference in the SAS program. If it does not, the macro processor then takes over and looks for the macro variable in the global symbol table.

To force the resolution of a macro variable reference by the macro processor and skip resolution by the SCL program, precede the macro variable reference with two ampersands.

Example 9.23: Using SUBMIT Blocks That Contain Macro Variables

Program 9.23 includes two SUBMIT blocks for two SAS programs; each program references a macro variable. Assume that there is a field on the SCL program screen for the user to enter a report date. When the user leaves this field blank and selects to run a report, the program in the first SUBMIT block executes. When the user specifies a report date, the program in the second SUBMIT block executes.

Both SUBMIT blocks reference a macro variable with the same name as the SCL variable. In the first SUBMIT block, two ampersands precede the macro variable name. Only the macro processor attempts to resolve the reference. In the second SUBMIT block, one ampersand precedes the macro variable name. Therefore, the SCL program is first to attempt resolution of the reference.

Program 9.23

```
init:
  control label;
  repdate=_blank_;
return;

term:
return;

runrep:
  if repdate=_blank_ then link reptoday;
  else link specrep;
  repdate=_blank_;
return;

reptoday:
  _msg_='NOTE: Today''s report is processing....';
  submit continue;
    %let repdate=&sysdate;
     proc print data=books.ytdsales;
       where saledate="&&repdate"D;
       title "Report for &&repdate";
       var section title saleprice;
     run;
  endsubmit;
return;
```

```
specrep:
  _msg_='NOTE: Past date report is processing....';
  submit continue;
    proc means data=books.ytdsales n sum;
      title "Sales Report for past date: &repdate";
      where saledate="&repdate"D;
      class section;
      var saleprice;
    run;
  endsubmit;
 return;
```

Part 2

Applying Your Knowledge of Macro Programming

Chapter 10 **Storing and Reusing Macro Programs 269**

Chapter 11 **Building a Library of Utilities 285**

Chapter 12 **Debugging Macro Programming and Adding Error Checking to Macro Programs 297**

Chapter 13 **A Stepwise Method for Writing Macro Programs 335**

Chapter 10

Storing and Reusing Macro Programs

Introduction 270

Saving Macro Programs with the Autocall Facility 270

- Creating an Autocall Library 271
- Making Autocall Libraries Available to Your Programs 273
- Maintaining Access to the Autocall Macro Programs That Ship with SAS 273
- Using the Autocall Facility under Windows, MVS/TSO, and Other Directory-Based Systems 275

Saving Macro Programs with the Stored Compiled Macro Facility 278

- Setting SAS Options to Create Stored Compiled Macro Programs 279
- Creating Stored Compiled Macro Programs 280
- Saving and Retrieving the Source Code of a Stored Compiled Macro Program 281
- Encrypting a Stored Compiled Macro Program 282

Resolving Macro Program References When Using the Autocall Facility and the Stored Compiled Macro Facility 283

Introduction

As your macro programming skills develop, you will find uses for your macro programs in several different applications. You might want to share these macro programs with your coworkers and make these macro programs available to your batch jobs. You might want to develop your own set of utilities. Since reusability is one of the great features of macro programs, it makes sense that there would be a systematic way to store macro programs in SAS. In fact, there are two ways to store your macro programs in SAS: the autocall facility and the stored compiled macro facility. This chapter describes how to use these two tools.

The ***autocall*** facility consists of external files or SOURCE entries in SAS catalogs that contain your macro programs. When you specify certain SAS options, the macro processor searches your autocall libraries when it is resolving a macro program reference.

The ***stored compiled*** macro facility consists of SAS catalogs that contain compiled macro programs. When you specify certain SAS options, the macro processor searches your catalogs of compiled macro programs when it is resolving a macro program reference.

Saving Macro Programs with the Autocall Facility

When you store a macro program in an autocall library, you do not have to submit the macro program for compilation before you reference the macro program. The macro processor does that for you if it finds the macro program in the autocall library.

Several SAS products ship with libraries of macro programs that you can reference, or that are referenced by the SAS products themselves.

The main disadvantage to the autocall facility is that the macro program must be compiled the first time it is used in a SAS session. This takes resources. Also, resources are used to search the autocall libraries for the macro program reference.

After the macro processor finds your macro program in your autocall library, it submits the macro program for compilation. If there are any macro language statements in open code, these statements are executed immediately. The macro program is compiled and stored in the session compiled macro program catalog, SASMACR, just as if you submitted it yourself. SASMACR is in the WORK directory.

The macro program can be reused within your SAS session. When it is, only the macro program itself is executed. Any macro language statements in open code that might have been stored with the macro program are not executed again. The compiled macro program is deleted at the end of the session when the catalog WORK.SASMACR is deleted. The code remains in the autocall library.

Creating an Autocall Library

The macro programs that you select for your autocall library can be stored as external files or as SOURCE entries in SAS catalogs.

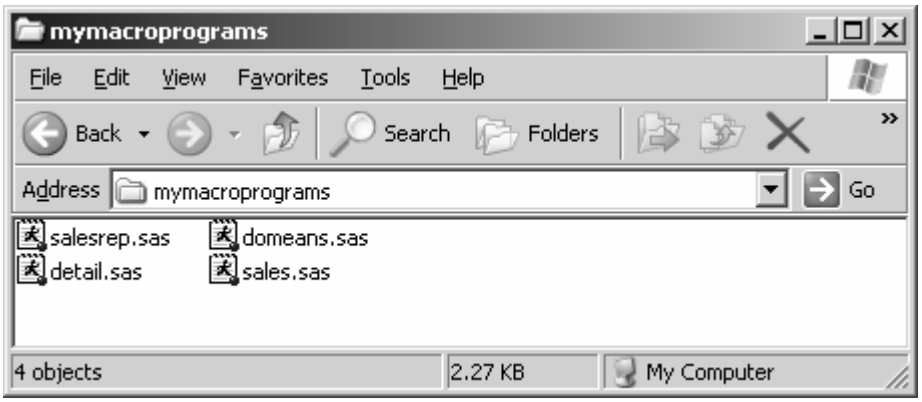
To store macro programs as external files in a directory-based system such as Windows, UNIX, and OpenVMS, you define the directory and add the macro programs to the directory. Each macro program is stored in an individual file with a file type or extension of SAS. The name given to the file must be the same as the macro program name. (Note that on UNIX platforms, the filename and macro program name must be in lowercase.)

Under MVS and TSO, macro programs that are stored as external files are saved as members of a partitioned data set. The name of the member should be the same as the name of the macro program.

When storing macro programs in a SAS catalog, make each macro program a separate SOURCE entry. The name of the SOURCE entry should be the same as the macro program name.

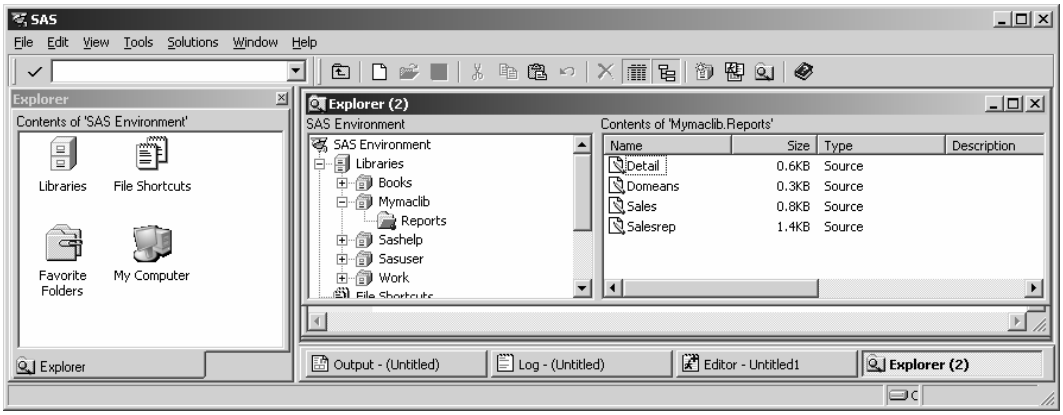
Display 10.1 shows an example of an autocall library where the four macro programs are stored as separate files.

Display 10.1 A Windows XP directory containing four autocall macro programs



Display 10.2 shows a SAS catalog that contains the four macro programs stored as SOURCE entries.

Display 10.2 A SAS catalog containing four autocall macro programs stored as SOURCE entries



Making Autocall Libraries Available to Your Programs

When you want SAS to search for your macro programs in autocall libraries, you must specify the two SAS options, MAUTOSOURCE and SASAUTOS. These options can be specified three ways:

- Add MAUTOSOURCE and SASAUTOS to the SAS command that starts the SAS session.
- Submit an OPTIONS statement with MAUTOSOURCE and SASAUTOS from within a SAS program.
- Submit an OPTIONS statement with MAUTOSOURCE and SASAUTOS from within an interactive SAS session.

The MAUTOSOURCE option must be enabled to tell the macro processor to search autocall libraries when resolving macro program references. By default, this option is enabled. Specify NOMAUTOSOURCE to turn off this option. A reason someone might disable MAUTOSOURCE is to save computing resources when not using autocall libraries.

```
options mautosource;
options nomautosource;
```

The SASAUTOS= option identifies the location of the autocall libraries for the macro processor. On the SASAUTOS= option, specify either the actual directory reference enclosed in single quotation marks or the filerefs that point to the directories. A FILENAME statement defines the fileref.

The syntax of the SASAUTOS= option follows. The first line shows how to specify one library. The second line shows how to specify multiple libraries. The macro processor searches the libraries in the order in which they are listed on the SASAUTOS= option.

```
options sasautos=library;
options sasautos=(library-1, library-2,..., library-n);
```

Maintaining Access to the Autocall Macro Programs That Ship with SAS

Autocall libraries in macro programs come with many SAS products. Chapter 6 describes many of these macro programs that ship with Base SAS. Your SAS session automatically assigns a fileref of SASAUTOS to the macro programs described in Chapter 6. Some applications of these macro programs include changing the case of a macro variable

value to lowercase (%LOWCASE) and trimming trailing blanks from a macro variable value (%TRIM).

With MAUTOSOURCE in effect and a typical installation of SAS that assigns the SASAUTOS= option to SASAUTOS, your SAS session automatically has access to these autocall macro programs.

To maintain access to the SASAUTOS autocall library, remember to include the SASAUTOS fileref when specifying references to your own libraries with the SASAUTOS= option. If you omit the SASAUTOS fileref when you issue your SASAUTOS= option, and you have not previously accessed the macro program shipped with SAS that you want to use, you will not have access to that macro program. If you had previously accessed one of the macro programs in the SASAUTOS library before removing SASAUTOS from the SASAUTOS= option, you will still be able to reference that macro program. This is because, on the first reference to that autocall macro program, the macro processor compiles the macro program and makes it available for the duration of the SAS session.

Defining Filerefs under Windows XP and Using Them to Identify Autocall Libraries

The next statements define two filerefs under Windows XP with SAS®9 and assigns them to SASAUTOS=. The OPTIONS statement includes these two filerefs plus the SASAUTOS fileref.

```
filename reports 'c:\mymacroprograms\repmacs';
filename graphs 'c:\mymacroprograms\graphmacs';
options sasautos=(reports graphs sasautos);
```

Explicitly Specifying the Directory Locations of Autocall Libraries on the OPTIONS Statement

To specify the same libraries as above without using filerefs, submit the following statement. Note the inclusion of the SASAUTOS fileref.

```
options sasautos=
    ('c:\mymacroprograms\repmacs' 'c:\mymacroprograms\graphmacs'
        sasautos);
```

Identifying Autocall Libraries That Are Stored in SAS Catalogs

An autocall library stored in a SAS catalog requires that you specify the CATALOG access method on the FILENAME statement that identifies the autocall library. The syntax of the FILENAME statement is

```
filename fileref catalog 'library.catalog';
```

The next statements reference a user-defined autocall library stored in a SAS catalog under Windows XP SAS®9. It also includes the SASAUTOS fileref.

```
filename mymacs catalog 'books.repmacs';
options sasautos=(mymacs sasautos);
```

Listing the Names of the Autocall Libraries That Are Defined in the SAS Session

If you want to check what autocall libaries are defined in the current SAS session, submit the following PROC step.

```
proc options option=sasautos;
run;
```

Using the Autocall Facility under Windows, MVS/TSO, and Other Directory-Based Systems

Under a directory-based system, all macro programs are stored as individual files in a directory. Each macro program should have a file extension of `.sas` and a filename identical to the macro program name. Preceding examples use the autocall facility under Windows XP.

Using the Autocall Facility under MVS Batch

Under the MVS operating system, autocall libraries are stored in partitioned data sets. Each macro program is a member in the partitioned data set. The name of the member is the same as the name of the macro program. A JCL DD statement assigns autocall libraries. The following example shows the beginning of the JCL for a batch job that specifies one autocall library. Note that the MAUTOSOURCE option is enabled.

```
//MYJOB    JOB  account....
//         EXEC SAS,OPTIONS='MAUTOSOURCE'
//SASAUTOS DD   DSN=MYAPPS.REPMACS,DISP=SHR
```

The next example shows how multiple macro libraries can be specified.

```
//MYJOB    JOB  account....
//         EXEC SAS,OPTIONS='MAUTOSOURCE'
//SASAUTOS DD   DSN=MYAPPS.REPMACS,DISP=SHR
//         DD   DSN= MYAPPS.GRPHMACS,DISP=SHR
```

An OPTIONS statement can also be submitted from within the SAS program to specify the use of autocall libraries. The following statement specifies one user-defined autocall library plus the SASAUTOS fileref.

```
options mautosource sasautos=('myapps.repmacs' sasautos);
```

The following statement specifies two user-defined autocall libraries plus the SASAUTOS fileref.

```
options mautosource sasautos= ('myapps.repmacs' 'myapps.grphmacs'
                   sasautos);
```

Using the Autocall Facility under TSO

As with MVS batch jobs, autocall libraries under TSO are stored in partitioned data sets; each macro program is a member.

The following example starts an interactive TSO session that assigns one user-defined autocall library and includes the SASAUTOS reference.

```
sas options('mautosource sasautos=("myapps.repmacs" sasautos)')
```

The next example starts an interactive TSO session that assigns two user-defined autocall libraries and includes the SASAUTOS reference.

```
sas options('mautosource sasautos=("myapps.repmacs"
            "myapps.grphmacs" sasautos)')
```

Autocall libraries can also be specified from within the SAS session by using the OPTIONS statement. The OPTIONS statement is written as shown in the preceding MVS Batch section.

Using the Autocall Facility under UNIX

As with other directory-based systems, autocall libraries under UNIX are made up of separate files, each with `.sas` as the extension. Each macro program is in a separate file. The name of the file is the same as the name of the macro program. Note that the filename and macro program name must be in lowercase on UNIX platforms.

The following example specifies one user-defined autocall library and includes the SASAUTOS reference.

```
sas -mautosource -sasautos '/mymacroprograms/repmacs'
    -append sasautos sasautos
```

The next example specifies two user-defined autocall libraries and includes the SASAUTOS reference.

```
sas -mautosource -sasautos '/mymacroprograms/graphmacs'
    -append sasautos '/mymacroprograms/repmacs'
    -append sasautos sasautos
```

From within a UNIX SAS session, the following line specifies one user-defined autocall library and includes the SASAUTOS reference.

```
options mautosource sasautos=('/mymacroprograms/repmacs',
                               sasautos)
```

The next OPTIONS statement specifies two user-defined autocall libraries from within a UNIX SAS session and includes the SASAUTOS reference.

```
options mautosource sasautos=
('/mymacroprograms/repmacs','/mymacroprograms/graphmacs',
    sasautos);
```

Using the Autocall Facility under OpenVMS

OpenVMS is a directory-based operating system. Macro programs in an autocall library are stored in a directory as separate files. The name of the file is the same as the name of the macro program. The extension of the macro program files should be `.sas`.

The following SAS command specifies one user-defined autocall library and includes the SASAUTOS reference.

```
sas /mautosource/sasautos=('[myapps.programs.repmacs]',
     sasautos)
```

The next SAS command specifies two user-defined autocall libraries and includes the SASAUTOS reference.

```
sas /mautosource/sasautos=
('[myapps.programs.repmacs]', '[myapps.programs.grphmacs]',
   sasautos)
```

From within a SAS program or SAS interactive session, the following OPTIONS statement specifies one user-defined autocall library and includes the SASAUTOS reference.

```
options mautosource sasautos=('[myapps.programs.repmacs]',
                               sasautos);
```

The next OPTIONS statement specifies two user-defined autocall libraries and includes the SASAUTOS reference.

```
options mautosource sasautos=
('[myapps.programs.repmacs]', '[myapps.programs.grphmacs]',
  sasautos);
```

Saving Macro Programs with the Stored Compiled Macro Facility

Macro programs that you want to save and do not expect to modify can be compiled and saved in SAS catalogs using the stored compiled macro facility. When a compiled macro program is referenced in a SAS program, the macro processor skips the compiling step, retrieves the compiled macro program, and executes the compiled code. The main advantage of this facility is that it prevents repeated compiling of macro programs that you use frequently.

A disadvantage of this facility is that the compiled versions of macro programs cannot be moved to other operating systems. The macro source code must be saved and recompiled under the new operating system. Further, if you are moving the compiled macro programs to a different release of SAS under the same operating system, you might also have to recompile the macro programs.

Macro source code is not stored by default with the compiled macro program. You are responsible for maintaining a copy of the macro source code. A convenient place to store the code is an autocall library. Also, you can save the source code as a SOURCE entry in a catalog if you specify the SOURCE option when compiling your macro program. Another way of saving the macro program code for later retrieval is shown in a later section where the SOURCE option is added to the %MACRO statement when creating a stored compiled macro program. This option stores the macro program code in the same entry as the compiled code, and you can retrieve this code later with the %COPY statement.

Setting SAS Options to Create Stored Compiled Macro Programs

You need to set two SAS options, MSTORED and SASMSTORE, before you can compile and store your macro programs.

The MSTORED option instructs SAS that you want to make stored compiled macro programs available to your SAS session.

```
options mstored;
```

To turn off the MSTORED option, submit the following OPTIONS statement.

```
options nomstored;
```

The value that you assign to the SASMSTORE option is the libref that points to the location of the SAS catalog containing the compiled macro programs. Here is an example of SASMSTORE under Windows XP in SAS®9:

```
libname myapps 'c:\mymacroprograms';
options mstored sasmstore=myapps;
```

SAS stores compiled macro programs in a catalog called SASMACR. The SASMACR catalog is stored in the directory specified by the SASMSTORE option. In this example, that directory has the libref of MYAPPS. Do not rename the SASMACR catalog. Use the CATALOG command or PROC CATALOG to view the list of macro programs stored in this catalog.

You can also tell the macro processor to search SASMACR catalogs in multiple locations for a stored compiled macro program by listing the multiple paths on the LIBNAME statement. The following code tells the macro processor to look in the SASMACR catalog in the three locations that are specified within the parentheses.

The order in which you list the paths is the order in which SAS searches for a stored compiled macro program. If you have a macro program with the same name in two locations, the program found in the first of the two paths is the one that executes.

```
libname myapps ('c:\mymacroprograms',
                'z:\mymacroprograms',
                'c:\legacy\macros');
options mstored sasmstore=myapps;
```

Creating Stored Compiled Macro Programs

Once the SAS options in the previous section are set, macro programs can be compiled and stored in a catalog by adding options to the %MACRO statement. The syntax of %MACRO when you want to compile and store a macro program follows:

```
%macro macro-name(parameters) / store <source secure
                                des="description">;
  macro-program-code
%mend macro-name;
```

The STORE keyword is required. The SOURCE, SECURE, and DES= options are not required.

The SOURCE option tells the macro processor to save a copy of the macro program's source code, along with the compiled macro program in the same SASMACR catalog. It does not have a separate entry in the catalog and is instead stored in the same MACRO entry as the compiled macro program.

Starting with SAS 9.2, you can use the SECURE option to encrypt the compiled macro program and prevent someone from easily obtaining the source code. Without the SECURE option, it is not easy, but it is possible to extract the code.

Use the DES= option to save up to 40 characters of text to describe your macro program. SAS displays the descriptive text when you view the contents of the catalog that holds the compiled stored macro programs.

Example 10.1: Creating a Stored Compiled Macro Program

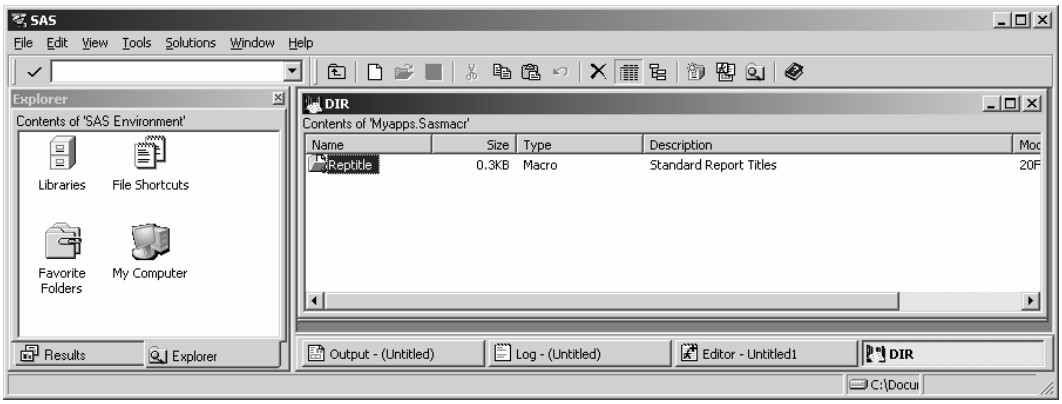
An example of defining a macro program and storing it in a catalog under Windows XP in SAS®9 follows:

Program 10.1

```
libname myapps 'c:\mymacroprograms';
options mstored sasmstore=myapps;

%macro reptitle(repprog) / store des='Standard Report Titles';
  title "Bookstore Report &repprog";
  title2 "Processing Date: &sysdate  SAS Version: &sysver";
%mend reptitle;
```

Display 10.3 shows the DIR window for the MYAPPS.SASMACR catalog after submitting this program.

Display 10.3 SAS DIR window for a catalog with one stored compiled macro program



Saving and Retrieving the Source Code of a Stored Compiled Macro Program

As mentioned earlier, the SOURCE option on the %MACRO statement in conjunction with the STORE option saves a copy of the source code of the compiled macro program. It is not saved as a separate entry that you can retrieve; it is embedded in the same entry as the compiled code. To retrieve a copy of the code, use the %COPY macro language statement. This statement can list the code in the SAS log or save the code to a file. The syntax of the %COPY statement follows. The three options are optional.

```
%COPY macro-program-name / <library= outfile= <fileref>
                           <'external file'> source >;
```

By default, if you do not specify a libref with the LIBRARY= option, the macro processor will look in the library specified by the current setting of SASMSTORE.

Example 10.2: Saving the Source Code of a Stored Compiled Macro Program

Program 10.1 is modified in Program 10.2 to save the macro program code along with the compiled macro program.

Program 10.2

```
libname myapps 'c:\mymacroprograms';
options mstored sasmstore=myapps;

%macro reptitle(repprog) / store source
                           des='Standard Report Titles';
  title "Bookstore Report &repprog";
  title2 "Processing Date: &sysdate  SAS Version: &sysver";
%mend reptitle;
```

If you want to view the code in the SAS log, submit the following statement:

```
%copy reptitle / library=myapps source;
```

The SAS log shows the results of submitting the %COPY statement.

```
9    %copy reptitle / library=myapps source;
%macro reptitle(repprog) / store source des='Standard Report
Titles';
  title "Bookstore Report &repprog";
  title2 "Processing Date: &sysdate  SAS Version: &sysver";
%mend reptitle;
```

If you want to save the code in a file called REPTITLE_SOURCE.SAS, submit the following %COPY statement.

```
%copy reptitle / library=myapps source
             outfile='c:\mymacroprograms\reptitle_source.sas';
```

Encrypting a Stored Compiled Macro Program

Starting with SAS 9.2, as mentioned earlier, the SECURE option on the %MACRO statement in conjunction with the STORE option encrypts the compiled macro program. The SECURE and SOURCE options are incompatible on the same %MACRO statement. Therefore, you must save a copy of your macro program code separately from the stored compiled macro program before you store and compile a macro program with the SECURE option.

Example 10.3: Encrypting a Stored Compiled Macro Program

Program 10.2 is modified in Program 10.3 so that the stored compiled macro program is secure and encrypted. Note that this program executes only in SAS 9.2 or later.

Program 10.3

```
libname myapps 'c:\mymacroprograms';
options mstored sasmstore=myapps;

%macro reptitle(repprog) / store secure
                           des='Standard Report Titles';
  title "Bookstore Report &repprog";
  title2 "Processing Date: &sysdate  SAS Version: &sysver";
%mend reptitle;
```

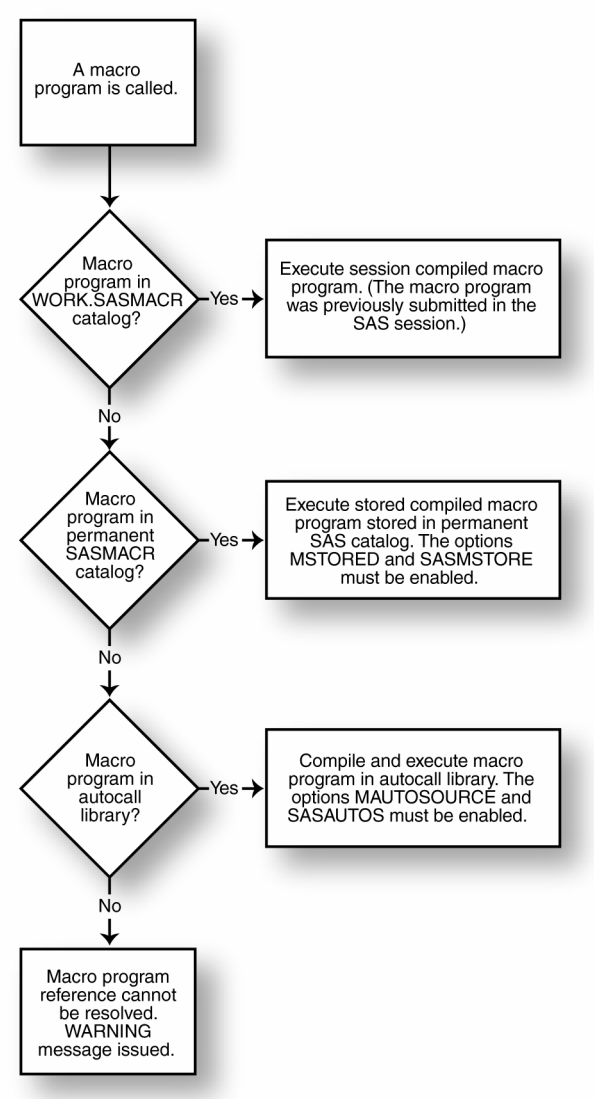
Resolving Macro Program References When Using the Autocall Facility and the Stored Compiled Macro Facility

The autocall facility and the stored compiled macro facility increase the scope of the tasks that the macro processor can do for you. Now, instead of explicitly submitting a macro program, you tell the macro processor where and how the macro program is stored. The macro processor understands that it should check these sources after looking within the SAS session for macro programs that were compiled during the session.

If the macro processor finds the macro program in your autocall library, it submits the macro program for compilation. When the macro processor finds the macro program in your SASMACR catalog, it submits for execution the compiled code that is stored in the catalog.

When you make autocall libraries and stored compiled macro programs available to your SAS session by enabling the options described above, the macro processor takes the steps in Figure 10.1 to resolve a macro program reference.

Figure 10.1 How the macro processor resolves calls to macro programs



Chapter 11

Building a Library of Utilities

Introduction 285
Writing a Macro Program to Behave Like a Function 286
Programming Routine Tasks 290

Introduction

Chapter 10 showed ways of saving your macro program code and compiled macro programs. With these tools, you can create and organize your own libraries of macro programs that you and your coworkers frequently use. This chapter introduces the concept of building your own library of macro programs that perform routine or frequent tasks.

Because your work requirements are very different from that of another reader, your library of utility macro programs is likely to be different from his or hers. As you read through the chapter, keep in mind the routine and commonly performed programming tasks that you could include in your own library of utility routines. The SAS website,

SAS conference proceedings, and SAS Press books are useful sources of code that you can adapt and add to your own libraries of tools.

Writing a Macro Program to Behave Like a Function

You can write a macro program to return a value as though it was a macro function. These values can be used to test conditions in your macro program code. These types of macro programs may be useful when you need to customize actions similar to those produced by existing macro functions that you frequently use, as shown in Examples 11.1 and 11.2.

Example 11.1: Examining Specific Data Set Characteristics

The SAS language function EXIST determines if a data set exists and sets a return code based on its determination. It returns 1 if the data set exists and 0 if the data set does not exist. The following code shows how you could test for the existence of a data set with this function. If the data set exists, PROC PRINT lists the first ten observations of the data set specified in the parameter DSNAME. Otherwise, the macro program LISTSAMPLE writes a message to the SAS log that the data set does not exist.

Program 11.1a

```
%macro listsample(dsname);
  %if %sysfunc(exist(&dsname)) %then %do;
    proc print data=&dsname(obs=10);
     title "First 10 Observations of &dsname";
    run;
  %end;
  %else  %put ERROR: ***** Data set &dsname does not exist.;
%mend listsample;

%listsample(books.ytdsales)
```

Program 11.1a successfully checks for the existence of a data set.

If your needs require that multiple features of a data set hold in addition to existence of the data set, you could extend the testing that LISTSAMPLE does. These tests could be put in another macro program that you could reference similarly to the way you reference the EXIST function. This macro program could return a value of 0 or 1 just as SAS language function EXIST does.

Macro program MULTCOND in Program 11.1b checks four conditions of a data set, including whether it exists. The macro program defines a macro variable RC whose values can be 0 or 1 depending on the macro program's evaluations of the data set. When the returned value is 0, the data set should not be processed. When the returned value is 1, the data set can be processed.

Macro program MULTCOND returns a value by applying the PUTN SAS language function to macro variable RC. The %SYSFUNC macro function is required to execute the PUTN function.

Note that all the macro variables used in macro program MULTCOND are defined as local macro variables on the %LOCAL statement. Since this macro program could be accessed in different situations, declaring these macro variables as local prevents conflicts in macro variable resolution if these macro variables had previously been defined in open code or by a macro program that called MULTCOND.

The four conditions that macro program MULTCOND examines are:

1. Data set existence with the EXIST SAS language function
2. Data set can be opened for input with the OPEN SAS language function
3. Data set has at least one undeleted observation as determined by the ATTRN SAS language function and NLOBS argument
4. Data set has a read access password as determined by the ATTRN SAS language function and READPW argument.

The macro program branches to label SETRC when a test fails. The %LET statement that follows this label sets the value of RC to zero. If a data set passes all tests, the value of RC is 1.

Note that the only modification to LISTSAMPLE in Program 11.1a was to replace `%sysfunc(exist)` with `%multcond(&dsname)`.

Program 11.1b

```
%macro multcond(dsname);
  %local rc dsid exist nlobs readpw;

  %*----Initialize return code to 1;
  %let rc=1;
  %*----Initialize data set id;
  %let dsid=0;

  %*----Does data set exist (condition 1);
  %let exist=%sysfunc(exist(&dsname));
```

```
  %*----Data set does not exist;
  %if &exist=0 %then %goto setrc;

  %let dsid=%sysfunc(open(&dsname,i));
  %*----Data set cannot be opened (condition 2);
  %if &dsid le 0 %then %goto setrc;
  %*----Any obs to list from this data set? (condition 3);
  %let nlobs=%sysfunc(attrn(&dsid,nlobs));
  %*----No obs to list;
  %if &nlobs le 0 %then %goto setrc;

  %*----Read password set on this data set? (condition 4);
  %let readpw=%sysfunc(attrn(&dsid,readpw));
  %*----READPW in effect, do not list;
  %if &readpw=1 %then %goto setrc;

  %*----Data set okay to list, skip over section
        that sets RC to 0;
  %goto exit;

  %*----Problems with data set, set RC to 0;
  %setrc:
  %let rc=0;

  %exit:
  %if &dsid ne %then %let closerc=%sysfunc(close(&dsid));

  %*----Return the value of macro variable RC;
  %sysfunc(putn(&rc,1.))
%mend;

%macro listsample(dsname);
  %if %multcond(&dsname)=1 %then %do;
    proc print data=&dsname(obs=10);
      title "First 10 Observations of &dsname";
    run;
  %end;
  %else  %put ERROR: ***** Data set &dsname cannot be listed.;
%mend listsample;

*----First call to LISTSAMPLE;
%listsample(books.ytdsales)

*----Second call to LISTSAMPLE;
%listsample(books.ytdsaless)
```

In the first call to LISTSAMPLE, assume BOOKS.YTDSALES exists and passes the four tests in MULTCOND. Therefore, the PROC PRINT step lists the first ten observations.

In the second call to LISTSAMPLE, the data set name is misspelled in order to cause MULTCOND to assign a value of 0 to macro variable RC. The value that %MULTCOND returns is a 0 and the %ELSE statement in LISTSAMPLE executes. Macro program LISTSAMPLE writes to the SAS log in red the following message after submission of the second call.

```
ERROR: ***** Data set books.ytdsaless cannot be listed.
```

Example 11.2: Editing Character Data for Comparisons

Character data such as names, titles, and addresses can be stored various ways. When you want to select observations from a data set based on a character data value, you may have to edit the value so that you can find as many matching observations as possible in the data set. From your text value, you may need to remove extra blanks and punctuation and convert the value to a specific case. If you commonly edit a type of value the same way, you may want to create a utility macro program that does this task for you, which you can later reference when needed.

Program 11.2 submits a PROC REPORT step that lists all titles for an author. It defines a macro program, TRIMNAME, that edits the author's name to remove extra blanks and all punctuation, except for commas, and converts the author's name to uppercase. This edited value is then supplied to the WHERE statement of the PROC REPORT step and is inserted in the title.

Using multiple SAS language and macro functions, TRIMNAME edits the parameter value it receives. Note there are no macro or SAS language statements in TRIMNAME. All that TRIMNAME does is apply the series of functions to the parameter value that it receives.

Note the usage of the quoting functions %NRBQUOTE and %SUPERQ. In this example, the author's name can contain a comma and without using these functions, the comma is interpreted as a separator between arguments to functions %CMPRES and COMPRESS, respectively. Omitting the quoting functions generates errors.

Program 11.2

```
%macro trimname(namevalue);
  %cmpres(%nrbquote(%upcase(%sysfunc(
      compress(%superq(namevalue),%str(, ),kA)))))
%mend trimname;
```

```
proc report data=books.ytdsales
   (where=(upcase(author)=
         "%trimname(%str(wright,   james.))"))
       nowd;
  title "Title list for %trimname(%str(Wright, james))";
  column booktitle n;
  define booktitle / group width=30;
run;
```

Output 11.1 shows the results of the PROC REPORT step, including the editing of the author's name for insertion in the title.

Output 11.1 Output from Program 11.2 that applies a macro program that behaves like a function

```
                Title list for WRIGHT, JAMES

          Title of Book                          n
          Internet Title 711                    12
          Programming and Applications T         1
```

Programming Routine Tasks

In your SAS programming, you may need to program the same process in different applications. For example, perhaps your company requires that all reports have the same dimensions, a title written a certain way, and a footnote identifying program name and programmer. It is often useful to save these routine, frequently used tasks as macro programs in a library of utilities.

Example 11.3: Standardizing RTF Output

Program 11.3 defines two macro programs that manage production of reports sent to the ODS RTF destination and applies these to the production of a report by PROC REPORT. The tasks that these two macro programs accomplish are examples of the kinds of routine tasks you might want to consider adding to your library of utility routines.

The first macro program, RTF_START, initializes settings when sending a report to the ODS RTF destination. The second macro program, RTF_END, resets options and closes the RTF destination after the report or reports are produced. Macro program RTF_START does the following tasks:

- closes the LISTING destination
- changes the orientation to that specified by the value of the ORIENTATION parameter
- turns off the SAS option DATE
- specifies an ODS style to use in producing the report
- specifies TITLE1 and FOOTNOTE1 statements.

Macro program RTF_END does the following tasks:

- closes the RTF destination
- opens the LISTING destination
- resets the orientation to PORTRAIT
- turns on SAS option DATE
- clears the TITLE1 and FOOTNOTE1 statements.

In Program 11.3, the two macro programs are first submitted and compiled. Then macro program RTF_START executes, followed by a PROC REPORT step. Last, macro program RTF_END executes.

The call to RTF_START specifies the ODS style MONEY that is shipped with SAS software and found in SASUSER.TMPLMST.

Program 11.3

```
%macro rtf_start(style=,orientation=);
  %* This macro program initializes settings to send reports
     to ODS RTF destination;
  ods listing close;

  options orientation=&orientation nodate;
  ods rtf style=&style;

  title1 justify=center "Bookstore";
  footnote justify=right "Report Prepared &sysdate9";
%mend rtf_start;

%macro rtf_end;
 %* This macro program resets options and closes the RTF
    destination after sending a report to the ODS RTF
    destination;
```

```
  ods rtf close;
  ods listing;

  options orientation=portrait date;
  title;
  footnote1;
%mend rtf_end;

%rtf_start(style=money,orientation=landscape)

proc report data=books.ytdsales nowd;
  column section saleprice;
  define section / group;
  define saleprice / sum analysis format=dollar11.2;
  rbreak after / summarize;
  compute after;
    section='** Totals **';
  endcomp;
run;

%rtf_end
```

The output report is in color. A gray-scale copy of the report follows in Output 11.2.

Output 11.2 Report produced by Program 11.3

Bookstore

Section	Sale Price
Certification and Training	$31,648.52
Internet	$62,295.78
Networks and Telecommunication	$30,803.81
Operating Systems	$39,779.11
Programming and Applications	$62,029.41
Web Design	$37,121.52
*** Totals *****	*$263,678.15*

Report Prepared 25FEB2008

Example 11.4: Documenting Characteristics of a Data Set

As a programmer, you may frequently want to list specific information about a data set in a specific order. You could submit multiple steps and review the output to do this. Alternatively, you could write a macro program to accomplish all the steps and save the macro program in your library of utilities to be referenced as needed.

Macro program FACTS in Program 11.4 determines specific information about a SAS data set, lists this information, and lists the first five observations of the data set. It saves the output in a PDF file. The information presented is available in several procedures. The goal of this macro program is to list only specific pieces of information in a specific order to produce customized documentation of the data set.

The only parameter to FACTS is DSNAME, which is the name of the data set that FACTS should examine. Macro program FACTS saves the data set characteristics in a data set and displays the information with PROC PRINT. Output from this program is directed to the ODS PDF destination. Temporary data sets created by FACTS are deleted at the conclusion of the macro program.

The program is long, but it does just a few tasks. Macro program FACTS creates several macro variables with PROC SQL and accesses the data set descriptive information from dictionary tables.

The data set that contains the information for the report has two variables, ATTRIBUTE and VALUE. The information obtained by PROC SQL and saved in macro variables is assigned to these two variables. The macro variables created by PROC SQL are in bold and underlined in Program 11.4.

Macro program FACTS could be improved with error checking. For example, the first task that could be done is to determine if the data set exists. If not, a different report could be produced. Actions could also be specified based on the results obtained from the dictionary tables. Different ODS destinations and further enhancements of the report could be made.

Note that all the macro variables created in macro program FACTS are defined as local macro variables on the %LOCAL statement, as was done in the definition for macro program MULTCOND in Program 11.1b. This action prevents conflicts in macro variable resolution if these macro variables had previously been defined in open code or by a macro program that calls FACTS.

Program 11.4

```
%macro facts(dsname);
  %local dslib dsmem varpos varalpha dslabel crdate modate
         nobs nvar;
  %let dsname=%upcase(&dsname);

  %*----Extract each part of data set name;
  %let dslib=%scan(&dsname,1,.);
  %let dsmem=%scan(&dsname,2,.);

  proc sql noprint;
    create table npos as
      select npos,name
      from dictionary.columns
      where libname="&dslib" and memname="&dsmem"
       order by npos;
    select name into :varpos separated by ', ' from npos;

    select name
      into :varalpha separated by ', '
      from dictionary.columns
      where libname="&dslib" and memname="&dsmem"
      order by name;

    select memlabel,crdate,modate,nobs,nvar
      into :dslabel,:crdate,:modate,:nobs,:nvar
      from dictionary.tables
      where libname="&dslib" and memname="&dsmem";
  quit;

  data temp;
    length attribute $ 35
           value $ 500;

    *----Create an observation for each characteristic of the
         data set;
    attribute='Creation Date and Time';
    value="&crdate";
    output;
    attribute='Last Modification Date and Time';
    value="&modate";
    output;
    attribute='Number of Observations';
```

```
    value="&nobs";
    output;
    attribute='Number of Variables';
    value="&nvar";
    output;
    attribute='Variables by Position';
    value="&varpos";
    output;
    attribute='Variables Alphabetically';
    value="&varalpha";
    output;
  run;

  ods listing close;
  ods pdf;
  title "Data Set Report for &dsname %trim(&dslabel)";
  proc print data=temp noobs label;
    var attribute value;
    label attribute='Attribute'
          value='Value';
  run;
  proc print data=&dsname(obs=5);
    title2 "First 5 Observations";
  run;
  ods pdf close;
  ods listing;

  proc datasets library=work nolist;
    delete temp npos;
  run;
  quit;
%mend facts;

%facts(books.ytdsales)
```

Output 11.3 presents the results of applying macro program FACTS to data set BOOKS.YTDSALES as specified in the last statement in Program 11.4.

Output 11.3 Report produced by Program 11.4

Data Set Report for BOOKS.YTDSALES Sales for 2007

Attribute	Value
Creation Date and Time	07JAN07:15:52:25
Last Modification Date and Time	22FEB08:15:52:25
Number of Observations	6096
Number of Variables	10
Variables by Position	saleid, cost, listprice, saleprice, datesold, section, saleinit, booktitle, author, publisher
Variables Alphabetically	author, booktitle, cost, datesold, listprice, publisher, saleid, saleinit, saleprice, section

Data Set Report for BOOKS.YTDSALES Sales for 2007

First 5 Observations

Obs	section	saleid	saleinit	datesold	booktitle	author	publisher	cost	listprice	Sale price
1	Web Design	10000001	LPL	01/18/2007	Web Design Title 160	Allen, Michael	IT Training Texts	$18.48	$36.95	$33.26
2	Certification and Training	10000002	MJM	01/07/2007	Certification and Training Title 115	Martinez, Robert	Popular Names Publishers	$22.48	$44.95	$44.95
3	Web Design	10000003	JAJ	01/24/2007	Web Design Title 150	Flores, Barbara	Technology Smith	$17.48	$34.95	$34.95
4	Programming and Applications	10000004	CAD	01/20/2007	Programming and Applications Title 330	Williams, Emma	Doe&Lee Ltd.	$22.48	$44.95	$44.95
5	Internet	10000005	SMA	01/05/2007	Internet Title 745	Harris, Ashley	AMZ Publishers	$18.48	$36.95	$36.95

Chapter 12

Debugging Macro Programming and Adding Error Checking to Macro Programs

Introduction 298

Understanding the Types of Errors That Can Occur in Macro Programming 298

Minimizing Errors in Developing SAS Programs That Contain Macro Language 299

Categorizing and Checking for Common Problems in Macro Programming 299

Understanding the Tools That Can Debug Macro Programming 303

- Using SAS System Options to Debug Macro Programming 304
- Using Macro Language Statements to Debug Macro Programming 305
- Using Macro Functions to Debug Macro Programming 306
- Using Automatic Macro Variables to Debug Macro Programming 306

Examples of Solving Errors in Macro Programming 307

Improving Your Macro Programming by Including Error Checking 326

Introduction

Despite your best efforts, your coding is going to include errors from time to time. To prevent and correct errors in your programs, you need to rely on your knowledge of SAS and macro processing concepts and features to help you track down the source of the problems. This chapter shows you where some errors can occur in your macro programs, and it describes tools and techniques that you can employ to help you prevent and correct those errors.

Understanding the Types of Errors That Can Occur in Macro Programming

Including macro language in your SAS programs can increase the complexity of debugging your programs. Errors can originate in your SAS code, errors can originate in your macro language code, and errors can originate in the SAS code generated by the macro language. When debugging a program that did not execute as you expected, you will need to determine the origin of the error.

Errors can also occur at the different stages of processing a program. A misspelled macro function name is detected *during compilation* when the macro processor cannot resolve the reference. An error that occurs *during execution* may not be detected by SAS or by the macro processor. All of the statements may be specified correctly, but the outcome is not what you intended. Most likely execution-time problems arise from logic errors in your SAS or macro language programming statements.

A *syntax error*, which is detected during compilation, occurs when macro language code does not follow macro language rules. Syntax errors are usually easy to fix. When your program contains a syntax error, such as a misspelled keyword, it does not execute and messages related to the syntax error are written to the SAS log.

An *execution* error, which usually results from problems in logic, may or may not be easy to fix. This is where your knowledge of the concepts of macro processing and SAS processing becomes more important. Messages related to the problem may or may not be written to the SAS log. You may need to employ some of the techniques described in this chapter and earlier in the book to find the source of your execution error.

Minimizing Errors in Developing SAS Programs That Contain Macro Language

As you begin to incorporate macro language in your SAS programs, you will probably find it most efficient to add this code in steps of increasing complexity and test each step as you develop your application. Attempting to write the complete application without testing its components could cause considerable difficulty in debugging your application.

A typical way to develop a macro application is to make sure that the SAS code, without any macro language features, does what you expect and after completing that task, then you add macro facility features. Chapter 13 breaks down this process into four steps and applies the process to an example. A list of the four steps follows.

1. Write your SAS code without any macro features.
2. Assign any hard-coded programming constants in your SAS code to macro variables defined in open code. Such constants could include data set name, time period in which to analyze the data, and title text.
3. Write a macro program and convert the open code macro variables defined in Step 2 to parameters to your macro program.
4. Add error checking to your macro program and generalize the processing in your macro program so that it can accommodate a wider range of processing tasks.

As you build and test your macro program in steps, you can use several macro facility features such as system options and macro language statements to verify that your macro program executes correctly. These are the same tools that can help you debug your macro programs, and these tools are described and demonstrated later in this chapter. Examples in previous chapters also apply these tools.

Categorizing and Checking for Common Problems in Macro Programming

Table 12.1 categorizes areas in which common problems can occur in SAS programs that contain macro language. Each category has a list of items that you can check as you develop your macro applications.

Table 12.1 Categories of common problems in macro programming and items to check in the code

Category	Items to Check
Punctuation	Do all statements end with a semicolon? Did you terminate your macro program call with a semicolon? If so, does the semicolon interfere with the resolution of the macro program call? Do you have any unmatched parentheses or quotation marks? If so, is it necessary to mask them? Are you using single quotation marks and double quotation marks correctly? For example, is title text enclosed with double quotation marks so that macro variable references in the text get resolved? Have you terminated your macro statement labels with a colon(:)? Does the statement label referenced on your %GOTO statement start with a percent sign? If so, you may need to remove it if you are explicitly referencing a statement label to prevent the label from being interpreted as a macro program reference.
Macro Variable Resolution	Do your macro variable references start with an ampersand? Do your macro variable references need a delimiter (the period) so that the macro processor can tell where the reference ends? If you are indirectly referencing a macro variable, do you have enough ampersands so that the macro processor attempts to resolve the references sufficiently to completely resolve your reference? Are you referencing a local macro variable outside of the macro program in which it has been defined? Do you use the same name for different macro variables and are the macro variables in different domains? If so, consider using unique variable names so that it becomes easier to distinguish the domain of a macro variable reference in your code.

Macro Variable Resolution (*continued*)	Did you create a macro variable in a DATA step with CALL SYMPUT or CALL SYMPUTX and then attempt to resolve it in the same DATA step? If so, you will need to modify the code to define the macro variable before the DATA step in which it is referenced or modify the code to use the RESOLVE SAS language function. Do any of your macro variables start with AF, DMS, or SYS? If so, you should rename them to prevent conflicts in names with automatic macro variables.
Macro Program Resolution	Have you submitted the macro program definition before calling the macro program? Do your macro program references start with a percent sign? Is your macro program stored in an autocall library or as a stored compiled macro program? If so, have you specified options correctly so that the macro processor can find the macro program?
System Options Settings	Is SAS option MACRO in effect so that you can access the macro facility? Is option MERROR in effect so that the macro processor displays warning messages when a macro program reference cannot be resolved? Is option SERROR in effect so that the macro processor displays warning messages when a macro variable reference cannot be resolved? Have all appropriate options been set (SASAUTOS, MAUTOSOURCE, MSTORED, SASMSTORE) when working with autocall libraries or stored compiled macro programs? Has MINDELIMITER been specified correctly if any of your macro language statements use the IN operator?
%MACRO, %MEND Statements	Do the names on the %MACRO and %MEND statements agree? Are your macro program definitions nested? If so, remove the nested macro program definitions so that it may become easier to identify the source of the problem.

Macro Parameters	If you're using positional parameters, have you specified the correct number of parameters in your macro program call, and have you specified them in the correct order? For parameters for which you do not want to specify a value, have you inserted a comma as placeholder for that missing parameter value? If you're using keyword parameters and you've specified defaults for them, have you specified the defaults correctly? If your macro program definition contains both positional and keyword parameters, have you placed the positional parameters ahead of the keyword parameters in the macro program call?
Macro Functions	Does your function call have the correct number of arguments? Have you quoted the arguments to the macro function as you would have with the SAS function counterpart? If so, remove the quotation marks. Does your function argument contain special characters such as commas? If so, you may have to mask the argument with macro quoting functions.
SAS Language vs. Macro Language	Have you mixed SAS language actions and macro language actions in the same statement? For example, is the result of a SAS language IF statement a macro language %LET statement? If so, you will have to modify your code because the macro statements execute before the SAS language statements. Have you forgotten percent signs on macro language keywords so that SAS interprets these as SAS language keywords instead?
%DO, %END statements	Do you have a matching %END for each %DO statement? On an iterative %DO statement, did you specify the index macro variable reference with an ampersand? If so, this may be incorrect if you are explicitly referencing a macro variable.

Processing Special Characters and Mnemonic Operators	Do your text strings contain special characters or mnemonic operators that should be interpreted as text? If so, you may need to apply a macro quoting function. When you are masking an ampersand or percent sign, are you using the "NR" version of the macro quoting function?
Calculations	Are you using %EVAL for integer arithmetic and %SYSEVALF for calculations that require floating-point arithmetic?
Logical Expressions	Can the operands in your expressions contain special characters or mnemonic operators? If so, you may need to mask the operands with macro quoting functions.

Understanding the Tools That Can Debug Macro Programming

As discussed previously and demonstrated in Chapter 13, you can minimize errors in your SAS programs that contain macro features by developing your programs in steps. When your programs do end up containing errors, you can use the tools described in this section to find the sources of the errors. These tools include system options, macro language statements, macro functions, and automatic macro variables. Using these tools can provide you with detail about the processing of your programs, and many have been applied in examples in previous chapters.

This section describes the use of these tools in the context of macro programming. Don't forget that macro programs can generate SAS programs. Those SAS programs may be in error, but your macro language processing may be correct. In those situations, you will need to employ your SAS language debugging skills. You may need to extract your SAS code that's in error and debug it. Then, if necessary, you would modify your macro language code to handle your corrected SAS code. For more information on debugging SAS language, refer to SAS documentation.

Using SAS System Options to Debug Macro Programming

Several system options can provide detail about the processing of your SAS programs that contain macro language and, thus, help you find the sources of errors in your programs. Table 12.2 lists these options. Options with the "NO" prefix turn off the associated system options.

Table 12.2 SAS system options useful in debugging macro programming

Option	Purpose
MLOGIC | NOMLOGIC	Traces the flow of execution of a macro program in the SAS log. MLOGIC shows the resolved values of macro parameters, the scope of macro variables (global or local), the true/false result of %IF statements, and the start and end of a macro program.
MLOGICNEST | NOMLOGICNEST	Displays the nesting level of macro programs in the SAS log. This information is displayed in the MLOGIC output in the SAS log, and MLOGIC must be enabled for this option to work.
MPRINT | NOMPRINT	Displays the SAS code generated by the execution of a macro program in the SAS log. Optionally, you can direct the output from MPRINT to an external file by specifying the MFILE system option.
MPRINTNEST | NOMPRINTNEST	Aligns the nesting level of macro programs and the SAS code generated by the execution of the macro programs in the SAS log.
SYMBOLGEN | NOSYMBOLGEN	Displays the resolution of macro variable references in the SAS log.

Chapter 3 presented several examples of showing resolution of macro variable references by enabling the SYMBOLGEN option. Chapter 4 introduced displaying the SAS code that a macro program generates by enabling the MPRINT option.

When processing and debugging macro language, it is important to verify that the three system options listed in Table 12.3 are enabled. These options affect whether macro processing is available and whether warnings and error messages related to the resolution of macro variables and invocation of macro programs are displayed. Options with the "NO" prefix turn off the associated system options.

Table 12.3 SAS system options that should be enabled when programming macro language

Option	Purpose
MACRO | NOMACRO	Controls whether the macro processor is available to recognize and process macro language
MERROR | NOMERROR	Controls whether the macro processor issues a warning message when a macro program reference cannot be resolved
SERROR | NOSERROR	Controls whether the macro processor issues a warning message when a macro variable reference cannot be resolved

Using Macro Language Statements to Debug Macro Programming

The primary macro language statement to use when you are debugging your macro programming is the %PUT statement. Adding %PUT statements to your macro code can help you determine the values of macro variables as your code processes. The %PUT statement writes text and/or macro variable values to the SAS log.

As described in the previous section, the SYMBOLGEN option also displays the values of macro variables. The difference between it and %PUT statements is that with the %PUT statement you tell the macro processor when and what to write to the SAS log, while the SYMBOLGEN option tells the macro processor to display the resolution of *all* macro variables as their references are encountered. You can generate a lot of output in the SAS log with SYMBOLGEN. The %PUT statement can reduce the amount of output you need to review in the SAS log and can add explanatory text to the display.

Additionally, when you debug a macro program you may want to temporarily add programming statements such as %IF and %GOTO to do specific actions, such as skipping over sections of code in the macro program in order to check on the processing.

If you are having problems with resolving macro variable references, you may want to delete the troublesome macro variables from the global symbol table using the %SYMDEL statement and attempt to execute your code again.

Using Macro Functions to Debug Macro Programming

The three macro functions, %SYMEXIST, %SYMGLOBL, and %SYMLOCAL, may be useful if you are having problems with resolving macro variable references and you suspect that the problem is with the scope (or domain) of the macro variables. You can add statements that call these functions to determine whether a macro variable exists and to determine the symbol table to which it belongs. See the section "Macro Variable Attribute Functions" in Chapter 6 and Table 6.5 for more information on these functions and an example of their usage.

Chapter 11 discussed building your library of utility macro programs and writing some of them to behave like functions. You may find that you always debug your macro programs a certain way and that you can create macro programs to do these actions. The macro programs that you develop for debugging purposes could then be added to your library of routines.

Accessing information using SAS language functions may also help you in finding solutions to problems in your code. For example, the ATTRN and ATTRC functions supply attribute information about SAS data files. The %SYSFUNC and %QSYSFUNC macro functions make these SAS language functions, and others, available to your macro programming, as described in Chapter 6 in the "Other Macro Functions" section.

Using Automatic Macro Variables to Debug Macro Programming

Chapter 3 presented information about automatic macro variables, which are the global macro variables that SAS defines that you can access with your macro programming statements. The values assigned to some of these automatic macro variables may be useful to you in debugging your macro programming. You may want to add statements to check the values of specific automatic macro variables and then execute specific statements based on these values.

Most of the automatic macro variables listed in Table 12.4 and shown in Table 3.2 could be useful in debugging your macro programs; several could be applied in debugging your SAS language programs as well. For example, starting in SAS 9.2, you could add %IF statements to check the values of SYSERRORTEXT and SYSWARNINGTEXT and then direct specific actions to take based on their values. Automatic macro variable SYSERRORTEXT contains the text of the last error message generated in the SAS log, and SYSWARNINGTEXT contains the text of the last warning message in the SAS log. These macro variables contain the error or warning text generated either by your SAS language statements or by your macro language statements. Use of these two macro variables is demonstrated in Example 12.8.

Table 12.4 Automatic macro variables useful in debugging

SYSDATE	SYSDATE9	SYSDAY	SYSDSN
SYSERR	SYSERRORTEXT	SYSFILRC	SYSLAST
SYSLIBRC	SYSMACRONAME	SYSPROCNAME	SYSRC
SYSTIME	SYSVER	SYSWARNINGTEXT	

Examples of Solving Errors in Macro Programming

In explaining the processing concepts of the macro facility, many examples in previous chapters included errors and showed how to resolve them. This section presents additional examples of errors in macro programming and how they could be detected and corrected.

Example 12.1: Reviewing System Options When Macro Facility Warnings and Error Messages Are Absent

There can be many possibilities to consider when your macro code does not execute and the macro facility does not write any warnings or error messages to the SAS log. In this situation, your first step might be to verify the values of system options MACRO, MERROR, and SERROR. Table 12.3 described these options.

The purpose of the following macro program PRINT10 in Program 12.1 is to list with PROC PRINT the first ten observations in the data set specified by the macro program's parameter DSNAME. Note that the reference to DSNAME in the title text is misspelled as DSNAMEE. Assume that option SERROR is disabled when the following program executes.

Program 12.1

```
options symbolgen;
%macro print10(dsname);
  proc print data=&dsname(obs=10);
    title "Listing First 10 Observations from &dsnamee";
  run;
%mend;
%print10(books.ytdsales)
```

Submitting the program does list the first ten observations if the data set specified by DSNAME exists. However, the TITLE statement resolves to the following:

```
Listing First 10 Observations from &dsnamee
```

The SAS log does not display any warnings or error messages, even with option SYMBOLGEN enabled. Since DSNAME exists, SYMBOLGEN can display its value.

```
67   %macro print10(dsname);
68     proc print data=&dsname(obs=10);
69       title "Listing First 10 Observations from &dsnamee";
70     run;
71   %mend;
72   %print10(books.ytdsales)

SYMBOLGEN:  Macro variable DSNAME resolves to books.ytdsales
NOTE: There were 10 observations read from the data set
BOOKS.YTDSALES.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The easiest condition to check is to verify the setting for the SERROR option. If SERROR was turned off, the macro processor would not inform you in the SAS log that it was unable to resolve a macro variable reference. Enabling SERROR and then submitting the macro program causes the following warning, which makes it easy to find and correct the problem.

```
WARNING: Apparent symbolic reference DSNAMEE not resolved.
```

Similarly, if your submitted code attempts to invoke a macro program and this program does not execute, nor do any warnings appear in the SAS log, you may want to verify the setting of the MERROR option. For example, consider the result of submitting the following macro program call with the MERROR option turned off. Assume that macro program PRINT100 is not available in this SAS session and the intent was to call macro program PRINT10.

```
%print100(books.ytdsales)
```

The SAS log looks like the following:

```
58   %print100(books.ytdsales)
     -
     180
ERROR 180-322: Statement is not valid or it is used out of
proper order.
```

An error message is generated, but it's not very helpful. With MERROR enabled, submitting the call to PRINT100 produces the following SAS log with a clear warning about the source of the problem.

```
65   %print100(books.ytdsales)
     -
     180
WARNING: Apparent invocation of macro PRINT100 not resolved.
ERROR 180-322: Statement is not valid or it is used out of
proper order.
```

Example 12.2: Attempting to Process a Macro Program When the Macro Processor Does Not Detect a %MEND Statement

Program 12.2 illustrates how SAS processes a macro program definition and subsequent SAS language code when the macro processor does not detect the end of the macro program definition. This situation most likely arises when your macro program definition is missing a semicolon or it contains unmatched quotation marks or parentheses.

Recall that the macro processor requires that a %MEND statement terminate a macro program definition. When the macro processor does not detect a %MEND statement, a cascade of problems can occur. All code submitted after your undetected %MEND statement becomes part of the macro program definition. This continues until you submit another %MEND statement that the macro processor detects.

All of the code that may get incorrectly added to your macro program definition can cause problems in your attempts to correct the situation of the missing %MEND statement. Your session may hang and any additional code you submit just adds to the problem. In such a situation, you may be able to free your SAS session by submitting the following string:

```
*'; *"; *); */; %mend; run;
```

Continue submitting this string until you see the message:

```
ERROR: No matching %MACRO statement for this %MEND statement.
```

This string can clear up the problems with unmatched quotation marks and parentheses as well as missing semicolons and %MEND statements. If these attempts do not work, you will have to close your SAS session and restart SAS.

The goal of macro program WHSTMT is to specify a WHERE statement based on the values of the two parameters to WHSTMT. The two optional parameters are GETSECTION and GETPUB. Macro parameter GETSECTION specifies the bookstore section to select from BOOKS.YTDSALES while macro parameter GETPUB specifies

the publisher. If you do not specify at least one of the parameters, the DATA step creates a copy of BOOKS.YTDSALES.

The DATA step creates a temporary data set TEMP. It calls macro program WHSTMT and directs that it specify a WHERE statement, which selects observations from the Internet section and from the publisher Technology Smith. The error in the macro program definition for WHSTMT is the missing semicolon on the last %END statement.

Program 12.2

```
%macro whstmt(getsection,getpub);
  %if &getsection ne or &getpub ne %then %do;
    (where=((
  %end;
  %if &getsection ne %then %do;
    section="&getsection"
    %if &getpub ne %then %do;
      and
    %end;
    %else %do;
      )))
    %end;
  %end;
  %if &getpub ne %then %do;
    publisher="&getpub")))
  %end
%mend whstmt;
data temp;
  set books.ytdsales
    %whstmt(Internet,Technology Smith)
  ;
run;
```

When you submit the code, the DATA step does not execute as shown in the following SAS log. The log does show that the macro processor has detected a possible problem in the macro program and that the problem is extraneous information on the %END statement. The extraneous information happens to be the %MEND statement.

Note that the log does not show any DATA step processing notes. The DATA step does not execute. It has become part of the incomplete macro program definition.

```
623  %macro whstmt(getsection,getpub);
624    %if &getsection ne or &getpub ne %then %do;
625      (where=((
626    %end;
627    %if &getsection ne %then %do;
628      section="&getsection"
```

```
629      %if &getpub ne %then %do;
630        and
631      %end;
632      %else %do;
633        )))
634      %end;
635    %end;
636    %if &getpub ne %then %do;
637      publisher="&getpub")))
638    %end
639  %mend whstmt;
NOTE: Extraneous information on %END statement ignored.
640  data temp;
641    set books.ytdsales
642      %whstmt(Internet,Technology Smith)
643    ;
644  run;
```

Submitting another %MEND statement does terminate the macro program definition, but the definition is incorrect. At this point, you would need to add the semicolon to the %END statement and then resubmit the entire program to replace the incorrect definition of WHSTMT.

Example 12.3: Tracing Problems in Expression Evaluation with the %PUT Statement and the MLOGIC System Option

This example illustrates the importance of understanding how the macro processor evaluates expressions, and it shows how you can determine the source of problems with expression evaluation using the %PUT statement and the MLOGIC system option. It also shows that debugging a program may be an iterative process: after resolving one problem, you may find that you still have other problems in your program.

Chapter 6 presented information about the two SAS evaluation functions, %EVAL and %SYSEVALF. Example 7.1 in Chapter 7 showed several usages of the two functions. In general, when you're working with integer values, you use %EVAL, and when you're working with floating-point numbers, you use %SYSEVALF.

Macro program MARKUP in Program 12.3a submits a DATA step that selects observations from BOOKS.YTDSALES for the publisher specified by the macro parameter PUBLISHER. The three other parameters to MARKUP are RATE1, RATE2, and RATE3. These parameters specify three markup rates to be applied to the data set variable COST. The DATA step creates four data set variables. Three of the variables, COST1, COST2, and COST3 contain new cost values based on multiplying data set variable COST by macro variables RATE1, RATE2, and RATE3 respectively. The

fourth data set variable RATEPLUS is a character variable whose value depends on the difference between macro variables RATE1 and RATE3.

The macro program definition in Program 12.3a compiles without error. The call to MARKUP specified below, however, does not execute as expected.

Program 12.3a

```
%macro markup(publisher,rate1,rate2,rate3);
  %let diffrate=&rate3-&rate1;

  data pubmarkup;
    set books.ytdsales(where=(publisher="&publisher"));

    %if &diffrate ge 5.00 %then %do;
      retain rateplus '+++';
    %end;
    %else %if &diffrate lt 5.00 and &diffrate ge 0.00 %then %do;
      retain rateplus '+';
    %end;
    %else %do;
      retain rateplus '-';
    %end;

    %do i=1 %to 3;
      cost&i=cost* (1+(&&rate&i/100));
    %end;
  run;
%mend markup;

%markup(Technology Smith,2.25,4,7.25)
```

The SAS log for Program 12.3a reports a problem in evaluating the expression on the first %IF statement. The DATA step also has not stopped processing. Submitting a RUN; statement terminates the DATA step.

```
854  %markup(Technology Smith,2.25,4,7.25)
ERROR: A character operand was found in the %EVAL function or
       %IF condition where a numeric operand is required. The
       condition was: &diffrate ge 5.00
ERROR: The macro MARKUP will stop executing.
```

The value of macro variable DIFFRATE should be a number, but the error states that a character operand was found in the expression. A first step might be to include after the

first %LET statement a %PUT statement that displays the value of macro variable DIFFRATE:

```
%put Value of DIFFRATE is &diffrate;
```

Submitting the edited macro program definition and the same macro program call writes the following text to the SAS log.

```
Value of DIFFRATE is 7.25-2.25
```

This output shows that the expression on the %LET statement was not treated as an arithmetic calculation. The value of DIFFRATE is equal to the expression that was supposed to be evaluated. You must tell the macro processor when to evaluate the expression rather than have it treat the expression simply as text. Placing the %SYSEVALF function around the expression tells the macro processor to compute a numeric value:

```
%let diffrate=%sysevalf(&rate3-&rate1);
```

If you used %EVAL instead of %SYSEVALF, you would see an error message similar to the one shown above. The %EVAL function tells the macro processor to do integer arithmetic. The %EVAL function interprets the decimal points as text and, thus, it cannot calculate a numeric result in this situation.

The program executes without error when the %SYSEVALF function encloses the expression on the %LET statement. Reviewing the parameters and the data in the output data set shows, however, that the value of RATEPLUS is incorrect.

The difference between the first and third parameters is 5. Therefore, RATEPLUS should equal +++. The value of RATEPLUS in the data set, however, is +. That means that the %ELSE-%IF statement was executed. Submitting the program with the MLOGIC option enabled verifies that the %ELSE-%IF statement executed.

```
1008  %markup(Technology smith,2.25,4,7.25)
MLOGIC(MARKUP):  Beginning execution.
MLOGIC(MARKUP):  Parameter PUBLISHER has value Technology Smith
MLOGIC(MARKUP):  Parameter RATE1 has value 2.25
MLOGIC(MARKUP):  Parameter RATE2 has value 4
MLOGIC(MARKUP):  Parameter RATE3 has value 7.25
MLOGIC(MARKUP):  %LET (variable name is DIFFRATE)
MLOGIC(MARKUP):  %IF condition &diffrate ge 5.00 is FALSE
MLOGIC(MARKUP):  %IF condition &diffrate lt 5.00 and &diffrate
                 ge 0.00 is TRUE
MLOGIC(MARKUP):  %DO loop beginning; index variable I; start
                 value is
                 1; stop value is 3; by value is 1.
MLOGIC(MARKUP):  %DO loop index variable I is now 2; loop will
                 iterate again.
```

```
MLOGIC(MARKUP):  %DO loop index variable I is now 3; loop will
                 iterate again.
MLOGIC(MARKUP):  %DO loop index variable I is now 4; loop will
                 not iterate again.

NOTE: There were 505 observations read from the data set
      BOOKS.YTDSALES.
      WHERE publisher='Technology Smith';
NOTE: The data set WORK.PUBMARKUP has 505 observations and 14
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
MLOGIC(MARKUP):  Ending execution.
```

The %ELSE-%IF statement executed because there is an implied %EVAL around an expression on a %IF statement. The decimal point in the expression makes this a text evaluation, and the text value 5 is less than the text value 5.00.

The final correction is to apply the %SYSEVALF function to the expressions on the %IF statement and on the %ELSE-%IF statement. The %SYSEVALF function executes first, yielding a true/false (1/0) result. The macro processor next applies the implicit %EVAL function to the true/false (1/0) result. The corrected program follows in Program 12.3b.

Program 12.3b

```
%macro markup(publisher,rate1,rate2,rate3);
  %let diffrate=%sysevalf(&rate3-&rate1);

  data pubmarkup;
    set books.ytdsales(where=(publisher="&publisher"));

    %if %sysevalf(&diffrate ge 5.00) %then %do;
      retain rateplus '+++';
    %end;
    %else %if %sysevalf(&diffrate lt 5.00) and
           %sysevalf(&diffrate ge 0.00) %then %do;
      retain rateplus '+';
    %end;
    %else %do;
      retain rateplus '-';
    %end;

    %do i=1 %to 3;
      cost&i=cost* (1+(&&rate&i/100));
    %end;
  run;
```

```
%mend markup;

%markup(Technology Smith,2.25,4,7.25)
```

The data in the PUBMARKUP data set is now correct with the value of RATEPLUS equal to +++. The SAS log for the revised program with the MLOGIC option enabled shows that the %IF statement executed.

```
1031  %markup(Technology Smith,2.25,4,7.25)
MLOGIC(MARKUP):  Beginning execution.
MLOGIC(MARKUP):  Parameter PUBLISHER has value Technology Smith
MLOGIC(MARKUP):  Parameter RATE1 has value 2.25
MLOGIC(MARKUP):  Parameter RATE2 has value 4
MLOGIC(MARKUP):  Parameter RATE3 has value 7.25
MLOGIC(MARKUP):  %LET (variable name is DIFFRATE)
MLOGIC(MARKUP):  %IF condition %sysevalf(&diffrate ge 5.00) is
                 TRUE
MLOGIC(MARKUP):  %DO loop beginning; index variable I; start
                 value is
                 1; stop value is 3; by value is 1.
MLOGIC(MARKUP):  %DO loop index variable I is now 2; loop will
                 iterate again.
MLOGIC(MARKUP):  %DO loop index variable I is now 3; loop will
                 iterate again.
MLOGIC(MARKUP):  %DO loop index variable I is now 4; loop will
                 not iterate again.

NOTE: There were 505 observations read from the data set
      BOOKS.YTDSALES.
      WHERE publisher='Technology Smith';
NOTE: The data set WORK.PUBMARKUP has 505 observations and 14
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
MLOGIC(MARKUP):  Ending execution.
```

Example 12.4: Using %PUT to Trace a Problem at Execution

This example illustrates the importance of distinguishing between macro language syntax and SAS language syntax. The macro program TABLES builds a PROC TABULATE TABLE statement for each variable in the parameter CLASS_STRING. Each variable in CLASS_STRING is a classification variable. The %SCAN function extracts each of the variable names and saves the variable name in macro variable CLASSVAR. The %DO %UNTIL loop should iterate for each variable name and stop when there are no more variable names.

The error in the macro program is the incorrect specification of the expression in the %DO %UNTIL statement. The %DO %UNTIL statement is partially written in SAS language syntax.

The macro program in Program 12.4a compiles without error, but the macro program's %DO %UNTIL loop executes indefinitely. This example uses a %PUT statement to display macro variable values during each iteration of a %DO %UNTIL loop to determine why it executes indefinitely.

Since %DO %UNTIL is a *macro language* statement, the expression should check whether CLASSVAR is null, not whether it is equal to the text ' '. Specifying the value of ' ' is the way to test for a missing character value in *SAS language*. Macro program TABLES in Program 12.4a executes indefinitely because the value extracted by %SCAN will never equal the text ' '.

Program 12.4a

```
%macro tables(class_string);
  class datesold &class_string;
  %let varnum=1;
  %let classvar=%scan(&class_string,&varnum);

  %do %until (&classvar=' ');
    tables datesold='Books Sold Quarter'
           all='Books Sold All Four Quarters',
           (&classvar all),
           (cost listprice saleprice)*sum=' '*f=dollar12.2 ;

    %let varnum=%eval(&varnum+1);
    %let classvar=%scan(&class_string,&varnum);
  %end;
%mend tables;

proc tabulate data=books.ytdsales;
  title "Quarterly Book Sales Summaries";
  var cost listprice saleprice;
  format datesold qtr.;
  keylabel all='Total';

  %tables(section publisher)
run;
```

A %DO %UNTIL loop evaluates the expression at the bottom of the loop. A first step in finding the problem might be to display the values of macro variables VARNUM and CLASSVAR in each iteration of the %DO %UNTIL loop. Add the following %PUT statement just before the %END statement.

```
%put **** VARNUM=&varnum   CLASSVAR=&classvar;
```

When the program with the %PUT statement included executes and is then canceled after it becomes obvious that it is going to execute indefinitely, the partial SAS log that follows is the result. The MPRINT option is set to show how the macro program builds the TABLE statements.

The parameter CLASS_STRING contains the name of two variables. The %SCAN function correctly extracts two variable names. When the value of VARNUM is 3, the program should stop. The %PUT statement shows that VARNUM continues to increment and that CLASSVAR has no value. The macro program continues to build TABLE statements.

The %PUT statement output shows that VARNUM increments correctly and that the %SCAN function extracts variable names correctly. That leaves the expression in the %DO %UNTIL statement as the likely source of the problem.

```
252  options mprint;
253  %macro tables(class_string);
254    class datesold &class_string;
255    %let varnum=1;
256    %let classvar=%scan(&class_string,&varnum);
257
258    %do %until (&classvar=' ');
259      tables datesold='Books Sold Quarter' all='Books Sold
259! All Four Quarters',
260             (&classvar all),
261             (cost listprice saleprice)*sum=' '*f=dollar12.2
261! ;
262
263      %let varnum=%eval(&varnum+1);
264      %let classvar=%scan(&class_string,&varnum);
265      %put **** VARNUM=&varnum   CLASSVAR=&classvar;
266    %end;
267  %mend tables;
268
269  proc tabulate data=books.ytdsales;
270    title "Quarterly Book Sales Summaries";
271    var cost listprice saleprice;
272    format datesold qtr.;
273    keylabel all='Total';
274
275    %tables(section publisher)
MPRINT(TABLES):   class datesold section publisher;
MPRINT(TABLES):   tables datesold='Books Sold Quarter'
all='Books Sold All Four Quarters', (section all), (cost
listprice saleprice)*sum=' '*f=dollar12.2 ;
```

```
**** VARNUM=2   CLASSVAR=publisher
MPRINT(TABLES):   tables datesold='Books Sold Quarter'
all='Books Sold All Four Quarters', (publisher all), (cost
listprice saleprice)*sum=' '*f=dollar12.2 ;
**** VARNUM=3   CLASSVAR=
MPRINT(TABLES):   tables datesold='Books Sold Quarter'
all='Books Sold All Four Quarters', ( all), (cost listprice
saleprice)*sum=' '*f=dollar12.2 ;
**** VARNUM=4   CLASSVAR=
MPRINT(TABLES):   tables datesold='Books Sold Quarter'
all='Books Sold All Four Quarters', ( all), (cost listprice
saleprice)*sum=' '*f=dollar12.2 ;
**** VARNUM=5   CLASSVAR=
MPRINT(TABLES):   tables datesold='Books Sold Quarter'
all='Books Sold All Four Quarters', ( all), (cost listprice
saleprice)*sum=' '*f=dollar12.2 ;
```

The program with the corrected %DO %UNTIL statement follows in Program 12.4b.

Program 12.4b

```
%macro tables(class_string);
  class datesold &class_string;
  %let varnum=1;
  %do %until (&classvar=);
    %let classvar=%scan(&class_string,&varnum);
    tables datesold='Books Sold Quarter'
           all='Books Sold All Four Quarters',
           (&classvar all),
           (cost listprice saleprice)*sum=' '*f=dollar12.2 ;

    %let varnum=%eval(&varnum+1);
    %let classvar=%scan(&class_string,&varnum);
  %end;
%mend tables;

proc tabulate data=books.ytdsales;
  title "Quarterly Book Sales Summaries";
  var cost listprice saleprice;
  format datesold qtr.;
  keylabel all='Total';

  %tables(section publisher)
run;
```

Example 12.5: Finding a Logic Error in the Execution of a Macro Program with the MLOGIC Option

This example shows how the MLOGIC option can help you trace execution of a macro program to identify the source of a logic error. When your macro program compiles without error, yet the output you expect is not produced, you may want to enable the MLOGIC option. The MLOGIC option lists in the SAS log the processing actions the macro processor takes.

The goal of the program is to process records for a specific publisher and create an HTML file and/or a Microsoft Excel spreadsheet of the selected records. To do this, the program defines three macro programs: MAKEHTML, MAKEXLS, and EXTFILES. Macro program MAKEHTML lists observations with PROC PRINT and creates an HTML file of the report. Macro program MAKEXLS creates a Microsoft Excel spreadsheet containing the selected observations. Macro program EXTFILES creates the subset data set and calls the other two macro programs when specified to do so by its parameter values.

Macro program EXTFILES has three parameters: PUBLISHER, HTML, and SPREADSHEET. Positional parameter PUBLISHER specifies the publisher name whose observations should be selected from BOOKS.YTDSALES. Keyword parameters HTML and SPREADSHEET are defined to have one of two values: Y or N. Specify the value Y for the parameter value when you want the specific output file produced; specify N when you do not.

The problem with Program 12.5a program is the %ELSE statement in macro program EXTFILES. The tests for producing the output should be independent of each other. That is, it should be possible to produce one or both of the reports based on the values of the parameters. The %ELSE statement prevents this. When the request is made to create an HTML file (HTML=Y) and to create an Excel file (SPREADSHEET=Y), the %ELSE statement prevents execution of %MAKEXLS.

The call to EXTFILES specifies that both an HTML and an Excel file should be created for publisher Eversons Books.

Program 12.5a

```
%macro extfiles(publisher,html=,spreadsheet=);
  data temp;
    set books.ytdsales(where=(publisher="&publisher")
                       drop=section saleid saleinit listprice);
  run;

  %if &html=Y %then %do;
    %makehtml
  %end;
```

```
  %else %if &spreadsheet=Y %then %do;
    %makexls
  %end;
%mend extfiles;

%macro makehtml;
  ods listing close;
  ods html;
  proc print data=temp;
    title "Publisher: &publisher";
  run;
  ods html close;
  ods listing;
%mend makehtml;

%macro makexls;
  proc export data=temp
              file="pubreports.xls"
              replace;
              sheet="&publisher";
  run;
%mend makexls;

%extfiles(Eversons Books,html=Y,spreadsheet=Y)
```

Program 12.5a creates only an HTML file. The SAS log does not show any errors in processing.

The next step is to submit the program again with the MLOGIC option enabled. The MLOGIC option displays the results of the %IF and %ELSE statements. The SAS log with MLOGIC enabled follows. It shows that the %ELSE statement did not execute and that MAKEXLS was not called. The conclusion is that the error is in the specification of the %ELSE statement.

```
185  %extfiles(Eversons Books,html=Y,spreadsheet=Y)
MLOGIC(EXTFILES):  Beginning execution.
MLOGIC(EXTFILES):  Parameter PUBLISHER has value Eversons Books
MLOGIC(EXTFILES):  Parameter HTML has value Y
MLOGIC(EXTFILES):  Parameter SPREADSHEET has value Y

NOTE: There were 542 observations read from the data set
BOOKS.YTDSALES.
      WHERE publisher='Eversons Books';
NOTE: The data set WORK.TEMP has 542 observations and 6
      variables.
```

```
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

MLOGIC(EXTFILES):  %IF condition &html=Y is TRUE
MLOGIC(MAKEHTML):  Beginning execution.
NOTE: Writing HTML Body file: sashtml2.htm
NOTE: There were 542 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.31 seconds
      cpu time            0.20 seconds

MLOGIC(MAKEHTML):  Ending execution.
MLOGIC(EXTFILES):  Ending execution.
```

Replacing the %ELSE statement with %IF corrects the logic error in EXTFILES. The revised EXTFILES macro program follows in Program 12.5b.

Program 12.5b

```
%macro extfiles(publisher,html=,spreadsheet=);
  data temp;
    set books.ytdsales(where=(publisher="&publisher")
                       drop=section saleid saleinit listprice);
  run;

  %if &html=Y %then %do;
    %makehtml
  %end;
  %if &spreadsheet=Y %then %do;
    %makexls
  %end;
%mend extfiles;
```

Now the SAS log shows that both MAKEHTML and MAKEXLS execute and that both the HTML file and the Excel file are created.

```
219  %extfiles(Eversons Books,html=Y,spreadsheet=Y)
MLOGIC(EXTFILES):  Beginning execution.
MLOGIC(EXTFILES):  Parameter PUBLISHER has value Eversons Books
MLOGIC(EXTFILES):  Parameter HTML has value Y
MLOGIC(EXTFILES):  Parameter SPREADSHEET has value Y

NOTE: There were 542 observations read from the data set
      BOOKS.YTDSALES.
      WHERE publisher='Eversons Books';
```

```
NOTE: The data set WORK.TEMP has 542 observations and 6
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MLOGIC(EXTFILES):  %IF condition &html=Y is TRUE
MLOGIC(MAKEHTML):  Beginning execution.
NOTE: Writing HTML Body file: sashtml3.htm
NOTE: There were 542 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.32 seconds
      cpu time            0.21 seconds

MLOGIC(MAKEHTML):  Ending execution.
MLOGIC(EXTFILES):  %IF condition &spreadsheet=Y is TRUE
MLOGIC(MAKEXLS):  Beginning execution.
NOTE: "Everyday_Books" was successfully created.
NOTE: PROCEDURE EXPORT used (Total process time):
      real time           0.40 seconds
      cpu time            0.04 seconds

MLOGIC(MAKEXLS):  Ending execution.
MLOGIC(EXTFILES):  Ending execution.
```

Example 12.6: Using the MPRINT Option to Find Errors in SAS Language That Was Generated by Macro Language

This example demonstrates how the MPRINT system option can list SAS language statements generated by a macro program. When the SAS language that your macro program generates isn't what you expect, enable MPRINT so that the macro processor lists the SAS language statements in the SAS log for your review. This technique helps you uncover both macro language and SAS language problems.

With NOMPRINT in effect, the macro processor does not list the SAS language statements generated by your macro program; your SAS log contains only the standard messages issued when a step ends.

This program generates a PROC TABULATE report that can summarize projected costs for future years from 2008 to 2012. The DATA step that creates data set PROJCOST computes projected costs using a different percentage increase for each of the five years from 2008 to 2012.

The macro program PROJCOST has one parameter, ANALYSISVARS. This parameter's value is the list of analysis variables for the PROC TABULATE table. Both the VAR statement and the TABLES statement in the PROC TABULATE step reference ANALYSISVARS.

The report should summarize cost information for each section. The categorical variable, SECTION, is specified on the CLASS statement. Each analysis variable should produce two statistics: mean and sum.

The error in Program 12.6a is in the SAS code generated by the macro program. Parentheses are missing around the macro variable reference to ANALYSISVARS in the TABLES statement. When you specify one analysis variable, PROC TABULATE computes the two statistics for the analysis variable. When you specify more than one analysis variable, PROC TABULATE computes the two statistics only for the last analysis variable in the list.

The three analysis variables specified in the call to PROJCOST are COST, PCOST2008, and PCOST2010.

Program 12.6a

```
%macro projcost(analysisvars);
  proc tabulate data=projcost;
    title "Projected Costs Report";
    class section;
    var &analysisvars;
    tables section all='All Sections',
           &analysisvars*(mean*f=dollar7.2 sum*f=dollar12.2);
  run;
%mend projcost;

data projcost;
  set books.ytdsales;

  array increase{5} increase2008-increase2012
      (1.12,1.08,1.10,1.15,1.18);
  array pcost{5} pcost2008-pcost2012;
  drop i;

  attrib pcost2008 label="Projected Cost 2008" format=dollar10.2
         pcost2009 label="Projected Cost 2009" format=dollar10.2
         pcost2010 label="Projected Cost 2010" format=dollar10.2
         pcost2011 label="Projected Cost 2011" format=dollar10.2
         pcost2012 label="Projected Cost 2012" format=dollar10.2;
```

```
  do i=1 to 5;
    pcost{i}=round(cost*increase{i},.01);
  end;
run;

%projcost(cost pcost2008 pcost2010)
```

Output 12.1 presents the output produced by the call to PROJCOST. It shows that PROC TABULATE computed only the SUM statistic for the first and second analysis variables, COST and PCOST2008. For the third analysis variable, PCOST2010, it computed both the mean and the sum statistics and formatted the results with the DOLLAR format.

Output 12.1 Report produced by a call to macro program PROJCOST in Program 12.6a

Projected Costs Report

	Wholesale Cost	Projected Cost 2008	Projected Cost 2010	
	Sum	Sum	Mean	Sum
Section				
Certification and Training	16503.31	18482.54	$25.00	$18,152.20
Internet	32378.67	36261.72	$24.46	$35,613.59
Networks and Telecommunication	16033.41	17956.18	$24.60	$17,635.25
Operating Systems	20669.25	23147.96	$24.66	$22,734.20
Programming and Applications	32299.31	36172.81	$24.86	$35,526.29
Web Design	19289.25	21602.49	$25.08	$21,216.39
All Sections	137173.19	153623.70	$24.75	$150,877.92

With MPRINT enabled, the macro processor lists in the SAS log the PROC TABULATE statements that macro program PROJCOST constructs. Examining the PROC TABULATE step shows that the parentheses were omitted.

The SAS log for the above program follows, now with MPRINT enabled. The code for PROC TABULATE shows statistics specified only for the last analysis variable specified in parameter ANALYSISVARS. When you do not specify a statistic, PROC TABULATE defaults to computing the sum, which it did for COST and PCOST2008, as shown in Output 12.1.

```
637  %projcost(cost pcost2008 pcost2010)
MPRINT(PROJCOST):   proc tabulate data=projcost;
MPRINT(PROJCOST):   title "Projected Costs Report";
MPRINT(PROJCOST):   class section;
MPRINT(PROJCOST):   var cost pcost2008 pcost2010;
MPRINT(PROJCOST):   tables section all='All Sections', cost
pcost2008 pcost2010*(mean*f=dollar7.2 sum*f=dollar12.2);
MPRINT(PROJCOST):   run;

NOTE: There were 6096 observations read from the data set
      WORK.PROJCOST.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.04 seconds
      cpu time            0.04 seconds
```

The following modified PROJCOST macro program in Program 12.6b includes parentheses around &ANALYSISVARS in the TABLE statement.

Program 12.6b

```
%macro projcost(analysisvars);
  proc tabulate data=projcost;
    title "Projected Costs Report";
    class section;
    var &analysisvars;
    tables section all='All Sections',
           (&analysisvars)*(mean*f=dollar7.2 sum*f=dollar12.2);
  run;
%mend projcost;
```

If you resubmit the call to PROJCOST after revising it, the program computes the two statistics for each of the three analysis variables and formats the results with the DOLLAR format.

Improving Your Macro Programming by Including Error Checking

The previous topics in this chapter show ways of identifying the sources of problems that exist in your macro programming. This section discusses how you can add statements to your macro programs that look for and prevent errors in the processing of your macro programs. When your code detects problems, you can program specific actions to occur to prevent abnormal termination of the program and execution of incorrect SAS language code.

Generally, the more use your macro program will have, the more time you should invest in adding error checking to your code. If you are the only user of the macro program and you only need to execute it a few times, you may not need to add many error checking statements. On the other hand, if your macro program is more complicated and you plan to distribute it to others, it becomes more important to extensively check for errors and provide messages to assist your users. The remainder of this section presents examples that include error checking in the macro program code.

Example 12.7: Evaluating Parameter Values

A common check to add to a macro program is to evaluate parameter values to determine if values were specified and whether the values were specified correctly. This example starts with code that evaluates the parameters before it processes the data set.

Macro program SELECTTITLES defines three keyword parameters, MONTHSOLD, MINSALEPRICE, and PUBLISHER. The values of these three parameters specify a subset of data set BOOKS.YTDSALES. PROC PRINT then lists the selected observations. Macro program SELECTTITLES requires that all three parameters be specified and that they should be specified as follows.

- The value of MONTHSOLD must be an integer from 1 to 12.
- The value of MINSALEPRICE must be numeric and positive. If the user includes a dollar sign or commas in the value, include code to remove them so that the WHERE statement applied on the PROC PRINT statement processes correctly.
- Quote the value of PUBLISHER so that special characters and mnemonic operators included in the value are masked. Compress multiple blanks and uppercase the value of PUBLISHER to minimize differences in the way the user specifies the value and the way the value is stored in the data set.

Program 12.7

```
%macro selecttitles(monthsold=,minsaleprice=,publisher=);
  %* All three parameters must be specified.;
  %* Quote the value of PUBLISHER in case it contains special
     characters or mnemonic operators;❶          ❷
  %if &monthsold= or &minsaleprice= or %superq(publisher)= %then
  %do;
    %put ****************************************************;
    %put * Macro program SELECTTITLES requires you to specify
all;
    %put * three parameters. At least one was not specified:;
    %put * MONTHSOLD=&monthsold;
    %put * MINSALEPRICE=&minsaleprice;
    %put * PUBLISHER=&publisher;
    %put * Please correct and resubmit.;
    %put ****************************************************;
    %goto exit;
  %end;

  %* Check if parameters are valid;
  %* MONTHSOLD must be numeric and 1 to 12; ❸
  %if %sysfunc(notdigit(&monthsold)) gt 0 or
        &monthsold lt 1 or &monthsold gt 12 %then %do;
    %put ****************************************************;
    %put ERROR: MONTHSOLD was not specified correctly: &monthsold;
    %put Specify MONTHSOLD as an integer from 1 to 12;
    %put ****************************************************;
    %goto exit;
  %end;

  %* MINSALEPRICE must be numeric greater than 0 and if dollar
     signs and commas included, remove them;
  %let minsaleprice=%sysfunc(compress (&minsaleprice,%str(,)$));❹
  %if %sysfunc(notdigit(%sysfunc(compress(&minsaleprice,.))))
      gt 0
          %then %do;  ❺
    %put ****************************************************;
    %put ERROR: MINSALEPRICE was not specified correctly:
     &minsaleprice;
    %put ****************************************************;
    %goto exit;
  %end;
```

```
  %* Uppercase value of PUBLISHER and remove multiple blanks.
     Use quoting functions since value might contain special
     characters or mnemonic operators;
  %let publisher=%qupcase(%superq(publisher)); ❻
  %let publisher=%qcmpres(%superq(publisher)); ❼

  proc print data=books.ytdsales(where=
                 (month(datesold)=&monthsold and
                  saleprice ge &minsaleprice and
                  upcase(publisher)="&publisher"))
                 noobs n="Number of Books Sold=";
    title "Titles Sold during Month
%sysfunc(putn(&monthsold,monname.))";
    title2 "Minimum Sale Price of $&minsaleprice";
    title3 "Publisher &publisher";
  run;

%exit:
%mend selecttitles;

%selecttitles(monthsold=2,minsaleprice=$50.95,
              publisher=%nrstr(Doe&Lee   Ltd.))
```

Sections of the macro program that process the parameter values are identified by number:

❶ All three parameters must have values.

❷ Quote the value of PUBLISHER for this test.

❸ The value of MONTHSOLD must be numeric and an integer from 1 to 12.

❹ Remove dollar signs and commas from the value of MINSALEPRICE.

❺ The value of MINSALEPRICE must be numeric and positive.

❻ Uppercase and quote the value of PUBLISHER.

❼ Remove multiple blanks from the value of PUBLISHER and quote the value of PUBLISHER.

All three parameters to SELECTTITLES are specified correctly in the call to the macro program, and a PROC PRINT report is produced. The value for PUBLISHER is quoted in the call with %NRSTR since its value contains special characters. SELECTTITLES removes the dollar sign in the value for MINSALEPRICE, and it removes the multiple blanks from the value of PUBLISHER.

Example 12.8: Reviewing SAS Processing Messages

Starting in SAS 9.2, automatic macro variables SYSERRORTEXT and SYSWARNINGTEXT, respectively, retain the text of the last error message and the last warning message generated in the SAS log in the current SAS session. Macro program LASTMSG in Program 12.8 checks whether any text has been stored in either of these two automatic variables. It writes messages to the log indicating what it finds.

The steps preceding the call to %LASTMSG contain problems and errors that generate warning and error messages. Macro program %LASTMSG, however, only lists the last error message and the last warning message generated in the SAS log.

The problems and errors in Program 12.8 are in bold. In this example, assume ODS style BOOKSTORE does not exist. In the PROC FREQ step, variable DATESOLD is misspelled. The libref BOOOKS referenced in the DATA step has not been defined. Assume that libref BOOKS was defined before Program 12.8 was submitted.

Program 12.8

```
%macro lastmsg;

  %* Check last warning message;
  %put;
  %if %bquote(&syswarningtext) eq %then
      %put No warnings generated so far in this SAS session;
  %else %do;
    %put Last warning message generated in this SAS session:;
    %put &syswarningtext;
  %end;

  %put;
  %* Check last error message;
  %if %bquote(&syserrortext) eq %then
    %put No error messages generated so far in this SAS session;
  %else %do;
    %put Last error message generated in this SAS session:;
    %put &syserrortext;
    %put;
  %end;
  %put;
%mend lastmsg;

ods rtf style=bookstore;
```

```
proc freq data=books.ytdsales;
  tables datesoldd;
  format datesold monname.;
run;

data profit;
  set boooks.ytdsales;

  profit=saleprice-cost;
run;

ods rtf close;

%lastmsg
```

The SAS log for Program 12.8 follows, starting with submission of the first ODS RTF statement. The error and warning messages are in bold. The last four lines shown in this excerpt are the results of the %PUT statements in macro program %LASTMSG. Note that the value of SYSWARNINGTEXT is equal to the second warning associated with the DATA step. The value of SYSERRORTEXT is equal to the second of the two error messages generated in the SAS log, the one that indicates that libref BOOOKS was not assigned.

```
3256  ods rtf style=bookstore;
WARNING: Style BOOKSTORE not found; Rtf style will be used
         instead.
NOTE: Writing RTF Body file: sasrtf.rtf
3257
3258  proc freq data=books.ytdsales;
3259    tables datesoldd;
ERROR: Variable DATESOLDD not found.
3260    format datesold monname.;
3261  run;

NOTE: The SAS System stopped processing this step because of
      errors.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

3262
3263  data profit;
3264    set boooks.ytdsales;
ERROR: Libname BOOOKS is not assigned.
3265
3266    profit=saleprice-cost;
```

```
3267  run;

NOTE: The SAS System stopped processing this step because of
      errors.
WARNING: The data set WORK.PROFIT may be incomplete.  When this
         step was stopped there were 0 observations and 3
         variables.
WARNING: Data set WORK.PROFIT was not replaced because this
         step was stopped.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
3268
3269  ods rtf close;
3270
3271  %lastmsg

Last warning message generated in this SAS session:
Data set WORK.PROFIT was not replaced because this step was
stopped.

Last error message generated in this SAS session:
Libname BOOOKS is not assigned.
```

Example 12.9: Reviewing SAS Processing Results

Chapter 7 presented examples of macro programs that use %IF-%THEN statements to execute specific SAS steps. When enhancing your macro programming code, you may want to execute specific sections based on the results. For example, if an analysis generates no results, you may not be able to produce a subsequent report. Your users may not understand why the macro program did not produce the expected report. To assist your users, your macro program could check the results of the analysis and then determine the next step that executes. An analysis with no results may indicate an error, and messages related to the situation could be written to the SAS log to inform your users of the problem.

Macro program AUTHORREPORT lists titles sold by a specific author. Its one parameter, AUTHOR, specifies the name of the author for whom a report is produced. A DATA step creates a subset of the observations for the author. The macro program examines the number of observations selected and determines the next step based on that number.

AUTHORREPORT can take three actions, as follows:

- When the DATA step finds no titles for an author, the macro program writes text to the SAS log indicating this condition and that no report will be produced.
- When the DATA step finds only one title for the author, a simple PROC PRINT step executes.
- When more than one title is found, a PROC TABULATE summarizes the author's sales.

This program calls macro program AUTHORREPORT three times, once for each of the three actions described above.

AUTHORREPORT starts out by converting the AUTHOR parameter value to uppercase and quoting the result. It uppercases the value to minimize differences in the way the user specifies the value and the way that the value is stored in the data set. It quotes the value to mask any special characters or mnemonic operators that may be in the author's name.

Program 12.9

```
%macro authorreport(author);
  %* Quote the value of AUTHOR in case it contains special
     characters or mnemonic operators;
  %let author=%qupcase(&author);

  data author;
    set books.ytdsales(where=(upcase(author)="&author"));
  run;
  proc sql noprint;
    select count(booktitle)
      into :nbooks
      from author;
  quit;
  %if &nbooks=0 %then %do;
    %put ****************************************************;
    %put ERROR: Author &author not found in data set
    BOOKS.YTDSALES;
    %put No report produced.;
    %put ****************************************************;
  %end;
  %else %if &nbooks=1 %then %do;
    proc print data=author label noobs;
      title "Book Sold for Author &author";
      var booktitle datesold cost saleprice;
      format datesold monname.;
```

```
    run;
  %end;
  %if &nbooks gt 1 %then %do;
    proc tabulate data=author;
      title "Books Sold for Author &author";
      class section datesold booktitle;
      var cost saleprice;
      tables section*datesold*booktitle all='Total',
             n*f=4. (cost saleprice)*sum='Total'*f=dollar10.2;
      format datesold monname.;
    run;
  %end;
%mend authorreport;

/* This author is not in data set */
%authorreport(%str(Allan, Michael))

/* This author sold one book */
%authorreport(%str(Adams, Cynthia))

/* This author sold more than one book */
%authorreport(%str(Flores, Barbara))
```

The first call to AUTHORREPORT produces the following message in the SAS log with the line starting with ERROR: in red.

```
************************************************************
ERROR: Author ALLAN, MICHAEL not found in data set BOOKS.YTDSALES
No report produced.
************************************************************
```

Output 12.2 shows the second call to AUTHORREPORT in Program 12.9.

Output 12.2 Report produced by a second call to macro program AUTHORREPORT in Program 12.9

Book Sold for Author ADAMS, CYNTHIA

Title of Book	Date Book Sold	Wholesale Cost	Sale Price
Internet Title 628	February	$20.48	$40.95

Output 12.3 shows the third call to AUTHORREPORT in Program 12.9.

Output 12.3 Report produced by a third call to macro program AUTHORREPORT in Program 12.9

Books Sold for Author FLORES, BARBARA

Section	Date Book Sold	Title of Book	N	Wholesale Cost Total	Sale Price Total
Web Design	January	Web Design Title 150	1	$17.48	$34.95
	August	Web Design Title 150	1	$17.48	$34.95
	September	Web Design Title 150	2	$34.95	$69.90
	December	Web Design Title 150	1	$17.48	$34.95
Total			5	$87.38	$174.75

Chapter 13

A Stepwise Method for Writing Macro Programs

Introduction 336

Building a Macro Program in Four Steps 336

Applying the Four Steps to an Example 337

- Step 1: Write, test, and debug the SAS program(s) that you want the macro program to build 338
- Step 2: Remove hard-coded programming constants from the program(s) in Step 1 and replace these constants with macro variables 344
- Step 3: Create macro program(s) from the program(s) in Step 2 348
- Step 4: Refine and generalize the macro program(s) in Step 3 by adding macro language statements like %IF-%THEN and %DO groups 352
- Executing the REPORT Macro Program 356
- Enhancing the Macro Program REPORT 366

Introduction

By now you've probably thought of at least one application that you could rewrite as a macro program. You've written the DATA steps and the PROC steps and you'd like to reuse this code. You've noticed ways that it can be generalized into a macro program.

After you decide that an application is appropriate to write as a macro program, build your macro program in steps. Developing your macro program in steps of increasing complexity ensures that your macro program ends up doing exactly what you want it to do. It is also easier to debug a macro program as you develop it.

This chapter describes the steps in taking SAS programming requests and writing a macro program to handle the requests. An example illustrates the process.

Building a Macro Program in Four Steps

The four basic steps in building a macro program are

Step 1. Write, test, and debug the SAS program(s) that you want the macro program to build.
Do not use any macro variables or macro language statements in this step.

Step 2. Remove hard-coded programming constants from the program(s) in Step 1 and replace these constants with macro variables.
Hard-coded programming constants are items like the values on a WHERE statement. Use %LET statements in open code to define the macro variables. Test and debug the program(s). Use the SYMBOLGEN option to verify the results of using the macro variables.

Step 3. Create macro program(s) from the program(s) in Step 2.
Add parameters to the macro program(s) if appropriate. Most likely, it would be appropriate to make the macro variables that you define in Step 2 parameters to the macro program(s) that you write in this step. Use SAS options MPRINT and SYMBOLGEN to review the results of processing this macro program.

Step 4. Refine and generalize the macro program(s) in Step 3 by adding macro language statements like %IF-%THEN and %DO groups.
After several macro programs are tested in Step 3, write programming statements to combine the macro programs into one macro program. Test the macro programming logic. Use the SAS options MPRINT, SYMBOLGEN, and MLOGIC to verify that your macro program works correctly.

Applying the Four Steps to an Example

Suppose that you have the ongoing task of producing sales reports for the computer books department of the bookstore using the year-to-date sales data set. These reports vary, but several items in the reports are the same and the layout of the reports is the same. To save yourself coding time each time a report is requested, you decide to develop a macro program that contains the framework of the reports. You customize the basic reports through the parameters that you specify to your macro program and the macro language statements contained within the program.

Your macro program should be able to perform the following tasks:

- Analyze any or all of the sales-related variables: COST, LISTPRICE, SALEPRICE, and PROFIT. Note that PROFIT is not saved in BOOKS.YTDALES and must be computed.
- Present these analyses for specific classifications. For example, the program should be able to compute overall sales; sales by section (variable SECTION); sales by publisher (variable PUBLISHER); sales by sales associate (variable SALEINIT); sales by combinations of the classification variables.
- Present the analyses for a specific time period based on the date a book was sold (variable DATESOLD). Set the default time period of analysis to be the beginning of the year to the current date.
- Direct the results to an output destination other than the output window, and specify an ODS style when requesting this alternate destination.

Some of the reports that this macro program could produce include:

- Total of COST, LISTPRICE, SALEPRICE, and PROFIT for a specific time period.
- Total of SALEPRICE and PROFIT by section of the store for a specific time period.
- Pie chart of SALEPRICE and PROFIT by section of the store when the specific time period is a quarter or full year.
- Total PROFIT by section of the store and publisher of the books sold.
- Send any of these reports to a destination other than the output window and optionally specify an ODS style.

The rest of this chapter applies the four steps to build a macro program that can perform the tasks listed above and generate the specific reports listed above and more. The application uses PROC TABULATE and PROC GCHART.

Step 1: Write, test, and debug the SAS program(s) that you want the macro program to build

The goal of the first step is to write a few sample programs that do not contain macro language code. This gives you the basic SAS coding framework that you can generalize later as you incorporate macro facility features.

Many different reports could be requested based on the preceding list. It would not be practical to write all possible programs. Instead, write a few representative sample programs that generally encompass the basic list of program requirements.

In this application, three sample programs are written to complete this step. The three are referred to as Report A, Report B, and Report C.

- **Report A** presents overall totals for COST, LISTPRICE, SALEPRICE, and PROFIT for a specific time period, July 1, 2007–August 31, 2007.
- **Report B** presents totals and pie charts by SECTION for SALEPRICE and PROFIT for the first quarter of 2007.
- **Report C** presents totals by SECTION and PUBLISHER for COST and PROFIT for the year-to-date. Report C is sent to an RTF destination using the style GEARS that is distributed with SAS software and found in SASUSER.TMPLMST.

Program for Report A with No Macro Facility Features

Report A presents overall totals for COST, LISTPRICE, SALEPRICE, and PROFIT for July 1, 2007, through August 31, 2007.

```
*----REPORT A;
options pageno=1;
title "Sales Report";
title2 "July 1, 2007 - August 31, 2007";
data temp;
  set books.ytdsales(where=
                ('01jul2007'd le datesold le '31aug2007'd));
  profit=saleprice-cost;

  attrib profit label='Profit' format=dollar10.2;
run;

proc tabulate data=temp;
  var cost listprice saleprice profit;
  tables n*f=6.
         (cost listprice saleprice profit)*
          sum='Total'*f=dollar11.2;
  keylabel n='Titles Sold';
run;
```

Output 13.1 presents the report produced by the Report A program.

Output 13.1 Output produced by the Step 1 Report A program

```
                        Sales Report                          1
              July 1, 2007 - August 31, 2007

 -------------------------------------------------------
 |      | Wholesale |           |           |           |
 |      |   Cost    |List Price |Sale Price |  Profit   |
 |Titles|-----------+-----------+-----------+-----------|
 | Sold |   Total   |   Total   |   Total   |   Total   |
 |------+-----------+-----------+-----------+-----------|
 |   700| $15,792.78| $31,000.00| $30,380.81| $14,588.03|
 -------------------------------------------------------
```

Program for Report B with No Macro Facility Features

Report B analyzes SALEPRICE and PROFIT for first quarter 2007. It presents a tabular report and two pie charts, one for each of the two analysis variables.

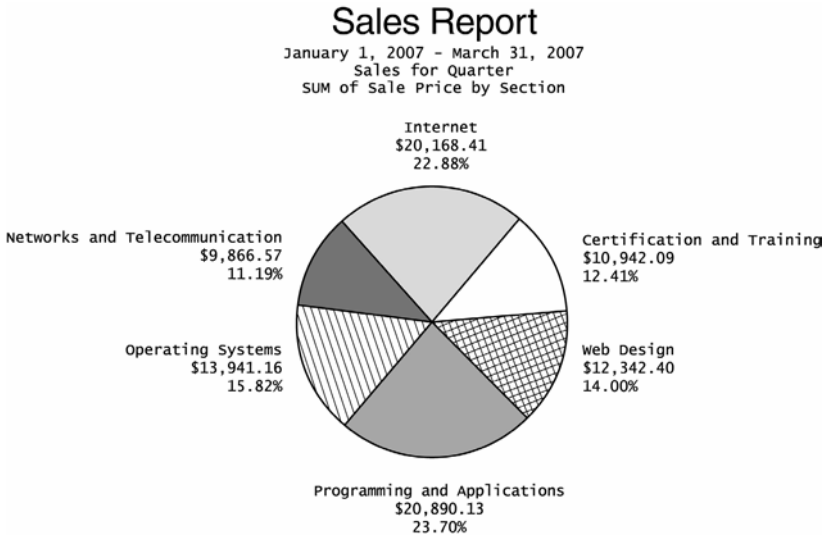
```
*----REPORT B;
options pageno=1;
title "Sales Report";
title2 "January 1, 2007 - March 31, 2007";
data temp;
  set books.ytdsales(where=
                ('01jan2007'd le datesold le '31mar2007'd));

  profit=saleprice-cost;

  attrib profit label='Profit' format=dollar10.2;
run;

proc tabulate data=temp;
  class section;
  var saleprice profit;
  tables section all,
    n*f=6. (saleprice profit)*sum='Total'*f=dollar11.2 /
    rts=30;
  keylabel all='Total Sales'
           n='Titles Sold';
run;

proc gchart data=temp;
  title3 "Sales for Quarter";
  pie section / type=sum sumvar=saleprice
                coutline=black percent=outside;
  run;
  pie section / type=sum sumvar=profit
                coutline=black percent=outside;
  run;
quit;
```

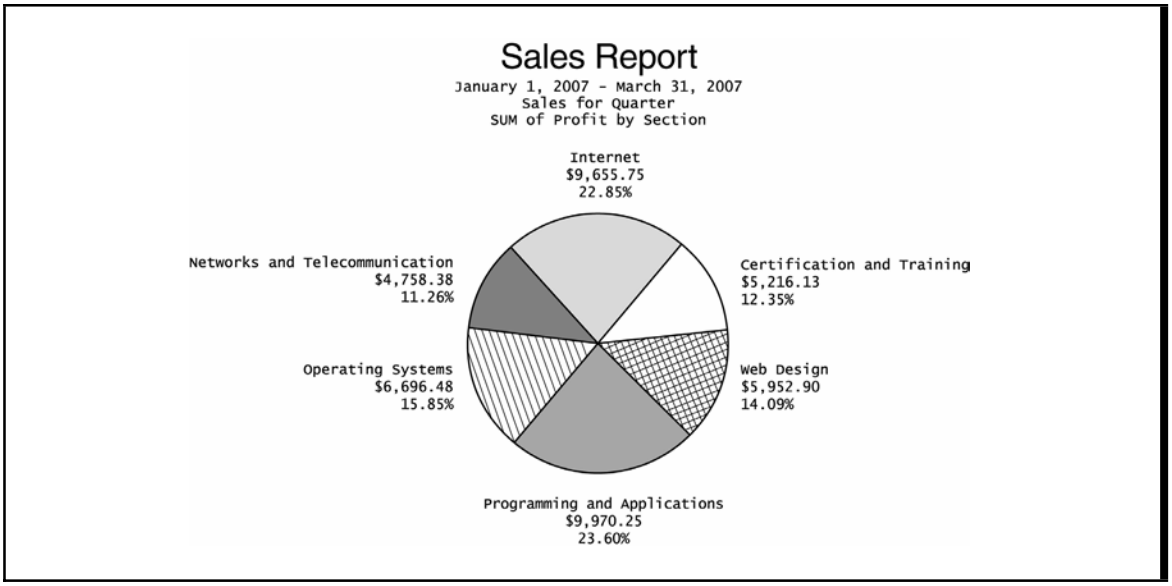
Output 13.2 presents the output produced by the Report B program.

Output 13.2 Output produced by the Step 1 Report B program

Sales Report 1
January 1, 2007 - March 31, 2007

	Titles Sold	Sale Price Total	Profit Total
Section			
Certification and Training	253	$10,942.09	$5,216.13
Internet	477	$20,168.41	$9,655.75
Networks and Telecommunication	229	$9,866.57	$4,758.38
Operating Systems	325	$13,941.16	$6,696.48
Programming and Applications	478	$20,890.13	$9,970.25
Web Design	280	$12,342.40	$5,952.90
Total Sales	2042	$88,150.76	$42,249.87





Program for Report C with No Macro Facility Features

Report C summarizes COST and PROFIT from the beginning of the year to the current date by section and publisher. Assume that the current date for the example is November 24, 2007. The output for this report is sent to the RTF output destination and uses style GEARS found in SASUSER.TMPLMST, which contains SAS supplied templates.

```
*----REPORT C;
ods listing close;
ods rtf style=gears;

title "Sales Report";
title2 "January 1, 2007 - November 24, 2007";
data temp;
  set books.ytdsales(where=
             ('01jan2007'd le datesold le '24nov2007'd));

  profit=saleprice-cost;
  attrib profit label='Profit' format=dollar10.2;

run;
```

```
proc tabulate data=temp;
  class section publisher;
  var cost profit;
  tables section*(publisher all) all,
    n*f=6. (cost profit)*sum*f=dollar11.2 / rts=30;
  keylabel all='Total Sales'
           n='Titles Sold';
run;
ods rtf close;
ods listing;
```

Output 13.3 presents the output produced by the Report C program. The report covers six pages. Output 13.3 shows the first and the sixth page of the report. The style selected prints the output in color. When you submit the program, your RTF output will display in the colors of the style. This book presents a grayscale copy of the output.

Output 13.3 Output produced by the Step 1 Report C program

Sales Report
January 1, 2007 - November 24, 2007

			Wholesale Cost	Profit
		Titles Sold	Sum	Sum
Section	Publisher			
Certification and Training	AMZ Publishers	56	$1,311.38	$1,243.47
	Bookstore Brand Titles	53	$1,208.56	$1,167.01
	Doe&Lee Ltd.	56	$1,231.20	$1,198.34
	Eversons Books	34	$815.22	$698.64
	IT Training Texts	59	$1,316.12	$1,229.62
	Mainst Media	102	$2,394.43	$2,070.97
	Nifty New Books	58	$1,365.23	$1,197.32
	Northern Associates Titles	35	$821.41	$747.99
	Popular Names Publishers	54	$1,177.01	$1,025.90
	Professional House Titles	42	$837.45	$834.16
	Technology Smith	55	$1,241.71	$1,193.17
	Wide-World Titles	53	$1,204.95	$1,100.27
	Total Sales	657	$14,924.67	$13,706.83

(*continued on the next page*)

Sales Report January 1, 2007 - November 24, 2007				
		Titles Sold	Wholesale Cost Sum	Profit Sum
Web Design	Publisher			
	AMZ Publishers	58	$1,408.43	$1,333.22
	Bookstore Brand Titles	50	$1,156.13	$1,022.96
	Doe&Lee Ltd.	73	$1,660.16	$1,566.98
	Eversons Books	90	$2,046.12	$1,863.22
	IT Training Texts	49	$1,083.14	$1,008.83
	Mainst Media	58	$1,240.74	$1,172.22
	Nifty New Books	67	$1,522.82	$1,430.33
	Northern Associates Titles	56	$1,277.66	$1,047.80
	Popular Names Publishers	50	$1,161.75	$1,109.11
	Professional House Titles	84	$1,925.74	$1,730.87
	Technology Smith	64	$1,429.16	$1,323.17
	Wide-World Titles	77	$1,824.07	$1,788.82
	Total Sales	776	$17,735.92	$16,397.52
Total Sales		5608	$126,228.60	$116,487.40

After running these three programs and verifying that they display the information required, move on to Step 2.

Step 2: Remove hard-coded programming constants from the program(s) in Step 1 and replace these constants with macro variables

When you review the three programs created in Step 1, some patterns emerge:

- Observations are selected within a certain range of dates. This range is specified in the WHERE clause in the DATA step and in the titles.
- Analysis variables are selected from a defined set of variables.
- Classification variables are selected from a defined set of variables.

The values in the preceding list are hard-coded programming constants in the three programs from Step 1. Macro variables can be created in open code to hold these values.

Program for Report A with Step 2 Modifications

A revised Report A program follows that includes open code macro language statements. The %LET statements and macro variable references are in bold. The macro values TITLESTART and TITLESTOP are assigned the formatted values of the reporting period.

```
*----REPORT A;
%let repyear=2007;
%let start=01jul&repyear;
%let stop=31aug&repyear;
%let vars=cost listprice saleprice profit;

%let titlestart=%sysfunc(putn("&start"d,worddate.));
%let titlestop=%sysfunc(putn("&stop"d,worddate.));

options pageno=1 symbolgen;
title "Sales Report";
title2 "&titlestart - &titlestop";
data temp;
  set books.ytdsales(where=
      ("&start"d le datesold le "&stop"d));

  profit=saleprice-cost;

  attrib profit label='Profit' format=dollar10.2;
run;

proc tabulate data=temp;
  var &vars;
  tables n*f=6.
         (&vars)*
          sum='Total'*f=dollar11.2;
  keylabel n='Titles Sold';
run;
```

Program for Report B with Step 2 Modifications

Report B is modified to define macro variables in open code. Some of the changes that were made to this program were made to the Report A program.

```
*----Report B;
%let repyear=2007;
%let start=01jan&repyear;
%let stop=31mar&repyear;

%let classvar=section;
%let vars=saleprice profit;

%let titlestart=%sysfunc(putn("&start"d,worddate.));
%let titlestop=%sysfunc(putn("&stop"d,worddate.));

options pageno=1 symbolgen;

title "Sales Report";
title2 "&titlestart through &titlestop";
data temp;
  set books.ytdsales(where=
      ("&start"d le datesold le "&stop"d));

  profit=saleprice-cost;

  attrib profit label='Profit' format=dollar10.2;
run;

proc tabulate data=temp;
  class &classvar;
  var &vars;
  tables section all,
    n*f=6. (&vars)*sum='Total'*f=dollar11.2 /
    rts=30;
  keylabel all='Total Sales'
           n='Titles Sold';
run;
```

```
proc gchart data=temp;
  title3 "Sales for Quarter";
  pie &classvar / type=sum sumvar=%scan(&vars,1)
                  coutline=black percent=outside;
  run;
  pie &classvar / type=sum sumvar=%scan(&vars,2)
                  coutline=black percent=outside;
  run;
quit;
```

Program for Report C with Step 2 Modifications

The program for Report C is modified with the creation of macro variables in open code. The features added to this program are similar to and include some of those added to the programs for Report A and Report B.

```
*----REPORT C;
%let repyear=2007;
%let start=01jan&repyear;
%let stop=&sysdate;

%let classvar=section publisher;
%let vars=cost profit;

%let outputdest=rtf;
%let outputstyle=gears;

%let titlestart=%sysfunc(putn("&start"d,worddate.));
%let titlestop=%sysfunc(putn("&stop"d,worddate.));

options symbolgen;

ods listing close;
ods &outputdest style=&outputstyle;

title "Sales Report";
title2 "&titlestart - &titlestop";
data temp;
  set books.ytdsales(where=
     ("&start"d le datesold le "&stop"d));

  profit=saleprice-cost;
  attrib profit label='Profit' format=dollar10.2;

run;
```

```
proc tabulate data=temp;
  class &classvar;
  var &vars;
  tables %scan(&classvar,1)*( %scan(&classvar,2) all) all,
    n*f=6. (&vars)*sum*f=dollar11.2 / rts=30;
  keylabel all='Total Sales'
           n='Titles Sold';
run;
ods &outputdest close;
ods listing;
```

Step 3: Create macro program(s) from the program(s) in Step 2

In Step 2, similar changes were made to each of the three programs:

- Macro variables were defined for the range in dates that were selected from the data set.
- Macro variables were defined to hold the classification variables and analysis variables.

It might be tempting to jump to writing %DO blocks and conditional processing statements, but complete Step 3 first. In Step 3, define macro programs that use parameters. The parameters to the macro programs will usually be the macro variables that were defined in Step 2. By not including macro language statements in these macro program definitions, you'll be sure that the parameters you define execute correctly.

Use the SYMBOLGEN and MPRINT options to verify that your programming changes do what you intend.

Program for Report A with Step 3 Modifications

The program for Report A is converted to a macro program. It has four keyword parameters, the same as the first four macro variables defined in open code in Step 2. The macro program assigns default values to three parameters: the start date, the stop date, and the analysis variables. The DATA and PROC steps in this program are the same as those in the Report A program in Step 2.

```
*----REPORT A;
options symbolgen mprint;

%macro reporta(repyear=,start=01JAN,stop=31DEC,
               vars=cost listprice saleprice profit);
```

```
  %let start=&start&repyear;
  %let stop=&stop&repyear;

  %let titlestart=%sysfunc(putn("&start"d,worddate.));
  %let titlestop=%sysfunc(putn("&stop"d,worddate.));

  title "Sales Report";
  title2 "&titlestart - &titlestop";
  data temp;
    set books.ytdsales(where=
        ("&start"d le datesold le "&stop"d));

    profit=saleprice-cost;

    attrib profit label='Profit' format=dollar10.2;
  run;

  proc tabulate data=temp;
    var &vars;
    tables n*f=6.
           (&vars)*
            sum='Total'*f=dollar11.2;
    keylabel n='Titles Sold';
  run;
%mend reporta;
```

The code to call REPORTA becomes

```
%reporta(repyear=2007,start=01jul,stop=31aug)
```

The start and stop dates for the reporting period are different than the default dates of January 1 and December 31. Therefore, you need to specify these two parameters. The analysis variables are the same as the default set of variables that are listed in the macro program definition for REPORTA. Therefore, the call to macro program REPORTA does not specify the VARS parameter.

Program for Report B with Step 3 Modifications

The program for Report B in Step 2 is converted into the following macro program. This macro program defines five keyword parameters. Two parameters, the start date and the stop date, are defined with default values. The DATA and PROC steps in this program are the same as those in the Report B program in Step 2. Note that the titles indicate that the processing is done for a quarter. Therefore, to be accurate, specify the START= and STOP= values to correspond to a quarter.

```
options symbolgen mprint;

%macro reportb(repyear=,start=01JAN,stop=31DEC,
               classvar=,vars=);

  options pageno=1;

  %let start=&start&repyear;
  %let stop=&stop&repyear;

  %let titlestart=%sysfunc(putn("&start"d,worddate.));
  %let titlestop=%sysfunc(putn("&stop"d,worddate.));

  title "Sales Report";
  title2 "&titlestart - &titlestop";
  data temp;
    set books.ytdsales(where=
        ("&start"d le datesold le "&stop"d));

    profit=saleprice-cost;

    attrib profit label='Profit' format=dollar10.2;
  run;

  proc tabulate data=temp;
    title3 "Sales for Quarter";
    class &classvar;
    var &vars;
    tables section all,
      n*f=6. (&vars)*sum='Total'*f=dollar11.2 /
      rts=30;
    keylabel all='Total Sales'
             n='Titles Sold';
  run;

  proc gchart data=temp;
    title3 "Sales for Quarter";
    pie &classvar / type=sum sumvar=%scan(&vars,1)
                    coutline=black percent=outside;
    run;
    pie &classvar / type=sum sumvar=%scan(&vars,2)
                    coutline=black percent=outside;
    run;
  quit;

%mend reportb;
```

The call to REPORTB is written as follows:

```
%reportb(repyear=2007,stop=31Mar,classvar=section,
         vars=saleprice profit)
```

The start date for the call to REPORTB is the same as the default value of January 1. Therefore, the start date does not have to be specified in the call to REPORTB. The stop date that is required to produce Report B is March 31. Since the default stop date is December 31, the value `31Mar` must be specified as the stop date parameter value. The information in the report is summarized by the classification variable, SECTION. Two analysis variables are specified: SALEPRICE and PROFIT.

Program for Report C with Step 3 Modifications

Next, the program for Report C in Step 2 is converted into a macro program. This macro program defines seven keyword parameters. Two parameters, the start date and the stop date, are defined with default values. The DATA and PROC steps in this program are the same as those in the Report C program in Step 2.

This macro program differs from macro programs REPORTA and REPORTB because it has two parameters that control the destination of the output. These previous two macro programs sent output to whatever the current destination in the SAS session was when they were called.

A problem with the following macro program is that the ODS statement with the STYLE= option always executes. The STYLE= option is not valid when specifying the LISTING destination. When you direct the output to the LISTING destination, the program generates an error. This problem is fixed in Step 4.

```
options symbolgen mprint;

%macro reportc(repyear=,start=01JAN,stop=31DEC,
               classvar=,vars=,
               outputdest=,style=);

  options pageno=1;

  %let start=&start&repyear;
  %let stop=&stop&repyear;

  %let titlestart=%sysfunc(putn("&start"d,worddate.));
  %let titlestop=%sysfunc(putn("&stop"d,worddate.));

  ods listing close;
  ods &outputdest style=&outputstyle;
  title "Sales Report";
  title2 "&titlestart - &titlestop";
```

```
  data temp;
    set books.ytdsales(where=
       ("&start"d le datesold le "&stop"d));

    profit=saleprice-cost;

    attrib profit label='Profit' format=dollar10.2;
  run;

  proc tabulate data=temp;
    class &classvar;
    var &vars;
    tables %scan(&classvar,1)*( %scan(&classvar,2) all) all,
      n*f=6. (&vars)*sum*f=dollar11.2 / rts=30;
    keylabel all='Total Sales'
             n='Titles Sold';
  run;
  ods &outputdest close;
  ods listing;
%mend reportc;
```

The call to REPORTC is specified as follows:

```
%reportc(repyear=2007,stop=24NOV,classvar=section publisher,
vars=cost profit,outputdest=rtf,style=gears)
```

The start date for the call to REPORTC is the default value of January 1. The stop date required to produce REPORTC is the current date of November 24 and must be specified since the default stop date is December 31. The information in the report is summarized by two classification variables, SECTION and PUBLISHER. Two analysis variables are specified, COST and PROFIT. The program directs the output to the RTF destination using the style GEARS.

Step 4: Refine and generalize the macro program(s) in Step 3 by adding macro language statements like %IF-%THEN and %DO groups

The goal in Step 4 for the example application is to consolidate the three macro programs into one. The main similarity among the three programs is that they have most of the same parameters. Macro language statements are required to handle the following differences and to further generalize the programs:

- No classification variable is specified in Report A. One classification variable is specified in Report B. Two classification variables are specified in Report C.
- Report A uses all the analysis variables. Reports B and C use some of the analysis variables.
- Report B is executed at the end of a quarter. Therefore, the third title is required for the pie charts.
- The number of PIE statements in Report B is equal to the number of analysis variables.
- Report C is sent to a destination other than the LISTING destination. The STYLE option on the ODS statement when LISTING is the destination causes an error and programming must eliminate this problem.

One enhancement that could be added to the macro program is to compute defaults for the report year and the stop date of the reports. Write the macro program so that when no report year is entered, use the current year. If stop date is specified as a null value, use the current date as the stop date for the report. If a default value has been specified for the stop date in the macro program definition, and the parameter is not included in the call to the program, the stop date will be the default value assigned to the parameter (31DEC).

The consolidated macro program incorporates conditional processing and iterative processing. One way to write this macro program follows. The changes are in bold. Comments are added to the macro program to describe the processing of the macro program.

```
options mprint mlogic symbolgen;

%macro report(repyear=,start=01JAN,stop=31DEC,
              classvar=,vars=cost listprice saleprice profit,
              outputdest=listing,style=);

  options pageno=1;

  %*----Check if a value was specified for report year.
        If no value specified,use current year;
  %if &repyear= %then %let repyear=
                 %sysfunc(year(%sysfunc(today())));
  %*----Check if stop date specified. If null, use
        current date as stop date;
  %if &stop= %then %let stop=%substr(&sysdate,1,5);

  %let start=&start&repyear;
  %let stop=&stop&repyear;
```

```
%let titlestart=%sysfunc(putn("&start"d,worddate.));
%let titlestop=%sysfunc(putn("&stop"d,worddate.));

%*----Check the output destination and style parameters;
%*----Close LISTING, open alternate destination if
      specified;
%*----Add STYLE if specified for the alternate
      destination;
%if %upcase(&outputdest) ne LISTING %then %do;
  ods listing close;
  ods &outputdest
  %if &style ne %then %do;
    style=&style
  %end;
  ;
%end;

title "Sales Report";
title2 "&titlestart - &titlestop";
data temp;
  set books.ytdsales(where=
     ("&start"d le datesold le "&stop"d));

  profit=saleprice-cost;

  attrib profit label='Profit' format=dollar10.2;
run;

proc tabulate data=temp;
  %*----Only submit a CLASS statement if there is a
       classification variable;
  %if &classvar ne %then %do;
     class &classvar;
  %end;
  var &vars;
  tables
    %if &classvar ne %then %do;
      %*---Determine leftmost row dimension variable;
      %let mainclas=%scan(&classvar,1);
      &mainclas
      %if %length(&mainclas) < %length(&classvar) %then %do;
        %*----If more than one classification variable, nest
              remaining classification variables under the
              first;
        %*----Use the substring function to extract
              classification variables after the first;
```

```
        %let pos2=%index(&classvar,%scan(&classvar,2));
        %*----Add the rest of the classification vars;
        * ( %substr(&classvar,&pos2) all)

      %end;
      all,
    %end;
    n*f=6. (&vars)*sum*f=dollar11.2;
    keylabel all='Total Sales'
             n='Titles Sold';
run;

%*----Check if date range is for a quarter or year;
%let strtmdy=%upcase(%substr(&start,1,5));
%let stopmdy=%upcase(%substr(&stop,1,5));
%if (&strtmdy=01JAN and &stopmdy=31MAR) or
    (&strtmdy=01APR and &stopmdy=30JUN) or
    (&strtmdy=01JUL and &stopmdy=30SEP) or
    (&strtmdy=01OCT and &stopmdy=31DEC) or
    (&strtmdy=01JAN and &stopmdy=31DEC) %then %do;

  %*----Special titles for Quarter and for Year;
  %if not (&strtmdy eq 01JAN and &stopmdy eq 31DEC)
        %then %do;
     title3 "Sales for Quarter";
  %end;
  %else %do;
    title3 "&repyear Annual Sales";
  %end;

  proc gchart data=temp;
    %*----For each analysis variable, do a pie chart;
    %let setchrt=1;
    %let chrtvar=%scan(&vars,1);
    %do %while (&chrtvar ne );
      pie &classvar / type=sum sumvar=&chrtvar
                      coutline=black percent=outside;
      run;

      %let setchrt=%eval(&setchrt+1);
      %let chrtvar=%scan(&vars,&setchrt);
    %end;
  quit;
%end;
```

```
  %*-----Close alternate destination if specified;
  %if %upcase(&outputdest) ne LISTING %then %do;
    ods &outputdest close;
    ods listing;
  %end;

%mend report;
```

In Step 4, the SYMBOLGEN, MPRINT, and MLOGIC options can verify that your macro program works correctly. After you thoroughly check your macro program, turn these options off to save computing time.

Executing the REPORT Macro Program

Many types of reports can now be generated by the REPORT macro program, including Reports A, B, and C.

Obtaining the Contents of Report A Using the REPORT Macro Program

The first request to sum sales information for July and August 2007 is as follows:

```
%report(repyear=2007,start=01jul,stop=31aug)
```

The SAS log for the above submission of the call to %REPORT follows. The SAS code that macro program REPORT submits is in bold. Options MLOGIC and MPRINT are in effect.

```
450  %report(repyear=2007,start=01jul,stop=31aug)
MLOGIC(REPORT):  Beginning execution.
MLOGIC(REPORT):  Parameter REPYEAR has value 2007
MLOGIC(REPORT):  Parameter START has value 01jul
MLOGIC(REPORT):  Parameter STOP has value 31aug
MLOGIC(REPORT):  Parameter CLASSVAR has value
MLOGIC(REPORT):  Parameter VARS has value cost listprice
      saleprice profit
MLOGIC(REPORT):  Parameter OUTPUTDEST has value listing
MLOGIC(REPORT):  Parameter STYLE has value
MPRINT(REPORT):   options pageno=1;
MLOGIC(REPORT):  %IF condition &repyear= is FALSE
MLOGIC(REPORT):  %IF condition &stop= is FALSE
MLOGIC(REPORT):  %LET (variable name is START)
MLOGIC(REPORT):  %LET (variable name is STOP)
MLOGIC(REPORT):  %LET (variable name is TITLESTART)
MLOGIC(REPORT):  %LET (variable name is TITLESTOP)
```

```
MLOGIC(REPORT):  %IF condition %upcase(&outputdest) ne LISTING
      is FALSE
MPRINT(REPORT):   title "Sales Report";
MPRINT(REPORT):   title2 "July 1, 2007 - August 31, 2007";
MPRINT(REPORT):   data temp;
MPRINT(REPORT):   set books.ytdsales(where= ("01jul2007"d le
datesold le "31aug2007"d));
MPRINT(REPORT):   profit=saleprice-cost;
MPRINT(REPORT):   attrib profit label='Profit'
format=dollar10.2;
MPRINT(REPORT):   run;

NOTE: There were 700 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01JUL2007'D and datesold<='31AUG2007'D);
NOTE: The data set WORK.TEMP has 700 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

MPRINT(REPORT):   proc tabulate data=temp;
MLOGIC(REPORT):  %IF condition &classvar ne is FALSE
MPRINT(REPORT):   var cost listprice saleprice profit;
MLOGIC(REPORT):  %IF condition &classvar ne is FALSE
MPRINT(REPORT):   tables n*f=6. (cost listprice saleprice
profit)*sum*f=dollar11.2;
MPRINT(REPORT):   keylabel all='Total Sales' n='Titles Sold';
MPRINT(REPORT):   run;

NOTE: There were 700 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.06 seconds
      cpu time            0.03 seconds

MLOGIC(REPORT):  %LET (variable name is STRTMDY)
MLOGIC(REPORT):  %LET (variable name is STOPMDY)
MLOGIC(REPORT):  %IF condition (&strtmdy=01JAN and
      &stopmdy=31MAR) or       (&strtmdy=01APR and
      &stopmdy=30JUN) or       (&strtmdy=01JUL and
      &stopmdy=30SEP) or       (&strtmdy=01OCT and
      &stopmdy=31DEC) or       (&strtmdy=01JAN and
      &stopmdy=31DEC) is FALSE
```

```
MLOGIC(REPORT):  %IF condition %upcase(&outputdest) ne LISTING
      is FALSE
MLOGIC(REPORT):  Ending execution.
```

Output 13.4 presents the output for the first call to REPORT. This output is identical to that in Output 13.1.

Output 13.4 Output for the first call to macro program REPORT, which generates the information that was specified for Report A

```
                        Sales Report                          1
              July 1, 2007 - August 31, 2007

 ---------------------------------------------------------
 |      | Wholesale |           |           |           |
 |      |   Cost    |List Price |Sale Price |  Profit   |
 |Titles|-----------+-----------+-----------+-----------|
 | Sold |    Sum    |    Sum    |    Sum    |    Sum    |
 |------+-----------+-----------+-----------+-----------|
 |   700| $15,792.78| $31,000.00| $30,380.81| $14,588.03|
 ---------------------------------------------------------
```

Obtaining the Contents of Report B Using the REPORT Macro Program

The second request to REPORT should generate statistics for sale price and profit by section for first quarter 2007. Since the reporting time period is a quarter, pie charts are also produced.

```
%report(repyear=2007,stop=31Mar,classvar=section,
         vars=saleprice profit)
```

The SAS log for the above submission of the call to %REPORT follows. The SAS code that macro program REPORT submits is in bold. Options MLOGIC and MPRINT are in effect.

```
451  %report(repyear=2007,stop=31Mar,classvar=section,
MLOGIC(REPORT):  Beginning execution.
452           vars=saleprice profit)
MLOGIC(REPORT):  Parameter REPYEAR has value 2007
MLOGIC(REPORT):  Parameter STOP has value 31Mar
MLOGIC(REPORT):  Parameter CLASSVAR has value section
MLOGIC(REPORT):  Parameter VARS has value saleprice profit
```

```
MLOGIC(REPORT):  Parameter START has value 01JAN
MLOGIC(REPORT):  Parameter OUTPUTDEST has value listing
MLOGIC(REPORT):  Parameter STYLE has value
MPRINT(REPORT):   options pageno=1;
MLOGIC(REPORT):  %IF condition &repyear= is FALSE
MLOGIC(REPORT):  %IF condition &stop= is FALSE
MLOGIC(REPORT):  %LET (variable name is START)
MLOGIC(REPORT):  %LET (variable name is STOP)
MLOGIC(REPORT):  %LET (variable name is TITLESTART)
MLOGIC(REPORT):  %LET (variable name is TITLESTOP)
MLOGIC(REPORT):  %IF condition %upcase(&outputdest) ne LISTING
      is FALSE
MPRINT(REPORT):   title "Sales Report";
MPRINT(REPORT):   title2 "January 1, 2007 - March 31, 2007";
MPRINT(REPORT):   data temp;
MPRINT(REPORT):   set books.ytdsales(where= ("01JAN2007"d le
datesold le "31Mar2007"d));
MPRINT(REPORT):   profit=saleprice-cost;
MPRINT(REPORT):   attrib profit label='Profit'
format=dollar10.2;
MPRINT(REPORT):   run;

NOTE: There were 2042 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2007'D and datesold<='31MAR2007'D);
NOTE: The data set WORK.TEMP has 2042 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

MPRINT(REPORT):   proc tabulate data=temp;
MLOGIC(REPORT):  %IF condition &classvar ne is TRUE
MPRINT(REPORT):   class section;
MPRINT(REPORT):   var saleprice profit;
MLOGIC(REPORT):  %IF condition &classvar ne is TRUE
MLOGIC(REPORT):  %LET (variable name is MAINCLAS)
MLOGIC(REPORT):  %IF condition %length(&mainclas) <
      %length(&classvar) is FALSE
MPRINT(REPORT):   tables section all, n*f=6. (saleprice
profit)*sum*f=dollar11.2;
MPRINT(REPORT):   keylabel all='Total Sales' n='Titles Sold';
MPRINT(REPORT):   run;
```

```
NOTE: There were 2042 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds

MLOGIC(REPORT):  %LET (variable name is STRTMDY)
MLOGIC(REPORT):  %LET (variable name is STOPMDY)
MLOGIC(REPORT):  %IF condition (&strtmdy=01JAN and
      &stopmdy=31MAR) or       (&strtmdy=01APR and
      &stopmdy=30JUN) or       (&strtmdy=01JUL and
      &stopmdy=30SEP) or       (&strtmdy=01OCT and
      &stopmdy=31DEC) or       (&strtmdy=01JAN and
      &stopmdy=31DEC) is TRUE
MLOGIC(REPORT):  %IF condition not (&strtmdy eq 01JAN and
      &stopmdy eq 31DEC) is TRUE
MPRINT(REPORT):   title3 "Sales for Quarter";
MPRINT(REPORT):   proc gchart data=temp;
MLOGIC(REPORT):  %LET (variable name is SETCHRT)
MLOGIC(REPORT):  %LET (variable name is CHRTVAR)
MLOGIC(REPORT):  %DO %WHILE(&chrtvar ne) loop beginning;
      condition is TRUE.
MPRINT(REPORT):   pie section / type=sum sumvar=saleprice
coutline=black percent=outside;
MPRINT(REPORT):   run;

MLOGIC(REPORT):  %LET (variable name is SETCHRT)
MLOGIC(REPORT):  %LET (variable name is CHRTVAR)
MLOGIC(REPORT):  %DO %WHILE(&chrtvar ne) condition is TRUE;
      loop will  iterate again.
MPRINT(REPORT):   pie section / type=sum sumvar=profit
coutline=black percent=outside;
MPRINT(REPORT):   run;
MLOGIC(REPORT):  %LET (variable name is SETCHRT)
MLOGIC(REPORT):  %LET (variable name is CHRTVAR)
MLOGIC(REPORT):  %DO %WHILE() condition is FALSE; loop will not
      iterate again.
MPRINT(REPORT):   quit;

NOTE: There were 2042 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE GCHART used (Total process time):
      real time           1.39 seconds
      cpu time            0.15 seconds
```

```
MLOGIC(REPORT):  %IF condition %upcase(&outputdest) ne LISTING
      is FALSE
MLOGIC(REPORT):  Ending execution.
```

Output 13.5 presents the output for the second call to REPORT. This output is identical to that in Output 13.2.

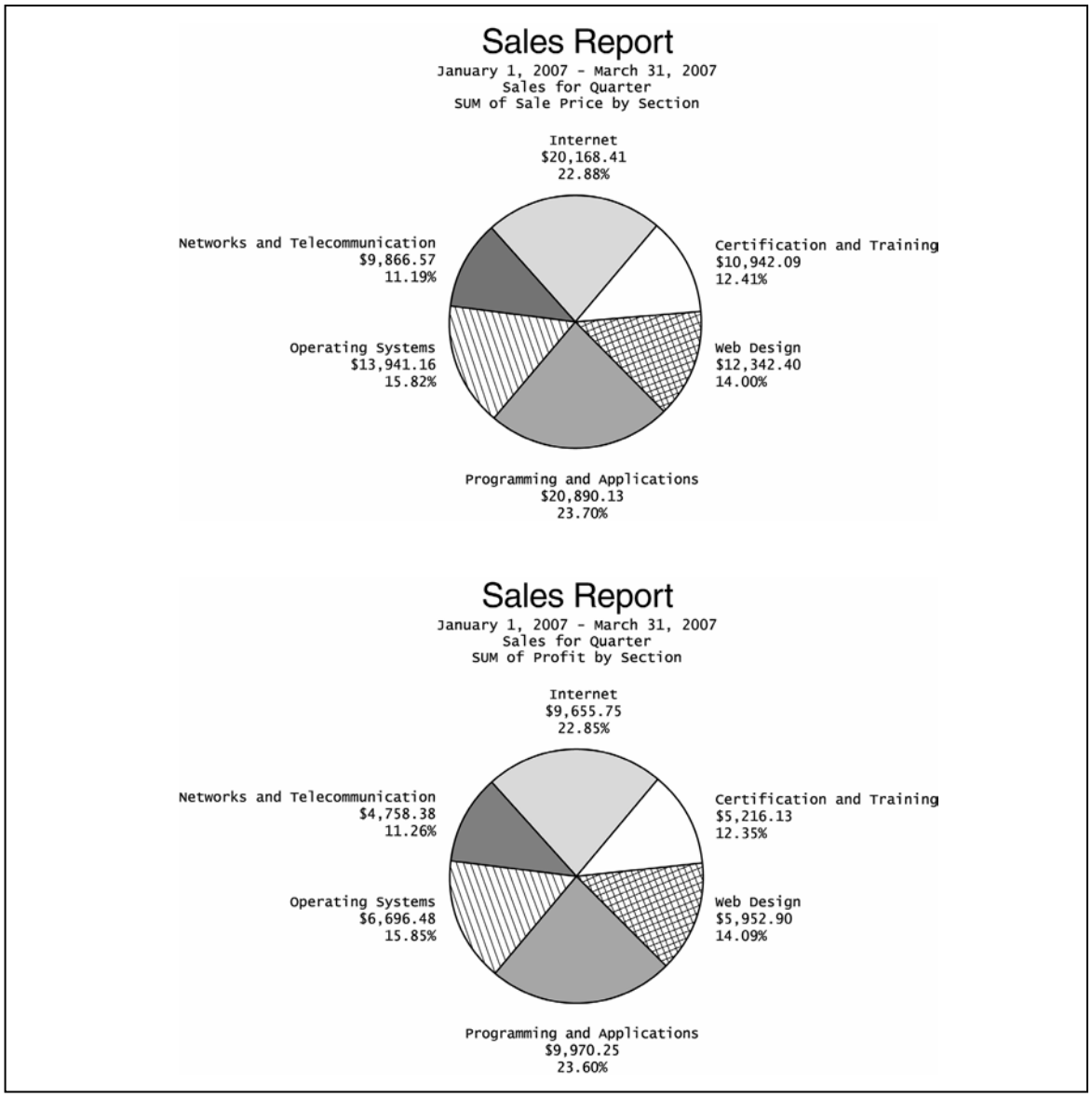
Output 13.5 Output for the second call to macro program REPORT, which generates the information that was specified for Report B

Sales Report 1

January 1, 2007 - March 31, 2007

		Sale Price	Profit
	Titles Sold	Sum	Sum
Section			
Certification and Training	253	$10,942.09	$5,216.13
Internet	477	$20,168.41	$9,655.75
Networks and Telecommunication	229	$9,866.57	$4,758.38
Operating Systems	325	$13,941.16	$6,696.48
Programming and Applications	478	$20,890.13	$9,970.25
Web Design	280	$12,342.40	$5,952.90
Total Sales	2042	$88,150.76	$42,249.87

Output 13.5 (*continued*)



Obtaining the Contents of Report C Using the REPORT Macro Program

The third report summarizes COST and PROFIT from the beginning of the current year through the current date. The analysis is done by SECTION and PUBLISHER. Assume that the program was submitted on November 24, 2007. The program sends output to the RTF destination using the style GEARS. The third call to REPORT follows.

```
%report(stop=,
        classvar=section publisher,
        vars=cost profit,
        outputdest=rtf,style=gears)
```

The SAS log for the above submission of the call to %REPORT follows. The SAS code that macro program REPORT submits is in bold. Options MLOGIC and MPRINT are in effect.

```
453  %report(stop=,
MLOGIC(REPORT):  Beginning execution.
454          classvar=section publisher,
455          vars=cost profit,
456          outputdest=rtf,style=gears)
MLOGIC(REPORT):  Parameter STOP has value
MLOGIC(REPORT):  Parameter CLASSVAR has value section publisher
MLOGIC(REPORT):  Parameter VARS has value cost profit
MLOGIC(REPORT):  Parameter OUTPUTDEST has value rtf
MLOGIC(REPORT):  Parameter STYLE has value gears
MLOGIC(REPORT):  Parameter REPYEAR has value
MLOGIC(REPORT):  Parameter START has value 01JAN
MPRINT(REPORT):   options pageno=1;
MLOGIC(REPORT):  %IF condition &repyear= is TRUE
MLOGIC(REPORT):  %LET (variable name is REPYEAR)
MLOGIC(REPORT):  %IF condition &stop= is TRUE
MLOGIC(REPORT):  %LET (variable name is STOP)
MLOGIC(REPORT):  %LET (variable name is START)
MLOGIC(REPORT):  %LET (variable name is STOP)
MLOGIC(REPORT):  %LET (variable name is TITLESTART)
MLOGIC(REPORT):  %LET (variable name is TITLESTOP)
MLOGIC(REPORT):  %IF condition %upcase(&outputdest) ne LISTING
      is TRUE
MPRINT(REPORT):   ods listing close;
MLOGIC(REPORT):  %IF condition &style ne is TRUE
MPRINT(REPORT):   ods rtf style=gears ;
NOTE: Writing RTF Body file: sasrtf.rtf
MPRINT(REPORT):   title "Sales Report";
MPRINT(REPORT):   title2 "January 1, 2007 - November 24, 2007";
MPRINT(REPORT):   data temp;
```

```
MPRINT(REPORT):   set books.ytdsales(where= ("01JAN2007"d le
datesold le "24NOV2007"d));
MPRINT(REPORT):   profit=saleprice-cost;
MPRINT(REPORT):   attrib profit label='Profit'
format=dollar10.2;
MPRINT(REPORT):   run;

NOTE: There were 5608 observations read from the data set
      BOOKS.YTDSALES.
      WHERE (datesold>='01JAN2007'D and datesold<='24NOV2007'D);
NOTE: The data set WORK.TEMP has 5608 observations and 11
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds


MPRINT(REPORT):   proc tabulate data=temp;
MLOGIC(REPORT):  %IF condition &classvar ne is TRUE
MPRINT(REPORT):   class section publisher;
MPRINT(REPORT):   var cost profit;
MLOGIC(REPORT):  %IF condition &classvar ne is TRUE
MLOGIC(REPORT):  %LET (variable name is MAINCLAS)
MLOGIC(REPORT):  %IF condition %length(&mainclas) <
      %length(&classvar) is TRUE
MLOGIC(REPORT):  %LET (variable name is POS2)
MPRINT(REPORT):   tables section * ( publisher all) all, n*f=6.
(cost profit)*sum*f=dollar11.2;
MPRINT(REPORT):   keylabel all='Total Sales' n='Titles Sold';
MPRINT(REPORT):   run;

NOTE: There were 5608 observations read from the data set
      WORK.TEMP.
NOTE: PROCEDURE TABULATE used (Total process time):
      real time           0.07 seconds
      cpu time            0.04 seconds


MLOGIC(REPORT):  %LET (variable name is STRTMDY)
MLOGIC(REPORT):  %LET (variable name is STOPMDY)
MLOGIC(REPORT):  %IF condition (&strtmdy=01JAN and
      &stopmdy=31MAR) or       (&strtmdy=01APR and
      &stopmdy=30JUN) or       (&strtmdy=01JUL and
      &stopmdy=30SEP) or       (&strtmdy=01OCT and
      &stopmdy=31DEC) or       (&strtmdy=01JAN and
      &stopmdy=31DEC) is FALSE
```

```
MLOGIC(REPORT):  %IF condition %upcase(&outputdest) ne LISTING
      is TRUE
MPRINT(REPORT):   ods rtf close;
MPRINT(REPORT):   ods listing;
MLOGIC(REPORT):  Ending execution.
```

Output 13.6 presents the output for the third call to REPORT. This output is identical to that in Output 13.3.

Output 13.6 Output for the third call to macro program REPORT, which generates the information that was specified for Report C

Sales Report
January 1, 2007 - November 24, 2007

		Titles Sold	Wholesale Cost	Profit
			Sum	Sum
Section	Publisher			
Certification and Training	AMZ Publishers	56	$1,311.38	$1,243.47
	Bookstore Brand Titles	53	$1,208.56	$1,167.01
	Doe&Lee Ltd.	56	$1,231.20	$1,198.34
	Eversons Books	34	$815.22	$698.64
	IT Training Texts	59	$1,316.12	$1,229.62
	Mainst Media	102	$2,394.43	$2,070.97
	Nifty New Books	58	$1,365.23	$1,197.32
	Northern Associates Titles	35	$821.41	$747.99
	Popular Names Publishers	54	$1,177.01	$1,025.90
	Professional House Titles	42	$837.45	$834.16
	Technology Smith	55	$1,241.71	$1,193.17
	Wide-World Titles	53	$1,204.95	$1,100.27
	Total Sales	657	$14,924.67	$13,706.83

(*continued on the next page*)

Sales Report January 1, 2007 - November 24, 2007				
			Wholesale Cost	Profit
		Titles Sold	Sum	Sum
Web Design	Publisher			
	AMZ Publishers	58	$1,408.43	$1,333.22
	Bookstore Brand Titles	50	$1,156.13	$1,022.96
	Doe&Lee Ltd.	73	$1,660.16	$1,566.98
	Eversons Books	90	$2,046.12	$1,863.22
	IT Training Texts	49	$1,083.14	$1,008.83
	Mainst Media	58	$1,240.74	$1,172.22
	Nifty New Books	67	$1,522.82	$1,430.33
	Northern Associates Titles	56	$1,277.66	$1,047.80
	Popular Names Publishers	50	$1,161.75	$1,109.11
	Professional House Titles	84	$1,925.74	$1,730.87
	Technology Smith	64	$1,429.16	$1,323.17
	Wide-World Titles	77	$1,824.07	$1,788.82
	Total Sales	776	$17,735.92	$16,397.52
Total Sales		5608	$126,228.60	$116,487.40

Enhancing the Macro Program REPORT

Numerous enhancements can be added to a macro program after completing Step 4. A balance needs to be made, however, between generalizing a macro program and hard-coding features of a macro program. Each programming situation is different. For example, a macro program that you intend to use repeatedly for years might be worth your investment of time to enhance the macro program. A macro program that you might use only once or twice, and was developed mainly as a timesaver in writing SAS code, might not be worth enhancing.

Enhancements to consider adding to the macro program REPORT include the following:

- Make the data set name a parameter so that the program can be applied to other data sets.
- Check that the data set exists and that it contains observations.
- Do more error checking on the parameter values passed to the program. For example, you might want to check that the start date is after the stop date. You might also want to verify that the output destination parameter value is valid and that the style exists.
- Refine the layout of the PROC TABULATE report when there are more than two classification variables.
- Improve the readability of the titles and add information to the titles: include number of observations, print the date the report was run in a different format, print the name of the data set in the title.
- Improve the PROC GCHART code. Specify options that are specific to the output device. Add parameters to improve the graphics output.
- Delete the temporary data set created by the macro program.

Part 3

Appendixes

Appendix A **Abridged Macro Language Reference 371**

Appendix B **Reserved Words in the Macro Facility 391**

Appendix C **Sample Data Set 393**

Appendix D **Reference to Programs in This Book 399**

Appendix A

Abridged Macro Language Reference

Selected SAS Options Used with the Macro Facility 372
Automatic Macro Variables 373
Macro Functions 377
Macro Language Statements 381
PROC SQL Interface to the Macro Facility 386
SAS Functions and Routines That Interface with the Macro Facility 387

Appendix A summarizes macro language elements. For complete information and usage of these elements, refer to *SAS Macro Language: Reference*.

Selected SAS Options Used with the Macro Facility

Table A.1 lists selected SAS options that affect actions that SAS and the macro processor take. Most of the options enable or disable features. For this set of options, the enabling version of the option is in bold; this action is described in the second column of Table A.1. The remaining options require that you specify information such as librefs and filerefs. These specifications are italicized.

Table A.1 Selected SAS options used with the macro facility

Option	Description
CMDMAC | NOCMDMAC	Controls command-style macro program invocation.
IMPLMAC | NOIMPLMAC	Controls statement-style macro program invocation.
MACRO | NOMACRO	Controls whether the SAS macro facility is available in the SAS session.
MAUTOSOURCE | NOMAUTOSOURCE	Controls whether the macro processor searches autocall libraries when resolving a macro program reference.
MERROR | NOMERROR	Controls whether the macro processor issues warning messages when it cannot resolve a macro program reference.
MFILE	Determines whether MPRINT output is sent to an external file.
MINDELIMITER=<"*option*">	Specifies the character to be used as the delimiter for the macro IN operator.
MLOGIC | NOMLOGIC	Controls whether the macro processor traces execution of macro programs in the SAS log.
MPRINT | NOMPRINT	Controls whether the macro processor lists, in the SAS log, the SAS language statements generated by macro program execution.

(*continued*)

Table A.1 (*continued*)

Option	Description
MSTORED | NOMSTORED	Controls whether the macro processor searches stored compiled macro programs when resolving a macro program reference. (MSTORED must be used in conjunction with SASMSTORE.)
SASAUTOS=*library* | *(library-1, library2, ..., library-n)*	Specifies one or more autocall libraries using either filerefs defined by the FILENAME statement or by enclosing each of the paths of the libraries in quotation marks.
SASMSTORE=libref	Specifies the libref of a SAS library that contains a catalog of stored compiled macro programs.
SERROR | NOSERROR	Controls whether the macro processor issues warning messages when it cannot resolve a macro variable reference.
SYMBOLGEN | NOSYMBOLGEN	Controls whether the macro processor displays in the SAS log the results of resolving macro variable references.

Automatic Macro Variables

Table A.2 lists the automatic macro variables. All automatic variables, except for SYSPBUFF, are global macro variables and are initialized when your SAS session starts. Host-specific automatic macro variables are described in SAS documentation for the host operating environment.

You cannot modify the values of automatic macro variables whose type is read-only, though you can modify the values of automatic macro variables whose type is read/write.

Table A.2 Selected SAS automatic macro variables

Automatic Macro Variable	Type	Description
SYSBUFFR	read/write	Contains text entered in response to a %INPUT macro language statement when there is no corresponding macro variable.
SYSCC	read/write	Contains current condition code that SAS returns to your operating environment.
SYSCHARWIDTH	read-only	Contains the character width value.
SYSCMD	read/write	Contains last unrecognized command from the command line of a macro window that is created with the %WINDOW macro language statement.
SYSDATE	read-only	Contains the date in DATE7. format that the SAS session or job started.
SYSDATE9	read-only	Contains the date in DATE9. format that the SAS session or job started.
SYSDAY	read-only	Contains the text value of the day of the week.
SYSDEVIC	read/write	Contains the name of the current graphics device.
SYSDMG	read/write	Contains a return code that reflects an action taken on a damaged data set.
SYSDSN	read/write	Contains the two-part name (libref and data set name) of the most recently created data set.
SYSENCODING	read-only	Contains the name of the current session encoding.
SYSENV	read-only	Contains the environment of the job: FORE for interactive processing and BACK for noninteractive or batch processing.

(*continued*)

Table A.2 (*continued*)

Automatic Macro Variable	Type	Description
SYSERR	read-only	Contains a return code set by some SAS procedures and the DATA step.
SYSFILRC	read/write	Contains the return code from most recent execution of the FILENAME statement.
SYSINDEX	read-only	Contains the number of macro programs that have started execution during the current SAS session or job.
SYSINFO	read-only	Contains return codes provided by some SAS procedures.
SYSJOBID	read-only	Contains the name assigned to the current batch job or userid.
SYSLAST	read/write	Contains the two-part name (libref and data set name) of the most recently created data set.
SYSLCKRC	read/write	Contains the return code from most recent LOCK statement; used with SAS/SHARE.
SYSLIBRC	read/write	Contains the return code from most recent execution of the LIBNAME statement.
SYSMACRONAME	read-only	Contains the name of the currently executing macro program.
SYSMENV	read-only	Contains the location where the currently executing macro program was invoked: S for part of a SAS program; D for invoked from command line of a SAS window.
SYSMSG	read/write	Contains text to display in the message area of a macro window created with the %WINDOW and %DISPLAY macro language statements.
SYSNCPU	read-only	Contains the current number of processors available to SAS for computations.

(*continued*)

Table A.2 (*continued*)

Automatic Macro Variable	Type	Description
SYSPBUFF	read/write	Contains text supplied as macro parameter values. Used in conjunction with macro programs defined with the PARMBUFF option.
SYSPROCESSID	read-only	Contains the process ID of the current SAS process.
SYSPROCESSNAME	read-only	Contains the process name of the current SAS process.
SYSPROCNAME	read-only	Contains the name of the PROC (or DATASTEP for DATA steps) currently being processed.
SYSRC	read/write	Contains the last return code generated by your operating environment.
SYSSCP	read-only	Contains an identifier for the operating environment being used.
SYSSCPL	read-only	Contains an identifier for the operating environment being used, usually longer than SYSSCP.
SYSSITE	read-only	Contains the SAS site number.
SYSSTARTID	read-only	Contains the identification number that was generated by the last STARTSAS statement.
SYSSTARTNAME	read-only	Contains the process name that was generated by the last STARTSAS statement.
SYSTIME	read-only	Contains the time in TIME5. format that the SAS session or job started.
SYSUSERID	read-only	Contains the userid or login of the current SAS process.
SYSVER	read-only	Contains the release number of SAS that is currently executing.
SYSVLONG	read-only	Contains the release number and maintenance level of SAS that is currently executing.

Table A.3 lists the automatic macro variables that PROC SQL creates when you use PROC SQL.

Table A.3 Automatic macro variables created by PROC SQL

SQL Automatic Macro Variable	Description
SQLOBS	Contains the number of rows produced with a SELECT statement.
SQLOOPS	Contains the number of iterations of the inner loop of PROC SQL.
SQLRC	Contains the return code from an SQL statement.
SQLXMSG	Contains the return code generated by a Pass-Through facility statement.
SQLXRC	Contains the return code generated by a Pass-Through facility statement.

Macro Functions

Macro functions process arguments and return text values. Macro functions can be used in open code and inside macro programs. This book describes five types of macro functions:

- *character* functions, which operate on strings of characters or on macro variables.
- *evaluation* functions, which evaluate arithmetic and logical expressions. They temporarily convert their arguments to numbers to perform calculations and then change the results back to text.
- *quoting* functions, which mask special characters and mnemonic operators from interpretation by the macro processor.
- *macro variable attribute functions,* which supply information about the existence and the domain (global vs. local) of macro variables.
- *other* functions, which communicate between the macro facility and the rest of SAS or the operating environment.

Table A.4 lists macro functions, their categories, and their actions. Items that you specify are italicized. Brackets (< >) enclose optional arguments. Vertical bars (|) separate mutually exclusive items.

Additionally, SAS distributes several autocall macro programs that you can use like macro functions. Table A.5 lists a selection of these autocall macro programs.

Table A.4 Macro functions

Macro Function	Type	Action
%BQUOTE(*character-string* | *text expression*) **%NRBQUOTE**(*character-string* | *text expression)*	Quoting	%BQUOTE masks from interpretation during execution all special characters and mnemonic operators, except for ampersands(&) and percent signs(%), in the resolved value of the argument to the function. %NRBQUOTE does the same as %BQUOTE and additionally masks ampersands and percent signs.
%EVAL(*expression*)	Evaluation	Evaluates arithmetic and logical *expressions* using integer arithmetic.
%INDEX(*source, string*)	Character	Returns the position in *source* of the first character of *string.*
%LENGTH(*string* | *text expression*)	Character	Returns the length of *string* or the length of the results of the resolution of *text expression.*
%SCAN(*argument, n <,delimiters>*) **%QSCAN**(*argument, n <,delimiters>*)	Character	%SCAN returns the nth word in *argument* where the words in *argument* are separated by *delimiters.* %QSCAN does the same as %SCAN and masks special characters and mnemonic operators in *argument*.
%STR(*character-string)* **%NRSTR**(*character-string)*	Character	%STR masks all special characters and mnemonic operators except for ampersands (&) and percent signs (%) in constant text *at macro compilation*. %NRSTR does the same as %STR and additionally masks ampersands and percent signs.

(*continued*)

Table A.4 (*continued*)

Macro Function	Type	Action
%SUBSTR(*argument,position <,length>)* **%QSUBSTR**(*argument,position<,length>)*	Character	%SUBSTR extracts a substring of *length* characters from *argument* starting at *position.* %QSUBSTR does the same as %SUBSTR and masks special characters or mnemonic operators in *argument*.
%SUPERQ(*macro-variable-name)*	Quoting	Masks all special characters including ampersands (&) and percent signs (%) and mnemonic operators *at macro execution* and prevents further resolution of the value. Returns the value of a macro variable and does not resolve any macro references contained in that macro variable's value.
%SYMEXIST(*macro-variable-name*)	Macro Variable Attribute	Returns a 0 or 1 depending on whether *macro-variable-name* exists.
%SYMGLOBL(*macro-variable-name*)	Macro Variable Attribute	Returns a 0 or 1 depending on whether *macro-variable-name* is found in the global symbol table.
%SYMLOCAL(*macro-variable-name*)	Macro variable Attribute	Returns a 0 or 1 depending on whether *macro-variable-name* is found in a local symbol table.
%SYSEVALF(*expression <,conversion-type>*)	Evaluation	Evaluates arithmetic and logical expressions using floating-point arithmetic. Optionally converts results to *conversion-type*, where *conversion-type* is BOOLEAN, CEIL, FLOOR, or INTEGER.
%SYSFUNC(*function(argument(s)) <,format>*) **%QSYSFUNC**(*function(argument(s)) <,format>*)	Character	%SYSFUNC executes SAS language function or user-written functions and returns the results to the macro facility. %QSYSFUNC does the same as %SYSFUNC and masks special characters or mnemonic operators in *argument.*

(*continued*)

Table A.4 (*continued*)

Macro Function	Type	Action
%SYSGET(*host-environment-variable*)	Other	Returns the value of *host-environment-variable* to the macro facility.
%SYSPROD(*SAS-product*)	Other	Returns a code to indicate whether *SAS-product* is licensed at the site where SAS is currently running.
%UNQUOTE(*character-string* | *text expression*)	Quoting	Unmasks all special characters and mnemonic operators in a value at *macro execution.*
%UPCASE(*string* | *text expression)* **%QUPCASE**(*string* | *text expression)*	Character	%UPCASE converts *character string* or *text expression* to uppercase. %QUPCASE does the same as %UPCASE and masks special characters or mnemonic operators in the argument.

Table A.5 Autocall macro programs that act like macro functions

Autocall Macro Program	Action
%CMPRES(*text* | *text expression*)	%CMPRES removes multiple, leading, and trailing blanks from the argument.
%QCMPRES(*text* | *text expression*)	%QCMPRES does the same as %CMPRES and masks special characters and mnemonic operators in the argument.
%DATATYP(*text* | *text expression*)	Returns the data type (CHAR or NUMERIC) of a value.
%LEFT(*text* | *text expression*)	%LEFT aligns an argument to the left by removing leading blanks.
%QLEFT(*text* | *text expression*)	%QLEFT does the same as %LEFT and masks special characters and mnemonic operators in the argument.

(*continued*)

Table A.5 (*continued*)

Autocall Macro Program	Action
%LOWCASE(*text* | *text expression*) **%QLOWCASE**(*text* | *text expression*)	%LOWCASE changes a value from uppercase characters to lowercase. %QLOWCASE does the same as %LOWCASE and masks special characters and mnemonic operators in the argument.
%VERIFY(*source* | *excerpt*)	Returns the position of the first character unique to an expression where • *source* is text or a text expression that you want to examine for characters that do not exist in *excerpt* • *excerpt* is text or a text expression that defines the set of characters that %VERIFY uses to examine *source*

Macro Language Statements

Tables A.6 and A.7 list the macro language statements. The statements in Table A.6 are those that can be used both in open code and in macro program definitions, while those in Table A.7 can be used only in macro program definitions.

Items that you specify are italicized. Brackets (< >) enclose optional arguments. Vertical bars (|) separate mutually exclusive items.

Table A.6 Macro language statements allowed in open code and in macro programs

Statement	Description
%* *comment;*	Adds comment text to your macro programming.
%COPY *macro-name* < / <*LIBRARY=libref* *OUTFILE=<fileref | 'external file'>* *SOURCE>* **>;**	Copies macro programs from a SAS macro library specified by the *LIBRARY*= option or by the SASMSTORE= system option to the SAS log or to an output file specified by the *OUTFILE*= option.
%DISPLAY *window <.group>* <*NOINPUT*> <*BLANK*> <*BELL*> <*DELETE*>*;*	Displays a macro window defined with %WINDOW that can display fields and accept user input.
%GLOBAL *macro-variable-1 macro-variable-2 ... macro-variable-n*;	Creates global macro variables, which are available until the SAS session ends or the macro variables are deleted with %SYMDEL.
%INPUT <*macro-variable-1 macro-variable-2 ... macro-variable-n*>;	Accepts input as entered by the user or by the program and updates macro variables with the values entered.
%LET *macro-variable=<value>;*	Creates a macro variable and assigns it a value.
%MACRO *name <(parameter-list)>* < / <*CMD*> <*DES='text'*> <*PARMBUFF*> <*SECURE*> <*SOURCE*> <*STMT*> <*STORE*> >;	Begins the definition of a macro program where: • *name* is the name of the macro program. • *parameter-list* can *contain* positional parameter names and *keyword parameter* names that can optionally be initialized with values. Both types can be specified on one %MACRO statement; if they are, place the positional parameters first in the list. • *CMD* specifies that the macro program can be invoked with either a name-style or command-style invocation. • *DES='text'* adds descriptive *text* to the macro program when stored in a macro catalog. • *PARMBUFF* (or *PBUFF*) assigns the entire list of parameter values in the call to the macro program to an automatic macro variable named SYSPBUFF.

%MACRO *name <(parameter-list)> < / <CMD> <DES='text'> <PARMBUFF> <SECURE> <SOURCE> <STMT> <STORE> >; (continued)*	• *SECURE* causes the contents of a macro program to be encrypted when stored in a stored compiled macro library. The SECURE option can be used only in conjunction with the STORE option. • *SOURCE* combines and stores the source of the compiled macro program with the compiled macro code as an entry in a SAS catalog. • *STMT* specifies that the macro program can be invoked with either a name-style invocation or a statement-style invocation. • *STORE* specifies that the compiled macro program should be stored in a SAS catalog that is identified with the SASMSTORE system option.
%PUT *<text> <_ALL_> <_AUTOMATIC_> <_GLOBAL_> <_LOCAL_> <_USER_> <ERROR:> <WARNING:> <NOTE:>*;	Writes text or macro variable values to the SAS log, where: • *text* is text or a macro variable reference • *_ALL_* lists the values of all automatic and user-defined macro variables • *_AUTOMATIC_* lists the values of all automatic macro variables • *_GLOBAL_* lists the values of user-defined global macro variables • *_LOCAL_* lists the values of user-defined macro variables within the currently executing macro program • *_USER_* lists the user-defined global and local macro variables • *ERROR:* simulates a SAS error message by displaying the text ERROR: and remaining specifications on the %PUT statement in red • *WARNING:* simulates a SAS warning message by displaying the text WARNING: and remaining specifications on the %PUT statement in green • *NOTE:* simulates a SAS note message by displaying the text NOTE: and remaining specifications on the %PUT statement in blue
%SYMDEL *macro-variable-1 < macro-variable-2 ... macro-variable-n> </ NOWARN>;*	Deletes the specified macro variables from the global macro symbol table. The NOWARN option suppresses the warning message when an attempt is made to delete a nonexistent macro variable.

%SYSCALL *call-routine <(call-routine-arguments)>*;	Invokes a SAS or user-defined CALL routine.
%SYSEXEC <*command*>;	Issues *command* to the operating environment. If you omit *command*, you are placed in operating environment mode. %SYSEXEC is operating system-dependent.
%SYSLPUT *remote-macro-variable=<value</ REMOTE= remote-session-id>>;*	Creates a new macro variable or modifies the value of an existing macro variable on a remote host or server.
%SYSRPUT *local-macro-variable= remote-macro-variable*;	Assigns the value of a macro variable on the remote host to a local macro variable.
%WINDOW *window-name <window-options>group-definition-1 <...group-definition-n>*;	Defines a customized window to display text and accept user input.

Table A.7 Macro language statements allowed only in macro programs

Statement	Description
%ABORT <*ABEND* | *RETURN*>*;*	Stops the macro program that is currently executing along with the current DATA step, SAS job, or SAS session.
%DO;	Signals the beginning of a %DO group. The statements that follow form a block of code that is terminated with a %END statement.
%DO *macro-variable=start* **%TO** *stop* <**%BY** *increment*>;	The iterative %DO statement repetitively executes a section of macro program code by using an index variable and the keywords %TO and %BY. The section of macro code is terminated with a %END statement. • *start* and *stop* are integers or macro expressions that define the bounds of the iterative %DO. • *increment* is an integer or macro expression that defines the increment to take from *start* to reach *stop*.

%DO %UNTIL (*expression*);	Repetitively executes a section of macro code *until* the *expression* is true. The section of macro code is terminated with a %END statement. Because *expression* is evaluated at the bottom of the loop, a %DO %UNTIL loop always executes at least once.
%DO %WHILE *(expression);*	Repetitively executes a section of macro code *while* the *expression* is true. The section of macro code is terminated with a %END statement. Because *expression* is evaluated at the top of the loop, a %DO %WHILE might not execute.
%END;	Terminates a %DO group.
%GOTO *label*;	Branches macro processing to the specified macro *label* within the macro program.
%IF *expression* **%THEN** *action*; **<%ELSE** *action*;>	Conditionally processes a section of a macro program.
%label*: macro-text*	Identifies a section of macro code where *label* is a valid SAS name. This statement is typically used as the destination of a %GOTO statement.
%LOCAL *macro-variable-1 macro-variable-2 ... macro-variable-n*;	Defines macro variables that are available only to the macro program in which the %LOCAL statement was issued.
%MEND <*name*>*;*	Terminates a macro program definition and optionally repeats the name of the macro program.
%RETURN;	Causes normal termination of the currently executing macro program.

PROC SQL Interface to the Macro Facility

The INTO clause on the SELECT statement creates and updates macro variables. See Table A.3 for the automatic macro variables that PROC SQL creates.

The information on the INTO clause that you specify is italicized. Brackets (< >) enclose optional information.

SELECT col1,col2,...
INTO *:macro-variable-specification-1*
<,:macro-variable-specification-2,..., macro-variable-n>
FROM table-expression
WHERE where-expression
other clauses;

Create or update macro variables with values that are produced by PROC SQL, where:

- macro variables can be listed
  :macro-variable-1, :macro-variable2, ..., :macro-variable-n
- macro variables can be written in a numeric list
  :macro-variable-1-:macro-variable-n
- values that are placed in a macro variable can be side-by-side and separated with a character when the SEPARATED BY clause is added:
  SEPARATED BY '*character*'

The NOTRIM option can be added to prevent leading and trailing blanks from being removed when the macro variable is created.

:macro-variable-1 notrim

SAS Functions and Routines That Interface with the Macro Facility

Table A.8 lists three functions and three routines in the SAS language that interface with the macro facility. Items that you specify are italicized. Brackets (< >) enclose optional arguments.

Table A.8 SAS language functions and routines that interface with the macro facility

Function or Routine	Description
SYMGET(*argument*)	SAS language function that retrieves a macro-variable value for use in a DATA step, where *argument* can be one of the following: • literal text that is enclosed in quotation marks • the name of a data set character variable whose values are the names of macro variables • a character expression that resolves to a macro variable name.
SYMGETN(*argument*)	SAS language function that retrieves a macro-variable value and stores it as a number, where *argument* can be one of the following: • literal text that is enclosed in quotation marks • the name of a data set character variable whose values are the names of macro variables • a character expression that resolves to a macro variable name. Note that SYMGETN is pre-production in SAS®9.
CALL SYMPUT(*macro-variable-name*, *value*);	SAS language routine that creates or updates a macro variable from within a DATA step, where *macro-variable-name* can be specified one of the following ways: • literal text that is enclosed in quotation marks • the name of a DATA step character variable whose values are the names of macro variables

CALL SYMPUT(*macro-variable-name*, *value*); (*continued*)	• a character expression that resolves to a macro variable name. and where *value* can be specified one of the following ways: • literal text enclosed in quotation marks • the name of a DATA step variable (character or numeric) • a DATA step expression. CALL SYMPUT does not trim leading and trailing blanks.
CALL SYMPUTX(*macro-variable-name*, *value <,symbol-table>*);	SAS language routine that creates or updates a macro variable from within a DATA step and optionally designates the symbol table in which to store macro-variable, where *macro-variable-name* can be specified one of three ways: • literal text that is enclosed in quotation marks • the name of a DATA step character variable whose values are the names of macro variables • a character expression that resolves to a macro variable name and where *value* can be specified one of the following ways: • literal text enclosed in quotation marks • the name of a DATA step variable (character or numeric) • a DATA step expression and where symbol table can be one of three characters: • *G* for the global symbol table • *L* for the most local symbol table • *F* for the most local symbol in which the macro variable exists and if it does not exist, store it in the most local symbol table. CALL SYMPUTX trims leading and trailing blanks.

CALL EXECUTE(*argument*);	SAS language routine that executes the resolved value of *argument* from within a DATA step. Arguments that resolve to a macro facility reference execute immediately. Any SAS language statements resulting from the resolution execute at the end of the step. Resolved values are usually macro facility references, where *argument* can be one of the following: • a character string enclosed in single or double quotation marks. Single quotation marks direct resolution to occur during execution of the DATA step. Double quotation marks direct resolution to occur before the DATA step is compiled. • the name of a DATA step variable. • a character expression that is resolved by the DATA step to a text expression.
RESOLVE(*argument*)	SAS language function that resolves *argument* during DATA step execution. The *argument* is a text expression that is resolved by the macro facility, where *argument* can be one of the following: • a character string enclosed in single quotation marks • the name of a DATA step variable • a character expression that is resolved by the DATA step to a text expression

Appendix B

Reserved Words in the Macro Facility

Table B.1 lists words that are reserved for use by the macro facility.

Do not use a reserved word to name a macro program, a macro variable, or a macro label. When you use a reserved word in macro language, the macro processor issues a warning and does not compile or execute the macro program.

Do not start the name of a macro program, macro variable, or macro label with SYS, AF, or DMS since this could conflict with names of automatic macro variables.

Table B.1 Reserved words in the macro facility

ABEND	DO	LET	QSYSFUNC	SYSEXEC
ABORT	EDIT	LIST	QUPCASE	SYSFUNC
ACT	ELSE	LISTM	RESOLVE	SYSGET
ACTIVATE	END	LOCAL	RETURN	SYSRPUT
BQUOTE	EVAL	MACRO	RUN	THEN
BY	FILE	MEND	SAVE	TO
CLEAR	GLOBAL	METASYM	SCAN	TSO
CLOSE	GO	NRBQUOTE	STOP	UNQUOTE
CMS	GOTO	NRQUOTE	STR	UNSTR
COMANDR	IF	NRSTR	SUBSTR	UNTIL
COPY	INC	ON	SUPERQ	UPCASE
DEACT	INCLUDE	OPEN	SYMDEL	WHILE
DEL	INDEX	PAUSE	SYMEXIST	WINDOW
DELETE	INFILE	PUT	SYMGLOBL	
DISPLAY	INPUT	QSCAN	SYMLOCAL	
DMIDSPLY	KEYDEF*	QSUBSTR	SYSCALL	
DMISPLIT	LENGTH	QUOTE	SYSEVALF	

* Note that KEYDEF was made obsolete in SAS 8.2, but it is still recognized as a reserved word.

Appendix C

Sample Data Set

The following DATA steps create the data set that is used in this book. Make sure you define a libref for BOOKS before submitting the second DATA step.

```
data bookdb;

  attrib section  length=$30 label='Section'
         booktitle length=$50 label='Title of Book'
         author   length=$50 label='First Author'
         publisher length=$50 label='Publisher'
         cost     length=8 label='Wholesale Cost'
                  format=dollar10.2
         listprice length=8 label='List Price'
                  format=dollar10.2
         saleprice length=8 label='Sale Price'
                  format=dollar10.2;

  array sname{6} $ 30 ('Internet'
                       'Networks and Telecommunication'
                       'Operating Systems'
                       'Programming and Applications'
                       'Certification and Training'
                       'Web Design');
```

```
  array ln{125} $ 15 _temporary_ (
'Smith     '  'Johnson    ' 'Williams   ' 'Jones      '
'Brown     '  'Davis      ' 'Miller     ' 'Wilson     '
'Moore     '  'Taylor     ' 'Anderson   ' 'Thomas     '
'Jackson   '  'White      ' 'Harris     ' 'Martin     '
'Thompson  '  'Garcia     ' 'Martinez   ' 'Robinson   '
'Clark     '  'Rodriguez  ' 'Lewis      ' 'Lee        '
'Walker    '  'Hall       ' 'Allen      ' 'Young      '
'Hernandez '  'King       ' 'Wright     ' 'Lopez      '
'Hill      '  'Scott      ' 'Green      ' 'Adams      '
'Baker     '  'Gonzalez   ' 'Nelson     ' 'Carter     '
'Mitchell  '  'Perez      ' 'Roberts    ' 'Turner     '
'Phillips  '  'Campbell   ' 'Parker     ' 'Evans      '
'Edwards   '  'Collins    ' 'Stewart    ' 'Sanchez    '
'Morris    '  'Rogers     ' 'Reed       ' 'Cook       '
'Morgan    '  'Bell       ' 'Murphy     ' 'Bailey     '
'Rivera    '  'Cooper     ' 'Richardson ' 'Cox        '
'Howard    '  'Ward       ' 'Torres     ' 'Peterson   '
'Gray      '  'Ramirez    ' 'James      ' 'Watson     '
'Brooks    '  'Kelly      ' 'Sanders    ' 'Price      '
'Bennett   '  'Wood       ' 'Barnes     ' 'Ross       '
'Henderson '  'Coleman    ' 'Jenkins    ' 'Perry      '
'Powell    '  'Long       ' 'Patterson  ' 'Hughes     '
'Flores    '  'Washington ' 'Butler     ' 'Simmons    '
'Foster    '  'Gonzales   ' 'Bryant     ' 'Alexander  '
'Russell   '  'Griffin    ' 'Diaz       ' 'Hayes      '
'Myers     '  'Ford       ' 'Hamilton   ' 'Graham     '
'Sullivan  '  'Wallace    ' 'Woods      ' 'Cole       '
'West      '  'Jordan     ' 'Owens      ' 'Reynolds   '
'Fisher    '  'Ellis      ' 'Harrison   ' 'Gibson     '
'Mcdonald  '  'Cruz       ' 'Marshall   ' 'Ortiz      '
'Gomez     '  'Murray     ' 'Freeman    ' 'Wells      '
'Webb      ');

  array fn{70} $ 11 _temporary_ (
'James      ' 'John       ' 'Robert     ' 'Michael    '
'William    ' 'David      ' 'Richard    ' 'Charles    '
'Joseph     ' 'Thomas     ' 'Christopher' 'Daniel     '
'Paul       ' 'Mark       ' 'Donald     ' 'George     '
'Kenneth    ' 'Steven     ' 'Edward     ' 'Brian      '
'Ronald     ' 'Anthony    ' 'Kevin      ' 'Jason      '
'Matthew    ' 'Gary       ' 'Timothy    ' 'Jose       '
'Larry      ' 'Jeffrey    ' 'Jacob      ' 'Joshua     '
'Ethan      ' 'Andrew     ' 'Nicholas   '
'Mary       ' 'Patricia   ' 'Linda      ' 'Barbara    '
'Elizabeth  ' 'Jennifer   ' 'Maria      ' 'Susan      '
'Margaret   ' 'Dorothy    ' 'Lisa       ' 'Nancy      '
```

```
'Karen      ' 'Betty      ' 'Helen      ' 'Sandra     '
'Donna      ' 'Carol      ' 'Ruth       ' 'Sharon     '
'Michelle   ' 'Laura      ' 'Sarah      ' 'Kimberly   '
'Deborah    ' 'Jessica    ' 'Shirley    ' 'Cynthia    '
'Angela     ' 'Melissa    ' 'Emily      ' 'Hannah     '
'Emma       ' 'Ashley     ' 'Abigail    ');


  array pubname{12} $ 30 ('AMZ Publishers'   'Technology Smith'
                  'Mainst Media'      'Nifty New Books'
                  'Wide-World Titles'
                  'Popular Names Publishers' 'Eversons Books'
                  'Professional House Titles'
                  'IT Training Texts' 'Bookstore Brand Titles'
                  'Northern Associates Titles'
                  'Doe&Lee Ltd.');

  array prices{13} p1-p13
(27,30,32,34,36,40,44,45,50,54,56,60,86);
  array smax{6} (850,450,555,890,470,500);

  keep section booktitle author publisher listprice saleprice
cost;
  do i=1 to 6;
    section=sname{i};
    sectionmax=smax{i};
    do j=1 to sectionmax;
      booktitle=catx(' ',section,'Title',put(j,4.));

      lnptr=round(125*(uniform(54321)),1.);
      if lnptr=0 then lnptr=125;
      author=cats(ln{lnptr},',');
      fnptr=round(70*(uniform(12345)),1.);
      if fnptr=0 then fnptr=70;
      author=catx(' ',author,fn{fnptr});

      pubptr=round(12*(uniform(7890)),1.);
      if pubptr=0 then pubptr=12;
      publisher=pubname{pubptr};

      pval=round(2*normal(3),1) + 7;
      if pval > 13 then pval=13;
      else if pval < 1 then pval=1;
      listprice=prices{pval} + .95;
      saleprice=listprice;
      if mod(j,8)=0 then saleprice=listprice*.9;
```

```
      if mod(j,17)=0 and mod(j,8) ne 0 then
saleprice=listprice*.8;
      cost=.5*listprice;
      if mod(j,12)=0 then cost=.6*listprice;

      ncopies=round(rangam(33,.5),1);
      do n=1 to ncopies;
        output;
      end;

      output;
    end;
  end;
run;


data books.ytdsales(label='Sales for 2007');

  keep section--saleprice;
  attrib section  length=$30 label='Section'
         saleid   length=8 label='Sale ID'
                  format=8.
         saleinit length=$3 label='Sales Person Initials'
         datesold length=4 label='Date Book Sold'
                  format=mmddyy10. informat=mmddyy10.
         booktitle length=$50 label='Title of Book'
         author   length=$50 label='First Author'
         publisher length=$50 label='Publisher'
         cost     length=8 label='Wholesale Cost'
                  format=dollar10.2
         listprice length=8 label='List Price'
                  format=dollar10.2
         saleprice length=8 label='Sale Price'
                  format=dollar10.2;

   array mos{12} _temporary_
(555,809,678,477,300,198,200,500,655,719,649,356);

   array momax{12} momax1-momax12
                    (30,27,30,29,30,29,30,30,29,30,29,30);

   array inits{7} $ 3 _temporary_
              ('MJM' 'BLT' 'JMB' 'JAJ' 'LPL' 'SMA' 'CAD');
   retain saleid 10000000;
```

```
    do m=1 to 12;
      do j=1 to mos{m};
        day=round(momax{m}*uniform(3),1)+1;
        datesold=mdy(m,day,2007);
        obsno=int(uniform(3929)*5366)+1;
        set bookdb point=obsno;

        person=mod(day,7)+1;
        saleinit=inits{person};

        saleid+1;
        output;
      end;
      if m=12 then stop;
    end;
  run;
```

The PROC CONTENTS of the sample data set follows.

The CONTENTS Procedure			
Data Set Name	BOOKS.YTDSALES	Observations	6096
Member Type	DATA	Variables	10
Engine	V9	Indexes	0
Created	Friday, August 04, 2006 03:52:25 PM	Observation Length	224
Last Modified	Friday, August 04, 2006 03:52:25 PM	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label	Sales for 2007		
Data Representation	WINDOWS_32		
Encoding	wlatin1 Western (Windows)		

(*continued*)

```
               Engine/Host Dependent Information

Data Set Page Size          16384
Number of Data Set Pages    85
First Data Page             1
Max Obs per Page            72
Obs in First Data Page      63
Number of Data Set Repairs  0
File Name                   c:\books\ytdsales.sas7bdat
Release Created             9.0101M3
Host Created                XP_PRO

           Alphabetic List of Variables and Attributes

 # Variable  Type Len Format     Informat  Label

 6 author    Char  50                      First Author
 5 booktitle Char  50                      Title of Book
 8 cost      Num    8 DOLLAR10.2           Wholesale Cost
 4 datesold  Num    4 MMDDYY10.  MMDDYY10. Date Book Sold
 9 listprice Num    8 DOLLAR10.2           List Price
 7 publisher Char  50                      Publisher
 2 saleid    Num    8 8.                   Sale ID
 3 saleinit  Char   3                      Sales Person Initials
10 saleprice Num    8 DOLLAR10.2           Sale Price
 1 section   Char  30                      Section
```

Appendix D

Reference to Programs in This Book

Table D.1 lists the programs that are used in the examples in this book. Macro program names for programs defining macro programs are included.

Table D.1 Macro programs used in the examples in this book

Chapter	Program	Macro Program Name	Function
1	1.1		Define a macro variable in open code and reference its value in a WHERE statement and TITLE statement
	1.2	SALES	Process an iterative %DO loop over 3 years and submit a PROC MEANS step for each year
	1.3a and 1.3b		Define macro variables in open code and reference them in TITLE statements, a PROC TABULATE step, and a PROC GCHART step
	1.4		Reference automatic macro variables in TITLE statements
	1.5	DAILY	Process two PROC MEANS steps, one done only if automatic macro variable SYSDAY=Friday
	1.6	MAKESETS	Process two iterative %DO loops, one to name multiple data sets on a DATA statement and one to generate ELSE statements
	1.7		Define a macro variable with CALL SYMPUTX in a DATA step and reference the macro variable in a subsequent TITLE statement
	1.8	DSREPORT	Obtain data set attribute information with %SYSFUNC and SAS language functions and insert results in TITLE statements
	1.9	STANDARDOPTS	Specify an OPTIONS, TITLE, and FOOTNOTE statement
3	3.1a and 3.1b		Define macro variables in open code, reference them, and reference automatic macro variables in TITLE statements, a PROC FREQ step, and a PROC MEANS step
	3.2		Define macro variables in open code and reference them and automatic macro variables in TITLE statements, a PROC FREQ step, and a PROC MEANS step
	3.3		List automatic macro variables
	3.4		List global user-defined macro variables
	3.5		List automatic and user-defined macro variable values with %PUT statements
	3.6		Run Program 3.1b with the SYMBOLGEN option enabled
	3.7		Reference automatic macro variables in IF, TITLE, and FOOTNOTE statements
	3.8		Demonstrate macro language rules in defining macro variables with %LET statements
	3.9		Demonstrate resolution of macro variables when text is placed before a macro variable reference. Used in DATA step and PROC FREQ step
	3.10b		Construct a PROC FREQ TABLES statement when a macro variable reference precedes text. Demonstrate necessity of macro variable delimiters.
	3.11b		Construct a data set name where the libref is specified by a macro variable value. Demonstrate necessity of macro variable delimiters.
	3.12		Define a series of macro variables in open code and demonstrate referencing them indirectly in a PROC MEANS step
	3.13a		Demonstrate resolution of concatenated macro variable references
	3.13b		Demonstrate resolution of indirect referencing of macro variables with SYMBOLGEN enabled
	3.14		Define two series of macro variables in open code and demonstrate resolution of macro variable references when multiple ampersands precede a macro variable reference

4	4.1	SALESCHART	Specify one PROC GCHART step
	4.2		Specify a PROC CATALOG step for the WORK.SASMACR catalog
	4.3		Create a data set and call SALESCHART defined in Program 4.1
	4.4a and 4.4b		Compile macro program SALESCHART from Program 4.1 with different settings for MCOMPILENOTE= option
	4.5	LISTPARM	Defined with three positional parameters that specify options for a PROC MEANS step and range in dates for selection of observations to analyze
	4.6	KEYPARM	Defined with three keyword parameters that specify options for a PROC MEANS step and range in dates for selection of observations to analyze. Same as 4.5 with positional parameters replaced with keyword parameters.
	4.7	MIXDPARM	Defined with two positional parameters and two keyword parameters that specify options for a PROC MEANS step and range in dates for selection of observations to analyze. Similar to 4.5 and 4.6 with differences in types of parameters.
	4.8	PBUFFPARMS	Defined with PARMBUFF option. Contains two PROC GCHART steps, which are conditionally executed based on parameter specifications. If parameters specified, one PROC GCHART step submitted per value.
5	5.1	MAKEDS	Create a subset of data set based on value of macro variable defined in open code
	5.2	MAKEDS	Modifies 5.1 MAKEDS with subset specified with positional parameter. Produces a PROC TABULATE report.
	5.3	MAKEDS	Modifies 5.2 MAKEDS to include PROC TABULATE step as well as the DATA step.
	5.4	MAKEDS	Modifies 5.1 MAKEDS to include a %GLOBAL statement so that macro variable worked with in MAKEDS is available in open code.
	5.5	LOCLMVAR	Do one PROC MEANS step on a subset specified by local macro variable not same-named global macro variable
6	6.1		Extract a substring of text from the value of an automatic macro variable and include this value in a title and to specify a subset on a WHERE statement
	6.2		Extract the nth word from a macro variable using the %SCAN function and insert this word in a title
	6.3	LISTTEXT	Select a subset to process with PROC PRINT by specifying the text that should be present in a variable's value. Insert that text string in the title.
	6.4	MAKEDS	Modified version of Program 5.4 to demonstrate use of %SYMDEL, %SYMGLOBL, and %SYMLOCAL
	6.5		Format today's date in a TITLE statement using %SYSFUNC and the DATE() function
	6.6	GETOPT	Use macro statements to list SAS option info by applying %SYSFUNC and the GETOPTION SAS language function
	6.7	CHECKVARNAME	Use %SYSFUNC and SAS language functions to determine if positional parameter value is valid as a SAS name. Results listed by %PUT statements.
	6.8a		Compute the mean of four values stored in macro variables using %SYSFUNC and the MEAN SAS language function
	6.8b		Compute the mean of four values stored in macro variables using %SYSEVALF
	6.9		Obtain attributes of a data set with %SYSFUNC and ATTRN and insert this information in a title. Format the information with %SYSFUNC and PUTN.
	6.10a	CHECKSURVEY	Use SAS autocall macro programs %VERIFY and %UPCASE to verify if value of positional parameter RESPONSE is in list of acceptable characters. Results listed by %PUT statements.
	6.10b	CHECKSURVEY	Use %SYSFUNC and SAS language functions VERIFY and UPCASE to do the same as Example 6.10a.

7	7.1	COMP2VALS	Compare precedence of values of two positional parameters. Results listed with %PUT.
	7.2	REPORTS	Specify two PROC TABULATE steps, one for detail report, one for summary report. With two positional parameters, specify report type and month to analyze. Use %IF-%THEN/%ELSE statements to select report step.
	7.3	PUBLISHERREPORT	Specify which statements and options to include in a PROC REPORT step based on value of positional parameter
	7.4	VENDORTITLES	Look up vendor name based on parameter value specified for PUBLISHER. Insert this value in a TITLE statement.
	7.5	MULTREP	Process an iterative %DO loop that executes for range of years specified by the two positional parameters. Each iteration specifies a PROC MEANS step and a PROC GCHART step.
	7.6	SUMYEARS	Process an iterative %DO loop that concatenates data sets within range of years specified by the two positional parameters. Execute PROC GCHART on this composite data set.
	7.7	MOSALES	Defined with PARMBUFF option in which %DO %UNTIL processes the parameters to do a PROC MEANS for each parameter value. Also produces an overall PROC MEANS when no parameters specified.
	7.8	STAFFSALES	Processes a %DO %WHILE loop to parse the value of one of the two positional parameters. Specify a PROC MEANS for each item extracted from the parameter value. Second positional parameter specifies additional subsetting.
	7.9	DETAIL	Defined with one positional parameter that specifies a data set. Check if data set exists. If it does, specify a PROC PRINT step. If it doesn't, branch with %GOTO to section that specifies a PROC DATASETS step.
8	8.1		Use %STR quoting function to save the code for a PROC SQL step in a macro variable
	8.2		Quote the first argument to %SUBSTR with %STR because the argument contains special characters
	8.3		Quote a text string with %STR to preserve leading and trailing blanks in the text string
	8.4		Quote a text string with %NRSTR that is being assigned to a macro variable because the text contains macro triggers
	8.5a		Use %STR and preceding percent signs to mask unbalanced quotation marks in a value being assigned to a macro variable. Use %QSCAN to select one of the words in the value.
	8.5b		Use %BQUOTE to mask unbalanced quotation marks in a value being assigned to a macro variable value. Use %QSCAN to select one of the words in the value.
	8.6		Use %NRSTR and preceding percent signs to mask unbalanced quotation marks and macro triggers in a value being assigned to a macro variable. Use %QSCAN to select one of the words in the value.
	8.7		Use %BQUOTE and %STR to mask mnemonic operators in a Boolean %SYSEVALF evaluation
	8.8		Use %SUPERQ to prevent resolution of special characters in a macro variable value
	8.9	MOSECTDETAIL	Specify a PROC PRINT step on subset of data set defined by the values of two positional parameters. Mask one of the parameters with quoting functions because it can contain special characters.
	8.10	PUBLISHERSALES	Specify ODS characteristics for a PROC REPORT step through values of three positional parameters. Mask two of the parameters with quoting functions because they can contain special characters.

8 (*continued*)	8.11	MYPAGES	Format TITLE and FOOTNOTE statements based on values of six keyword parameters, some of which are defined with default values. Mask elements of the parameter values with quoting functions because they can contain special characters and mnemonic operators.
	8.12	MAR	Only contains text to demonstrate use of %UNQUOTE
	8.13		Quote the result of applying a SAS language function with %QSYSFUNC
	8.14		Quote the result of taking a substring of a macro variable value with %QSUBSTR
9	9.1		Use a data set variable name as the argument to the SYMGET function
	9.2		Retrieve macro variable values in a DATA step with SYMGETN to create numeric data set variables
	9.3		Use the resolution of a character expression as an argument to SYMGET in a DATA step
	9.4		Execute CALL SYMPUT once in a DATA step when processing reaches the last observation. Sum a variable in a DATA step and save the sum in a macro variable by executing CALL SYMPUT once when processing reaches the last observation.
	9.5		Execute CALL SYMPUTX multiple times in a DATA step
	9.6		Save PROC FREQ results in a data set. Process the data set in a DATA step and create several macro variables with CALL SYMPUT and CALL SYMPUTX
	9.7	STATSECTION	Save statistics computed with PROC MEANS in an output data set. Process this data set in a DATA step and store the statistics in global macro variables with CALL SYMPUTX that are later referenced in TITLE statements.
	9.8	LISTAUTOMATIC	Contains %PUT statements and lists automatic macro variables. Macro program is called by CALL EXECUTE from a DATA step.
	9.9	LISTLIBRARY	Specifies a PROC DATASETS step on BOOKS library. Macro program is called by CALL EXECUTE from a DATA step.
	9.10	REP60K	Specify a PROC REPORT step on a subset specified by the value of the single positional parameter. Before REP60K is called, a PROC MEANS step executes, saving results in an output data set. A DATA step evaluates each observation in the output data set and executes a CALL EXECUTE depending on the evaluation.
	9.11	HIGHREPORT	Specify a PROC REPORT step on a subset specified by the value of the single positional parameter. Before HIGHREPORT is called, a PROC MEANS step saves results in an output data set. A DATA step evaluates each observation in the output data set and executes a CALL EXECUTE depending on the evaluation. The same DATA step calls LOWREPORT depending on the evaluation.
		LOWREPORT	Specify a PROC REPORT step on a subset specified by the value of the single positional parameter. Before LOWREPORT is called, a PROC MEANS step saves results in an output data set. A DATA step evaluates each observation in the output data set and executes a CALL EXECUTE depending on the evaluation. The same DATA step calls HIGHREPORT depending on the evaluation.
	9.12		Obtain a macro variable value with RESOLVE by resolving a character expression
	9.13	GETSALENAME	Resolves to text string based on value of single positional parameter. GETSALENAME is called from a DATA step by RESOLVE SAS language function. The result of the call to GETSALENAME is saved in a data set character variable.
	9.14		Save PROC SQL summarizations in macro variables using the INTO clause
	9.15		Save with PROC SQL the first row of a table in macro variables using the INTO clause
	9.16		Create a macro variable for each row in a table using the INTO Clause in PROC SQL
	9.17		Store all unique values of a data set variable in one macro variable using the INTO clause in PROC SQL. Separate the values with a specific character.

9 (*continued*)	9.18		Store all values of a PROC SQL dictionary table column in one macro variable using the INTO clause in PROC SQL. Separate the values with a specific character.
	9.19	LISTSQLPUB	Process an iterative %DO loop to demonstrate use of PROC SQL and SQL macro variable SQLOBS. LISTSQLPUB lists values saved in macro variables by PROC SQL step.
	9.20a		Display the value of SAS option MACRO by executing a PROC SQL step on DICTIONARY.OPTIONS
	9.20b		Save a SAS option setting in a macro variable by using PROC SQL and DICTIONARY.OPTIONS
	9.21		Access macro variable characteristics with PROC SQL and dictionary tables
	9.22a and 9.22b		Create a macro variable in an SCL program and reference it in another SCL program
	9.23		Specify SCL SUBMIT blocks that reference macro variables
10	10.1	REPTITLE	Demonstrates use of STORE option on %MACRO statement. Constructs two TITLE statements using value of single positional parameter and automatic macro variable SYSVER.
	10.2	REPTITLE	Modifies 10.1 by adding SOURCE to %MACRO statement.
	10.3	REPTITLE	Modifies 10.1 by adding SECURE to %MACRO statement.
11	11.1a	LISTSAMPLE	Check if data set specified by single positional parameter exists and, if so, specify a PROC PRINT step to list the first 10 observations of the data set.
	11.1b	MULTCOND	Checks multiple characteristics of a data set and returns a return code value. This macro program is called by LISTSAMPLE in 11.1a from a %IF statement.
	11.2	TRIMNAME	Edits text passed to TRIMNAME by its single positional parameter and converts the text to uppercase, removes extra blanks between words, and removes all nonalphanumeric characters except for commas and a single blank between words. Macro program TRIMNAME returns the converted text and is called from a TITLE statement and a PROC REPORT step.
	11.3	RTF_START	Specify ODS characteristics, options, and TITLE and FOOTNOTE statements. Close the LISTING destination and open the RTF destination. Specify a style and page orientation with two keyword parameters. RTF_START precedes a PROC REPORT step and RTF_END follows the PROC REPORT step.
		RTF_END	Specify ODS characteristics, options, and TITLE and FOOTNOTE statements. Close the RTF destination and open the LISTING destination. Set the orientation to portrait. Set option DATE. Clear TITLE1 and FOOTNOTE1 statements. RTF_START precedes a PROC REPORT step and RTF_END follows the PROC REPORT step.
	11.4	FACTS	Document in a PDF report various attributes of a data set specified by the single positional parameter to FACTS and list the first five observations in the data set.

12	12.1	PRINT10	List the first ten observations of a data set specified by the single positional parameter
	12.2	WHSTMT	Construct part of a WHERE statement based on the value of two keyword parameters. The WHERE statement is used as an option on a SET statement.
	12.3b	MARKUP	Include a specific RETAIN statement in a DATA step based on the values of three of four of the positional parameters. Define assignment statements with values from these three parameters. Subset the data set with the fourth parameter.
	12.4b	TABLES	Construct a CLASS statement and TABLES statements for PROC TABULATE based on the value specified for the single positional parameter. The parameter contains the names of classification variables for the PROC TABULATE step. Parse the parameter with %DO %UNTIL constructing a TABLES statement for each value.
	12.5a	MAKEHTML	Called by macro program EXTFILES. Close the LISTING destination, open the HTML destination, run PROC PRINT, close the HTML destination, and open the LISTING destination.
		MAKEXLS	Called by EXTFILES. Submit a PROC EXPORT step to create a Microsoft Excel worksheet.
	12.5b	EXTFILES	Subset a data set by the value specified by one of the three positional parameters. The other two parameters specify whether a PROC PRINT report sent to an HTML destination should be produced for the subset and whether a Microsoft Excel spreadsheet should be produced for the subset. Call MAKEHTML and/or MAKEXLS based on these two parameter values.
	12.6b	PROJCOST	For a data set created by a DATA step preceding the call to PROJCOST, run PROJCOST to generate a PROC TABULATE report on the analysis variables specified in the single positional parameter, ANALYSISVARS.
	12.7	SELECTTITLES	Submit a PROC PRINT step for a subset defined by the values of the three keyword parameters. Includes macro language statements to check validity of parameter values.
	12.8	LASTMSG	Check if automatic variables SYSWARNINGTEXT and SYSERRORTEXT are nonnull. List contents of these automatic variables if nonnull.
	12.9	AUTHORREPORT	Based on the number of observations in a data set, as determined with PROC SQL, write messages to the SAS log, specify a PROC PRINT step, or specify a PROC TABULATE step.

13	N/A	REPORTA	Specify a PROC TABULATE step for a subset of a data set and on specific analysis variables. Specify the subset with three of the four keyword parameters. Specify the list of analysis variables on the fourth keyword parameter.
		REPORTB	Specify a PROC TABULATE step and a PROC GCHART step for a subset of a data set for specific classification variables on specific analysis variables. Specify the subset with three of the five keyword parameters. Specify the list of classification variables on the fourth keyword parameter. Specify the list of analysis variables on the fifth keyword parameter.
		REPORTC	Specify a PROC TABULATE step for a subset of a data set for specific classification variables on specific analysis variables. Specify the subset with three of the seven keyword parameters. Specify the list of classification variables on the fourth keyword parameter. Specify the list of analysis variables on the fifth keyword parameter. Specify ODS output characteristics on the sixth and seventh keyword parameters.
		REPORT	Combine functions of REPORTA, REPORTB, and REPORTC. Specify a PROC TABULATE step and a PROC GCHART step for a subset of a data set for specific classification variables on specific analysis variables. Specify the subset with three of the seven keyword parameters. Specify the list of classification variables on the fourth keyword parameter. Specify the list of analysis variables on the fifth keyword parameter. Specify ODS output characteristics on the sixth and seventh keyword parameters. Conditionally execute steps based on values of the keyword parameters.

Index

A

%ABORT statement 162, 384
ALL option, %PUT statement 47, 383
ampersand (&)
  as delimiter 201
  as macro trigger 24, 28, 35, 59
  common problems with 302
  %DO statements and 177
  INTO clause, SELECT statement 252
  quoting functions and 193–194, 198
  referencing macro variables 42
  referencing macro variables indirectly 65–71
AND operator 164
arithmetic expressions 163–165
  calculations and 164
  infix operator 166, 176
  macro expressions and 165
  macro functions and 138
ASCII character set 136, 167
ATTRC function 306
ATTRIB statement 220, 222, 247
ATTRN function 19, 153
  debugging with 306
  utility routines and 287
autocall facility 270
  catalogs and 270
  resolving macro program references 283–284
  saving macro programs with 270–278
  under various systems 275–278
autocall libraries 21–22
  common problems with 301
  creating 271–272
  identifying in catalogs 275
  identifying in sessions 275
  identifying with filerefs 274
  macro functions and 134
  macro programs and 77, 270–271, 273–275
  making available to programs 273
  searching 273
  specifying locations of 274
autocall macro programs
  listed 380–381
  macro functions and 154–157
  maintaining access to 273–275
  quoting functions and 191, 214–216
automatic macro variables 40–41, 52–55
  debugging with 306–307
  displaying system information 14
  %DO statement and 259
  listed 373–377
  macro symbol tables and 104
  SQL procedure and 258–260, 377
AUTOMATIC option, %PUT statement 47–49, 383

B

blanks
  See leading blanks
  See trailing blanks
BOOLEAN conversion type 139, 165
%BQUOTE function 142, 193–194, 378
  mnemonic operators and 201–202
  quotation marks and 199–200
  special characters and 210
branching in macro processing 184–187
BYLINE option, OPTIONS statement 21–22

C

calculations
  arithmetic expressions and 164
  common problems with 303
  macro functions and 138
  with macro variables 41, 57, 59, 134
  with SELECT statement 251

CALL statement (SCL) 263
case sensitivity in UNIX environment 271, 276
CATALOG method, FILENAME statement 275
CATALOG procedure 78, 279
catalogs
  See also SASMACR catalog
  autocall facility and 270
  executing macro programs 79
  identifying autocall libraries in 275
  macro programs in 278
  referencing with macro variables 63–65
  searching 279
  SOURCE entries in SAS catalogs 271–272, 278
  stored compiled macro facility and 270
  storing macro programs 77–78, 271–272, 280
CATS function 241
CEIL conversion type 139
character data, editing 289–290
character expressions
  RESOLVE function and 246–249
  SYMGET function and 225–226
  SYMPUT routine and 227
character functions 134–138, 377
  quoting functions and 214–216
CLOSE function 19, 153
CMD option, %MACRO statement 75, 382
CMDMAC system option 372
%CMPRES autocall program 155, 289, 380
colon (:) 185, 252
columns in tables 256–257
comma (,)
  as delimiter 166, 197
  keyword parameters in 89
  macro program parameters and 204
  positional parameters and 86
  quoting functions example 197
COMMA format 228
comparison operators 164–165
Compile command (SCL) 25
compilers/compiling
  defined 24
  displaying notes about macro programs 80–82
  errors during 298
  macro programs 77, 122–126, 278
  SAS processing and 25–26, 31, 34, 37–38
COMPRESS function 214, 289
concatenation
  CATS function and 241
  macro variables and 59, 63–65
conditional processing
  calling macro programs 238–241
  of DATA step 167
  with %DO-%UNTIL statements 180–182
  with macro statements 167–177
constants, removing hard-coded 336, 344–348
conversion 137–139, 165
%COPY statement 161, 382
  LIBRARY= option 281, 382
  OUTFILE= option 382
  stored compiled macro facility 278, 281–282

D

data set variables
  EXECUTE routine and 234
  macro variables and 42
  RESOLVE function and 246–247, 249–251
  SYMGET function and 221–223
  SYMGETN function and 223–224
  SYMPUT routine and 227
data sets
  displaying information about 153–154
  documenting characteristics of 293–296
  enhancing macro programs 367
  examining specific characteristics of 286–289
  executing steps on multiple data sets 6

permanent 63–65
sample 393–398
DATA step
conditional processing of 167
creating macro programs 348–352
EXECUTE routine and 234–245
iterative processing of 17–18, 177, 179–180
%LET statement in 56
macro facility interfaces 218–251
macro functions and 134
macro statements and 160
macro symbol tables and 104
macro variables in 40
passing information between 18–19
%PUT statement 46
RESOLVE function and 245–251
reviewing processing results 331–332
SAS macro facility support 13
SAS processing example 37
SCL and 263
solving errors example 310
SYMGET function and 219–226
SYMGETN function and 219–226
SYMPUT routine and 226–234
SYMPUTX routine and 226–234
tokenizing 28–29
WHERE clause 344
DATASETS procedure 185–187, 236–238
%DATATYP autocall program 155, 380
DATETIME format 153
DD statement (JCL) 275
debugging
ATTRN function 306
automatic macro variables and 306–307
%GOTO statement 305
macro programs 298, 303–307, 336, 338–344
minimizing errors 299
SYMBOLGEN option, OPTIONS statement 51
delimiters
ampersand as 201
comma as 166, 197
macro variables and 59–60
period as 59, 61, 63
%SCAN function 136
DES= option, %MACRO statement 75, 280, 382
dictionary tables 257, 260–262
DIR command 77, 280–281
%DISPLAY statement 163, 382
%DO statement 162, 384
ampersand and 177
automatic macro variables and 259
building macro programs 337, 352–356
common problems with 302
%EVAL function and 165
iterative processing of steps 17, 177–180
%DO-%TO-%BY statements 384
%DO-%UNTIL statements 162, 385
conditional iteration with 180–182
%EVAL function and 165
macro program example 96
tracing problems at execution 315–318
%DO-%WHILE statements 162, 385
conditional iteration with 182–184
%EVAL function and 165
documenting data set characteristics 293–296
DOLLAR format 324–325
domain of macro variables 118–122, 143–147, 306

E

EBCDIC character set 136, 167
Editor window 56, 77–78
ELSE statement 17–18
%ELSE statement
conditional processing and 167
logic errors during execution 319–322
tracing problems in expression evaluation 313–314
utility routines and 289
encryption 280, 282–283
%END statement 162, 302, 385

solving errors with 310–311
enhancing macro programs 366–367
ENTRY statement (SCL) 263
EQ operator 164
equal sign (=) 206–207
ERROR: option, %PUT statement 47, 383
errors
checking in macro programs 326–334
examples of solving in macro programs 307–325
in macro programs 298
logic errors 298, 319–322
%MEND statement and 309–311
minimizing in macro programs 299
MLOGIC system option and 311–315, 319–322
MPRINT system option and 322–325
%PUT statement and 311–318
reviewing system options 307–309
solving with %END statement 310–311
syntax 298
%EVAL function 378
common problems with 303
conditional processing 167
converting macro variables 152
%DO statement and 165
%DO-%UNTIL statements and 165
%DO-%WHILE statements and 165
examples of 140
integer arithmetic and 139
macro expressions and 163–165
syntax 138
tracing problems in expression evaluation 311–315
evaluation functions 138–140, 163–164, 377
Excel spreadsheet 319, 321
EXECUTE routine 234–245, 388–389
conditionally call macro programs 238–241
data set variables and 234
invoking macro programs that submit macro statements 235–236
invoking specific macro programs 241–245
PROC steps and 236–238
execution
errors during 298, 319–322
for multiple data sets 6
of macro programs 78–80, 127–132
of REPORT macro program 356–366
tracing problems at 314–318
EXIST function 286–287
expression evaluation, tracing problems in 311–315

F

FILENAME statement
CATALOG method 275
defining filerefs 273
filerefs
defining 273–274
identifying autocall libraries with 274
FLOOR conversion type 139
FOOTNOTE statement
library of utility routines 21–22
masking special characters in 208–213
quoting functions example 203
FREQ procedure
macro variables example 42–44, 50
referencing permanent data set names 64
RESOLVE function example 248–249
reviewing processing messages 329
SYMPUT routine example 230–231
functions
See also macro functions
See also quoting functions
assigning results to macro variables 149–150
character functions 134–138, 214–216, 377
checking valid names 150–151
evaluation functions 138–140, 163–164, 377
interfacing functions 19–21
macro programs behaving like 286–290

quoting results from 214–215
statistical 152–154

G

GCHART procedure
enhancing macro programs 367
iterative %DO loop 178
macro program example 96–99
passing information between steps 18
GE operator 164
GETOPTION function 149–150
GLOBAL option, %PUT statement 47, 49, 383
%GLOBAL statement 161, 382
defining domain of macro variables 118–122
macro symbol tables and 104, 113
global symbol table 104–108
macro variables in 41, 116–118, 144–147
referencing permanent data set names 64
SCL and 263
SQL procedure and 252
SYMPUTX routine and 227, 232
%GOTO statement 162, 385
branching with 184–187
common problems with 300
debugging with 305
greater than (>) 138, 163
GT operator 164

H

HTML files 319–321

I

IF statement 302
%IF statement
debugging with 305–306
%EVAL function and 165
IN operator and 166
logic errors during execution 320
masking parameters and 210–211
mnemonic operators and 208
solving errors example 312–314
%IF-%THEN-%ELSE statements 162, 385
building macro programs 96, 337, 352–356
conditional processing with 167
IN operator and 176–177
modifying statements within steps 172–175
reviewing processing results 331–334
IMPLMAC system option 372
IN operator
%IF-%THEN-%ELSE statements and 176–177
macro statements and 166
precedence order of 164
WHERE statement 204
%INDEX function 135, 378
infix operator 166, 176
input stack
defined 24
SAS processing and 25–26, 31–36
tokenization and 28
%INPUT statement 163, 382
integer arithmetic 139
INTEGER conversion type 139
interfacing functions 19–21
INTO clause, SELECT statement (SQL) 386
ampersand and 252
automatic macro variables 258
creating and updating macro variables 251–256
default saving action 253–255
saving macro variable summarizations 252–253
iterative processing
iterative %DO loop 178
of DATA step 17–18, 177, 179–180
with macro statements 177–184

J

JCL DD statement 275

K

keyword parameters
common problems with 302
in commas 89
masking special characters in 206–207
specifying in macro programs 88–95

L

LE operator 164
leading blanks
macro variable values and 59
NOTRIM option, SQL procedure 386
quoting functions and 198
SYMPUT routine and 226
SYMPUTX routine and 227
%LEFT autocall program 155, 380
%LENGTH function 135, 378
LENGTH statement 220, 247
less than (>) 138, 163
%LET statement 161, 382
common problems with 302
creating macro variables with 56–59
defining domain of macro variables 119–120
defining macro variables 42–44, 336
in DATA step 56
macro programs and 74, 287
macro quoting functions and 141
macro symbol tables and 113
masking characters 192
placing text after macro variable reference 61–63
placing text before macro variable reference 60–61
quoting functions examples 195–201, 214–216
referencing macro variables 42–44
replacing constants with macro variables 345
SAS processing and 31, 33
solving errors example 313
library of utility routines 21–22
LIBRARY= option, %COPY statement 281, 382
LISTING destination 351, 353
literal token 28, 37
LOCAL option, %PUT statement 47, 49, 383
%LOCAL statement 162, 385
defining domain of macro variables 118–122
macro variables and 287, 293
local symbol table 108–115
macro variables in 41, 116–118, 144–147
SQL procedure and 251
SYMPUTX routine and 227–228, 232
logic errors
execution-time problems 298
MLOGIC option and 319–322
logical expressions 163, 165
common problems with 303
conditional processing with 167–169
infix operator 166, 176
macro expressions and 165
macro functions and 138
%SYSEVALF function 201
%LOWCASE autocall program 154–155, 274, 381
LOWCASE function 155
LT operator 164

M

macro expressions
arithmetic expressions and 165
branching in 184–185
constructing 163–166
%EVAL function and 163–165
macro facility
See SAS macro facility
macro functions 133–134
See also quoting functions
arithmetic expressions and 138
autocall libraries and 134
autocall macro programs and 154–157

calculations and 138
character 134–138, 214–216, 377
common problems with 302
DATA step and 134
evaluation 138–140, 163–164, 377
macro expressions and 165
masking names 215
miscellaneous 147–154, 377
text expressions and 163
variable attribute 143–147
macro parameters
common problems with 302
macro symbol tables and 109
passing values through 85–95
macro processor/processing
branching in 184–187
defined 24
defining domain of macro variables 118–122
detecting errors 298
displaying messages 82–85
iterative 17–18, 177–184
%LET statement and 56
macro functions and 141
macro programs and 77, 122–132
macro statements and 160
macro symbol tables and 104, 108, 116–118
macro triggers and 59
mechanics of 23–38
referencing macro variables indirectly 65–71
RESOLVE function and 246
SAS processing and 29–34
searching autocall libraries 273
selecting steps for 169–171
single quotation marks and 248
macro programs
See also autocall macro programs
accepting varying number of parameter values 95–99
applying four steps 337–366
autocall libraries and 77, 270–271, 273–275
behaving like functions 286–290
branching in 184–187
building in four steps 336–357
building with %DO statement 337, 352–356
calling 108
calling conditionally 238–241
common problems in 299–303
compiling 77, 122–126, 278
conditional processing of steps 15–16
creating 74–78
creating stored compiled 280–281
creating with DATA step 348–352
debugging 298, 303–307, 336, 338–344
displaying compilation notes 80–82
displaying processing messages 82–85
encrypting 282–283
enhancing 366–367
error checking in 326–334
errors in 298
examples of 7
EXECUTE routine and 234–245
executing 78–80, 127–132
executing steps multiple times 6
in catalogs 278
invoking specific programs 241–245
logic errors in 319–322
macro functions and 134
macro statements and 160–163
macro symbol tables and 102
macro variables in 40–41
masking macro triggers in 198–199
minimizing errors in 299
mnemonic operator support 203–213
passing values through parameters 85–95
processing 77, 122–132
references in book 399–406
RESOLVE function and 245–251
resolving references 283–284
SAS macro facility support 13

macro programs (*continued*)
saving with autocall facility 270–278
saving with stored compiled macro facility 278–283
solving errors in 307–325
special characters support 203–213
storing 271, 280
storing in catalogs 77–78, 271–272, 280
submitting macro statements 235–236
submitting macro variables 236–238
text expressions and 163
typical functionality 5
writing 336, 338–344
macro source code
retrieving 281–282
storing 278
%MACRO statement 161, 382–383
CMD option 75, 382
common problems with 301
creating macro programs 74–75
DES= option 75, 280, 382
macro parameter names 85–86, 88–89
PARMBUFF option 75, 95–99, 180, 382
PBUFF option 75, 382
processing macro programs 124
SECURE option 280, 282–283, 383
SOURCE option 76, 278, 280–282, 383
SRC option 76
START= option 349–350
STMT option 75, 383
STOP= option 349–350
STORE option 76, 280, 282–283, 383
storing macro programs 280
macro statements 160–163, 381–385
autocall facility and 271
branching in 184
conditional processing with 167–177
DATA step and 160
debugging with 305
EXECUTE routine and 235–236
IN operator in 166
iterative processing with 177–184
macro programs and 74
processing 30–38
macro symbol tables 102–103
See also global symbol table
See also local symbol table
automatic macro variables and 104
DATA step and 104
defined 24
%GLOBAL statement 104, 113
%LET statement 31
macro programs and 128
macro variables and 24, 32, 102, 108–109, 219–226
memory and 24
SAS processing and 32–34
SYMGET function 219–226
SYMGETN function 219–226
SYMPUTX routine example 231–234
MACRO system option 305, 372
common problems with 301
verifying value of 307
macro triggers
ampersand as 24, 28, 35, 59
defined 24
macro functions and 141
macro processor and 59, 128
masking 141, 198–199, 201
SAS processing and 31, 33, 35
macro variable attribute functions 143–147, 306, 377
macro variables 40–42
See also automatic macro variables
applying statistical functions to 152–154
assigning results to 150–151
attribute functions 143–147, 306, 377
calculations with 41, 57, 59, 134
combining with text 59–65
common problems with 300–301
concatenation and 59, 63–65
conditional processing of steps 15–16
converting 152
creating for each row in tables 255–256
data set variables and 42
defining 7–12, 42–44

delimiters and 59–60
displaying values 46–52
domain of 118–122, 143–147, 306
extracting nth item from 136–137
in DATA step 40
in global symbol table 41, 116–118, 144–147
%LET statement 31
macro expressions and 165
macro functions and 134, 141, 143–147
macro option settings in 260–262
macro programs and 74, 336, 348–352
macro statements and 163
macro symbol table and 24, 32, 102, 108–109, 219–226
masking macro triggers in 198–199
maximum text length 42
null values for 59
quoting functions and 194
referencing 40, 42–44, 60–65
referencing catalogs with 63–65
referencing indirectly 65–71
replacing constants with 336, 344–348
RESOLVE function and 245–251
SAS processing 31–32, 34–37
saving summarizations 252–253
SCL and 263–266
selecting observations for processing 5
special characters in 202–203
SQL Pass-Through Facility 258
SQL procedure and 251–260, 293
storing all column values 257
storing unique column values 256–257
SYMBOLGEN option and 82
SYMPUT routine and 226–234
SYMPUTX routine and 226–234
text expressions and 163
typical uses for 5
user-defined 104
MACROS dictionary table 260–262
masking
equal sign 206–207
%IF statement and 210–211
macro triggers 141, 198–199, 201
mnemonic operators 191–204
quoting functions and 303
results of %SUBSTR function 215–216
special characters 191–203
special characters in parameters 204–205, 208–213
text strings 201
MAUTOSOURCE system option 372
autocall libraries and 273–276, 284
common problems with 301
MAX operator 163
MCOMPILENOTE= option, OPTIONS statement 80–81
MEAN statistical function 152–154
MEANS procedure
conditional processing of steps 15–16, 180–181, 184
EXECUTE routine example 238–241
executing on multiple data sets 6
invoking specific macro programs 241
iterative %DO loop 178
MISSING option 93
quoting functions examples 202
referencing macro variables example 43
RESOLVE function example 249
resolving macro variables 45
%SUBSTR macro function 136
SYMPUTX routine example 231–232
memory, and macro symbol table 24
%MEND statement 162, 385
common problems with 301
creating macro programs 74, 76
executing macro programs 131
solving errors example 309–312
MERROR system option 305, 372
common problems with 301
solving errors example 307–309
messages, reviewing processing 329–331
MFILE system option 372
Microsoft Excel spreadsheet 319, 321
MIN operator 163

MINDELIMITER system option 372
common problems with 301
overriding 176
setting 166, 261
MINOPERATOR system option 166
minus sign (-) 138, 163
MISSING option, MEANS procedure 93
MLOGIC system option 304, 372
building macro programs 337, 356, 358, 363
functionality 82, 84–85, 87
logic errors with 319–322
tracing problems in expression evaluation 311–315
MLOGICNEST system option 304
mnemonic operators
%BQUOTE function and 201–202
common problems with 303
interpreting as text 190
macro language and 190
macro program parameters with 203–213
macro quoting functions and 140–142
masking 191–203
percent signs in 163–164
preventing interruption of 201–202
modularity, and SAS macro facility 6
MPRINT system option 304, 372
building macro programs 336–337, 356, 358, 363
errors generated by 322–325
functionality 82–84, 98
tracing problems at execution 317
MPRINTNEST system option 304
MSTORED system option 373
common problems with 301
stored compiled macro programs and 279, 284
MVS environment
autocall facility in 275–276
storing macro programs 271

N

names token 28
naming conventions for macro variables 42
NE operator 163–164
nesting, common problems with 301
NOCMDMAC system option 372
NODATE option, OPTIONS statement 21–22
NOIMPLMAC system option 372
NOMACRO system option 305, 372
NOMAUTOSOURCE system option 372
NOMERROR system option 305, 372
NOMINOPERATOR system option 166
NOMLOGIC system option 304, 372
NOMLOGICNEST system option 304
NOMPRINT system option 304, 322, 372
NOMPRINTNEST system option 304
NOMSTORED system option 373
NOSERROR system option 305, 373
NOSYMBOLGEN system option 304, 373
NOT operator 163–164
NOTE: option, %PUT statement 47, 383
NOTNAME function 150–151
NOTRIM option, SQL procedure 252, 386
NOWARN option, %SYMDEL statement 383
%NRBQUOTE function 142, 193–194, 378
editing character data 289
%NRQUOTE function 142, 194
%NRSTR function 142, 192–193, 378
evaluating parameter values 328
masking macro function names 215
masking macro triggers 141, 198–199, 201
masking special characters in parameters 204–205, 211
null values
keyword parameters and 89
macro variables with 59
positional parameters and 86

NUMBER option, OPTIONS statement 21–22
numbers token 28
NVALID function 150–151

O

observations
  enhancing macro programs 367
  in WHERE statement 344
  selecting for processing 5
ODS PDF destination 293
ODS RTF destination 290–292, 330, 342
ODS statement 351, 353
  STYLE= option 351, 353
OPEN function 19, 153, 287
OpenVMS environment
  autocall facility in 277–278
  storing macro programs in 271
operators
  See also mnemonic operators
  See also specific operators
  comparison 164–165
  infix 166, 176
  order of precedence for 163–164
OPTIONS dictionary table 260–262
OPTIONS statement
  BYLINE option 21–22
  displaying macro variables 50–52
  library of utility routines 21–22
  MCOMPILENOTE= option 80–81
  NODATE option 21–22
  NUMBER option 21–22
  SYMBOLGEN option 51
OR operator 164, 166
order of precedence for operators 163–164
OUTFILE= option, %COPY statement 382

P

parameters
  See also keyword parameters
  See also positional parameters
  enhancing macro programs 367
  evaluation values 326–328
parentheses ()
  keyword parameters in 89
  macro expressions and 163
  positional parameters in 86
  quoting functions and 193–194
PARMBUFF option, %MACRO statement 75, 382
  accepting varying number of parameter values 95–99
  conditional iteration and 180
Pass-Through Facility (SQL) 258
PBUFF option, %MACRO statement 75, 382
PDF destination 293
percent sign (%)
  as macro trigger 24, 28, 33, 59
  branching in macro processing 185
  common problems with 302
  executing macro programs 79
  macro statements and 382
  masking 199–200
  mnemonic operators and 163–164
  quoting functions and 193–194, 201
period (.) 59, 61, 63
permanent data sets 63–65
PIE statement 353
plus sign (+) 138, 163
positional parameters
  commas and 86
  common problems with 302
  masking special characters in 206–207
  specifying in macro programs 85–88, 92–95
pound sign (#) 166, 176
precedence (order of) for operators 163–164
PRINT procedure
  default saving action for INTO clause 253–255
  macro quoting functions and 191–192
PROC steps
  conditional processing of 15–16, 167
  creating macro programs 348–352
  EXECUTE routine example 236–238

PROC steps (*continued*)
executing on multiple data sets 6
identifying autocall libraries in sessions 275
invoking specific macro programs 241–245
iterative processing of 177–180
macro variables in 40
passing information between 18–19
%PUT statement 46
SAS macro facility support 13
processing
See conditional processing
See SAS processing
punctuation, common problems with 300
%PUT_LOCAL statement 233
%PUT statement 161, 383
ALL option 47, 383
assigning statistical functions to macro variables 152
automatic macro variables and 259
AUTOMATIC option 47–49, 383
debugging with 305
default saving action for INTO clause 253–255
displaying macro variable values 46–50, 57
ERROR option 47, 383
errors and 311–318
EXECUTE routine example 235–238
GLOBAL option 47, 49, 383
LOCAL option 47, 49, 383
macro evaluation functions and 140
macro programs and 74
NOTE: option 47, 383
quoting functions examples 195–199, 215–216
reviewing processing messages 330
saving macro variable summarizations 252–253
storing unique column values 257
%SYSFUNC function example 149
tracing problems at execution 314–318
tracing problems in expression evaluation 311–315
USER_ option 47, 383
WARNING option 47, 383
PUTN function 19, 153, 287

Q

%QCMPRES autocall program 155, 380
%QLEFT autocall program 155, 380
%QLOWCASE autocall program 155, 381
%QSCAN function 378
%EVAL function and 165
masking and 200
special characters and 135
%QSUBSTR function 379
%EVAL macro function and 165
masking results of 215–216
special characters and 135
%QSYSFUNC function 147–148, 379
debugging with 306
quoting versions of 214–215
quotation marks (")
%BQUOTE function and 199–200
macro variable values and 59
quoting functions and 193–194, 199–201
referencing macro variables example 42–45
text strings in 234
tokens and 27
%QUOTE function 142, 194
quoting functions 140–142, 191–194, 377
ampersand and 193–194, 198
applying 195–203
autocall macro programs and 191, 214–216
character functions and 214–216
common problems with 303
examples 197, 203
masking and 303
%QUPCASE function 135, 380

R

referencing
  macro programs 78–80
  macro symbol tables 102, 104, 116–118
  macro variables 40, 42–44, 60–65
  macro variables indirectly 65–71
  permanent data set names 63–65
repetition, and SAS macro facility 6
REPORT macro program 356–366
REPORT procedure
  EXECUTE routine example 238–241
  masking equal signs in parameters 206–207
reserved words in macro facility 391–392
RESOLVE function 245–251, 389
  character expressions and 246–249
  common problems with 301
  data set variables and 246–247, 249–251
  DATA step and 245–251
  FREQ procedure example 248–249
resolving macro program references 283–284
%RETURN statement 162, 385
rows in tables 255–256
RTF destination 290–292, 330, 342
RTF output, standardizing 290–292
RUN statement 37–38

S

sample data sets 393–398
SAS/ACCESS 258
SAS Component Language
  See SCL
SAS/CONNECT 13, 161
.sas file extension 276
SAS/GRAPH 13
SAS logs
  displaying macro program compilation notes 80–82
  displaying macro program processing messages 82–85
SAS macro facility 4–6, 30
  # symbol 166, 176
  advantages of 6–12
  building macro programs and 339–344
  conditional processing of steps 15–16, 167
  DATA step interfaces 218–251
  DATA step support 13
  displaying system information 14
  examples of 13–22
  EXECUTE routine and 234–245
  interfacing functions 19–21
  iterative processing of steps 17–18
  library of utility routines 21–22
  modularity and 6
  options used with 372–373
  passing information between steps 18–19
  product compatibility 12–13
  reserved words 391–392
  RESOLVE function 245–251
  reviewing system options 307–309
  SAS processing 31–32
  SQL procedure and 251–262, 386
  SYMGET function 219–226
  SYMGETN function 219–226
  SYMPUT routine 226–234
  SYMPUTX routine 226–234
SAS macro programs
  See macro programs
SAS macro variables
  See macro variables
SAS processing
  compilers and 25–26, 31, 34, 37–38
  DATA step example 37
  macro processing and 29–30
  reviewing messages 329–331
  reviewing results 331–334
  selecting steps for 169–171
  SUBMIT blocks and 25, 262, 264–266
  vocabulary of 24–25
  without macro activity 25–26
SAS programs
  %LET statement in 56
  macro statements and 160
  macro variables in 40–41

SAS programs (*continued*)
processing with macro language 30–38
tokenizing 28–29
SAS statements
iterative %DO loops and 179–180
macro functions and 134
semicolon in 195–196
SAS/TOOLKIT 13, 134
SASAUTOS system option 22, 373
autocall facility support 275–278
autocall libraries and 273–274, 276
common problems with 301
defining filerefs 274–276
resolving macro program references 284
SASMACR catalog
location of 279
resolving macro program references 283–284
searching 279
storing macro programs 77–79, 271, 280
SASMSTORE system option 373
common problems with 301
%COPY statement and 382
%MACRO statement and 383
stored compiled macro programs 279, 284
saving macro programs 270–278
%SCAN function 96, 378
conditional processing of steps 181, 183
delimiters 136
%EVAL function and 165
extracting nth item from macro variables 136–137
special characters and 135
tracing problems at execution 317
SCL (SAS Component Language)
CALL statement 263
Compile command 25
DATA step and 263
ENTRY statement 263
global symbol table and 263
macro variables in 40
SAS macro facility and 13, 262–266
SAS processing and 25
SUBMIT blocks 25, 262, 264–266
searching
autocall libraries 273
catalogs 279
SASMACR catalogs 279
SECURE option, %MACRO statement 383
encryption and 282–283
stored compiled macro programs and 280
SELECT statement, SQL procedure
automatic macro variables and 258–260
calculations with 251
INTO clause 251–256, 258, 386
semicolon (;)
executing macro programs and 79
iterative %DO loops and 179
%LET statement and 56, 59
macro functions and 134
masking 192
SAS processing and 33
solving errors example 310
%STR function and 195–196
SERROR system option 305, 373
common problems with 301
solving errors with 307–308
sessions, identifying autocall libraries in 275
SET statement 179
single quotation marks (')
macro statements 166
quoting functions and 191, 193–194, 200
RESOLVE function and 248
resolving macro variables 44–45
source code 278, 281–282
SOURCE entries in SAS catalogs
macro programs as 271–272
macro source code in 278
SOURCE option, %MACRO statement 383
defining macro programs 76
stored compiled macro facility and 278, 280–282
special characters

%BQUOTE function and 210
common problems with 303
interpreting as text 190
macro program parameters with 203–213
macro quoting functions and 140–142
macro variable values and 59
masking 191–205, 208–213
%SCAN function and 135
special token 28
SQL Pass-Through Facility 258
SQL procedure
automatic macro variables and 258–260, 377
displaying macro option settings 260–262
global symbol table and 252
macro variables and 251–260, 293
NOTRIM option 252, 386
quoting functions example 195–196
SAS macro facility and 251–262, 386
SELECT statement 251–256, 258–260, 386
SQLOBS automatic macro variable 258–260, 377
SQLOOPS automatic macro variable 258, 377
SQLRC automatic macro variable 258, 377
SQLXMSG automatic macro variable 258, 377
SQLXRC automatic macro variable 258, 377
SRC option, %MACRO statement 76
standardizing RTF output 290–292
START= option, %MACRO statement 349–350
statistical functions 152–154
STMT option, %MACRO statement 75, 383
STOP= option, %MACRO statement 349–350
STORE option, %MACRO statement 383
defining macro programs 76
stored compiled macro programs 280, 282–283
stored compiled macro facility 270
catalogs and 270
resolving macro program references 283–284
saving macro programs with 278–283
storing
all column values in tables 257
column values in dictionary tables 257
macro programs 271, 280
macro programs in catalogs 77–78, 271–272, 280
macro source code 278
unique column values for macro variables 256–257
%STR function 142, 192–193, 378
comma and 197
leading/trailing blanks and 198
masking special characters in parameters 204–205, 210–211
masking text strings 201
quotation marks and 199–200
semicolon and 195–196
STYLE= option, ODS statement 351, 353
SUBMIT block (SCL) 25, 262, 264–266
%SUBSTR function 379
%EVAL function and 165
extracting text with 136
interpreting delimiters and 197
masking results of 215–216
special characters and 135
%SUPERQ function 142, 193–194, 379
editing character data example 289
special characters and 202–203, 210–211
SYMBOLGEN option, OPTIONS statement 51
SYMBOLGEN system option 304, 373
automatic macro variables and 259
building macro programs 336–337, 356
debugging with 305, 308
displaying macro variables 50–52, 68–69, 82
quoting functions and 195

resolving multiple ampersands 70
%SYMDEL statement 161, 383
debugging with 305
%GLOBAL statement and 382
macro symbol tables and 109, 144
NOWARN option 383
SYMPUTX routine example 234
%SYMEXIST function 379
debugging with 306
determining existence of macro variables 145
macro attribute variable functions 143
SYMGET function 219–226, 387
character expressions and 225–226
data set variables and 221–223
macro variables and 263–264
RESOLVE function and 246
SYMGETN function 219–226, 387
data set variables and 223–224
SCL and 263
%SYMGLOBL function 379
debugging with 306
determining existence of macro variables 145
macro attribute variable functions 143
%SYMLOCAL function 379
debugging with 306
determining existence of macro variables 145
macro attribute variable functions 143
SYMPUT routine 226–234, 387–388
character expressions and 227
common problems with 301
creating macro variables 230–231
data set variables and 227
DATA step interfaces 220
executing once in DATA step 228–230
FREQ procedure example 230–231
INTO clause, SELECT statement and 251
macro symbol tables and 102, 104
macro variables and 246, 263
SYMPUTN routine 263
SYMPUTX routine 226–234, 388
common problems with 301
creating macro variables 230–234
DATA step interfaces 220
executing multiple times in DATA step 230
global symbol table and 227, 232
INTO clause, SELECT statement and 251
macro symbol tables and 102, 104
macro variables and 246, 263
passing information between steps 18–19
quoting functions examples 202
syntax errors 298
SYSBUFFR automatic macro variable 374
%SYSCALL statement 161, 384
SYSCC automatic macro variable 374
SYSCHARWIDTH automatic macro variable 374
SYSCMD automatic macro variable 374
SYSDATE automatic macro variable
debugging with 307
defined 14, 53, 374
SYSDATE9 automatic macro variable
debugging with 307
defined 53, 374
macro symbol tables and 104
SYSDAY automatic macro variable
debugging with 307
defined 14, 53, 374
macro symbol tables and 104
referencing macro variables example 42–43
SYSDEVIC automatic macro variable 374
SYSDMG automatic macro variable 374
SYSDSN automatic macro variable 54, 307
SYSENCODING automatic macro variable 374
SYSENV automatic macro variable 374
SYSERR automatic macro variable
debugging with 307
defined 53, 375

SYSERRORTEXT automatic macro variable
debugging with 306–307
defined 53
reviewing processing messages 329–331
%SYSEVALF function 379
arithmetic expressions and 138, 164
common problems with 303
conversion types supported 139
converting macro variables 152
logical expressions and 138, 165, 201
tracing problems in expression evaluation 311–315
%SYSEXEC statement 161, 384
SYSFILRC automatic macro variable
debugging with 307
defined 53, 375
%SYSFUNC function 147–148, 379
debugging with 306
examples 149–154
LOWCASE function and 155
PUTN function and 287
quoting versions of 214–215
%SYSCALL statement and 161
UPCASE function and 157
VERIFY function and 157
%SYSGET function 148, 380
SYSINDEX automatic macro variable 375
SYSINFO automatic macro variable 375
SYSJOBID automatic macro variable 375
SYSLAST automatic macro variable
debugging with 307
defined 54, 375
SYSLCKRC automatic macro variable 375
SYSLIBRC automatic macro variable
debugging with 307
defined 53, 375
%SYSLPUT statement 161, 218, 384
SYSMACRONAME automatic macro variable
debugging with 307
defined 54, 375
modifying statements with 172
SYSMENV automatic macro variable 375
SYSMSG automatic macro variable 375
SYSNCPU automatic macro variable 375
SYSPBUFF automatic macro variable 382
conditional iteration and 180–181
defining macro programs 75, 95–97
global support and 376
SYSPROCESSID automatic macro variable 376
SYSPROCESSNAME automatic macro variable 376
SYSPROCNAME automatic macro variable 307, 376
%SYSPROD function 148, 380
SYSRC automatic macro variable
debugging with 307
defined 54, 376
%SYSRPUT statement 161, 218, 384
SYSSCP automatic macro variable 376
SYSSCPL automatic macro variable 376
SYSSITE automatic macro variable 376
SYSSTARTID automatic macro variable 376
SYSSTARTNAME automatic macro variable 376
system information, displaying 14
system options
See also specific system options
debugging macro programs with 304–305
OPTIONS dictionary table 260–262
reviewing 307–309
SYSTIME automatic macro variable
debugging with 307
defined 14, 53, 376
SYSUSERID automatic macro variable 376
SYSVER automatic macro variable
debugging with 307
defined 14, 53, 376
macro symbol tables and 104
masking special characters example 210–211

SYSVLONG automatic macro variable 376
SYSWARNINGTEXT automatic macro variable
  debugging with 306–307
  defined 54
  reviewing processing messages 329–331

T

TABLE statement, TABULATE procedure 315, 317, 325
tables
  creating macro variables for each row in 255–256
  default saving action for INTO clause 253–255
  displaying macro option settings 260–262
  INTO clause, SELECT statement 252
  storing all column values 257
  storing unique column values 256–257
TABLES statement 42, 323
TABULATE procedure
  enhancing macro programs 367
  macro symbol tables and 111–113
  MPRINT system option and 322–325
  reviewing processing results 332
  selecting steps for processing 169–170
  TABLE statement 315, 317, 325
  tracing problems at execution 315
  VAR statement 323
templates, building macro programs 342
testing
  macro programs 336, 338–344
  minimizing errors 299
text expressions
  defined 163
  EXECUTE routine and 234
  RESOLVE function and 246–247
text strings/values
  automatic macro variables and 54
  displaying 49–50
  in quotation marks 234
  macro variables and 41, 59–65
  masking 201
  substituting 4, 74
TITLE statement
  conditional processing 181–182
  library of utility routines 21–22
  macro variables example 42–44
  masking special characters in 208–213
  passing information between steps 18–19
  resolving macro variables 44–45
  solving errors example 308
  SYMPUTX routine example 232–233
  %SYSFUNC function example 149
  unmasking text 213
  %UPCASE function example 137–138
tokens (tokenization) 26–27
  DATA step 28–29
  defined 24
  literal 28, 37
  macro programs and 77, 124, 129–131
  maximum length of 26
  names 28
  numbers 28
  quoting functions and 193
  SAS programs and 28–29, 32, 34, 37
  SCL and 262, 264
  special 28
trailing blanks
  quoting functions and 198
  SQL procedure and 252, 386
  SYMPUT routine and 226
  SYMPUTX routine and 227
triggers
  See macro triggers
%TRIM autocall program 274
TSO environment
  autocall facility in 276
  storing macro programs 271

U

underscore (_) 24, 27
UNIQUE function 256–257

UNIX environment
  autocall facility in 276–277
  case sensitivity in 271, 276
  storing macro programs in 271
unmasking text 213
%UNQUOTE function 142, 380
  masking special characters in parameters 204–205
  unmasking text 213
UPCASE function 157
%UPCASE function 135, 380
  converting macro variables values 137–138
  special characters and 135, 156–157
user-defined macro variables 104
USER option, %PUT statement 47, 383
utility routines
  building and saving 21–22, 285–296
  library of 21–22
  macro programs behaving like functions 286–290
  programming routine tasks 290–296

V

VAR statement, TABULATE procedure 323
variables
  See data set variables
  See macro variables
%VERIFY autocall program 155–157, 381
VERIFY function 157
views 252

W

WARNING: option, %PUT statement 47, 383
WHERE clause, DATA step 344
WHERE statement
  conditional iteration and 181
  editing character data example 289
  evaluating parameter values 326
  IN operator and 204
  resolving multiple ampersands in 69–71
  solving errors example 309–310
  %SUBSTR function 135
%WINDOW statement 163, 382, 384
Windows environment
  autocall facility in 275
  defining filerefs 274
  identifying autocall libraries in catalogs 275
  setting SAS options 279
  storing macro programs in 271–272
word scanner
  defined 24
  macro programs and 77, 124, 128–131
  SAS processing and 25–26, 31–36
  SCL statements and 262
  tokens and 26, 28–29
WORK library
  See SASMACR catalog

Symbols

& (ampersand)
  See ampersand
: (colon) 185, 252
, (comma)
  See comma
= (equal sign) 206–207
> (greater than) 138, 163
< (less than) 138, 163
- (minus sign) 138, 163
() parentheses
  See parentheses
% (percent sign)
  See percent sign
. (period) 59, 61, 63
+ (plus sign) 138, 163
# (pound sign) 166, 176
" (quotation marks)
  See quotation marks
; (semicolon)
  See semicolon
' (single quotation marks)
  See single quotation marks
_ (underscore) 24, 27

Books Available from SAS Press

Advanced Log-Linear Models Using SAS®
by **Daniel Zelterman**

Analysis of Clinical Trials Using SAS®: A Practical Guide
by **Alex Dmitrienko, Geert Molenberghs, Walter Offen,** *and* **Christy Chuang-Stein**

Annotate: Simply the Basics
by **Art Carpenter**

Applied Multivariate Statistics with SAS® Software, Second Edition
by **Ravindra Khattree** *and* **Dayanand N. Naik**

Applied Statistics and the SAS® Programming Language, Fifth Edition
by **Ronald P. Cody** *and* **Jeffrey K. Smith**

An Array of Challenges — Test Your SAS® Skills
by **Robert Virgile**

Carpenter's Complete Guide to the SAS® Macro Language, Second Edition
by **Art Carpenter**

The Cartoon Guide to Statistics
by **Larry Gonick** *and* **Woollcott Smith**

Categorical Data Analysis Using the SAS® System, Second Edition
by **Maura E. Stokes, Charles S. Davis,** *and* **Gary G. Koch**

Cody's Data Cleaning Techniques Using SAS® Software
by **Ron Cody**

Common Statistical Methods for Clinical Research with SAS® Examples, Second Edition
by **Glenn A. Walker**

The Complete Guide to SAS® Indexes
by **Michael A. Raithel**

Data Management and Reporting Made Easy with SAS® Learning Edition 2.0
by **Sunil K. Gupta**

Data Preparation for Analytics Using SAS®
by **Gerhard Svolba**

Debugging SAS® Programs: A Handbook of Tools and Techniques
by **Michele M. Burlew**

Decision Trees for Business Intelligence and Data Mining: Using SAS® Enterprise Miner™
by **Barry de Ville**

Efficiency: Improving the Performance of Your SAS® Applications
by **Robert Virgile**

The Essential Guide to SAS® Dates and Times
by **Derek P. Morgan**

Fixed Effects Regression Methods for Longitudinal Data Using SAS®
by **Paul D. Allison**

Genetic Analysis of Complex Traits Using SAS®
by **Arnold M. Saxton**

A Handbook of Statistical Analyses Using SAS®, Second Edition
by **B.S. Everitt**
and **G. Der**

Health Care Data and SAS®
by **Marge Scerbo, Craig Dickstein,**
and **Alan Wilson**

The How-To Book for SAS/GRAPH® Software
by **Thomas Miron**

In the Know... SAS® Tips and Techniques From Around the Globe, Second Edition
by **Phil Mason**

Instant ODS: Style Templates for the Output Delivery System
by **Bernadette Johnson**

Integrating Results through Meta-Analytic Review Using SAS® Software
by **Morgan C. Wang**
and **Brad J. Bushman**

Introduction to Data Mining Using SAS® Enterprise Miner™
by **Patricia B. Cerrito**

Learning SAS® in the Computer Lab, Second Edition
by **Rebecca J. Elliott**

The Little SAS® Book: A Primer
by **Lora D. Delwiche**
and **Susan J. Slaughter**

The Little SAS® Book: A Primer, Second Edition
by **Lora D. Delwiche**
and **Susan J. Slaughter**
(updated to include SAS 7 features)

The Little SAS® Book: A Primer, Third Edition
by **Lora D. Delwiche**
and **Susan J. Slaughter**
(updated to include SAS 9.1 features)

The Little SAS® Book for Enterprise Guide® 3.0
by **Susan J. Slaughter**
and **Lora D. Delwiche**

The Little SAS® Book for Enterprise Guide® 4.1
by **Susan J. Slaughter**
and **Lora D. Delwiche**

Logistic Regression Using the SAS® System: Theory and Application
by **Paul D. Allison**

Longitudinal Data and SAS®: A Programmer's Guide
by **Ron Cody**

Maps Made Easy Using SAS®
by **Mike Zdeb**

Models for Discrete Date
by **Daniel Zelterman**

Multiple Comparisons and Multiple Tests Using SAS® Text and Workbook Set
(books in this set also sold separately)
by **Peter H. Westfall, Randall D. Tobias, Dror Rom, Russell D. Wolfinger,**
and **Yosef Hochberg**

Multiple-Plot Displays: Simplified with Macros
by **Perry Watts**

Multivariate Data Reduction and Discrimination with SAS® Software
by **Ravindra Khattree**
and **Dayanand N. Naik**

Output Delivery System: The Basics
by **Lauren E. Haworth**

Painless Windows: A Handbook for SAS® Users, Third Edition
by **Jodie Gilmore**
(updated to include SAS 8 and SAS 9.1 features)

Pharmaceutical Statistics Using SAS®: A Practical Guide
Edited by **Alex Dmitrienko, Christy Chuang-Stein,**
and **Ralph D'Agostino**

support.sas.com/pubs

The Power of PROC FORMAT
by **Jonas V. Bilenas**

PROC SQL: Beyond the Basics Using SAS®
by **Kirk Paul Lafler**

PROC TABULATE by Example
by **Lauren E. Haworth**

Professional SAS® Programmer's Pocket Reference, Fifth Edition
by **Rick Aster**

Professional SAS® Programming Shortcuts, Second Edition
by **Rick Aster**

Quick Results with SAS/GRAPH® Software
by **Arthur L. Carpenter**
and **Charles E. Shipp**

Quick Results with the Output Delivery System
by **Sunil Gupta**

Reading External Data Files Using SAS®: Examples Handbook
by **Michele M. Burlew**

Regression and ANOVA: An Integrated Approach Using SAS® Software
by **Keith E. Muller**
and **Bethel A. Fetterman**

SAS® for Forecasting Time Series, Second Edition
by **John C. Brocklebank**
and **David A. Dickey**

SAS® for Linear Models, Fourth Edition
by **Ramon C. Littell, Walter W. Stroup,**
and **Rudolf Freund**

SAS® for Mixed Models, Second Edition
by **Ramon C. Littell, George A. Milliken, Walter W. Stroup, Russell D. Wolfinger,** *and* **Oliver Schabenberger**

SAS® for Monte Carlo Studies: A Guide for Quantitative Researchers
by **Xitao Fan, Ákos Felsővályi, Stephen A. Sivo,**
and **Sean C. Keenan**

SAS® Functions by Example
by **Ron Cody**

SAS® Guide to Report Writing, Second Edition
by **Michele M. Burlew**

SAS® Macro Programming Made Easy, Second Edition
by **Michele M. Burlew**

SAS® Programming by Example
by **Ron Cody**
and **Ray Pass**

SAS® Programming for Researchers and Social Scientists, Second Edition
by **Paul E. Spector**

SAS® Programming in the Pharmaceutical Industry
by **Jack Shostak**

SAS® Survival Analysis Techniques for Medical Research, Second Edition
by **Alan B. Cantor**

SAS® System for Elementary Statistical Analysis, Second Edition
by **Sandra D. Schlotzhauer**
and **Ramon C. Littell**

SAS® System for Regression, Third Edition
by **Rudolf J. Freund**
and **Ramon C. Littell**

SAS® System for Statistical Graphics, First Edition
by **Michael Friendly**

The SAS® Workbook and *Solutions* Set
(books in this set also sold separately)
by **Ron Cody**

Selecting Statistical Techniques for Social Science Data: A Guide for SAS® Users
by **Frank M. Andrews, Laura Klem, Patrick M. O'Malley, Willard L. Rodgers, Kathleen B. Welch,**
and **Terrence N. Davidson**

Statistical Quality Control Using the SAS® System
by **Dennis W. King**

A Step-by-Step Approach to Using the SAS® System for Factor Analysis and Structural Equation Modeling
by **Larry Hatcher**

A Step-by-Step Approach to Using SAS® for Univariate and Multivariate Statistics, Second Edition
by **Norm O'Rourke, Larry Hatcher,** *and* **Edward J. Stepanski**

Step-by-Step Basic Statistics Using SAS®: Student Guide and *Exercises*
(books in this set also sold separately)
by **Larry Hatcher**

Survival Analysis Using SAS®: A Practical Guide
by **Paul D. Allison**

Tuning SAS® Applications in the OS/390 and z/OS Environments, Second Edition
by **Michael A. Raithel**

Univariate and Multivariate General Linear Models: Theory and Applications Using SAS® Software
by **Neil H. Timm** *and* **Tammy A. Mieczkowski**

Using SAS® in Financial Research
by **Ekkehart Boehmer, John Paul Broussard,** *and* **Juha-Pekka Kallunki**

Using the SAS® Windowing Environment: A Quick Tutorial
by **Larry Hatcher**

Visualizing Categorical Data
by **Michael Friendly**

Web Development with SAS® by Example, Second Edition
by **Frederick E. Pratter**

Your Guide to Survey Research Using the SAS® System
by **Archer Gravely**

JMP® Books

JMP® for Basic Univariate and Multivariate Statistics: A Step-by-Step Guide
by **Ann Lehman, Norm O'Rourke, Larry Hatcher,** *and* **Edward J. Stepanski**

JMP® Start Statistics, Third Edition
by **John Sall, Ann Lehman,** *and* **Lee Creighton**

Regression Using JMP®
by **Rudolf J. Freund, Ramon C. Littell,** *and* **Lee Creighton**

support.sas.com/pubs